

# Persistance Java 5 par la pratique

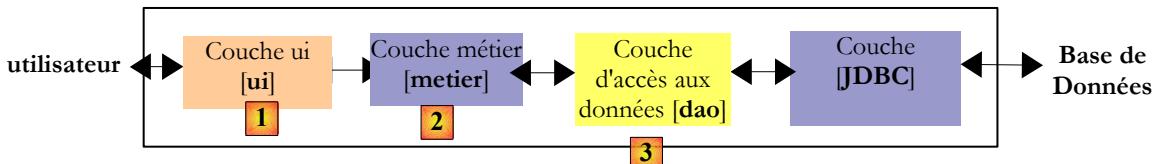
serge.tahe@istia.univ-angers.fr  
mai et juin 2007

# 1 Introduction

## 1.1 Objectifs

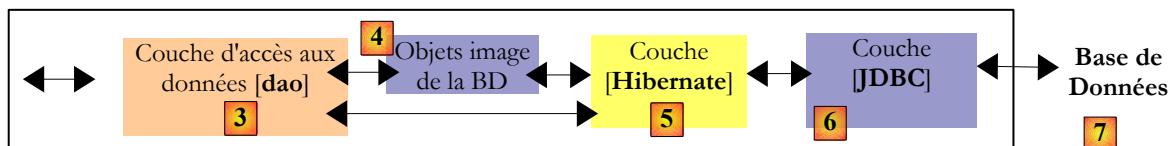
On se propose ici de découvrir les concepts principaux de la persistance de données avec l'API JPA (Java Persistence API). Après avoir lu ce document et en avoir testé les exemples, le lecteur devrait avoir acquis les bases nécessaires pour voler ensuite de ses propres ailes.

L'API JPA est récente. Elle n'a été disponible qu'à partir du JDK 1.5. La couche JPA a sa place dans une architecture multi-couches. Considérons une telle architecture assez répandue, celle à trois couches :



- la couche [1], appelée ici [ui] (User Interface) est la couche qui dialogue avec l'utilisateur, via une interface graphique Swing, une interface console ou une interface web. Elle a pour rôle de fournir des données provenant de l'utilisateur à la couche [2] ou bien de présenter à l'utilisateur des données fournies par la couche [2].
- la couche [2], appelée ici [metier] est la couche qui applique les règles dites métier, c.a.d. la logique spécifique de l'application, sans se préoccuper de savoir d'où viennent les données qu'on lui donne, ni où vont les résultats qu'elle produit.
- la couche [3], appelée ici [dao] (Data Access Object) est la couche qui fournit à la couche [2] des données pré-enregistrées (fichiers, bases de données,...) et qui enregistre certains des résultats fournis par la couche [2].
- la couche [JDBC] est la couche standard utilisée en Java pour accéder à des bases de données. C'est ce qu'on appelle habituellement le pilote Jdbc du SGBD.

De multiples efforts ont été faits pour faciliter l'écriture des ces différentes couches par les développeurs. Parmi ceux-ci, JPA vise à faciliter l'écriture de la couche [dao], celle qui gère les données dites persistantes, d'où le nom de l'API (Java Persistence API). Une solution qui a percé ces dernières années dans ce domaine, est celle d'**Hibernate** :



La couche [Hibernate] vient se placer entre la couche [dao] écrite par le développeur et la couche [Jdbc]. Hibernate est un ORM (Object Relational Mapping), un outil qui fait le pont entre le monde relationnel des bases de données et celui des objets manipulés par Java. Le développeur de la couche [dao] ne voit plus la couche [Jdbc] ni les tables de la base de données dont il veut exploiter le contenu. Il ne voit que l'image objet de la base de données, image objet fournie par la couche [Hibernate]. Le pont entre les tables de la base de données et les objets manipulés par la couche [dao] est fait principalement de deux façons :

- par des fichiers de configuration de type XML
- par des annotations Java dans le code, technique disponible seulement depuis le JDK 1.5

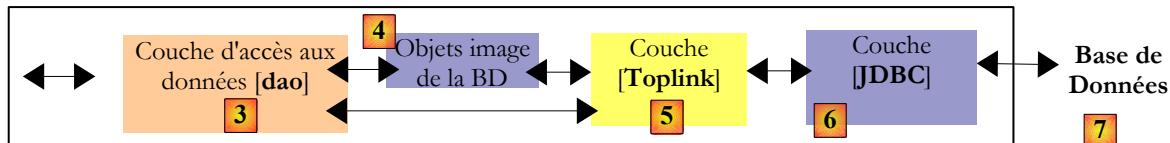
La couche [Hibernate] est une couche d'abstraction qui se veut la plus transparente possible. L'idéal visé est que le développeur de la couche [dao] puisse ignorer totalement qu'il travaille avec une base de données. C'est envisageable si ce n'est pas lui qui écrit la configuration qui fait le pont entre le monde relationnel et le monde objet. La configuration de ce pont est assez délicate et nécessite une certaine habileté.

La couche [4] des objets, image de la BD est appelée "contexte de persistance". Une couche [dao] s'appuyant sur Hibernate fait des actions de persistance (CRUD, create - read - update - delete) sur les objets du contexte de persistance, actions traduites par Hibernate en ordres SQL. Pour les actions d'interrogation de la base (le SQL Select), Hibernate fournit au développeur, un langage HQL (Hibernate Query Language) pour interroger le contexte de persistance [4] et non la BD elle-même.

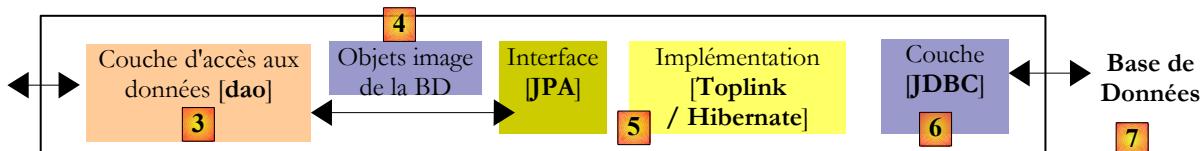
Hibernate est populaire mais complexe à maîtriser. La courbe d'apprentissage souvent présentée comme facile est en fait assez raide. Dès qu'on a une base de données avec des tables ayant des relations un-à-plusieurs ou plusieurs-à-plusieurs, la configuration

du pont relationnel / objets n'est pas à la portée du premier débutant venu. Des erreurs de configuration peuvent alors conduire à des applications peu performantes.

Dans le monde commercial, il existait un produit équivalent à Hibernate appelé Toplink :



Devant le succès des produits ORM, Sun le créateur de Java, a décidé de standardiser une couche ORM via une spécification appelée JPA apparue en même temps que Java 5. La spécification JPA a été implémentée par les deux produits **Toplink** et **Hibernate**. Toplink qui était un produit commercial est devenu depuis un produit libre. Avec JPA, l'architecture précédente devient la suivante :

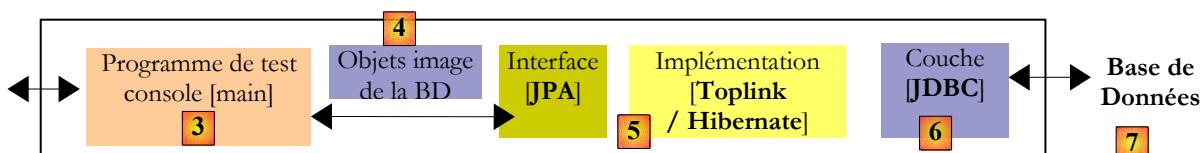


La couche [dao] dialogue maintenant avec la spécification JPA, un ensemble d'interfaces. Le développeur y a gagné en standardisation. Avant, s'il changeait sa couche ORM, il devait également changer sa couche [dao] qui avait été écrite pour dialoguer avec un ORM spécifique. Maintenant, il va écrire une couche [dao] qui va dialoguer avec une couche JPA. Quelque soit le produit qui implémente celle-ci, l'interface de la couche JPA présentée à la couche [dao] reste la même.

Ce document va présenter des exemples JPA dans divers domaines :

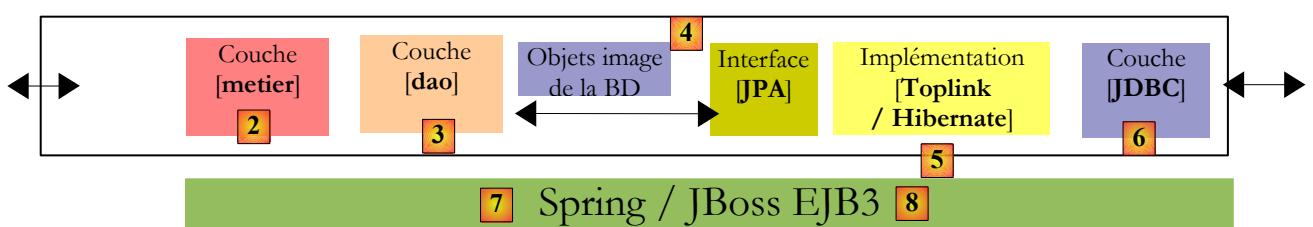
- ✗ tout d'abord, nous nous intéresserons au pont relationnel / objet que la couche ORM construit. Celui-ci sera créé à l'aide d'annotations Java 5 pour des bases de données dans lesquelles on trouvera des relations entre tables de type :
  - ✗ un à un
  - ✗ un à plusieurs
  - ✗ plusieurs à plusieurs

Pour illustrer ce domaine, nous créerons des architectures de test suivantes :



Nos programmes de tests seront des applications console qui interrogeront directement la couche JPA. Nous découvrirons à cette occasion les principales méthodes de la couche JPA. Nous serons dans un environnement dit "Java SE" (Standard Edition). JPA fonctionne à la fois dans un environnement Java SE et Java EE5 (Edition Entreprise).

- ✗ lorsque nous maîtriserons à la fois la configuration du pont relationnel / objet et l'utilisation des méthodes de la couche JPA, nous reviendrons à une architecture multi-couches plus classique :



La couche [JPA] sera accédée via une architecture à 2 couches [metier] et [dao]. Le framework Spring [7], puis le conteneur EJB3 de JBoss seront utilisés pour lier ces couches entre-elles.

Nous avons dit plus haut que JPA était disponible dans les environnements SE et EE5. L'environnement Java EE5 délivre de nombreux services dans le domaine de l'accès aux données persistantes notamment les pools de connexion, les gestionnaire de transactions, ... Il peut être intéressant pour un développeur de profiter de ces services. L'environnement Java EE5 n'est pas encore très répandu (mai 2007). On le trouve actuellement sur le serveurs d'application *Sun Application Server 9.x (Glassfish)*. Un serveur d'application est essentiellement un serveur d'applications web. Si on construit une application graphique autonome de type Swing, on ne peut disposer de l'environnement EE et des services qu'il apporte. C'est un problème. On commence à voir des environnements EE "stand-alone", c.a.d. pouvant être utilisés en-dehors d'un serveur d'applications. C'est le cas de JBos EJB3 que nous allons utiliser dans ce document.

Dans un environnement EE5, les couches sont implémentées par des objets appelés EJB (Enterprise Java Bean). Dans les précédentes versions d'EE, les EJB (EJB 2.x) sont réputés difficiles à mettre en oeuvre, à tester et parfois peu-performants. On distingue les EJB2.x "entity" et les EJB2.x "session". Pour faire court, un EJB2.x "entity" est l'image d'une ligne de table de base de données et EJB2.x "session" un objet utilisé pour implémenter les couches [metier], [dao] d'une architecture multi-couches. L'un des principaux reproches faits aux couches implémentées avec des EJB est qu'elles ne sont utilisables qu'au sein de conteneurs EJB, un service délivré par l'environnement EE. Cela rend problématiques les tests unitaires. Ainsi dans le schéma ci-dessus, les tests unitaires des couches [metier] et [dao] construits avec des EJB nécessiteraient la mise en place d'un serveur d'application, une opération assez lourde qui n'incite pas vraiment le développeur à faire fréquemment des tests.

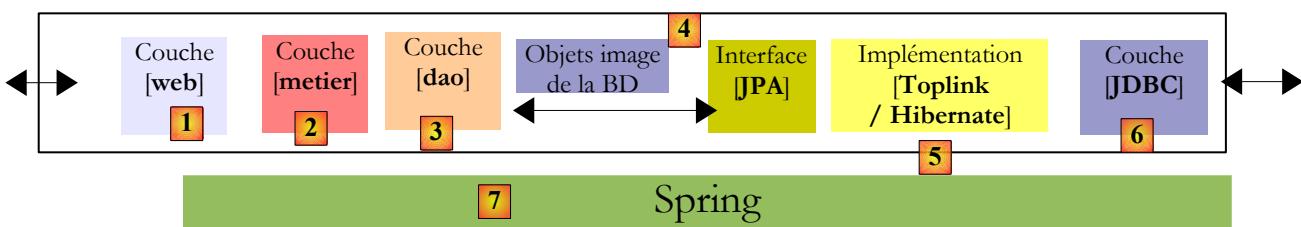
Le framework **Spring** est né en réaction à la complexité des EJB2. Spring fournit dans un environnement SE un nombre important des services habituellement fournis par les environnements EE. Ainsi dans la partie "Persistance de données" qui nous intéresse ici, Spring fournit les pools de connexion et les gestionnaires de transactions dont ont besoin les applications. L'émergence de Spring a favorisé la culture des tests unitaires, devenus d'un seul coup beaucoup plus faciles à mettre en oeuvre. Spring permet l'implémentation des couches d'une application par des objets Java classiques (POJO, Plain Old/Ordinary Java Object), permettant la réutilisation de ceux-ci dans un autre contexte. Enfin, il intègre de nombreux outils tiers de façon assez transparente, notamment des outils de persistance tels que Hibernate, Ibatis, ...

Java EE5 a été conçu pour corriger les lacunes de la précédente spécification EE. Les EJB 2.x sont devenus les EJB3. Ceux-ci sont des POJOs tagués par des annotations qui en font des objets particuliers lorsqu'ils sont au sein d'un conteneur EJB3. Dans celui-ci, l'EJB3 va pouvoir bénéficier des services du conteneur (pool de connexions, gestionnaire de transactions, ...). En-dehors du conteneur EJB3, l'EJB3 devient un objet Java normal. Ses annotations EJB sont ignorées.

Ci-dessus, nous avons représenté Spring et JBoss EJB3 comme infrastructure (framework) possible de notre architecture multi-couches. C'est cette infrastructure qui délivrera les services dont nous avons besoin : un pool de connexions et un gestionnaire de transactions.

- avec Spring, les couches seront implémentées avec des POJOs. Ceux-ci auront accès aux services de Spring (pool de connexions, gestionnaire de transaction) par injection de dépendances dans ces POJOs : lors de la construction de ceux-ci, Spring leur injecte des références sur les services dont il vont avoir besoin.
- JBoss EJB3 est un conteneur EJB pouvant fonctionner en-dehors d'un serveur d'application. Son principe de fonctionnement (pour le développeur) est analogue à celui décrit pour Spring. Nous trouverons peu de différences.

- ✗ nous terminerons le document avec un exemple d'application web à trois couches, basique mais néanmoins représentative :



## 1.2 Références

[ref1] : **Java Persistence with Hibernate**, de Christian Bauer et Gavin King, chez Manning.

[ref1] est le document qui a servi de fondement à ce qui suit. C'est un livre exhaustif de plus de 800 pages sur l'utilisation de l'ORM Hibernate dans deux contextes différents : avec ou sans JPA. L'utilisation d'Hibernate sans JPA est en effet toujours d'actualité pour les développeurs utilisant un JDK 1.4 ou inférieur, JPA n'étant apparu qu'avec le JDK 1.5.

Ayant lu plus des trois-quarts du livre, et survolé le reste, il m'est apparu que tout était utile dans ce document. L'utilisateur averti d'Hibernate devrait connaître la quasi-totalité des informations données dans les 800 pages. Christian Bauer et Gavin King ont été exhaustifs mais rarement pour décrire des situations qu'on ne rencontrera jamais. Tout est à lire. Le livre est écrit de façon pédagogique : il y a une réelle volonté de ne rien laisser dans l'obscurité. Le fait qu'il ait été écrit pour une utilisation d'Hibernate à la fois avec et sans JPA est une difficulté pour ceux qui ne sont intéressés que par l'une ou l'autre de ces technologies. Par exemple, les auteurs décrivent, autour de nombreux exemples, le pont relationnel / objet dans les deux contextes. Les concepts utilisés sont très proches puisque JPA s'est fortement inspiré d'Hibernate. Mais ils présentent quelques différences. Si bien qu'une chose qui est vraie pour Hibernate peut ne plus l'être pour JPA et cela finit par créer de la confusion chez le lecteur.

Les auteurs montrent des exemples d'applications trois couches dans le contexte d'un conteneur EJB3. Ils ne parlent pas de Spring. Nous verrons sur un exemple que Spring est pourtant plus simple à utiliser et à visée plus globale que le conteneur JBoss EJB3 utilisé dans [ref1]. Néanmoins "**Java Persistence with Hibernate**" est un excellent livre que je conseille pour tous les fondamentaux qu'on y apprend sur les ORM.

Utiliser un ORM est complexe pour un débutant.

- il y a des concepts à comprendre pour configurer le pont relationnel / objet.
- il y a la notion de contexte de persistance avec ses notions d'objets dans un état "persisté", "détaché", "neuf"
- il y a la mécanique autour de la persistance (transactions, pools de connexions), généralement des services délivrés par un conteneur
- il y a les réglages à faire pour les performances (cache de second niveau)
- ...

Nous introduirons ces concepts sur des exemples. Nous ferons peu de développements théoriques autour de ceux-ci. Notre objectif est simplement, à chaque fois, de permettre au lecteur de comprendre l'exemple et de se l'approprier jusqu'à être capable d'y amener lui-même des modifications ou de le rejouer dans un autre contexte.

## 1.3 Outils utilisés

Les exemples de ce document utilisent les outils suivants. Certains sont décrits en annexes (téléchargement, installation, configuration, utilisation). Dans ce cas, on donne le n° du paragraphe et la page.

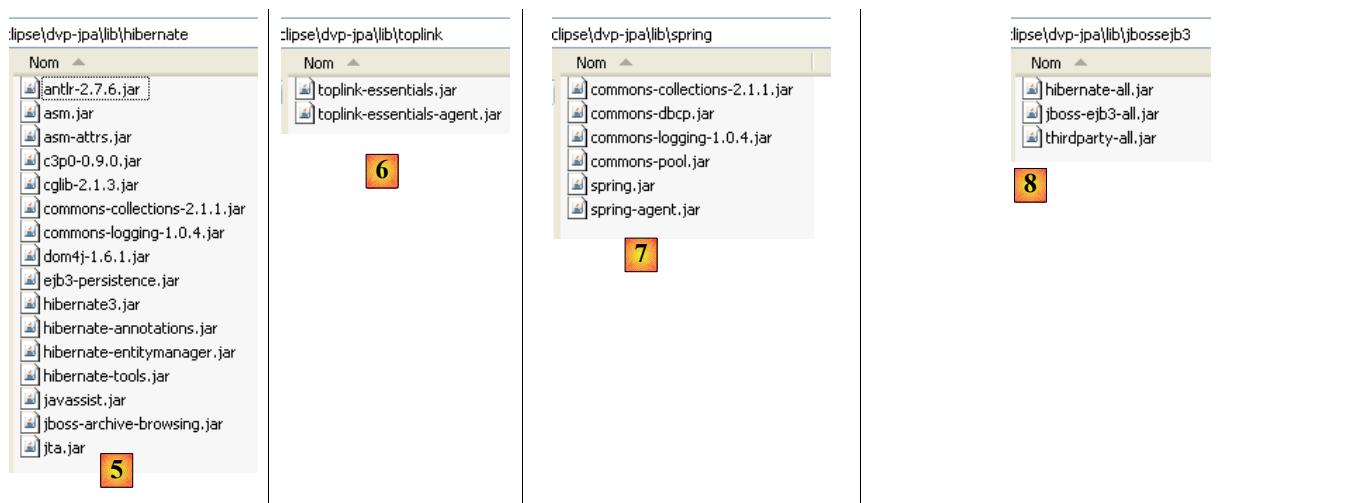
- un JDK 1.6 (paragraphe 5.1, page 209)
- l'IDE de développement Java Eclipse 3.2.2 (paragraphe 5.2, page 209)
- plugin Eclipse WTP (Web Tools Package) (paragraphe 5.2.3, page 215)
- plugin Eclipse SQL explorer (paragraphe 5.2.6, page 219)
- plugin Eclipse Hibernate Tools (paragraphe 5.2.5, page 218)
- plugin Eclipse TestNG (paragraphe 5.2.4, page 217)
- conteneur de servlets Tomcat 5.5.23 (paragraphe 5.3, page 221)
- SGBD Firebird 2.1 (paragraphe 5.4, page 240)
- SGBD MySQL5 (paragraphe 5.5, page 258)
- SGBD PosgreSQL (paragraphe 5.6, page 268)
- SGBD Oracle 10g Express (paragraphe 5.7, page 276)
- SGBD SQL Server 2005 Express (paragraphe 5.8, page 284)
- SGBD HSQLDB (paragraphe 5.9, page 291)
- SGBD Apache Derby (paragraphe 5.10, page 295)
- Spring 2.1 (paragraphe 5.11, page 299)
- conteneur EJB3 de JBoss (paragraphe 5.12, page 301)

## 1.4 Téléchargement des exemples

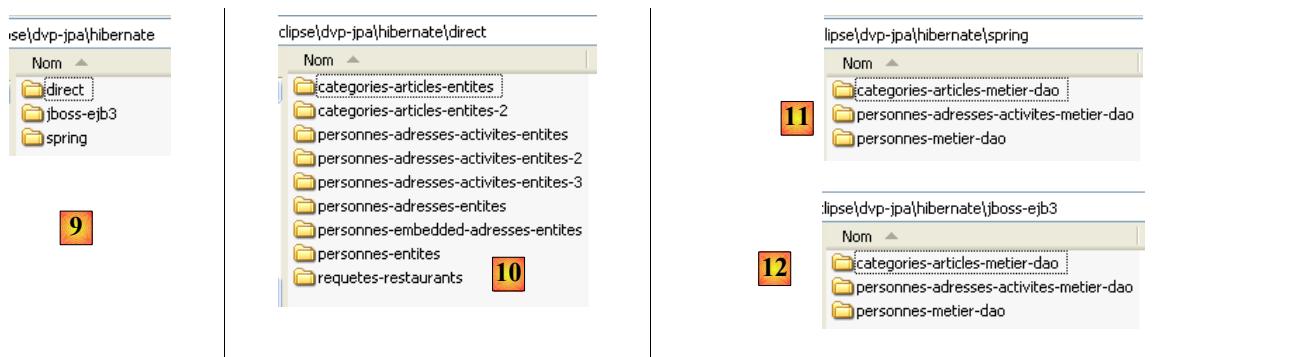
Sur le site de ce document, les exemples étudiés sont téléchargeables sous la forme d'un fichier zip, qui une fois décompressé, donne naissance au dossier suivant :



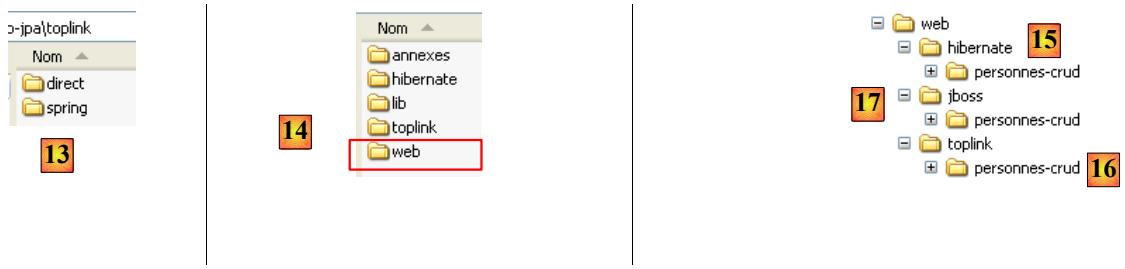
- en [1] : l'arborescence des exemples
- en [2] : le dossier <annexes> contient des éléments présentés dans la partie ANNEXES, page 209. En particulier, le dossier <jdbc> contient les pilotes Jdbc des SGBD utilisés pour les exemples du tutoriel.
- en [3] : le dossier <lib> regroupe en 5 dossiers les différentes archives .jar utilisées par le tutoriel
- en [4] : le dossier <lib/divers> regroupe les archives :
  - des pilotes Jdbc des SGBD
  - de l'outil de test unitaire [testNG]
  - de l'outil de logs [log4j]



- en [5] : les archives de l'implémentation JPA/Hibernate et d'outils tierces nécessaires à Hibernate
- en [6] : les archives de l'implémentation JPA/Toplink
- en [7] : les archives de Spring 2.x et d'outils tierces nécessaires à Spring
- en [8] : les archives du conteneur EJB3 de JBoss



- en [9] : le dossier <hibernate> regroupe les exemples traités avec la couche de persistance JPA/Hibernate
- en [10] : le dossier <hibernate/direct> regroupe les exemples où la couche JPA est exploitée directement avec un programme de type [Main].
- en [11] et [12] : des exemples où la couche JPA est exploitée via des couches [metier] et [dao] dans une architecture multi-couches, ce qui est le cas normal d'exploitation. Les services (pool de connexions, gestionnaire de transactions) utilisés par les couches [metier] et [dao] sont fournis soit par Spring [11] soit par JBoss EJB3 [12].

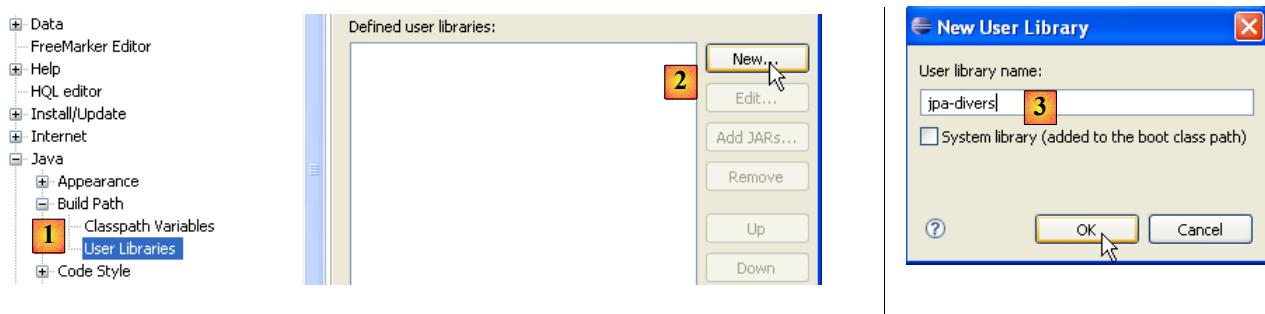


- en [13] : le dossier <toplink> reprend les exemples du dossier <hibernate> [9] mais avec cette fois une couche de persistance JPA/Toplink au lieu de JPA/Hibernate. Il n'y a pas dans [13] de dossier <jbossejb3> car il n'a pas été possible de faire fonctionner un exemple où la couche de persistance est assurée par Toplink et les services assurés par le conteneur EJB3 de JBoss.
- en [14] : un dossier <web> regroupe trois exemples d'applications web avec une couche de persistance JPA :
  - [15] : un exemple avec Spring / JPA / Hibernate
  - [16] : le même exemple avec Spring / JPA / Toplink
  - [17] : le même exemple avec JBoss EJB3 / JPA / Hibernate. Cet exemple ne fonctionne pas, probablement pour un problème de configuration non élucidé. Il a été néanmoins laissé afin que le lecteur puisse se pencher dessus et éventuellement trouver une solution à ce problème.

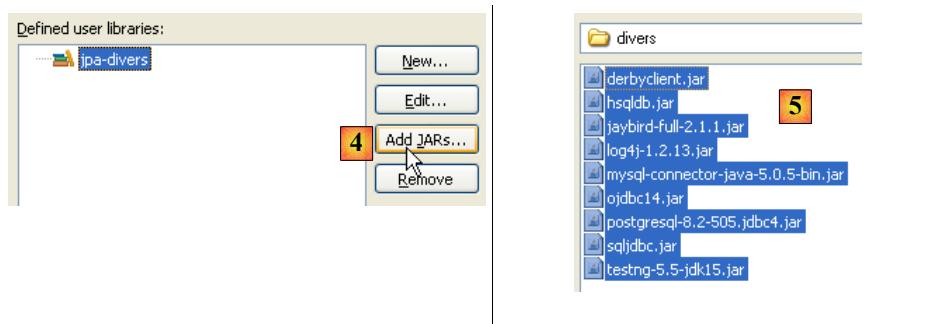
Le tutoriel fait souvent référence à cette arborescence, notamment lors des tests des exemples étudiés. Le lecteur est invité à télécharger ces exemples et à les installer. Par la suite, nous appellerons <exemples>, l'arborescence des exemples décrite ci-dessus.

## 1.5 Configuration des projets Eclipse des exemples

Les exemples utilisent des bibliothèques "utilisateur". Ce sont des archives .jar réunies sous un même nom. Lorsqu'on inclut une telle bibliothèque dans le *classpath* d'un projet Java, toutes les archives qu'elle contient sont alors incluses dans ce classpath. Voyons comment procéder sous Eclipse :



- en [1] : [Window / Preferences / Java / Build Path / User Libraries]
- en [2] : on crée une nouvelle bibliothèque
- en [3] : on lui donne un nom et on valide



- en [4] : on va sélectionner les jars qui feront partie de la bibliothèque [jpa-divers]
- en [5] : on sélectionne tous les jars du dossier <exemples>/lib/divers



- en [6] : la bibliothèque utilisateur [jpa-divers] a été définie
- en [7] : on refait la même démarche pour créer 4 autres bibliothèques :

Bibliothèque	Dossier des jars de la bibliothèque
jpa-hibernate	<exemples>/lib/hibernate
jpa-toplink	<exemples>/lib/toplink
jpa-spring	<exemples>/lib/spring
jpa-jbossejb3	<exemples>/lib/jbossejb3

## 2 Les entités JPA

### 2.1 Exemple 1 - Représentation objet d'une table unique

#### 2.1.1 La table [personne]

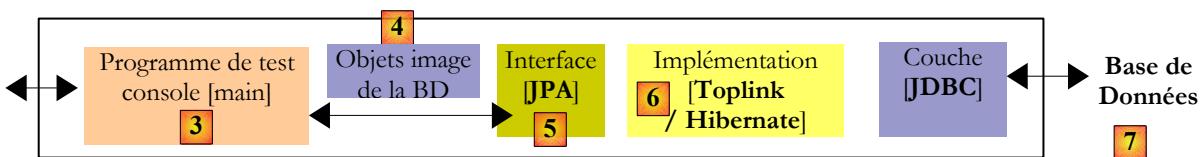
Considérons une base de données ayant une unique table [personne] dont le rôle est de mémoriser quelques informations sur des individus :

Table Name: personne		Database: ipa	
		Columns and Indices	
		Table Options	
Column Name	Datatype	NOT NULL	AUTO INC
ID	INTEGER	✓	✓
VERSION	INTEGER	✓	
NOM	VARCHAR(30)	✓	
PRENOM	VARCHAR(30)	✓	
DATENAISANCE	DATE	✓	
MARIE	BIT(1)	✓	
NBENFANTS	INTEGER	✓	

ID	clé primaire de la table
VERSION	version de la ligne dans la table. A chaque fois que la personne est modifiée, son n° de version est incrémenté.
NOM	nom de la personne
PRENOM	son prénom
DATENAISANCE	sa date de naissance
MARIE	entier 0 (non marié) ou 1 (marié)
NBENFANTS	nombre d'enfants de la personne

#### 2.1.2 L'entité [Personne]

Nous nous plaçons dans l'environnement d'exécution suivant :



La couche JPA [5] doit faire un pont entre le monde relationnel de la base de données [7] et le monde objet [4] manipulé par les programmes Java [3]. Ce pont est fait par configuration et il y a deux façons de le faire :

1. avec des fichiers XML. C'était quasiment l'unique façon de faire jusqu'à l'avènement du JDK 1.5
2. avec des annotations Java depuis le JDK 1.5

Dans ce document, nous utiliserons quasi exclusivement la seconde méthode.

L'objet [Personne] image de la table [personne] présentée précédemment pourrait être le suivant :

```
1. ...
2.
3. @SuppressWarnings("unused")
4. @Entity
5. @Table(name="Personne")
```

```

6. public class Personne implements Serializable{
7.
8.     @Id
9.     @Column(name = "ID", nullable = false)
10.    @GeneratedValue(strategy = GenerationType.AUTO)
11.    private Integer id;
12.
13.    @Column(name = "VERSION", nullable = false)
14.    @Version
15.    private int version;
16.
17.    @Column(name = "NOM", length = 30, nullable = false, unique = true)
18.    private String nom;
19.
20.    @Column(name = "PRENOM", length = 30, nullable = false)
21.    private String prenom;
22.
23.    @Column(name = "DATENAISSANCE", nullable = false)
24.    @Temporal(TemporalType.DATE)
25.    private Date datenaissance;
26.
27.    @Column(name = "MARIE", nullable = false)
28.    private boolean marie;
29.
30.    @Column(name = "NBENFANTS", nullable = false)
31.    private int nbefants;
32.
33.    // constructeurs
34.    public Personne() {
35.    }
36.
37.    public Personne(String nom, String prenom, Date datenaissance, boolean marie,
38.                      int nbefants) {
39.        setNom(nom);
40.        setPrenom(prenom);
41.        setDatenaissance(datenaissance);
42.        setMarie(marie);
43.        setNbenebants(nbefants);
44.    }
45.
46.    // toString
47.    public String toString() {
48.    ...
49.    }
50.
51.    // getters and setters
52.    ...
53. }

```

La configuration se fait à l'aide d'annotations Java **@Annotation**. Les annotations Java sont soit exploitées par le compilateur, soit par des outils spécialisés au moment de l'exécution. En-dehors de l'annotation de la ligne 3 destinée au compilateur, toutes les annotations sont ici destinées à l'implémentation JPA utilisée, Hibernate ou Toplink. Elles seront donc exploitées à l'exécution. En l'absence des outils capables de les interpréter, ces annotations sont ignorées. Ainsi la classe [Personne] ci-dessus pourrait être exploitée dans un contexte hors JPA.

Il faut distinguer deux cas d'utilisation des annotations JPA dans une classe C associée à une table T :

1. la table T existe déjà : les annotations JPA doivent alors reproduire l'existant (nom et définition des colonnes, contraintes d'intégrité, clés étrangères, clés primaires, ...)
2. la table T n'existe pas et elle va être créée d'après les annotations trouvées dans la classe C.

Le cas 2 est le plus facile à gérer. A l'aide des annotations JPA, nous indiquons la structure de la table T que nous voulons. Le cas 1 est souvent plus complexe. La table T a pu être construite, il y a longtemps, en-dehors de tout contexte JPA. Sa structure peut alors être mal adaptée au pont relationnel / objet de JPA. Pour simplifier, nous nous plaçons dans le cas 1 où la table T associée à la classe C va être créée d'après les annotations JPA de la classe C.

Commentons les annotations JPA de la classe [Personne] :

- ligne 4 : l'annotation **@Entity** est la première annotation indispensable. Elle se place avant la ligne qui déclare la classe et indique que la classe en question doit être gérée par la couche de persistance JPA. En l'absence de cette annotation, toutes les autres annotations JPA seraient ignorées.
- ligne 5 : l'annotation **@Table** désigne la table de la base de données dont la classe est une représentation. Son principal argument est **name** qui désigne le nom de la table. En l'absence de cet argument, la table portera le nom de la classe, ici [Personne]. Dans notre exemple, l'annotation **@Table** est donc superflue.
- ligne 8 : l'annotation **@Id** sert à désigner le champ dans la classe qui est image de la clé primaire de la table. Cette annotation est obligatoire. Elle indique ici que le champ *id* de la ligne 11 est l'image de la clé primaire de la table.

- ligne 9 : l'annotation **@Column** sert à faire le lien entre un champ de la classe et la colonne de la table dont le champ est l'image. L'attribut **name** indique le nom de la colonne dans la table. En l'absence de cet attribut, la colonne porte le même nom que le champ. Dans notre exemple, l'argument **name** n'était donc pas obligatoire. L'argument **nullable=false** indique que la colonne associée au champ ne peut avoir la valeur NULL et que donc le champ doit avoir nécessairement une valeur.
- ligne 10 : l'annotation **@GeneratedValue** indique comment est générée la clé primaire lorsqu'elle est générée automatiquement par le SGBD. Ce sera le cas dans tous nos exemples. Ce n'est pas obligatoire. Ainsi notre personne pourrait avoir un n° étudiant qui servirait de clé primaire et qui ne serait pas généré par le SGBD mais fixé par l'application. Dans ce cas, l'annotation **@GeneratedValue** serait absente. L'argument **strategy** indique comment est générée la clé primaire lorsqu'elle est générée par le SGBD. Les SGBD n'ont pas tous la même technique de génération des valeurs de clé primaire. Par exemple :

<b>Firebird</b>	utilise un générateur de valeurs appelée avant chaque insertion
<b>SQL server</b>	le champ clé primaire est défini comme ayant le type <i>Identity</i> . On a un résultat similaire au générateur de valeurs de Firebird, si ce n'est que la valeur de la clé n'est connue qu'après l'insertion de la ligne.
<b>Oracle</b>	utilise un objet appelé SEQUENCE qui là encore joue le rôle d'un générateur de valeurs

La couche JPA doit générer des ordres SQL différents selon les SGBD pour créer le générateur de valeurs. On lui indique par configuration le type de SGBD qu'elle a à gérer. Du coup, elle peut savoir quelle est la stratégie habituelle de génération de valeurs de clé primaire de ce SGBD. L'argument **strategy = GenerationType.AUTO** indique à la couche JPA qu'elle doit utiliser cette stratégie habituelle. Cette technique a fonctionné dans tous les exemples de ce document pour les sept SGBD utilisés.

- ligne 14 : l'annotation **@Version** désigne le champ qui sert à gérer les accès concurrents à une même ligne de la table.

Pour comprendre ce problème d'accès concurrents à une même ligne de la table [personne], supposons qu'une application web permette la mise à jour d'une personne et examinons le cas suivant :

Au temps T1, un utilisateur U1 entre en modification d'une personne P. A ce moment, le nombre d'enfants est 0. Il passe ce nombre à 1 mais avant qu'il ne valide sa modification, un utilisateur U2 entre en modification de la même personne P. Puisque U1 n'a pas encore validé sa modification, U2 voit sur son écran le nombre d'enfants à 0. U2 passe le nom de la personne P en majuscules. Puis U1 et U2 valident leurs modifications dans cet ordre. C'est la modification de U2 qui va gagner : dans la base, le nom va passer en majuscules et le nombre d'enfants va rester à zéro alors même que U1 croit l'avoir changé en 1.

La notion de version de personne nous aide à résoudre ce problème. On reprend le même cas d'usage :

Au temps T1, un utilisateur U1 entre en modification d'une personne P. A ce moment, le nombre d'enfants est 0 et la version V1. Il passe le nombre d'enfants à 1 mais avant qu'il ne valide sa modification, un utilisateur U2 entre en modification de la même personne P. Puisque U1 n'a pas encore validé sa modification, U2 voit le nombre d'enfants à 0 et la version à V1. U2 passe le nom de la personne P en majuscules. Puis U1 et U2 valident leurs modifications dans cet ordre. Avant de valider une modification, on vérifie que celui qui modifie une personne P détient la même version que la personne P actuellement enregistrée. Ce sera le cas de l'utilisateur U1. Sa modification est donc acceptée et on change alors la version de la personne modifiée de V1 à V2 pour noter le fait que la personne a subi un changement. Lors de la validation de la modification de U2, on va s'apercevoir que U2 détient une version V1 de la personne P, alors qu'actuellement la version de celle-ci est V2. On va alors pouvoir dire à l'utilisateur U2 que quelqu'un est passé avant lui et qu'il doit repartir de la nouvelle version de la personne P. Il le fera, récupérera une personne P de version V2 qui a maintenant un enfant, passera le nom en majuscules, validera. Sa modification sera acceptée si la personne P enregistrée a toujours la version V2. Au final, les modifications faites par U1 et U2 seront prises en compte alors que dans le cas d'usage sans version, l'une des modifications était perdue.

La couche [dao] de l'application cliente peut gérer elle-même la version de la classe [Personne]. A chaque fois qu'il y aura une modification d'un objet P, la version de cet objet sera incrémentée de 1 dans la table. L'annotation **@Version** permet de transférer cette gestion à la couche JPA. Le champ concerné n'a pas besoin de s'appeler *version* comme dans l'exemple. Il peut porter un nom quelconque.

Les champs correspondant aux annotations **@Id** et **@Version** sont des champs présents à cause de la persistance. On n'en aurait pas besoin si la classe [Personne] n'avait pas besoin d'être persistée. On voit donc qu'un objet n'a pas la même représentation selon qu'il a besoin ou non d'être persisté.

- ligne 17 : de nouveau l'annotation **@Column** pour donner des informations sur la colonne de la table [personne] associée au champ *nom* de la classe Personne. On trouve ici deux nouveaux arguments :
  - unique=true** indique que le nom d'une personne doit être unique. Cela va se traduire dans la base de données par l'ajout d'une contrainte d'unicité sur la colonne NOM de la table [personne].

- **length=30** fixe à 30 le nombre de caractères de la colonne NOM. Cela signifie que le type de cette colonne sera VARCHAR(30).
- ligne 24 : l'annotation **@Temporal** sert à indiquer quel type SQL donner à une colonne / champ de type date / heure. Le type *TemporalType.DATE* désigne une date seule sans heure associée. Les autres types possibles sont *TemporalType.TIME* pour coder une heure et *TemporalType.TIMESTAMP* pour coder une date avec heure.

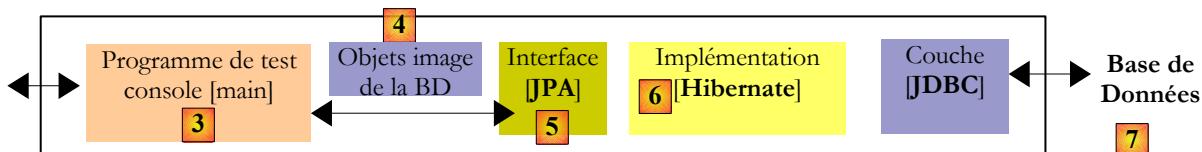
Commentons maintenant le reste du code de la classe [Personne] :

- ligne 6 : la classe implémente l'interface *Serializable*. La *sérialisation* d'un objet consiste à le transformer en une suite de bits. La *désérialisation* est l'opération inverse. La sérialisation / désérialisation est notamment utilisée dans les applications client / serveur où des objets sont échangés via le réseau. Les applications clientes ou serveur sont ignorantes de cette opération qui est faite de façon transparente par les JVM. Pour qu'elle soit possible, il faut cependant que les classes des objets échangés soit " taguées " avec le mot clé *Serializable*.
- ligne 37 : un constructeur de la classe. On notera que les champs **id** et **version** ne font pas partie des paramètres. En effet, ces deux champs sont gérés par la couche JPA et non par l'application.
- lignes 51 et au-delà : les méthodes **get** et **set** de chacun des champs de la classe. Il est à noter que les annotations JPA peuvent être placées sur les méthodes **get** des champs au lieu d'être placées sur les champs eux-mêmes. La place des annotations indique le mode que doit utiliser JPA pour accéder aux champs :
  - si les annotations sont mises au niveau champ, JPA accédera directement aux champs pour les lire ou les écrire
  - si les annotations sont mises au niveau **get**, JPA accèdera aux champs via les méthodes **get** / **set** pour les lire ou les écrire

C'est la position de l'annotation **@Id** qui fixe la position des annotations JPA d'une classe. Placée au niveau champ, elle indique un accès direct aux champs et placée au niveau **get**, un accès aux champs via les **get** et **set**. Les autres annotations doivent alors être placées de la même façon que l'annotation **@Id**.

### 2.1.3 Le projet Eclipse des tests

Nous allons mener nos premières expérimentations avec l'entité [Personne] précédente. Nous les mènerons avec l'architecture suivante :



- en [7] : la base de données qui sera générée à partir des annotations de l'entité [Personne] ainsi que de configurations complémentaires faites dans un fichier appelé [persistence.xml]
- en [5, 6] : une couche JPA implémentée par Hibernate
- en [4] : l'entité [Personne]
- en [3] : un programme de test de type console

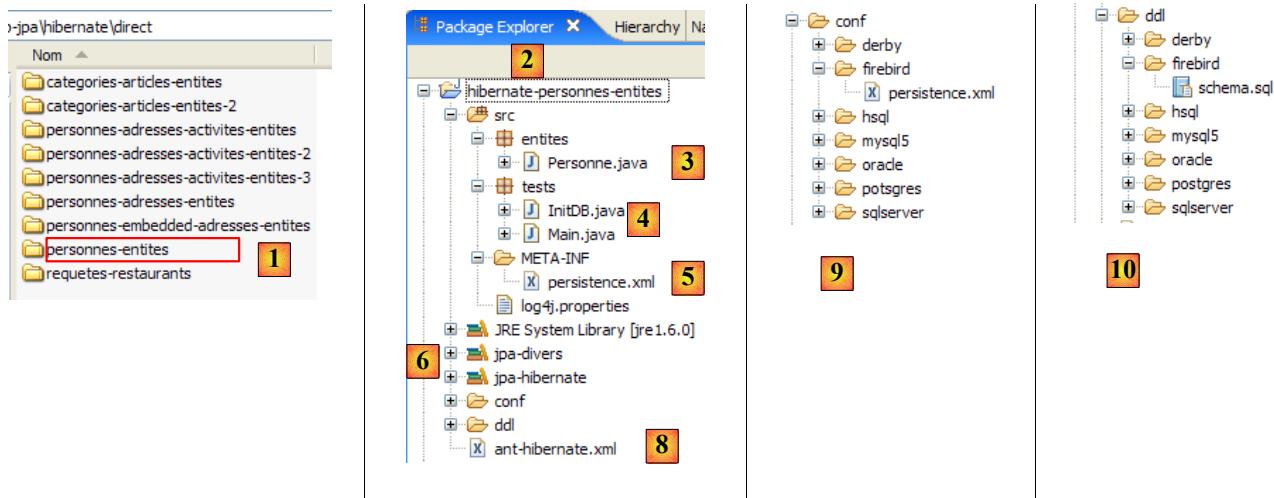
Nous ferons diverses expérimentations :

- générer le schéma de la BD à partir d'un script **ant** et de l'outil **Hibernate Tools**
- générer la BD et l'initialiser avec quelques données
- exploiter la BD et réaliser les quatre opérations de base sur la table [personne] (insertion, mise à jour, suppression, interrogation)

Les outils nécessaires sont les suivants :

- Eclipse et ses plugins décrit au paragraphe 5.2, page 209.
- le projet [hibernate-personnes-entites] qu'on trouvera dans le dossier <exemples>/hibernate/direct/personnes-entites
- les divers SGBD décrits en annexes (page 209 et au-delà).

Le projet Eclipse est le suivant :



- en [1] : le dossier du projet Eclipse
- en [2] : le projet importé dans Eclipse (File / Import)
- en [3] : l'entité [Personne] objet des tests
- en [4] : les programmes de test
- en [5] : [persistence.xml] est le fichier de configuration de la couche JPA
- en [6] : les bibliothèques utilisées. Elles ont été décrites au paragraphe 1.5, page 7.
- en [8] : un script ant qui sera utilisé pour générer la table associée à l'entité [Personne]
- en [9] : les fichiers [persistence.xml] pour chacun des SGBD utilisés
- en [10] : les schémas de la base de données générée pour chacun des SGBD utilisés

Nous allons décrire ces éléments les uns après les autres.

## 2.1.4 L'entité [Personne] (2)

Nous amenons une légère modification à la description faite précédemment de l'entité [Personne] ainsi qu'un complément d'information :

```

1. package entites;
2.
3. ...
4.
5. @SuppressWarnings({ "unused", "serial" })
6. @Entity
7. @Table(name="jpa01_personne")
8. public class Personne implements Serializable{
9.
10.    @Id
11.    @Column(name = "ID", nullable = false)
12.    @GeneratedValue(strategy = GenerationType.AUTO)
13.    private Integer id;
14.
15.    @Column(name = "VERSION", nullable = false)
16.    @Version
17.    private int version;
18.
19.    @Column(name = "NOM", length = 30, nullable = false, unique = true)
20.    private String nom;
21.
22.    @Column(name = "PRENOM", length = 30, nullable = false)
23.    private String prenom;
24.
25.    @Column(name = "DATENAISSANCE", nullable = false)
26.    @Temporal(TemporalType.DATE)
27.    private Date datenaissance;
28.
29.    @Column(name = "MARIE", nullable = false)
30.    private boolean marie;
31.
32.    @Column(name = "NBENFANTS", nullable = false)
33.    private int nbenfants;
34.
35.    // constructeurs
36.    public Personne() {
37.    }
38.

```

```

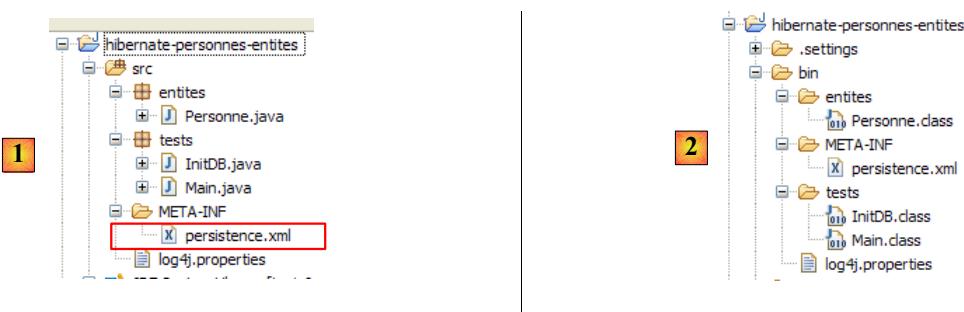
39. public Personne(String nom, String prenom, Date datenaissance, boolean marie,
40.         int nbenfants) {
41.     ...
42. }
43.
44. // toString
45. public String toString() {
46.     return String.format("[%d,%d,%s,%s,%s,%d]", getId(), getVersion(),
47.             getNom(), getPrenom(), new SimpleDateFormat("dd/MM/yyyy")
48.                 .format(getDatenaissance()), isMarie(), getNbenfants());
49. }
50.
51. // getters and setters
52. ...
53. }

```

- ligne 7 : nous donnons le nom [jpa01\_personne] à la table associée à l'entité [Personne]. Dans le document, diverses tables vont être créées dans un schéma toujours appelé **jpa**. A la fin de ce tutoriel, le schéma **jpa** contiendra de nombreuses tables. Afin que le lecteur s'y retrouve, les tables liées entre elles auront le même préfixe **jpxxx\_**.
- ligne 45 : une méthode [toString] pour afficher un objet [Personne] sur la console.

## 2.1.5 Configuration de la couche d'accès aux données

Dans le projet Eclipse ci-dessus, la configuration de la couche JPA est assurée par le fichier [META-INF/persistence.xml] :



A l'exécution, le fichier [META-INF/persistence.xml] est cherché dans le *classpath* de l'application. Dans notre projet Eclipse, tout ce qui est dans le dossier [/src] [1] est copié dans un dossier [/bin] [2]. Celui-ci fait partie du *classpath* du projet. C'est pour cette raison que [META-INF/persistence.xml] sera trouvé lorsque la couche JPA se configurera.

Par défaut, Eclipse ne met pas les codes sources dans le dossier [/src] du projet mais directement sous le dossier lui-même. Tous nos projets Eclipse seront eux configurés pour que les sources soient dans [/src] et les classes compilées dans [/bin] comme il est montré au paragraphe 5.2.1, page 212.

Examinons la configuration de la couche JPA faite dans le fichier [persistence.xml] de notre projet :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.     <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.         <!-- provider -->
5.         <provider>org.hibernate.ejb.HibernatePersistence</provider>
6.         <properties>
7.             <!-- Classes persistantes -->
8.             <property name="hibernate.archive.autodetection" value="class, hbm" />
9.             <!-- logs SQL
10.            <property name="hibernate.show_sql" value="true"/>
11.            <property name="hibernate.format_sql" value="true"/>
12.            <property name="use_sql_comments" value="true"/>
13.        <!-->
14.        <!-- connexion JDBC -->
15.        <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
16.        <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/jpa" />
17.        <property name="hibernate.connection.username" value="jpa" />
18.        <property name="hibernate.connection.password" value="jpa" />
19.        <!-- création automatique du schéma -->
20.        <property name="hibernate.hbm2ddl.auto" value="create" />
21.        <!-- Dialecte -->
22.        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
23.        <!-- propriétés DataSource c3p0 -->
24.        <property name="hibernate.c3p0.min_size" value="5" />
25.        <property name="hibernate.c3p0.max_size" value="20" />
26.        <property name="hibernate.c3p0.timeout" value="300" />
27.        <property name="hibernate.c3p0.max_statements" value="50" />

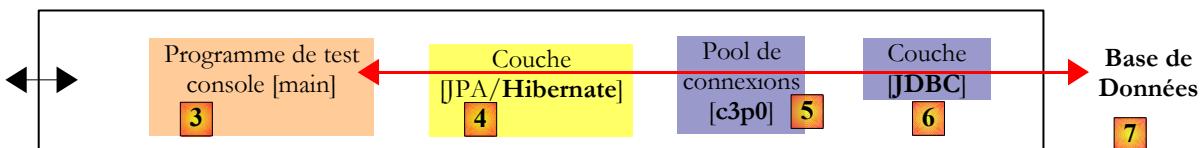
```

```

28.      <property name="hibernate.c3p0.idle_test_period" value="3000" />
29.    </properties>
30.  </persistence-unit>
31. </persistence>

```

Pour comprendre cette configuration, il nous faut revenir sur l'architecture de l'accès aux données de notre application :



- le fichier [persistence.xml] va configurer les couches [4, 5, 6]
- [4] : implémentation Hibernate de JPA
- [5] : Hibernate accède à la base de données via un pool de connexions. Un pool de connexions est une réserve de connexions ouvertes avec le SGBD. Un SGBD est accédé par de multiples utilisateurs alors même que pour des raisons de performances, il ne peut dépasser un nombre limite N de connexions ouvertes simultanément. Un code bien écrit ouvre une connexion avec le SGBD un minimum de temps : il émet des ordres SQL et ferme la connexion. Il va faire cela de façon répétée, à chaque fois qu'il a besoin de travailler avec la base. Le coût d'ouverture / fermeture d'une connexion n'est pas négligeable et c'est là qu'intervient le pool de connexions. Celui-ci va au démarrage de l'application ouvrir N1 connexions avec le SGBD. C'est à lui que l'application demandera une connexion ouverte lorsqu'elle en aura besoin. Celle-ci sera rendue au pool dès que l'application n'en aura plus besoin, de préférence le plus vite possible. La connexion n'est pas fermée et reste disponible pour l'utilisateur suivant. Un pool de connexions est donc un système de partage de connexions ouvertes.
- [6] : le pilote JDBC du SGBD utilisé

Maintenant voyons comment le fichier [persistence.xml] configure les couches [4, 5, 6] ci-dessus :

- ligne 2 : la balise racine du fichier XML est <persistence>.
- ligne 3 : <persistence-unit> sert à définir une unité de persistance. Il peut y avoir plusieurs unités de persistance. Chacune d'elles a un nom (attribut **name**) et un type de transactions (attribut **transaction-type**). L'application aura accès à l'unité de persistance via le nom de celle-ci, ici **jpa**. Le type de transaction RESOURCE\_LOCAL indique que l'application gère elle-même les transactions avec le SGBD. Ce sera le cas ici. Lorsque l'application s'exécute dans un conteneur EJB3, elle peut utiliser le service de transactions de celui-ci. Dans ce cas, on mettra **transaction-type=JTA** (Java Transaction API). JTA est la valeur par défaut lorsque l'attribut **transaction-type** est absent.
- ligne 5 : la balise <provider> sert à définir une classe implémentant l'interface [javax.persistence.spi.PersistenceProvider], interface qui permet à l'application d'initialiser la couche de persistance. Parce qu'on utilise une implémentation JPA / Hibernate, la classe utilisée ici est une classe d'Hibernate.
- ligne 6 : la balise <properties> introduit des propriétés propres au provider particulier choisi. Ainsi selon qu'on a choisi Hibernate, Toplink, Kodo, ... on aura des propriétés différentes. Celles qui suivent sont propres à Hibernate.
- ligne 8 : demande à Hibernate d'explorer le *classpath* du projet pour y trouver les classes ayant l'annotation **@Entity** afin de les gérer. Les classes **@Entity** peuvent également être déclarées par des balises <class>nom\_de\_la\_classe</class>, directement sous la balise <persistence-unit>. C'est ce que nous ferons avec le provider JPA / Toplink.
- les lignes 10-12, ici mises en commentaires configurent les logs console d'Hibernate :
  - ligne 10 : pour afficher ou non les ordres SQL émis par Hibernate sur le SGBD. Ceci est très utile lors de la phase d'apprentissage. À cause du pont relationnel / objet, l'application travaille sur des objets persistants sur lesquels elle applique des opérations de type [persist, merge, remove]. Il est très intéressant de savoir quels sont les ordres SQL réellement émis sur ces opérations. En les étudiant, peu à peu on en vient à deviner les ordres SQL qu'Hibernate va générer lorsqu'on fait telle opération sur les objets persistants et le pont relationnel / objet commence à prendre consistance dans l'esprit.
  - ligne 11 : les ordres SQL affichés sur la console peuvent être formatés joliment pour rendre leur lecture plus aisée
  - ligne 12 : les ordres SQL affichés seront de plus commentés
- les lignes 15-19 définissent la couche JDBC (couche [6] dans l'architecture) :
  - ligne 15 : la classe du pilote JDBC du SGBD, ici MySQL5
  - ligne 16 : l'url de la base de données utilisée
  - lignes 17, 18 : l'utilisateur de la connexion et son mot de passe

Nous utilisons ici des éléments expliqués en annexes au paragraphe 5.5, page 258. Le lecteur est invité à lire cette section sur MySQL5.

- ligne 22 : Hibernate a besoin de connaître le SGBD qu'il a en face de lui. En effet, les SGBD ont tous des extensions SQL propriétaires, une façon propre de gérer la génération automatique des valeurs d'une clé primaire, ... qui font qu'Hibernate a besoin de connaître le SGBD avec qui il travaille afin de lui envoyer les ordres SQL que celui-ci comprendra. [MySQL5InnoDBialect] désigne le SGBD MySQL5 avec des tables de type InnoDB qui supportent les transactions.
- les lignes 24-28 configurent le pool de connexions c3p0 (couche [5] dans l'architecture) :

- lignes 24, 25 : le nombre minimal (défaut 3) et maximal de connexions (défaut 15) dans le pool. Le nombre initial de connexions par défaut est 3.
- ligne 26 : durée maximale en milli-secondes d'attente d'une demande de connexion de la part du client. Passé ce délai, c3p0 lui renverra une exception.
- ligne 27 : pour accéder à la BD, Hibernate utilise des ordres SQL préparés (PreparedStatement) que c3p0 peut mettre en cache. Cela signifie que si l'application demande une seconde fois un ordre SQL préparé déjà en cache, celui-ci n'aura pas besoin d'être préparé (la préparation d'un ordre SQL a un coût) et celui qui est en cache sera utilisé. Ici, on indique le nombre maximal d'ordres SQL préparés que le cache peut contenir, toutes connexions confondues (un ordre SQL préparé appartient à une connexion).
- ligne 28 : fréquence de vérification en milli-secondes de la validité des connexions. Une connexion du pool peut devenir invalide pour diverses raisons (le pilote JDBC invalide la connexion parce qu'elle est trop longue, le pilote JDBC présente des "bugs", ...).
- ligne 20 : on demande ici, qu'à l'initialisation de l'unité de persistance, la base de données image des objets `@Entity` soit générée. Hibernate a désormais tous les outils pour émettre les ordres SQL de génération des tables de la base de données :

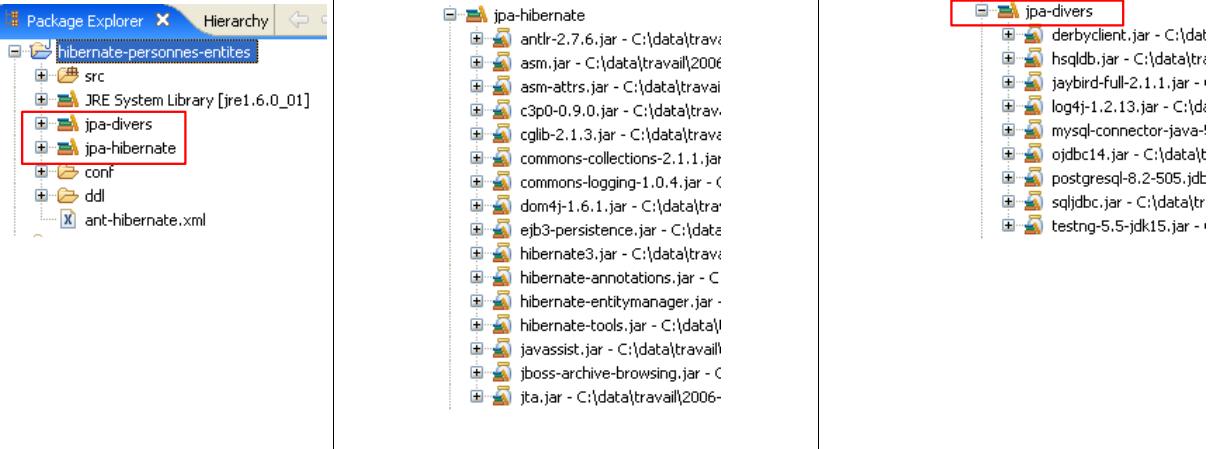
  - la configuration des objets `@Entity` lui permet de connaître les tables à générer
  - les lignes 15-18 et 24-28 lui permettent d'obtenir une connexion avec le SGBD
  - la ligne 22 lui permet de savoir quel dialecte SQL utiliser pour générer les tables

Ainsi le fichier [persistence.xml] utilisé ici recrée une base neuve à chaque nouvelle exécution de l'application. Les tables sont recréées (*create table*) après avoir été détruites (*drop table*) si elles existaient. On notera que ce n'est évidemment pas à faire avec une base en production...

Les tests ont montré que la phase *drop / create* des tables pouvait échouer. Cela a notamment été le cas lorsque, pour un même test, on passait d'une couche JPA/Hibernate à une couche JPA/Toplink ou vice-versa. A partir des mêmes objets `@Entity`, les deux implémentations ne génèrent pas strictement les mêmes tables, générateurs, séquences, ... et il est arrivé parfois, que la phase *drop / create* échoue et qu'on soit obligés de supprimer les tables à la main. La partie "Annexes", page 209 et au-delà, décrit les applications utilisables pour faire ce travail à la main. On notera que l'implémentation JPA/Hibernate s'est montrée la plus efficace dans cette phase de création initiale du contenu de la base : rares ont été les plantages.

Les outils utilisés par la couche JPA / Hibernate sont dans la bibliothèque [jpa-hibernate], présentée au paragraphe 1.5, page 7. Les pilotes JDBC nécessaires pour accéder aux SGBD sont dans la bibliothèque [jpa-divers]. Ces deux bibliothèques ont été mises dans le *classpath* du projet étudié ici. Nous rappelons ci-dessous leur contenu :





## 2.1.6 Génération de la base de données avec un script Ant

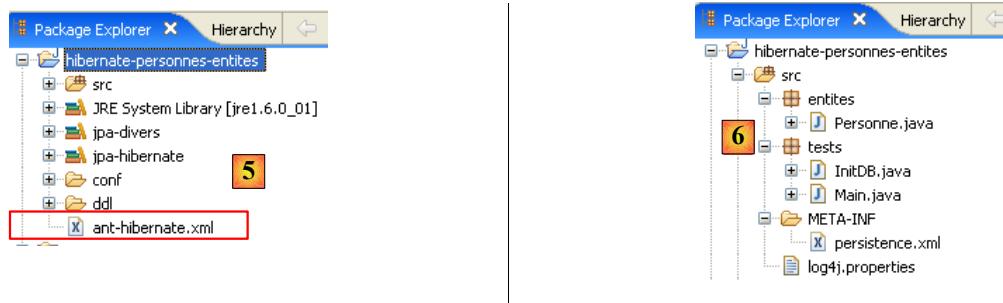
Nous venons de le voir, Hibernate fournit des outils pour générer la base de données image des objets @Entity de l'application. Hibernate peut :

- générer le fichier texte des ordres SQL générant la base. Seul le dialecte dans [persistence.xml] est alors utilisé.
- créer les tables images des objets @Entity dans la base de données cible définie dans [persistence.xml]. C'est alors la totalité du fichier [persistence.xml] qui est utilisé.

Nous allons présenter un script **Ant** capable de générer le schéma de la base de données, image des objets @Entity. Ce script n'est pas le mien : il reprend un script analogue de [ref1]. **Ant** (Another Neat Tool) est un outil de batch de tâches Java. Les scripts *Ant* ne sont pas simples à comprendre pour le néophyte. Nous n'en utiliserons qu'un seul, celui que nous commentons maintenant :



- en [1] : l'arborescence des exemples de ce tutoriel.
- en [2] : le dossier [personnes-entites] du projet Eclipse actuellement étudié
- en [3] : le dossier <lib> contenant les cinq bibliothèques de jars définies page 7.
- en [4] : l'archive [hibernate-tools.jar] nécessaire à l'une des tâches du script [ant-hibernate.xml] que nous allons étudier.



- en [5] : le projet Eclipse et le script [ant-hibernate.xml]
- en [6] : le dossier [src] du projet

Le script [ant-hibernate.xml] [5] va utiliser les archives jars du dossier <lib> [3], notamment l'archive [hibernate-tools.jar] [4] du dossier [lib/hibernate]. Nous avons reproduit l'arborescence des dossiers afin que le lecteur voit que pour trouver le dossier [lib] à partir du dossier [personnes-entites] [2] du script [ant-hibernate.xml], il faut suivre le chemin : ../../..//lib.

Examinons le script [ant-hibernate.xml] :

```

1. <project name="jpa-hibernate" default="compile" basedir=".">
2.
3. <!-- nom du projet et version -->
4. <property name="proj.name" value="jpa-hibernate" />
5. <property name="proj.shortname" value="jpa-hibernate" />
6. <property name="version" value="1.0" />
7.
8. <!-- Propriété globales -->
9. <property name="src.java.dir" value="src" />
10. <property name="lib.dir" value="../../..//lib" />
11. <property name="build.dir" value="bin" />
12.
13. <!-- le Classpath du projet -->
14. <path id="project.classpath">
15.   <fileset dir="${lib.dir}">
16.     <include name="**/*.jar" />
17.   </fileset>
18. </path>
19.
20. <!-- les fichiers de configuration qui doivent être dans le classpath-->
21. <patternset id="conf">
22.   <include name="**/*.xml" />
23.   <include name="**/*.properties" />
24. </patternset>
25.
26. <!-- Nettoyage projet -->
27. <target name="clean" description="Nettoyer le projet">
```

```

28.      <delete dir="${build.dir}" />
29.      <mkdir dir="${build.dir}" />
30.  </target>
31.
32.  <!-- Compilation projet -->
33. <target name="compile" depends="clean">
34.   <javac srcdir="${src.java.dir}" destdir="${build.dir}" classpathref="project.classpath" />
35.  </target>
36.
37.  <!-- Copier les fichiers de configuration dans le classpath -->
38. <target name="copyconf">
39.   <mkdir dir="${build.dir}" />
40.   <copy todir="${build.dir}">
41.     <fileset dir="${src.java.dir}">
42.       <patternset refid="conf" />
43.     </fileset>
44.   </copy>
45.  </target>
46.
47.  <!-- Hibernate Tools -->
48. <taskdef name="hibernatetool" classname="org.hibernate.tool.ant.HibernateToolTask"
classpathref="project.classpath" />
49.
50.  <!-- Générer la DDL de la base -->
51. <target name="DDL" depends="compile, copyconf" description="Génération DDL base">
52.
53.   <hibernatetool destdir="${basedir}">
54.     <classpath path="${build.dir}" />
55.     <!-- Utiliser META-INF/persistence.xml -->
56.     <jpaconfiguration />
57.     <!-- export -->
58.     <hb2ddl drop="true" create="true" export="false" outputfilename="ddl/schema.sql"
delimiter=";" format="true" />
59.   </hibernatetool>
60.  </target>
61.
62.  <!-- Générer la base -->
63. <target name="BD" depends="compile, copyconf" description="Génération BD">
64.
65.   <hibernatetool destdir="${basedir}">
66.     <classpath path="${build.dir}" />
67.     <!-- Utiliser META-INF/persistence.xml -->
68.     <jpaconfiguration />
69.     <!-- export -->
70.     <hb2ddl drop="true" create="true" export="true" outputfilename="ddl/schema.sql"
delimiter=";" format="true" />
71.   </hibernatetool>
72.  </target>
73. </project>

```

- ligne 1 : le projet [ant] s'appelle "**jpa-hibernate**". Il rassemble un ensemble de tâches dont l'une est la tâche par défaut : ici la tâche nommée "**compile**". Un script *ant* est appelé pour exécuter une tâche T. Si celle-ci n'est pas précisée, c'est la tâche par défaut qui est exécutée. **basedir="."** indique que pour tous les chemins relatifs trouvés dans le script, le point de départ est le dossier dans lequel se trouve le script *ant*, ici le dossier <exemples>/hibernate/direct/personnes-entites.
- lignes 3-11 : définissent des variables de script avec la balise **<property name="nomVariable" value="valeurVariable"/>**. La variable peut ensuite être utilisée dans le script avec la notation  **\${nomVariable}**. Les noms peuvent être quelconques. Attardons-nous sur les variables définies aux lignes 9-11 :
  - ligne 9 : définit une variable nommée "**src.java.dir**" (le nom est libre) qui va, dans la suite du script, désigner le dossier qui contient les codes source Java. Sa valeur est "**src**", un chemin relatif au dossier désigné par l'attribut **basedir** (ligne 1). Il s'agit donc du chemin ".src" où . désigne ici le dossier <exemples>/hibernate/direct/personnes-entites. C'est bien dans le dossier <personnes-entites>/src que se trouvent les codes source Java (cf [6] plus haut).
  - ligne 10 : définit une variable nommée "**lib.dir**" qui va, dans la suite du script, désigner le dossier qui contient les archives jars dont ont besoin les tâches Java du script. Sa valeur **.../..../lib** désigne le dossier <exemples>/lib (cf [3] plus haut).
  - ligne 11 : définit une variable nommée "**build.dir**" qui va, dans la suite du script, désigner le dossier où doivent être générés les .class issus de la compilation des sources .java. Sa valeur "**bin**" désigne le dossier <personnes-entites>/bin. Nous avons déjà expliqué que dans le projet Eclipse étudié, le dossier <bin> était celui où étaient générés les .class. Ant va faire de même.
  - lignes 14-18 : la balise **<path>** sert à définir des éléments du *classpath* que devront utiliser les tâches *ant*. Ici, le path "**project.classpath**" (le nom est libre) rassemble toutes les archives .jar de l'arborescence du dossier <exemples>/lib.
  - lignes 21-24 : la balise **<patternset>** sert à désigner un ensemble de fichiers par des modèles de noms. Ici, le **patternset** nommé **conf** désigne tous les fichiers ayant le suffixe **.xml** ou **.properties**. Ce **patternset** va servir à désigner les fichiers .xml et .properties du dossier <src> (persistence.xml, log4j.properties) (cf [6]) qui sont des fichiers de configuration de l'application. Au moment de l'exécution de certaines tâches, ces fichiers doivent être

recopiés dans le dossier <bin> afin qu'ils soient dans le *classpath* du projet. On utilisera alors le *patternset conf*, pour les désigner.

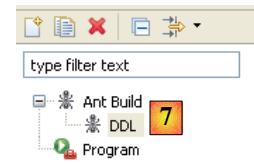
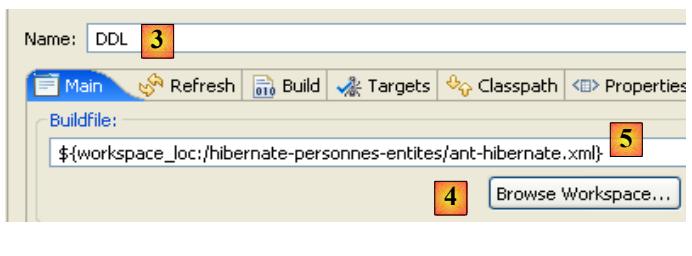
- lignes 27-30 : la balise <**target**> désigne une tâche du script. C'est la première que nous rencontrons. Tout ce qui a précédé relève de la configuration de l'environnement d'exécution du script *ant*. La tâche s'appelle **clean**. Elle s'exécute en deux temps : le dossier <bin> est supprimé (ligne 28) pour être ensuite recréé (ligne 29).
- lignes 33-35 : la tâche **compile** qui est la tâche par défaut du script (ligne 1). Elle dépend (attribut **depends**) de la tâche **clean**. Cela signifie qu'avant d'exécuter la tâche **compile**, *ant* doit exécuter la tâche **clean**, c.a.d. nettoyer le dossier <bin>. Le but de la tâche **compile** est ici de compiler les sources Java du dossier <src>.
- ligne 34 : appel du compilateur Java avec trois paramètres :
  - **srcdir** : le dossier contenant les sources java, ici le dossier <src>
  - **destdir** : le dossier où doivent être rangés les .class générés, ici le dossier <bin>
  - **classpathref** : le classpath à utiliser pour la compilation, ici toutes les archives jar de l'arborescence du dossier <lib>
- lignes 38-45 : la tâche **copyconf** dont le but est de copier dans le dossier <bin> tous les fichiers .xml et .properties du fichier <src>.
- ligne 48 : définition d'une tâche à l'aide de la balise <**taskdef**>. Une telle tâche a vocation à être réutilisée ailleurs dans le script. C'est une facilité de codage. Parce que la tâche est utilisée à divers endroits du script, on la définit une fois avec la balise <taskdef> et on la réutilise ensuite via son nom, lorsqu'on en a besoin.
  - la tâche s'appelle *hibernatetool* (attribut **name**).
  - sa classe est définie par l'attribut **classname**. Ici, la classe désignée sera trouvée dans l'archive [hibernate-tools.jar] dont nous avons déjà parlée.
  - l'attribut **classpathref** indique à *ant* où chercher la classe précédente
- les lignes 51-60 concernent la tâche qui nous intéresse ici, celle de la génération du schéma de la base de données image des objets @Entity de notre projet Eclipse.
  - ligne 51 : la tâche s'appelle **DDL** (comme Data Definition Language, le SQL associé à la création des objets d'une base de données). Elle dépend des tâches **compile** et **copyconf** dans cet ordre. La tâche DDL va donc provoquer, dans l'ordre, l'exécution des tâches **clean**, **compile** et **copyconf**. Lorsque la tâche DDL démarre, le dossier <bin> contient les .class des sources .java, notamment des objets **@Entity**, ainsi que le fichier [META-INF/persistence.xml] qui configure la couche JPA / Hibernate.
  - lignes 53-59 : la tâche [hibernatetool] définie ligne 48 est appellée. On lui passe de nombreux paramètres, outre ceux déjà définis ligne 48 :
    - ligne 53 : le dossier de sortie des résultats produits par la tâche sera le dossier courant .
    - ligne 54 : le *classpath* de la tâche sera le dossier <bin>
    - ligne 56 : indique à la tâche [hibernatetool] comment elle peut connaître son environnement d'exécution : la balise <**jpaconfiguration**>/> lui indique qu'elle est dans un environnement JPA et qu'elle doit donc utiliser le fichier [META-INF/persistence.xml] qu'elle trouvera ici dans son *classpath*.
    - la ligne 58 fixe les conditions de génération de la base de données : **drop=true** indique que des ordres SQL *drop table* doivent être émis avant la création des tables, **create=true** indique que le fichier texte des ordres SQL de création de la base doit être créé, **outputfilename** indique le nom de ce fichier SQL - ici *schema.sql* dans le dossier <dd> du projet Eclipse, **export=false** indique que les ordres SQL générés ne doivent pas être joués dans une connexion au SGBD. Ce point est important : il implique que pour exécuter la tâche, le SGBD cible n'a pas besoin d'être lancé. **delimiter** fixe le caractère qui sépare deux ordres SQL dans le schéma généré, **format=true** demande à ce qu'un formatage de base soit fait sur le texte généré.
  - les lignes 63-72 définissent la tâche nommée **BD**. Elle est identique à la tâche **DDL** précédente, si ce n'est que cette fois elle génère la base de données (**export="true"** de la ligne 70). La tâche ouvre une connexion sur le SGBD avec les informations trouvées dans [persistence.xml], pour y jouer le schéma SQL et générer la base de données. Pour exécuter la tâche BD, il faut donc que le SGBD soit lancé.

## 2.1.7 Exécution de la tâche ant DDL

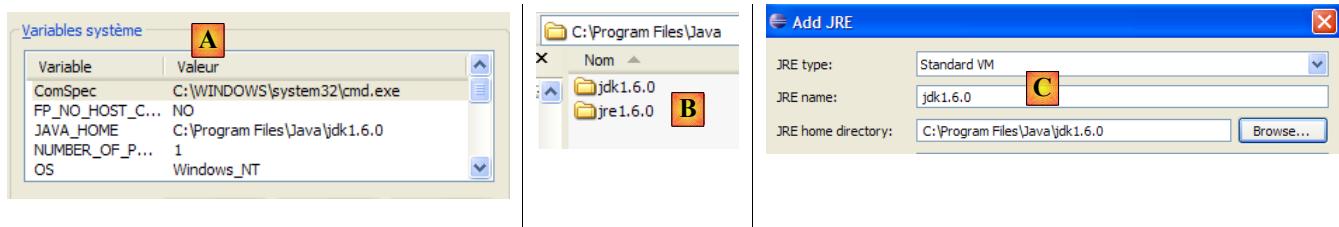
Pour exécuter le script [ant-hibernate.xml], il nous faut faire tout d'abord quelques configurations au sein d'Eclipse.



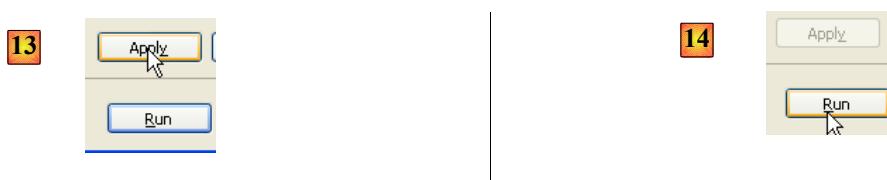
- en [1] : sélectionner [External Tools]
- en [2] : créer une nouvelle configuration *ant*



- en [3] : donner un nom à la configuration *ant*
- en [5] : désigner le script *ant* à l'aide du bouton [4]
- en [6] : appliquer les modifications
- en [7] : on a créé la configuration *ant* DDL



- en [8] : dans l'onglet JRE, on définit le JRE à utiliser. Le champ [10] est normalement pré-rempli avec le JRE utilisé par Eclipse. Il n'y a donc normalement rien à faire sur ce panneau. Néanmoins j'ai rencontré un cas où le script *ant* n'arrivait pas à trouver le compilateur <javac>. Celui-ci n'est pas dans un JRE (Java Runtime Environment) mais dans un JDK (Java Development Kit). L'outil *ant* d'Eclipse trouve ce compilateur via la variable d'environnement JAVA\_HOME (Démarrer / Panneau de configuration / Performances et Maintenance / Système / onglet Avancé / bouton Variables d'environnement) [A]. Si cette variable n'a pas été définie, on peut permettre à *ant* de trouver le compilateur <javac> en mettant dans [10], non pas un JRE mais un JDK. Celui-ci est disponible dans le même dossier que le JRE [B]. On utilisera le bouton [9] pour déclarer le JDK parmi les JRE disponibles [C] afin de pouvoir ensuite le sélectionner dans [10].
- en [12] : dans l'onglet [Targets], on sélectionne la tâche DDL. Ainsi la configuration *ant* que nous avons appelée DDL [7] correspondra à l'exécution de la tâche appelée DDL [12] qui, on le sait, génère le schéma DDL de la base de données image des objets @Entity de l'application.



- en [13] : on valide la configuration
- en [14] : on l'exécute

On obtient dans la vue [console] des logs de l'exécution de la tâche *ant* DDL :

```

1. Buildfile: C:\data\2006-2007\eclipse\dvp-jpa\hibernate\direct\personnes-entites\ant-hibernate.xml
2. clean:
3.      [delete] Deleting directory C:\data\2006-2007\eclipse\dvp-jpa\hibernate\direct\personnes-
        entites\bin
4.      [mkdir] Created dir: C:\data\2006-2007\eclipse\dvp-jpa\hibernate\direct\personnes-entites\bin
5. compile:
6.      [javac] Compiling 3 source files to C:\data\2006-2007\eclipse\dvp-
        jpa\hibernate\direct\personnes-entites\bin
7. copyconf:
8.      [copy] Copying 2 files to C:\data\2006-2007\eclipse\dvp-jpa\hibernate\direct\personnes-
        entites\bin
9. DDL:
10. [hibernatetool] Executing Hibernate Tool with a JPA Configuration
11. [hibernatetool] 1. task: hbm2ddl (Generates database schema)
12. [hibernatetool] drop table if exists jpa01_personne;
13. [hibernatetool] create table jpa01_personne (
14. [hibernatetool] ID integer not null auto_increment,
15. [hibernatetool] VERSION integer not null,
16. [hibernatetool] NOM varchar(30) not null unique,
17. [hibernatetool] PRENOM varchar(30) not null,
18. [hibernatetool] DATENAISSANCE date not null,
19. [hibernatetool] MARIE bit not null,
20. [hibernatetool] NBENFANTS integer not null,
```

```

21. [hibernatetool] primary key (ID)
22. [hibernatetool] ) ENGINE=InnoDB;
23. BUILD SUCCESSFUL
24. Total time: 5 seconds

```

- on se rappelle que la tâche DDL a pour nom [hibernatetool] (ligne 10) et qu'elle dépend des tâches *clean* (ligne 2), *compile* (ligne 5) et *copyconf* (ligne 7).
- ligne 10 : la tâche [hibernatetool] exploite le fichier [persistence.xml] d'une configuration JPA
- ligne 11 : la tâche [hbm2ddl] va générer le schéma DDL de la base de données
- lignes 12-22 : le schéma DDL de la base de données

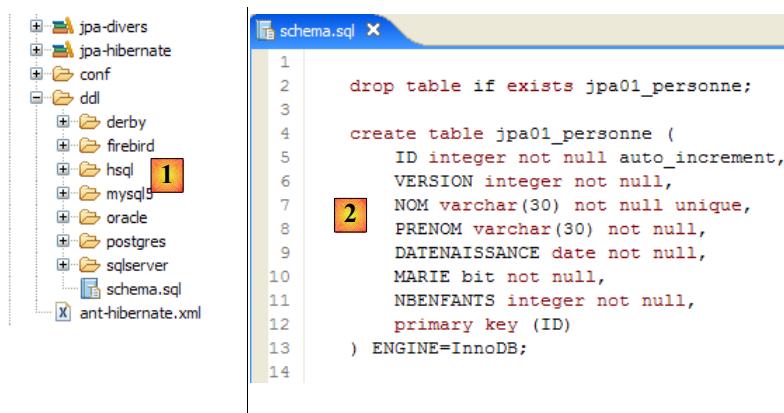
On se souvient qu'on avait demandé à la tâche [hbm2ddl] de générer le schéma DDL à un endroit précis :

```

74.      <hbm2ddl drop="true" create="true" export="true" outputfilename="ddl/schema.sql"
           delimiter=";" format="true" />

```

- ligne 74 : le schéma doit être généré dans le fichier *ddl/schema.sql*. Vérifions :



- en [1] : le fichier *ddl/schema.sql* est bien présent (faire F5 pour rafraîchir l'arborescence)
- en [2] : son contenu. Celui-ci est le schéma d'une base MySQL5. Le fichier [persistence.xml] de configuration de la couche JPA précisait en effet un SGBD MySQL5 (ligne 8 ci-dessous) :

```

1.      <!-- connexion JDBC -->
2.      <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
3.      ...
4.      <!-- création automatique du schéma -->
5.      <property name="hibernate.hbm2ddl.auto" value="create" />
6.      <!-- Dialecte -->
7.      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
8.      <!-- propriétés DataSource c3p0 -->
9.      ...
10.     ...

```

Examinons le pont objet / relationnel qui a été fait ici en examinant la configuration de l'objet @Entity Personne et le schéma DDL généré :

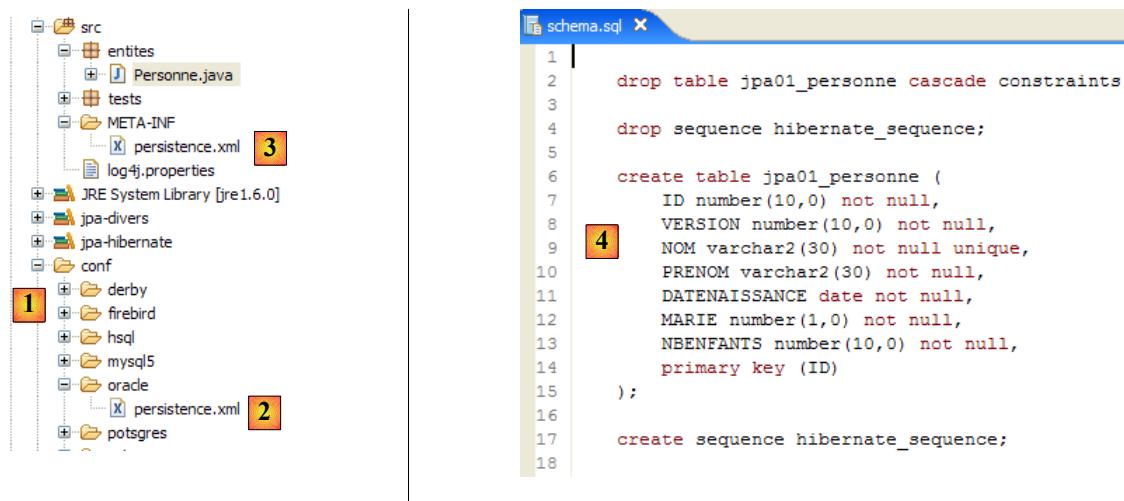
<code>@Entity @Table(name="jpa01_personne") public class Personne {</code>	<b>A1</b>	<code>drop table if exists jpa01_personne;</code>
<code>    @Id     @Column(name = "ID", nullable = false)     @GeneratedValue(strategy = GenerationType.AUTO)     private Integer id;</code>	<b>A2</b>	<code>B1 create table jpa01_personne (</code>
<code>    @Column(name = "VERSION", nullable = false)     @Version     private int version;</code>		<code>        create table jpa01_personne (</code>
<code>    @Column(name = "NOM", length = 30, nullable = false,     unique = true)     private String nom;</code>		<code>            ID integer not null auto_increment,</code>
<code>    @Column(name = "PRENOM", length = 30, nullable = false)     private String prenom;</code>		<code>            primary key (ID)</code>
<code>    @Column(name = "DATENAISSEANCE", nullable = false)</code>		<code>        ) ENGINE=InnoDB;</code>
		<code>        VERSION integer not null,</code>
		<code>        NOM varchar(30) not null unique,</code>
		<code>        PRENOM varchar(30) not null,</code>
		<code>        DATENAISSEANCE date not null,</code>

<pre>@Temporal(TemporalType.DATE) private Date datenaissance;</pre>		
<pre>@Column(name = "MARIE", nullable = false) private boolean marie; A3</pre>	<b>B3</b>	MARIE bit not null,
<pre>@Column(name = "NBENFANTS", nullable = false) private int nbefants;</pre>		NBENFANTS integer not null,

On notera quelques points :

- A1-B1 : le nom de la table précisée en A1 est bien celle utilisée en B1. On notera le *drop* qui précède le *create* en B1.
- A2-B2 : montrent le mode de génération de la clé primaire. Le mode AUTO précisé en A2 s'est traduit par l'attribut *autoincrement* propre à MySQL5. Le mode de génération de la clé primaire est le plus souvent spécifique au SGBD.
- A3-B3 : montrent le type SQL **bit** propre à MySQL5 pour représenter un type *boolean* Java.

Recommençons ce test avec un autre SGBD :



- le dossier [conf] [1] contient les fichiers [persistence.xml] pour divers SGBD. Prendre celui d'Oracle [2] par exemple et le mettre dans le dossier [META-INF] [3] à la place du précédent. Son contenu est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.     <!-- provider -->
5.     <provider>org.hibernate.ejb.HibernatePersistence</provider>
6.     <properties>
7.       <!-- Classes persistantes -->
8.       <property name="hibernate.archive.autodetection" value="class, hbm" />
9.       <!-- logs SQL
10.        <property name="hibernate.show_sql" value="true"/>
11.        <property name="hibernate.format_sql" value="true"/>
12.        <property name="use_sql_comments" value="true"/>
13.      -->
14.      <!-- connexion JDBC -->
15.      <property name="hibernate.connection.driver_class" value="oracle.jdbc.OracleDriver" />
16.      <property name="hibernate.connection.url" value="jdbc:oracle:thin:@localhost:1521:xe" />
17.      <property name="hibernate.connection.username" value="jpa" />
18.      <property name="hibernate.connection.password" value="jpa" />
19.      <!-- création automatique du schéma -->
20.      <property name="hibernate.hbm2ddl.auto" value="create" />
21.      <!-- Dialecte -->
22.      <property name="hibernate.dialect" value="org.hibernate.dialect.OracleDialect" />
23.      <!-- propriétés DataSource C3P0 -->
24.      <property name="hibernate.c3p0.min_size" value="5" />
25.      <property name="hibernate.c3p0.max_size" value="20" />
26.      <property name="hibernate.c3p0.timeout" value="300" />
27.      <property name="hibernate.c3p0.max_statements" value="50" />
28.      <property name="hibernate.c3p0.idle_test_period" value="3000" />
29.    </properties>
30.  </persistence-unit>
31. </persistence>

```

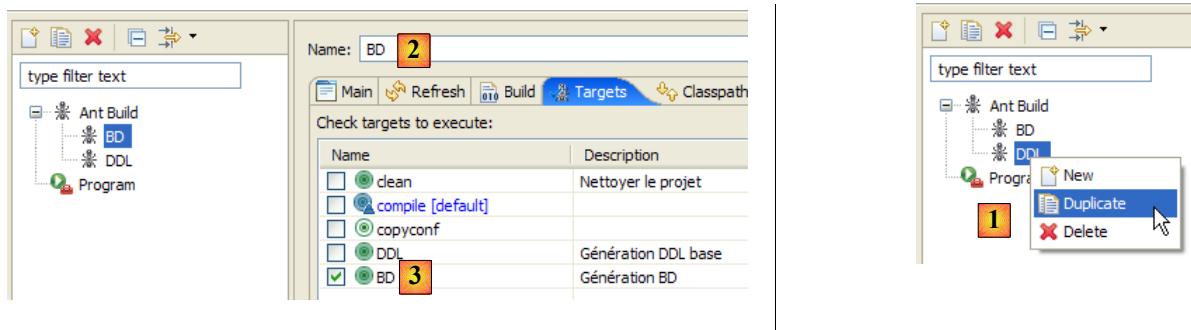
Le lecteur est invité à lire en annexes, la section sur Oracle (paragraphe 5.7, page 276), notamment pour comprendre la configuration JDBC.

Seule la ligne 25 est véritablement importante ici : on indique à Hibernate que désormais le SGBD est un SGBD Oracle. L'exécution de la tâche ant DDL donne le résultat [4] ci-dessus. On remarquera que le schéma Oracle est différent du schéma MySQL5. C'est un point fort de JPA : le développeur n'a pas besoin de se préoccuper de ces détails, ce qui augmente considérablement la portabilité de ses développements.

## 2.1.8 Exécution de la tâche ant BD

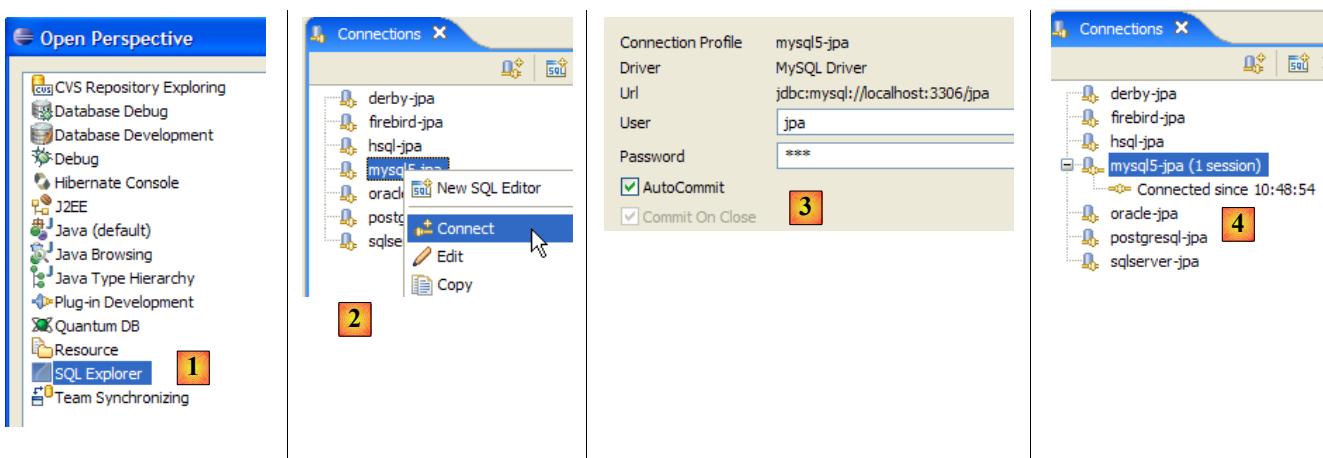
On se rappelle peut-être que la tâche *ant* nommée BD fait la même chose que la tâche *ant* DDL mais génère de plus la base de données. Il faut donc que le SGBD soit lancé. Nous nous placerons dans le cas du SGBD MySQL5 et nous invitons le lecteur à copier le fichier [conf/mysql5/persistence.xml] dans le dossier [src/META-INF]. Pour contrôler le fonctionnement de la tâche, nous allons utiliser le plugin SQL Explorer (cf paragraphe 5.2.6, page 219) pour vérifier l'état de la BD **jpa** avant et après exécution de la tâche *ant* BD.

Tout d'abord, il nous faut créer une nouvelle configuration *ant* pour exécuter la tâche BD. Le lecteur est invité à suivre la démarche exposée pour la configuration ant DDL au paragraphe 2.1.7, page 20. La nouvelle configuration *ant* s'appellera BD :

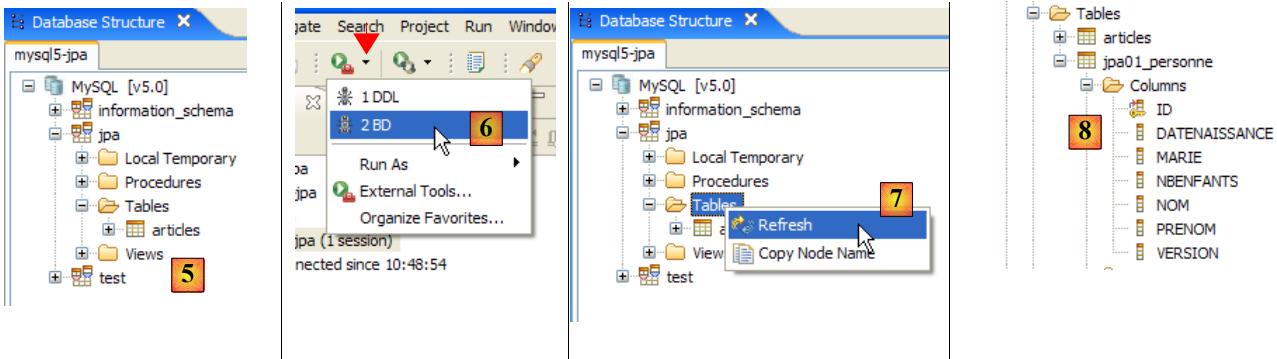


- en [1] : on duplique la configuration précédente appelée DDL
- en [2] : on nomme BD la nouvelle configuration. Elle exécute la tâche *ant* BD [3] qui génère physiquement la base de données.
- ceci fait, lancer le SGBD MySQL5 (paragraphe 5.5, page 258).

Nous utilisons maintenant le plugin SQL Explorer pour explorer les bases gérées par le SGBD. Le lecteur doit auparavant prendre en main ce plugin si besoin est (cf paragraphe 5.2.6, page 219).



- [1] : on ouvre la perspective SQL Explorer [Window / Open Perspective / Other]
- [2] : on crée si besoin est une connexion [mysql5-jpa] (cf paragraphe 5.5.5, page 266) et on l'ouvre
- [3] : on s'identifie jpa / jpa
- [4] : on est connectés à MySQL5.



- en [5] : la BD **jpa** n'a qu'une table : [articles]
- en [6] : on lance l'exécution de la tâche *ant BD*. Parce qu'on est dans la perspective [SQL Explorer], on ne voit pas la vue [Console] qui nous montre les logs de la tâche. On peut afficher cette vue [Window / Show View / ...] ou revenir à la perspective Java [Window / Open Perspective / ...].
- en [7] : une fois la tâche *ant BD* achevée, revenir éventuellement dans la perspective [SQL Explorer] et rafraîchir l'arborescence de la BD **jpa**.
- en [8] : on voit la table [**jpa01\_personne**] qui a été créée.

Le lecteur est invité à refaire cette génération de BD avec d'autres SGBD. La procédure à suivre est la suivante :

- copier le fichier [conf/<sgbd>/persistence.xml] dans le dossier [src/META-INF] où <sgbd> est le SGBD testé
- lancer <sgbd> en suivant les instructions en annexes concernant celui-ci
- dans la perspective SQL Explorer, créer une connexion à <sgbd>. Ceci est également expliqué en annexes pour chacun des SGBD
- refaire les tests précédents

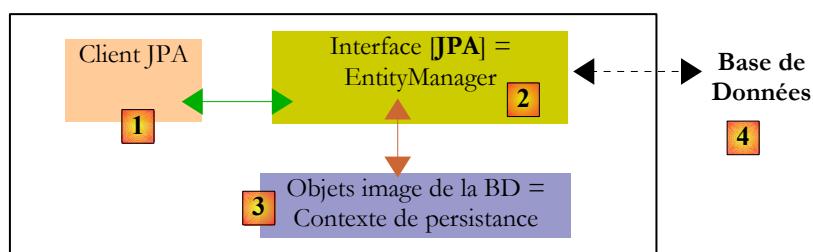
Arrivés ici, nous avons un certain nombre d'acquis :

- nous comprenons mieux la notion de pont objet / relationnel. Ici il a été réalisé par Hibernate. Nous utiliserons plus tard Toplink.
- nous savons que ce pont objet / relationnel est configuré à deux endroits :
  - dans les objets **@Entity**, où on indique les liens entre champs des objets et colonnes des tables de la BD
  - dans **[META-INF/persistence.xml]**, où on donne à l'implémentation JPA des informations sur les deux éléments du pont objet / relationnel : les objets **@Entity** (objet) et la base de données (relationnel).
- nous avons créé deux tâches **ant**, appelées DDL et BD qui nous permettent de créer la base de données à partir de la configuration précédente, avant même toute écriture de code Java.

Maintenant que la couche JPA de notre application est correctement configurée, nous pouvons commencer à explorer l'API JPA avec du code Java.

## 2.1.9 Le contexte de persistance d'une application

Explicitons un peu l'environnement d'exécution d'un client JPA :



Nous savons que le couche JPA [2] crée un pont objet [3] / relationnel [4]. On appelle "contexte de persistance" l'ensemble des objets gérés par la couche JPA dans le cadre de ce pont objet / relationnel. Pour accéder aux données du contexte de persistance, un client JPA [1] doit passer par la couche JPA [2] :

1. il peut créer un objet et demander à la couche JPA de le rendre persistant. L'objet fait alors partie du contexte de persistance.
2. il peut demander à la couche JPA une référence d'un objet persistant existant.
3. il peut modifier un objet persistant obtenu de la couche JPA.

- il peut demander à la couche JPA de supprimer un objet du contexte de persistance.

La couche JPA présente au client une interface appelée [EntityManager] qui, comme son nom l'indique permet de gérer les objets @Entity du contexte de persistance. Nous présentons ci-dessous, les principales méthodes de cette interface :

<code> void persist(Object entity)</code>	met <i>entity</i> dans le contexte de persistance
<code> void remove(Object entity)</code>	enlève <i>entity</i> du contexte de persistance
<code> &lt;T&gt; T merge(T entity)</code>	fusionne un objet <i>entity</i> du client non géré par le contexte de persistance avec l'objet <i>entity</i> du contexte de persistance ayant la même clé primaire. Le résultat rendu est l'objet <i>entity</i> du contexte de persistance.
<code> &lt;T&gt; T find(Class&lt;T&gt; entityClass, Object primaryKey)</code>	met dans le contexte de persistance, un objet cherché dans la base de données via sa clé primaire. Le type T de l'objet permet à la couche JPA de savoir quelle table requêter. L'objet persistant ainsi créé est rendu au client.
<code> Query createQuery(String queryText)</code>	crée un objet <b>Query</b> à partir d'une requête JPQL (Java Persistence Query Language). Une requête JPQL est analogue à une requête SQL si ce n'est qu'on requête des objets plutôt que des tables.
<code> Query createNativeQuery(String queryText)</code>	méthode analogue à la précédente, si ce n'est que <i>queryText</i> est un ordre SQL et non JPQL.
<code> Query createNamedQuery(String name)</code>	méthode identique à <i>createQuery</i> , si ce n'est que l'ordre JPQL <i>queryText</i> a été externalisé dans un fichier de configuration et associé à un nom. C'est ce nom qui est le paramètre de la méthode.

Un objet **EntityManager** a un cycle de vie qui n'est pas forcément celui de l'application. Il a un début et une fin. Ainsi un client JPA peut travailler successivement avec différents objets *EntityManager*. Le contexte de persistance associé à un *EntityManager* a le même cycle de vie que lui. Ils sont indissociables l'un de l'autre. Lorsque un objet *EntityManager* est fermé, son contexte de persistance est si nécessaire synchronisé avec la base de données puis il n'existe plus. Il faut créer un nouvel *EntityManager* pour avoir de nouveau un contexte de persistance.

Le client JPA peut créer un *EntityManager* et donc un contexte de persistance avec l'instruction suivante :

```
| EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa"); |
```

- **javax.persistence.Persistence** est une classe statique permettant d'obtenir une fabrique (factory) d'objets *EntityManager*. Cette fabrique est liée à une unité de persistance précise. On se rappelle que le fichier de configuration [META-INF/persistence.xml] permet de définir des unités de persistance et que celles-ci ont un nom :

```
| <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL"> |
```

Ci-dessus, l'unité de persistance s'appelle *jpa*. Avec elle, vient toute une configuration qui lui est propre, notamment le SGBD avec qui elle travaille. L'instruction [Persistence.createEntityManagerFactory("jpa")] crée une fabrique d'objets de type *EntityManagerFactory* capable de fournir des objets *EntityManager* destinés à gérer des contextes de persistance liés à l'unité de persistance nommée *jpa*. L'obtention d'un objet *EntityManager* et donc d'un contexte de persistance se fait à partir de l'objet *EntityManagerFactory* de la façon suivante :

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

Les méthodes suivantes de l'interface [EntityManager] permettent de gérer le cycle de vie du contexte de persistance :

`|void close()` le contexte de persistance est fermé. Force la synchronisation du contexte de persistance avec la base de données :

- si un objet du contexte n'est pas présent dans la base, il y est mis par une opération SQL INSERT)
- si un objet du contexte est présent dans la base et qu'il a été modifié depuis qu'il a été lu, une opération SQL UPDATE est faite pour persister la modification
- si un objet du contexte a été marqué comme "supprimé" à l'issue d'une opération *remove* sur lui, une opération SQL DELETE est faite pour le supprimer de la base.

`|void clear()` le contexte de persistance est vidé de tous ses objets mais pas fermé.

`|void flush()` le contexte de persistance est synchronisé avec la base de données de la façon décrite pour *close()*

Le client JPA peut forcer la synchronisation du contexte de persistance avec la base de données avec la méthode [EntityManager].*flush* précédente. La synchronisation peut être explicite ou implicite. Dans le premier cas, c'est au client de faire des opérations *flush* lorsqu'il veut faire des synchronisations, sinon celle-ci se font à certains moments que nous allons préciser. Le mode de synchronisation est géré par les méthodes suivantes de l'interface [EntityManager] :

void setFlushMode(FlushModeType flushMode)	Il y a deux valeurs possibles pour <i>flushmode</i> :
	<b>FlushModeType.AUTO</b> (défaut): la synchronisation a lieu avant chaque requête SELECT faite sur la base.
FlushModeType getFlushMode()	<b>FlushModeType.COMMIT</b> : la synchronisation n'a lieu qu'à la fin des transactions sur la base. rend le mode actuel de synchronisation

Résumons. En mode **FlushModeType.AUTO** qui est le mode par défaut, le contexte de persistance sera synchronisé avec la base de données aux moments suivants :

1. avant chaque opération SELECT sur la base
2. à la fin d'une transaction sur la base
3. à la suite d'une opération *flush* ou *close* sur le contexte de persistance

En mode **FlushModeType.COMMIT**, c'est la même chose sauf pour l'opération 1 qui n'a pas lieu. Le mode normal d'interaction avec la couche JPA est un mode transactionnel. Le client fait diverses opérations sur le contexte de persistance, à l'intérieur d'une transaction. Dans ce cas, les moments de synchronisation du contexte de persistance avec la base de données sont les cas 1 et 2 ci-dessus en mode AUTO, et le cas 2 uniquement en mode COMMIT.

Terminons par l'API de l'interface **Query**, interface qui permet d'émettre des ordres JPQL sur le contexte de persistance ou bien des ordres SQL directement sur la base pour y retrouver des données. L'interface **Query** est la suivante :

3	int <u>executeUpdate()</u> Execute an update or delete statement.
1	List <u>getResultSet()</u> Execute a SELECT query and return the query results as a List.
2	Object <u>getSingleResult()</u> Execute a SELECT query that returns a single result.
	Query <u>setFirstResult(int startPosition)</u> Set the position of the first result to retrieve.
	Query <u>setFlushMode(FlushModeType flushMode)</u> Set the flush mode type to be used for the query execution.
	Query <u>setHint(String hintName, Object value)</u> Set an implementation-specific hint.
	Query <u>setMaxResults(int maxResult)</u> Set the maximum number of results to retrieve.
	Query <u>setParameter(int position, Calendar value, TemporalType temporalType)</u> Bind an instance of java.util.Calendar to a positional parameter.
	Query <u>setParameter(int position, Date value, TemporalType temporalType)</u> Bind an instance of java.util.Date to a positional parameter.
5	Query <u>setParameter(int position, Object value)</u> Bind an argument to a positional parameter.
	Query <u>setParameter(String name, Calendar value, TemporalType temporalType)</u> Bind an instance of java.util.Calendar to a named parameter.
	Query <u>setParameter(String name, Date value, TemporalType temporalType)</u> Bind an instance of java.util.Date to a named parameter.
4	Query <u>setParameter(String name, Object value)</u> Bind an argument to a named parameter.

Nous serons amenés à utiliser les méthodes 1 à 4 ci-dessus :

- 1 - la méthode **getResultSet** exécute un SELECT qui ramène plusieurs objets. Ceux-ci seront obtenus dans un objet *List*. Cet objet est une interface. Celle-ci offre un objet *Iterator* qui permet de parcourir les éléments de la liste L sous la forme suivante :

```
1.   Iterator iterator = L.iterator();
2.   while (iterator.hasNext()) {
3.       // exploiter l'objet iterator.next() qui représente l'élément courant de la liste
4.   ...
5. }
```

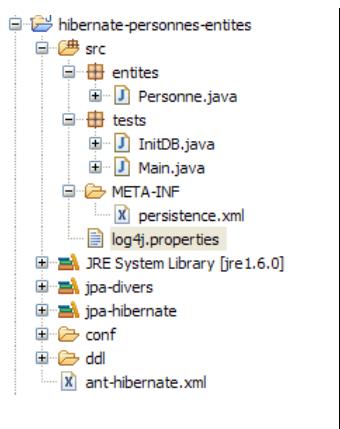
La liste L peut également être exploitée avec un *for*:

```
1.   for (Object o : L) {  
2.       // exploiter objet o  
3.   }
```

- 2 - la méthode **getSingleResult** exécute un ordre JPQL / SQL SELECT qui ramène un unique objet.
- 3 - la méthode **executeUpdate** exécute un ordre SQL **update** ou **delete** et rend le nombre de lignes affectées l'opération.
- 4 - la méthode **setParameter(String, Object)** permet de donner une valeur à un paramètre nommé d'un ordre JPQL paramétré
- 5 - la méthode **setParameter(int, Object)** mais le paramètre n'est pas désigné par son nom mais par sa position dans l'ordre JPQL.

## 2.1.10 Un premier client JPA

Revenons dans une perspective Java du projet :



Nous connaissons maintenant à peu près tout de ce projet sauf le contenu du dossier [src/tests] que nous examinons maintenant. Le dossier contient deux programmes de test de la couche JPA :

- [InitDB.java] est un programme qui met quelques lignes dans la table [jpa01\_personne] de la base. Son code va nous donner les premiers éléments de la couche JPA.
- [Main.java] est un programme qui fait les opérations CRUD sur la table [jpa01\_personne]. L'étude de son code va nous permettre d'aborder les concepts fondamentaux du contexte de persistance et du cycle de vie des objets de ce contexte.

### 2.1.10.1 Le code

Le code du programme [InitDB.java] est le suivant :

```
1. package tests;  
2.  
3. import java.text.ParseException;  
4. import java.text.SimpleDateFormat;  
5.  
6. import javax.persistence.EntityManager;  
7. import javax.persistence.EntityManagerFactory;  
8. import javax.persistence.EntityTransaction;  
9. import javax.persistence.Persistence;  
10.  
11. import entites.Personne;  
12.  
13. public class InitDB {  
14.     // constantes  
15.     private final static String TABLE_NAME = "jpa01_personne";  
16.  
17.     public static void main(String[] args) throws ParseException {  
18.         // Unité de persistance  
19.         EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");  
20.         // récupérer un EntityManagerFactory à partir de l'unité de persistance  
21.         EntityManager em = emf.createEntityManager();  
22.         // début transaction  
23.         EntityTransaction tx = em.getTransaction();  
24.         tx.begin();  
25.         // supprimer les éléments de la table des personnes  
26.         em.createNativeQuery("delete from " + TABLE_NAME).executeUpdate();
```

```

27.    // créer deux personnes
28.    Personne p1 = new Personne("Martin", "Paul", new
      SimpleDateFormat("dd/MM/yy").parse("31/01/2000"), true, 2);
29.    Personne p2 = new Personne("Durant", "Sylvie", new
      SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
30.    // persistance des personnes
31.    em.persist(p1);
32.    em.persist(p2);
33.    // affichage personnes
34.    System.out.println("[personnes]");
35.    for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList())
{
36.        System.out.println(p);
37.    }
38.    // fin transaction
39.    tx.commit();
40.    // fin EntityManager
41.    em.close();
42.    // fin EntityManagerFactory
43.    emf.close();
44.    // log
45.    System.out.println("terminé ...");
46. }
47. }
```

Il faut lire ce code à la lumière de ce qui a été expliqué au paragraphe 2.1.9, page 26.

- ligne 19 : on demande un objet **EntityManagerFactory** emf pour l'unité de persistance **jpa** (définie dans *persistence.xml*). Cette opération n'est faite normalement qu'une fois dans la vie d'une application.
- ligne 21 : on demande un objet **EntityManager** em pour gérer un contexte de persistance.
- ligne 23 : on demande un objet **Transaction** pour gérer une transaction. On rappelle ici que les opérations sur le contexte de persistance se font à l'intérieur d'une transaction. On verra que ce n'est pas obligatoire mais qu'alors on peut rencontrer des problèmes. Si l'application s'exécute dans un conteneur EJB3, alors les opérations sur le contexte de persistance se font toujours à l'intérieur d'une transaction.
- ligne 24 : la transaction commence
- ligne 26 : exécute un ordre SQL **delete** sur la table " jpa01\_personne " (nativeQuery). On fait cela pour vider la table de tout contenu et ainsi mieux voir le résultat de l'exécution de l'application [InitDB]
- lignes 28-29 : deux objets *Personne* p1 et p2 sont créés. Ce sont des objets normaux et n'ont pour l'instant rien à voir avec le contexte de persistance. Vis à vis du contexte de persistance, Hibernate dit que ces objets sont dans un état **passager** (transient) pour les opposer aux objets **persistants** (persistent) qui sont gérés par le contexte de persistance. Nous parlerons plutôt d'objets **non persistants** (expression non française) pour indiquer qu'ils ne sont pas encore gérés par le contexte de persistance et d'objets **persistants** pour ceux qui sont gérés par celui-ci. Nous trouverons une troisième catégorie d'objets, des objets **détachés** (detached) qui sont des objets précédemment persistants mais dont le contexte de persistance a été fermé. Le client peut détenir des références sur de tels objets, ce qui explique qu'ils ne sont pas nécessairement détruits à la fermeture du contexte de persistance. On dit alors qu'ils sont dans état **détaché**. L'opération **[EntityManager].merge** permet de les réattacher à un contexte de persistance nouvellement créé.
- lignes 31-32 : les personnes p1 et p2 sont intégrés au contexte de persistance par l'opération **[EntityManager].persist**. Ils deviennent alors des objets persistants.
- lignes 35-37 : on exécute un ordre JPQL " select p from Personne p order by p.nom asc ". Personne n'est pas la table (elle s'appelle jpa01\_personne) mais l'objet **@Entity** associé à la table. On a ici une requête JPQL (Java Persistence Query Language) sur le contexte de persistance et non un ordre SQL sur la base de données. Ceci dit, en-dehors de l'objet Personne qui a remplacé la table jpa01\_personne, les syntaxes sont identiques. Une boucle for parcourt la liste (de personnes) résultat du select pour en afficher chaque élément sur la console. On cherche à vérifier ici qu'on retrouve bien dans la table les éléments mis dans le contexte de persistance lignes 31-32. De façon transparente, une synchronisation du contexte de persistance avec la base va avoir lieu. En effet, une requête select va être émise et on a dit que c'était l'un des cas où était faite une synchronisation. C'est donc à ce moment, qu'en arrière-plan, JPA / Hibernate va émettre les deux ordres SQL insert qui vont insérer les deux personnes dans la table jpa01\_personne. L'opération persist ne l'avait pas fait. Cette opération intègre des objets dans le contexte de persistance sans que ça ait une conséquence sur la base. Les choses réelles se font aux synchronisations, ici juste avant le select sur la base.
- ligne 39 : on termine la transaction commencée ligne 24. Une synchronisation va de nouveau avoir lieu. Rien ne se passera ici puisque le contexte de persistance n'a pas changé depuis le dernière synchronisation.
- ligne 41 : on ferme le contexte de persistance.
- ligne 43 : on ferme la fabrique d'*EntityManager*.

### 2.1.10.2 L'exécution du code

- lancer le SGBD MySQL5
- mettre conf/mysql5/persistence.xml dans META-INF/persistence.xml si besoin est
- exécuter l'application [InitDB]

On obtient les résultats suivants :

- en [1] : l'affichage console dans la perspective Java. On obtient ce qui était attendu.
- en [2] : on vérifie le contenu de la table [jpa01\_personne] avec la perspective SQL Explorer tel qu'il a été expliqué au paragraphe 2.1.8, page 25. On peut remarquer deux points :
  - la clé primaire ID a été générée sans qu'on s'en occupe
  - idem pour le n° de version. On constate que la première version a le n° 0..

Nous avons là, les premiers éléments de la culture JPA. Nous avons réussi à insérer des données dans une table. Nous allons construire sur ces acquis pour écrire le second test mais auparavant parlons de logs.

## 2.1.11 Mettre en oeuvre les logs d'Hibernate

Il est possible de connaître les ordres SQL émis sur la base par la couche JPA / Hibernate. Il est intéressant de des connaître pour voir si la couche JPA est aussi efficace qu'un développeur qui aurait écrit lui-même les ordres SQL.

Avec JPA / Hibernate, les logs SQL peuvent être contrôlés dans le fichier [persistence.xml] :

```

1.      <!-- Classes persistantes -->
2.      <property name="hibernate.archive.autodetection" value="class, hbm" />
3.      <!-- logs SQL
4.          <property name="hibernate.show_sql" value="true"/>
5.          <property name="hibernate.format_sql" value="true"/>
6.          <property name="use_sql_comments" value="true"/>
7.      -->
8.      <!-- connexion JDBC -->
9.      <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
10.

```

- lignes 4-6 : les logs SQL n'étaient pour l'instant pas activés. On les active désormais en enlevant la balise des commentaires des lignes 3 et 7.

On réexécute l'application [InitDB]. Les affichages console deviennent alors les suivants :

```

1. Hibernate:
2.     delete
3.     from
4.         jpa01_personne
5. Hibernate:
6.     insert
7.     into
8.         jpa01_personne
9.         (VERSION, NOM, PRENOM, DATENAISANCE, MARIE, NBENFANTS)
10.    values
11.        (?, ?, ?, ?, ?, ?)
12. Hibernate:
13.     insert
14.     into
15.         jpa01_personne
16.         (VERSION, NOM, PRENOM, DATENAISANCE, MARIE, NBENFANTS)
17.     values
18.        (?, ?, ?, ?, ?, ?)
19. [personnes]
20. Hibernate:
21.     select
22.         personne0_.ID as ID0_,
23.         personne0_.VERSION as VERSION0_,
24.         personne0_.NOM as NOM0_,
25.         personne0_.PRENOM as PRENOM0_,
26.         personne0_.DATENAISANCE as DATENAISSE0_,
27.         personne0_.MARIE as MARIE0_,
28.         personne0_.NBENFANTS as NBENFANTS0_
29.     from
30.         jpa01_personne personne0_
31.     order by
32.         personne0_.NOM asc

```

```

33. [2,0,Durant,Sylvie,05/07/2001,false,0]
34. [1,0,Martin,Paul,31/01/2000,true,2]
35. terminé ...

```

- lignes 2-4 : l'ordre SQL *delete* issu de l'instruction :

```
// supprimer les éléments de la table des personnes
em.createNativeQuery("delete from " + TABLE_NAME).executeUpdate();
```

- lignes 5-18 : les ordres SQL *insert* issus des instructions :

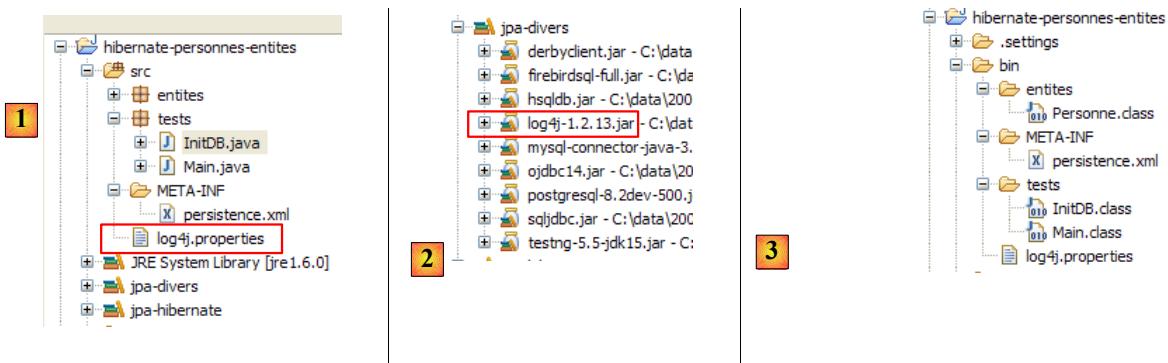
```
// persistance des personnes
em.persist(p1);
em.persist(p2);
```

- lignes 21-32 : l'ordre SQL *select* issu de l'instruction :

```
for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList())
```

Si on fait des affichages console intermédiaires, on verra que l'écriture des logs SQL d'une instruction I du code Java se fait lorsque que l'instruction I est exécutée. Cela ne veut pas dire que l'ordre SQL affiché est exécuté sur la base à ce moment là. Il est en fait mis en cache pour exécution lors de la prochaine synchronisation du contexte de persistance avec la base.

D'autres logs peuvent être obtenus via le fichier [src/log4j.properties] :



- en [1], le fichier [log4j.properties] est exploité par l'archive [log4j-1.2.13.jar] [2] de l'outil appelé LOG4j (Logs for Java) disponible à l'url [<http://logging.apache.org/log4j/docs/index.html>]. Placé dans le dossier [src] du projet Eclipse, nous savons que [log4j.properties] sera recopié automatiquement dans le dossier [bin] du projet [3]. Ceci fait, il est désormais dans le classpath du projet et c'est là que l'archive [2] ira le chercher.

Le fichier [log4j.properties] nous permet de contrôler certains logs d'Hibernate. Lors des exécutions précédentes son contenu était le suivant :

```

1. # Direct log messages to stdout
2. log4j.appenders.stdout=org.apache.log4j.ConsoleAppender
3. log4j.appenders.stdout.Target=System.out
4. log4j.appenders.stdout.layout=org.apache.log4j.PatternLayout
5. log4j.appenders.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
6.
7. # Root logger option
8. log4j.rootLogger=ERROR, stdout
9.
10. # Hibernate logging options (INFO only shows startup messages)
11. #log4j.logger.org.hibernate=INFO
12.
13. # Log JDBC bind parameter runtime arguments
14. #log4j.logger.org.hibernate.type=DEBUG

```

Je commenterai peu cette configuration n'ayant jamais pris le temps de m'informer sérieusement sur LOG4j.

- les lignes 1-8 se retrouvent dans tous les fichiers *log4j.properties* que j'ai pu rencontrer
- les lignes 10-14 sont présentes dans les fichiers *log4j.properties* des exemples d'Hibernate.
- ligne 11 : contrôle les logs généraux d'Hibernate. La ligne étant commentée, ces logs sont ici inhibés. On peut avoir plusieurs niveaux de logs : INFO (informations générales sur ce que fait Hibernate), WARN (Hibernate nous avertit d'un possible problème), DEBUG (logs détaillés). Le niveau INFO est le moins verbeux, le mode DEBUG le plus verbeux. Activer la ligne 11 permet de savoir ce que fait Hibernate, notamment au démarrage de l'application. C'est souvent intéressant.

- la ligne 12, si elle est active, permet de connaître les arguments effectivement utilisés lors de l'exécution des requêtes SQL paramétrées.

Commençons par décommenter la ligne 14

```
# Log JDBC bind parameter runtime arguments
|log4j.logger.org.hibernate.type=DEBUG
```

et réexécutons [InitDB]. Les nouveaux logs amenés par cette modification sont les suivants (vue partielle) :

```
1. Hibernate:
2.     insert
3.     into
4.         jpa01_personne
5.             (VERSION, NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS)
6.     values
7.         (?, ?, ?, ?, ?, ?)
8. 07:20:03,843 DEBUG IntegerType:80 - binding '0' to parameter: 1
9. 07:20:03,843 DEBUG StringType:80 - binding 'Durant' to parameter: 2
10. 07:20:03,843 DEBUG StringType:80 - binding 'Sylvie' to parameter: 3
11. 07:20:03,843 DEBUG DateType:80 - binding '05 juillet 2001' to parameter: 4
12. 07:20:03,843 DEBUG BooleanType:80 - binding 'false' to parameter: 5
13. 07:20:03,843 DEBUG IntegerType:80 - binding '0' to parameter: 6
```

- les lignes 8-10 sont de nouveaux logs amenés par l'activation de la ligne 14 de [log4j.properties]. Ils indiquent les 5 valeurs affectés aux paramètres formels ? de la requête paramétrée des lignes 2-7. Ainsi on voit que la colonne VERSION va recevoir la valeur 0 (ligne 8).

Maintenant activons la ligne 11 de [log4j.properties] :

```
# Hibernate logging options (INFO only shows startup messages)
|log4j.logger.org.hibernate=INFO
```

et réexécutons [InitDB] :

```
1. 07:50:23,937 INFO Version:15 - Hibernate EntityManager 3.2.0.CR3
2. 07:50:23,968 INFO Version:15 - Hibernate Annotations 3.2.0.CR3
3. 07:50:23,984 INFO Environment:500 - Hibernate 3.2.0.cr5
4. 07:50:23,984 INFO Environment:533 - hibernate.properties not found
5. 07:50:23,984 INFO Environment:667 - Bytecode provider name : cglib
6. 07:50:24,000 INFO Environment:584 - using JDK 1.4 java.sql.Timestamp handling
7. 07:50:24,375 INFO AnnotationBinder:387 - Binding entity from annotated class: entites.Personne
8. 07:50:24,421 INFO EntityBinder:340 - Bind entity entites.Personne on table jpa01_personne
9. 07:50:24,609 INFO C3P0ConnectionProvider:50 - C3P0 using driver: com.mysql.jdbc.Driver at URL:
jdbc:mysql://localhost:3306/jpa
10. 07:50:24,609 INFO C3P0ConnectionProvider:51 - Connection properties: {user=jpa, password=****,
autocommit=true, release_mode=auto}
11. 07:50:24,609 INFO C3P0ConnectionProvider:54 - autocommit mode: true
12. 07:50:25,296 INFO SettingsFactory:81 - RDBMS: MySQL, version: 5.0.37-community-nt
13. 07:50:25,296 INFO SettingsFactory:82 - JDBC driver: MySQL-AB JDBC Driver, version: mysql-
connector-java-3.1.9 ( $Date: 2005/05/19 15:52:23 $, $Revision: 1.1.2.2 $ )
14. 07:50:25,312 INFO Dialect:141 - Using dialect: org.hibernate.dialect.MySQLInnoDBDialect
15. 07:50:25,312 INFO TransactionFactoryFactory:34 - Transaction strategy:
org.hibernate.transaction.JDBCTransactionFactory
16. 07:50:25,312 INFO TransactionManagerLookupFactory:33 - No TransactionManagerLookup configured (in
JTA environment, use of read-write or transactional second-level cache is not recommended)
17. 07:50:25,328 INFO SettingsFactory:134 - Automatic flush during beforeCompletion(): disabled
18. 07:50:25,328 INFO SettingsFactory:138 - Automatic session close at end of transaction: disabled
19. 07:50:25,328 INFO SettingsFactory:145 - JDBC batch size: 15
20. 07:50:25,328 INFO SettingsFactory:148 - JDBC batch updates for versioned data: disabled
21. 07:50:25,328 INFO SettingsFactory:153 - Scrollable result sets: enabled
22. 07:50:25,328 INFO SettingsFactory:161 - JDBC3 getGeneratedKeys(): enabled
23. 07:50:25,328 INFO SettingsFactory:169 - Connection release mode: auto
24. 07:50:25,328 INFO SettingsFactory:193 - Maximum outer join fetch depth: 2
25. 07:50:25,328 INFO SettingsFactory:196 - Default batch fetch size: 1
26. 07:50:25,328 INFO SettingsFactory:200 - Generate SQL with comments: disabled
27. 07:50:25,328 INFO SettingsFactory:204 - Order SQL updates by primary key: disabled
28. 07:50:25,328 INFO SettingsFactory:369 - Query translator:
org.hibernate.hql.ast.ASTQueryTranslatorFactory
29. 07:50:25,328 INFO ASTQueryTranslatorFactory:24 - Using ASTQueryTranslatorFactory
30. 07:50:25,328 INFO SettingsFactory:212 - Query language substitutions: {}
31. 07:50:25,328 INFO SettingsFactory:217 - JPA-QL strict compliance: enabled
32. 07:50:25,328 INFO SettingsFactory:222 - Second-level cache: enabled
33. 07:50:25,328 INFO SettingsFactory:226 - Query cache: disabled
34. 07:50:25,328 INFO SettingsFactory:356 - Cache provider: org.hibernate.cache.NoCacheProvider
35. 07:50:25,328 INFO SettingsFactory:241 - Optimize cache for minimal puts: disabled
36. 07:50:25,328 INFO SettingsFactory:250 - Structured second-level cache entries: disabled
37. 07:50:25,343 INFO SettingsFactory:270 - Echoing all SQL to stdout
38. 07:50:25,343 INFO SettingsFactory:277 - Statistics: disabled
39. 07:50:25,343 INFO SettingsFactory:281 - Deleted entity synthetic identifier rollback: disabled
40. 07:50:25,343 INFO SettingsFactory:296 - Default entity-mode: pojo
```

```

41. 07:50:25,468  INFO SessionFactoryImpl:161 - building session factory
42. 07:50:25,750  INFO SessionFactoryObjectFactory:82 - Not binding factory to JNDI, no JNDI name
configured
43. 07:50:25,765  INFO SchemaExport:154 - Running hbm2ddl schema export
44. 07:50:25,765  INFO SchemaExport:179 - exporting generated schema to database
45. 07:50:25,968  INFO SchemaExport:196 - schema export complete
46. Hibernate:
47.     delete
48.     from
49.         jpa01_personne
50. Hibernate:
51. ...

```

La lecture de ces logs apporte beaucoup d'informations intéressantes :

- ligne 7 : Hibernate indique le nom d'une classe `@Entity` qu'il a trouvée
- ligne 8 : indique que la classe [Personne] va être liée à la table [jpa01\_personne]
- ligne 9 : indique le pool de connexions C3P0 qui va être utilisé, le nom du pilote Jdbc, l'url de la base de données à gérer
- ligne 10 : donne d'autres caractéristiques de la liaison Jdbc : propriétaire, type du commit, ...
- ligne 14 : le dialecte utilisé pour dialoguer avec le SGBD
- ligne 15 : le type de transaction utilisée. `JDBCTransactionFactory` indique que l'application gère elle-même ses transactions. Elle ne s'exécute pas dans un conteneur EJB3 qui fournirait son propre service de transactions.
- les lignes suivantes se rapportent à des options de configuration d'Hibernate que nous n'avons pas rencontrées. Le lecteur intéressé est invité à lire la documentation d'Hibernate.
- ligne 37 : les ordres SQL vont être affichés sur la console. Cela a été demandé dans [persistence.xml] :

```

<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />
<property name="use_sql_comments" value="true" />

```

- lignes 43-45 : le schéma de la base de données est exporté vers le SGBD, c.a.d. que la base de données est vidée puis recréée. Ce mécanisme vient de la configuration faite dans [persistence.xml] (ligne 4 ci-dessous) :

```

1. ...
2. <property name="hibernate.connection.password" value="jpa" />
3. <!-- création automatique du schéma -->
4. <property name="hibernate.hbm2ddl.auto" value="create" />
5. <!-- Dialecte -->
6. ...

```

Lorsqu'une application "plante" avec une exception Hibernate qu'on ne comprend pas, on commencera par activer les logs d'Hibernate en mode DEBUG dans [log4j.properties] pour y voir plus clair :

```

1. # Root logger option
2. log4j.rootLogger=ERROR, stdout
3.
4. # Hibernate logging options (INFO only shows startup messages)
5. log4j.logger.org.hibernate=DEBUG

```

Dans la suite de ce document, les logs sont inhibés par défaut afin d'avoir un affichage console plus lisible.

## 2.1.12 Découvrir le langage JPQL / HQL avec la console Hibernate

**Note :** Cette section nécessite le plugin Hibernate Tools (paragraphe 5.2.5, page 218).

Dans le code de l'application [InitDB], nous avons utilisé une requête JPQL. JPQL (Java Persistence Query Language) est un langage pour requêter le contexte de persistance. La requête rencontrée était la suivante :

```
select p from Personne p order by p.nom asc
```

Elle sélectionnait tous les éléments de la table associée à l'`@Entity` [Personne] et les rendait par ordre croissant du nom. Dans la requête ci-dessus, `p.nom` est le champ nom d'une instance `p` de la classe [Personne]. Une requête JPQL travaille donc sur les objets `@Entity` du contexte de persistance et non directement sur les tables de la base. La couche JPA va elle traduire cette requête JPQL en une requête SQL appropriée au SGBD avec lequel elle travaille. Ainsi dans le cas d'une implémentation JPA / Hibernate reliée à un SGBD MySQL5, la requête JPQL précédente est traduite en la requête SQL suivante :

```

select
    personne0_.ID as ID0_,
    personne0_.VERSION as VERSION0_,
    personne0_.NOM as NOM0,
    personne0_.PRENOM as PRENOM0,
    personne0_.DATENAISSE as DATENAIS5_0,

```

```

personne0_.MARIE as MARIE0_,
personne0_.NBENFANTS as NBENFANTSO_
from
jpa01_personne personne0_
order by
personne0_.NOM asc

```

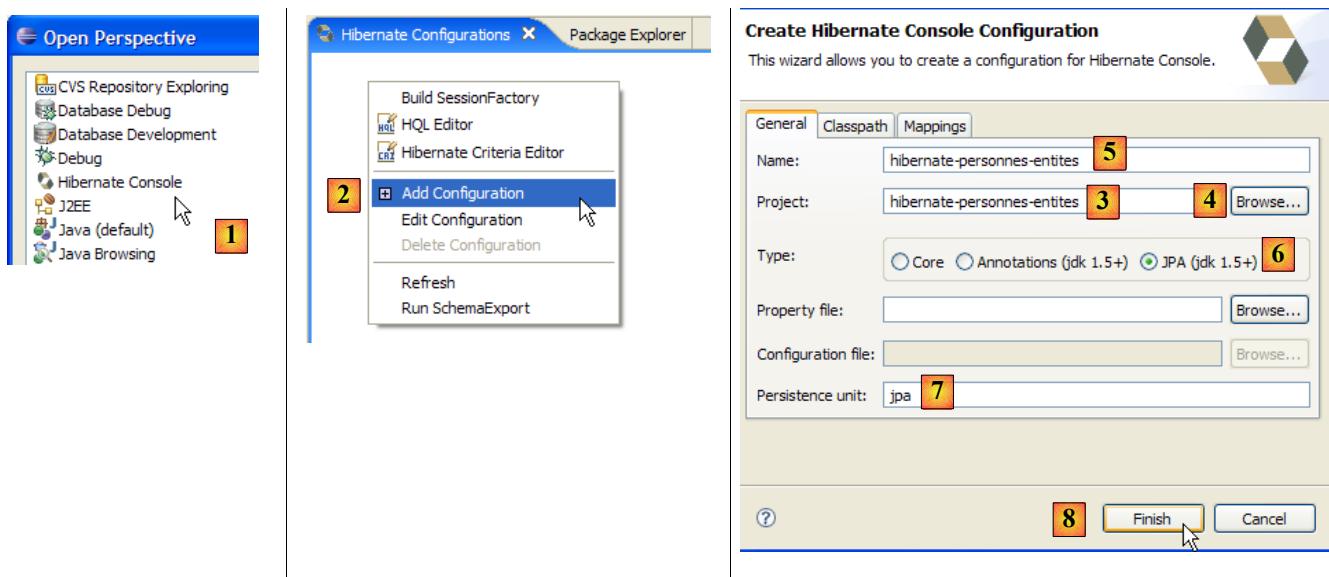
La couche JPA a utilisé la configuration de l'objet @Entity [Personne] pour générer l'ordre SQL correct. C'est le pont objet / relationnel qui a été mis en oeuvre ici.

Le plugin [Hibernate Tools] (paragraphe 5.2.5, page 218) offre un outil appelé "Console Hibernate" qui permet

- d'émettre des ordres JPQL ou du sur-ensemble HQL (Hibernate Query Language) sur le contexte de persistance
- d'en obtenir les résultats
- de connaître l'équivalent SQL qui a été exécuté sur la base

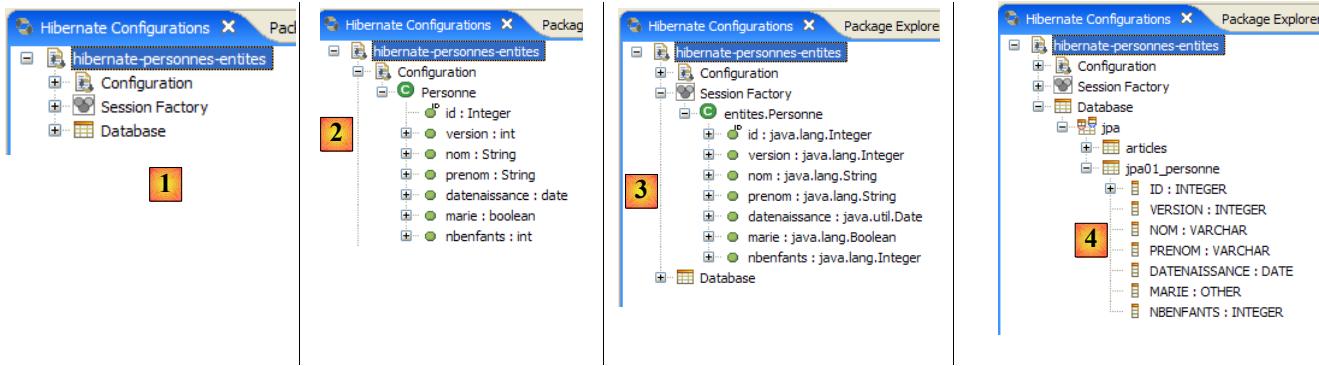
La console Hibernate est un outil de première valeur pour apprendre le langage JPQL et se familiariser au pont JPQL / SQL. On sait que JPA s'est fortement inspiré d'outils ORM comme Hibernate ou Toplink. JPQL est très proche du langage HQL d'Hibernate mais ne reprend pas toutes ses fonctionnalités. Dans la console Hibernate, on peut émettre des ordres HQL qui seront exécutés normalement dans la console mais qui ne font pas partie du langage JPQL et qu'on ne pourrait donc utiliser dans un client JPA. Lorsque ce sera le cas, nous le signalerons.

Créons une console Hibernate pour notre projet Eclipse actuel :

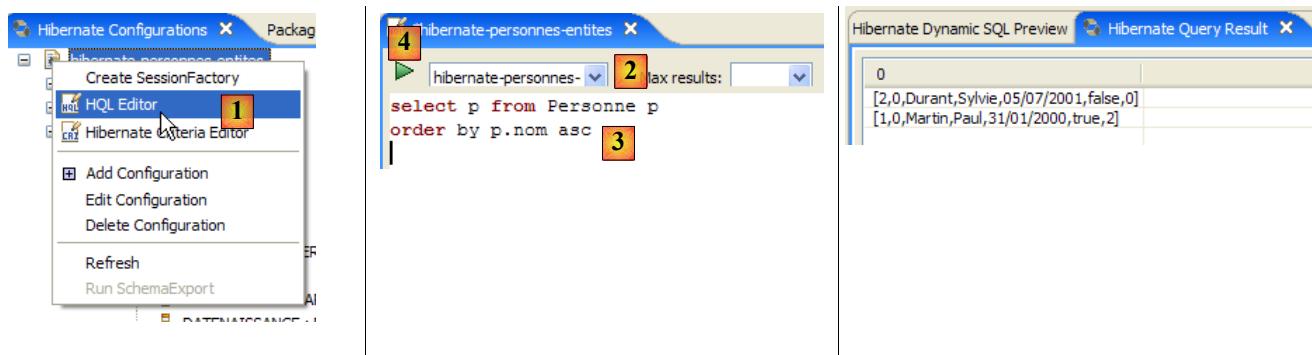


- [1] : nous passons dans une perspective [Hibernate Console] (Window / Open Perspective / Other)
- [2] : nous créons une nouvelle configuration dans la fenêtre [Hibernate Configuration]
- à l'aide du bouton [4], nous sélectionnons le projet Java pour lequel est créée la configuration Hibernate. Son nom s'affiche dans [3].
- en [5], nous donnons le nom que nous voulons à cette configuration. Ici, nous avons repris [3].
- en [6], nous indiquons que nous utilisons une configuration JPA afin que l'outil sache qu'il doit exploiter le fichier [META-INF/persistence.xml]
- en [7] : nous indiquons que dans ce fichier [META-INF/persistence.xml], il faut utiliser l'unité de persistance qui s'appelle **jpa**.
- en [8], on valide la configuration.

Pour la suite, il faut que le SGBD soit lancé. Ici, il s'agit de MySQL5.



- en [1] : la configuration créée présente une arborescence à trois branches
- en [2] : la branche [Configuration] liste les objets que la console a utilisés pour se configurer : ici l'@Entity Personne.
- en [3] : la Session Factory est une notion Hibernate proche de l'EntityManager de JPA. Elle réalise le pont objet / relationnel grâce aux objets de la branche [Configuration]. En [3] sont présentés les objets du contexte de persistance, ici de nouveau l'@Entity Personne.
- en [4] : la base de données accédée au moyen de la configuration trouvée dans [persistence.xml]. On y retrouve la table [jpa01\_personne].



- en [1], on crée un éditeur HQL
- dans l'éditeur HQL,
  - en [2], on choisit la configuration Hibernate à utiliser s'il y en a plusieurs
  - en [3], on tape la commande JPQL qu'on veut exécuter
  - en [4], on l'exécute
- en [5], on obtient les résultats de la requête dans la fenêtre [Hibernate Query Result]. On peut rencontrer deux difficultés ici :
  - on n'obtient rien (aucune ligne). La console Hibernate a utilisé le contenu de [persistence.xml] pour créer une connexion avec le SGBD. Or cette configuration a une propriété qui dit de vider la base de données :

```
<property name="hibernate.hbm2ddl.auto" value="create" />
```

- Il faut donc réexécuter l'application [InitDB] avant de rejouer la commande JPQL ci-dessus.
- on n'a pas la fenêtre [Hibernate Query Result]. On la demande par [Window / Show View / ...]

La fenêtre [Hibernate Dynamic SQL preview] ([1] ci-dessous) permet de voir la requête SQL qui va être jouée pour exécuter la commande JPQL qu'on est en train d'écrire. Dès que la syntaxe de la commande JPQL est correcte, la commande SQL correspondante apparaît dans cette fenêtre :

The screenshot shows five windows illustrating the execution of an HQL query:

- 1**: A screenshot of the "Hibernate Dynamic SQL Preview" window showing the raw SQL query: `SQL #0 types: entites.Personne` followed by the full HQL command: `select personne0_.ID as ID0_, personne0_.VERSION as VERSION0_, personne0_.NOM as NOM0_, personne0_.PRENOM as PRENOM0_, personne0_.DATENAISANCE as DATENAIS5_0_, personne0_.MARIE as MARIE0_, personne0_.NBENFANTS as NBENFANTS0_ from jpa01_personne personne0_ order by personne0_.NOM asc`.
- 2**: A screenshot of the "hibernate-personnes-entites" window showing the HQL command: `select p from Personne p order by p.nom asc`. The "Clear HQL Editor" button is highlighted.
- 3**: A screenshot of the "hibernate-personnes-entites" window showing the HQL command: `select p.prenom, p.nom from Personne p where p.marie=true and p.nbenfants>1`.
- 4**: A screenshot of the "Hibernate Query Result" window showing the result of the query: 

0	1
Paul	Martin

.
- 5**: A screenshot of the "Hibernate Dynamic SQL Preview" window showing the raw SQL command generated by the HQL: `SQL #0 types: string, string` followed by the generated SQL: `select personne0_.PRENOM as col_0_0_, personne0_.NOM as col_1_0_ from jpa01_personne personne0_ where personne0_.MARIE=1 and personne0_.NBENFANTS>1`.

- en [2], on efface la précédente commande HQL
- en [3], on exécute une nouvelle
- en [4], le résultat
- en [5], la commande SQL qui a été exécutée sur la base

L'éditeur HQL offre une aide à l'écriture des commandes HQL :

The screenshot shows three windows illustrating the use of the HQL editor:

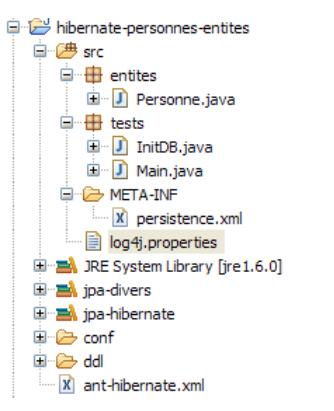
- 1**: A screenshot of the "hibernate-personnes-entites" window showing the HQL command: `select p from Personne p where p.`. A tooltip appears below the cursor, listing available fields: `datenaissance - Personne`, `id - Personne`, `marie - Personne`, `nbenfants - Personne`, `nom - Personne`, `prenom - Personne`, and `version - Personne`.
- 2**: A screenshot of the "hibernate-personnes-entites" window showing the HQL command: `select p from Personne p where p.marie and p.nbenfants=1`. The "Console" tab is selected at the bottom.
- 3**: A screenshot of the "Hibernate Dynamic SQL Preview" window showing the error message: `org.hibernate.hql.ast.QuerySyntaxException: unexpected AST node: . near line 2, column 8 [select p from entites.Personne p where p.marie and p.nbenfants=1]`.

- en [1] : une fois que l'éditeur sait que `p` est un objet `Personne`, il peut nous proposer les champs de `p` lors de la frappe.
- en [2] : une commande HQL incorrecte. Il faut écrire `where p.marie=true`.
- en [3] : l'erreur est signalée dans la fenêtre [SQL Preview]

Nous invitons le lecteur à émettre d'autres commandes HQL / JPQL sur la base.

## 2.1.13 Un second client JPA

Revenons dans une perspective Java du projet :



- [InitDB.java] est un programme qui mettait quelques lignes dans la table [jpa01\_personne] de la base. L'étude de son code nous a permis d'acquérir les premiers éléments de l'API JPA.

- [Main.java] est un programme qui fait les opérations CRUD sur la table [jpa01\_personne]. L'étude se son code va nous permettre de revenir sur les concepts fondamentaux du contexte de persistance et du cycle de vie des objets de ce contexte.

### 2.1.13.1 La structure du code

[Main.java] va enchaîner une série de tests où chacun vise à montrer une facette particulière de JPA :



La méthode [main]

- appelle successivement les méthodes *test1* à *test11*. Nous présenterons séparément le code de chacune de ces méthodes.
- utilise par ailleurs des méthodes utilitaires privées : *clean*, *dump*, *log*, *getEntityManager*, *getNewEntityManager*.

Nous présentons la méthode *main* et les méthodes dites utilitaires :

```

1. package tests;
2.
3. ...
4. import entites.Personne;
5.
6. @SuppressWarnings("unchecked")
7. public class Main {
8.
9.     // constantes
10.    private final static String TABLE_NAME = "jpa01_personne";
11.
12.    // Contexte de persistance
13.    private static EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
14.    private static EntityManager em = null;
15.
16.    // objets partagés
17.    private static Personne p1, p2, newp1;
18.
19.    public static void main(String[] args) throws Exception {
20.        // nettoyage base
21.        log("clean"); clean();
22.
23.        // dump table
24.        dump();
25.
26.        // test1
27.        log("test1"); test1();
28.
29.    ... // test11
30.    log("test11"); test11();
31.
32.    // fin contexte de persistance
33.    if (em.isOpen())
34.        em.close();
35.
36.

```

```

37.     // fermeture EntityManagerFactory
38.     emf.close();
39. }
40.
41. // récupérer l'EntityManager courant
42. private static EntityManager getEntityManager() {
43.     if (em == null || !em.isOpen()) {
44.         em = emf.createEntityManager();
45.     }
46.     return em;
47. }
48.
49. // récupérer un EntityManager neuf
50. private static EntityManager getNewEntityManager() {
51.     if (em != null && em.isOpen()) {
52.         em.close();
53.     }
54.     em = emf.createEntityManager();
55.     return em;
56. }
57.
58. // affichage contenu table
59. private static void dump() {
60.     // contexte de persistance courant
61.     EntityManager em = getEntityManager();
62.     // début transaction
63.     EntityTransaction tx = em.getTransaction();
64.     tx.begin();
65.     // affichage personnes
66.     System.out.println("[personnes]");
67.     for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList())
68.     {
69.         System.out.println(p);
70.     }
71.     // fin transaction
72.     tx.commit();
73. }
74.
75. // raz BD
76. private static void clean() {
77.     // contexte de persistance
78.     EntityManager em = getEntityManager();
79.     // début transaction
80.     EntityTransaction tx = em.getTransaction();
81.     tx.begin();
82.     // supprimer les éléments de la table PERSONNES
83.     em.createNativeQuery("delete from " + TABLE_NAME).executeUpdate();
84.     // fin transaction
85.     tx.commit();
86.
87. // logs
88. private static void log(String message) {
89.     System.out.println("main : ----- " + message);
90. }
91.
92. // création d'objets
93. public static void test1() throws ParseException {
94. ...
95. }
96.
97. // modifier un objet du contexte
98. public static void test2() {
99. ...
100. }
101.
102. // demander des objets
103. public static void test3() {
104. ...
105. }
106.
107. // supprimer un objet appartenant au contexte de persistance
108. public static void test4() {
109. ...
110. }
111.
112. // détacher, réattacher et modifier
113. public static void test5() {
114. ...
115. }
116.
117. // supprimer un objet n'appartenant pas au contexte de persistance
118. public static void test6() {
119. ...
120. }

```

```

121.
122. // modifier un objet n'appartenant pas au contexte de persistance
123. public static void test7() {
124. ...
125. }
126.
127. // réattacher un objet au contexte de persistance
128. public static void test8() {
129. ...
130. }
131.
132. // une requête select provoque une synchronisation
133. // de la base avec le contexte de persistance
134. public static void test9() {
135. ...
136. }
137.
138. // contrôle de version (optimistic locking)
139. public static void test10() {
140. ...
141. }
142.
143. // rollback d'une transaction
144. public static void test11() throws ParseException {
145. ...
146. }
147.
148. }
```

- ligne 13 : l'objet *EntityManagerFactory* **emf** construit à partir de l'unité de persistance **jpa** définie dans [persistence.xml]. Il va nous permettre de créer au fil de l'application divers contextes de persistance.
- ligne 14 : un contexte de persistance *EntityManager* **em** encore non initialisé
- ligne 17 : trois objets [Personne] partagés par les tests
- ligne 21 : la table *jpa01\_personne* est vidée puis affichée ligne 24 pour s'assurer qu'on part d'une table vide.
- lignes 27-31 : enchaînement des tests
- lignes 34-35 : fermeture du contexte de persistance **em** s'il était ouvert.
- ligne 38 : fermeture de l'objet *EntityManagerFactory* **emf**.
- lignes 42-47 : la méthode [*getEntityManager*] rend l'*EntityManager* (ou contexte de persistance) courant ou neuf s'il n'existe pas (lignes 43-44).
- lignes 50-56 : la méthode [*getNewEntityManager*] rend un contexte de persistance neuf. S'il en existait un auparavant, il est fermé (lignes 51-52)
- lignes 59-72 : la méthode [*dump*] affiche le contenu de la table [*jpa01\_personne*]. Ce code a déjà été rencontré dans [InitDB].
- lignes 75-85 : la méthode [*clean*] vide la table [*jpa01\_personne*]. Ce code a déjà été rencontré dans [InitDB].
- lignes 88-90 : la méthode [*log*] affiche sur la console le message qu'on lui passe en paramètre de façon à ce qu'il soit remarqué.

Nous pouvons maintenant passer à l'étude des tests.

### 2.1.13.2 Test 1

Le code de **test1** est le suivant :

```

1.// création d'objets
2.public static void test1() throws ParseException {
3.    // contexte de persistance
4.    EntityManager em = getEntityManager();
5.    // création personnes
6.    p1 = new Personne("Martin", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"), true, 2);
7.    p2 = new Personne("Durant", "Sylvie", new SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false,
   0);
8.    // début transaction
9.    EntityTransaction tx = em.getTransaction();
10.   tx.begin();
11.   // persistance des personnes
12.   em.persist(p1);
13.   em.persist(p2);
14.   // fin transaction
15.   tx.commit();
16.   // on affiche la table
17.   dump();
18.
19.}
```

Ce code a déjà été rencontré dans [InitDB] : il crée deux personnes et les place dans le contexte de persistance.

- ligne 4 : on demande le contexte de persistance courant
- lignes 6-7 : on crée les deux personnes
- lignes 9-15 : les deux personnes sont placées dans le contexte de persistance à l'intérieur d'une transaction.
- ligne 15 : à cause du commit de la transaction, il y a synchronisation du contexte de persistance avec la base. Les deux personnes vont être ajoutées à la table [jpa01\_personne].
- ligne 17 : on affiche la table

L'affichage console de ce premier test est le suivant :

```
1 main : -----
2 [personnes]
3 [2,0,Durant,Sylvie,05/07/2001,false,0]
4 [1,0,Martin,Paul,31/01/2000,true,2]
```

### 2.1.13.3 Test 2

Le code de **test2** est le suivant :

```
1 // modifier un objet du contexte
2 public static void test2() {
3     // contexte de persistance
4     EntityManager em = getEntityManager();
5     // début transaction
6     EntityTransaction tx = em.getTransaction();
7     tx.begin();
8     // on incrémenté le nbre d'enfants de p1
9     p1.setNbenfants(p1.getNbenfants() + 1);
10    // on modifie son état marital
11    p1.setMarie(false);
12    // l'objet p1 est automatiquement sauvegardé (dirty checking)
13    // lors de la prochaine synchronisation (commit ou select)
14    // fin transaction
15    tx.commit();
16    // on affiche la nouvelle table
17    dump();
18 }
```

- le test 2 a pour objectif de modifier un objet du contexte de persistance et d'afficher ensuite le contenu de la table pour voir si la modification a eu lieu
- ligne 4 : on récupère le contexte de persistance courant
- lignes 6-7 : les choses se feront dans une transaction
- lignes 9, 11 : le nombre d'enfants de la personne p1 est changé ainsi que son état marital
- ligne 15 : fin de la transaction, donc synchronisation du contexte de persistance avec la base
- ligne 17 : affichage table

L'affichage console du test 2 est le suivant :

```
1 main : -----
2 [personnes]
3 [2,0,Durant,Sylvie,05/07/2001,false,0]
4 [1,0,Martin,Paul,31/01/2000,true,2]
5 main : -----
6 [personnes]
7 [2,0,Durant,Sylvie,05/07/2001,false,0]
8 [1,1,Martin,Paul,31/01/2000,false,3]
```

- ligne 4 : la personne p1 avant modification
- ligne 8 : la personne p1 après modification. **On notera que son n° de version est passé à 1**. Celui-ci est augmenté de 1 à chaque mise à jour de la ligne.

### 2.1.13.4 Test 3

Le code de **test3** est le suivant :

```
1 // demander des objets
2 public static void test3() {
3     // contexte de persistance
4     EntityManager em = getEntityManager();
5     // début transaction
6     EntityTransaction tx = em.getTransaction();
7     tx.begin();
8     // on demande la personne p1
```

```

9. Personne p1b = em.find(Personne.class, p1.getId());
10. // parce que p1 est déjà dans le contexte de persistance, il n'y a pas eu d'accès à la base
11. // p1b et p1 sont les mêmes références
12. System.out.format("p1==p1b ? %s%n", p1 == p1b);
13. // demander un objet qui n'existe pas rend 1 pointeur null
14. Personne px = em.find(Personne.class, -4);
15. System.out.format("px==null ? %s%n", px == null);
16. // fin transaction
17. tx.commit();
18.

```

- le test 3 s'intéresse à la méthode [EntityManager.find] qui permet d'aller chercher un objet dans la base pour le mettre dans le contexte de persistance. Nous n'expliquons plus désormais la transaction qui a lieu dans tous les tests sauf lorsqu'elle est utilisée de façon inhabituelle.
- ligne 9 : on demande au contexte de persistance, la personne qui a la même clé primaire que la personne p1. Il y a deux cas :
  - p1 se trouve déjà dans le contexte de persistance. C'est le cas ici. Alors aucun accès à la base n'est fait. La méthode *find* se contente de rendre une référence sur l'objet persisté.
  - p1 n'est pas dans le contexte de persistance. Alors un accès à la base est fait, via la clé primaire qui a été donnée. La ligne récupérée est mise dans le contexte de persistance et *find* rend la référence de ce nouvel objet persisté.
- ligne 12 : on vérifie que *find* a rendu la référence de l'objet p1 déjà dans le contexte
- ligne 14 : on demande un objet qui n'existe ni dans le contexte de persistance, ni dans la base. La méthode *find* rend alors le pointeur *null*. Ce point est vérifié ligne 15.

L'affichage console du test 3 est le suivant :

```

1. main : -----
2. p1==p1b ? true
3. px==null ? true

```

### 2.1.13.5 Test 4

Le code de **test4** est le suivant :

```

1. // supprimer un objet appartenant au contexte de persistance
2. public static void test4() {
3.     // contexte de persistance
4.     EntityManager em = getEntityManager();
5.     // début transaction
6.     EntityTransaction tx = em.getTransaction();
7.     tx.begin();
8.     // on supprime l'objet persisté p2
9.     em.remove(p2);
10.    // fin transaction
11.    tx.commit();
12.    // on affiche la nouvelle table
13.    dump();
14.}

```

- le test 4 s'intéresse à la méthode [EntityManager.remove] qui permet de supprimer un élément du contexte de persistance et donc de la base.
- ligne 9 : la personne p2 est enlevée du contexte de persistance
- ligne 11 : synchronisation du contexte avec la base
- ligne 13 : affichage de la table. Normalement, la personne p2 ne doit plus être là.

L'affichage console du test 4 est le suivant :

```

1. main : -----
2. [personnes]
3. [2,0,Durant,Sylvie,05/07/2001,false,0]
4. [1,0,Martin,Paul,31/01/2000,true,2]
5. main : -----
6. [personnes]
7. [2,0,Durant,Sylvie,05/07/2001,false,0]
8. [1,1,Martin,Paul,31/01/2000,false,3]
9. main : -----
10. p1==p1b ? true
11. px==null ? true
12. main : -----
13. [personnes]
14. [1,1,Martin,Paul,31/01/2000,false,3]

```

- ligne 3 : la personne p2 dans *test1*
- lignes 12-14 : elle n'existe plus à l'issue de *test4*.

### 2.1.13.6 Test 5

Le code de **test5** est le suivant :

```
1.// détacher, réattacher et modifier
2.public static void test5() {
3.    // nouveau contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // début transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // p1 détaché
9.    Personne oldp1=p1;
10.   // on réattache p1 au nouveau contexte
11.   p1 = em.find(Personne.class, p1.getId());
12.   // vérification
13.   System.out.format("p1==oldp1 ? %s%n", p1 == oldp1);
14.   // fin transaction
15.   tx.commit();
16.   // on incrémente le nbre d'enfants de p1
17.   p1.setNbenfants(p1.getNbenfants() + 1);
18.   // on affiche la nouvelle table
19.   dump();
20. }
```

- le test 5 s'intéresse à la vie des objets persistés au travers de plusieurs contextes de persistance successifs. Jusqu'ici, nous avions toujours utilisé le même contexte de persistance au travers des différents tests.
- ligne 4 : un nouveau contexte de persistance est demandé. La méthode [getNewEntityManager] ferme le précédent et en ouvre un nouveau. Cela a pour conséquence que les objets p1 et p2 détenus par l'application ne sont plus dans un état persistant. Ils appartenaient à un contexte qui a été fermé. On dit qu'ils sont dans un état **détaché**. Ils n'appartiennent pas au nouveau contexte de persistance.
- lignes 6-7 : début de la transaction. Elle va ici, être utilisée de façon inhabituelle.
- ligne 9 : on note l'adresse de l'objet p1 maintenant détaché.
- ligne 11 : on demande au contexte de persistance la personne p1 (ayant la clé primaire de p1). Comme le contexte est nouveau, la personne p1 ne s'y trouve pas. Un accès à la base va donc avoir lieu. L'objet ramené va être mis dans le nouveau contexte.
- ligne 13 : on vérifie que l'objet persistant *p1* du contexte est différent de l'objet *oldp1* qui était l'ancien objet *p1* détaché.
- ligne 15 : la transaction est terminée
- ligne 17 : on modifie, hors transaction, le nouvel objet persisté p1. Que se passe-t-il dans ce cas ? On veut le savoir.
- ligne 19 : on demande l'affichage de la table. On rappelle qu'à cause du *select* émis par la méthode *dump*, une synchronisation du contexte de persistance avec la base est opérée automatiquement.

L'affichage console du test 5 est le suivant :

```
1. main : ----- test4
2. [personnes]
3. [1,1,Martin,Paul,31/01/2000,false,3]
4. main : ----- test5
5. p1==oldp1 ? false
6. [personnes]
7. [1,2,Martin,Paul,31/01/2000,false,4]
```

- ligne 5 : la méthode *find* a bien fait un accès à la base, sinon les deux pointeurs seraient égaux
- lignes 7 et 3 : le nombre d'enfants de p1 a bien augmenté de 1. La modification, faite hors transaction, a donc été prise en compte. Cela est en fait dépendant du SGBD utilisé. Dans un SGBD, un ordre SQL s'exécute toujours au sein d'une transaction. si le client JPA ne démarre pas lui-même une transaction **explicite**, le SGBD va alors démarrer une transaction **implicite**. Il y a deux cas courants :
  - 1 - chaque ordre SQL individuel fait l'objet d'une transaction, ouverte avant l'ordre et fermée après. On dit qu'on est en mode **autocommit**. Tout se passe donc comme si le client JPA faisait des transactions pour chaque ordre SQL.
  - 2 - le SGBD n'est pas en mode **autocommit** et commence une transaction implicite au 1er ordre SQL que le client JPA émet hors d'une transaction et il laisse le client la fermer. Tous les ordres SQL émis par le client JPA font alors partie de la transaction **implicite**. Celle-ci peut se terminer sur différents événements : le client ferme la connexion, commence une nouvelle transaction, ...

On est dans une situation dépendant de la configuration du SGBD. On a donc du code non portable. Nous montrerons un peu plus loin, un code sans transactions et nous verrons que tous les SGBD n'ont pas le même comportement vis à vis de ce code. On considérera donc que travailler hors transactions est une erreur de programmation.

- ligne 7 : on notera que le n° de version est passé à 2.

## 2.1.13.7 Test 6

Le code de **test6** est le suivant :

```
1.// supprimer un objet n'appartenant pas au contexte de persistance
2.public static void test6() {
3.    // nouveau contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // début transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // on supprime p1 qui n'appartient pas au nouveau contexte
9.    try {
10.        em.remove(p1);
11.        // fin transaction
12.        tx.commit();
13.    } catch (RuntimeException e1) {
14.        System.out.format("Erreur à la suppression de p1 : [%s,%s]%n", e1.getClass().getName(),
e1.getMessage());
15.        // on fait un rollback de la transaction
16.        try {
17.            if (tx.isActive())
18.                tx.rollback();
19.        } catch (RuntimeException e2) {
20.            System.out.format("Erreur au rollback [%s,%s]%n", e2.getClass().getName(),
e2.getMessage());
21.        }
22.    }
23.    // on affiche la nouvelle table
24.    dump();
25.}
```

- le test 6 cherche à supprimer un objet qui n'appartient pas au contexte de persistance.
- ligne 4 : un nouveau contexte de persistance est demandé. L'ancien est donc fermé et les objets qu'il contenait deviennent détachés. C'est le cas de l'objet p1 du test 5 précédent.
- lignes 6-7 : début de la transaction.
- ligne 10 : on supprime l'objet détaché p1. On sait que cela va provoquer une exception, aussi a-t-on entouré l'opération d'un try/catch.
- ligne 12 : le commit n'aura pas lieu.
- lignes 16-21 : une transaction doit se terminer par un *commit* (toutes les opérations de la transaction sont validées) ou un *rollback* (toutes les opérations de la transaction sont annulées). On a eu une exception, donc on fait un *rollback* de la transaction. Il n'y a rien à défaire puisque l'unique opération de la transaction a échoué, mais le *rollback* met un terme à la transaction. C'est la première fois que nous utilisons l'opération **[EntityTransaction].rollback**. On aurait du le faire depuis les premiers exemples. C'est pour garder un code simple que nous ne l'avons pas fait. Le lecteur doit néanmoins conserver en mémoire que le cas du *rollback* de la transaction **doit toujours être prévu dans le code**.
- ligne 24 : on affiche la table. Normalement, elle n'a pas du changer.

L'affichage console du test 6 est le suivant :

```
1. main : ----- test5
2. p1==oldp1 ? false
3. [personnes]
4. [1,2,Martin,Paul,31/01/2000,false,4]
5. main : ----- test6
6. Erreur à la suppression de p1 : [java.lang.IllegalArgumentException, Removing a detached instance
entites.Personne#1]
7. [personnes]
8. [1,2,Martin,Paul,31/01/2000,false,4]
```

- ligne 6 : la suppression de p1 a échoué. Le message de l'exception explique qu'on a voulu supprimer un objet détaché donc ne faisant pas partie du contexte. Ce n'est pas possible.
- ligne 8 : la personne p1 est toujours là.

## 2.1.13.8 Test 7

Le code de **test7** est le suivant :

```
1.// modifier un objet n'appartenant pas au contexte de persistance
2.public static void test7() {
3.    // nouveau contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // début transaction
```

```

6. EntityTransaction tx = em.getTransaction();
7. tx.begin();
8. // on incrémente le nbre d'enfants de p1 qui n'appartient pas au nouveau contexte
9. p1.setNbenfants(p1.getNbenfants() + 1);
10. // fin transaction
11. tx.commit();
12. // on affiche la nouvelle table - elle n'a pas du changer
13. dump();
14. }

```

- le test 7 cherche à modifier un objet qui n'appartient pas au contexte de persistance et voir l'impact que cela a sur la base. On peut imaginer que ce n'en a pas. C'est ce que montrent les résultats du test.
- ligne 4 : un nouveau contexte de persistance est demandé. On a donc un contexte neuf sans objets persistés dedans.
- lignes 6-7 : début de la transaction.
- ligne 9 : on modifie l'objet détaché p1. C'est une opération qui n'implique pas le contexte de persistance **em**. On n'a donc pas à s'attendre à une exception ou quelque chose de ce genre. C'est une opération basique sur un POJO.
- ligne 11 : le commit provoque la synchronisation du contexte avec la base. Ce contexte est vide. La base n'est donc pas modifiée.
- ligne 24 : on affiche la table. Normalement, elle n'a pas du changer.

L'affichage console du test 7 est le suivant :

```

1. main : -----
2. Erreur à la suppression de p1 : [java.lang.IllegalArgumentException, Removing a detached instance
   entites.Personne#1]
3. [personnes]
4. [1,2,Martin,Paul,31/01/2000,false,4]
5. main : -----
6. [personnes]
7. [1,2,Martin,Paul,31/01/2000,false,4]

```

- ligne 7 : la personne p1 n'a pas changé dans la base. Pour le test suivant, on se souviendra quand même qu'en mémoire son nombre d'enfants est désormais à 5.

### 2.1.13.9 Test 8

Le code de **test8** est le suivant :

```

1.// réattacher un objet au contexte de persistance
2.public static void test8() {
3.    // nouveau contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // début transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // on réattache l'objet détaché p1 au nouveau contexte
9.    newp1 = em.merge(p1);
10.   // c'est newp1 qui fait désormais partie du contexte, pas p1
11.   // fin transaction
12.   tx.commit();
13.   // on affiche la nouvelle table - le nbre d'enfants de p1 a du changer
14.   dump();
15. }

```

- le test 8 réattache au contexte de persistance un objet détaché.
- ligne 4 : un nouveau contexte de persistance est demandé. On a donc un contexte neuf sans objets persistants dedans.
- lignes 6-7 : début de la transaction.
- ligne 9 : on réattache au contexte de persistance l'objet détaché p1. L'opération **merge** peut impliquer plusieurs opérations :
  - cas 1 : il existe dans le contexte de persistance un objet persistant *ps1* ayant la même clé primaire que l'objet détaché *p1*. Le contenu de *p1* est copié dans *ps1* et *merge* rend la référence de *ps1*.
  - cas 2 : il n'existe pas dans le contexte de persistance un objet persistant *ps1* ayant la même clé primaire que l'objet détaché *p1*. La base est alors interrogée pour savoir si l'objet cherché existe dans la base. Si oui, cet objet est amené dans le contexte de persistance, devient l'objet persistant *ps1* et on retombe sur le cas 1 précédent.
  - cas 3 : il n'existe, ni dans le contexte de persistance, ni dans la base, un objet de même clé primaire que l'objet détaché *p1*. Un nouvel objet [Personne] (new) est alors créé, puis mis dans le contexte de persistance. On retombe ensuite sur le cas 1.
  - au final : l'objet détaché *p1* **reste détaché**. L'opération *merge* rend une référence (ici *newp1*) sur l'objet persistant *ps1* issu du *merge*. L'application cliente doit désormais travailler avec l'objet persistant *ps1* et non avec l'objet détaché *p1*.
  - on notera une différence entre les cas 1 et 3 quant à l'ordre SQL programmé pour le *merge* : dans les cas 1 et 2, c'est ordre UPDATE alors que dans le cas 3, c'est un ordre INSERT.

- ligne 12 : le commit provoque la synchronisation du contexte avec la base. Ce contexte n'est plus vide. Il contient l'objet `newp1`. Celui-ci va être persisté dans la base.
- ligne 24 : on affiche la table pour le vérifier.

L'affichage console du test 8 est le suivant :

```

1. main : -----
2. Erreur à la suppression de p1 : [java.lang.IllegalArgumentException, Removing a detached instance
   entites.Personne#1]
3. [personnes]
4. [1,2,Martin,Paul,31/01/2000,false,4]
5. main : -----
6. [personnes]
7. [1,2,Martin,Paul,31/01/2000,false,4]
8. main : -----
9. [personnes]
10. [1,3,Martin,Paul,31/01/2000,false,5]

```

- le nombre d'enfants de `p1` était à 4 dans le test 6 (ligne 4), puis avait été passé à 5 dans le test 7 mais n'avait pas été persisté dans la base (ligne 7). Après le `merge`, `newp1` a été persisté dans la base : ligne 10, on a bien 5 enfants.
- ligne 10 : le n° de version de `newp1` est passé à 3.

### 2.1.13.10 Test 9

Le code de **test9** est le suivant :

```

1.// une requête select provoque une synchronisation
2.// de la base avec le contexte de persistance
3.public static void test9() {
4.    // contexte de persistance
5.    EntityManager em = getEntityManager();
6.    // début transaction
7.    EntityTransaction tx = em.getTransaction();
8.    tx.begin();
9.    // on incrémenté le nbre d'enfants de newp1
10.   newp1.setNbenfants(newp1.getNbenfants() + 1);
11.   // affichage personnes - le nbre d'enfants de newp1 a du changer
12.   System.out.println("[personnes]");
13.   for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList()) {
14.       System.out.println(p);
15.   }
16.   // fin transaction
17.   tx.commit();
18. }

```

- le test 9 veut montrer le mécanisme de synchronisation du contexte qui se produit automatiquement avant un `select`.
- ligne 5 : on ne change pas le contexte de persistance. `newp1` est donc dedans.
- lignes 7-8 : début de la transaction.
- ligne 10 : le nombre d'enfants de l'objet persistant `newp1` est augmenté de 1 (5 -> 6).
- lignes 12-15 : on affiche la table par un select. Le contexte va être synchronisé avec la base avant l'exécution du `select`.
- ligne 17 : fin de la transaction

Pour voir la synchronisation, on met en route l'affichage des logs Hibernate en mode DEBUG (log4j.properties) :

```

1. # Root logger option
2. log4j.rootLogger=ERROR, stdout
3.
4. # Hibernate logging options (INFO only shows startup messages)
5. log4j.logger.org.hibernate=DEBUG

```

L'affichage console du test 9 est le suivant :

```

1. main : -----
2. 14:27:27,250 DEBUG JDBCTransaction:54 - begin
3. 14:27:27,250 DEBUG ConnectionManager:415 - opening JDBC connection
4. 14:27:27,250 DEBUG JDBCTransaction:59 - current autocommit status: true
5. 14:27:27,250 DEBUG JDBCTransaction:62 - disabling autocommit
6. 14:27:27,250 DEBUG JDBCContext:210 - after transaction begin
7. [personnes]
8. 14:27:27,250 DEBUG QueryPlanCache:76 - located HQL query plan in cache (select p from Personne p
   order by p.nom asc)
9. 14:27:27,250 DEBUG AbstractFlushingEventListener:58 - flushing session
10. ...
11. 14:27:27,250 DEBUG AbstractEntityPersister:3116 - entites.Personne.nbenfants is dirty
12. 14:27:27,250 DEBUG DefaultFlushEntityEventListener:229 - Updating entity: [entites.Personne#1]
13. 14:27:27,250 DEBUG Versioning:27 - Incrementing: 3 to 4

```

```

14. ...
15. 14:27:27,250 DEBUG AbstractFlushingEventListener:85 - Flushed: 0 insertions, 1 updates, 0
   deletions to 1 objects
16. ...
17. 14:27:27,250 DEBUG ConnectionManager:463 - registering flush begin
18. 14:27:27,250 DEBUG AbstractEntityPersister:2274 - Updating entity: [entites.Personne#1]
19. 14:27:27,265 DEBUG AbstractEntityPersister:2276 - Existing version: 3 -> New version: 4
20. 14:27:27,265 DEBUG AbstractBatcher:358 - about to open PreparedStatement (open PreparedStatements:
   0, globally: 0)
21. 14:27:27,265 DEBUG SQL:393 - update jpa01_personne set VERSION=?, NOM=?, PRENOM=?,
   DATENAISANCE=?, MARIE=?, NBENFANTS=? where ID=? and VERSION=?
22. 14:27:27,265 DEBUG AbstractBatcher:476 - preparing statement
23. 14:27:27,265 DEBUG AbstractEntityPersister:1927 - Dehydrating entity: [entites.Personne#1]
24. 14:27:27,265 DEBUG IntegerType:80 - binding '4' to parameter: 1
25. 14:27:27,265 DEBUG StringType:80 - binding 'Martin' to parameter: 2
26. 14:27:27,265 DEBUG StringType:80 - binding 'Paul' to parameter: 3
27. 14:27:27,265 DEBUG DateType:80 - binding '31 janvier 2000' to parameter: 4
28. 14:27:27,265 DEBUG BooleanType:80 - binding 'false' to parameter: 5
29. 14:27:27,265 DEBUG IntegerType:80 - binding '6' to parameter: 6
30. 14:27:27,265 DEBUG IntegerType:80 - binding '1' to parameter: 7
31. 14:27:27,265 DEBUG IntegerType:80 - binding '3' to parameter: 8
32. 14:27:27,265 DEBUG AbstractBatcher:366 - about to close PreparedStatement (open
   PreparedStatements: 1, globally: 1)
33. 14:27:27,265 DEBUG AbstractBatcher:525 - closing statement
34. 14:27:27,265 DEBUG ConnectionManager:472 - registering flush end
35. 14:27:27,265 DEBUG HQLQueryPlan:150 - find: select p from Personne p order by p.nom asc
36. 14:27:27,265 DEBUG QueryParameters:277 - named parameters: {}
37. 14:27:27,265 DEBUG AbstractBatcher:358 - about to open PreparedStatement (open PreparedStatements:
   0, globally: 0)
38. 14:27:27,265 DEBUG SQL:393 - select personne0_.ID as ID0_, personne0_.VERSION as VERSION0_,
   personne0_.NOM as NOM0_, personne0_.PRENOM as PRENOM0_, personne0_.DATENAISANCE as DATENAIS5_0_,
   personne0_.MARIE as MARIE0_, personne0_.NBENFANTS as NBENFANTS0_ from jpa01_personne personne0_
   order by personne0_.NOM asc
39. ...
40. 14:27:27,265 DEBUG Loader:1164 - result row: EntityKey[entites.Personne#1]
41. ...
42. 14:27:27,265 DEBUG Loader:839 - total objects hydrated: 0
43. 14:27:27,265 DEBUG StatefulPersistenceContext:748 - initializing non-lazy collections
44. [1,4,Martin,Paul,31/01/2000,false,6]
45. 14:27:27,265 DEBUG JDBCTransaction:103 - commit
46. 14:27:27,265 DEBUG SessionImpl:337 - automatically flushing session
47. ...
48. 14:27:27,265 DEBUG AbstractFlushingEventListener:91 - Flushed: 0 (re)creations, 0 updates, 0
   removals to 0 collections
49. ...
50. 14:27:27,296 DEBUG JDBCTransaction:116 - committed JDBC Connection
51. ...

```

- ligne 1 : le test 9 démarre
- lignes 2-6 : la transaction Jdbc démarre. Le mode autocommit du SGBD est désactivé (ligne 5)
- ligne 7 : affichage provoqué par la ligne 12 du code Java. Les lignes suivantes du code Java vont provoquer un *select* et donc une synchronisation du contexte de persistance avec la base.
- ligne 8 : l'ordre JPQL que l'on veut émettre a déjà été émis. Hibernate le trouve dans son cache de "requêtes préparées".
- ligne 9 : Hibernate annonce qu'il va procéder à un flush du contexte de persistance
- lignes 11-12 : Hibernate(Hb) découvre que l'entité Personne#1 (de clé primaire 1) a été changée (**dirty**).
- lignes 12-13 : Hb annonce qu'il met à jour cet élément et passe son n° de version de 3 à 4.
- ligne 15 : la synchronisation du contexte va provoquer 0 insertion, 1 mise à jour (update), 0 suppression (delete)
- lignes 17-34 : synchronisation du contexte (flush). A noter : l'incrément de la version (ligne 19), l'ordre SQL update préparé (ligne 21), les valeurs des paramètres de l'ordre *update* (lignes 24-31).
- ligne 35 : le *select* commence
- ligne 38 : l'ordre SQL qui va être exécuté
- ligne 40 : le *select* ne ramène qu'une ligne
- ligne 42 : Hb découvre qu'il a déjà dans son contexte de persistance, l'entité Personne#1 que le select a ramenée de la base. Il ne copie pas alors la ligne obtenue de la base dans le contexte, opération qu'il appelle "hydratation".
- ligne 43 : il vérifie si les objets ramenés par le *select* ont des dépendances (clés étrangères en général) qu'il faudrait également charger (non-lazy collections). Ici il n'y en a pas.
- ligne 44 : affichage provoqué par le code Java
- ligne 45 : fin de la transaction Jdbc demandée par le code Java
- ligne 46 : la synchronisation automatique du contexte qui a lieu lors des *commit*, commence.
- ligne 48 : Hb découvre que le contexte n'a pas changé depuis la synchronisation précédente.
- ligne 50 : fin du *commit*.

De nouveau, les logs d'Hibernate en mode DEBUG se montrent très utiles pour savoir exactement ce que fait Hibernate.

### 2.1.13.11 Test 10

Le code de **test10** est le suivant :

```
1.// contrôle de version (optimistic locking)
2.public static void test10() {
3.    // contexte de persistance
4.    EntityManager em = getEntityManager();
5.    // début transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // incrémenter la version de newp1 directement dans la base (native query)
9.    em.createNativeQuery(String.format("update %s set VERSION=VERSION+1 WHERE ID=%d", TABLE_NAME,
newp1.getId())).executeUpdate();
10.   // fin transaction
11.   tx.commit();
12.   // début nouvelle transaction
13.   tx = em.getTransaction();
14.   tx.begin();
15.   // on incrémente le nbre d'enfants de newp1
16.   newp1.setNbEnfants(newp1.getNbEnfants() + 1);
17.   // fin transaction - elle doit échouer car newp1 n'a plus la bonne version
18.   try {
19.       tx.commit();
20.   } catch (RuntimeException e1) {
21.       System.out.format("Erreur lors de la mise à jour de newp1 [%s,%s,%s,%s]%n",
e1.getClass().getName(), e1.getMessage(), e1.getCause().getClass().getName(),
e1.getCause().getMessage());
22.       // on fait un rollback de la transaction
23.       try {
24.           if (tx.isActive())
25.               tx.rollback();
26.       } catch (RuntimeException e2) {
27.           System.out.format("Erreur au rollback [%s,%s]%n", e2.getClass().getName(),
e2.getMessage());
28.       }
29.   }
30.   // on ferme le contexte qui n'est plus à jour
31.   em.close();
32.   // dump de la table - la version de p1 a du changer
33.   dump();
34. }
```

- le test 10 veut montrer le mécanisme amené par le champ *version* de l'**@Entity Personne**, qui est doté de l'attribut JPA **@Version**. Nous avons expliqué que cette annotation faisait que dans la base, la valeur de la colonne associée à l'annotation **@Version** était incrémentée à chaque *update* fait sur la ligne à laquelle elle appartient. Ce mécanisme, appelé également **verrouillage optimiste** (*optimistic locking*), impose que le client qui veut modifier un objet O dans la base ait la dernière version de celui-ci. S'il ne l'a pas, c'est que l'objet a été modifié depuis qu'il l'a obtenu et on doit l'en avertir.
- ligne 4 : on ne change pas le contexte de persistance. *newp1* est donc dedans.
- lignes 6-7 : début d'une transaction.
- ligne 9 : la version de l'objet *newp1* est augmentée de 1 (4 -> 5) directement dans la base. Les requêtes de type *nativeQuery* court-circuitent le contexte de persistance et tapent directement dans la base. Le résultat est que l'objet persistant *newp1* et son image dans la base n'ont plus la même version.
- ligne 10 : fin de la première transaction
- lignes 13-14 : début d'une seconde transaction
- ligne 16 : le nombre d'enfants de l'objet persistant *newp1* est augmenté de 1 (6 -> 7).
- ligne 19 : fin de la transaction. Une synchronisation a donc lieu. Elle va provoquer la mise à jour du nombre d'enfants de *newp1* dans la base. Celle-ci va échouer car l'objet persistant *newp1* a la version 4, alors que dans la base l'objet à mettre à jour a la version 5. Une exception va être lancée ce qui justifie le try / catch du code.
- ligne 21 : on affiche l'exception et sa cause.
- ligne 25 : rollback de la transaction
- ligne 33 : affichage de la table : on devrait voir que la version de *newp1* est 5 dans la base.

L'affichage console du test 10 est le suivant :

```
1. main : -----
2. [personnes]
3. [1,4,Martin,Paul,31/01/2000,false,6]
4. main : -----
5. Erreur lors de la mise à jour de newp1 [javax.persistence.RollbackException,Error while committing
the transaction,org.hibernate.StaleObjectStateException,Row was updated or deleted by another
transaction (or unsaved-value mapping was incorrect): [entites.Personne#1]]
6. [personnes]
7. [1,5,Martin,Paul,31/01/2000,false,6]
```

- ligne 5 : le commit lance bien une exception. Elle est de type *[javax.persistence.RollbackException]*. Le message associé est vague. Si on s'intéresse à la cause de cette exception (*Exception.getCause*), on voit qu'on a une exception Hibernate due au fait qu'on cherche à modifier une ligne de la base sans avoir la bonne version.

- ligne 7 : on voit que la version de *newp1* dans la base a bien été passée à 5 par la *nativeQuery*.

### 2.1.13.12 Test 11

Le code de **test11** est le suivant :

```

1.// rollback d'une transaction
2.public static void test11() throws ParseException {
3.    // contexte de persistance
4.    EntityManager em = getEntityManager();
5.    // début transaction
6.    EntityTransaction tx = null;
7.    try {
8.        tx = em.getTransaction();
9.        tx.begin();
10.       // on réattache p1 au contexte en allant le chercher dans la base
11.       p1 = em.find(Personne.class, p1.getId());
12.       // on incrémenté le nbre d'enfants de p1
13.       p1.setNbEnfants(p1.getNbEnfants() + 1);
14.       // affichage personnes - le nbre d'enfants de p1 a du changer
15.       System.out.println("[personnes]");
16.       for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList())
17.       {
18.           System.out.println(p);
19.       }
20.       // création de 2 personnes de nom identique, ce qui est interdit par la DDL
21.       Personne p3 = new Personne("X", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
22.       true, 2);
23.       Personne p4 = new Personne("X", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
24.       true, 2);
25.       // persistance des personnes
26.       em.persist(p3);
27.       em.persist(p4);
28.       // fin transaction
29.       tx.commit();
30.   } catch (RuntimeException e1) {
31.       // on a eu un pb
32.       System.out.format("Erreur dans transaction [%s,%s,%s,%s,%s,%s]%n", e1.getClass().getName(),
33.           e1.getMessage(),
34.           e1.getCause().getClass().getName(), e1.getCause().getMessage(),
35.           e1.getCause().getCause().getClass().getName(), e1.getCause().getCause()
36.               .getMessage());
37.       try {
38.           if (tx.isActive())
39.               tx.rollback();
40.       } catch (RuntimeException e2) {
41.           System.out.format("Erreur au rollback [%s]%n", e2.getMessage());
42.       }
43.   }
44. }
```

- le test 11 s'intéresse au mécanisme du *rollback* d'une transaction. Une transaction fonctionne en tout ou rien : les opérations SQL qu'elle contient sont soit toutes exécutées avec succès (*commit*), soit toutes annulées en cas d'échec de l'une d'elles (*rollback*).
- ligne 4 : on continue avec le même contexte de persistance. Le lecteur se souvient peut-être que le contexte a été fermé à l'issue du crash du test précédent. Dans ce cas, [getEntityManager] délivre un contexte tout neuf, donc vide.
- lignes 7-27 : un unique try / catch pour contrôler les problèmes qu'on va rencontrer
- lignes 8-9 : début d'une transaction qui va contenir plusieurs opérations SQL
- ligne 11 : p1 est cherché dans la base et mis dans le contexte
- ligne 13 : on augmente le nombre d'enfants de p1 (6 -> 7)
- lignes 15-18 : on affiche le contenu de la base, ce qui va forcer une synchronisation du contexte. Dans la base, le nombre d'enfants de p1 va passer à 7, ce que devrait confirmer l'affichage console.
- lignes 20-21 : création de 2 personnes p3 et p4 de même nom. Or le champ **nom** de l'@Entity Personne a l'attribut **unique=true**, ce qui a eu pour conséquence d'engendrer une contrainte d'unicité sur la colonne NOM de la table [jpa01\_personne] de la table.
- lignes 23-24 : les personnes p3 et p4 sont mises dans le contexte de persistance.
- ligne 26 : la transaction est committée. S'ensuit une deuxième synchronisation du contexte, la première ayant eu lieu à l'occasion du select. JPA va émettre deux ordres SQL *insert* pour les personnes p3 et p4. p3 va être inséré. Pour p4, le SGBD va lancer une exception, car p4 porte le même nom que p3. p4 n'est donc pas inséré et le pilote Jdbc remonte une exception au client.
- ligne 27 : on gère l'exception

- lignes 29-31 : on affiche l'exception et ses deux précédentes causes dans la chaîne des exceptions qui nous ont amené jusque là.
- ligne 34 : on fait un rollback de la transaction actuellement active. Celle-ci a commencé ligne 9 du code Java. Depuis une opération *update* a été faite pour modifier le nombre d'enfants de *p1* puis une opération *insert* pour la personne *p3*. Tout cela va être annulé par le rollback.
- ligne 39 : le contexte de persistance est vidé
- ligne 42 : la table [jpa01\_personne] est affichée. Il faut vérifier que *p1* a toujours 6 enfants et que ni *p3*, ni *p4* ne sont dans la table.

L'affichage console du test 11 est le suivant :

```

1. main : -----
2. [personnes]
3. [1,6,Martin,Paul,31/01/2000,false,7]
4. 14:50:30,312 ERROR JDBCExceptionReporter:72 - Duplicate entry 'X' for key 2
5. Erreur dans transaction
[javax.persistence.EntityExistsException,org.hibernate.exception.ConstraintViolationException:
could not insert: [entites.Personne],org.hibernate.exception.ConstraintViolationException,could
not insert: [entites.Personne],java.sql.SQLException,Duplicate entry 'X' for key 2]
6. [personnes]
7. [1,5,Martin,Paul,31/01/2000,false,6]
```

- ligne 3 : le nombre d'enfants de *p1* est passé de 6 à 7 dans la base, la version de *p1* est passée à 6.
- ligne 4 : l'exception récupérée à l'occasion du commit de la transaction. Si on lit bien, on voit que la cause est une clé dupliquée X (le nom). C'est l'insertion de *p4* qui provoque cette erreur alors que *p3* déjà inséré a également le nom X.
- ligne 7 : la table après le rollback. *p1* a retrouvé sa version 5 et son nombre d'enfants 6, *p3* et *p4* n'ont pas été insérés.

### 2.1.13.13 Test 12

Le code de **test12** est le suivant :

```

1. // on refait la même chose mais sans les transactions
2. // on obtient le même résultat qu'auparavant avec les SGBD : FIREBIRD, ORACLE XE, POSTGRES, MYSQL5
3. // avec SQLSERVER on a une table vide. La connexion est laissée dans un état qui empêche la réexécution
4. // du programme. Il faut alors relancer le serveur.
5. // idem avec le SGBD Derby
6. // HSQL insère la 1ère personne - il n'y a pas de rollback
7.
8. public static void test12() throws ParseException {
9.     // on réattache p1
10.    p1 = em.find(Personne.class, p1.getId());
11.    // on incrémente le nbre d'enfants de p1
12.    p1.setNbenfants(p1.getNbenfants() + 1);
13.    // affichage personnes - le nbre d'enfants de p1 a du changer
14.    System.out.println("[personnes]");
15.    for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList()) {
16.        System.out.println(p);
17.    }
18.    // création de 2 personnes de nom identique, ce qui est interdit par la DDL
19.    Personne p3 = new Personne("X", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
20.        true, 2);
21.    Personne p4 = new Personne("X", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
22.        true, 2);
23.    // persistance des personnes
24.    em.persist(p3);
25.    em.persist(p4);
26.    // dump qui va provoquer la synchro du contexte em avec la BD
27.    try {
28.        dump();
29.    } catch (RuntimeException e3) {
30.        System.out.format("Erreur dans dump [%s,%s,%s,%s]%n", e3.getClass().getName(),
31.            e3.getMessage(), e3.getCause().getClass().getName(), e3
32.                .getCause().getMessage());
33.    }
34.    // on ferme le contexte actuel
35.    em.close();
36.    // dump
37.    dump();
38.}
```

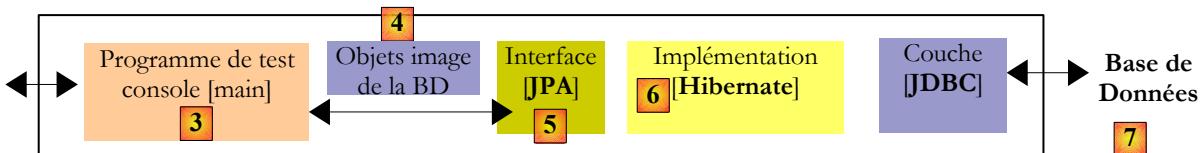
- le test 12 refait la même chose que le test 11 mais hors transaction. On veut voir ce qui se passe dans ce cas.
- lignes 1-6 : donnent les résultats des tests avec divers SGBD :

- avec un certain nombre de SGBD (Firebird, Oracle, MySQL5, Postgres) on a le même résultat qu'avec le test 11. Ce qui fait penser que ces SGBD ont initié d'eux-mêmes une transaction couvrant tous les ordres SQL reçus jusqu'à celui qui a provoqué l'erreur et qu'ils ont eux-mêmes initié un *rollback*.
- avec d'autres SGBD (SQL Server, Apache Derby) on a un plantage de l'application et / ou du SGBD.
- avec le SGBD HSQLDB, il semble que la transaction ouverte par le SGBD soit en mode *autocommit* : la modification du nombre d'enfants de *p1* et l'insertion de *p3* sont rendues permanentes. Seule l'insertion de *p4* échoue.

On a donc un résultat dépendant du SGBD, ce qui rend l'application non portable. On retiendra que les opérations sur le contexte de persistance doivent toujours se faire au sein d'une transaction.

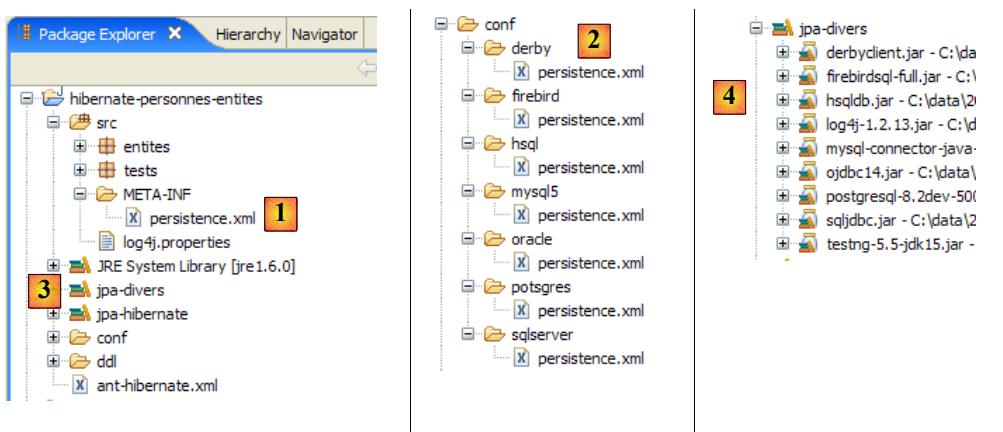
## 2.1.14 Changer de SGBD

Revenons sur l'architecture de test de notre projet actuel :



L'application cliente [3] ne voit que l'interface JPA [5]. Elle ne voit pas ni l'implémentation réelle de celle-ci, ni le SGBD cible. On doit donc pouvoir changer ces deux éléments de la chaîne sans changements dans le client [3]. C'est ce que nous essayons de voir maintenant en commençant par changer le SGBD. Nous avions utilisé jusqu'à maintenant MySQL5. Nous en présentons six autres décrits en annexes (paragraphe 5, page 209) en espérant que parmi ceux-ci, il y a le SGBD favori du lecteur.

Dans tous les cas, la modification à faire dans le projet Eclipse est simple (cf ci-dessous) : remplacer le fichier **persistence.xml** [1] de configuration de la couche JPA par l'un de ceux du dossier *conf* [2] du projet. Les pilotes JDBC de ces SGBD sont déjà présents dans la bibliothèque [jpa-divers] [3] et [4].



### 2.1.14.1 Oracle 10g Express

Oracle 10g Express est présenté en Annexes au paragraphe 5.7, page 276. Le fichier *persistence.xml* d'Oracle est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.     <!-- provider -->
5.     <provider>org.hibernate.ejb.HibernatePersistence</provider>
6.     <properties>
7.       <!-- Classes persistantes -->
8.       <property name="hibernate.archive.autodetection" value="class, hbm" />
9.       <!-- logs SQL -->
10.      <property name="hibernate.show_sql" value="true"/>
11.      <property name="hibernate.format_sql" value="true"/>
12.      <property name="use_sql_comments" value="true"/>
13.    <!-->
14.    <!-- connexion JDBC -->
15.    <property name="hibernate.connection.driver_class" value="oracle.jdbc.OracleDriver" />
16.    <property name="hibernate.connection.url" value="jdbc:oracle:thin:@localhost:1521:xe" />
17.    <property name="hibernate.connection.username" value="jpa" />

```

```

18.      <property name="hibernate.connection.password" value="jpa" />
19.      <!-- création automatique du schéma -->
20.      <property name="hibernate.hbm2ddl.auto" value="create" />
21.      <!-- Dialecte -->
22.      <property name="hibernate.dialect" value="org.hibernate.dialect.OracleDialect" />
23.      <!-- propriétés DataSource c3p0 -->
24.      <property name="hibernate.c3p0.min_size" value="5" />
25.      <property name="hibernate.c3p0.max_size" value="20" />
26.      <property name="hibernate.c3p0.timeout" value="300" />
27.      <property name="hibernate.c3p0.max_statements" value="50" />
28.      <property name="hibernate.c3p0.idle_test_period" value="3000" />
29.    </properties>
30.  </persistence-unit>
31.</persistence>

```

Cette configuration est identique à celle faite pour le SGBD MySQL5, aux détails près suivants :

- lignes 15-18 qui configurent la liaison JDBC avec la base de données
- ligne 22 : qui fixe le dialecte SQL à utiliser

Pour les exemples à venir, nous ne préciserons que les lignes qui changent. Pour une explication de la configuration on consultera l'annexe consacrée au SGBD utilisé. Un exemple d'utilisation de la liaison JDBC y est donné à chaque fois, dans le contexte du plugin [SQL Explorer]. Avec les informations de l'annexe, le lecteur pourra répéter l'opération de vérification du résultat de l'application [InitDB] faite au paragraphe 2.1.10.2, page 30.

Nous procédons comme indiqué au paragraphe sus-nommé :

- lancer le SGBD Oracle
- mettre conf/oracle/persistence.xml dans META-INF/persistence.xml
- exécuter l'application [InitDB]

On obtient les résultats suivants sur la console :

```

[personnes]
[2,0,Durant,Sylvie,05/07/2001,false,0]
[1,0,Martin,Paul,31/01/2000,true,2]
terminé ...

```

Par la suite, nous ne présenterons plus cette copie d'écran qui est toujours la même. Plus intéressante est la perspective SQL Explorer de la liaison JDBC avec le SGBD. On suivra la démarche expliquée au paragraphe 2.1.8, page 25.

COLUMN_NAME	DATA_TYPE	TYPE_NAME	COLUMN_SIZE
ID	3	NUMBER	10
VERSION	3	NUMBER	10
NOM	12	VARCHAR2	30
PRENOM	12	VARCHAR2	30
DATENAISANCE	91	DATE	7
MARIE	3	NUMBER	1
NBENFANTS	3	NUMBER	10

ID	VERSION	NOM	PR...	DATENAISANCE	MARIE	NBENFANTS
1	0	Martin	Paul	2000-01-31 00:00:00	1	2
2	0	Durant	Sylvie	2001-07-05 00:00:00	0	0

- en [1] : la connexion avec Oracle
- en [2] : l'arborescence de la connexion après exécution de [InitDB]

- en [3] : la structure de la table [jpa01\_personne]
- en [4] : son contenu.

Ceci fait, le lecteur est invité à exécuter l'application [Main] puis à arrêter le SGBD.

### 2.1.14.2 PostgreSQL 8.2

PostgreSQL 8.2 est présenté en Annexes au paragraphe 5.6, page 268. Son fichier *persistence.xml* est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.   ...
5.     <!-- connexion JDBC -->
6.     <property name="hibernate.connection.driver_class" value="org.postgresql.Driver" />
7.     <property name="hibernate.connection.url" value="jdbc:postgresql:jpa" />
8.     <property name="hibernate.connection.username" value="jpa" />
9.     <property name="hibernate.connection.password" value="jpa" />
10. ...
11.     <!-- Dialecte -->
12.     <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
13. ...
14. </persistence-unit>
15. </persistence>
```

Pour exécuter [InitDB] :

- lancer le SGBD PostgreSQL
- mettre conf/postgres/persistence.xml dans META-INF/persistence.xml
- exécuter l'application [InitDB]

La perspective SQL Explorer de la liaison JDBC avec le SGBD est la suivante :

COLUMN_NAME	DATA_TYPE	TYPE_NAME	COLUMN_SIZE
id	4	int4	10
version	4	int4	10
nom	12	varchar	30
prenom	12	varchar	30
datenaissance	91	date	13
marie	-7	bool	1
nbenfants	4	int4	10

id	version	nom	pr...	datenaissance	marie	nbenfants
1	0	Martin	Paul	2000-01-31 00:00:00	t	2
2	0	Durant	Sylvie	2001-07-05 00:00:00	f	0

- en [1] : la connexion avec PostgreSQL
- en [2] : l'arborescence de la connexion après exécution de [InitDB]
- en [3] : la structure de la table [jpa01\_personne]
- en [4] : son contenu.

Ceci fait, le lecteur est invité à exécuter l'application [Main] puis à arrêter le SGBD

### 2.1.14.3 SQL Server Express 2005

SQL Server Express 2005 est présenté en Annexes au paragraphe 5.8, page 284. Son fichier *persistence.xml* est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
```

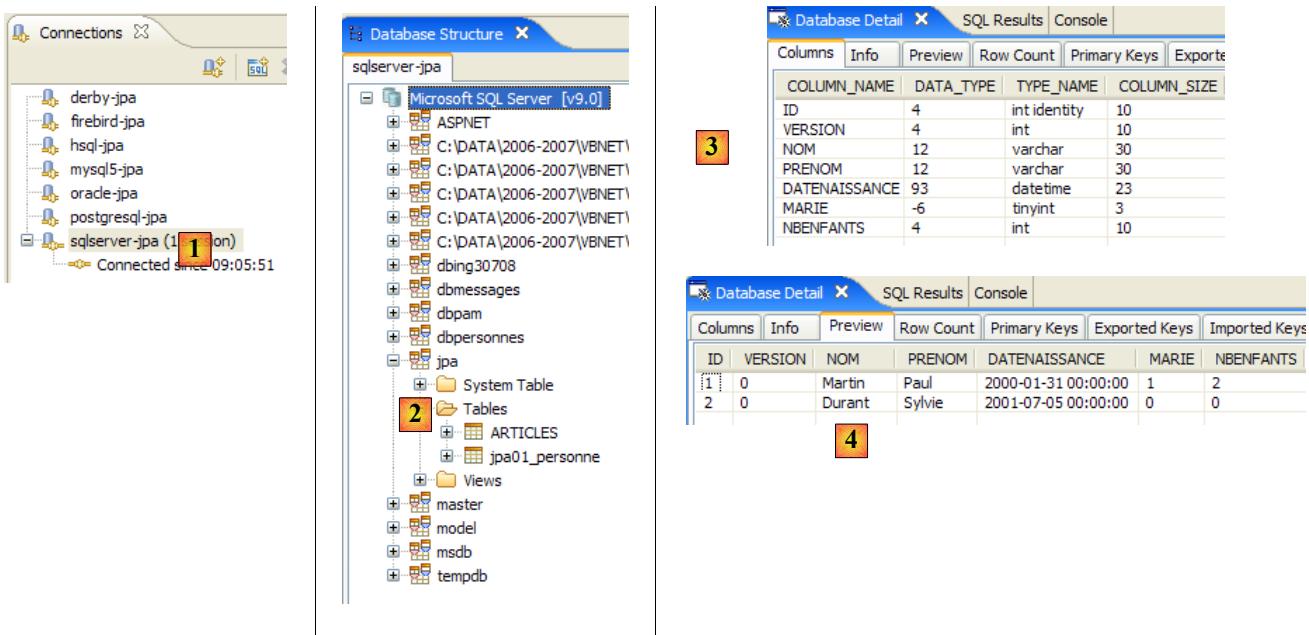
```

4. ...
5.      <!-- connexion JDBC -->
6.      <property name="hibernate.connection.driver_class"
7.          value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
8.      <property name="hibernate.connection.url"
9.          value="jdbc:sqlserver://localhost\\SQLEXPRESS:1433;databaseName=jpa" />
10.     <property name="hibernate.connection.username" value="jpa" />
11.     <property name="hibernate.connection.password" value="jpa" />
12. ...
13. ...
14. </persistence-unit>
15.</persistence>
```

Pour exécuter [InitDB] :

- lancer le SGBD SQL Server
- mettre conf/sqlserver/persistence.xml dans META-INF/persistence.xml
- exécuter l'application [InitDB]

La perspective SQL Explorer de la liaison JDBC avec le SGBD est la suivante :



- en [1] : la connexion avec SQL Server
- en [2] : l'arborescence de la connexion après exécution de [InitDB]
- en [3] : la structure de la table [jpa01\_personne]
- en [4] : son contenu.

Ceci fait, le lecteur est invité à exécuter l'application [Main] puis à arrêter le SGBD

#### 2.1.14.4 Firebird 2.0

Firebird 2.0 est présenté en Annexes au paragraphe 5.4, page 240. Son fichier *persistence.xml* est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.   ...
5.       <!-- connexion JDBC -->
6.       <property name="hibernate.connection.driver_class" value="org.firebirdsql.jdbc.FBDriver" />
7.       <property name="hibernate.connection.url"
8.           value="jdbc:firebirdsql:localhost/3050:C:\\data\\2006-2007\\eclipse\\dvp-jpa\\annexes\\firebird\\jpa.fdb"
9.           />
10.      <property name="hibernate.connection.username" value="sysdba" />
11.      <property name="hibernate.connection.password" value="masterkey" />
12. ...
13. ...
14. </persistence-unit>
15.</persistence>
```

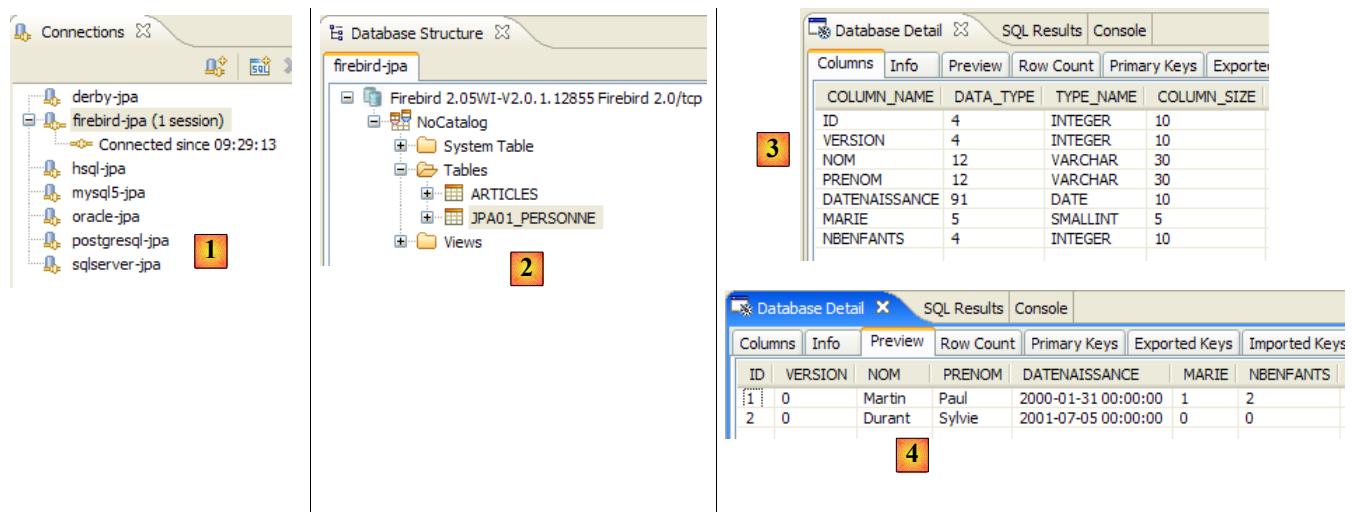
```

14. </persistence-unit>
15. </persistence>
```

Pour exécuter [InitDB] :

- lancer le SGBD Firebird
- mettre conf/firebird/persistence.xml dans META-INF/persistence.xml
- exécuter l'application [InitDB]

La perspective SQL Explorer de la liaison JDBC avec le SGBD est la suivante :



- en [1] : la connexion avec Firebird
- en [2] : l'arborescence de la connexion après exécution de [InitDB]
- en [3] : la structure de la table [jpa01\_personne]
- en [4] : son contenu.

Ceci fait, le lecteur est invité à exécuter l'application [Main] puis à arrêter le SGBD.

### 2.1.14.5 Apache Derby

Apache Derby est présenté en Annexes au paragraphe 5.10, page 295. Son fichier *persistence.xml* est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.     ...
5.       <!-- connexion JDBC -->
6.       <property name="hibernate.connection.driver_class"
  value="org.apache.derby.jdbc.ClientDriver" />
7.       <property name="hibernate.connection.url" value="jdbc:derby://localhost:1527//data/2006-
  2007/eclipse/dvp-jpa/annexes/derby/jpa;create=true" />
8.       <property name="hibernate.connection.username" value="jpa" />
9.       <property name="hibernate.connection.password" value="jpa" />
10.    ...
11.    <!-- Dialecte -->
12.    ...
13.  </persistence-unit>
14. </persistence>
```

Pour exécuter [InitDB] :

- lancer le SGBD Apache Derby
- mettre conf/derby/persistence.xml dans META-INF/persistence.xml
- exécuter l'application [InitDB]

La perspective SQL Explorer de la liaison JDBC avec le SGBD est la suivante :

The figure consists of three side-by-side screenshots from the Eclipse IDE:

- Screenshot 1:** The Connections view shows a list of available databases, with "derby-jpa (1 session)" selected. A red box labeled [1] highlights this selection.
- Screenshot 2:** The Database Structure view for the "derby-jpa" connection. It displays the schema hierarchy under "Apache Derby [v10.2]". A red box labeled [2] highlights the "Tables" node, which contains "ARTICLES" and "HIBERNATE\_UNIQUE\_KEY".
- Screenshot 3:** The Database Detail view for the "derby-jpa" connection, showing the structure of the "JPA01\_PERSONNE" table. A red box labeled [3] highlights the table structure.
- Screenshot 4:** The Database Detail view showing the contents of the "JPA01\_PERSONNE" table. A red box labeled [4] highlights the data rows.

- en [1] : la connexion avec Apache Derby
- en [2] : l'arborescence de la connexion après exécution de [InitDB]. On remarquera la table [HIBERNATE\_UNIQUE\_KEY] créée par JPA / Hibernate pour générer automatiquement les valeurs successives de la clé primaire ID. Nous avons déjà indiqué que ce mécanisme était souvent propriétaire. On le voit clairement ici. Grâce à JPA, le développeur n'a pas à entrer dans ces détails de SGBD.
- en [3] : la structure de la table [jpa01\_personne]
- en [4] : son contenu.

Ceci fait, le lecteur est invité à exécuter l'application [Main] puis à arrêter le SGBD.

### 2.1.14.6 HSQLDB

HSQLDB est présenté en Annexes au paragraphe 5.9, page 291. Son fichier *persistence.xml* est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.   ...
5.     <!-- connexion JDBC -->
6.     <property name="hibernate.connection.driver_class" value="org.hsqldb.jdbcDriver" />
7.     <property name="hibernate.connection.url" value="jdbc:hsqldb:hsql://localhost" />
8.     <property name="hibernate.connection.username" value="sa" />
9.     <!--
10.       <property name="hibernate.connection.password" value="" />
11.     -->
12.   ...
13.     <!-- Dialecte -->
14.     <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
15.   ...
16.   </properties>
17. </persistence-unit>
18.</persistence>
```

Pour exécuter [InitDB] :

- lancer le SGBD HSQL
- mettre conf/hsql/persistence.xml dans META-INF/persistence.xml
- exécuter l'application [InitDB]

La perspective SQL Explorer de la liaison JDBC avec le SGBD est la suivante :

The screenshot shows the GlassFish Admin Console interface. On the left, under 'Connections', there is a list of JPA connections: derby-jpa, firebird-jpa, hsql-jpa (1 session), mysql5-jpa, oracle-jpa, postgresql-jpa, and sqlserver-jpa. A red box labeled '1' highlights the 'hsqldb-jpa' connection. In the center, the 'Database Structure' window for the 'hsqldb-jpa' connection is open, showing the schema tree. A red box labeled '2' highlights the 'Tables' node, which contains 'ARTICLES' and 'JPA01\_PERSONNE'. On the right, two 'Database Detail' windows are shown. The top one displays the columns of the 'JPA01\_PERSONNE' table:

COLUMN_NAME	DATA_TYPE	TYPE_NAME	COLUMN_SIZE
ID	4	INTEGER	<null>
VERSION	4	INTEGER	<null>
NOM	12	VARCHAR	30
PRENOM	12	VARCHAR	30
DATENAISSE	91	DATE	<null>
MARIE	16	BOOLEAN	<null>
NBENFANTS	4	INTEGER	<null>

The bottom 'Database Detail' window shows the data for the 'JPA01\_PERSONNE' table:

ID	VERSION	NOM	PRENOM	DATENAISSE	MARIE	NBENFANTS
1	0	Martin	Paul	2000-01-31 00:00:00	true	2
2	0	Durant	Sylvie	2001-07-05 00:00:00	false	0

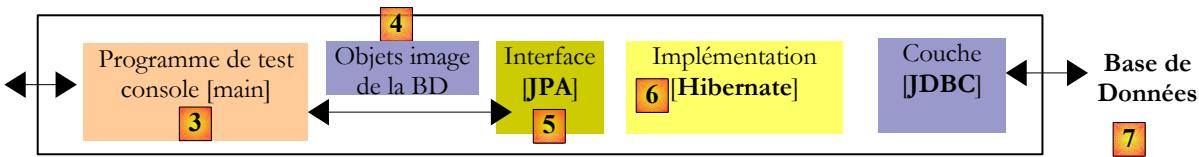
A red box labeled '3' points to the first row of the top table, and a red box labeled '4' points to the second row of the bottom table.

- en [1] : la connexion avec HSQL
- en [2] : l'arborescence de la connexion après exécution de [InitDB].
- en [3] : la structure de la table [jpa01\_personne]
- en [4] : son contenu.

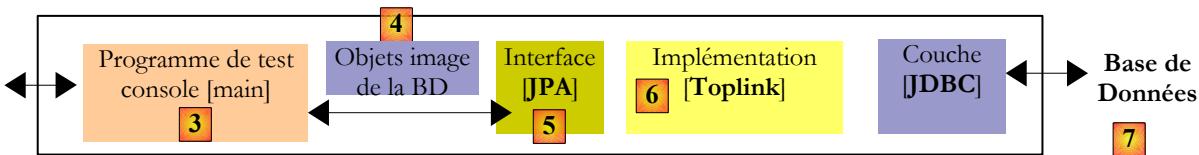
Ceci fait, le lecteur est invité à exécuter l'application [Main] puis à arrêter le SGBD.

## 2.1.15 Changer d'implémentation JPA

Revenons sur l'architecture de test de notre projet actuel :

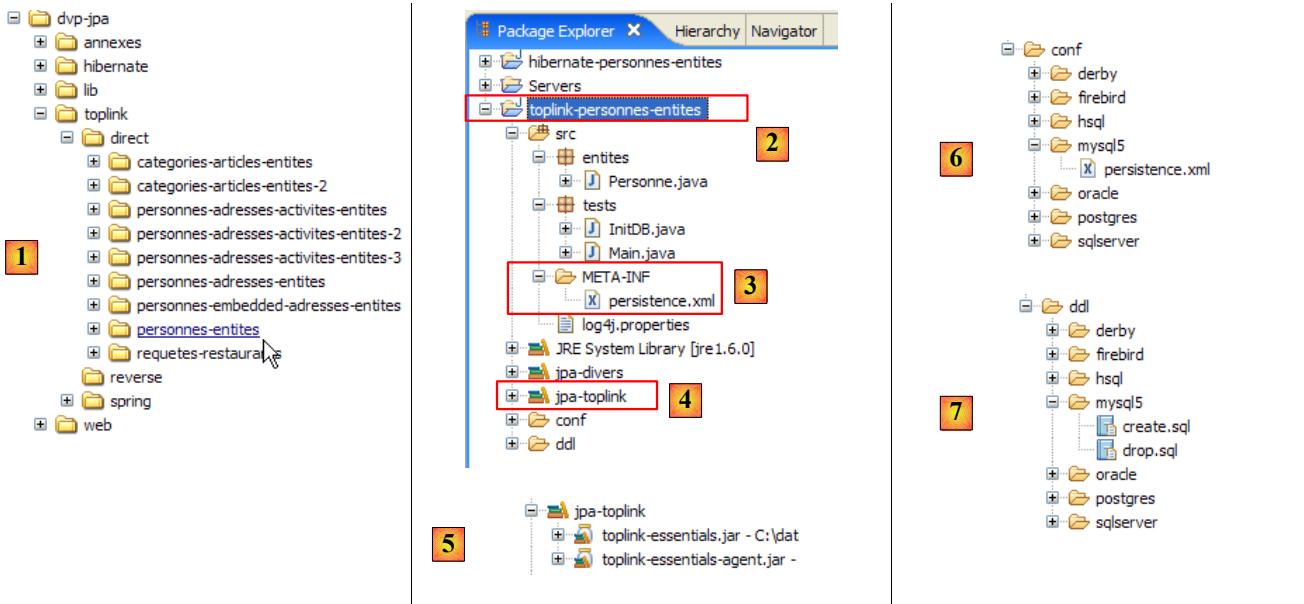


L'étude précédente a montré que nous avons pu changer le SGBD [7] sans rien changer au code client [3]. Nous changeons maintenant l'implémentation JPA [6] et montrons là encore que cela se fait de façon transparente pour le code client [3]. Nous prenons une implémentation **TopLink** [<http://www.oracle.com/technology/products/ias/toplink/jpa/index.html>] :



### 2.1.15.1 Le projet Eclipse

A l'occasion du changement d'implémentation JPA, nous créons un nouveau projet Eclipse afin de ne pas polluer le projet existant. En effet, le nouveau projet utilise des bibliothèques de persistance qui peuvent entrer en conflit avec celles d'Hibernate :



- en [1] : le dossier [<exemples>/toplink/direct/personnes-entites] contient le projet Eclipse. Importer celui-ci.
- en [2] : le projet [toplink-personnes-entites] importé. Il est **identique** (il a été obtenu par recopie) au projet [hibernate-personne-entites] à deux détails près :
  - le fichier [META-INF/persistence.xml] [3] configure désormais une couche JPA / Toplink
  - la bibliothèque [jpa-hibernate] a été remplacée par la bibliothèque [jpa-toplink] [4] et [5] (cg paragraphe 1.5, page 7).
- en [6] : le dossier [conf] contient une version du fichier [persistence.xml] pour chaque SGBD.
- en [7] : le dossier [ddl] qui va contenir les scripts SQL de génération du schéma de la base de données.

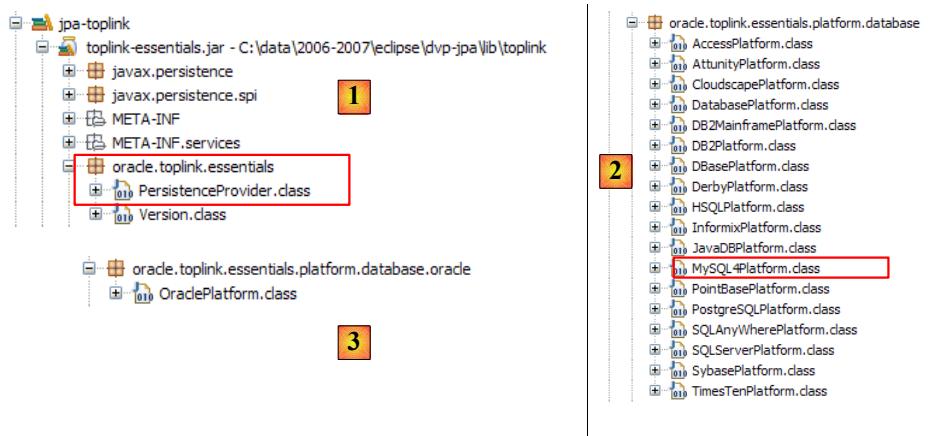
### 2.1.15.2 La configuration de la couche JPA / Toplink

Nous savons que la couche JPA est configurée par le fichier [META-INF/persistence.xml]. Celui-ci configure désormais une implémentation JPA / Toplink. Son contenu pour une couche JPA interfacée avec le SGBD MySQL5 est le suivant ;

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.     <!-- provider -->
5.     <provider>oracle.toplink.essentials.PersistenceProvider</provider>
6.     <!-- classes persistantes -->
7.     <class>entites.Personne</class>
8.     <!-- propriétés de l'unité de persistance -->
9.     <properties>
10.       <!-- connexion JDBC -->
11.         <property name="toplink.jdbc.driver" value="com.mysql.jdbc.Driver" />
12.         <property name="toplink.jdbc.url" value="jdbc:mysql://localhost:3306/jpa" />
13.         <property name="toplink.jdbc.user" value="jpa" />
14.         <property name="toplink.jdbc.password" value="jpa" />
15.         <property name="toplink.jdbc.read-connections.max" value="3" />
16.         <property name="toplink.jdbc.read-connections.min" value="1" />
17.         <property name="toplink.jdbc.write-connections.max" value="5" />
18.         <property name="toplink.jdbc.write-connections.min" value="2" />
19.       <!-- SGBD -->
20.       <property name="toplink.target-database" value="MySQL4" />
21.       <!-- serveur d'application -->
22.       <property name="toplink.target-server" value="None" />
23.       <!-- génération schéma -->
24.       <property name="toplink.ddl-generation" value="drop-and-create-tables" />
25.       <property name="toplink.application-location" value="ddl/mysql5" />
26.       <property name="toplink.create-ddl-jdbc-file-name" value="create.sql" />
27.       <property name="toplink.drop-ddl-jdbc-file-name" value="drop.sql" />
28.       <property name="toplink.ddl-generation.output-mode" value="both" />
29.     <!-- logs -->
30.     <property name="toplink.logging.level" value="OFF" />
31.   </properties>
32. </persistence-unit>
33. </persistence>
```

- ligne 3 : n'a pas changé
- ligne 5 : le provider est désormais Toplink. La classe nommée ici sera trouvée dans la bibliothèque [jpa-toplink] ([1] ci-dessous) :



- ligne 7 : la balise <class> sert à nommer toutes les classes @Entity du projet, ici seulement la classe Personne. Hibernate avait une option de configuration qui nous évitait de nommer ces classes. Il explorait le classpath du projet pour y trouver les classes @Entity.
- ligne 9 : la balise <properties> qui introduit des propriétés propres à l'implémentation JPA utilisée, ici Toplink.
- lignes 11-14 : configuration de la liaison Jdbc avec le SGBD MySQL5
- lignes 15-18 : configuration du pool de connexions Jdbc géré nativement par Toplink :
  - lignes 15, 16 : nombre maximum et minimum de connexions dans le pool de connexions en lecture. Défaut (2,2)
  - lignes 17,18 : nombre maximum et minimum de connexions dans le pool de connexions en écriture. Défaut (10,2)
- ligne 20 : le SGBD cible. La liste des SGBD utilisables est disponible dans le paquetage [oracle.toplink.essentials.platform.database] (cf [2] ci-dessus). Le SGBD MySQL5 n'est pas présent dans la liste [2], aussi a-t-on choisi MySQL4. Toplink supporte un peu moins de SGBD qu'Hibernate. Ainsi des sept SGBD utilisés dans nos exemples, Firebird n'est pas supporté. On ne trouve pas non plus Oracle dans la liste. Il est en fait dans un autre paquetage ([3] ci-dessus). Si dans ces deux paquetages, le SGBD cible est désigné par la classe <Sgbdb>Platform.class, la balise s'écrira :

```
<property name="toplink.target-database" value="<Sgbdb>" />
```

- ligne 22 : fixe le serveur d'application si l'application s'exécute dans un tel serveur. Les valeurs possibles actuelles (None, OC4J\_10\_1\_3, SunAS9). Défaut (None).
- lignes 24-28 : lorsque la couche JPA s'initialisera, on lui demande de faire un nettoyage de la base de données définie par la liaison Jdbc des lignes 11-14. On partira ainsi d'une base vide.
  - ligne 24 : on demande à Toplink de faire un drop suivi d'un create des tables du schéma de la base de données
  - ligne 25 : on va demander à Toplink de générer les scripts SQL des opérations drop et create. application-location fixe le dossier dans lequel seront générés ces scripts. Défaut : (dossier courant).
  - ligne 26 : nom du script SQL des opérations create. Défaut : createDDL\_jdbc.
  - ligne 27 : nom du script SQL des opérations drop. Défaut : dropDDL\_jdbc.
  - ligne 28 : mode de génération du schéma (Défaut : both) :
    - both : scripts et base de données
    - database : base de données seulement
    - sql-script : scripts seulement
- ligne 30 : on inhibe (OFF) les logs de Toplink. Les différents niveaux de login disponibles sont les suivants : OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST. Défaut : INFO.

On consultera l'url [<http://www.oracle.com/technology/products/ias/toplink/JPA/essentials/toplink-jpa-extensions.html>] pour une définition exhaustive des balises <property> utilisables avec Toplink.

### 2.1.15.3 Test [InitDB]

Il n'y a rien de plus à faire. Nous sommes prêts à exécuter le premier test [InitDB] :

- lancer le SGBD, ici MySQL5
- exécuter [InitDB]

1. Console output showing the results of the JPA/Hibernate execution:

```
<terminated> InitDB [Java Application] C:\Program Files\Java\jre1.6.0_20\bin>
[personnes]
[3,1,Durant,Sylvie,05/07/2001,false,0]
[2,1,Martin,Paul,31/01/2000,true,2]
terminé ...
```

2. Connections perspective showing the mysql5-jpa connection (highlighted with a red box).

3. Database Structure view showing the MySQL [v5.0] schema with tables jpa01\_personne and sequence.

- en [1] : l'affichage console. On retrouve les résultats déjà obtenus avec JPA / Hibernate.
- en [3] : on ouvre la perspective [SQL Explorer] puis on ouvre la connexion [mysql5-jpa]
- en [4] : l'arborescence de la base **jpa**. On découvre que l'exécution de [InitDB] a créé deux tables : [jpa01\_personne] qui était attendue et la table [sequence] qui l'était moins.

5. Database Detail view showing the structure of the jpa01\_personne table:

COLUMN_NAME	DATA_TYPE	TYPE_NAME	COLUMN_SIZE
ID	4	int	10
PRENOM	12	varchar	30
DATENAISANCE	91	date	<null>
NOM	12	varchar	30
MARIE	-7	BIT	<null>
VERSION	4	int	10
NBENFANTS	4	int	10

6. Database Detail view showing the content of the jpa01\_personne table:

ID	PRENOM	DATENAISANCE	NOM	MARIE	VERSION	NBENFANTS
2	Paul	2000-01-31 00:00:00	Martin	1	1	2
3	Sylvie	2001-07-05 00:00:00	Durant	0	1	0

7. Database Detail view showing the structure of the sequence table:

COLUMN_NAME	DATA_TYPE	TYPE_NAME	COLUMN_SIZE
SEQ_NAME	12	varchar	50
SEQ_COUNT	3	decimal	38

8. Database Detail view showing the content of the sequence table:

SEQ_NAME	SEQ_COUNT
SEQ_GEN	51

- en [5] : la structure de la table [jpa01\_personne] et en [6] son contenu
- en [7] : la structure de la table [sequence] et en [8] son contenu.

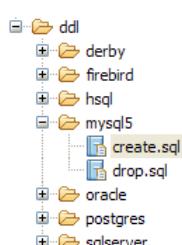
Le fichier de configuration [persistence.xml] demandait la génération des scripts de la DDL :

```

1.      <!-- génération schéma -->
2.      <property name="toplink.ddl-generation" value="drop-and-create-tables" />
3.      <property name="toplink.application-location" value="ddl/mysql5" />
4.      <property name="toplink.create-ddl-jdbc-file-name" value="create.sql" />
5.      <property name="toplink.drop-ddl-jdbc-file-name" value="drop.sql" />
6.      <property name="toplink.ddl-generation.output-mode" value="both" />

```

Regardons ce qui a été généré dans le dossier [ddl/mysql5] :



### create.sql

```

1. CREATE TABLE jpa01_personne (ID INTEGER NOT NULL, PRENOM VARCHAR(30) NOT NULL, DATENAISANCE DATE
NOT NULL, NOM VARCHAR(30) UNIQUE NOT NULL, MARIE TINYINT(1) default 0 NOT NULL, VERSION INTEGER
NOT NULL, NBENFANTS INTEGER NOT NULL, PRIMARY KEY (ID))
2. CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(38), PRIMARY KEY
(SEQ_NAME))
3. INSERT INTO SEQUENCE(SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 1)

```

- lignes 1 : la DDL de la table [jpa01\_personne]. On constate que Toplink n'a pas utilisé l'attribut **autoincrement** pour la clé primaire ID. Ce qui fait qu'on n'a pas une incrémentation automatique de celle-ci lors des insertions de lignes.
- ligne 2 : la DDL de la table [sequence]. Son nom semble indiquer que Toplink utilise cette table pour générer les valeurs de la clé primaire ID.
- ligne 3 : insertion d'une unique ligne dans [SEQUENCE]

### drop.sql

```
1. DROP TABLE jpa01_personne
2. DELETE FROM SEQUENCE WHERE SEQ_NAME = 'SEQ_GEN'
```

- ligne 1 : suppression de la table [jpa01\_personne]
- ligne 2 : suppression d'une ligne particulière de la table [SEQUENCE]. La table elle-même n'est pas supprimée ni les autres lignes éventuelles qu'elle pourrait contenir.

Pour en savoir plus sur le rôle de la table [SEQUENCE], on active dans [persistence.xml], les logs de Toplink au niveau FINE, un niveau qui trace les ordres SQL émis par Toplink :

```
1.      <!-- logs -->
2.  <property name="toplink.logging.level" value="FINE" />
```

On réexécute InitDB. Ci-dessous, on n'a conservé qu'une vue partielle de l'affichage console :

```
1. ...
2. [TopLink Config]: 2007.05.28 12:07:52.796--ServerSession(12910198)--Connection(30708295)--
   Thread(Thread[main,5,main])--Connected: jdbc:mysql://localhost:3306/jpa
3.   User: jpa@localhost
4.   Database: MySQL Version: 5.0.37-community-nt
5.   Driver: MySQL-AB JDBC Driver Version: mysql-connector-java-3.1.9 ( $Date: 2005/05/19 15:52:23 $,
   $Revision: 1.1.2.2 $ )
6. ...
7. [TopLink Fine]: 2007.05.28 12:07:53.093--ServerSession(12910198)--Connection(19255406)--
   Thread(Thread[main,5,main])--DROP TABLE jpa01_personne
8. [TopLink Fine]: 2007.05.28 12:07:53.265--ServerSession(12910198)--Connection(30708295)--
   Thread(Thread[main,5,main])--CREATE TABLE jpa01_personne (ID INTEGER NOT NULL, PRENOM VARCHAR(30)
   NOT NULL, DATENAISANCE DATE NOT NULL, NOM VARCHAR(30) UNIQUE NOT NULL, MARIE TINYINT(1) default 0
   NOT NULL, VERSION INTEGER NOT NULL, NBENFANTS INTEGER NOT NULL, PRIMARY KEY (ID))
9. [TopLink Fine]: 2007.05.28 12:07:53.468--ServerSession(12910198)--Connection(19255406)--
   Thread(Thread[main,5,main])--CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT
   DECIMAL(38), PRIMARY KEY (SEQ_NAME))
10. [TopLink Warning]: 2007.05.28 12:07:53.468--ServerSession(12910198)--Thread(Thread[main,5,main])--
    Exception [TOPLINK-4002] (Oracle TopLink Essentials - 2.0 (Build b41-beta2 (03/30/2007))): 
    oracle.toplink.essentials.exceptions.DatabaseException
11. Internal Exception: java.sql.SQLException: Table 'sequence' already exists
12. Error Code: 1050
13. Call: CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(38), PRIMARY KEY
   (SEQ_NAME))
14. Query: DataModifyQuery()
15. [TopLink Fine]: 2007.05.28 12:07:53.468--ServerSession(12910198)--Connection(30708295)--
   Thread(Thread[main,5,main])--DELETE FROM SEQUENCE WHERE SEQ_NAME = 'SEQ_GEN'
16. [TopLink Fine]: 2007.05.28 12:07:53.609--ServerSession(12910198)--Connection(19255406)--
   Thread(Thread[main,5,main])--SELECT * FROM SEQUENCE WHERE SEQ_NAME = 'SEQ_GEN'
17. [TopLink Fine]: 2007.05.28 12:07:53.609--ServerSession(12910198)--Connection(30708295)--
   Thread(Thread[main,5,main])--INSERT INTO SEQUENCE(SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 1)
18. [TopLink Fine]: 2007.05.28 12:07:53.734--ClientSession(15308417)--Connection(14069849)--
   Thread(Thread[main,5,main])--delete from jpa01_personne
19. [TopLink Fine]: 2007.05.28 12:07:53.750--ClientSession(15308417)--Connection(14069849)--
   Thread(Thread[main,5,main])--UPDATE SEQUENCE SET SEQ_COUNT = SEQ_COUNT + ? WHERE SEQ_NAME = ?
20. bind => [50, SEQ_GEN]
21. [TopLink Fine]: 2007.05.28 12:07:53.750--ClientSession(15308417)--Connection(14069849)--
   Thread(Thread[main,5,main])--SELECT SEQ_COUNT FROM SEQUENCE WHERE SEQ_NAME = ?
22. bind => [SEQ_GEN]
23. [personnes]
24. [TopLink Fine]: 2007.05.28 12:07:53.906--ClientSession(15308417)--Connection(14069849)--
   Thread(Thread[main,5,main])--INSERT INTO jpa01_personne (ID, PRENOM, DATENAISANCE, NOM, MARIE,
   VERSION, NBENFANTS) VALUES (?, ?, ?, ?, ?, ?, ?)
25. bind => [3, Sylvie, 2001-07-05, Durant, false, 1, 0]
26. [TopLink Fine]: 2007.05.28 12:07:53.921--ClientSession(15308417)--Connection(14069849)--
   Thread(Thread[main,5,main])--INSERT INTO jpa01_personne (ID, PRENOM, DATENAISANCE, NOM, MARIE,
   VERSION, NBENFANTS) VALUES (?, ?, ?, ?, ?, ?, ?)
27. bind => [2, Paul, 2000-01-31, Martin, true, 1, 2]
28. [TopLink Fine]: 2007.05.28 12:07:53.937--ClientSession(15308417)--Connection(14069849)--
   Thread(Thread[main,5,main])--SELECT ID, PRENOM, DATENAISANCE, NOM, MARIE, VERSION, NBENFANTS FROM
   jpa01_personne ORDER BY NOM ASC
29. [3,1,Durant,Sylvie,05/07/2001,false,0]
30. [2,1,Martin,Paul,31/01/2000,true,2]
31. [TopLink Config]: 2007.05.28 12:07:54.062--ServerSession(12910198)--Connection(30708295)--
   Thread(Thread[main,5,main])--disconnect
```

```

32. [TopLink Info]: 2007.05.28 12:07:54.062--ServerSession(12910198)--Thread(Thread[main,5,main])--
  file:/C:/data/2006-2007/eclipse/dvp-jpa/toplink/direct/personnes-entites/bin/-jpa logout
  successful
33. ...
34. terminé ...

```

- lignes 2-5 : une connexion au SGBD avec ses paramètres. En fait, les logs montrent qu'en réalité Toplink crée 3 connexions avec le SGBD. Il faudrait voir si ce nombre est relié à l'une des valeurs de configuration utilisées pour le pool de connexions Jdbc :

```

1.      <property name="toplink.jdbc.read-connections.max" value="3" />
2.      <property name="toplink.jdbc.read-connections.min" value="1" />
3.      <property name="toplink.jdbc.write-connections.max" value="5" />
4.  <property name="toplink.jdbc.write-connections.min" value="2" />

```

- ligne 7 : suppression de la table [jpa01\_personne]. Normal, puisque le fichier [persistence.xml] demande le nettoyage de la base **jpa**.
- ligne 8 : création de la table [jpa01\_personne]. On constate que la clé primaire ID n'a pas l'attribut *autoincrement*.
- ligne 9 : création de la table [SEQUENCE] qui existe déjà, créée lors de la précédente exécution.
- lignes 10-13 : Toplink signale l'erreur de création de la table [SEQUENCE].
- lignes 15-18 : Toplink nettoie la table [SEQUENCE]. A l'issue de ce nettoyage, la table [SEQUENCE] a une ligne (SEQ\_NAME, SEQ\_COUNT) avec les valeurs ('SEQ\_GEN', 1).
- ligne 18 : la table [jpa01\_personne] est vidée.
- lignes 19-20 : Toplink passe l'unique ligne où SEQ\_NAME='SEQ\_GEN' de la table [SEQUENCE], de la valeur ('SEQ\_GEN', 1) à la valeur ('SEQ\_GEN', 51)
- ligne 21 : Toplink récupère la valeur 51 de la ligne ('SEQ\_GEN', 51) de la table [SEQUENCE].
- lignes 24-27 : Toplink insère dans la table [jpa01\_personne] les deux personnes 'Martin' et 'Durant'. Il y a un mystère ici : les clés primaires de ces deux lignes reçoivent les valeurs 2 et 3 sans qu'on sache comment ont été obtenues ces valeurs. On ne sait pas si la valeur SEQ\_COUNT (51) obtenue ligne 21 a servi à quelque chose. On notera que la valeur de la version des lignes est 1, alors qu'Hibernate commençait à 0.
- ligne 28 : Toplink fait le SELECT pour obtenir toutes les lignes de la table [jpa01\_personne]
- lignes 29-30 : lignes affichées par le client Java
- lignes 31-32 : Toplink ferme une connexion. Il va répéter l'opération pour chacune des connexions ouvertes initialement.

Au final, on ne connaît pas exactement le rôle de la table [SEQUENCE] mais il semble quand même qu'elle joue un rôle dans la génération des valeurs de la clé primaire ID. En prenant le niveau de logs le plus fin, FINEST, on en apprend un peu plus sur le rôle de la table [SEQUENCE].

```

<!-- logs -->
<property name="toplink.logging.level" value="FINEST" />

```

Nous n'avons gardé ci-dessous que les logs concernant l'insertion des deux personnes dans la table. C'est là qu'on voit le mécanisme de génération des valeurs de la clé primaire :

```

1.[TopLink Finest]: 2007.05.28 03:05:04.046--ClientSession(30617157)--Thread(Thread[main,5,main])--
  Execute query ValueReadQuery()
2.[TopLink Fine]: 2007.05.28 03:05:04.046--ClientSession(30617157)--Connection(13301441)--
  Thread(Thread[main,5,main])--SELECT SEQ_COUNT FROM SEQUENCE WHERE SEQ_NAME = ?
3.bind => [SEQ_GEN]
4.[TopLink Finest]: 2007.05.28 03:05:04.062--ClientSession(30617157)--Connection(13301441)--
  Thread(Thread[main,5,main])--local sequencing preallocation for SEQ_GEN: objects: 50 , first: 2, last:
  51
5.[TopLink Finest]: 2007.05.28 03:05:04.062--UnitOfWork(19864560)--Thread(Thread[main,5,main])--assign
  sequence to the object (2 -> [null,0,Martin,Paul,31/01/2000,true,2])
6.[TopLink Finest]: 2007.05.28 03:05:04.062--UnitOfWork(19864560)--Thread(Thread[main,5,main])--Execute
  query DoesExistQuery()
7.[TopLink Finest]: 2007.05.28 03:05:04.062--UnitOfWork(19864560)--Thread(Thread[main,5,main])--PERSIST
  operation called on: [null,0,Durant,Sylvie,05/07/2001,false,0].
8.[TopLink Finest]: 2007.05.28 03:05:04.062--UnitOfWork(19864560)--Thread(Thread[main,5,main])--assign
  sequence to the object (3 -> [null,0,Durant,Sylvie,05/07/2001,false,0])
9.[personnes]
10.[TopLink Finest]: 2007.05.28 03:05:04.203--UnitOfWork(19864560)--Thread(Thread[main,5,main])--Execute
    query InsertObjectQuery([3,0,Durant,Sylvie,05/07/2001,false,0])
11.[TopLink Finest]: 2007.05.28 03:05:04.203--UnitOfWork(19864560)--Thread(Thread[main,5,main])--Assign
    return row DatabaseRecord(
12.  jpa01_personne.VERSION => 1)
13.[TopLink Fine]: 2007.05.28 03:05:04.203--ClientSession(30617157)--Connection(13301441)--
  Thread(Thread[main,5,main])--INSERT INTO jpa01_personne (ID, PRENOM, DATENAISANCE, NOM, MARIE,
  VERSION, NBENFANTS) VALUES (?, ?, ?, ?, ?, ?, ?)
14. bind => [3, Sylvie, 2001-07-05, Durant, false, 1, 0]
15.[TopLink Finest]: 2007.05.28 03:05:04.203--UnitOfWork(19864560)--Thread(Thread[main,5,main])--Execute
  query InsertObjectQuery([2,0,Martin,Paul,31/01/2000,true,2])
16.[TopLink Finest]: 2007.05.28 03:05:04.203--UnitOfWork(19864560)--Thread(Thread[main,5,main])--Assign
    return row DatabaseRecord(
17.  jpa01_personne.VERSION => 1)

```

```

18.[TopLink Fine]: 2007.05.28 03:05:04.203--ClientSession(30617157)--Connection(13301441)--
Thread(Thread[main,5,main])--INSERT INTO jpa01_personne (ID, PRENOM, DATENAISSANCE, NOM, MARIE,
VERSION, NBENFANTS) VALUES (?, ?, ?, ?, ?, ?, ?)
19.bind => [2, Paul, 2000-01-31, Martin, true, 1, 2]

```

- ligne 4 : on voit que le nombre 51 récupéré dans la table [SEQUENCE] à la ligne 2 sert à délimiter un intervalle de valeurs pour la clé primaire : [2,51]
- ligne 5 : la première personne reçoit la valeur 2 pour clé primaire
- ligne 8 : la seconde personne reçoit la valeur 3 pour clé primaire
- ligne 12 : montre la gestion de version de la première personne
- ligne 17 : idem pour la seconde personne

Le niveau de logs [FINEST] montre également les limites des transactions émises par Toplink. L'étude de ces logs montre ce que fait Toplink et c'est un grand moyen de comprendre le pont objet / relationnel.

On retiendra de ce qui précède :

- que des implémentations JPA différentes vont générer des schémas de bases de données différents. Dans cet exemple, Hibernate et Toplink n'ont pas généré les mêmes schémas.
- que les niveaux de logs FINE, FINER, FINEST de Toplink seront à utiliser dès qu'on souhaitera des éclaircissements sur ce que fait exactement Toplink.

#### 2.1.15.4 Test [Main]

Nous exécutons maintenant le test [Main] :

The screenshot shows the Eclipse IDE interface. On the left, the 'Console' tab is selected, displaying Java application logs. Annotations are present: a yellow box labeled '1' highlights the line 'main : ----- test10'. A yellow box labeled '2' highlights the line 'main : ----- test11'. A yellow box labeled '3' highlights the line 'try {'. On the right, the code editor shows Java code with corresponding line numbers 372 through 378. Annotations are also present: a yellow box labeled '1' highlights the line 'System.out.format("Erreur dans transaction [%s,%s,%s,%s,%s,%s]%n", e1.getClassName(), e1.getMessage(), e1.getCause().getClassName(), e1.getCause().getMessage(), e1.getCause().getClassName(), e1.getCause().getCause().getMessage());'. A yellow box labeled '2' highlights the line 'if (tx.isActive())'. A yellow box labeled '3' highlights the line 'try {'. The code is as follows:

```

1. } catch (RuntimeException e1) {
2.     // on a eu un pb
3.     System.out.format("Erreur dans transaction [%s,%s,%s,%s,%s,%s]%n", e1.getClassName(),
e1.getMessage(),
4.         e1.getCause().getClassName(), e1.getCause().getMessage(),
e1.getCause().getClassName(), e1.getCause().getClassName(),
e1.getCause().getCause().getMessage());
5.     try {
6.         ...
7.     }

```

- en [1] : tous les tests passent sauf le test 11 [2]
- en [3] : ligne 376, la ligne de code où s'est produite l'exception

Le code qui produit l'exception est le suivant :

```

1. } catch (RuntimeException e1) {
2.     // on a eu un pb
3.     System.out.format("Erreur dans transaction [%s,%s,%s,%s,%s,%s]%n", e1.getClassName(),
e1.getMessage(),
4.         e1.getCause().getClassName(), e1.getCause().getMessage(),
e1.getCause().getClassName(), e1.getCause().getClassName(),
e1.getCause().getCause().getMessage());
5.     try {
6.         ...
7.     }

```

- ligne [3] : la ligne de l'exception. On a un *NullPointerException*, ce qui laisse penser que l'une des méthodes *getCause* des lignes 4 et 5 a rendu un pointeur *null*. Une expression telle que [e1.getCause().getCause()] suppose que la chaîne des exceptions a 3 éléments [e1.getCause().getCause(), e1.getCause(), e1]. Si elle n'en a que deux, la première expression causera une exception.

Nous changeons le code précédent pour qu'il n'affiche que les deux dernières exceptions de la chaîne des exceptions :

```

1.     } catch (RuntimeException e1) {
2.         // on a eu un pb
3.         System.out.format("Erreur dans transaction [%s,%s,%s,%s,%s,%s]%n", e1.getClassName(),
e1.getMessage(),
4.             e1.getCause().getClassName(), e1.getCause().getMessage());
5.         try {

```

A l'exécution, on a alors le résultat suivant :

```

1. ...
2. [personnes]
3. [2,5,Martin,Paul,31/01/2000,false,6]
4. main : ----- test11
5. [personnes]
6. Erreur dans transaction [javax.persistence.OptimisticLockException,Exception] [TOPLINK-5006]
   (Oracle TopLink Essentials - 2.0 (Build b41-beta2 (03/30/2007))):
   oracle.toplink.essentials.exceptions.OptimisticLockException
7. Exception Description: The object [[2,6,Martin,Paul,31/01/2000,false,7]] cannot be updated because
   it has changed or been deleted since it was last read.
8. Class> entites.Personne Primary Key>
   [2],oracle.toplink.essentials.exceptions.OptimisticLockException,
9. Exception Description: The object [[2,6,Martin,Paul,31/01/2000,false,7]] cannot be updated because
   it has changed or been deleted since it was last read.
10. Class> entites.Personne Primary Key> [2,]
11. [personnes]
12. [2,5,Martin,Paul,31/01/2000,false,6]

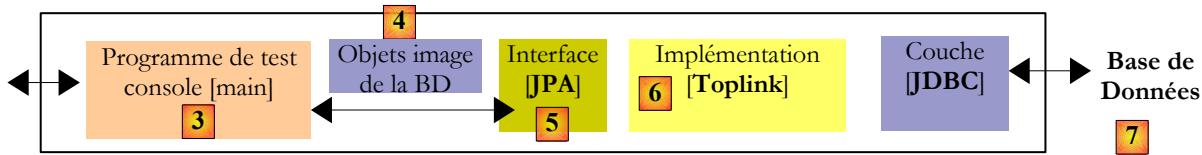
```

Cette fois-ci, le test 11 passe. Les affichages sur l'exception (lignes 6-10) ont été demandés par le code Java (ligne 3 du code plus haut). On rappelle que le test 11 enchaînait, dans une même transaction, plusieurs opérations SQL dont l'une échouait et devait entraîner un rollback de la transaction. Les états de la table [jpa01\_personne] avant (ligne 3) et après le test (ligne 12) sont bien identiques montrant que le rollback a eu lieu.

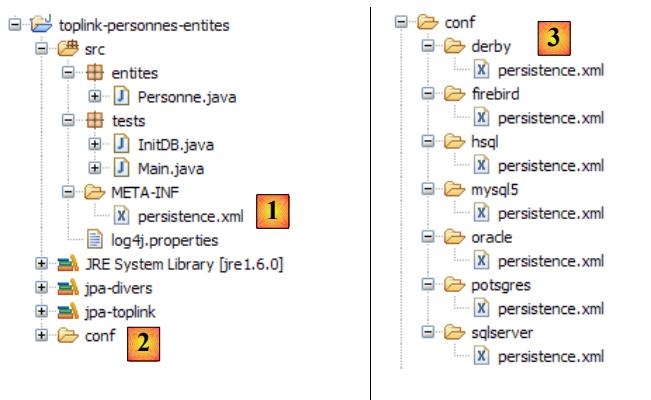
On notera ici un **point important** : les implémentations JPA / Hibernate et JPA / Toplink ne sont pas interchangeables à 100%. Dans cet exemple, il nous faut changer le code du client JPA pour éviter un *NullPointerException*. Nous retrouverons ce problème ultérieurement et de nouveau dans le cadre d'une exception.

## 2.1.16 Changer de SGBD dans l'implémentation JPA / Toplink

Revenons sur l'architecture de test de notre projet actuel :



Précédemment, le SGBD utilisé en [7] était MySQL5. Nous montrons avec Oracle comment changer de SGBD. Dans tous les cas, la modification à faire dans le projet Eclipse est simple (cf ci-dessous) : remplacer le fichier **persistence.xml** [1] de configuration de la couche JPA par l'un de ceux du dossier *conf* ([2] et [3]) du projet.



### 2.1.16.1 Oracle 10g Express

Oracle 10g Express est présenté en Annexes au paragraphe 5.7, page 276. Le fichier *persistence.xml* d'Oracle pour Toplink est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">

```

```

3. <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.   <!-- provider -->
5.   <provider>oracle.toplink.essentials.PersistenceProvider</provider>
6.   <!-- classes persistantes -->
7.   <class>entites.Personne</class>
8.   <!-- propriétés de l'unité de persistance -->
9.   <properties>
10.     <!-- connexion JDBC -->
11.     <property name="toplink.jdbc.driver" value="oracle.jdbc.OracleDriver" />
12.     <property name="toplink.jdbc.url" value="jdbc:oracle:thin:@localhost:1521:xe" />
13.     <property name="toplink.jdbc.user" value="jpa" />
14.     <property name="toplink.jdbc.password" value="jpa" />
15.     <property name="toplink.jdbc.read-connections.max" value="3" />
16.     <property name="toplink.jdbc.read-connections.min" value="1" />
17.     <property name="toplink.jdbc.write-connections.max" value="5" />
18.     <property name="toplink.jdbc.write-connections.min" value="2" />
19.     <!-- SGBD -->
20.     <property name="toplink.target-database" value="Oracle" />
21.     <!-- servir d'application -->
22.     <property name="toplink.target-server" value="None" />
23.     <!-- génération schéma -->
24.     <property name="toplink.ddl-generation" value="drop-and-create-tables" />
25.     <property name="toplink.application-location" value="ddl/oracle" />
26.     <property name="toplink.create-ddl-jdbc-file-name" value="create.sql" />
27.     <property name="toplink.drop-ddl-jdbc-file-name" value="drop.sql" />
28.     <property name="toplink.ddl-generation.output-mode" value="both" />
29.     <!-- logs -->
30.     <property name="toplink.logging.level" value="OFF" />
31.   </properties>
32. </persistence-unit>
33. </persistence>

```

Cette configuration est identique à celle faite pour le SGBD MySQL5, aux détails près suivants :

- lignes 11-14 qui configurent la liaison JDBC avec la base de données
- ligne 20 : qui fixe le SGBD cible
- ligne 25 : qui fixe le dossier de génération des scripts SQL de la DDL

Pour exécuter le test [InitDB] :

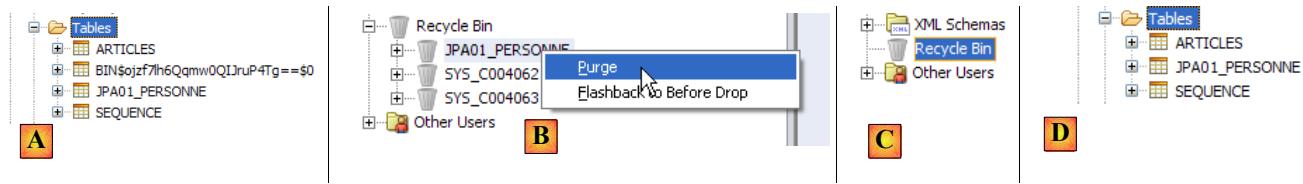
- lancer le SGBD Oracle
- mettre conf/oracle/persistence.xml dans META-INF/persistence.xml
- exécuter l'application [InitDB]

On obtient les résultats suivants sur la console et dans la perspective [SQL Explorer] :

The screenshot shows the Oracle Database perspective in a Java IDE. It includes several windows:

- Console Window (1):** Displays the output of the [InitDB] application, showing the creation of a Personne table with columns ID, PRENOM, DATENAISANCE, NOM, MARIE, VERSION, and NBENFANTS.
- Connections Window (2):** Shows a list of database connections, with 'oracle-jpa (1 session)' highlighted.
- Database Detail Window (3):** Shows the structure of the Personne table.
- Database Detail Window (4):** Shows the structure of a sequence table named SEQ\_GEN.
- Database Structure Window (5):** Shows the tree view of the database schema, including JPA, Functions, Java Sources, Materialized Views, Packages, Procedures, Sequences, Synonyms, and Tables (ARTICLES, JPA01\_PERSONNE, SEQUENCE).
- Database Detail Window (6):** Shows the data for the Personne table.
- Database Detail Window (7):** Shows the data for the SEQ\_GEN sequence table.
- Database Structure Window (8):** Shows the data for the SEQ\_GEN sequence table.

- [1] : l'affichage console
- [2] : la connexion [oracle-jpa] dans SQL Explorer
- [3] : la base de données jpa
- [4] : InitDB a créé deux tables : JPA01\_PERSONNE et SEQUENCE, comme avec MySQL5. Parfois en [4], on voit apparaître des tables [BIN\*]. Elles correspondent à des tables détruites. Pour voir le phénomène, il suffit de réexécuter [InitDB]. La phase d'initialisation de la couche JPA comporte un nettoyage de la base de données *jpa* au cours de laquelle la table [JPA01\_PERSONNE] est détruite :



En [A], on voit apparaître une table [BIN]. Oracle ne supprime pas définitivement une table ayant subi un *drop* mais la met dans une corbeille [Recycle Bin]. Cette corbeille est visible [B] avec l'outil SQL Developer décrit au paragraphe 5.7.4, page 280. En [B], on peut purger la table [JPA01\_PERSONNE] qui est dans la corbeille. Cela vide la corbeille [C]. Si dans SQL Explorer, on rafraîchit (clic droit / Refresh) les tables, on voit que la table BIN n'est plus là [D].

- [5, 6] : la structure et le contenu de la table [JPA01\_PERSONNE]
- [7, 8] : la structure et le contenu de la table [SEQUENCE]

Voilà ! Le lecteur est maintenant invité à exécuter l'application [Main] sur Oracle.

### 2.1.16.2 Les autres SGBD

Nous montrerons peu sur les autres SGBD. Il faut simplement reproduire la procédure suivie pour Oracle. On notera les points suivants :

- quelque soit le SGBD, Toplink utilise toujours la même technique pour la génération des valeurs de la clé primaire ID de la table [JPA01\_PERSONNE] : il utilise la table [SEQUENCE] détaillée plus haut.
- Toplink ne reconnaît pas le SGBD Firebird. Il existe une base de données générique pour ces cas là :

```
<property name="toplink.target-database" value="Auto" />
```

Avec cette base générique appelée [Auto], les tests avec Firebird échouent sur des erreurs de syntaxe SQL. Toplink utilise pour la clé primaire ID, un type SQL *Number(10)* que ne reconnaît pas Firebird. Il faut alors choisir un SGBD ayant les mêmes types SQL que Firebird (pour cet exemple). C'est le cas d'Apache Derby :

```
1.      <!-- connexion JDBC -->
2.      <property name="toplink.jdbc.driver" value="org.firebirdsql.jdbc.FBDriver" />
3. ...
4.      <!-- SGBD -->
5.      <!--
6.      TopLink ne reconnaît pas Firebird pour l'instant (05/07). Derby convient pour remplacer.
7.      -->
8.      <property name="toplink.target-database" value="Derby" />
9. ...
```

- Toplink ne sait pas générer le schéma original de la base pour le SGBD HSQLDB. C'est à dire que la directive :

```
1.      <!-- génération schéma -->
2.      <property name="toplink.ddl-generation" value="drop-and-create-tables" />
```

échoue pour HSQLDB. La cause en est une erreur de syntaxe à la création de la table [jpa01\_personne] :

```
1. [TopLink Fine]: 2007.05.29 09:44:18.515--ServerSession(12910198)--Connection(29775659)--
Thread(Thread[main,5,main])--DROP TABLE jpa01_personne
2. [TopLink Fine]: 2007.05.29 09:44:18.531--ServerSession(12910198)--Connection(29775659)--
Thread(Thread[main,5,main])--CREATE TABLE jpa01_personne (ID INTEGER NOT NULL, PRENOM VARCHAR(30)
NOT NULL, DATENAISANCE DATE NOT NULL, NOM VARCHAR(30) UNIQUE NOT NULL, MARIE TINYINT NOT NULL,
VERSION INTEGER NOT NULL, NBENFANTS INTEGER NOT NULL, PRIMARY KEY (ID))
3. [TopLink Warning]: 2007.05.29 09:44:18.531--ServerSession(12910198)--Thread(Thread[main,5,main])--_
Exception [TOPLINK-4002] (Oracle TopLink Essentials - 2.0 (Build b41-beta2 (03/30/2007))):_
oracle.toplink.exceptions.DatabaseException
4. Internal Exception: java.sql.SQLException: Unexpected token: UNIQUE in statement [CREATE TABLE
jpa01_personne (ID INTEGER NOT NULL, PRENOM VARCHAR(30) NOT NULL, DATENAISANCE DATE NOT NULL, NOM
VARCHAR(30) UNIQUE]
```

Ligne 4, la syntaxe NOM VARCHAR(30) UNIQUE NOT NULL n'est pas acceptée par HSQL. Hibernate avait utilisé la syntaxe : NOM VARCHAR(30) NOT NULL, UNIQUE(NOM).

De façon générale, Hibernate a été plus efficace que Toplink pour reconnaître les SGBD avec lesquels les tests de ce document ont été faits.

## 2.1.17 Conclusion

L'étude de l'@Entity [Personne] s'arrête là. Du point de vue conceptuel, assez peu a été fait : nous avons étudié le pont objet / relationnel dans un cas le plus simple : un objet @Entity <--> une table. Son étude nous a cependant permis de présenter les outils que nous utiliserons dans tout le document. Cela nous permettra d'aller un peu plus vite dorénavant dans l'étude des autres cas du pont objet / relationnel que nous allons étudier :

- à l'@Entity [Personne] précédente, on va ajouter un champ *adresse* modélisé par une classe [Adresse]. Du côté base de données, nous verrons deux implémentations possibles. Les objets [Personne] et [Adresse] donnent naissance à
  - une unique table [personne] incluant l'adresse
  - deux tables [personne] et [adresse] liées par une relation de clé étrangère de type **un-à-un**.
- un exemple de relation **un-à-plusieurs** où une table [article] est liée à une table [categorie] par une clé étrangère
- un exemple de relation **plusieurs-à-plusieurs** où deux tables [personne] et [activite] sont reliées par une table de jointure [personne\_activite].

## 2.2 Exemple 2 : relation un-à-un via une inclusion

### 2.2.1 Le schéma de la base de données

jpa02_personne	
id	BIGINT(19)
version	INT(10)
nom	VARCHAR(30)
prenom	VARCHAR(30)
datenaissance	DATE
marie	BIT
nbenfants	INT(10)
adr1	VARCHAR(30)
adr2	VARCHAR(30)
adr3	VARCHAR(30)
codePostal	VARCHAR(5)
ville	VARCHAR(20)
cedex	VARCHAR(3)
pays	VARCHAR(20)

```
1. drop table if exists jpa02_personne;
2.
3.
4. create table jpa02_personne (
5.     id bigint not null auto_increment,
6.     version integer not null,
7.     nom varchar(30) not null unique,
8.     prenom varchar(30) not null,
9.     datenaissance date not null,
10.    marie bit not null,
11.    nbenfants integer not null,
12.    adr1 varchar(30) not null,
13.    adr2 varchar(30),
14.    adr3 varchar(30),
15.    codePostal varchar(5) not null,
16.    ville varchar(20) not null,
17.    cedex varchar(3),
18.    pays varchar(20) not null,
19.    primary key (id)
20.) ENGINE=InnoDB;
```

- en [1] : la base de données (plugin Azurri Clay)
- en [2] : la DDL générée par Hibernate pour MySQL5

La table [jpa02\_personne] est la table [jpa01\_personne] étudiée précédemment à laquelle on a rajouté une adresse (lignes 12-18 de la DDL).

### 2.2.2 Les objets @Entity représentant la base de données

L'adresse d'une personne sera représentée par la classe [Adresse] suivante :

```
1. package entites;
2.
3. ...
4. @SuppressWarnings("serial")
5. @Embeddable
6. public class Adresse implements Serializable {
7.
8.     // champs
9.     @Column(length = 30, nullable = false)
10.    private String adr1;
11.
12.    @Column(length = 30)
13.    private String adr2;
14.
15.    @Column(length = 30)
16.    private String adr3;
17.
18.    @Column(length = 5, nullable = false)
19.    private String codePostal;
20.
21.    @Column(length = 20, nullable = false)
22.    private String ville;
23.
24.    @Column(length = 3)
25.    private String cedex;
26.
27.    @Column(length = 20, nullable = false)
28.    private String pays;
29.
30.    // constructeurs
31.    public Adresse() {
32.
33.    }
34.
35.    public Adresse(String adr1, String adr2, String adr3, String codePostal, String ville, String
36.        cedex, String pays) {
37.    }
```

```

38.
39. // getters et setters
40. ...
41.
42. // toString
43. public String toString() {
44.     return String.format("A[%s,%s,%s,%s,%s,%s]", getAdr1(), getAdr2(), getAdr3(),
45.         getCodePostal(), getVille(), getCedex(), getPays());
46. }

```

- la principale innovation réside dans l'annotation `@Embeddable` de la ligne 5. La classe [Adresse] n'est pas destinée à donner naissance à une table, aussi n'a-t-elle pas l'annotation `@Entity`. L'annotation `@Embeddable` indique que la classe a vocation à être intégrée dans un objet `@Entity` et donc dans la table associée à celui-ci. C'est pourquoi, dans le schéma de la base de données, la classe [Adresse] n'apparaît pas comme une table à part, mais comme faisant partie de la table associée à l'`@Entity` [Personne].

L'`@Entity` [Personne] évolue peu par rapport à sa version précédente : on lui ajoute simplement un champ *adresse* :

```

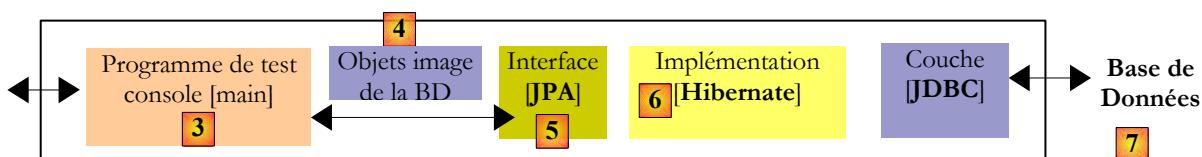
1. package entites;
2.
3. ...
4. @Entity
5. @Table(name = "jpa02_tb_personne")
6. public class Personne implements Serializable{
7.
8.     @Id
9.     @Column(nullable = false)
10.    @GeneratedValue(strategy = GenerationType.AUTO)
11.    private Long id;
12.
13.    @Column(nullable = false)
14.    @Version
15.    private int version;
16.
17.    @Column(length = 30, nullable = false, unique = true)
18.    private String nom;
19.
20.    @Column(length = 30, nullable = false)
21.    private String prenom;
22.
23.    @Column(nullable = false)
24.    @Temporal(TemporalType.DATE)
25.    private Date datenaissance;
26.
27.    @Column(nullable = false)
28.    private boolean marie;
29.
30.    @Column(nullable = false)
31.    private int nbefants;
32.
33.    @Embedded
34.    private Adresse adresse;
35.
36.    // constructeurs
37.    public Personne() {
38.    }
39. ...
40. }

```

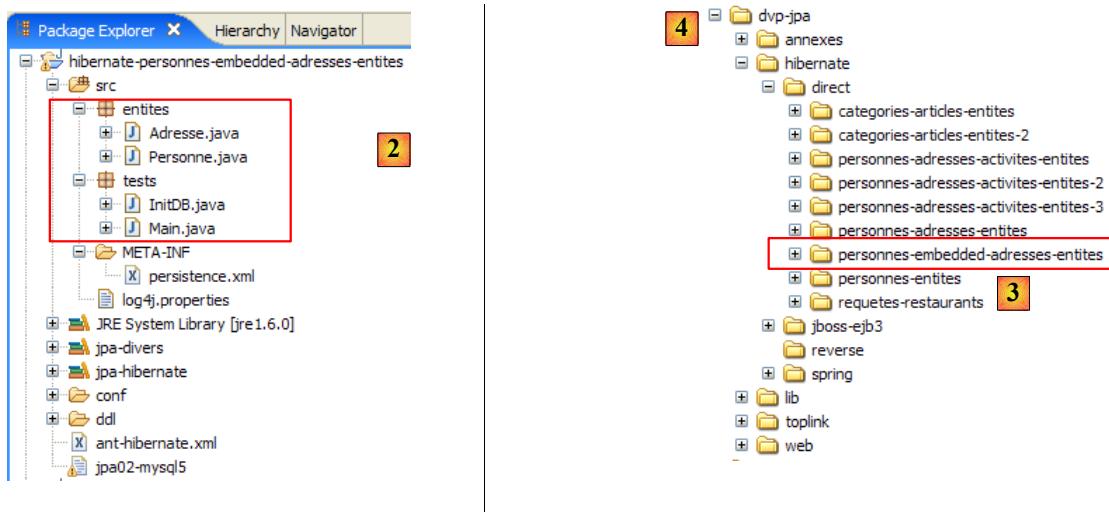
- la modification a lieu lignes 33-34. L'objet [Personne] a désormais un champ *adresse* de type *Adresse*. Ca c'est pour le POJO. L'annotation `@Embedded` est destinée au pont objet / relationnel. Elle indique que le champ [Adresse adresse] devra être encapsulé dans la même table que l'objet [Personne].

### 2.2.3 L'environnement des tests

Nous allons procéder à des tests très semblables à ceux étudiés précédemment. Ils seront faits dans le contexte suivant :



L'implémentation utilisée est JPA / Hibernate [6]. Le projet Eclipse des tests est le suivant :



Le projet Eclipse [1] ne diffère du précédent que par ses codes Java [2]. L'environnement (bibliothèques – persistence.xml – sgbd - dossiers conf, ddl – script ant) est celui déjà étudié précédemment, en particulier au paragraphe 2.1.5, page 14. Ce sera toujours le cas pour les projets Hibernate à venir et, sauf exception, nous ne reviendrons plus sur cet environnement. Notamment, les fichiers *persistence.xml* qui configurent la couche JPA/Hibernate pour différents SGBD sont ceux déjà étudiés et qui se trouvent dans le dossier <conf>.

S'il a un doute sur les procédures à suivre, le lecteur est invité à revenir sur celles suivies dans l'étude précédente.

Le projet Eclipse est présent [3] dans le dossier des exemples [4]. On l'importera.

## 2.2.4 Génération de la DDL de la base de données

En suivant les instructions du paragraphe 2.1.7, page 20, la DDL obtenue pour le SGBD MySQL5 est la suivante :

```

1. drop table if exists jpa02_hb_personne;
2.
3. create table jpa02_hb_personne (
4.     id bigint not null auto_increment,
5.     version integer not null,
6.     nom varchar(30) not null unique,
7.     prenom varchar(30) not null,
8.     datenaissance date not null,
9.     marie bit not null,
10.    nbenfants integer not null,
11.    adr1 varchar(30) not null,
12.    adr2 varchar(30),
13.    adr3 varchar(30),
14.    codePostal varchar(5) not null,
15.    ville varchar(20) not null,
16.    cedex varchar(3),
17.    pays varchar(20) not null,
18.    primary key (id)
19.) ENGINE=InnoDB;
```

Hibernate a correctement reconnu le fait que l'adresse de la personne devait être intégrée dans la table associée à l'`@Entity Personne` (lignes 11-17).

## 2.2.5 InitDB

Le code de [InitDB] est le suivant :

```

1. package tests;
2. ...
3.
4. public class InitDB {
5.
6.     // constantes
7.     private final static String TABLE_NAME = "jpa02_hb_personne";
8.
9.     public static void main(String[] args) throws ParseException {
```

```

10.
11.    // Contexte de persistance
12.    EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
13.    EntityManager em = null;
14.    // on récupère un EntityManager à partir de l'EntityManagerFactory précédent
15.    em = emf.createEntityManager();
16.    // début transaction
17.    EntityTransaction tx = em.getTransaction();
18.    tx.begin();
19.    // requête
20.    Query sql1;
21.    // supprimer les éléments de la table PERSONNE
22.    sql1 = em.createNativeQuery("delete from " + TABLE_NAME);
23.    sql1.executeUpdate();
24.    // création personnes
25.    Personne p1 = new Personne("Martin", "Paul", new
        SimpleDateFormat("dd/MM/yy").parse("31/01/2000"), true, 2);
26.    Personne p2 = new Personne("Durant", "Sylvie", new
        SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
27.    // création adresses
28.    Adresse a1 = new Adresse("8 rue Boileau", null, null, "49000", "Angers", null, "France");
29.    Adresse a2 = new Adresse("Apt 100", "Les Mimosas", "15 av Foch", "49002", "Angers", "03",
        "France");
30.    // associations personne <--> adresse
31.    p1.setAdresse(a1);
32.    p2.setAdresse(a2);
33.    // persistance des personnes
34.    em.persist(p1);
35.    em.persist(p2);
36.    // affichage personnes
37.    System.out.println("[personnes]");
38.    for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList())
{
39.        System.out.println(p);
40.    }
41.    // fin transaction
42.    tx.commit();
43.    // fin EntityManager
44.    em.close();
45.    // fin EntityManagerFactory
46.    emf.close();
47.    // log
48.    System.out.println("terminé...");
49.
50. }
51. }

```

Il n'y a rien de neuf dans ce code. Tout a déjà été rencontré. L'exécution de [InitDB] avec MySQL5 donne les résultats suivants :

The screenshot shows the MySQL Workbench interface. On the left, the 'Database Structure' pane displays the database schema with tables like 'articles', 'jpa02\_tb\_personne', and 'test'. A red box labeled '2' highlights the 'jpa' schema. On the right, the 'SQL Results' pane shows the output of the Java code execution, displaying two rows of Personne objects: P[2,0,Durant,Sylvie,05/07/2001,false,0,A[Apt 100,Les Mimosas,15 av Foch,49002,Angers,03,France]] and P[1,0,Martin,Paul,31/01/2000,true,2,A[8 rue Boileau,null,null,49000,Angers,null,France]]. A red box labeled '1' highlights this output.

The screenshot shows the MySQL Workbench interface. On the left, the 'Database Detail' pane displays the columns and data for the 'articles' table. A red box labeled '3' highlights the 'version' column. On the right, the 'Database Detail' pane displays the columns and data for the 'jpa02\_tb\_personne' table. A red box labeled '4' highlights the first row of data.

id	version	nom	prenom	datenaissance	marie	nbenfants	adr1	adr2	adr3	codePostal	ville	cedex	pays
1	0	Martin	Paul	2000-01-31 00:00:00	1	2	8 rue Boileau	<null>	<null>	49000	Angers	<null>	France
2	0	Durant	Sylvie	2001-07-05 00:00:00	0	0	Apt 100	Les Mimosas	15 av Foch	49002	Angers	03	France

- [1] : l'affichage console
- [2] : la table [jpa02\_hb\_personne] dans la perspective SQL Explorer
- [3] et [4] : sa structure et son contenu.

## 2.2.6 Main

La classe [Main] est la suivante :

```

1. package tests;
2.
3. ...
4. import entites.Adresse;
5. import entites.Personne;
6.
7. @SuppressWarnings( { "unused", "unchecked" })
8. public class Main {
9.
10. // constantes
11. private final static String TABLE_NAME = "jpa02_hb_personne";
12.
13. // Contexte de persistance
14. private static EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
15.
16. private static EntityManager em = null;
17.
18. // objets partagés
19. private static Personne p1, p2, newp1;
20.
21. private static Adresse a1, a2, a3, a4, newa1, newa4;
22.
23. public static void main(String[] args) throws Exception {
24.     // on récupère un EntityManager à partir de l'EntityManagerFactory
25.     em = emf.createEntityManager();
26.
27.     // nettoyage base
28.     log("clean");clean();
29.
30.     // dump table
31.     dumpPersonne();
32.
33.     // test1
34.     log("test1"); test1();
35.
36.     // test2
37.     log("test2"); test2();
38.
39.     // test3
40.     log("test3"); test3();
41.
42.     // test4
43.     log("test4"); test4();
44.
45.     // test5
46.     log("test5");test5();
47.
48.     // fin contexte de persistance
49.     if (em != null && em.isOpen())
50.         em.close();
51.
52.     // fermeture EntityManagerFactory
53.     emf.close();
54. }
55.
56. // récupérer l'EntityManager courant
57. private static EntityManager getEntityManager() {
58. ...
59. }
60.
61. // récupérer un EntityManager neuf
62. private static EntityManager getNewEntityManager() {
63. ...
64. }
65.
66. // affichage contenu table Personne
67. private static void dumpPersonne() {
68. ...
69. }
70.
71. // raz BD
72. private static void clean() {
73. ...
74. }
```

```

75.
76. // logs
77. private static void log(String message) {
78. ...
79. }
80.
81. // création d'objets
82. public static void test1() throws ParseException {
83.     // contexte de persistance
84.     EntityManager em = getEntityManager();
85.     // création personnes
86.     p1 = new Personne("Martin", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
87.         true, 2);
88.     p2 = new Personne("Durant", "Sylvie", new SimpleDateFormat("dd/MM/yy").parse("05/07/2001"),
89.         false, 0);
90.     // création adresses
91.     a1 = new Adresse("8 rue Boileau", null, null, "49000", "Angers", null, "France");
92.     a2 = new Adresse("Apt 100", "Les Mimosas", "15 av Foch", "49002", "Angers", "03", "France");
93.     // associations personne <--> adresse
94.     p1.setAdresse(a1);
95.     p2.setAdresse(a2);
96.     // début transaction
97.     EntityTransaction tx = em.getTransaction();
98.     tx.begin();
99.     // persistance des personnes
100.    em.persist(p1);
101.    em.persist(p2);
102.    // fin transaction
103.    tx.commit();
104.    // dump
105.    dumpPersonne();
106.}
107.// modifier un objet du contexte
108. public static void test2() {
109.     // contexte de persistance
110.     EntityManager em = getEntityManager();
111.     // début transaction
112.     EntityTransaction tx = em.getTransaction();
113.     tx.begin();
114.     // on incrémente le nbre d'enfants de p1
115.     p1.setNbenfants(p1.getNbenfants() + 1);
116.     // on modifie son état marital
117.     p1.setMarie(false);
118.     // l'objet p1 est automatiquement sauvegardé (dirty checking)
119.     // lors de la prochaine synchronisation (commit ou select)
120.     tx.commit();
121.     // on affiche la nouvelle table
122.     dumpPersonne();
123. }
124.
125.// supprimer un objet appartenant au contexte de persistance
126. public static void test4() {
127.     // contexte de persistance
128.     EntityManager em = getEntityManager();
129.     // début transaction
130.     EntityTransaction tx = em.getTransaction();
131.     tx.begin();
132.     // on supprime l'objet attaché p2
133.     em.remove(p2);
134.     // fin transaction
135.     tx.commit();
136.     // on affiche la nouvelle table
137.     dumpPersonne();
138. }
139.
140.// détacher, réattacher et modifier
141. public static void test5() {
142.     // nouveau contexte de persistance
143.     EntityManager em = getNewEntityManager();
144.     // début transaction
145.     EntityTransaction tx = em.getTransaction();
146.     tx.begin();
147.     // on réattache p1 au nouveau contexte
148.     p1 = em.find(Personne.class, p1.getId());
149.     // fin transaction
150.     tx.commit();
151.     // on change l'adresse de p1
152.     p1.getAdresse().setVille("Paris");
153.     // on affiche la nouvelle table
154.     dumpPersonne();
155. }
156.

```

De nouveau, rien qui n'ait déjà été vu. L'affichage console est le suivant :

```

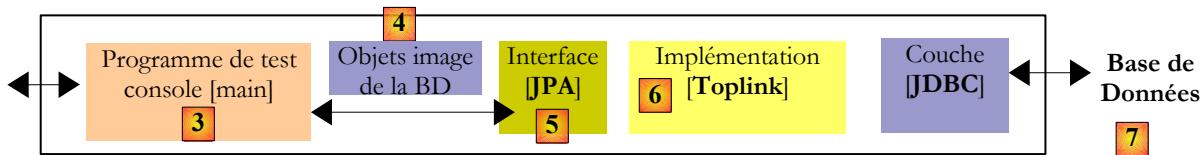
1. main : ----- clean
2. [personnes]
3. main : ----- test1
4. [personnes]
5. P[2,0,Durant,Sylvie,05/07/2001,false,0,A[Apt 100,Les Mimosas,15 av Foch,49002,Angers,03,France]]
6. P[1,0,Martin,Paul,31/01/2000,true,2,A[8 rue Boileau,null,null,49000,Angers,null,France]]
7. main : ----- test2
8. [personnes]
9. P[2,0,Durant,Sylvie,05/07/2001,false,0,A[Apt 100,Les Mimosas,15 av Foch,49002,Angers,03,France]]
10. P[1,1,Martin,Paul,31/01/2000,false,3,A[8 rue Boileau,null,null,49000,Angers,null,France]]
11. main : ----- test4
12. [personnes]
13. P[1,1,Martin,Paul,31/01/2000,false,3,A[8 rue Boileau,null,null,49000,Angers,null,France]]
14. main : ----- test5
15. [personnes]
16. P[1,2,Martin,Paul,31/01/2000,false,3,A[8 rue Boileau,null,null,49000,Paris,null,France]]

```

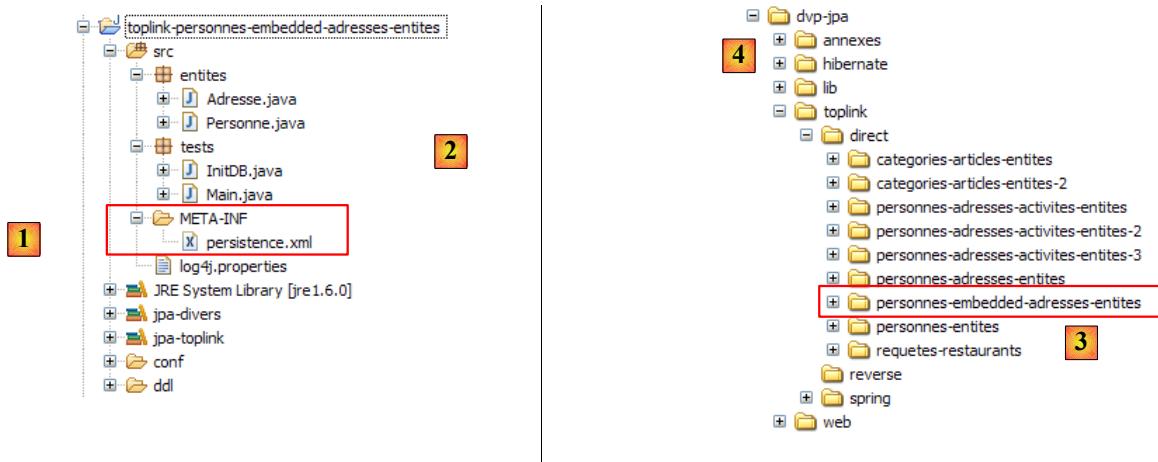
Le lecteur est invité à faire le lien entre les résultats et le code.

## 2.2.7 Implémentation JPA / Toplink

Nous utilisons maintenant une implémentation JPA / Toplink :



Le nouveau projet Eclipse des tests est le suivant :



Les codes Java sont identiques à ceux du projet Hibernate précédent. L'environnement (bibliothèques – persistence.xml – sgbd - dossiers conf, ddl – script ant) est celui déjà étudié au paragraphe 2.1.15.2, page 58. Ce sera toujours le cas pour les projets Toplink à venir et, sauf exception, nous ne reviendrons plus sur cet environnement. Notamment, les fichiers *persistence.xml* qui configurent la couche JPA/Toplink pour différents SGBD sont ceux déjà étudiés et qui se trouvent dans le dossier `<conf>`.

S'il a un doute sur les procédures à suivre, le lecteur est invité à revenir sur celles suivies dans l'étude précédente.

Le projet Eclipse est présent [3] dans le dossier des exemples [4]. On l'importera.

L'exécution de [InitDB] avec le SGBD MySQL5 donne les résultats suivants :

The screenshot displays the Eclipse IDE interface with the following components:

- Top Bar:** Problems, Javadoc, Declaration, Console, TestNG, Servers, Data Source Explorer.
- Console Tab:** Shows the output of a Java application: <terminated> InitDB [Java Application] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (29 mai 07 14:12:58) [personnes] P[3,1,Durant,Sylvie,05/07/2001,false,0,A[Apt 100,Les Mimosas,15 av Foch,49002,Angers,03,France]] P[2,1,Martin,Paul,31/01/2000,true,2,A[8 rue Boileau,null,null,49000,Angers,null,France]] terminé... (1)
- Database Structure View:** Shows the MySQL database structure with the following schema:
  - MySQL [v5.0]
    - information\_schema
    - jpa
      - Local Temporary
      - Procedures
      - Tables
        - articles
        - jpa02\_tb\_personne
        - jpa02\_tl\_personne
        - sequence
      - Views
    - mysql
    - test
(2)
- Database Detail View:** Shows the structure of the jpa02\_tl\_personne table with columns: ID, PRENOM, DATENAISANCE, VERSION, MARIE, NBENFANTS, NOM, CODEPOSTAL, ADR1, VILLE, ADR3, CEDEX, ADR2, PAYS.
 

COLUMN_NAME	DATA_TYPE	TYPE_NAME	COLUMN_SIZE
ID	5	bigint	19
PRENOM	12	varchar	30
DATENAISANCE	91	date	<null>
VERSION	4	int	10
MARIE	7	BIT	<null>
NBENFANTS	4	int	10
NOM	12	varchar	30
CODEPOSTAL	12	varchar	5
ADR1	12	varchar	30
VILLE	12	varchar	20
ADR3	12	varchar	30
CEDEX	12	varchar	3
ADR2	12	varchar	30
PAYS	12	varchar	20

(3)
- Database Detail View:** Shows the contents of the jpa02\_tl\_personne table with two rows:
 

ID	PRENOM	DATENAISANCE	VERSION	MARIE	NBENFANTS	NOM	CODEPOSTAL	ADR1	VILLE	ADR3	CEDEX	ADR2	PAYS
2	Paul	2000-01-31 00:00:00	1	1	2	Martin	49000	8 rue Boileau	Angers	<null>	<null>	<null>	France
3	Sylvie	2001-07-05 00:00:00	1	0	0	Dur...	49002	Apt 100	Angers	15 av Foch	03	Les Mimosas	France

(4)
- Script Folders:** Shows the MySQL ddl folder containing create.sql and drop.sql scripts.
  - ddl
    - derby
    - firebird
    - hsqldb
    - mysql5
      - create.sql
      - drop.sql
    - oracle
    - postgres
    - sqlserver
(5)

- [1] : l'affichage console
- [2] : les tables [jpa02\_tl\_personne] et [SEQUENCE] dans la perspective SQL Explorer
- [3] et [4] : la structure et le contenu de [jpa02\_tl\_personne].

Les scripts SQL générés dans ddl/mysql5 [5] sont les suivants :

#### create.sql

```

1. CREATE TABLE jpa02_tl_personne (ID BIGINT NOT NULL, PRENOM VARCHAR(30) NOT NULL, DATENAISANCE DATE NOT NULL, VERSION INTEGER NOT NULL, MARIE TINYINT(1) default 0 NOT NULL, NBENFANTS INTEGER NOT NULL, NOM VARCHAR(30) UNIQUE NOT NULL, CODEPOSTAL VARCHAR(5) NOT NULL, ADR1 VARCHAR(30) NOT NULL, VILLE VARCHAR(20) NOT NULL, ADR3 VARCHAR(30), CEDEX VARCHAR(3), ADR2 VARCHAR(30), PAYS VARCHAR(20) NOT NULL, PRIMARY KEY (ID))
2. CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(38), PRIMARY KEY (SEQ_NAME))
3. INSERT INTO SEQUENCE (SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 1)

```

#### drop.sql

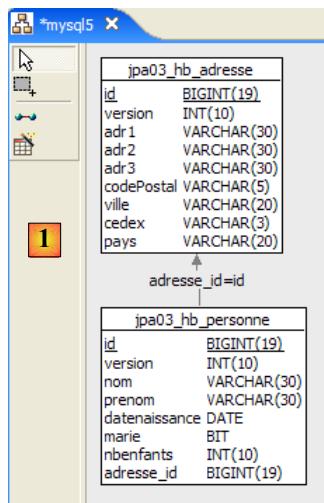
```

1. DROP TABLE jpa02_tl_personne
2. DELETE FROM SEQUENCE WHERE SEQ_NAME = 'SEQ_GEN'

```

## 2.3 Exemple 2 : relation un-à-un via une clé étrangère

### 2.3.1 Le schéma de la base de données



```
1.     alter table jpa03_hb_personne
2.         drop
3.             foreign key FKFBBFDD05FE379D0;
4.
5. drop table if exists jpa03_hb_adresse;
6.
7. drop table if exists jpa03_hb_personne;
8.
9. create table jpa03_hb_adresse (
10.     id bigint not null auto increment,
11.     version integer not null,
12.     adr1 varchar(30) not null,
13.     adr2 varchar(30),
14.     adr3 varchar(30),
15.     codePostal varchar(5) not null,
16.     ville varchar(20) not null,
17.     cedex varchar(3),
18.     pays varchar(20) not null,
19.     primary key (id)
20. ) ENGINE=InnoDB;
21.
22. create table jpa03_hb_personne (
23.     id bigint not null auto increment,
24.     version integer not null,
25.     nom varchar(30) not null unique,
26.     prenom varchar(30) not null,
27.     datenaissance date not null,
28.     marie bit not null,
29.     nbefants integer not null,
30.     adresse_id bigint not null unique,
31.     primary key (id)
32. ) ENGINE=InnoDB;
33.
34. alter table jpa03_hb_personne
35.     add index FKFBBFDD05FE379D0 (adresse_id),
36.     add constraint FKFBBFDD05FE379D0
37.         foreign key (adresse_id)
38.             references jpa03_hb_adresse (id);
```

- en [1] : la base de données. Cette fois-ci, l'adresse de la personne est mise dans une table [adresse] qui lui est propre. La table [personne] est liée à cette table par une clé étrangère.
- en [2] : la DDL générée par Hibernate pour MySQL5 :
  - lignes 9-20 : la table [adresse] qui va être liée à la classe [Adresse] devenue un objet @Entity.
  - ligne 10 : la clé primaire de la table [adresse]
  - ligne 30 : au lieu d'une adresse complète, on trouve désormais dans la table [personne], l'identifiant [adresse\_id] de cette adresse.
  - lignes 34-38 : personne(adresse\_id) est clé étrangère sur adresse(id).

### 2.3.2 Les objets @Entity représentant la base de données

Une personne avec adresse est représentée maintenant par la classe [Personne] suivante :

```
1. package entites;
2. ...
3. @Entity
4. @Table(name = "jpa03_hb_personne")
5. public class Personne implements Serializable{
6.
7.     @Id
8.     @Column(nullable = false)
9.     @GeneratedValue(strategy = GenerationType.AUTO)
10.    private Long id;
11.
12.    @Column(nullable = false)
13.    @Version
14.    private int version;
15.
16.    @Column(length = 30, nullable = false, unique = true)
17.    private String nom;
```

```

18.
19. @Column(length = 30, nullable = false)
20. private String prenom;
21.
22. @Column(nullable = false)
23. @Temporal(TemporalType.DATE)
24. private Date datenaissance;
25.
26. @Column(nullable = false)
27. private boolean marie;
28.
29. @Column(nullable = false)
30. private int nbenfants;
31.
32. @OneToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)
33. @JoinColumn(name = "adresse_id", unique = true, nullable = false)
34. private Adresse adresse;
35. ...
36. }

```

- lignes 32-34 : l'adresse de la personne

- ligne 32 : l'annotation `@OneToOne` désigne une relation **un-à-un** : une personne a au moins et au plus une adresse. L'attribut `cascade = CascadeType.ALL` signifie que toute opération (persist, merge, remove) sur l'`@Entity [Personne]` doit être cascadée sur l'`@Entity [Adresse]`. Du point de vue du contexte de persistance `em`, cela signifie la chose suivante. Si `p` est une personne et `a` son adresse :
  - une opération `em.persist(p)` explicite entraînera une opération `em.persist(a)` implicite
  - une opération `em.merge(p)` explicite entraînera une opération `em.merge(a)` implicite
  - une opération `em.remove(p)` explicite entraînera une opération `em.remove(a)` implicite

L'expérience montre que ces cascades implicites ne sont pas la panacée. Le développeur finit par oublier ce qu'elles font. On pourra préférer des opérations explicites dans le code. Il existe différents types de cascade. L'annotation `@OneToOne` aurait pu être écrite comme suit :

```

1. // @OneToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)
2. @OneToOne(cascade = {CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH,
   CascadeType.REMOVE}, fetch=FetchType.LAZY)

```

L'attribut `cascade` admet pour valeur ici un tableau de constantes précisant les types de cascades désirées.

L'attribut `fetch=FetchType.LAZY` demande à Hibernate de charger la dépendance au dernier moment. Lorsqu'on met une liste de personnes dans le contexte de persistance, on ne veut pas forcément y mettre leurs adresses. Par exemple, on ne peut vouloir cette adresse que pour une personne particulière choisie par un utilisateur au travers d'une interface web. L'attribut `fetch=FetchType.EAGER` lui, demande le chargement immédiat des dépendances.

- ligne 33 : l'annotation `@JoinColumn` définit la clé étrangère que possède la table de l'`@Entity [Personne]` sur la table de l'`@Entity [Adresse]`. L'attribut `name` définit le nom de la colonne qui sert de clé étrangère. L'attribut `unique=true` force la relation un-à-un : on ne peut avoir deux fois la même valeur dans la colonne `[adresse_id]`. L'attribut `nullable=false` force une personne à avoir une adresse.

L'adresse d'une personne est désormais représentée par l'`@Entity [Adresse]` suivante :

```

1. package entites;
2.
3. ...
4. @Entity
5. @Table(name = "jpa03_hb_adresse")
6. public class Adresse implements Serializable {
7.
8.   // champs
9.   @Id
10.  @Column(nullable = false)
11.  @GeneratedValue(strategy = GenerationType.AUTO)
12.  private Long id;
13.
14.  @Column(nullable = false)
15.  @Version
16.  private int version;
17.
18.  @Column(length = 30, nullable = false)
19.  private String adr1;
20.
21.  @Column(length = 30)
22.  private String adr2;
23.
24.  @Column(length = 30)
25.  private String adr3;
26.
27.  @Column(length = 5, nullable = false)

```

```

28. private String codePostal;
29.
30. @Column(length = 20, nullable = false)
31. private String ville;
32.
33. @Column(length = 3)
34. private String cedex;
35.
36. @Column(length = 20, nullable = false)
37. private String pays;
38.
39. @OneToOne(mappedBy = "adresse", fetch=FetchType.LAZY)
40. private Personne personne;
41.
42. // constructeurs
43. public Adresse() {
44.
45. }
46. ...
47. }

```

- ligne 4 : la classe [Adresse] devient un objet **@Entity**. Elle va donc faire l'objet d'une table dans la base de données.
- lignes 9-12 : comme tout objet **@Entity**, [Adresse] a une clé primaire. Elle a été nommée **Id** et présente les mêmes annotations (standard) de la clé primaire *Id* de l'**@Entity** [Personne].
- lignes 39-40 : la relation **un-à-un** avec l'**@Entity** [Personne]. Il y a plusieurs subtilités ici :
  - tout d'abord le champ *personne* n'est pas obligatoire. Il nous permet, à partir d'une adresse de remonter à l'unique personne ayant cette adresse. Si nous n'avions pas désiré cette commodité, le champ *personne* n'existerait pas et tout marcherait quand même.
  - la relation **un-à-un** qui lie les deux entités [Personne] et [Adresse] a déjà été configurée dans l'**@Entity** [Personne] :

```

1. @OneToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)
2. @JoinColumn(name = "adresse_id", unique = true, nullable = false)
3. private Adresse adresse;

```

Pour que les deux configurations **un-à-un** n'entrent pas en conflit l'une avec l'autre, l'une est considérée comme *principale* et l'autre comme *inverse*. C'est la relation dite *principale* qui est gérée par le pont objet / relationnel. L'autre relation dite *inverse*, n'est pas gérée directement : elle l'est indirectement par la relation *principale*. Dans **@Entity** [Adresse] :

```

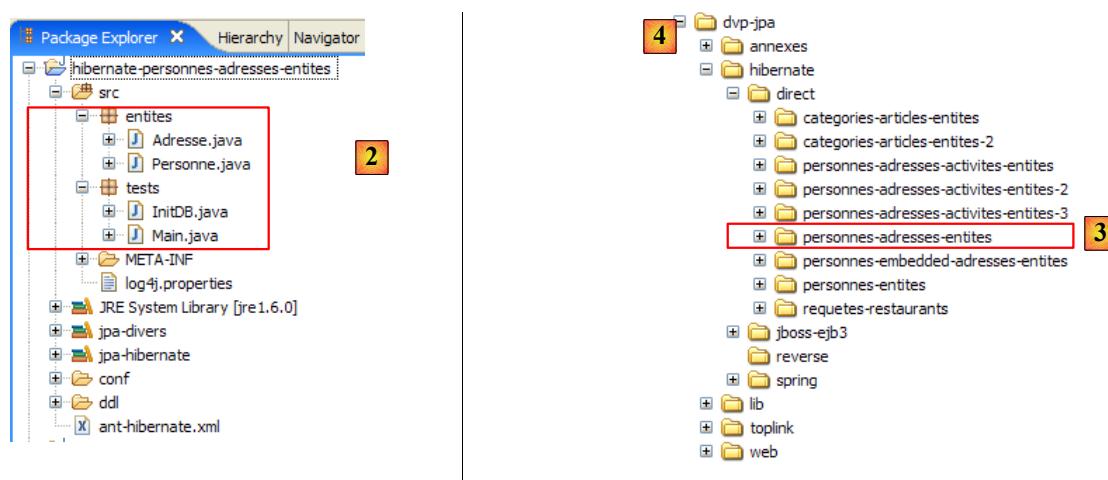
1. @OneToOne(mappedBy = "adresse", fetch=FetchType.LAZY)
2. private Personne personne;

```

c'est l'attribut **mappedBy** qui fait de la relation **un-à-un** ci-dessus, la relation *inverse* de la relation *principale* **un-à-un** définie par le champ *adresse* de **@Entity** [Personne].

### 2.3.3 Le projet Eclipse / Hibernate 1

L'implémentation JPA utilisée ici est celle d'Hibernate. Le projet Eclipse des tests est le suivant :



Le projet est présent [3] dans le dossier des exemples [4]. On l'importera.

### 2.3.4 Génération de la DDL de la base de données

En suivant les instructions du paragraphe 2.1.7, page 20, la DDL obtenue pour le SGBD MySQL5 est celle montrée au début de ce paragraphe.

### 2.3.5 InitDB

Le code de [InitDB] est le suivant :

```
1. package tests;
2. ...
3. import entites.Adresse;
4. import entites.Personne;
5.
6. public class InitDB {
7.
8.     // constantes
9.     private final static String TABLE_PERSONNE = "jpa03_hb_personne";
10.
11.    private final static String TABLE_ADRESSE = "jpa03_hb_adresse";
12.
13.    public static void main(String[] args) throws ParseException {
14.        // Contexte de persistance
15.        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
16.        EntityManager em = null;
17.        // on récupère un EntityManager à partir de l'EntityManagerFactory précédent
18.        em = emf.createEntityManager();
19.        // début transaction
20.        EntityTransaction tx = em.getTransaction();
21.        tx.begin();
22.        // requête
23.        Query sql1;
24.        // supprimer les éléments de la table PERSONNE
25.        sql1 = em.createNativeQuery("delete from " + TABLE_PERSONNE);
26.        sql1.executeUpdate();
27.        // supprimer les éléments de la table ADRESSE
28.        sql1 = em.createNativeQuery("delete from " + TABLE_ADRESSE);
29.        sql1.executeUpdate();
30.        // création personnes
31.        Personne p1 = new Personne("Martin", "Paul", new
SimpleDateFormat("dd/MM/yy").parse("31/01/2000"), true, 2);
32.        Personne p2 = new Personne("Durant", "Sylvie", new
SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
33.        // création adresses
34.        Adresse a1 = new Adresse("8 rue Boileau", null, null, "49000", "Angers", null, "France");
35.        Adresse a2 = new Adresse("Apt 100", "Les Mimosas", "15 av Foch", "49002", "Angers", "03",
"France");
36.        Adresse a3 = new Adresse("x", "x", "x", "x", "x", "x");
37.        Adresse a4 = new Adresse("y", "y", "y", "y", "y", "y");
38.        // associations personne <--> adresse
39.        p1.setAdresse(a1);
40.        a1.setPersonne(p1);
41.        p2.setAdresse(a2);
42.        a2.setPersonne(p2);
43.        // persistance des personnes et par cascade de leurs adresses
44.        em.persist(p1);
45.        em.persist(p2);
46.        // et des adresses a3 et a4 non liées à des personnes
47.        em.persist(a3);
48.        em.persist(a4);
49.        // affichage personnes
50.        System.out.println("[personnes]");
51.        for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList())
{
52.            System.out.println(p);
53.        }
54.        // affichage adresses
55.        System.out.println("[adresses]");
56.        for (Object a : em.createQuery("select a from Adresse a").getResultList()) {
57.            System.out.println(a);
58.        }
59.
60.        // fin transaction
61.        tx.commit();
62.        // fin EntityManager
63.        em.close();
64.        // fin EntityManagerFactory
65.        emf.close();
66.        // log
67.        System.out.println("terminé...");
68.
69.    }
```

Nous ne commentons que ce qui présente un intérêt nouveau vis à vis de ce qui a déjà été étudié :

- lignes 31-32 : on crée deux personnes
- lignes 34-37 : on crée quatre adresses
- lignes 39-42 : on associe les personnes (p1,p2) aux adresses (a1,a2). Les adresses (a3,a4) sont orphelines. Aucune personne ne les référence. La DDL le permet. Si une personne a forcément une adresse, l'inverse n'est pas vrai.
- lignes 44-45 : on persiste les personnes (p1,p2). Comme on a mis un attribut `cascade = CascadeType.ALL` sur la relation un-à-un qui lie une personne à son adresse, les adresses (a1,a2) de ces deux personnes devraient subir également un `persist`. C'est ce qu'on veut vérifier. Pour les adresses orphelines (a3,a4), on est obligés de faire les choses explicitement (lignes 47-48).
- lignes 51-53 : affichage de la table des personnes
- lignes 56-57 : affichage de la table des adresses

L'exécution de [InitDB] avec MySQL5 donne les résultats suivants :

The screenshot shows the Eclipse IDE interface with several open windows:

- Console View:** Displays the output of the [InitDB] application. It shows the creation of two persons (P[1,0,Martin,Paul,...] and P[2,0,Durant,Sylvie,...]), four addresses (A[1,0,...], A[2,0,...], A[3,0,...], A[4,0,...]), and a termination message.
- Database Structure View:** Shows the MySQL schema named "mysql5-jpa". It contains the "jpa" package which includes Local Temporary, Procedures, Tables (articles, jpa02\_hb\_personne, jpa02\_tl\_personne, jpa03\_hb\_adresse, jpa03\_hb\_personne, sequence), and Views. A red box labeled "1" highlights the "Tables" node.
- Database Detail View - Persons:** Shows a table with columns: id, version, nom, prenom, datenaissance, marie, nbenfants, adresse\_id. It contains two rows (id 0: Martin, id 2: Durant). A red box labeled "3" highlights the table.
- Database Detail View - Addresses:** Shows a table with columns: id, version, adr1, adr2, adr3, codePostal, ville, cedex, pays. It contains four rows (id 0: 8 rue Boileau, id 2: Apt 100, id 3: x, id 4: y). A red box labeled "4" highlights the table.

- [1] : l'affichage console
- [2] : les tables [jpa03\_hb\_\*] dans la perspective SQL Explorer
- [3] : la table des personnes
- [4] : la table des adresses. Elles sont bien toutes là. On notera également le lien qu'a la colonne [adresse\_id] dans [3] avec la colonne [id] dans [4] (clé étrangère).

## 2.3.6 Main

La classe [Main] enchaîne six tests que nous passons en revue.

### 2.3.6.1 Test1

Ce test est le suivant :

```

1.// création d'objets
2.public static void test1() throws ParseException {
3.    // contexte de persistance
4.    EntityManager em = getEntityManager();
5.    // création personnes
6.    p1 = new Personne("Martin", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"), true, 2);
7.    p2 = new Personne("Durant", "Sylvie", new SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false,
0);
8.    // création adresses
9.    a1 = new Adresse("8 rue Boileau", null, null, "49000", "Angers", null, "France");
10.   a2 = new Adresse("Apt 100", "Les Mimosas", "15 av Foch", "49002", "Angers", "03", "France");
11.   a3 = new Adresse("x", "x", "x", "x", "x", "x", "x");
12.   a4 = new Adresse("y", "y", "y", "y", "y", "y", "y");

```

```

13.    // associations personne <--> adresse
14.    p1.setAdresse(a1);
15.    a1.setPersonne(p1);
16.    p2.setAdresse(a2);
17.    a2.setPersonne(p2);
18.    // début transaction
19.    EntityTransaction tx = em.getTransaction();
20.    tx.begin();
21.    // persistance des personnes
22.    em.persist(p1);
23.    em.persist(p2);
24.    // et des adresses a3 et a4 non liées à des personnes
25.    em.persist(a3);
26.    em.persist(a4);
27.    // fin transaction
28.    tx.commit();
29.    // on affiche les tables
30.    dumpPersonne();
31.    dumpAdresse();
32. }
```

Ce code est repris de [InitDB]. Son résultat est le suivant :

```

1. main : -----
2. [personnes]
3. P[2,0,Durant,Sylvie,05/07/2001,false,0,2]
4. P[1,0,Martin,Paul,31/01/2000,true,2,1]
5. [adresses]
6. A[1,0,8 rue Boileau,null,null,49000,Angers,null,France]
7. A[2,0,Apt 100,Les Mimosas,15 av Foch,49002,Angers,03,France]
8. A[3,0,x,x,x,x,x,x,x]
9. A[4,0,y,Y,Y,Y,Y,y,y]
```

Les deux tables ont été remplies.

### 2.3.6.2 Test2

Ce test est le suivant :

```

1. // modifier un objet du contexte
2. public static void test2() {
3.     // contexte de persistance
4.     EntityManager em = getEntityManager();
5.     // début transaction
6.     EntityTransaction tx = em.getTransaction();
7.     tx.begin();
8.     // on incrémenté le nbre d'enfants de p1
9.     p1.setNbenfants(p1.getNbenfants() + 1);
10.    // on modifie son état marital
11.    p1.setMarie(false);
12.    // l'objet p1 est automatiquement sauvegardé (dirty checking)
13.    // lors de la prochaine synchronisation (commit ou select)
14.    // fin transaction
15.    tx.commit();
16.    // on affiche la nouvelle table
17.    dumpPersonne();
18. }
```

Son résultat est le suivant :

```

1. main : -----
2. [personnes]
3. P[2,0,Durant,Sylvie,05/07/2001,false,0,2]
4. P[1,1,Martin,Paul,31/01/2000,false,3,1]
```

- ligne 4 : la personne p1 a vu son nombre d'enfants augmenter de 1, et sa version passer de 0 à 1

### 2.3.6.3 Test4

Ce test est le suivant :

```

1. // supprimer un objet appartenant au contexte de persistance
2. public static void test4() {
3.     // contexte de persistance
4.     EntityManager em = getEntityManager();
5.     // début transaction
6.     EntityTransaction tx = em.getTransaction();
7.     tx.begin();
8.     // on supprime l'objet attaché p2
9.     em.remove(p2);
```

```

10.    // fin transaction
11.    tx.commit();
12.    // on affiche les nouvelles tables
13.    dumpPersonne();
14.    dumpAdresse();
15.}

```

- ligne 9 : on supprime la personne p2. Celle-ci a une relation de cascade avec l'adresse a2. Donc l'adresse a2 devrait être également supprimée.

Le résultat du test 4 est le suivant :

```

1. main : -----
2. [personnes]
3. P[2,0,Durant,Sylvie,05/07/2001,false,0,2]
4. P[1,0,Martin,Paul,31/01/2000,true,2,1]
5. [adresses]
6. A[1,0,8 rue Boileau,null,null,49000,Angers,null,France]
7. A[2,0,Apt 100,Les Mimosas,15 av Foch,49002,Angers,03,France]
8. A[3,0,x,x,x,x,x,x,x]
9. A[4,0,y,y,y,y,y,y]
10. main : -----
11. [personnes]
12. P[2,0,Durant,Sylvie,05/07/2001,false,0,2]
13. P[1,1,Martin,Paul,31/01/2000,false,3,1]
14. main : -----
15. [personnes]
16. P[1,1,Martin,Paul,31/01/2000,false,3,1]
17. [adresses]
18. A[1,0,8 rue Boileau,null,null,49000,Angers,null,France]
19. A[3,0,x,x,x,x,x,x,x]
20. A[4,0,y,y,y,y,y,y]

```

- la personne p2 présente ligne 3 du test 1 n'est plus présente dans le test 4
- il en est de même pour son adresse a2, en ligne 7 du test 1 et absente du test 4.

### 2.3.6.4 Test5

Ce test est le suivant :

```

1.// détacher, réattacher et modifier
2.public static void test5() {
3.    // nouveau contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // début transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // on réattache p1 au nouveau contexte
9.    p1 = em.find(Personne.class, p1.getId());
10.   // on change l'adresse de p1
11.   p1.getAdresse().setVille("Paris");
12.   // fin transaction
13.   tx.commit();
14.   // on affiche les nouvelles tables
15.   dumpPersonne();
16.   dumpAdresse();
17. }

```

- ligne 4 : on a un contexte de persistance neuf, donc vide.
- ligne 9 : on met la personne p1 dedans. p1 est cherché dans la base parce qu'il n'est pas dans le contexte. Les éléments dépendants de p1 (son adresse) eux, ne sont pas ramenés de la base parce qu'on a écrit :

```
| @OneToOne(..., fetch=FetchType.LAZY)
```

C'est le concept du "**lazy loading**" ou "chargement en juste à temps" : les dépendances d'un objet persistant ne sont amenées en mémoire que lorsqu'on en a besoin.

- ligne 11 : on modifie le champ ville de l'adresse de p1. A cause du *getAdresse* et si l'adresse de p1 n'était pas déjà dans le contexte de persistance, elle va y être amenée par une lecture de la base.
- ligne 13 : on valide la transaction, ce qui va entraîner la synchronisation du contexte de persistance avec la base. Celui-ci va constater que l'adresse de la personne p1 a été modifiée et va la sauvegarder.

L'exécution de *test5* donne les résultats suivants :

```

1. main : -----
2. [personnes]
3. P[1,1,Martin,Paul,31/01/2000,false,3,1]

```

```

4. [adresses]
5. A[1,0,x,rue Boileau,null,null,49000,Angers,null,France]
6. A[3,0,x,x,x,x,x,x]
7. A[4,0,y,y,Y,Y,Y,Y]
8. main : -----
9. [personnes]
10. P[1,1,Martin,Paul,31/01/2000,false,3,1]
11. [adresses]
12. A[1,1,8 rue Boileau,null,null,49000,Paris,null,France]
13. A[3,0,x,x,x,x,x,x]
14. A[4,0,y,y,Y,Y,Y,Y,Y]

```

- la personne p1 (ligne 3 test4, ligne 10 test5) a bien vu sa ville passer d'Angers (ligne 5 test4) à Paris (ligne 12 test5).

### 2.3.6.5 Test6

Ce test est le suivant :

```

1.// supprimer un objet Adresse
2.public static void test6() {
3.    EntityTransaction tx = null;
4.    // nouveau contexte de persistance
5.    EntityManager em = getNewEntityManager();
6.    // début transaction
7.    tx = em.getTransaction();
8.    tx.begin();
9.    // on réattache l'adresse a3 au nouveau contexte
10.   a3 = em.find(Adresse.class, a3.getId());
11.   System.out.println(a3);
12.   // on la supprime
13.   em.remove(a3);
14.   // fin transaction
15.   tx.commit();
16.   // dump table Adresse
17.   dumpAdresse();
18. }

```

- ligne 5 : on est dans un contexte de persistance neuf, donc vide.
- ligne 10 : on met l'adresse a3 dans le contexte de persistance
- ligne 13 : on la supprime. C'était une adresse orpheline (non liée à une personne). La suppression est donc possible.

Le résultat de l'exécution est le suivant :

```

1. main : -----
2. [personnes]
3. P[1,1,Martin,Paul,31/01/2000,false,3,1]
4. [adresses]
5. A[1,1,8 rue Boileau,null,null,49000,Paris,null,France]
6. A[3,0,x,x,x,x,x,x]
7. A[4,0,y,y,Y,Y,Y,Y]
8. main : -----
9. A[3,0,x,x,x,x,x,x]
10. [adresses]
11. A[1,1,8 rue Boileau,null,null,49000,Paris,null,France]
12. A[4,0,y,y,Y,Y,Y,Y]

```

- l'adresse a3 du test 5 (ligne 6) a disparu des adresses du test 6 (lignes 11-12)

### 2.3.6.6 Test7

Ce test est le suivant :

```

1.// rollback
2.public static void test7() {
3.    EntityTransaction tx = null;
4.    try {
5.        // nouveau contexte de persistance
6.        EntityManager em = getNewEntityManager();
7.        // début transaction
8.        tx = em.getTransaction();
9.        tx.begin();
10.       // on réattache l'adresse a1 au nouveau contexte
11.       newa1 = em.find(Adresse.class, a1.getId());
12.       // on réattache l'adresse a4 au nouveau contexte
13.       newa4 = em.find(Adresse.class, a4.getId());
14.       // on essaie de les supprimer - devrait lancer une exception car on ne peut supprimer une
     adresse liée à une personne, ce qui est le cas de newa1
15.       em.remove(newa4);
16.       em.remove(newa1);

```

```

17.     // fin transaction
18.     tx.commit();
19. } catch (RuntimeException e1) {
20.     // on a eu un pb
21.     System.out.format("Erreur dans transaction [%s%n%s%n%s%n]", e1.getClass().getName(),
22.     e1.getMessage(), e1.getCause(), e1.getCause()
23.     .getCause());
24.     try {
25.         if (tx.isActive())
26.             tx.rollback();
27.     } catch (RuntimeException e2) {
28.         System.out.format("Erreur au rollback [%s]", e2.getMessage());
29.     }
30.     // on abandonne le contexte courant
31.     em.clear();
32. } // dump - la table Adresse n'a pas du changer à cause du rollback
33. dumpAdresse();
34. }

```

- *test7*: on teste un rollback d'une transaction
  - ligne 6 : on est dans un contexte de persistance neuf, donc vide.
  - ligne 11 : on met l'adresse *a1* dans le contexte de persistance, sous la référence *newa1*
  - ligne 13 : on met l'adresse *a4* dans le contexte de persistance, sous la référence *newa4*
  - lignes 15-16 : on supprime les deux adresses *newa1* et *newa4*. *newa1* est l'adresse de la personne *p1* et donc dans la base *p1* référence *newa1* par une clé étrangère. Supprimer *newa1* va donc échouer et lancer une exception lors de la synchronisation du contexte de persistance au commit de la transaction (ligne 18). Celle-ci va subir un *rollback* (ligne 25) et donc les deux opérations de la transaction vont être annulées. On devrait donc constater que l'adresse *newa4*, qui aurait pu légalement être supprimée, ne l'a pas été.

L'exécution donne le résultat suivant :

```

1. main : -----
2. A[3,0,x,x,x,x,x,x,x]
3. [adresses]
4. A[1,1,8 rue Boileau,null,null,49000,Paris,null,France]
5. A[4,0,y,y,Y,Y,Y,Y]
6. main : -----
7. Erreur dans transaction [javax.persistence.RollbackException]
8. Error while committing the transaction
9. org.hibernate.ObjectDeletedException: deleted entity passed to persist: [entites.Adresse#<null>]
10. null
11. [adresses]
12. A[1,1,8 rue Boileau,null,null,49000,Paris,null,France]
13. A[4,0,y,y,Y,Y,Y,Y]

```

- la table des adresses du test 7 (lignes 12-13) est identique à celle du test 6 (lignes 4-5). Le rollback semble avoir eu lieu. Ceci dit, le message d'erreur de la ligne 9 est une énigme et mérite d'être creusée. Il semblerait que l'exception qui s'est produite ne soit pas celle attendue. Il faut passer les logs d'Hibernate dans *log4j.properties* en mode DEBUG pour y voir plus clair :

```

1. # Root logger option
2. log4j.rootLogger=ERROR, stdout
3.
4. # Hibernate logging options (INFO only shows startup messages)
5. log4j.logger.org.hibernate=DEBUG

```

On constate alors, que lorsque l'adresse *a1* a été placée dans le contexte de persistance, Hibernate y a placé également la personne *p1*, probablement à cause de la relation **un-à-un** de l'*@Entity* [Adresse] :

```

1. @OneToOne(mappedBy = "adresse", fetch=FetchType.LAZY)
2. private Personne personne;

```

Bien qu'on ait demandé le "LazyLoading" ici, la dépendance [Personne] est pourtant immédiatement chargée. Cela signifie probablement que l'attribut *fetch=FetchType.LAZY* n'a pas de sens ici. On constate ensuite qu'au commit de la transaction, Hibernate a préparé la suppression des adresses *a1* et *a4* mais également la sauvegarde de la personne *p1*. Et c'est là que se produit l'exception : parce que la personne *p1* a une cascade sur son adresse, Hibernate veut persister également l'adresse *a1* alors qu'elle vient d'être détruite. C'est Hibernate qui lance l'exception et non le pilote Jdbc. D'où le message de la ligne 9 plus haut. Par ailleurs, on peut constater que le *rollback* de la ligne 25 n'est jamais exécuté car la transaction est devenue inactive. Le test de la ligne 24 empêche donc le *rollback*.

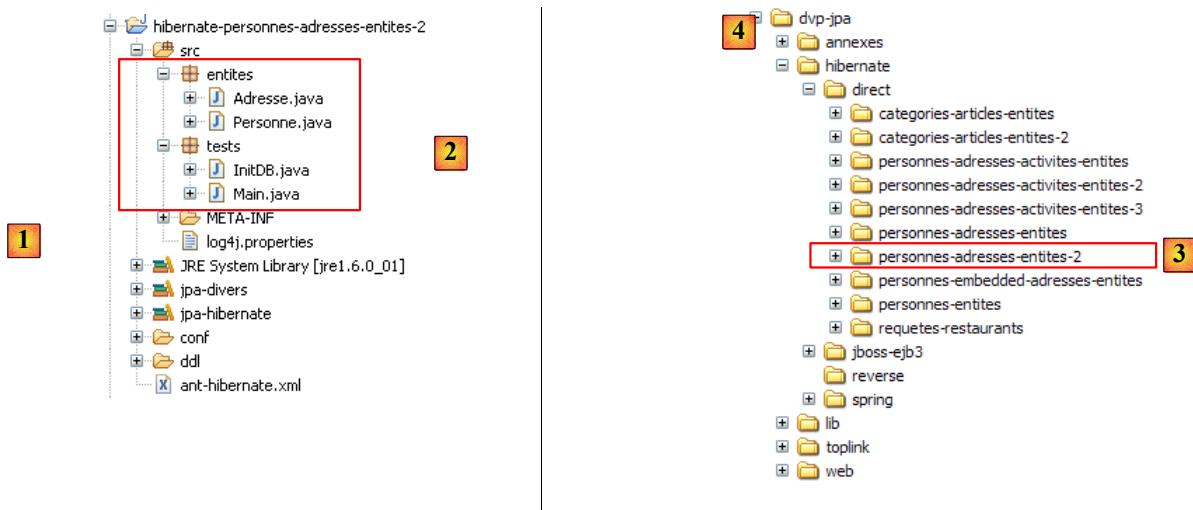
On n'a donc pas atteint l'objectif désiré : montrer un rollback. Aucun ordre SQL n'a en fait été émis sur la base. On retiendra quelques points :

- l'intérêt d'activer des logs fins afin de comprendre ce que fait l'ORM

- si un ORM peut faciliter la vie du développeur, il peut également la lui compliquer en masquant des comportements que le développeur aurait besoin de connaître. Ici, le mode de chargement des dépendances d'une @Entity.

### 2.3.7 Projet Eclipse / Hibernate 2

Nous copions / collons le projet Eclipse / Hibernate afin de modifier légèrement la configuration des objets @Entity :



Le projet est présent [3] dans le dossier des exemples [4]. On l'importera.

Nous modifions uniquement l'@Entity [Adresse] afin qu'elle n'ait plus de relation inverse un-à-un avec l'@Entity [Personne] :

```

1. package entites;
2. ...
3. @Entity
4. @Table(name = "jpa04_hb_adresse")
5. public class Adresse implements Serializable {
6.
7.     // champs
8.     @Id
9.     @Column(nullable = false)
10.    @GeneratedValue(strategy = GenerationType.AUTO)
11.    private Long id;
12.
13.    @Column(nullable = false)
14.    @Version
15.    private int version;
16.
17.    @Column(length = 30, nullable = false)
18.    private String adr1;
19.
20.    ...
21.
22.    @Column(length = 20, nullable = false)
23.    private String pays;
24.
25.    // @OneToOne(mappedBy = "adresse", fetch=FetchType.LAZY)
26.    // private Personne personne;
27.
28.    // constructeurs
29.    public Adresse() {
30.
31. }
```

- lignes 25-26 : la relation @OneToOne inverse est supprimée. Il faut bien comprendre qu'une relation inverse n'est jamais indispensable. Seule la relation principale l'est. La relation inverse peut être utilisée par commodité. Ici, elle permettait d'avoir de façon simple, le propriétaire d'une adresse. Une relation inverse peut toujours être remplacée par une requête JPQL. C'est ce que nous allons montrer dans l'exemple qui suit.

Les programmes de test sont repris à l'identique. Celui qui nous intéresse est uniquement le test 7, celui dans lequel on a vu la relation inverse **un-à-un**, en action. Nous ajoutons par ailleurs un test 8 pour montrer comment, sans relation inverse *Adresse* -> *Personne*, on peut néanmoins récupérer la personne ayant telle adresse.

Le test 7 ne change pas. Son exécution donne maintenant les résultats suivants (logs désactivés) :

```
1. main : ----- test6
Persistance Java 5 par la pratique
```

```

2. A[3,0,x,x,x,x,x,x,x]
3. [adresses]
4. A[1,1,8 rue Boileau,null,null,49000,Paris,null,France]
5. A[4,0,y,y,Y,Y,Y,Y,Y]
6. main : -----
7. Erreur dans transaction [javax.persistence.RollbackException]
8. Error while committing the transaction
9. org.hibernate.exception.ConstraintViolationException: could not delete: [entites.Adresse#1]
10. java.sql.SQLException: Cannot delete or update a parent row: a foreign key constraint fails
    (`jpa/jpa04_hb_personne`, CONSTRAINT `FKEA3F04515FE379D0` FOREIGN KEY (`adresse_id`) REFERENCES
     `jpa04_hb_adresse` (`id`))
11. [adresses]
12. A[1,1,8 rue Boileau,null,null,49000,Paris,null,France]
13. A[4,0,y,y,Y,Y,Y,Y,Y]

```

- cette fois-ci, on a bien l'exception attendue : celle lancée par le pilote Jdbc parce qu'on a voulu supprimer dans la table [adresse] une ligne référencée par une clé étrangère d'une ligne de la table [personne]. La ligne [10] est explicite sur la cause de l'erreur.
- le rollback a bien eu lieu : à l'issue du test 7, la table [adresse] (lignes 12-13) est celle qu'on avait à l'issue du test 6 (lignes 4-5).

Quelle est la différence avec le test 7 du projet Eclipse précédent ? Pourquoi a-t-on ici une exception Jdbc qu'on n'avait pas pu avoir lors du test précédent ? Parce que l'@Entity [Adresse] n'a plus de relation inverse un-à-un avec l'@Entity [Personne], elle est gérée de façon isolée par Hibernate. Lorsque l'adresse *newa1* a été amenée dans le contexte de persistance, Hibernate n'a pas mis également dans ce contexte, la personne *p1* ayant cette adresse. La suppression des adresses *newa1* et *newa4* s'est donc faite sans entités Personne dans le contexte.

Maintenant, comment à partir de l'adresse *newa1* pourrait-on avoir la personne *p1* ayant cette adresse ? C'est une question légitime. Le test 8 suivant y répond :

```

1.// relation inverse un-à-un
2.// réalisée par une requête JPQL
3.public static void test8() {
4.    EntityTransaction tx = null;
5.    // nouveau contexte de persistance
6.    EntityManager em = getNewEntityManager();
7.    // début transaction
8.    tx = em.getTransaction();
9.    tx.begin();
10.   // on réattache l'adresse a1 au nouveau contexte
11.   newa1 = em.find(Adresse.class, a1.getId());
12.   // on récupère la personne propriétaire de cette adresse
13.   Personne p1 = (Personne) em.createQuery("select p from Personne p join p.adresse a where
    a.id=:adresseId").setParameter("adresseId", newa1.getId())
14.       .getSingleResult();
15.   // on les affiche
16.   System.out.println("adresse=" + newa1);
17.   System.out.println("personne=" + p1);
18.   // fin transaction
19.   tx.commit();
20. }

```

- ligne 6 : nouveau contexte de persistance vide
- lignes 8-9 : début transaction
- ligne 11 : l'adresse *a1* est amenée dans le contexte de persistance et référencée par *newa1*.
- ligne 13 : on récupère la personne *p1* ayant l'adresse *newa1* par une requête JPQL. On sait que [Personne] et [Adresse] sont liées par une relation de clé étrangère. Dans la classe [Personne], c'est le champ [adresse] qui a l'annotation @OneToOne qui matérialise cette relation. L'écriture JPQL "select p from Personne p join p.adresse a" réalise une jointure entre les tables [personne] et [adresse]. L'équivalent SQL généré dans une console Hibernate (cf exemples du paragraphe 2.1.12, page 34) est le suivant :

```

1. SQL #0 types: entites.Personne
2. -----
3. select
4.     personne0_.id as id1_,
5.     personne0_.version as version1_,
6.     personne0_.nom as nom1_,
7.     personne0_.prenom as prenom1_,
8.     personne0_.datenaissance as datenaiss5_1_,
9.     personne0_.marie as marie1_,
10.    personne0_.nbenfants as nbenfants1_,
11.    personne0_.adresse_id as adresse8_1_
12.   from
13.      jpa04_hb_personne personne0_
14.     inner join
15.      jpa04_hb_adresse adresse1_
16.    on personne0_.adresse_id=adresse1_.id

```

On voit clairement la jointure des deux tables. Chaque personne est maintenant reliée à son adresse. Il reste à préciser qu'on ne s'intéresse qu'à l'adresse `newa1`. La requête devient "`select p from Personne p join p.adresse a where a.id=:adresseId`". On notera l'utilisation des alias `p` et `a`. Les requêtes JPQL utilisent les alias de façon intensive. Ainsi l'expression "`from Personne p join p.adresse a`" fait qu'une personne est représentée par l'alias `p` et son adresse (`p.adresse`) par l'alias `a`. L'opération de restriction "`where a.id=:adresseId`" restreint les lignes demandées aux seules personnes `p` ayant la valeur `:adresseId` comme identifiant de leur adresse `a`. `:adresseId` est appelé un paramètre, et l'ordre JPQL un ordre JPQL paramétré. A l'exécution, ce paramètre doit recevoir une valeur. C'est la méthode

```
|Query setParameter(String nomParamètre, Object valeurParamètre)|
```

qui permet de donner une valeur à un paramètre identifié par son nom. On notera que `setParameter` rend un objet `Query`, comme la méthode `createQuery`. Si bien qu'on peut enchaîner les appels de méthodes `[em.createQuery(...).setParameter(...).getSingleResult(...)]`, les méthodes `[setParameter, getSingleResult]` étant des méthodes de l'interface `Query`. La méthode `[getSingleResult]` est utilisée pour les requêtes `Select` ne rendant qu'un unique résultat. C'est le cas ici.

- lignes 16-17 : on affiche l'adresse `newa1` et la personne `p1` ayant cette adresse, pour vérification.

Le résultat obtenu est le suivant :

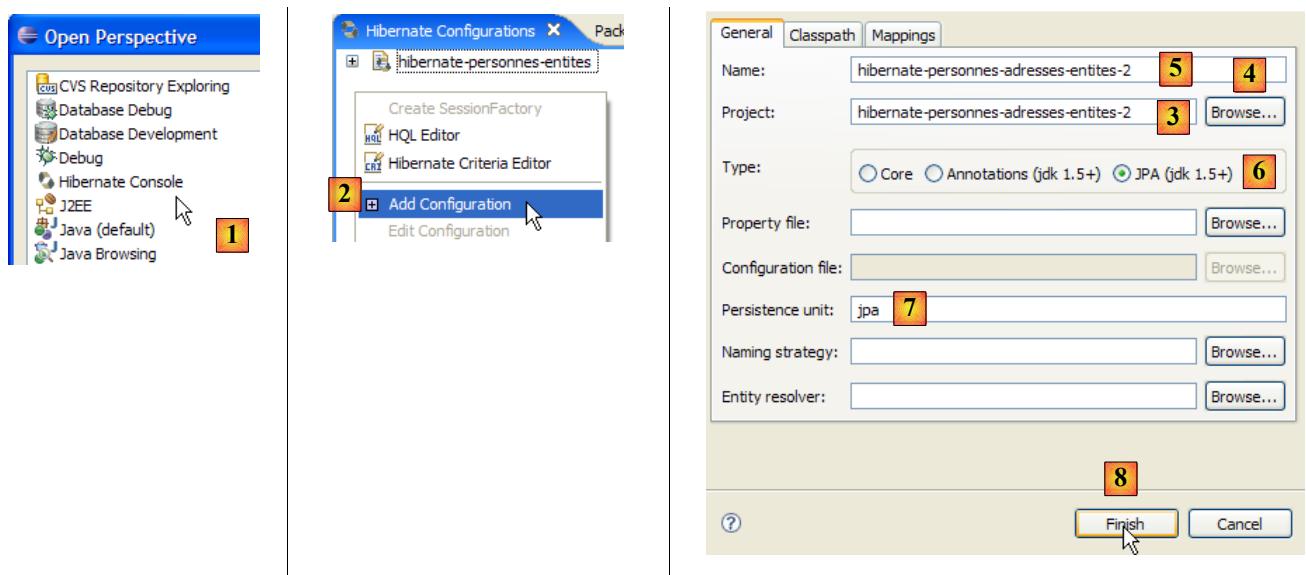
```
|1. main : -----
2. adresse=A[1,1,8 rue Boileau,null,null,49000,Paris,null,France]
3. personne=P[1,1,Martin,Paul,31/01/2000,false,3,1]|
```

Il est correct. On retiendra de cet exemple que la relation **inverse un-à-un** de l'`@entity [Adresse]` vers l'`@entity [Personne]` n'était pas indispensable. L'expérience a montré ici que sa suppression amenait un comportement plus prévisible du code. C'est souvent le cas.

### 2.3.8 Console Hibernate

Le test 8 précédent a utilisé une commande JPQL pour faire une jointure entre les entités `Personne` et `Adresse`. Bien qu'analogues au langage SQL, les langages JPQL de JPA ou HQL d'Hibernate nécessitent un apprentissage et la console Hibernate est excellente pour cela. Nous l'avons déjà utilisée au paragraphe 2.1.12, page 34, pour exploiter une unique table. Nous recommençons ici pour exploiter deux tables liées par une relation de clé étrangère.

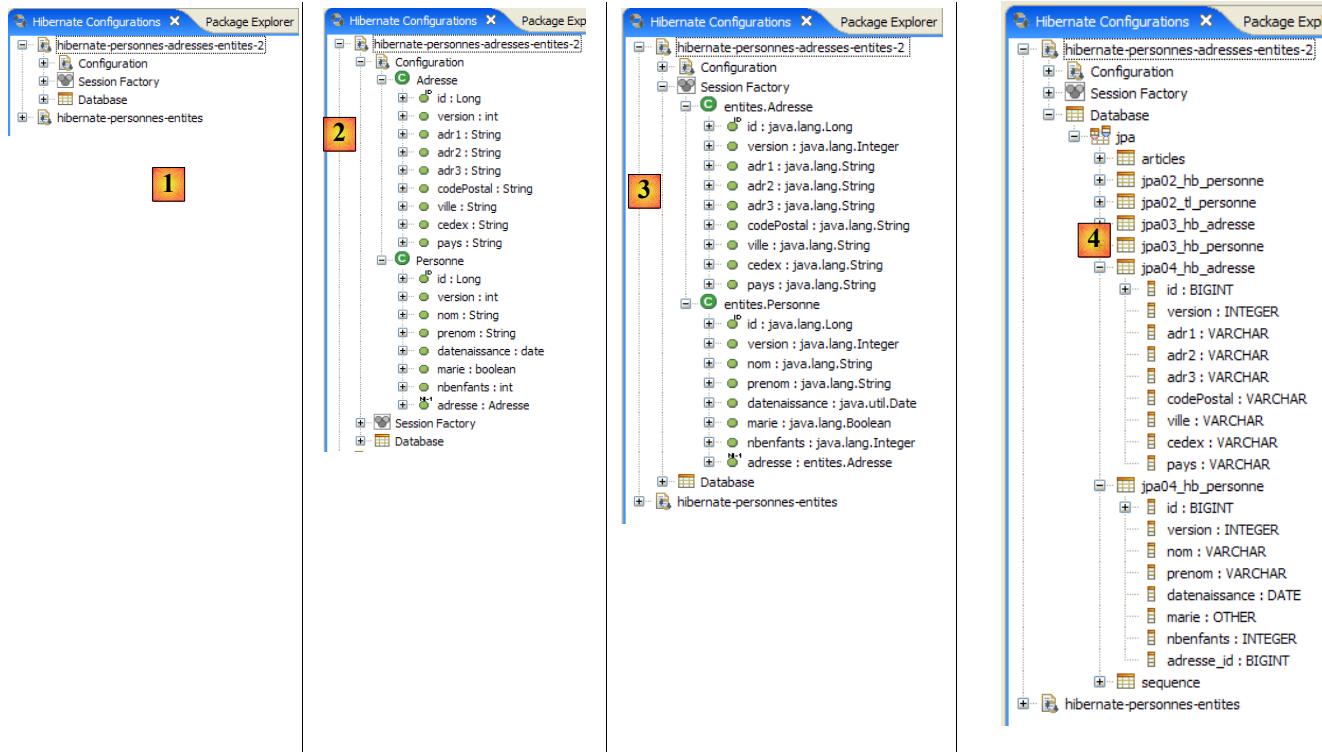
Créons une console Hibernate pour notre projet Eclipse actuel :



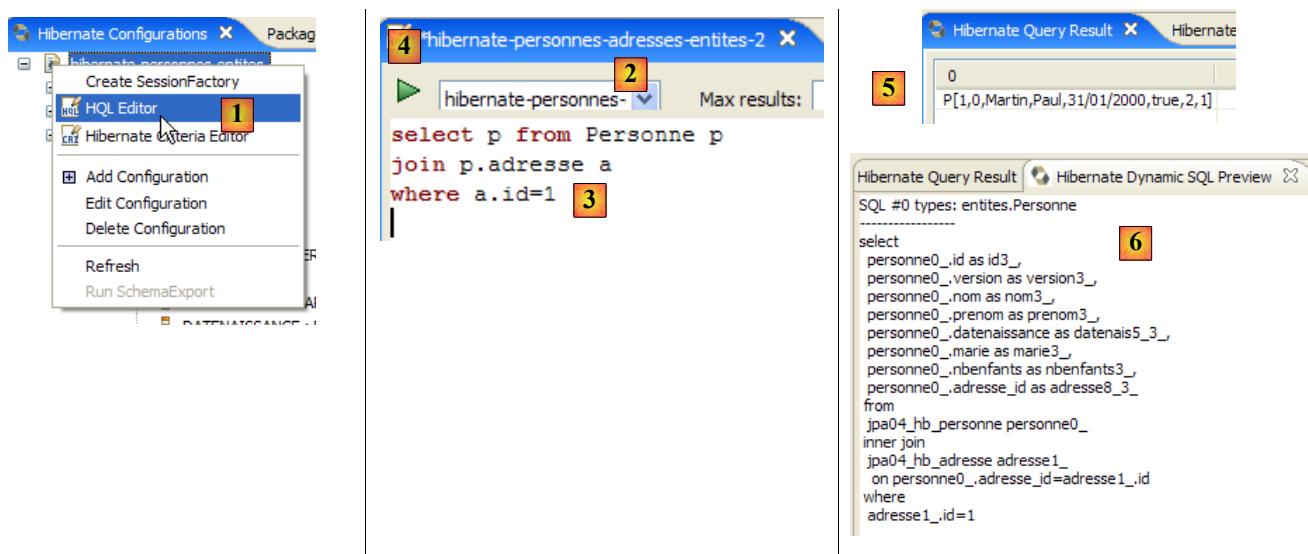
- [1] : nous passons dans une perspective [Hibernate Console] (Window / Open Perspective / Other)
- [2] : nous créons une nouvelle configuration
- à l'aide du bouton [4], nous sélectionnons le projet Java pour lequel est créé la configuration Hibernate. Son nom s'affiche dans [3].
- en [5], nous donnons le nom que nous voulons à cette configuration. Ici, nous avons repris le nom du projet Java.

- en [6], nous indiquons que nous utilisons une configuration JPA afin que l'outil sache qu'il doit exploiter le fichier [META-INF/persistence.xml]
- en [7] : nous indiquons dans ce fichier [META-INF/persistence.xml], il faut utiliser l'unité de persistance qui s'appelle **jpa**.
- en [8], on valide la configuration.

Pour la suite, il faut que le SGBD soit lancé. Ici, il s'agit de MySQL5.



- en [1] : la configuration créée présente une arborescence à trois branches
- en [2] : la branche [Configuration] liste les objets que la console a utilisés pour se configurer : ici les **@Entity Personne** et **Adresse**.
- en [3] : la *SessionFactory* est une notion Hibernate proche de l'*EntityManager* de JPA. Elle réalise le pont objet / relationnel grâce aux objets de la branche [Configuration]. En [3] sont présentés les objets du contexte de persistance, ici de nouveau les **@Entity Personne** et **Adresse**.
- en [4] : la base de données accédée au moyen de la configuration trouvée dans [persistence.xml]. On y retrouve les tables [**jpa04\_hb\_\***] générées par notre projet Eclipse actuel.



- en [1], on crée un éditeur HQL
- dans l'éditeur HQL,
  - en [2], on choisit la configuration Hibernate à utiliser s'il y en a plusieurs (c'est le cas ici)
  - en [3], on tape la commande JPQL qu'on veut exécuter, ici la commande JPQL du test 8

- en [4], on l'exécute
- en [5], on obtient les résultats de la requête dans la fenêtre [Hibernate Query Result].
- en [6], la fenêtre [Hibernate Dynamic SQL preview] permet de voir la requête SQL qui a été jouée.

Une autre façon d'obtenir le même résultat :

1

2

3

```

HQL *hibernate-personnes-adresses-entites-2 X
[hibernate-personnes-] Max results:
select p from Personne p, Adresse a
where p.adresse.id=a.id
and a.id=1

Hibernate Query Result X Hibernate Dynamic SQL #0 types: entites.Personne
-----
select
personne0_.id as id3_,
personne0_.version as version3_,
personne0_.nom as nom3_,
personne0_.prenom as prenom3_,
personne0_.datenaissance as datenaiss5_3_,
personne0_.marie as marie3_,
personne0_.nbenfants as nbefants3_,
personne0_.adresse_id as adresse8_3_
from
jpa04_hb_personne personne0_,
jpa04_hb_adresse adresse1_
where
personne0_.adresse_id=adresse1_.id
and adresse1_.id=1

```

P[1,0,Martin,Paul,31/01/2000,true,2,1]

- en [1] : la commande JPQL opérant la jointure des entités *Personne* et *Adresse*. [ref1] appelle cette forme " jointure theta ".
- en [2] : l'équivalent SQL
- en [3] : le résultat

Une troisième forme acceptée uniquement par Hibernate (HQL) :

1

2

3

```

HQL *hibernate-personnes-adresses-entites-2 X
[hibernate-personnes-] Max results:
select p from Personne p
where p.adresse.id=1

Hibernate Query Result X Hibernate Dynamic SQL #0 types: entites.Personne
-----
select
personne0_.id as id3_,
personne0_.version as version3_,
personne0_.nom as nom3_,
personne0_.prenom as prenom3_,
personne0_.datenaissance as datenaiss5_3_,
personne0_.marie as marie3_,
personne0_.nbefants as nbefants3_,
personne0_.adresse_id as adresse8_3_
from
jpa04_hb_personne personne0_
where
personne0_.adresse_id=1

```

P[1,0,Martin,Paul,31/01/2000,true,2,1]

- en [1] : la commande HQL. JPQL n'accepte pas la notation *p.adresse.id*. Il n'accepte qu'un niveau d'indirection.
- en [2] : l'équivalent SQL. On voit qu'il évite la jointure entre tables.
- en [3] : le résultat

Voici d'autres exemples :

**Section 1:**

```
hibernate-personnes-adresses-entities-2
[1] hibernate-personnes- Max resu
select p,a from Personne p
join p.adresse a
```

**Section 2:**

```
Hibernate Query Result [2] Hibernate Dynamic SQL Preview
SQL #0 types: entites.Personne, entites.Adresse
-----
select
    personne0_.id as id3_0_,
    adresse1_.id as id2_1_,
    personne0_.version as version3_0_,
    personne0_.nom as nom3_0_,
    personne0_.prenom as prenom3_0_,
    personne0_.datenaissance as datenaiss5_3_0_,
    personne0_.marie as marie3_0_,
    personne0_.nenfants as nbenfants3_0_,
    personne0_.adresse_id as adresse8_3_0_,
    adresse1_.version as version2_1_,
    adresse1_.adr1 as adr3_2_1_,
    adresse1_.adr2 as adr4_2_1_,
    adresse1_.adr3 as adr5_2_1_,
    adresse1_.codePostal as codePostal2_1_,
    adresse1_.ville as ville2_1_,
    adresse1_.cedex as cedex2_1_,
    adresse1_.pays as pays2_1_
from
    jpa04_hb_personne personne0_
inner join
    jpa04_hb_adresse adresse1_
    on personne0_.adresse_id=adresse1_.id
```

**Section 3:**

0	1
P[1,0,Martin,Paul,31/01/2000,true,2,1]	A[1,0,8 rue Boileau,null,null,49000,Angers,null,France]
P[2,0,Durant,Sylvie,05/07/2001,false,0,2]	A[2,0,Apt 100,Les Mimosas,15 av Foch,49002,Angers,03,France]

- en [1] : la liste des personnes avec leur adresse
- en [2] : l'équivalent SQL.
- en [3] : le résultat

The screenshot shows the Hibernate Query Result interface with three panes:

- Panier 1:** Shows the JPQL query: `select a,p from Personne p right join p.adresse a`.
- Panier 2:** Shows the generated SQL query:

```
select
adresse1_.id as id2_0_,
personne0_.id as id3_1_,
adresse1_.version as version2_0_,
adresse1_.adr1 as adr3_2_0_,
adresse1_.adr2 as adr4_2_0_,
adresse1_.adr3 as adr5_2_0_,
adresse1_.codePostal as codePostal2_0_,
adresse1_.ville as ville2_0_,
adresse1_.cedex as cedex2_0_,
adresse1_.pays as pays2_0_,
personne0_.version as version3_1_,
personne0_.nom as nom3_1_,
personne0_.prenom as prenom3_1_,
personne0_.datenaissance as datenaiss5_3_1_,
personne0_.marie as marie3_1_,
personne0_.nbEnfants as nbEnfants3_1_,
personne0_.adresse_id as adresse8_3_1_
from
jpa04_hb_personne personne0_
right outer join
jpa04_hb_adresse adresse1_
on personne0_.adresse_id=adresse1_.id
```
- Panier 3:** Shows the resulting table with 3 rows of data.

- en [1] : la liste des adresses avec leur propriétaire s'il y en a un ou aucun sinon (jointure externe **droite** : l'entité *Adresse* qui va fournir les lignes sans relation avec *Personne* est à **droite** du mot clé *join*).
- en [2] : l'équivalent SQL.
- en [3] : le résultat

On notera que seule l'entité *Personne* détient une relation avec l'entité *Adresse*. L'inverse n'est plus vrai depuis qu'on a supprimé la relation inverse un-à-un appelée *personne* dans l'entité *Adresse*. Si cette relation inverse existait, on aurait pu écrire :

The screenshot shows the Hibernate Query Result interface with three panes:

- Panier 1:** Shows the JPQL query: `select a,p from Adresse a left join a.personne p`.
- Panier 2:** Shows the generated SQL query:

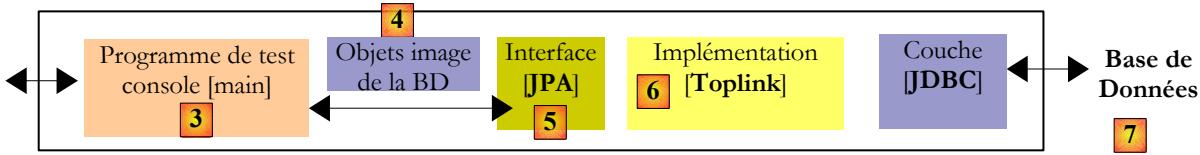
```
select
adresse0_.id as id28_0_,
personne1_.id as id29_1|,
adresse0_.version as version28_0_,
adresse0_.adr1 as adr3_28_0_,
adresse0_.adr2 as adr4_28_0_,
adresse0_.adr3 as adr5_28_0_,
adresse0_.codePostal as codePostal28_0_,
adresse0_.ville as ville28_0_,
adresse0_.cedex as cedex28_0_,
adresse0_.pays as pays28_0_,
personne1_.version as version29_1|,
personne1_.nom as nom29_1|,
personne1_.prenom as prenom29_1|,
personne1_.datenaissance as datenaiss5_29_1|,
personne1_.marie as marie29_1|,
personne1_.nbEnfants as nbEnfants29_1|,
personne1_.adresse_id as adresse8_29_1|
from
jpa03_hb_adresse adresse0_
left outer join
jpa03_hb_personne personne1_
on adresse0_.id=personne1_.adresse_id
```
- Panier 3:** Shows the resulting table with 3 rows of data.

- en [1] : la liste des adresses avec leur propriétaire s'il y en a un ou aucun sinon (jointure externe **gauche** : l'entité *Adresse* qui va fournir les lignes sans relation avec *Personne* est à **gauche** du mot clé *join*).
- en [2] : l'équivalent SQL.
- en [3] : le résultat

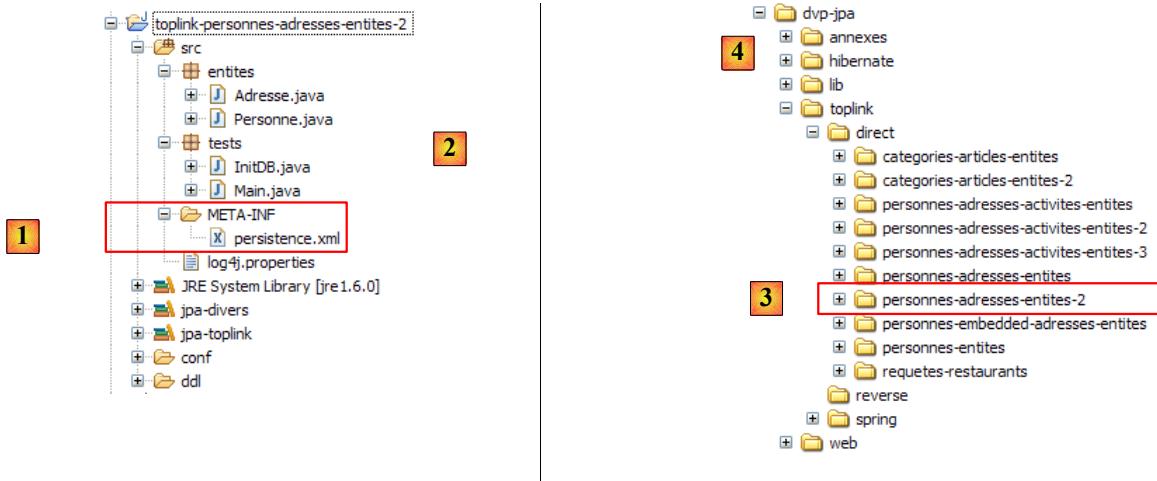
Nous invitons très vivement le lecteur à s'entraîner au langage JPQL avec la console Hibernate.

### 2.3.9 Implémentation JPA / Toplink

Nous utilisons maintenant une implémentation JPA / Toplink :



Le nouveau projet Eclipse des tests est le suivant :



Les codes Java sont identiques à ceux du projet Hibernate précédent. L'environnement (bibliothèques – persistence.xml – sgbд – dossiers conf, ddl – script ant) est celui étudié au paragraphe 2.1.15.2, page 58. Le projet Eclipse est présent [3] dans le dossier des exemples [4]. On l'importera.

Le fichier <persistence.xml> est modifié en un point, celui des entités déclarées :

```

1. <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
2.   <!-- provider -->
3.   <provider>oracle.toplink.essentials.PersistenceProvider</provider>
4.   <!-- classes persistantes -->
5.   <class>entites.Personne</class>
6.   <class>entites.Adresse</class>
7.   <!-- propriétés de l'unité de persistance -->
8. ...

```

- lignes 5 et 6 : les deux entités gérées

L'exécution de [InitDB] avec le SGBD MySQL5 donne les résultats suivants :

The screenshot shows the Eclipse IDE with two windows. The left window is a 'Console' showing the output of the 'InitDB' application. It lists several entities and their details, such as 'P[4,1,Durant,Sylvie,05/07/2001,false,0,5]' and 'A[3,1,rue Boileau,null,null,49000,Angers,null,France]'. A red box highlights the number '1' at the bottom. The right window is a 'Database Structure' for 'mysql5-jpa'. It shows the MySQL schema with tables like 'jpa02\_tb\_personne', 'jpa02\_tb\_adresse', 'jpa03\_tb\_personne', 'jpa04\_tb\_adresse', and 'jpa04\_tb\_adresse'. A red box highlights the table 'jpa04\_tb\_adresse'. The bottom right pane shows a file tree for 'ddl' (containing 'derby', 'firebird', 'hsqldb', 'mysql5' with 'create.sql' and 'drop.sql', 'oracle', 'postgres', and 'sqlserver'). A red box highlights the number '2' next to the MySQL schema. A red box also highlights the 'mysql5' folder in the file tree.

En [1], l'affichage console, en [2], les deux tables [jpa04\_t] générées, en [3] les scripts SQL générés. Leur contenu est le suivant :

### create.sql

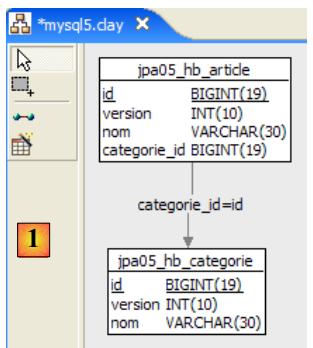
```
1. CREATE TABLE jpa04_t1_personne (ID BIGINT NOT NULL, PRENOM VARCHAR(30) NOT NULL, DATENAISSEANCE DATE NOT NULL, VERSION INTEGER NOT NULL, MARIE TINYINT(1) default 0 NOT NULL, NBENFANTS INTEGER NOT NULL, NOM VARCHAR(30) UNIQUE NOT NULL, adresse_id BIGINT UNIQUE NOT NULL, PRIMARY KEY (ID))
2. CREATE TABLE jpa04_t1_adresse (ID BIGINT NOT NULL, ADR3 VARCHAR(30), CODEPOSTAL VARCHAR(5) NOT NULL, ADR1 VARCHAR(30) NOT NULL, VILLE VARCHAR(20) NOT NULL, VERSION INTEGER NOT NULL, CEDEX VARCHAR(3), ADR2 VARCHAR(30), PAYS VARCHAR(20) NOT NULL, PRIMARY KEY (ID))
3. ALTER TABLE jpa04_t1_personne ADD CONSTRAINT FK_jpa04_t1_personne_adresse_id FOREIGN KEY (adresse_id) REFERENCES jpa04_t1_adresse (ID)
4. CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(38), PRIMARY KEY (SEQ_NAME))
5. INSERT INTO SEQUENCE(SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 1)
```

### drop.sql

```
1. ALTER TABLE jpa04_t1_personne DROP FOREIGN KEY FK_jpa04_t1_personne_adresse_id
2. DROP TABLE jpa04_t1_personne
3. DROP TABLE jpa04_t1_adresse
4. DELETE FROM SEQUENCE WHERE SEQ_NAME = 'SEQ_GEN'
```

## 2.4 Exemple 3 : relation un-à-plusieurs

### 2.4.1 Le schéma de la base de données



```
1. alter table jpa06_article
2. drop
3. foreign key FKFFBDD9D8ECCE8750;
4.
5. drop table if exists jpa06_article;
6.
7. drop table if exists jpa06_categorie;
8.
9. create table jpa06_article (
10.    id bigint not null auto_increment,
11.    version integer not null,
12.    nom varchar(30),
13.    categorie_id bigint not null,
14.    primary key (id)
15. ) ENGINE=InnoDB;
16.
17. create table jpa06_categorie (
18.    id bigint not null auto_increment,
19.    version integer not null,
20.    nom varchar(30),
21.    primary key (id)
22. ) ENGINE=InnoDB;
23.
24. alter table jpa06_article
25.     add index FKFFBDD9D8ECCE8750 (categorie_id),
26.     add constraint FKFFBDD9D8ECCE8750
27.         foreign key (categorie_id)
28.             references jpa06_categorie (id);
```

- en [1], la base de données et en [2], sa DDL (MySQL5)

Un **article** A(id, version, nom) appartient exactement à une catégorie C(id, version, nom). Une **catégorie** C peut contenir 0, 1 ou plusieurs articles. On a une relation **un-à-plusieurs** (Categorie -> Article) et la relation inverse **plusieurs-à-un** (Article -> Categorie). Cette relation est matérialisée par la clé étrangère que possède la table [article] sur la table [categorie] (lignes 24-28 de la DDL).

### 2.4.2 Les objets @Entity représentant la base de données

Un article est représenté par l'**@Entity** [Article] suivante :

```
1. package entites;
2.
3. ...
4. @Entity
5. @Table(name="jpa05_hb_article")
6. public class Article implements Serializable {
7.
8.     // champs
9.     @Id
10.    @GeneratedValue(strategy = GenerationType.AUTO)
11.    private Long id;
12.
13.    @SuppressWarnings("unused")
14.    @Version
15.    private int version;
16.
17.    @Column(length = 30)
18.    private String nom;
19.
20.    // relation principale Article (many) -> Category (one)
21.    // implémentée par une clé étrangère (categorie_id) dans Article
22.    // 1 Article a nécessairement 1 Catégorie (nullable=false)
23.    @ManyToOne(fetch=FetchType.LAZY)
24.    @JoinColumn(name = "categorie_id", nullable = false)
25.    private Categorie categorie;
26.
27.    // constructeurs
28.    public Article() {
```

```

29. }
30.
31. // getters et setters
32. ...
33. // toString
34. public String toString() {
35.     return String.format("Article[%d,%d,%s,%d]", id, version, nom, categorie.getId());
36. }
37.
38. }

```

- lignes 9-11 : clé primaire de l'@Entity
- lignes 13-15 : son n° de version
- lignes 17-18 : nom de l'article
- lignes 20-25 : relation **plusieurs-à-un** qui relie l'@Entity *Article* à l'@Entity *Categorie* :
  - ligne 23 : l'annotation **ManyToOne**. Le **Many** se rapport à l'@Entity *Article* dans lequel on se trouve et le **One** à l'@Entity *Categorie* (ligne 25). Une catégorie (One) peut avoir plusieurs articles (Many).
  - ligne 24 : l'annotation **ManyToOne** définit la colonne clé étrangère dans la table [article]. Elle s'appellera (*name*) *categorie\_id* et chaque ligne devra avoir une valeur dans cette colonne (*nullable=false*).
  - ligne 25 : la catégorie à laquelle appartient l'article. Lorsqu'un article sera mis dans le contexte de persistance, on demande à ce que sa catégorie n'y soit pas mise immédiatement (*fetch=FetchType.LAZY*, ligne 23). On ne sait pas si cette demande a un sens. On verra.

Une catégorie est représentée par l'@Entity [Categorie] suivante :

```

1. package entites;
2. ...
3. @Entity
4. @Table(name="jpa05_hb_categorie")
5. public class Categorie implements Serializable {
6.
7.     // champs
8.     @Id
9.     @GeneratedValue(strategy = GenerationType.AUTO)
10.    private Long id;
11.
12.    @SuppressWarnings("unused")
13.    @Version
14.    private int version;
15.
16.    @Column(length = 30)
17.    private String nom;
18.
19.    // relation inverse Categorie (one) -> Article (many) de la relation Article (many) -> Categorie
20.    // (one)
21.    // cascade insertion Categorie -> insertion Articles
22.    // cascade maj Categorie -> maj Articles
23.    // cascade suppression Categorie -> suppression Articles
24.    @OneToMany(mappedBy = "categorie", cascade = { CascadeType.ALL })
25.    private Set<Article> articles = new HashSet<Article>();
26.
27.    // constructeurs
28.    public Categorie() {
29.    }
30.
31.    // getters et setters
32.    ...
33.    // toString
34.    public String toString() {
35.        return String.format("Categorie[%d,%d,%s]", id, version, nom);
36.    }
37.
38.    // association bidirectionnelle Categorie <-> Article
39.    public void addArticle(Article article) {
40.        // l'article est ajouté dans la collection des articles de la catégorie
41.        articles.add(article);
42.        // l'article change de catégorie
43.        article.setCategorie(this);
44.    }

```

- lignes 8-11 : la clé primaire de l'@Entity
- lignes 12-14 : sa version
- lignes 16-17 : le nom de la catégorie
- lignes 19-24 : l'ensemble (set) des articles de la catégorie

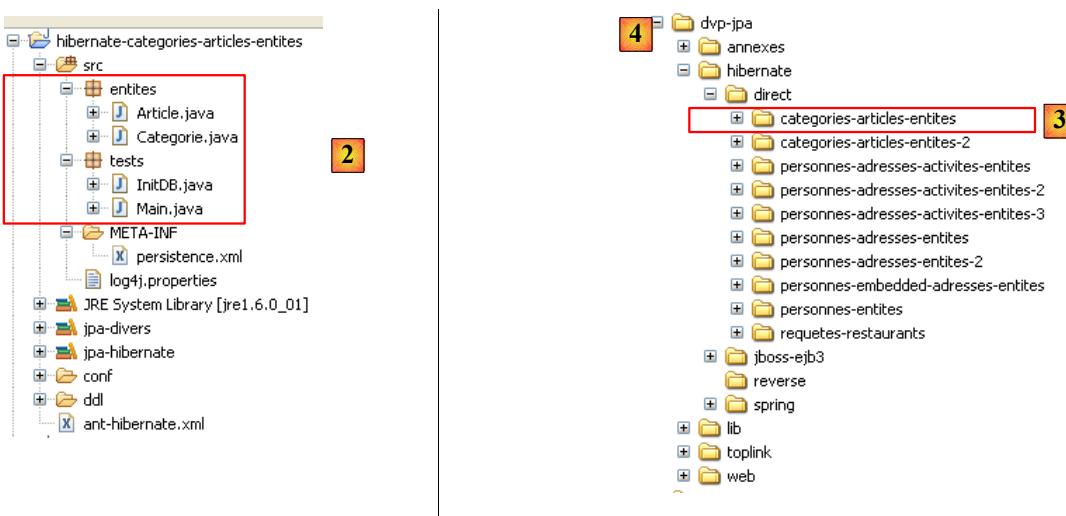
- ligne 23 : l'annotation `@OneToOne` désigne une relation **un-à-plusieurs**. Le **One** désigne l'`@Entity [Categorie]` dans laquelle on se trouve, le **Many** le type `[Article]` de la ligne 24 : une (One) catégorie a plusieurs (Many) articles.
- ligne 23 : l'annotation est l'inverse (`mappedBy`) de l'annotation `ManyToOne` placée sur le champ `categorie` de l'`@Entity Article` : `mappedBy=categorie`. La relation `ManyToOne` placée sur le champ `categorie` de l'`@Entity Article` est la **relation principale**. Elle est indispensable. Elle matérialise la relation de clé étrangère qui lie l'`@Entity Article` à l'`@Entity Categorie`. La relation `OneToMany` placée sur le champ `articles` de l'`@Entity Categorie` est la **relation inverse**. Elle n'est pas indispensable. C'est une commodité pour obtenir les articles d'une catégorie. Sans cette commodité, ces articles seraient obtenus par une requête JPQL.
- ligne 23 : `cascadeType.ALL` demande à que les opérations (persist, merge, remove) faites sur une `@Entity Categorie` soient cascadées sur ses articles.
- ligne 24 : les articles d'une catégorie seront placés dans un objet de type `Set<Article>`. Le type `Set` n'accepte pas les doublons. Ainsi on ne peut mettre deux fois le même article dans l'objet `Set<Article>`. Que veut dire "le même article" ? Pour dire que l'article **a** est le même que l'article **b**, Java utilise l'expression `a.equals(b)`. Dans la classe `Object`, mère de toutes les classes, `a.equals(b)` est vraie si `a==b`, c.a.d. si les objets *a* et *b* ont le même emplacement mémoire. On pourrait vouloir dire que les articles *a* et *b* sont les mêmes s'ils ont le même nom. Dans ce cas, le développeur doit redéfinir deux méthodes dans la classe `[Article]` :
  - `equals` : qui doit rendre vrai si les deux articles ont le même nom
  - `hashCode` : doit rendre une valeur entière identique pour deux objets `[Article]` que la méthode `equals` considère comme égaux. Ici, la valeur sera donc construite à partir du nom de l'article. La valeur rendue par `hashCode` peut être un entier quelconque. Elle est utilisée dans différents conteneurs d'objets, notamment les dictionnaires (Hashtable).

La relation `OneToMany` peut utiliser d'autres types que le `Set` pour stocker le `Many`, des objets `List`, par exemple. Nous n'aborderons pas ces cas dans ce document. Le lecteur les trouvera dans [ref1].

- ligne 38 : la méthode `[addArticle]` nous permet d'ajouter un article à une catégorie. La méthode prend soin de mettre à jour les deux extrémités de la relation `OneToMany` qui lie `[Categorie]` à `[Article]`.

### 2.4.3 Le projet Eclipse / Hibernate 1

L'implémentation JPA utilisée ici est celle d'Hibernate. Le projet Eclipse des tests est le suivant :



Le projet est présent [3] dans le dossier des exemples [4]. On l'importera.

### 2.4.4 Génération de la DDL de la base de données

En suivant les instructions du paragraphe 2.1.7, page 20, la DDL obtenue pour le SGBD MySQL5 est celle montrée au début de cette exemple, au paragraphe 2.4.1, page 94.

### 2.4.5 InitDB

Le code de `[InitDB]` est le suivant :

```
1. package tests;
2.
3. ...
```

```

4. public class InitDB {
5.
6.     // constantes
7.     private final static String TABLE_ARTICLE = "jpa05_hb_article";
8.
9.     private final static String TABLE_CATEGORIE = "jpa05_hb_categorie";
10.
11.    public static void main(String[] args) {
12.        // Contexte de persistance
13.        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
14.        EntityManager em = null;
15.        // on récupère un EntityManager à partir de l'EntityManagerFactory précédent
16.        em = emf.createEntityManager();
17.        // début transaction
18.        EntityTransaction tx = em.getTransaction();
19.        tx.begin();
20.        // requête
21.        Query sql1;
22.        // supprimer les éléments de la table ARTICLE
23.        sql1 = em.createNativeQuery("delete from " + TABLE_ARTICLE);
24.        sql1.executeUpdate();
25.        // supprimer les éléments de la table CATEGORIE
26.        sql1 = em.createNativeQuery("delete from " + TABLE_CATEGORIE);
27.        sql1.executeUpdate();
28.        // créer trois catégories
29.        Categorie categorieA = new Categorie();
30.        categorieA.setNom("A");
31.        Categorie categorieB = new Categorie();
32.        categorieB.setNom("B");
33.        Categorie categorieC = new Categorie();
34.        categorieC.setNom("C");
35.        // créer 3 articles
36.        Article articleA1 = new Article();
37.        articleA1.setNom("A1");
38.        Article articleA2 = new Article();
39.        articleA2.setNom("A2");
40.        Article articleB1 = new Article();
41.        articleB1.setNom("B1");
42.        // les relier à leur catégorie
43.        categorieA.addArticle(articleA1);
44.        categorieA.addArticle(articleA2);
45.        categorieB.addArticle(articleB1);
46.        // persister les catégories et par cascade (insertion) les articles
47.        em.persist(categorieA);
48.        em.persist(categorieB);
49.        em.persist(categorieC);
50.        // affichage catégories
51.        System.out.println("[categories]");
52.        for (Object p : em.createQuery("select c from Categorie c order by c.nom
asc").getResultList()) {
53.            System.out.println(p);
54.        }
55.        // affichage articles
56.        System.out.println("[articles]");
57.        for (Object p : em.createQuery("select a from Article a order by a.nom asc").getResultList())
{
58.            System.out.println(p);
59.        }
60.        // fin transaction
61.        tx.commit();
62.        // fin EntityManager
63.        em.close();
64.        // fin EntityMangerFactory
65.        emf.close();
66.        // log
67.        System.out.println("terminé...");
68.
69.    }
70. }

```

- lignes 22-27 : les tables [article] et [categorie] sont vidées. On notera qu'on est obligés de commencer par celle qui a la clé étrangère. Si on commençait par la table [categorie] on supprimerait des catégories référencées par des lignes de la table [article] et cela le SGBD le refuserait.
- lignes 29-34 : on crée trois catégories A, B, C
- lignes 36-41 : on crée trois articles A1, A2, B1 (la lettre indique la catégorie)
- lignes 43-45 : les 3 articles sont mis dans leurs catégories respectives
- lignes 47-49 : les 3 catégories sont mises dans le contexte de persistance. A cause de la cascade Categorie -> Article, leurs articles vont y être placés également. Donc tous les objets créés sont maintenant dans le contexte de persistance.

- lignes 50-59 : le contexte de persistance est requêté pour obtenir la liste des catégories et articles. On sait que cela va provoquer une synchronisation du contexte avec la base. C'est à ce moment que les catégories et articles vont être enregistrés dans leurs tables respectives.

L'exécution de [InitDB] avec MySQL5 donne les résultats suivants :

The screenshot shows the Eclipse IDE interface with several windows open:

- Console View (1):** Displays the output of the [InitDB] Java code, which creates three categories (A, B, C) and three articles (A1, A2, B1). A red box labeled '1' highlights the console output.
- Database Structure View (2):** Shows the database schema. It includes a tree view of MySQL 5.0 schema, the jpa schema, and its Tables folder. Inside the Tables folder, there are many temporary tables starting with 'jpa05\_hb\_'. A red box labeled '2' highlights the 'Tables' folder.
- Database Detail View (3):** Shows the 'Categories' table with three rows: id=1, version=0, nom=A; id=2, version=0, nom=B; id=3, version=0, nom=C. A red box labeled '3' highlights this view.
- Database Detail View (4):** Shows the 'Articles' table with three rows: id=1, version=0, nom=A1, categorie\_id=1; id=2, version=0, nom=A2, categorie\_id=1; id=3, version=0, nom=B1, categorie\_id=2. A red box labeled '4' highlights this view.

- [1] : l'affichage console
- [2] : les tables [jpa05\_hb\_\*] dans la perspective SQL Explorer
- [3] : la table des catégories
- [4] : la table des articles. On notera le lien de [categorie\_id] dans [4] avec [id] dans [3] (clé étrangère).

## 2.4.6 Main

La classe [Main] enchaîne des tests que nous passons en revue sauf les tests 1 et 2 qui reprennent le code de [InitDB] pour initialiser la base.

### 2.4.6.1 Test3

Ce test est le suivant :

```
1. // rechercher un élément particulier
2. public static void test3() {
3.     // nouveau contexte de persistance
4.     EntityManager em = getNewEntityManager();
5.     // transaction
6.     EntityTransaction tx = em.getTransaction();
7.     tx.begin();
8.     // chargement catégorie
9.     Categorie categorie = em.find(Categorie.class, categorieA.getId());
10.    // affichage catégorie et ses articles associés
11.    System.out.format("Articles de la catégorie %s :%n", categorie);
12.    for (Article a : categorie.getArticles()) {
13.        System.out.println(a);
14.    }
15.    // fin transaction
16.    tx.commit();
17.}
```

- ligne 4 : on a un contexte de persistance neuf donc vide
- lignes 6-7 : début transaction
- ligne 9 : la catégorie A est amenée de la base dans le contexte de persistance
- ligne 11 : on affiche la catégorie A
- lignes 12-14 : on affiche les articles de la catégorie A. On montre là l'intérêt de la relation inverse **OneToMany articles** de l'@Entity *Categorie*. Sa présence nous évite de faire une requête JPQL pour demander les articles de la catégorie A. Pour obtenir ceux-ci, on utilise la méthode *get* du champ *articles*.

Les résultats sont les suivants :

```
1. main : -----
2. [categories]
3. Categorie[1,0,A]
```

```

4. Categorie[2,0,B]
5. Categorie[3,0,C]
6. [articles]
7. Article[1,0,A1,1]
8. Article[2,0,A2,1]
9. Article[3,0,B1,2]
10. main : -----
11. 3 categorie(s) trouvée(s) :
12. A
13. B
14. C
15. 3 article(s) trouvé(s) :
16. A1
17. A2
18. B1
19. main : -----
20. Articles de la catégorie Categorie[1,0,A] :
21. Article[2,0,A2,1]
22. Article[1,0,A1,1]

```

- ligne 20 : la catégorie A
- lignes 21-22 : les deux articles de la catégorie A

#### 2.4.6.2 Test4

Ce test est le suivant :

```

1. // supprimer un article
2. @SuppressWarnings("unchecked")
3. public static void test4() {
4.     // nouveau contexte de persistance
5.     EntityManager em = getNewEntityManager();
6.     // transaction
7.     EntityTransaction tx = em.getTransaction();
8.     tx.begin();
9.     // chargement article A1
10.    Article newarticle1 = em.find(Article.class, articleA1.getId());
11.    // suppression article A1 (aucune catégorie n'est actuellement chargée)
12.    em.remove(newarticle1);
13.    // toplink : l'article doit être enlevé de sa catégorie sinon le test6 plante
14.    // hibernate : ce n'est pas nécessaire
15.    newarticle1.getCategory().getArticles().remove(newarticle1);
16.    // fin transaction
17.    tx.commit();
18.    // dump des articles
19.    dumpArticles();
20. }

```

- le test 4 supprime l'article A1
- ligne 5 : on part d'un contexte neuf et vide
- ligne 10 : l'article A1 est amené dans le contexte de persistance. Il y sera référencé par *newarticle1*.
- ligne 12 : il est supprimé du contexte
- ligne 15 : les catégories A, B et C et les articles A1, A2 et B1, s'ils ne sont plus persistants sont néanmoins encore en mémoire. Ils sont simplement détachés du contexte de persistance. L'article A1 qui fait partie des articles de la catégorie A en est enlevé. Cela va rendre possible ultérieurement le réattachement de la catégorie A au contexte de persistance. Si on ne le fait pas, la catégorie A sera rattachée avec un ensemble d'articles dont l'un a été supprimé. Cela ne semble pas gêner Hibernate mais plante Toplink.
- ligne 19 : on affiche tous les articles pour vérifier que A1 a disparu.

Les résultats sont les suivants :

```

1. main : -----
2. [articles]
3. Article[2,0,A2,1]
4. Article[3,0,B1,2]

```

L'article A1 a bien disparu.

#### 2.4.6.3 Test5

Ce test est le suivant :

```

1.// modification d'un article
2. public static void test5() {
3.     // nouveau contexte de persistance
4.     EntityManager em = getNewEntityManager();
5.     // transaction

```

```

6. EntityTransaction tx = em.getTransaction();
7. tx.begin();
8. // modification articleA2
9. articleA2.setNom(articleA2.getNom() + "-");
10. // articleA2 est remis dans le contexte de persistance
11. em.merge(articleA2);
12. // fin transaction
13. tx.commit();
14. // dump des articles
15. dumpArticles();
16. }

```

- le test 5 change le nom de l'article A2
- ligne 4 : on part d'un contexte neuf et vide
- ligne 9 : on change le nom de l'article détaché A2 qui va devenir "A2-".
- ligne 11 : l'article détaché A2 est réattaché au contexte de persistance. On notera que A2 reste toujours un objet détaché. C'est l'objet `em.merge(articleA2)` qui fait partie désormais du contexte de persistance. Cet objet n'a pas été ici mémorisé dans une variable comme il est d'usage. Il est donc inaccessible.
- ligne 13 : synchronisation du contexte de persistance avec la base. L'article A2 va être modifié dans la base et voir son n° de version passer de N à N+1. La version mémoire détachée `articleA2` n'est plus valide. Il en est de même de l'objet détaché représentant la catégorie A parce que celui-ci contient `articleA2` parmi ses articles.
- ligne 15 : on affiche tous les articles pour vérifier le changement de nom de l'article A2

Les résultats sont les suivants :

```

1. main : -----
2. [articles]
3. Article[2,1,A2-,1]
4. Article[3,0,B1,2]

```

L'article A2 a bien changé de nom.

#### 2.4.6.4 Test6

Ce test est le suivant :

```

1.// modification d'une catégorie et de ses articles
2.public static void test6() {
3.    // nouveau contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // chargement catégorie
9.    categorieA = em.find(Categorie.class, categorieA.getId());
10.   // liste des articles de la catégorie A
11.   for (Article a : categorieA.getArticles()) {
12.       a.setNom(a.getNom() + "-");
13.   }
14.   // modification nom catégorie
15.   categorieA.setNom(categorieA.getNom() + "-");
16.   // fin transaction
17.   tx.commit();
18.   // dump des catégories et des articles
19.   dumpCategories();
20.   dumpArticles();
21. }

```

- le test 6 change le nom de la catégorie A et de tous ses articles
- ligne 4 : on part d'un contexte neuf et vide
- ligne 9 : on va chercher la catégorie A dans la base. On ne fait pas un `merge` de l'objet détaché `categorieA` car on sait qu'il a une référence sur l'article A2 devenu obsolète. On repart donc de zéro.
- lignes 11-12 : on change le nom de tous les articles de la catégorie A. De nouveau on utilise la relation inverse **OneToMany** via la méthode `getArticles`.
- ligne 15 : le nom de la catégorie est également modifié
- ligne 17 : fin de la transaction. Une synchronisation du contexte avec la base est fait. Tous les objets du contexte qui ont été modifiés vont être mis à jour dans la base.
- lignes 21-22 : on affiche les articles et les catégories pour vérification

Les résultats sont les suivants :

```

1. main : -----
2. [categories]
3. Categorie[1,2,A-]

```

```

4. Categorie[2,0,B]
5. Categorie[3,0,C]
6. [articles]
7. Article[2,2,A2--,1]
8. Article[3,0,B1,2]

```

L'article A2 a bien changé une nouvelle fois de nom ainsi que la catégorie A.

### 2.4.6.5 Test7

Ce test est le suivant :

```

1.// suppression d'une catégorie
2.public static void test7() {
3.    // nouveau contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // persistance catégorieB et par cascade (merge) les articles associés
9.    Categorie mergedcategorieB = em.merge(categorieB);
10.   // suppression catégorie et par cascade (delete) les articles associés
11.   em.remove(mergedcategorieB);
12.   // fin transaction
13.   tx.commit();
14.   // dump des catégories et des articles
15.   dumpCategories();
16.   dumpArticles();
17. }

```

- le test 7 supprime la catégorie B et par cascade ses articles
- ligne 4 : on part d'un contexte neuf et vide
- ligne 9 : la catégorie B existe en mémoire en tant qu'objet détaché du contexte de persistance. On la réintègre (*merge*) au contexte de persistance. Par cascade, ses articles (l'article B1) vont subir un *merge* et donc réintégrer le contexte de persistance.
- ligne 11 : maintenant que la catégorie B est dans le contexte, on peut la supprimer (*remove*). Par cascade, ses articles vont également subir un *remove*. C'est parce que l'opération *merge* de la ligne 9 les a réintégrés au contexte de persistance que cette opération est possible.
- ligne 13 : fin de la transaction. Le contexte va être synchronisé. Les objets du contexte ayant subi un *remove* vont être supprimés de la base.
- lignes 15-16 : on affiche les articles et les catégories pour vérification

Les résultats sont les suivants :

```

1. main : -----
2. [categories]
3. Categorie[1,2,A-]
4. Categorie[3,0,C]
5. [articles]
6. Article[1,2,A2--,1]

```

La catégorie B et l'article B1 ont bien disparu.

### 2.4.6.6 Test8

Ce test est le suivant :

```

1.// requêtes
2.@SuppressWarnings("unchecked")
3.public static void test8() {
4.    // nouveau contexte de persistance
5.    EntityManager em = getNewEntityManager();
6.    // transaction
7.    EntityTransaction tx = em.getTransaction();
8.    tx.begin();
9.    // liste des articles de la catégorie A
10.   List articles = em
11.       .createQuery(
12.           "select a from Categorie c join c.articles a where c.nom like 'A%' order by a.nom
13.           asc")
13.       .getResultList();
14.   // affichages articles
15.   System.out.println("Articles de la catégorie A");
16.   for (Object a : articles) {
17.       System.out.println(a);
18.   }
19.   // fin transaction
20.   tx.commit();

```

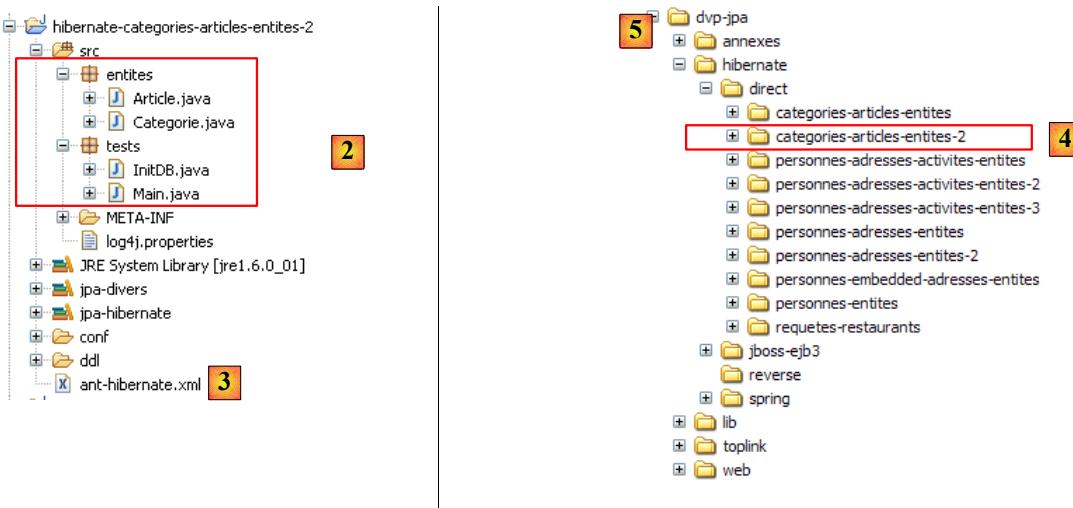
- le test 7 montre comment récupérer les articles d'une catégorie sans passer par la relation inverse. Cela montre que celle-ci n'est donc pas indispensable.
- ligne 4 : on part d'un contexte neuf et vide
- ligne 10 : une requête JPQL qui demande tous les articles d'une catégorie ayant un nom commençant par A
- lignes 15-17 : affichage du résultat de la requête.

Les résultats sont les suivants :

```
1. main : ----- test8
2. Articles de la catégorie A
3. Article[2,2,A2--,1]
```

## 2.4.7 Projet Eclipse / Hibernate 2

Nous copions / collons le projet Eclipse / Hibernate afin de préciser un point sur la notion relation principale / relation inverse que nous avons créée autour de l'annotation **@ManyToOne** (principale) de l'**@Entity [Article]** et la relation inverse **@OneToMany**(inverse) de l'**@Entity [Categorie]**. Nous voulons montrer que si cette dernière relation n'est pas déclarée inverse de l'autre, alors le schéma généré pour la base de données est tout autre que celui généré précédemment.



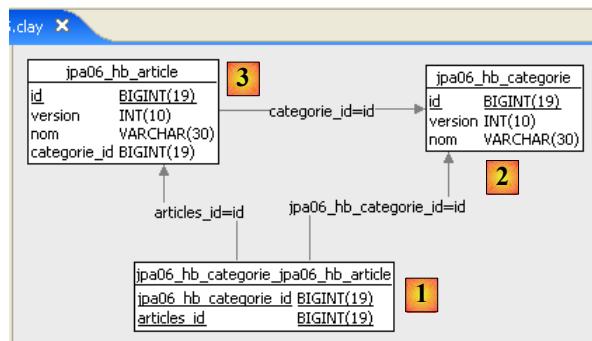
En [1] le nouveau projet Eclipse. En [2] les codes Java, en [3] le script *ant* qui va générer le schéma SQL de la base de données. Le projet est présent [4] dans le dossier des exemples [5]. On l'importera.

Nous modifions uniquement l'**@Entity [Categorie]** afin que sa relation **@OneToMany** avec l'**@Entity [Article]** ne soit plus déclarée inverse de la relation **@ManyToOne** qu'a l'**@Entity [Article]** avec l'**@Entity [Categorie]** :

```
1. ...
2. @Entity
3. @Table(name="jpa05_hb_categorie")
4. public class Categorie implements Serializable {
5.
6.     // champs
7.     @Id
8.     @GeneratedValue(strategy = GenerationType.AUTO)
9.     private Long id;
10.
11.    @SuppressWarnings("unused")
12.    @Version
13.    private int version;
14.
15.    @Column(length = 30)
16.    private String nom;
17.
18.    // relation OneToMany non inverse (absence de mappedBy) Categorie (one) -> Article (many)
19.    // implémentée par une table de jointure Categorie_Article pour qu'à partir d'une catégorie
20.    // on puisse atteindre les articles de cette catégorie
21.    @OneToMany(cascade=CCascadeType.ALL, fetch=FetchType.LAZY)
22.    private Set<Article> articles = new HashSet<Article>();
23.
24.    // constructeurs
25.    ...
```

- lignes 18-22 : on veut encore garder la possibilité de trouver les articles d'une catégorie donnée grâce à la relation **@OneToMany** de la ligne 21. Mais on veut connaître l'influence de l'attribut **mappedBy** qui fait d'une relation, l'inverse d'une relation principale définie ailleurs, dans une autre **@Entity**. Ici, le **mappedBy** a été enlevé.

Nous exécutons la tâche **ant-DLL** (cf paragraphe 2.1.7, page 20) avec le SGBD MySQL5. Le schéma obtenu est le suivant :



On notera les points suivants :

- une nouvelle table [categorie\_article] [1] a été créée. Elle n'existe pas auparavant.
- c'est une table de jointure entre les tables [categorie] [2] et [article] [3]. Si les objets Article  $a_1, a_2$  font partie de la catégorie  $c_1$ , on trouvera dans la table de jointure, les lignes :

```
[c1,a1]
[c1,a2]
```

où  $c_1, a_1, a_2$  sont les clés primaires des objets correspondants.

- la table de jointure [categorie\_article] [1] a été créée par Hibernate afin qu'à partir d'un objet *Categorie c*, on puisse retrouver les objets *Article a* appartenant à *c*. C'est la relation **@OneToMany** qui a forcé la création de cette table. Parce qu'on ne l'a pas déclarée **inverse** de la relation principale **@ManyToOne** de l'**@Entity Article**, Hibernate ne savait pas qu'il pouvait utiliser cette relation principale pour récupérer les articles d'une catégorie *c*. Il s'est donc débrouillé autrement.
- avec cet exemple, on comprend mieux les notions de relations *principale* et *inverse*. L'une (l'inverse) utilise les propriétés de l'autres (la principale).

Le schéma SQL de cette base de données pour MySQL5 est le suivant :

```

1. alter table jpa05_hb_categorie_jpa06_hb_article
2.   drop
3.   foreign key FK79D4BA1D26D17756;
4.
5. alter table jpa05_hb_categorie_jpa06_hb_article
6.   drop
7.   foreign key FK79D4BA1D424C61C9;
8.
9. alter table jpa06_hb_article
10.  drop
11.  foreign key FK4547168FECCE8750;
12.
13. drop table if exists jpa05_hb_categorie;
14.
15. drop table if exists jpa05_hb_categorie_jpa06_hb_article;
16.
17. drop table if exists jpa06_hb_article;
18.
19. create table jpa05_hb_categorie (
20.   id bigint not null auto_increment,
21.   version integer not null,
22.   nom varchar(30),
23.   primary key (id)
24. ) ENGINE=InnoDB;
25.
26. create table jpa05_hb_categorie_jpa06_hb_article (
27.   jpa05_hb_categorie_id bigint not null,
28.   articles_id bigint not null,
29.   primary key (jpa05_hb_categorie_id, articles_id),
30.   unique (articles_id)
31. ) ENGINE=InnoDB;
32.
33. create table jpa06_hb_article (
34.   id bigint not null auto_increment,
35.   version integer not null,
36.   nom varchar(30),
37.   categorie_id bigint not null,
```

```

38.      primary key (id)
39. ) ENGINE=InnoDB;
40.
41. alter table jpa05_hb_categorie_jpa06_hb_article
42.   add index FK79D4BA1D26D17756 (jpa05_hb_categorie_id),
43.   add constraint FK79D4BA1D26D17756
44.     foreign key (jpa05_hb_categorie_id)
45.       references jpa05_hb_categorie (id);
46.
47. alter table jpa05_hb_categorie_jpa06_hb_article
48.   add index FK79D4BA1D424C61C9 (articles_id),
49.   add constraint FK79D4BA1D424C61C9
50.     foreign key (articles_id)
51.       references jpa06_hb_article (id);
52.
53. alter table jpa06_hb_article
54.   add index FK4547168FECCE8750 (categorie_id),
55.   add constraint FK4547168FECCE8750
56.     foreign key (categorie_id)
57.       references jpa05_hb_categorie (id);

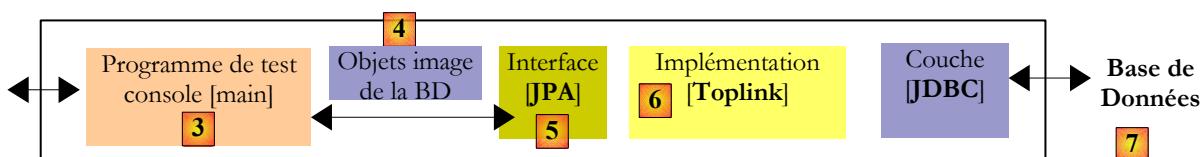
```

- lignes 19-24, création de la table [categorie] et lignes 33-39, création de la table [article]. On notera qu'elles sont identiques à ce qu'elles étaient dans l'exemple précédent.
- lignes 26-31 : création de la table de jointure [categorie\_article] due à la présence de la relation non inverse **@OneToMany** de l'**@Entity Categorie**. Les lignes de cette table sont de type **[c,a]** où **c** est la clé primaire d'une catégorie **c** et **a** la clé primaire d'un article **a** appartenant à la catégorie **c**. La clé primaire de cette table de jointure est constituée des deux clés primaires **[c,a]** concaténées (ligne 29).
- lignes 41-45 : la contrainte de clé étrangère de la table [categorie\_article] vers la table [categorie]
- lignes 47-51 : la contrainte de clé étrangère de la table [categorie\_article] vers la table [article]
- lignes 53-57 : la contrainte de clé étrangère de la table [article] vers la table [categorie]

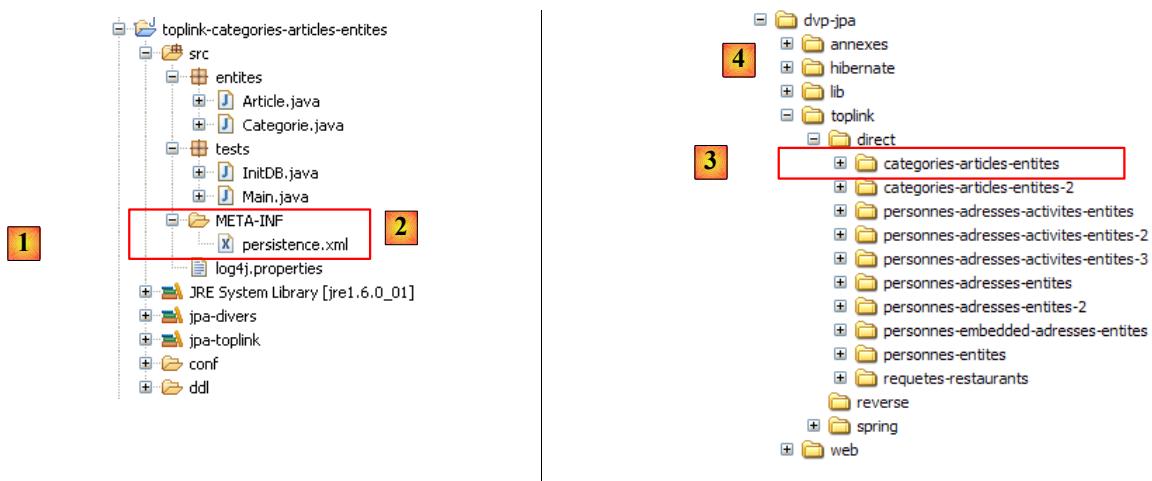
Le lecteur est invité à exécuter les tests [InitDB] et [Main]. Ils donnent les mêmes résultats qu'auparavant. Le schéma de la base de données est cependant redondant et les performances seront dégradées vis à vis de la version précédente. Il faudrait sans doute approfondir cette question de relations *inverse / principale* pour voir si la nouvelle configuration n'amène pas de plus, des conflits dus au fait qu'on a deux relations indépendantes pour représenter la même chose : la relation **plusieurs-à-un** qu'a la table [article] avec la table [categorie].

## 2.4.8 Implémentation JPA / Toplink - 1

Nous utilisons maintenant une implémentation JPA / Toplink :



Le projet Eclipse avec Toplink est une copie du projet Eclipse avec Hibernate, version 1 :



Les codes Java sont identiques à ceux du projet Hibernate - version 1 - précédent. L'environnement (bibliothèques – persistence.xml – sgbds – dossiers conf, ddl – script ant) est celui étudié au paragraphe 2.1.15.2, page 58. Le projet Eclipse est présent [3] dans le dossier des exemples [4]. On l'importera.

Le fichier <persistence.xml> [2] est modifié en un point, celui des entités déclarées :

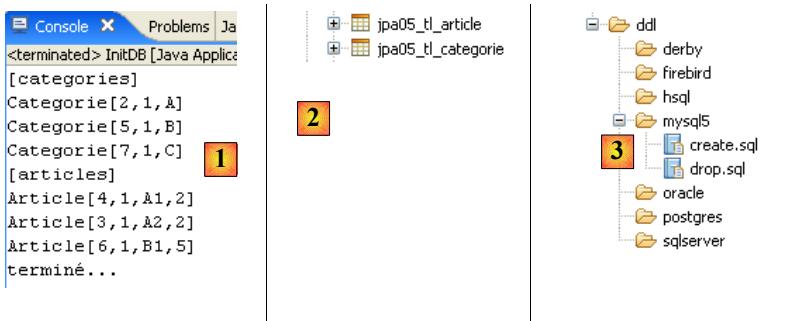
```

1. ...
2.   <!-- classes persistantes -->
3.   <class>entites.Categorie</class>
4.   <class>entites.Article</class>
5. ...

```

- lignes 3 et 4 : les deux entités gérées

L'exécution de [InitDB] avec le SGBD MySQL5 donne les résultats suivants :



En [1], l'affichage console, en [2], les deux tables [jpa05\_tl] générées, en [3] les scripts SQL générés. Leur contenu est le suivant :

#### create.sql

```

1. CREATE TABLE jpa05_tl_article (ID BIGINT NOT NULL, VERSION INTEGER, NOM VARCHAR(30), categorie_id
2. BIGINT NOT NULL, PRIMARY KEY (ID))
3. CREATE TABLE jpa05_tl_categorie (ID BIGINT NOT NULL, VERSION INTEGER, NOM VARCHAR(30), PRIMARY KEY
4. (ID))
5. ALTER TABLE jpa05_tl_article ADD CONSTRAINT FK_jpa05_tl_article_categorie_id FOREIGN KEY
6. (categorie_id) REFERENCES jpa05_tl_categorie (ID)
7. CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(38), PRIMARY KEY
8. (SEQ_NAME))
9. INSERT INTO SEQUENCE(SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 1)

```

#### drop.sql

```

1. ALTER TABLE jpa05_tl_article DROP FOREIGN KEY FK_jpa05_tl_article_categorie_id
2. DROP TABLE jpa05_tl_article
3. DROP TABLE jpa05_tl_categorie
4. DELETE FROM SEQUENCE WHERE SEQ_NAME = 'SEQ_GEN'

```

L'exécution de [Main] se passe sans erreur.

## 2.4.9 Implémentation JPA / Toplink - 2

Ce projet Eclipse est issu du précédent par recopie. Comme il a été fait avec Hibernate, on enlève l'attribut **mappedBy** de la relation **@OneToOne** de l'**@Entity Categorie**.

```

1. @Entity
2. @Table(name = "jpa06_tl_categorie")
3. public class Categorie implements Serializable {
4.
5.   // champs
6.   @Id
7.   @GeneratedValue(strategy = GenerationType.AUTO)
8.   private Long id;
9.
10.  @Version
11.  private int version;
12.
13.  @Column(length = 30)
14.  private String nom;
15.
16.  // relation OneToMany non inverse (absence de mappedby) Categorie (one) ->
17.  // Article (many)
18.  // implémentée par une table de jointure Categorie_Article pour qu'à partir

```

```

19. // d'une catégorie
20. // on puisse atteindre plusieurs articles
21. @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
22. private Set<Article> articles = new HashSet<Article>();

```

Le schéma SQL généré pour MySQL5 est alors le suivant :

### create.sql

```

1. CREATE TABLE jpa06_t1_categorie (ID BIGINT NOT NULL, VERSION INTEGER, NOM VARCHAR(30), PRIMARY KEY
   (ID))
2. CREATE TABLE jpa06_t1_categorie_jpa06_t1_article (Categorie_ID BIGINT NOT NULL, articles_ID BIGINT
   NOT NULL, PRIMARY KEY(Categorie_ID, articles_ID))
3. CREATE TABLE jpa06_t1_article (ID BIGINT NOT NULL, VERSION INTEGER, NOM VARCHAR(30), categorie_id
   BIGINT NOT NULL, PRIMARY KEY (ID))
4. ALTER TABLE jpa06_t1_categorie_jpa06_t1_article ADD CONSTRAINT
   FK_jpa06_t1_categorie_jpa06_t1_article_articles_ID FOREIGN KEY (articles_ID) REFERENCES
   jpa06_t1_article (ID)
5. ALTER TABLE jpa06_t1_categorie_jpa06_t1_article ADD CONSTRAINT
   jpa06_t1_categorie_jpa06_t1_article_Categorie_ID FOREIGN KEY (Categorie_ID) REFERENCES
   jpa06_t1_categorie (ID)
6. ALTER TABLE jpa06_t1_article ADD CONSTRAINT FK_jpa06_t1_article_categorie_id FOREIGN KEY
   (categorie_id) REFERENCES jpa06_t1_categorie (ID)
7. CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(38), PRIMARY KEY
   (SEQ_NAME))
8. INSERT INTO SEQUENCE(SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 1)

```

- ligne 2 : la table de jointure qui matérialise la relation **@OneToMany** non inverse précédente.

L'exécution de [InitDB] se passe sans erreur mais celle de [Main] plante au test 7 avec les logs (FINEST) suivants :

```

1. main : ----- test7
2. [TopLink Finer]: 2007.06.01 01:41:48.734--ServerSession(15290002)--Thread(Thread[main,5,main])--
   client acquired
3. [TopLink Finest]: 2007.06.01 01:41:48.734--UnitOfWork(26285048)--Thread(Thread[main,5,main])--
   Merge clone with references Categorie[5,1,B]
4. [TopLink Finest]: 2007.06.01 01:41:48.734--UnitOfWork(26285048)--Thread(Thread[main,5,main])--
   Register the existing object Article[6,1,B1]
5. [TopLink Finest]: 2007.06.01 01:41:48.734--UnitOfWork(26285048)--Thread(Thread[main,5,main])--
   Register the existing object Categorie[5,1,B]
6. [TopLink Finest]: 2007.06.01 01:41:48.734--UnitOfWork(26285048)--Thread(Thread[main,5,main])--The
   remove operation has been performed on: Categorie[5,1,B]
7. [TopLink Finest]: 2007.06.01 01:41:48.734--UnitOfWork(26285048)--Thread(Thread[main,5,main])--The
   remove operation has been performed on: Article[6,1,B1]
8. [TopLink Finer]: 2007.06.01 01:41:48.750--UnitOfWork(26285048)--Thread(Thread[main,5,main])--begin
   unit of work commit
9. [TopLink Finer]: 2007.06.01 01:41:48.750--ClientSession(15014700)--Connection(6330655)--
   Thread(Thread[main,5,main])--begin transaction
10. [TopLink Finest]: 2007.06.01 01:41:48.750--UnitOfWork(26285048)--Thread(Thread[main,5,main])--
   Execute query DeleteObjectQuery(Article[6,1,B1])
11. [TopLink Fine]: 2007.06.01 01:41:48.750--ClientSession(15014700)--Connection(6330655)--
   Thread(Thread[main,5,main])--DELETE FROM jpa06_t1_article WHERE ((ID = ?) AND (VERSION = ?))
12. bind => [6, 1]
13. [TopLink Warning]: 2007.06.01 01:41:48.750--UnitOfWork(26285048)--Thread(Thread[main,5,main])--
   Local Exception Stack:
14. Exception [TOPLINK-4002] (Oracle TopLink Essentials - 2.0 (Build b41-beta2 (03/30/2007))):  

   oracle.toplink.essentials.exceptions.DatabaseException
15. Internal Exception: com.mysql.jdbc.exceptions.MySQLIntegrityConstraintViolationException: Cannot
   delete or update a parent row: a foreign key constraint fails
   (`jpa/jpa06_t1_categorie_jpa06_t1_article`, CONSTRAINT
   `FK_jpa06_t1_categorie_jpa06_t1_article_articles_ID` FOREIGN KEY (`articles_ID`) REFERENCES
   `jpa06_t1_article` (`ID`))
16. Error Code: 1451
17. Call: DELETE FROM jpa06_t1_article WHERE ((ID = ?) AND (VERSION = ?))
18. bind => [6, 1]

```

- ligne 3 : le *merge* sur la catégorie B
- ligne 4 : l'article dépendant B1 est mis dans le contexte
- ligne 5 : idem pour la catégorie B elle-même
- ligne 6 : le *remove* sur la catégorie B
- ligne 7 : le *remove* sur l'article B1 (par cascade)
- ligne 8 : le *commit* de la transaction est demandé par le code Java
- ligne 9 : une transaction démarre - elle n'avait donc apparemment pas commencé.
- ligne 10 : l'article B1 va être détruit par une opération DELETE sur la table [article]. C'est là qu'est le problème. La table de jointure [categorie\_article] a une référence sur la ligne B1 de la table [article]. La suppression de B1 dans [article] va enfreindre une contrainte de clé étrangère.
- lignes 13 et au-delà : l'exception se produit

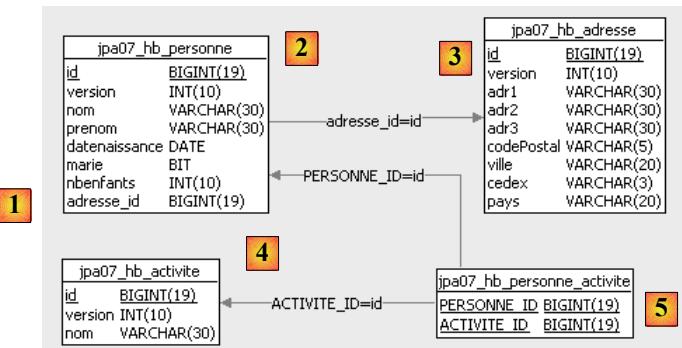
Que conclure ?

Persistance Java 5 par la pratique

- de nouveau, on a un problème de portabilité entre Hibernate et Toplink : Hibernate avait réussi ce test
- Toplink supporte mal que lorsque deux relations sont en fait inverses l'une de l'autre, l'une d'elles ne soit pas déclarée principale et l'autre inverse. On peut l'accepter car ce cas représente en fait une erreur de configuration. Dans notre exemple, la table [article] n'a pas de relation avec la table de jointure [categorie\_article]. Il semble alors naturel que lors d'une opération sur la table [article] Toplink ne cherche pas à travailler avec la table [categorie\_article].

## 2.5 Exemple 4 : relation plusieurs-à-plusieurs avec une table de jointure explicite

### 2.5.1 Le schéma de la base de données



- en [1], la base de données MySQL5

Nous connaissons déjà les tables [personne] [2] et [adresse] [3]. Elles ont été étudiées au paragraphe 2.3.1, page 76. Nous prenons la version où l'adresse de la personne fait l'objet d'une table propre [adresse] [3]. Dans la table [personne], la relation qui lie une personne à son adresse est matérialisée par une contrainte de clé étrangère.

Une personne pratique des activités. Celles-ci sont présentes dans la table [activite] [4]. Une personne peut pratiquer plusieurs activités et une activité peut être pratiquée par plusieurs personnes. Une relation **plusieurs-à-plusieurs** lie donc les tables [personne] et [activite]. Celle-ci est matérialisée par la table de jointure [personne\_activite] [5].

### 2.5.2 Les objets @Entity représentant la base de données

Les tables précédentes vont être représentées par les @Entity suivantes :

- l'@Entity Personne représentera la table [personne]
- l'@Entity Adresse représentera la table [adresse]
- l'@Entity Activite représentera la table [activite]
- l'@Entity PersonneActivite représentera la table [personne\_activite]

Les relations entre ces entités sont les suivantes :

- une relation **un-à-un** relie l'entité Personne à l'entité Adresse : une personne  $p$  a une adresse  $a$ . L'entité Personne qui détient la clé étrangère aura la relation principale, l'entité Adresse la relation inverse.
- une relation **plusieurs-à-plusieurs** relie les entités Personne et Activite : une personne a plusieurs activités et une activité est pratiquée par plusieurs personnes. Cette relation pourrait être réalisée directement par une annotation **@ManyToMany** dans chacune des deux entités, l'une étant déclarée inverse de l'autre. Cette solution sera explorée ultérieurement. Ici, nous réalisons la relation **plusieurs-à-plusieurs** au moyen de deux relations un-à-plusieurs :
  - une relation **un-à-plusieurs** qui relie l'entité Personne à l'entité PersonneActivite : une ligne (One) de la table [personne] est référencée par plusieurs (Many) lignes de la table [personne\_activite]. La table [personne\_activite] détenant la clé étrangère détient la relation **@ManyToOne** principale et l'entité Personne la relation **@OneToMany** inverse.
  - une relation **un-à-plusieurs** qui relie l'entité Activite à l'entité PersonneActivite : une ligne (One) de la table [activite] est référencée par plusieurs (Many) lignes de la table [personne\_activite]. La table [personne\_activite] détenant la clé étrangère détient la relation **@ManyToOne** principale et l'entité Activite la relation **@OneToMany** inverse.

L'@Entity Personne est la suivante :

```

1. @Entity
2. @Table(name = "jpa07_hb_personne")
3. public class Personne implements Serializable {
4.
5.     @Id
6.     @Column(nullable = false)
7.     @GeneratedValue(strategy = GenerationType.AUTO)
8.     private Long id;

```

```

9.
10. @Column(nullable = false)
11. @Version
12. private int version;
13.
14. @Column(length = 30, nullable = false, unique = true)
15. private String nom;
16.
17. @Column(length = 30, nullable = false)
18. private String prenom;
19.
20. @Column(nullable = false)
21. @Temporal(TemporalType.DATE)
22. private Date datenaissance;
23.
24. @Column(nullable = false)
25. private boolean marie;
26.
27. @Column(nullable = false)
28. private int nbenfants;
29.
30. // relation principale Personne (one) -> Adresse (one)
31. // implémentée par la clé étrangère Personne(adresse_id) -> Adresse
32. // cascade insertion Personne -> insertion Adresse
33. // cascade maj Personne -> maj Adresse
34. // cascade suppression Personne -> suppression Adresse
35. // une Personne doit avoir 1 Adresse (nullable=false)
36. // 1 Adresse n'appartient qu'à 1 personne (unique=true)
37. @OneToOne(cascade = CascadeType.ALL)
38. @JoinColumn(name = "adresse_id", unique = true, nullable = false)
39. private Adresse adresse;
40.
41. // relation Personne (one) -> PersonneActivite (many)
42. // inverse de la relation existante PersonneActivite (many) -> Personne (one)
43. // cascade suppression Personne -> suppression PersonneActivite
44. @OneToMany(mappedBy = "personne", cascade = { CascadeType.REMOVE })
45. private Set<PersonneActivite> activites = new HashSet<PersonneActivite>();
46.
47. // constructeurs

```

Cette `@Entity` est connue. Nous ne commentons que les relations qu'elle a avec les autres entités :

- lignes 30-39 : une relation **un-à-un `@OneToOne`** avec l'`@Entity Adresse`, matérialisée par une clé étrangère [adresse\_id] (ligne 38) qu'aura la table [personne] sur la table [adresse].
- lignes 41-45 : une relation **un-à-plusieurs `@OneToMany`** avec l'`@Entity PersonneActivite`. Une personne (One) est référencée par plusieurs (Many) lignes de la table de jointure [personne\_activite] représentée par l'`@Entity PersonneActivite`. Ces objets `PersonneActivite` seront placés dans un type `Set<PersonneActivite>` où `PersonneActivite` est un type que nous allons définir prochainement.
- ligne 44 : la relation **un-à-plusieurs** définie ici, est la relation inverse d'une relation principale définie sur le champ `personne` de l'`@Entity PersonneActivite` (mot clé `mappedBy`). On a une cascade `Personne -> Activite` sur les suppressions : la suppression d'une personne *p* entraînera la suppression des éléments persistants de type `PersonneActivite` trouvés dans l'ensemble *p.activites*.

L'`@Entity Adresse` est la suivante :

```

1. @Entity
2. @Table(name = "jpa07_hb_adresse")
3. public class Adresse implements Serializable {
4.
5.     // champs
6.     @Id
7.     @Column(nullable = false)
8.     @GeneratedValue(strategy = GenerationType.AUTO)
9.     private Long id;
10.    @Column(nullable = false)
11.    @Version
12.    private int version;
13.
14.    @Column(length = 30, nullable = false)
15.    private String adr1;
16.    @Column(length = 30)
17.    private String adr2;
18.    @Column(length = 30)
19.    private String adr3;
20.    @Column(length = 5, nullable = false)
21.    private String codePostal;
22.    @Column(length = 20, nullable = false)
23.    private String ville;

```

```

24. @Column(length = 3)
25. private String cedex;
26. @Column(length = 20, nullable = false)
27. private String pays;
28. @OneToOne(mappedBy = "adresse")
29. private Personne personne;
30.

```

- lignes 28-29 : la relation **@OneToOne** inverse de la relation **@OneToOne adresse** de l'@Entity *Personne* (lignes 37-38 de *Personne*).

L'@Entity *Activite* est la suivante

```

1. @Entity
2. @Table(name = "jpa07_hb_activite")
3. public class Activite implements Serializable {
4.
5.     // champs
6.     @Id
7.     @Column(nullable = false)
8.     @GeneratedValue(strategy = GenerationType.AUTO)
9.     private Long id;
10.
11.    @Column(nullable = false)
12.    @Version
13.    private int version;
14.
15.    @Column(length = 30, nullable = false, unique = true)
16.    private String nom;
17.
18.    // relation Activite (one) -> PersonneActivite (many)
19.    // inverse de la relation existante PersonneActivite (many) -> Activite (one)
20.    // cascade suppression Activite -> suppression PersonneActivite
21.    @OneToMany(mappedBy = "activite", cascade = { CascadeType.REMOVE })
22.    private Set<PersonneActivite> personnes = new HashSet<PersonneActivite>();
23.

```

- lignes 6-9 : la clé primaire de l'activité
- lignes 11-13 : le n° de version de l'activité
- lignes 15-16 : le nom de l'activité
- lignes 18-22 : la relation **un-à-plusieurs** qui lie l'@Entity *Activite* à l'@Entity *PersonneActivite* : une activité (One) est référencée par plusieurs (Many) lignes de la table de jointure [personne\_activite] représentée par l'@Entity *PersonneActivite*. Ces objets *PersonneActivite* seront placés dans un type *Set<PersonneActivite>*.
- ligne 22 : la relation **un-à-plusieurs** définie ici, est la relation inverse d'une relation principale définie sur le champ *activite* dans l'@Entity *PersonneActivite* (mot clé **mappedBy**). On a une cascade *Activite -> PersonneActivite* sur les suppressions : la suppression de la table [activite] d'une activité *a* entraînera la suppression de la table de jointure [personne\_activite] des éléments persistants de type *PersonneActivite* trouvés dans l'ensemble *a.personnes*.

L'@Entity *PersonneActivite* est la suivante :

```

1. @Entity
2. // table de jointure
3. @Table(name = "jpa07_hb_personne_activite")
4. public class PersonneActivite {
5.
6.     @Embeddable
7.     public static class Id implements Serializable {
8.         // composantes de la clé composite
9.         // pointe sur une Personne
10.        @Column(name = "PERSONNE_ID")
11.        private Long personneId;
12.
13.        // pointe sur une Activite
14.        @Column(name = "ACTIVITE_ID")
15.        private Long activiteId;
16.
17.        // constructeurs
18.        ...
19.        // getters et setters
20.        ...
21.        // toString
22.        public String toString() {
23.            return String.format("[%d,%d]", getPersonneId(), getActiviteId());
24.        }
25.    }
26.
27.    // champs de la classe Personne_Activite
28.    // clé composite

```

```

30. @EmbeddedId
31. private Id id = new Id();
32.
33. // relation principale PersonneActivite (many) -> Personne (one)
34. // implémentée par la clé étrangère : personneId (PersonneActivite (many) -> Personne (one))
35. // personneId est en même temps élément de la clé primaire composite
36. // JPA ne doit pas gérer cette clé étrangère (insertable = false, updatable = false) car c'est
   fait par l'application elle-même dans son constructeur
37. @ManyToOne
38. @JoinColumn(name = "PERSONNE_ID", insertable = false, updatable = false)
39. private Personne personne;
40.
41. // relation principale PersonneActivite -> Activite
42. // implémentée par la clé étrangère : activiteId (PersonneActivite (many) -> Activite (one))
43. // activiteId est en même temps élément de la clé primaire composite
44. // JPA ne doit pas gérer cette clé étrangère (insertable = false, updatable = false) car c'est
   fait par l'application elle-même dans son constructeur
45. @ManyToOne()
46. @JoinColumn(name = "ACTIVITE_ID", insertable = false, updatable = false)
47. private Activite activite;
48.
49. // constructeurs
50. public PersonneActivite() {
51.
52. }
53.
54. public PersonneActivite(Personne p, Activite a) {
55.     // les clés étrangères sont fixées par l'application
56.     getId().setPersonneId(p.getId());
57.     getId().setActiviteId(a.getId());
58.     // associations bidirectionnelles
59.     this.setPersonne(p);
60.     this.setActivite(a);
61.     p.getActivites().add(this);
62.     a.getPersonnes().add(this);
63. }
64.
65. // getters et setters
66. ...
67. // toString
68. public String toString() {
69.     return String.format("[%s,%s,%s]", getId(), getPersonne().getNom(), getActivite().getNom());
70. }
71. }

```

Cette classe est plus complexe que les précédentes.

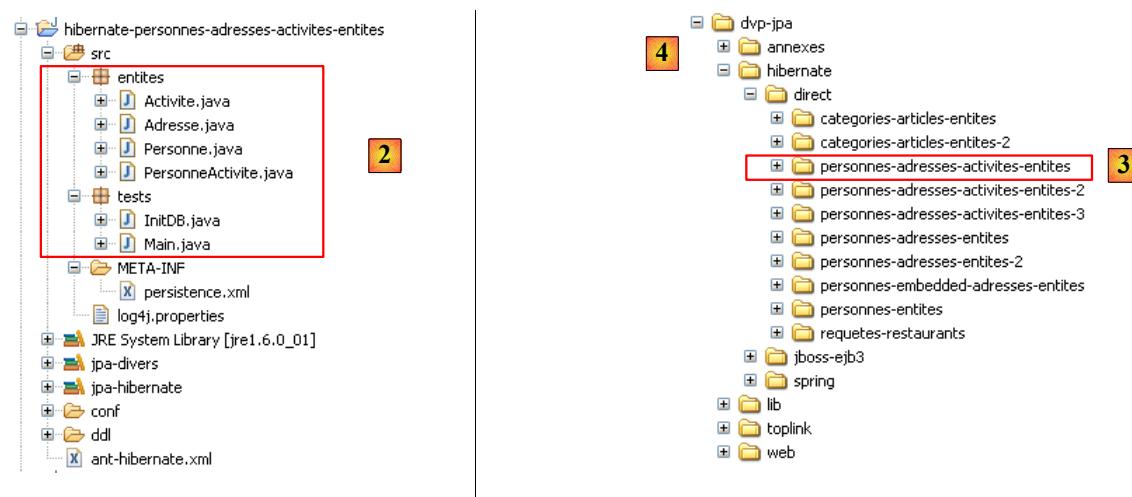
- la table [personne\_activite] a des lignes de la forme [p,a] où *p* est la clé primaire d'une personne et *a* la clé primaire d'une activité. Toute table doit avoir une clé primaire et [personne\_activite] n'échappe pas à la règle. Jusqu'à maintenant, on avait défini des clés primaires générées dynamiquement par le SGBD. On pourrait le faire également ici. On va utiliser une autre technique, celle où l'application définit elle-même les valeurs de la clé primaire d'une table. Ici une ligne [p1,a1] désigne le fait qu'une personne *p1* pratique l'activité *a1*. On ne peut retrouver une deuxième fois cette même ligne dans la table. Ainsi le couple (p,a) est un bon candidat pour être clé primaire. On appelle cela une **clé primaire composite**.
- lignes 30-31 : la clé primaire composite. L'annotation **@EmbeddedId** (habituellement c'était **@Id**) est analogue à la notation **@Embedded** appliquée au champ *Adresse* d'une personne. Dans ce dernier cas, cela signifiait que le champ *Adresse* faisait l'objet d'une classe externe mais devait être inséré dans la même table que la personne. Ici la signification est la même si ce n'est que pour indiquer qu'on a affaire à la clé primaire, la notation devient **@EmbeddedId**.
- ligne 31 : un objet vide représentant la clé primaire *id* est construit dès la construction de l'objet [PersonneActivite]. La classe représentant la clé primaire est définie aux lignes 7-26, comme une classe publique statique interne à la classe [PersonneActivite]. Le fait qu'elle soit publique et statique est imposé par Hibernate. Si on remplace *public static* par *private*, une exception survient et on voit dans le message d'erreur associé qu'Hibernate a essayé d'exécuter l'instruction *new PersonneActivite\$Id*. Il faut donc que la classe *Id* soit à la fois statique et publique.
- ligne 6 : la classe **Id** de la clé primaire est déclarée **@Embeddable**. On se rappelle que la clé primaire *id* de la ligne 31 a été déclarée **@EmbeddedId**. La classe correspondante doit alors avoir l'annotation **@Embeddable**.
- nous avons dit que la clé primaire de la table [personne\_activite] était composée du couple (p,a) où *p* est la clé primaire d'une personne et *a* la clé primaire d'une activité. On trouve les deux éléments (p,a) de la clé composite, ligne 11 (*personneId*) et ligne 15 (*activiteId*). Les colonnes associées à ces deux champs sont nommés : PERSONNE\_ID pour la personne, ACTIVITE\_ID pour l'activité.
- ligne 31 : la clé primaire a été définie avec ses deux colonnes (PERSONNE\_ID, ACTIVITE\_ID). Il n'y a pas d'autres colonnes dans la table [personne\_activite]. Il ne reste plus qu'à définir les relations qui existent entre l'**@Entity PersonneActivite** que nous décrivons actuellement et les autres **@Entity** du schéma relationnel. Ces relations traduisent les contraintes de clés étrangères qu'a la table [personne\_activite] avec les autres tables.
- lignes 33-39 : définissent la clé étrangère qu'a la table [personne\_activite] sur la table [personne]
- ligne 37 : la relation est de type **@ManyToOne** : une ligne (One) de la table [personne] est référencée par plusieurs (Many) lignes de la table [personne\_activite].

- ligne 38 : on nomme la colonne clé étrangère. On reprend le même nom que celui donné pour la composante "personne" de la clé étrangère (ligne 10). Les attributs **insertable=false**, **updatable=false** sont là pour empêcher Hibernate de gérer la clé étrangère. Celle-ci est en effet la composante d'une clé primaire calculée par l'application et Hibernate ne doit pas intervenir.
- lignes 41-47 : définissent la clé étrangère qu'a la table [personne\_activite] sur la table [activite]. Les explications sont les mêmes que celles données précédemment.
- lignes 54-63 : constructeur d'un objet *PersonneActivite* à partir d'une personne *p* et d'une activité *a*. On se rappelle qu'à la construction d'un objet *PersonneActivite*, la clé primaire *id* de la ligne 31 pointait sur un objet *Id* vide. Les lignes 56-57 donnent une valeur à chacun des champs (personneId, activiteId) de l'objet *Id*. Ces valeurs sont respectivement les clés primaires de la personne *p* et de l'activité *a* passées en paramètre du constructeur. La clé primaire *id* (ligne 31) a donc maintenant une valeur.
- ligne 59 : le champ *personne* de la ligne 39 reçoit la valeur *p*
- ligne 60 : le champ *activite* de la ligne 47 reçoit la valeur *a*
- un objet [PersonneActivite] est désormais créé et initialisé. On met à jour les relations inverses qu'ont les @Entity *Personne* (ligne 61) et *Activite* (ligne 62) avec l'@Entity *PersonneActivite* qui vient d'être créée.

Nous avons terminé la description des entités de la base de données. Nous sommes dans une situation complexe mais malheureusement fréquente. Nous verrons qu'il existe une autre configuration possible de la couche JPA qui cache une partie de cette complexité : la table de jointure devient implicite, construite et gérée par la couche JPA. Nous avons choisi ici la solution la plus complexe mais qui permet au schéma relationnel d'évoluer. Elle permet ainsi d'ajouter des colonnes à la table de jointure ce que ne permet pas la configuration où la table de jointure n'est pas une @Entity explicite. [ref1] conseille la solution que nous sommes en train d'étudier. C'est dans [ref1] qu'ont été trouvées les informations qui ont permis l'élaboration de cette solution.

### 2.5.3 Le projet Eclipse / Hibernate

L'implémentation JPA utilisée ici est celle d'Hibernate. Le projet Eclipse des tests est le suivant :



En [1], le projet Eclipse, en [2] les codes Java. Le projet est présent en [3] dans le dossier des exemples [4]. On l'importera.

### 2.5.4 Génération de la DDL de la base de données

En suivant les instructions du paragraphe 2.1.7, page 20, la DDL obtenue pour le SGBD MySQL5 est la suivante :

```

1. alter table jpa07_hb_personne
2.      drop
3.      foreign key FKB5C817D45FE379D0;
4.
5. alter table jpa07_hb_personne_activite
6.      drop
7.      foreign key FKD3E49B06CD852024;
8.
9. alter table jpa07_hb_personne_activite
10.     drop
11.     foreign key FKD3E49B0668C7A284;
12.
13. drop table if exists jpa07_hb_activite;
14.
15. drop table if exists jpa07_hb_adresse;
16.
17. drop table if exists jpa07_hb_personne;

```

```

18.
19.    drop table if exists jpa07_hb_personne_activite;
20.
21.    create table jpa07_hb_activite (
22.        id bigint not null auto_increment,
23.        version integer not null,
24.        nom varchar(30) not null unique,
25.        primary key (id)
26.    ) ENGINE=InnoDB;
27.
28.    create table jpa07_hb_adresse (
29.        id bigint not null auto_increment,
30.        version integer not null,
31.        adr1 varchar(30) not null,
32.        adr2 varchar(30),
33.        adr3 varchar(30),
34.        codePostal varchar(5) not null,
35.        ville varchar(20) not null,
36.        cedex varchar(3),
37.        pays varchar(20) not null,
38.        primary key (id)
39.    ) ENGINE=InnoDB;
40.
41.    create table jpa07_hb_personne (
42.        id bigint not null auto_increment,
43.        version integer not null,
44.        nom varchar(30) not null unique,
45.        prenom varchar(30) not null,
46.        datenaissance date not null,
47.        marie bit not null,
48.        nbenfants integer not null,
49.        adresse_id bigint not null unique,
50.        primary key (id)
51.    ) ENGINE=InnoDB;
52.
53.    create table jpa07_hb_personne_activite (
54.        PERSONNE_ID bigint not null,
55.        ACTIVITE_ID bigint not null,
56.        primary key (PERSONNE_ID, ACTIVITE_ID)
57.    ) ENGINE=InnoDB;
58.
59.    alter table jpa07_hb_personne
60.        add index FKB5C817D45FE379D0 (adresse_id),
61.        add constraint FKB5C817D45FE379D0
62.            foreign key (adresse_id)
63.            references jpa07_hb_adresse (id);
64.
65.    alter table jpa07_hb_personne_activite
66.        add index FKD3E49B06CD852024 (ACTIVITE_ID),
67.        add constraint FKD3E49B06CD852024
68.            foreign key (ACTIVITE_ID)
69.            references jpa07_hb_activite (id);
70.
71.    alter table jpa07_hb_personne_activite
72.        add index FKD3E49B0668C7A284 (PERSONNE_ID),
73.        add constraint FKD3E49B0668C7A284
74.            foreign key (PERSONNE_ID)
75.            references jpa07_hb_personne (id);

```

- lignes 21-26 : la table [activite]
- lignes 28-39 : la table [adresse]
- lignes 41-51 : la table [personne]
- lignes 53-57 : la table de jointure [personne\_activite]. On notera la clé composite (ligne 56)
- lignes 59-63 : la clé étrangère de la table [personne] vers la table [adresse]
- lignes 65-69 : la clé étrangère de la table [personne\_activite] vers la table [activite]
- lignes 71-75 : la clé étrangère de la table [personne\_activite] vers la table [personne]

## 2.5.5 InitDB

Le code de [InitDB] est le suivant :

```

1. package tests;
2.
3. ...
4. public class InitDB {
5.
6.     // constantes
7.     private final static String TABLE_PERSONNE_ACTIVITE = "jpa07_hb_personne_activite";
8.
9.     private final static String TABLE_PERSONNE = "jpa07_hb_personne";
10.
11.    private final static String TABLE_ACTIVITE = "jpa07_hb_activite";

```

```

12.
13. private final static String TABLE_ADRESSE = "jpa07_tb_adresse";
14.
15. public static void main(String[] args) throws ParseException {
16.     // Contexte de persistance
17.     EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
18.     EntityManager em = null;
19.     // on récupère un EntityManager à partir de l'EntityManagerFactory
20.     // précédent
21.     em = emf.createEntityManager();
22.     // début transaction
23.     EntityTransaction tx = em.getTransaction();
24.     tx.begin();
25.     // requête
26.     Query sql1;
27.     // supprimer les éléments de la table PERSONNE_ACTIVITE
28.     sql1 = em.createNativeQuery("delete from " + TABLE_PERSONNE_ACTIVITE);
29.     sql1.executeUpdate();
30.     // supprimer les éléments de la table PERSONNE
31.     sql1 = em.createNativeQuery("delete from " + TABLE_PERSONNE);
32.     sql1.executeUpdate();
33.     // supprimer les éléments de la table ACTIVITE
34.     sql1 = em.createNativeQuery("delete from " + TABLE_ACTIVITE);
35.     sql1.executeUpdate();
36.     // supprimer les éléments de la table ADRESSE
37.     sql1 = em.createNativeQuery("delete from " + TABLE_ADRESSE);
38.     sql1.executeUpdate();
39.     // création activites
40.     Activite act1 = new Activite();
41.     act1.setNom("act1");
42.     Activite act2 = new Activite();
43.     act2.setNom("act2");
44.     Activite act3 = new Activite();
45.     act3.setNom("act3");
46.     // persistance activites
47.     em.persist(act1);
48.     em.persist(act2);
49.     em.persist(act3);
50.     // création personnes
51.     Personne p1 = new Personne("p1", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
52.         true, 2);
52.     Personne p2 = new Personne("p2", "Sylvie", new
53.         SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
53.     Personne p3 = new Personne("p3", "Sylvie", new
54.         SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
54.     // création adresses
55.     Adresse adr1 = new Adresse("adr1", null, null, "49000", "Angers", null, "France");
56.     Adresse adr2 = new Adresse("adr2", "Les Mimosas", "15 av Foch", "49002", "Angers", "03",
57.         "France");
57.     Adresse adr3 = new Adresse("adr3", "x", "x", "x", "x", "x", "x");
58.     Adresse adr4 = new Adresse("adr4", "y", "y", "y", "y", "y", "y");
59.     // associations personne <--> adresse
60.     p1.setAdresse(adr1);
61.     adr1.setPersonne(p1);
62.     p2.setAdresse(adr2);
63.     adr2.setPersonne(p2);
64.     p3.setAdresse(adr3);
65.     adr3.setPersonne(p3);
66.     // persistance des personnes et donc des adresses associées
67.     em.persist(p1);
68.     em.persist(p2);
69.     em.persist(p3);
70.     // persistance de l'adresse a4 non liée à une personne
71.     em.persist(adr4);
72.     // affichage personnes
73.     System.out.println("[personnes]");
74.     for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList())
    {
5.         System.out.println(p);
6.     }
7.     // affichage adresses
8.     System.out.println("[adresses]");
9.     for (Object a : em.createQuery("select a from Adresse a").getResultList()) {
10.         System.out.println(a);
11.     }
12.     System.out.println("[activites]");
13.     for (Object a : em.createQuery("select a from Activite a").getResultList()) {
14.         System.out.println(a);
15.     }
16.     // associations personne <-->activite
17.     PersonneActivite p1act1 = new PersonneActivite(p1, act1);
18.     PersonneActivite p1act2 = new PersonneActivite(p1, act2);
19.     PersonneActivite p2act1 = new PersonneActivite(p2, act1);
20.     PersonneActivite p2act3 = new PersonneActivite(p2, act3);

```

```

91.    // persistance des associations personne <--> activite
92.    em.persist(p1act1);
93.    em.persist(p1act2);
94.    em.persist(p2act1);
95.    em.persist(p2act3);
96.    // affichage personnes
97.    System.out.println("[personnes]");
98.    for (Object p : em.createQuery("select p from Personne p order by p.nom asc").getResultList())
{
99.        System.out.println(p);
100.    }
101.    // affichage adressee
102.    System.out.println("[adresses]");
103.    for (Object a : em.createQuery("select a from Adresse a").getResultList()) {
104.        System.out.println(a);
105.    }
106.    System.out.println("[activites]");
107.    for (Object a : em.createQuery("select a from Activite a").getResultList()) {
108.        System.out.println(a);
109.    }
110.    System.out.println("[personnes/activites]");
111.    for (Object pa : em.createQuery("select pa from PersonneActivite pa").getResultList()) {
112.        System.out.println(pa);
113.    }
114.    // fin transaction
115.    tx.commit();
116.    // fin EntityManager
117.    em.close();
118.    // fin EntityManagerFactory
119.    emf.close();
120.    // log
121.    System.out.println("terminé...");
122.
123.}
124.

```

- lignes 27-38 : les tables [personne\_activite], [personne], [adresse] et [activite] sont vidées. On notera qu'on est obligés de commencer par les tables qui ont les clés étrangères.
- lignes 40-45 : on crée trois activités *act1*, *act2* et *act3*
- lignes 47-49 : elles sont mises dans le contexte de persistance.
- lignes 51-53 : on crée trois personnes *p1*, *p2* et *p3*.
- lignes 55-58 : on crée quatre adresses *adr1* à *adr4*.
- lignes 60-65 : les adresses *adr<sub>i</sub>* sont associées aux personnes *p<sub>i</sub>*. Il y a à chaque fois deux opérations à faire car la relation *Personne* <-> *Adresse* est bidirectionnelle.
- lignes 67-69 : les personnes *p1* à *p3* sont mises dans le contexte de persistance. A cause de la cascade *Personne* -> *Adresse*, ce sera également le cas pour les adresses *adr1* à *adr3*.
- ligne 71 : la 4ième adresse *adr4* non associée à une personne est mise explicitement dans le contexte de persistance.
- lignes 73-85 : le contexte de persistance est requêté pour avoir la listes des entités de type [Personne], [Adresse] et [Activite].. On sait que ces requêtes vont provoquer la synchronisation du contexte avec la base : les entités créées vont être insérées dans la base et obtenir leur clé primaire. Il est important de le comprendre pour la suite.
- lignes 87-90 : on crée 4 associations *Personne* <-> *Activite*. Leur nom indique quelle personne est liée à quelle activité. On se souvient peut-être que la clé primaire d'une entité *PersonneActivite* est une clé composite formée de la clé primaire d'une personne et de celle d'une activité. C'est donc parce que les entités *Personne* et *Activite* ont obtenu leurs clés primaires lors d'une synchronisation précédente que cette opération est possible.
- lignes 92-95 : ces 4 associations sont mises dans le contexte de persistance.
- lignes 87-86 : le contexte de persistance est requêté pour avoir la listes des entités de type [Personne], [Adresse], [Activite] et [PersonneActivite]. On sait que ces requêtes vont provoquer la synchronisation du contexte avec la base : les entités *PersonneActivite* créées vont être insérées dans la base.

L'exécution de [InitDB] avec MySQL5 donne l'affichage console suivant :

```

1. [personnes]
2. P[1,0,p1,Paul,31/01/2000,true,2,1]
3. P[2,0,p2,Sylvie,05/07/2001,false,0,2]
4. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
5. [adresses]
6. A[1,adr1,null,null,49000,Angers,null,France]
7. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
8. A[3,adr3,x,x,x,x,x,x]
9. A[4,adr4,y,y,y,y,Y,Y]
10. [activites]
11. Ac[1,0,act1]
12. Ac[2,0,act2]
13. Ac[3,0,act3]
14. [personnes]
15. P[1,1,p1,Paul,31/01/2000,true,2,1]
16. P[2,1,p2,Sylvie,05/07/2001,false,0,2]

```

```
17. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
18. [adresses]
19. A[1,adr1,null,null,49000,Angers,null,France]
20. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
21. A[3,adr3,x,x,x,x,x,x]
22. A[4,adr4,y,y,Y,Y,Y,Y,Y]
23. [activites]
24. Ac[1,1,act1]
25. Ac[2,1,act2]
26. Ac[3,1,act3]
27. [personnes/activites]
28. [[1,1],p1,act1]
29. [[2,1],p2,act1]
30. [[1,2],p1,act2]
31. [[2,3],p2,act3]
32. terminé...
```

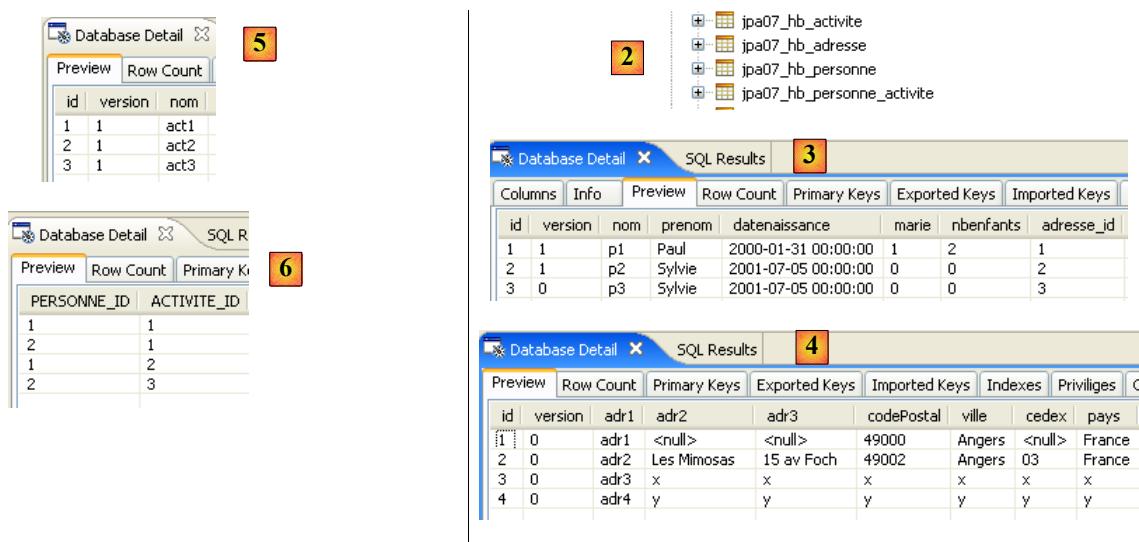
On peut s'étonner de voir que lignes 15-16 les personnes p1 et p2 ont leur n° de version à 1 et qu'il en est de même, lignes 24-26 pour les trois activités. Essayons de comprendre.

Lignes 2-4, les n°s de version de personnes sont à 0 et lignes 11-13, les n°s de version des activités sont à 0. Les affichages précédents ont lieu avant la création des relations *Personne* <-> *Activite*. Lignes 87-90 du code Java, des relations sont créées entre les personnes *p1* et *p2* et les activités *act1*, *act2*, *act3*. Elles sont réalisées au moyen du constructeur de l'@Entity *PersonneActivite* (cf paragraphe 2.5.2, page 111). La lecture du code de ce constructeur montre que lorsqu'une personne *p* est liée à une activité *a* :

- l'activité  $a$  est ajoutée à l'ensemble  $p.activities$
  - la personne  $p$  est ajoutée à l'ensemble  $a.personnes$

Ainsi lorsqu'on écrit `new PersonneActivite(p,a)`, la personne *p* et l'activité *a* subissent une modification en mémoire. Lorsque lignes 97-113 de [InitDB], le contexte de persistance est synchronisé avec la base, JPA / Hibernate découvre que les éléments persistants *p1*, *p2*, *act1*, *act2* et *act3* ont été modifiés. Ces modifications doivent être faites dans la base. Celles-ci sont en fait inscrites dans la table de jointure [personne\_activite] mais JPA / Hibernate incrémenté quand même le n° de version de chacun des éléments persistants modifiés.

Dans la perspective SQL Explorer, les résultats sont les suivants :



- [2] : les tables [jpa07\_hb\_\*]
  - [3] : la table des personnes
  - [4] : la table des adresses.
  - [5] : la table des activités
  - [6] : la table de jointure personne <-> activité

## 2.5.6 Main

La classe [Main] enchaîne des tests que nous passons en revue sauf le test 1 qui reprend le code de [InitDB] pour initialiser la base.

### 2.5.6.1 Test2

Ce test est le suivant :

|1.// suppression Personne p1

```

2. public static void test2() {
3.     // contexte de persistance
4.     EntityManager em = getEntityManager();
5.     // début transaction
6.     EntityTransaction tx = em.getTransaction();
7.     tx.begin();
8.     // suppression dépendances sur p1 : pas nécessaire à hibernate mais
9.     // indispensable à toplink
10.    act1.getPersonnes().remove(p1act1);
11.    act2.getPersonnes().remove(p1act2);
12.    // suppression personne p1
13.    em.remove(p1);
14.    // fin transaction
15.    tx.commit();
16.    // on affiche les nouvelles tables
17.    dumpPersonne();
18.    dumpActivite();
19.    dumpAdresse();
20.    dumpPersonne_Activite();
21. }

```

- ligne 4 : on utilise le contexte de persistance de *test1*, où la personne *p1* est un objet du contexte.
- ligne 13 : suppression de la personne *p1*. A cause de l'attribut :
  - *cascadeType.ALL* sur *Adresse*, l'adresse de la personne *p1* va être supprimée
  - *cascadeType.REMOVE* sur *PersonneActivite*, les activités de la personne *p1* vont être supprimées.
- lignes 10-11 : on supprime les dépendances qu'ont les autres entités sur la personne *p1* qui va être supprimée ligne 13. Les activités *act1* et *act2* sont pratiquées par la personne *p1*. Les liens ont été créés par le constructeur de l'entité *PersonneActivite* dont le code est le suivant :

```

1. public PersonneActivite(Personne p, Activite a) {
2.     // les clés étrangères sont fixées par l'application
3.     getId().setPersonneId(p.getId());
4.     getId().setActiviteId(a.getId());
5.     // associations bidirectionnelles
6.     setPersonne(p);
7.     setActivite(a);
8.     p.getActivites().add(this);
9.     a.getPersonnes().add(this);
10. }

```

ligne 9, l'activité *a* reçoit un élément supplémentaire de type *PersonneActivite* dans son ensemble *personnes*. Cet élément est de type (*p,a*) pour indiquer que la personne *p* pratique l'activité *a*. Dans *test1* de [Main], deux liens (*p1,act1*) et (*p1,act2*) ont été ainsi créés. Les lignes 10 et 11 de *test2* supprime ces dépendances. Il faut noter qu'Hibernate fonctionne sans la suppression de ces dépendances sur la personne *p1* mais pas Toplink.

- lignes 17-20 : on affiche toutes les tables

Les résultats sont les suivants :

```

1. main : -----
2. [personnes]
3. P[1,1,p1,Paul,31/01/2000,true,2,1]
4. P[2,1,p2,Sylvie,05/07/2001,false,0,2]
5. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
6. [activites]
7. Ac[1,1,act1]
8. Ac[2,1,act2]
9. Ac[3,1,act3]
10. [adresses]
11. A[1,adr1,null,null,49000,Angers,null,France]
12. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
13. A[3,adr3,x,x,x,x,x,x]
14. A[4,adr4,y,Y,Y,Y,Y]
15. [personnes/activites]
16. [[1,1],p1,act1]
17. [[2,1],p2,act1]
18. [[1,2],p1,act2]
19. [[2,3],p2,act3]
20. main : -----
21. [personnes]
22. P[2,1,p2,Sylvie,05/07/2001,false,0,2]
23. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
24. [activites]
25. Ac[1,1,act1]
26. Ac[2,1,act2]
27. Ac[3,1,act3]
28. [adresses]
29. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
30. A[3,adr3,x,x,x,x,x]
31. A[4,adr4,y,Y,Y,Y,Y]

```

```

32. [personnes/activites]
33. [[2,1],p2,act1]
34. [[2,3],p2,act3]

```

- la personne  $p1$  présente dans  $test1$  (ligne 3) ne l'est plus à l'issue de  $test2$  (lignes 22-23)
- l'adresse  $adr1$  de la personne  $p1$  présente dans  $test1$  (ligne 11) ne l'est plus à l'issue de  $test2$  (lignes 29-31)
- les activités ( $p1,act1$ ) (ligne 16) et ( $p1,act2$ ) (ligne 18) de la personne  $p1$ , présentes dans  $test1$  ne le sont plus plus à l'issue de  $test2$  (lignes 33-34)

## 2.5.6.2 Test3

Ce test est le suivant :

```

1.// suppression activite act1
2.public static void test3() {
3.    // contexte de persistance
4.    EntityManager em = getEntityManager();
5.    // debut transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // suppression dependances sur act1 : pas necessaire a hibernate mais
9.    // indispensable a toplink
10.   p2.getActivites().remove(p2act1);
11.   // suppression activite act1
12.   em.remove(act1);
13.   // fin transaction
14.   tx.commit();
15.   // on affiche les nouvelles tables
16.   dumpPersonne();
17.   dumpActivite();
18.   dumpAdresse();
19.   dumpPersonne_Activite();
20. }

```

- ligne 4 : on utilise le contexte de persistance de  $test2$
- ligne 12 : suppression de l'activité  $act1$ . A cause de l'attribut :
  - *cascadeType.REMOVE* sur *PersonneActivite*, les lignes ( $p, act1$ ) de la table [personne\_activite] vont être supprimées.
- ligne 10 : avant de mettre  $act1$  en-dehors du contexte de persistance, on supprime les dépendances que peuvent avoir d'autres entités sur cet objet persistant. Après la suppression de la personne  $p1$  au test précédent, seule la personne  $p2$  pratique l'activité  $act1$ .
- lignes 13-16 : on affiche toutes les tables

Les résultats sont les suivants :

```

1. main : -----
2. [personnes]
3. P[2,1,p2,Sylvie,05/07/2001,false,0,2]
4. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
5. [activites]
6. Ac[1,1,act1]
7. Ac[2,1,act2]
8. Ac[3,1,act3]
9. [adresses]
10. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
11. A[3,adr3,x,x,x,x,x,x]
12. A[4,adr4,y,Y,Y,Y,Y]
13. [personnes/activites]
14. [[2,1],p2,act1]
15. [[2,3],p2,act3]
16. main : -----
17. [personnes]
18. P[2,1,p2,Sylvie,05/07/2001,false,0,2]
19. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
20. [activites]
21. Ac[2,1,act2]
22. Ac[3,1,act3]
23. [adresses]
24. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
25. A[3,adr3,x,x,x,x,x,x]
26. A[4,adr4,y,Y,Y,Y,Y]
27. [personnes/activites]
28. [[2,3],p2,act3]

```

- dans  $test2$ , l'activité  $act1$  existe (ligne 6). Dans  $test3$ , elle n'existe plus (lignes 21-22)
- dans  $test2$ , le lien ( $p2,act1$ ) existe (ligne 14). Dans  $test3$ , il n'existe plus (ligne 28)

### 2.5.6.3 Test4

Ce test est le suivant :

```
1.// récupération activités d'une personne
2.public static void test4() {
3.    // contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // début transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // on récupère la personne p2
9.    p2 = em.find(Personne.class, p2.getId());
10.   System.out.format("1 - Activités de la personne p2 (JPQL) :%n");
11.   // on scanne ses activités
12.   for (Object pa : em.createQuery("select a.nom from Activite a join a.personnes pa where
pa.personne.nom='p2'").getResultList()) {
13.       System.out.println(pa);
14.   }
15.   // on passe par la relation inverse de p2
16.   p2 = em.find(Personne.class, p2.getId());
17.   System.out.format("2 - Activités de la personne p2 (relation inverse) :%n");
18.   // on scanne ses activités
19.   for (PersonneActivite pa : p2.getActivites()) {
20.       System.out.println(pa.getActivite().getNom());
21.   }
22.   // fin transaction
23.   tx.commit();
24. }
```

- le test 4 affiche les activités de la personne *p2*.
- ligne 4 : on part d'un contexte neutre et vide
- lignes 12-14 : on affiche les noms des activités pratiquées par la personne *p2* à l'aide d'une requête JPQL.
  - une jointure *Activite* (*a*) / *PersonneActivite* (*pa*) est faite (*join a.personnes*)
  - dans les lignes de cette jointure (*a,pa*), on affiche le nom de l'activité (*a.nom*) pour la personne *p2* (*pa.personne.nom='p2'*).
- lignes 16-21 : on fait la même chose que précédemment, mais avec l'aide de la relation *OneToMany* *p2.activites* de la personne *p2*. La requête JPQL sera générée par JPA. On voit là l'intérêt de la relation inverse *OneToMany* : elle évite une requête JPQL.

Les résultats sont les suivants :

```
1. main : -----
2. 1 - Activités de la personne p2 (JPQL) :
3. act3
4. 2 - Activités de la personne p2 (relation inverse) :
5. act3
```

### 2.5.6.4 Test5

Ce test est le suivant :

```
1.// récupération personnes faisant une activité donnée
2.public static void test5() {
3.    // contexte de persistance
4.    EntityManager em = getNewEntityManager();
5.    // début transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    System.out.format("1 - Personnes pratiquant l'activité act3 (JPQL) :%n");
9.    // on demande les activités de p2
10.   for (Object pa : em.createQuery("select p.nom from Personne p join p.activites pa where
pa.activite.nom='act3'").getResultList()) {
11.       System.out.println(pa);
12.   }
13.   // on passe par la relation inverse de act3
14.   System.out.format("2 - Personnes pratiquant l'activité act3 (relation inverse) :%n");
15.   act3 = em.find(Activite.class, act3.getId());
16.   for (PersonneActivite pa : act3.getPersonnes()) {
17.       System.out.println(pa.getPersonne().getNom());
18.   }
19.   // fin transaction
20.   tx.commit();
21. }
```

- le test 6 affiche les personnes faisant l'activité *act3*. La démarche est analogue à celle du test 6. Nous laissons au lecteur le soin de faire le lien entre les deux codes.

Les résultats sont les suivants :

```

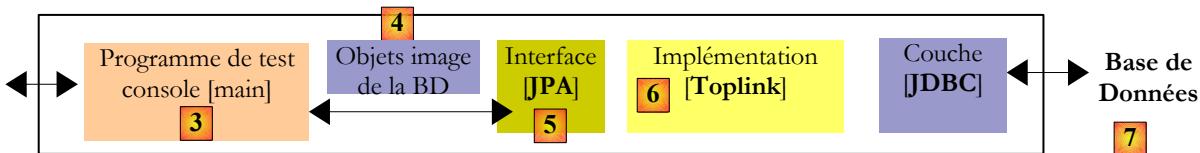
1. main : -----
2. 1 - Personnes pratiquant l'activité act3 (JPQL) :
3. p2
4. 2 - Personnes pratiquant l'activité act3 (relation inverse) :
5. p2

```

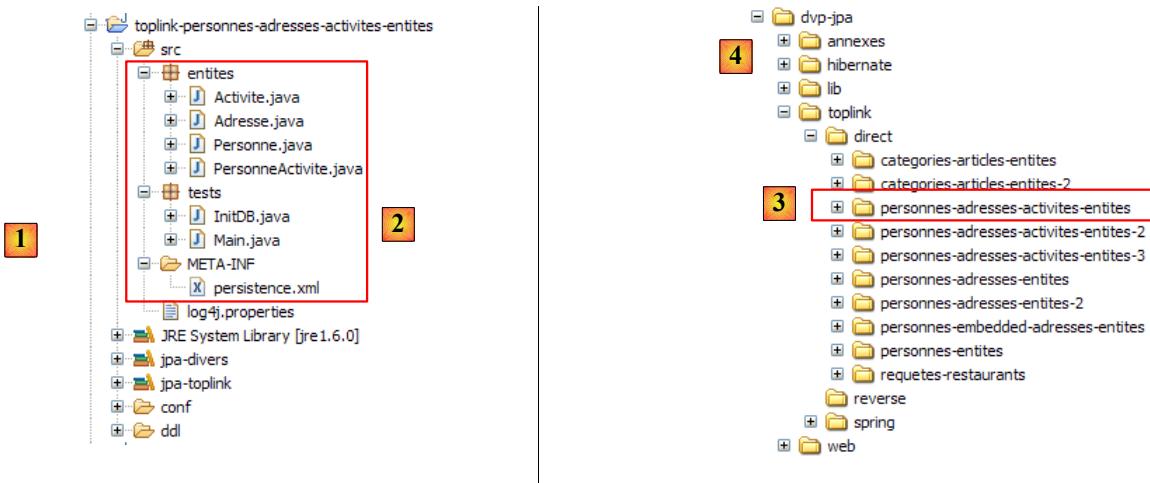
Les tests 4 et 5 avaient pour but de montrer de nouveau qu'une relation inverse n'est jamais indispensable et peut toujours être remplacée par une requête JPQL.

## 2.5.7 Implémentation JPA / Toplink

Nous utilisons maintenant une implémentation JPA / Toplink :



Le projet Eclipse avec Toplink est une copie du projet Eclipse avec Hibernate :



Les codes Java sont identiques à ceux du projet Hibernate précédent à quelques détails près que nous allons évoquer. L'environnement (bibliothèques – persistence.xml – sgbд - dossiers conf, ddl – script ant) est celui étudié au paragraphe 2.1.15.2, page 58. Le projet Eclipse est présent [3] dans le dossier des exemples [4]. On l'importera.

Le fichier <persistence.xml> [2] est modifié en un point, celui des entités déclarées :

```

1. <!-- classes persistantes -->
2. <class>entites.Activite</class>
3. <class>entites.Adresse</class>
4. <class>entites.Personne</class>
5. <class>entites.PersonneActivite</class>

```

- lignes 2-5 : les quatre entités gérées

L'exécution de [InitDB] avec le SGBD MySQL5 donne les résultats suivants :

```

Console X Problems Javadoc Declaration Servers TestNG
<terminated> InitDB [Java Application] C:\Program Files\Java\jre1.6.0_01\bin\javaw.exe (7 juillet 2009)
[personnes]
P[5,1,p1,Paul,31/01/2000,true,2,6]
P[7,1,p2,Sylvie,05/07/2001,false,0,8]
P[9,1,p3,Sylvie,05/07/2001,false,0,10]
[adresses]
A[6,adr1,null,null,49000,Angers,null,France]
A[8,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
A[10,adr3,x,x,x,x,x,x]
A[11,adr4,y,y,y,y,y,y]
[activites]
Ac[2,1,act1]
Ac[3,1,act2]
Ac[4,1,act3]
[personnes]
P[5,1,p1,Paul,31/01/2000,true,2,6]
P[7,1,p2,Sylvie,05/07/2001,false,0,8]
P[9,1,p3,Sylvie,05/07/2001,false,0,10]
[adresses]
A[6,adr1,null,null,49000,Angers,null,France]
A[8,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
A[10,adr3,x,x,x,x,x,x]
A[11,adr4,y,y,y,y,y,y]
[activites]
Ac[2,1,act1]
Ac[3,1,act2]
Ac[4,1,act3]
[personnes/activites]
[[5,2],p1,act1]
[[7,2],p2,act1]
[[5,3],p1,act2]
[[7,4],p2,act3]
terminé...

```

En [1], l'affichage console, en [2], les tables [jpa07\_tl] générées, en [3] les scripts SQL générés. Leur contenu est le suivant :

#### create.sql

```

1. CREATE TABLE jpa07_tl_activite (ID BIGINT NOT NULL, VERSION INTEGER NOT NULL, NOM VARCHAR(30) UNIQUE NOT NULL, PRIMARY KEY (ID))
2. CREATE TABLE jpa07_tl_adresse (ID BIGINT NOT NULL, ADR3 VARCHAR(30), CODEPOSTAL VARCHAR(5) NOT NULL, VERSION INTEGER NOT NULL, VILLE VARCHAR(20) NOT NULL, ADR2 VARCHAR(30), CEDEX VARCHAR(3), ADR1 VARCHAR(30) NOT NULL, PAYS VARCHAR(20) NOT NULL, PRIMARY KEY (ID))
3. CREATE TABLE jpa07_tl_personne_activite (PERSONNE_ID BIGINT NOT NULL, ACTIVITE_ID BIGINT NOT NULL, PRIMARY KEY (PERSONNE_ID, ACTIVITE_ID))
4. CREATE TABLE jpa07_tl_personne (ID BIGINT NOT NULL, DATEDENAISANCE DATE NOT NULL, MARIE TINYINT(1) default 0 NOT NULL, NOM VARCHAR(30) UNIQUE NOT NULL, NBNFANTS INTEGER NOT NULL, VERSION INTEGER NOT NULL, PRENOM VARCHAR(30) NOT NULL, adresse_id BIGINT UNIQUE NOT NULL, PRIMARY KEY (ID))
5. ALTER TABLE jpa07_tl_personne_activite ADD CONSTRAINT FK_jpa07_tl_personne_activite_ACTIVITE_ID FOREIGN KEY (ACTIVITE_ID) REFERENCES jpa07_tl_activite (ID)
6. ALTER TABLE jpa07_tl_personne_activite ADD CONSTRAINT FK_jpa07_tl_personne_activite_PERSONNE_ID FOREIGN KEY (PERSONNE_ID) REFERENCES jpa07_tl_personne (ID)
7. ALTER TABLE jpa07_tl_personne ADD CONSTRAINT FK_jpa07_tl_personne_adresse_id FOREIGN KEY (adresse_id) REFERENCES jpa07_tl_adresse (ID)
8. CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(38), PRIMARY KEY (SEQ_NAME))
9. INSERT INTO SEQUENCE (SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 1)

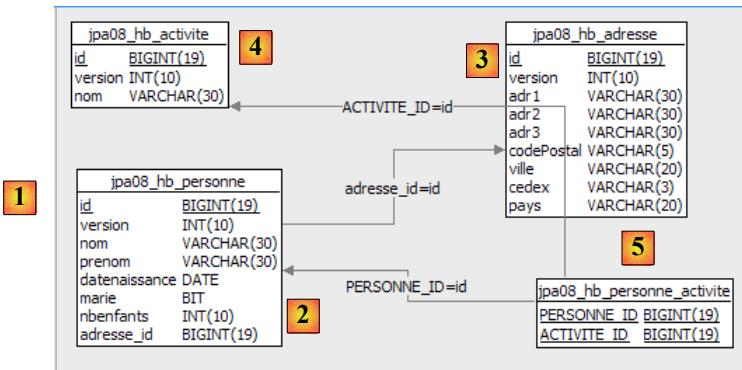
```

L'exécution de [InitDB] et de [Main] se passent sans erreurs.

## 2.6 Exemple 5 : relation plusieurs-à-plusieurs avec une table de jointure implicite

Nous reprenons l'exemple 4 mais nous le traitons maintenant avec une table de jointure implicite générée par la couche JPA elle-même.

### 2.6.1 Le schéma de la base de données



- en [1], la base de données MySQL5 - en [2] : la table [personne] – en [3] : la table [adresse] associée – en [4] : la table [activite] des activités – en [5] : la table de jointure [personne\_activite] qui fait le lien entre des personnes et des activités.

### 2.6.2 Les objets @Entity représentant la base de données

Les tables précédentes vont être représentées par les @Entity suivantes :

- l'@Entity Personne représentera la table [personne]
- l'@Entity Adresse représentera la table [adresse]
- l'@Entity Activite représentera la table [activite]
- la table [personne\_activite] n'est plus représentée par une @Entity

Les relations entre ces entités sont les suivantes :

- une relation **un-à-un** relie l'entité Personne à l'entité Adresse : une personne  $p$  a une adresse  $a$ . L'entité Personne qui détient la clé étrangère aura la relation principale, l'entité Adresse la relation inverse.
- une relation **plusieurs-à-plusieurs** relie les entités Personne et Activite : une personne a plusieurs activités et une activité est pratiquée par plusieurs personnes. Cette relation va être matérialisée par une annotation **@ManyToMany** dans chacune des deux entités, l'une étant déclarée inverse de l'autre.

L'@Entity Personne est la suivante :

```
1. @Entity
2. @Table(name = "jpa08_hb_personne")
3. public class Personne implements Serializable {
4.
5.     @Id
6.     @Column(nullable = false)
7.     @GeneratedValue(strategy = GenerationType.AUTO)
8.     // toplink sqlserver :@GeneratedValue(strategy = GenerationType.IDENTITY)
9.     private Long id;
10.
11.    @Column(nullable = false)
12.    @Version
13.    private int version;
14.
15.    @Column(length = 30, nullable = false, unique = true)
16.    private String nom;
17.
18.    @Column(length = 30, nullable = false)
19.    private String prenom;
20.
21.    @Column(nullable = false)
22.    @Temporal(TemporalType.DATE)
```

```

23. private Date datenaissance;
24.
25. @Column(nullable = false)
26. private boolean marie;
27.
28. @Column(nullable = false)
29. private int nbenfants;
30.
31. // relation principale Personne (one) -> Adresse (one)
32. // implémentée par la clé étrangère Personne(adresse_id) -> Adresse
33. // cascade insertion Personne -> insertion Adresse
34. // cascade maj Personne -> maj Adresse
35. // cascade suppression Personne -> suppression Adresse
36. // une Personne doit avoir 1 Adresse (nullable=false)
37. // 1 Adresse n'appartient qu'à 1 personne (unique=true)
38. @OneToOne(cascade = CascadeType.ALL)
39. @JoinColumn(name = "adresse_id", unique = true, nullable = false)
40. private Adresse adresse;
41.
42. // relation Personne (many) -> Activite (many) via une table de jointure personne_activite
43. // personne_activite(PERSONNE_ID) est clé étrangère sur Personne(id)
44. // personne_activite(ACTIVITE_ID) est clé étrangère sur Activite(id)
45. // cascade=CascadeType.PERSIST : persistance d'1 personne entraîne celle de ses activités
46. @ManyToMany(cascade={CascadeType.PERSIST})
47. @JoinTable(name="jpa08_tb_personne_activite",joinColumns = @JoinColumn(name = "PERSONNE_ID"),
inverseJoinColumns = @JoinColumn(name = "ACTIVITE_ID"))
48. private Set<Activite> activites = new HashSet<Activite>();
49.
50. // constructeurs
51. public Personne() {
52. }
53.

```

Nous ne commentons que la relation **@ManyToMany** des lignes 46-48 qui relie l'**@Entity Personne** à l'**@Entity Activite**:

- ligne 48 : une personne a des activités. Le champ **activites** représentera celles-ci. Dans la version précédente, le type des éléments de l'ensemble *activites* était *PersonneActivite*. Ici, c'est *Activite*. On accède donc directement aux activités d'une personne, alors que dans la version précédente il fallait passer par l'entité intermédiaire *PersonneActivite*.
- ligne 46 : la relation qui lie l'**@Entity Personne** que nous étudions à l'**@Entity Activite** de l'ensemble *activites* de la ligne 48 est de type **plusieurs-à-plusieurs** (**ManyToMany**):
  - une personne (One) a plusieurs activités (Many)
  - une activité (One) est pratiquée par plusieurs personnes (Many)
  - au final les **@Entity Personne** et **Activite** sont reliées par une relation **ManyToMany**. Comme dans la relation **OneToOne**, il y a symétrie des entités dans cette relation. On peut choisir librement l'**@Entity** qui détiendra la relation principale et celle qui aura la relation inverse. Ici, nous décidons que l'**@Entity Personne** aura la relation principale.
  - comme nous l'avons vu dans l'exemple précédent, la relation **@ManyToMany** nécessite une table de jointure. Alors que précédemment, nous avions défini celle-ci à l'aide d'une **@Entity**, la table de jointure ici est définie à l'aide de l'annotation **@JoinTable** de la ligne 47.
    - l'attribut **name** donne un nom à la table.
    - la table de jointure est constituée des clés étrangères sur les tables qu'elle joint. Ici, il y a deux clés étrangères : une sur la table [personne], l'autre sur la table [activite]. Ces colonnes clés étrangères sont définies par les attributs **joinColumns** et **inverseJoinColumns**.
    - l'annotation **@JoinColumn** de l'attribut **joinColumns** définit la clé étrangère sur la table de l'**@Entity** détenant la relation principale **@ManyToMany**, ici la table [personne]. Cette colonne clé étrangère s'appellera PERSONNE\_ID.
    - l'annotation **@JoinColumn** de l'attribut **inverseJoinColumns** définit la clé étrangère sur la table de l'**@Entity** détenant la relation inverse **@ManyToMany**, ici la table [activite]. Cette colonne clé étrangère s'appellera ACTIVITE\_ID.

L'**@Entity Adresse** est la suivante :

```

1. @Entity
2. @Table(name = "jpa07_tb_adresse")
3. public class Adresse implements Serializable {
4.
5. // champs
6. @Id
7. @Column(nullable = false)
8. @GeneratedValue(strategy = GenerationType.AUTO)
9. private Long id;
10. @Column(nullable = false)
11. @Version
12. private int version;

```

```

13.
14. @Column(length = 30, nullable = false)
15. private String adr1;
16. @Column(length = 30)
17. private String adr2;
18. @Column(length = 30)
19. private String adr3;
20. @Column(length = 5, nullable = false)
21. private String codePostal;
22. @Column(length = 20, nullable = false)
23. private String ville;
24. @Column(length = 3)
25. private String cedex;
26. @Column(length = 20, nullable = false)
27. private String pays;
28. @OneToOne(mappedBy = "adresse")
29. private Personne personne;
30.

```

- lignes 28-29 : la relation **@OneToOne** inverse de la relation **@OneToOne adresse** de l'**@Entity Personne** (lignes 37-38 de **Personne**).

L'**@Entity Activite** est la suivante

```

1. @Entity
2. @Table(name = "jpa08_hb_activite")
3. public class Activite implements Serializable {
4.
5.     // champs
6.     @Id()
7.     @Column(nullable = false)
8.     @GeneratedValue(strategy = GenerationType.AUTO)
9.     // toplink sqlserver : @GeneratedValue(strategy = GenerationType.IDENTITY)
10.    private Long id;
11.
12.    @Column(nullable = false)
13.    @Version
14.    private int version;
15.
16.    @Column(length = 30, nullable = false, unique = true)
17.    private String nom;
18.
19.    // relation inverse Activite -> Personne
20.    @ManyToMany(mappedBy = "activites")
21.    private Set<Personne> personnes = new HashSet<Personne>();
22. ...

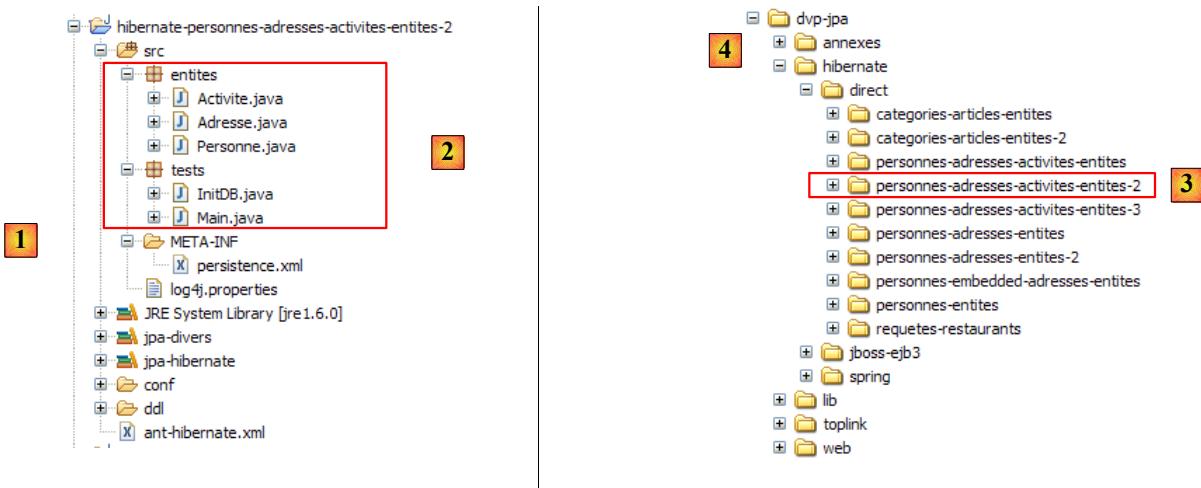
```

- lignes 20-21 : la relation **plusieurs-à-plusieurs** qui lie l'**@Entity Activite** à l'**@Entity Personne**. Cette relation a déjà été définie dans l'**@Entity Personne**. On se contente donc ici de dire que la relation est **inverse** (**mappedBy**) de la relation **@ManyToMany** existant sur le champ **activites** (**mappedBy= "activites "**) de l'**@Entity Personne**.
- rappelons qu'une relation inverse est toujours facultative. Ici, nous l'utilisons pour obtenir les personnes pratiquant l'activité courante. C'est l'ensemble **Set<Personne> personnes** qui permettra d'obtenir celles-ci. Le mode de chargement des dépendances **Personne** de l'**@Entity Activite** n'est pas précisé. Nous ne l'avions pas précisé non plus dans l'exemple précédent. Par défaut, ce mode est **fetch=FetchType.LAZY**.

Nous avons terminé la description des entités de la base de données. Elle a été plus simple que dans le cas où la table de jointure [personne\_activite] fait l'objet d'une table explicite. Cette solution plus simple peut présenter des inconvénients au fil du temps : elle ne permet pas d'ajouter des colonnes à la table de jointure. Cela peut pourtant s'avérer nécessaire pour satisfaire des besoins nouveaux, par exemple ajouter à la table [personne\_activite] une colonne indiquant la date d'inscription de la personne à l'activité.

## 2.6.3 Le projet Eclipse / Hibernate

L'implémentation JPA utilisée ici est celle d'Hibernate. Le projet Eclipse des tests est le suivant :



En [1], le projet Eclipse, en [2] les codes Java. Le projet est présent en [3] dans le dossier des exemples [4]. On l'importera.

## 2.6.4 Génération de la DDL de la base de données

En suivant les instructions du paragraphe 2.1.7, page 20, la DDL obtenue pour le SGBD MySQL5 est la suivante :

```

1. alter table jpa08_hb_personne
2.      drop
3.      foreign key FKA44B1E555FE379D0;
4.
5. alter table jpa08_hb_personne_activite
6.      drop
7.      foreign key FK5A6A55A5CD852024;
8.
9. alter table jpa08_hb_personne_activite
10.     drop
11.     foreign key FK5A6A55A568C7A284;
12.
13. drop table if exists jpa08_hb_activite;
14.
15. drop table if exists jpa08_hb_adresse;
16.
17. drop table if exists jpa08_hb_personne;
18.
19. drop table if exists jpa08_hb_personne_activite;
20.
21. create table jpa08_hb_activite (
22.     id bigint not null auto_increment,
23.     version integer not null,
24.     nom varchar(30) not null unique,
25.     primary key (id)
26. ) ENGINE=InnoDB;
27.
28. create table jpa08_hb_adresse (
29.     id bigint not null auto_increment,
30.     version integer not null,
31.     adr1 varchar(30) not null,
32.     adr2 varchar(30),
33.     adr3 varchar(30),
34.     codePostal varchar(5) not null,
35.     ville varchar(20) not null,
36.     cedex varchar(3),
37.     pays varchar(20) not null,
38.     primary key (id)
39. ) ENGINE=InnoDB;
40.
41. create table jpa08_hb_personne (
42.     id bigint not null auto_increment,
43.     version integer not null,
44.     nom varchar(30) not null unique,
45.     prenom varchar(30) not null,
46.     datenaissance date not null,
47.     marie bit not null,
48.     nbenfants integer not null,
49.     adresse_id bigint not null unique,
50.     primary key (id)
51. ) ENGINE=InnoDB;
52.
53. create table jpa08_hb_personne_activite (
54.     PERSONNE_ID bigint not null,
55.     ACTIVITE_ID bigint not null,
56.     primary key (PERSONNE_ID, ACTIVITE_ID)

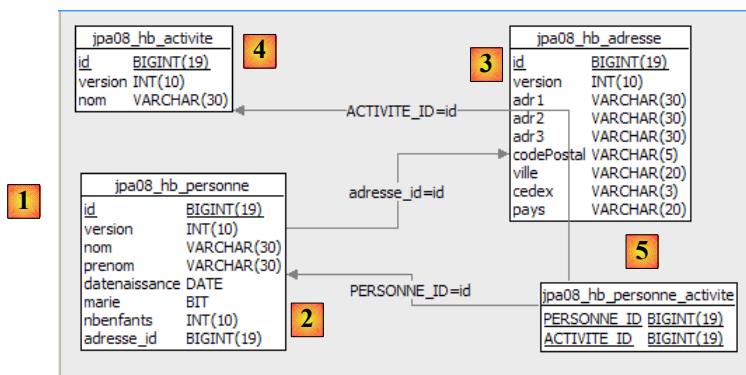
```

```

57.     ) ENGINE=InnoDB;
58.
59. alter table jpa08_hb_personne
60.   add index FKA44B1E555FE379D0 (adresse_id),
61.   add constraint FKA44B1E555FE379D0
62.   foreign key (adresse_id)
63.   references jpa08_hb_adresse (id);
64.
65. alter table jpa08_hb_personne_activite
66.   add index FK5A6A55A5CD852024 (ACTIVITE_ID),
67.   add constraint FK5A6A55A5CD852024
68.   foreign key (ACTIVITE_ID)
69.   references jpa08_hb_activite (id);
70.
71. alter table jpa08_hb_personne_activite
72.   add index FK5A6A55A568C7A284 (PERSONNE_ID),
73.   add constraint FK5A6A55A568C7A284
74.   foreign key (PERSONNE_ID)
75.   references jpa08_hb_personne (id);

```

Cette DDL est analogue à celle obtenue avec la table de jointure explicite et correspond au schéma déjà présenté :



## 2.6.5 InitDB

Nous commenterons peu la classe [InitDB] identique à sa version précédente et donnant les mêmes résultats. Simplement, attardons-nous sur le code suivant qui affiche la jointure *Personne* <-> *Activité* :

```

1. // affichage personnes/activités
2. System.out.println("[personnes/activités]");
3. Iterator iterator = em.createQuery("select p.id,a.id from Personne p join p.activites
a").getResultSet().iterator();
4. while (iterator.hasNext()) {
5.     Object[] row = (Object[]) iterator.next();
6.     System.out.format("%d,%d%n", (Long) row[0], (Long) row[1]);
7. }

```

- ligne 3 : l'ordre JPQL qui fait la jointure. Le résultat du *select* ramène les identifiants des entités *Personne* et *Activité* liées entre-elles par la table de jointure. La liste rendue par le *select* est formée de lignes comportant deux objets de type *Long*. Pour parcourir cette liste, la ligne 3 demande un objet *Iterator* sur la liste.
- lignes 4-7 : à l'aide de l'objet de type *Iterator* précédent, on parcourt la liste.
  - ligne 5 : chaque élément de la liste est un tableau contenant une ligne résultat du *select*
  - ligne 6 : on récupère les éléments de la ligne courante résultat du *select* en faisant les changements types adéquats.

Le résultat de [InitDB] est le suivant :

```

1. [personnes]
2. P[1,0,p1,Paul,31/01/2000,true,2,1]
3. P[2,0,p2,Sylvie,05/07/2001,false,0,2]
4. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
5. [adresses]
6. A[1,adr1,null,null,49000,Angers,null,France]
7. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
8. A[3,adr3,x,x,x,x,x]
9. A[4,adr4,y,y,y,y,y]
10. [activites]
11. Ac[1,1,act1]
12. Ac[2,1,act2]
13. Ac[3,1,act3]

```

```

14. [personnes/activites]
15. [1,1]
16. [1,2]
17. [2,1]
18. [2,3]
19. terminé...

```

## 2.6.6 Main

La classe [Main] enchaîne des tests dont nous passons certains en revue.

### 2.6.6.1 Test3

Ce test est le suivant :

```

1.// suppression activite act1
2.public static void test3() {
3.    // contexte de persistance
4.    EntityManager em = getEntityManager();
5.    // début transaction
6.    EntityTransaction tx = em.getTransaction();
7.    tx.begin();
8.    // suppression activité act1 de p2
9.    p2.getActivites().remove(act1);
10.   // on retire act1 du contexte de persistance
11.   em.remove(act1);
12.   // fin transaction
13.   tx.commit();
14.   // on affiche les nouvelles tables
15.   dumpPersonne();
16.   dumpActivite();
17.   dumpAdresse();
18.   dumpPersonne_Activite();
19. }

```

- ligne 11 : l'activité *act1* est retirée du contexte de persistance
- ligne 9 : l'activité *act1* fait partie des activités de la seule personne restant dans le contexte, la personne *p2*. La ligne 9 retire l'activité *act1* des activités de la personne *p2*. Nous faisons cela pour garder cohérent le contexte de persistance car nous le conservons pour la suite.

Les résultats sont les suivants :

```

1. main : -----
2. [personnes]
3. P[1,0,p1,Paul,31/01/2000,true,2,1]
4. P[2,0,p2,Sylvie,05/07/2001,false,0,2]
5. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
6. [activites]
7. Ac[1,0,act1]
8. Ac[2,0,act2]
9. Ac[3,0,act3]
10. [adresses]
11. A[1,adr1,null,null,49000,Angers,null,France]
12. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
13. A[3,adr3,x,x,x,x,x,x]
14. A[4,adr4,y,y,Y,Y,Y,y]
15. [personnes/activites]
16. [1,1]
17. [1,2]
18. [2,1]
19. [2,3]
20. main : -----
21. [personnes]
22. P[2,0,p2,Sylvie,05/07/2001,false,0,2]
23. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
24. [activites]
25. Ac[1,0,act1]
26. Ac[2,0,act2]
27. Ac[3,0,act3]
28. [adresses]
29. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
30. A[3,adr3,x,x,x,x,x,x]
31. A[4,adr4,y,y,Y,Y,Y,y]
32. [personnes/activites]
33. [2,1]
34. [2,3]
35. main : -----
36. [personnes]
37. P[2,1,p2,Sylvie,05/07/2001,false,0,2]
38. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
39. [activites]

```

```

40. Ac[2,0,act2]
41. Ac[3,0,act3]
42. [adresses]
43. A[2,adr2,Les Mimosas,15 av Foch,49002,Angers,03,France]
44. A[3,adr3,x,x,x,x,x,x]
45. A[4,adr4,y,y,y,y,y,y]
46. [personnes/activites]
47. [2,3]

```

- l'activité *act1* présente ligne 26 dans *test2* a disparu des activités de *test3* (lignes 40-41)
- la personne *p2* avait dans *test2* l'activité *act1* (ligne 33). A l'issue de *test3*, elle ne l'a plus (ligne 47)

## 2.6.6.2 Test6

Ce test est le suivant :

```

1. // modification des activités d'une personne
2. public static void test6() {
3.     // contexte de persistance
4.     EntityManager em = getNewEntityManager();
5.     // début transaction
6.     EntityTransaction tx = em.getTransaction();
7.     tx.begin();
8.     // on récupère la personne p2
9.     p2 = em.find(Personne.class, p2.getId());
10.    // on récupère l'activité act2
11.    act2 = em.find(Activite.class, act2.getId());
12.    // p2 ne pratique plus que l'activité act2
13.    p2.getActivites().clear();
14.    p2.getActivites().add(act2);
15.    // fin transaction
16.    tx.commit();
17.    // on affiche les nouvelles tables
18.    dumpPersonne();
19.    dumpActivite();
20.    dumpPersonne_Activite();
21. }

```

- ligne 4 : on utilise un contexte de persistance neuf et vide
- ligne 9 : la personne *p2* est amenée de la base dans le contexte de persistance
- ligne 11 : l'activité *act2* est amenée de la base dans le contexte de persistance
- ligne 13 : les activités de la personne *p2* (*act3*) sont amenées de la base dans le contexte (fetchType.LAZY). C'est l'appel [getActivites] qui provoque ce chargement. On supprime les activités de *p2*. Il ne s'agit pas d'une suppression réelle d'activités (remove) mais d'une modification de l'état de la personne *p2*. Elle ne pratique plus d'activités.
- ligne 14 : on ajoute à la personne *p2* l'activité *act2*. Au final, l'ensemble des activités nouvelles de la personne *p2* est l'ensemble {*act2*}.
- ligne 16 : fin de la transaction. La synchronisation va passer en revue les objets du contexte (*p2*, *act2*, *act3*) et va découvrir que l'état de *p2* a changé. Les ordres SQL répercutant ce changement sur la base vont être exécutés.
- lignes 18-20 : on affiche toutes les tables

Les résultats sont les suivants :

```

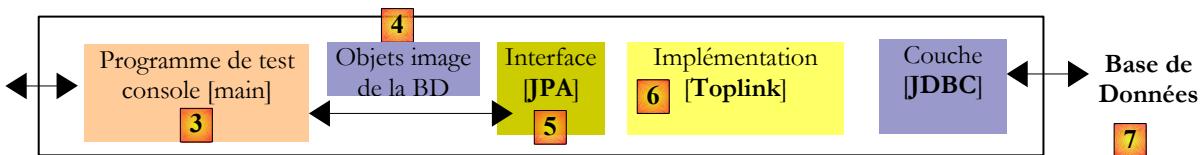
1. main : -----
2. 1 - Activités de la personne p2 (JPQL) :
3. act3
4. 2 - Activités de la personne p2 (relation principale) :
5. act3
6. main : -----
7. 1 - Personnes pratiquant l'activité act3 (JPQL) :
8. p2
9. 2 - Personnes pratiquant l'activité act3 (relation inverse) :
10. p2
11. main : -----
12. [personnes]
13. P[2,2,p2,Sylvie,05/07/2001,false,0,2]
14. P[3,0,p3,Sylvie,05/07/2001,false,0,3]
15. [activites]
16. Ac[2,0,act2]
17. Ac[3,0,act3]
18. [personnes/activites]
19. [2,2]

```

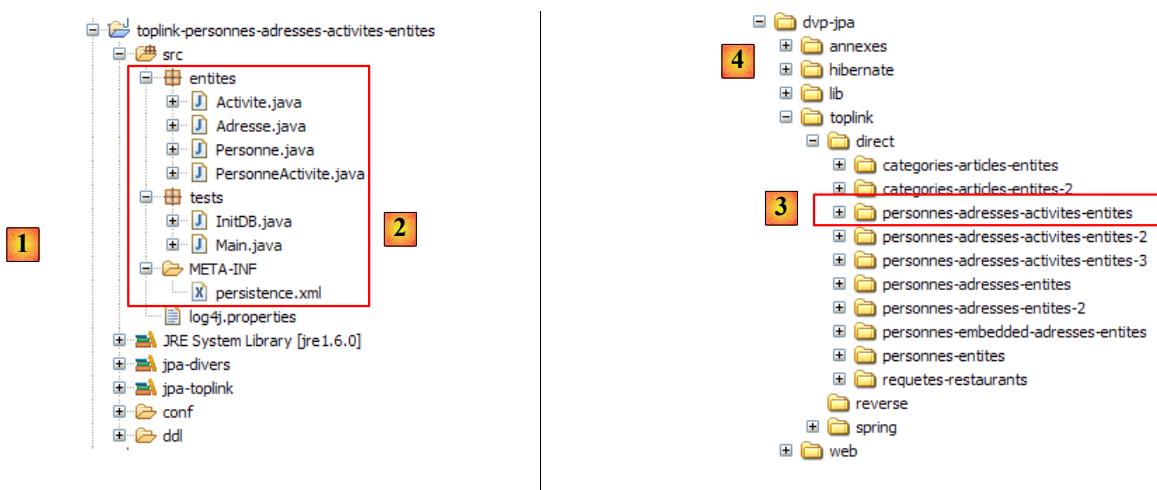
- à l'issue du test 4, la personne *p2* pratiquait l'activité *act3* (ligne 3).
- à l'issue du test 6 (ligne 19), la personne *p2* ne pratique plus l'activité *act3* (ligne 3) et elle pratique l'activité *act2*.

### 2.6.7 Implémentation JPA / Toplink

Nous utilisons maintenant une implémentation JPA / Toplink :



Le projet Eclipse avec Toplink est une copie du projet Eclipse avec Hibernate :

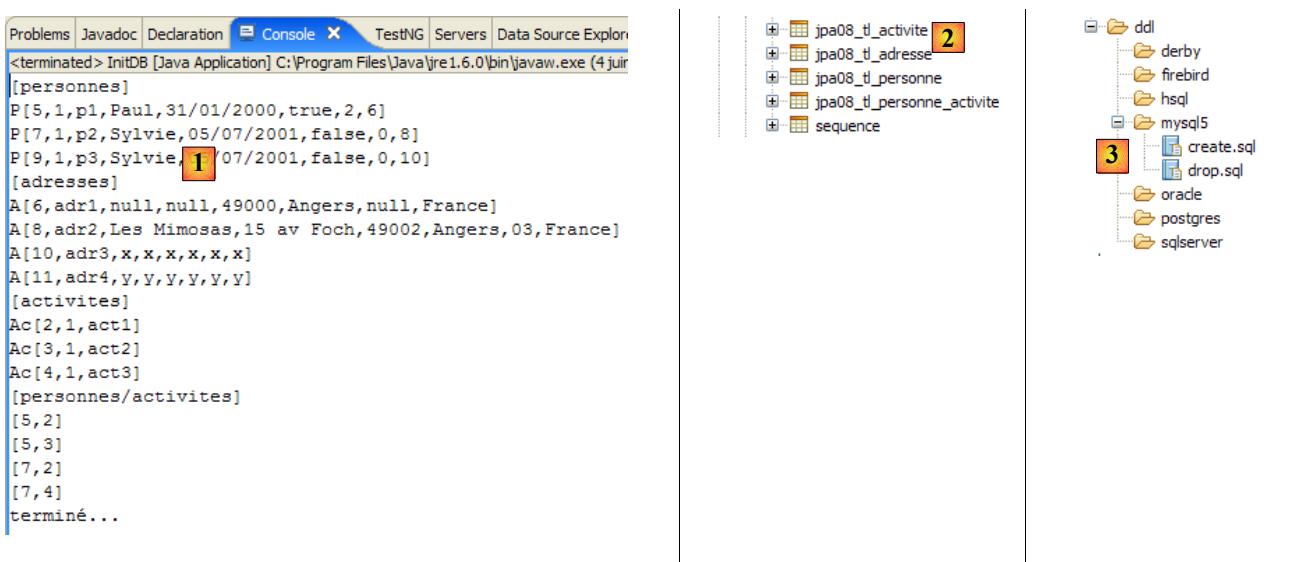


Le fichier <persistence.xml> [2] est modifié en un point, celui des entités déclarées :

```
1.      <!-- provider -->
2.      <provider>oracle.toplink.essentials.PersistenceProvider</provider>
3.      <!-- classes persistantes -->
4.      <class>entites.Activite</class>
5.      <class>entites.Adresse</class>
6.      <class>entites.Personne</class>
7.  ...
```

- lignes 4-6 : les entités gérées

L'exécution de [InitDB] avec le SGBD MySQL5 donne les résultats suivants :



En [1], l'affichage console, en [2], les tables [jpa07\_tl] générées, en [3] les scripts SQL générés. Leur contenu est le suivant : Persistance Java 5 par la pratique

## create.sql

```

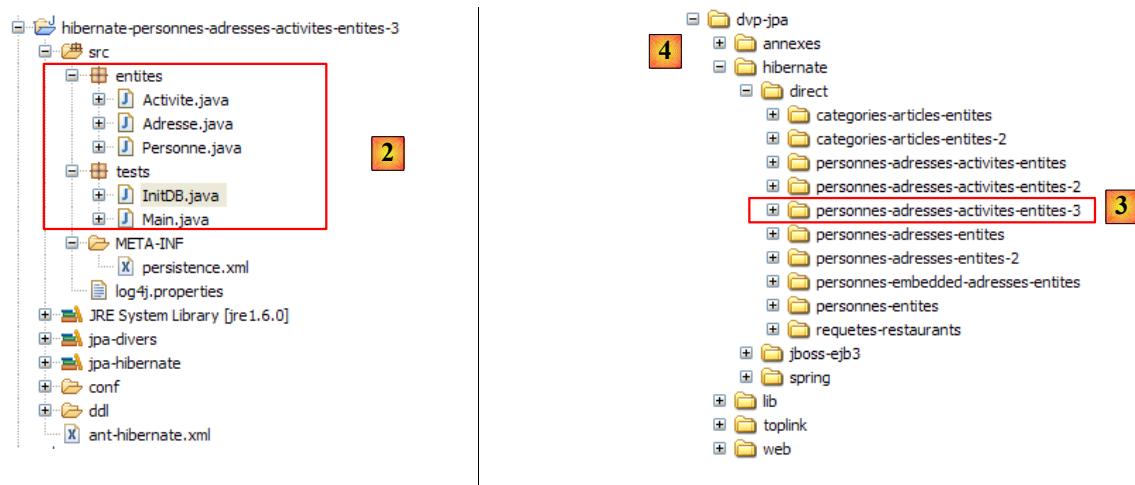
1. CREATE TABLE jpa08_t1_personne_activite (PERSONNE_ID BIGINT NOT NULL, ACTIVITE_ID BIGINT NOT NULL,
   PRIMARY KEY (PERSONNE_ID, ACTIVITE_ID))
2. CREATE TABLE jpa08_t1_activite (ID BIGINT NOT NULL, VERSION INTEGER NOT NULL, NOM VARCHAR(30)
   UNIQUE NOT NULL, PRIMARY KEY (ID))
3. CREATE TABLE jpa08_t1_personne (ID BIGINT NOT NULL, DATENAISANCE_DATE NOT NULL, MARIE TINYINT(1)
   default 0 NOT NULL, NOM VARCHAR(30) UNIQUE NOT NULL, NBENFANTS INTEGER NOT NULL, VERSION INTEGER
   NOT NULL, PRENOM VARCHAR(30) NOT NULL, adresse_id BIGINT UNIQUE NOT NULL, PRIMARY KEY (ID))
4. CREATE TABLE jpa08_t1_adresse (ID BIGINT NOT NULL, ADR3 VARCHAR(30), CODEPOSTAL VARCHAR(5) NOT
   NULL, VERSION INTEGER NOT NULL, VILLE VARCHAR(20) NOT NULL, ADR2 VARCHAR(30), CEDEX VARCHAR(3),
   ADR1 VARCHAR(30) NOT NULL, PAYS VARCHAR(20) NOT NULL, PRIMARY KEY (ID))
5. ALTER TABLE jpa08_t1_personne_activite ADD CONSTRAINT FK_jpa08_t1_personne_activite_ACTIVITE_ID
   FOREIGN KEY (ACTIVITE_ID) REFERENCES jpa08_t1_activite (ID)
6. ALTER TABLE jpa08_t1_personne_activite ADD CONSTRAINT FK_jpa08_t1_personne_activite_PERSONNE_ID
   FOREIGN KEY (PERSONNE_ID) REFERENCES jpa08_t1_personne (ID)
7. ALTER TABLE jpa08_t1_personne ADD CONSTRAINT FK_jpa08_t1_personne_adresse_id FOREIGN KEY
   (adresse_id) REFERENCES jpa08_t1_adresse (ID)
8. CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(38), PRIMARY KEY
   (SEQ_NAME))
9. INSERT INTO SEQUENCE (SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 1)

```

L'exécution de [InitDB] et celle de [Main] se passent sans erreurs.

## 2.6.8 Le projet Eclipse / Hibernate 2

Nous construisons un projet Eclipse issu du précédent par recopie :



En [1], le projet Eclipse, en [2] les codes Java. Le projet est présent en [3] dans le dossier des exemples [4]. On l'importera.

Nous modifions la relation qui lie *Personne* à *Activité* de la façon suivante :

### Personne

```

1.// relation Personne (many) -> Activite (many) via une table de jointure personne_activite
2.// personne_activite(PERSONNE_ID) est clé étrangère sur Personne(id)
3.// personne_activite(ACTIVITE_ID) est clé étrangère sur Activite(id)
4.// plus de cascade sur les activités
5.// @ManyToMany(cascade={CascadeType.PERSIST})
6.// @ManyToMany()
7.@JoinTable(name = "jpa09_sb_personne_activite", joinColumns = @JoinColumn(name = "PERSONNE_ID"),
   inverseJoinColumns = @JoinColumn(name = "ACTIVITE_ID"))
8.private Set<Activite> activites = new HashSet<Activite>();

```

- ligne 6 : la relation principale @ManyToMany n'a plus de cascade de persistance Personne -> Activite (cf ancienne version ligne 5)

### Activité

```

1.// plus de relation inverse avec Personne
2.// @ManyToMany(mappedBy = "activites")
3.// private Set<Personne> personnes = new HashSet<Personne>();

```

- lignes 2-3 : la relation inverse `@ManyToMany Activite -> Personne` est supprimée

Nous cherchons à montrer que les attributs supprimés (cascade et relation inverse) ne sont pas indispensables. Le premier changement amené par cette nouvelle configuration se trouve dans [InitDB] :

```

1. // associations personnes <--> activites
2. p1.getActivites().add(act1);
3. p1.getActivites().add(act2);
4. p2.getActivites().add(act1);
5. p2.getActivites().add(act3);
6. // persistance des activites
7. em.persist(act1);
8. em.persist(act2);
9. em.persist(act3);
10. // persistance des personnes
11. em.persist(p1);
12. em.persist(p2);
13. em.persist(p3);
14. // et de l'adresse a4 non liée à une personne
15. em.persist(addr4);

```

- lignes 7-9 : on est obligés de mettre explicitement les activités `act1` à `act3` dans le contexte de persistance. Lorsque la cascade de persistance `Personne -> Activite` existait, les lignes 11-13 persistaient à la fois les personnes `p1` à `p3` et les activités de ces personnes `act1` à `act3`.

Un second changement est visible dans [Main] :

```

1. // récupération personnes faisant une activité donnée
2. public static void test5() {
3.     // contexte de persistance
4.     EntityManager em = getNewEntityManager();
5.     // début transaction
6.     EntityTransaction tx = em.getTransaction();
7.     tx.begin();
8.     System.out.format("1 - Personnes pratiquant l'activité act3 (JPQL) :%n");
9.     // on demande les activités de p2
10.    for (Object pa : em.createQuery("select p.nom from Personne p join p.activites a where
   a.nom='act3'").getResultList()) {
11.        System.out.println(pa);
12.    }
13.    // fin transaction
14.    tx.commit();
15.}

```

- lignes 9-12 : la requête JPQL obtenant les personnes pratiquant l'activité `act3`
- dans la version précédente, le même résultat avait été également obtenu via la relation inverse `Activite -> Personne` maintenant supprimée :

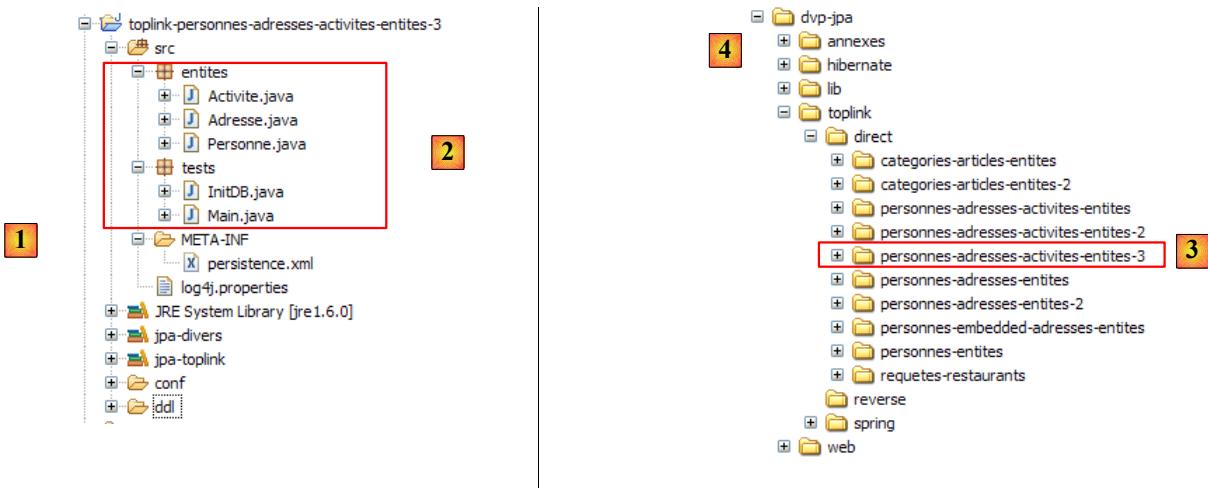
```

1. // on passe par la relation inverse de act3
2. System.out.format("2 - Personnes pratiquant l'activité act3 (relation inverse) :%n");
3. act3 = em.find(Activite.class, act3.getId());
4. for (Personne p : act3.getPersonnes()) {
5.     System.out.println(p.getNom());
6. }

```

## 2.6.9 Le projet Eclipse / Toplink 2

Nous construisons un projet Eclipse issu du précédent projet Eclipse / Toplink par recopie :



En [1], le projet Eclipse, en [2] les codes Java. Le projet est présent en [3] dans le dossier des exemples [4]. On l'importera.

Les codes Java sont identiques à ceux de la version Hibernate.

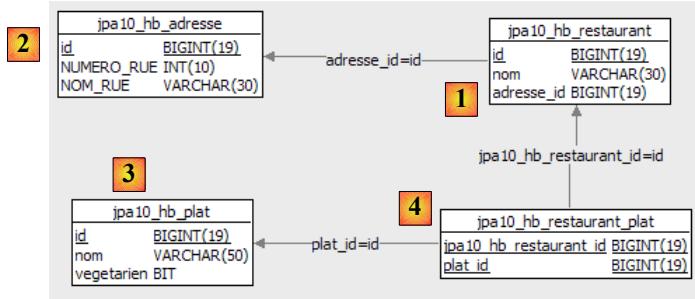
## 2.7 Exemple 6 : utiliser des requêtes nommées

Nous terminons cette longue présentation des entités JPA commencée page 9 par un dernier exemple qui montre l'utilisation de requêtes JPQL externalisées dans un fichier de configuration. Cet exemple tire son origine de la source suivante :

[ref2] : " Getting started With JPA in Spring 2.0 " de **Mark Fisher** à l'url  
[<http://blog.springframework.com/markf/archives/2006/05/30/getting-started-with-jpa-in-spring-20/>].

### 2.7.1 La base de données exemple

La base de données est la suivante :



- en [1] : une liste de restaurants avec leurs nom et adresse
- en [2] : la table des adresses des restaurants, limitées au n° dans la rue et nom de la rue. On a une relation **un-à-un** entre les tables *restaurant* et *adresse* : un restaurant a une adresse et une seule.
- en [3] : une table de plats avec leur nom et un indicateur vrai / faux pour dire si le plat est végétarien ou non
- en [4] : la table de jointure restaurants / plats : un restaurant sert plusieurs plats et un même plat peut être servi par plusieurs restaurants. On a une relation **plusieurs-à-plusieurs** entre les tables *restaurant* et *plat*.

### 2.7.2 Les objets @Entity représentant la base de données

Les tables précédentes vont être représentées par les @Entity suivantes :

- l'@Entity *Restaurant* représentera la table [restaurant]
- l'@Entity *Adresse* représentera la table [adresse]
- l'@Entity *Plat* représentera la table [plat]

Les relations entre ces entités sont les suivantes :

- une relation **un-à-un** relie l'entité *Restaurant* à l'entité *Adresse* : un restaurant *r* a une adresse *a*. L'entité *Restaurant* qui détient la clé étrangère aura la relation principale. L'entité *Adresse* n'aura pas de relation inverse.
- une relation **plusieurs-à-plusieurs** relie les entités *Restaurant* et *Plat* : un restaurant sert plusieurs plats et un même plat peut être servi par plusieurs restaurants. Cette relation va être matérialisée par une annotation **@ManyToMany** dans l'entité *Restaurant*. L'entité *Plat* n'aura pas de relation inverse.

L'@Entity *Restaurant* est la suivante :

```
1. package entites;
2.
3. ...
4. @Entity
5. @Table(name = "jpa10_hb_restaurant")
6. public class Restaurant implements java.io.Serializable {
7.
8.     private static final long serialVersionUID = 1L;
9.
10.    @Id
11.    @GeneratedValue(strategy = GenerationType.AUTO)
12.    private long id;
13.
14.    @Column(unique = true, length = 30, nullable = false)
15.    private String nom;
16.
17.    @OneToOne(cascade = CascadeType.ALL)
```

```

18. private Adresse adresse;
19.
20. @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
21. @JoinTable(name = "jpa10_hb_restaurant_plat", inverseJoinColumns = @JoinColumn(name = "plat_id"))
22. private Set<Plat> plats = new HashSet<Plat>();
23.
24. // constructeurs
25. public Restaurant() {
26.
27. }
28.
29. public Restaurant(String name, Adresse address, Set<Plat> entrees) {
30. ...
31. }
32.
33. // getters et setters
34. ...
35.
36. // toString
37. public String toString() {
38.     String signature = "R[" + getNom() + "," + getAddress();
39.     for (Plat e : getPlats()) {
40.         signature += "," + e;
41.     }
42.     return signature + "]";
43. }
44. }
```

- ligne 17 : la relation **un-à-un** qu'a l'entité *Restaurant* avec l'entité *Adresse*. Toutes les opérations de persistance sur un restaurant sont cascadées sur son adresse.
- ligne 20 : la relation qui lie l'**@Entity** *Restaurant* à l'**@Entity** *Plat* de l'ensemble *plats* de la ligne 22 est de type **plusieurs-à-plusieurs** (*ManyToMany*) :
  - un restaurant (One) a plusieurs plats (Many)
  - un plat (One) peut être servi par plusieurs restaurants (Many)
  - au final les **@Entity** *Restaurant* et *Plat* sont reliées par une relation *ManyToMany*. Nous décidons que l'**@Entity** *Restaurant* aura la relation principale et que l'**@Entity** *Plat* n'aura pas de relation inverse.
  - la relation **@ManyToMany** nécessite une table de jointure. Celle-ci est définie à l'aide de l'annotation **@JoinTable** de la ligne 47.
    - l'attribut **name** donne un nom à la table.
    - la table de jointure est constituée des clés étrangères sur les tables qu'elle joint. Ici, il y a deux clés étrangères : une sur la table [restaurant], l'autre sur la table [plat]. Ces colonnes clés étrangères sont définies par les attributs **joinColumns** et **inverseJoinColumns**.
    - l'attribut **joinColumns** définit la clé étrangère sur la table de l'**@Entity** détenant la relation principale **@ManyToMany**, ici la table [restaurant]. L'attribut **joinColumns** est ici absent. JPA a une valeur par défaut dans ce cas : [table]\_[clé\_primaire\_de\_table], ici [jpa10\_hb\_restaurant\_id].
    - l'annotation **@JoinColumn** de l'attribut **inverseJoinColumns** définit la clé étrangère sur la table de l'**@Entity** détenant la relation inverse **@ManyToMany**, ici la table [plat]. Cette colonne clé étrangère s'appellera *plat\_id*.

L'**@Entity** *Adresse* est la suivante :

```

1. package entites;
2.
3. ...
4. @Entity
5. @Table(name="jpa10_hb_adresse")
6. public class Adresse implements java.io.Serializable {
7.
8.     @Id
9.     @GeneratedValue(strategy = GenerationType.AUTO)
10.    private long id;
11.
12.    @Column(name = "NUMERO_RUE")
13.    private int numeroRue;
14.
15.    @Column(name = "NOM_RUE", length=30, nullable=false)
16.    private String nomRue;
17.
18.    // getters et setters
19. ...
20.
21.    // constructeurs
22.    public Adresse(int streetNumber, String streetName) {
23. ...
24.    }
25.
26.    public Adresse() {
27. }
```

```

28. }
29.
30. // toString
31. public String toString(){
32.     return "A["+getNumeroRue()+", "+getNomRue()+"]";
33. }
34. }

```

- l'@Entity *Adresse* est une entité sans relation directe avec les autres entités. On ne peut la persister qu'au travers d'une entité *Restaurant*.
- une adresse est définie par un nom de rue (ligne 16) et un n° dans la rue (ligne 13).

L'@Entity *Plat* est la suivante

```

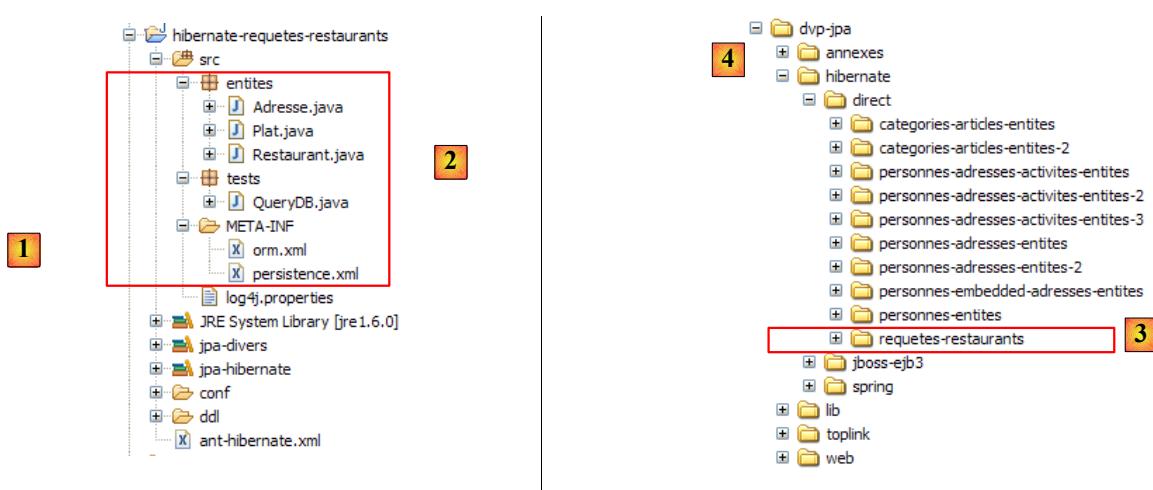
1. package entites;
2. ...
3. @Entity
4. @Table(name="jpa10_hb_plat")
5. public class Plat implements java.io.Serializable {
6.
7.     @Id
8.     @GeneratedValue(strategy = GenerationType.AUTO)
9.     private long id;
10.
11.    @Column(unique=true, length=50, nullable=false)
12.    private String nom;
13.
14.    private boolean vegetarien;
15.
16.    // constructeurs
17.    public Plat() {
18.
19.    }
20.
21.    public Plat(String name, boolean vegetarian) {
22.    ...
23.    }
24.
25.    // getters et setters
26.    ...
27.
28.    // toString
29.    public String toString() {
30.        return "E[" + getNom() + "," + isVegetarien() + "]";
31.    }
32.
33. }

```

- l'@Entity *Plat* est une entité sans relation directe avec les autres entités. On ne peut la persister qu'au travers d'une entité *Restaurant*.
- un plat est défini par un nom (ligne 12) et un type végétarien ou non (ligne 14).

### 2.7.3 Le projet Eclipse / Hibernate

L'implémentation JPA utilisée ici est celle d'Hibernate. Le projet Eclipse des tests est le suivant :



En [1], le projet Eclipse, en [2] les codes Java et la configuration de la couche JPA. On remarquera la présence d'un fichier [orm.xml] encore jamais rencontré. Le projet est présent en [3] dans le dossier des exemples [4]. On l'importera.

## 2.7.4 Génération de la DDL de la base de données

En suivant les instructions du paragraphe 2.1.7, page 20, la DDL obtenue pour le SGBD MySQL5 est la suivante :

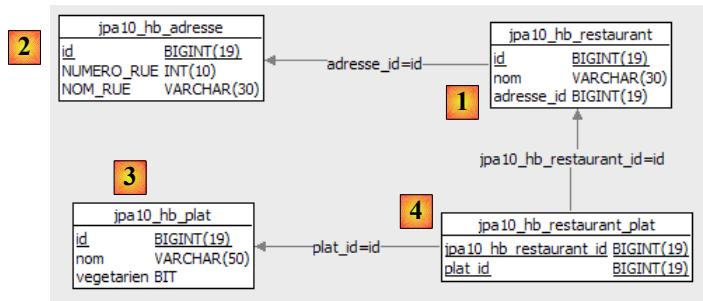
```

1. alter table jpa10_hb_restaurant
2.      drop
3.          foreign key FK3E8E4F5D5FE379D0;
4.
5. alter table jpa10_hb_restaurant_plat
6.      drop
7.          foreign key FK1D2D06D11F0F78A4;
8.
9. alter table jpa10_hb_restaurant_plat
10.     drop
11.        foreign key FK1D2D06D1AFAC3E44;
12.
13. drop table if exists jpa10_hb_adresse;
14.
15. drop table if exists jpa10_hb_plat;
16.
17. drop table if exists jpa10_hb_restaurant;
18.
19. drop table if exists jpa10_hb_restaurant_plat;
20.
21. create table jpa10_hb_adresse (
22.     id bigint not null auto_increment,
23.     NUMERO_RUE integer,
24.     NOM_RUE varchar(30) not null,
25.     primary key (id)
26. ) ENGINE=InnoDB;
27.
28. create table jpa10_hb_plat (
29.     id bigint not null auto_increment,
30.     nom varchar(50) not null unique,
31.     vegetarien bit not null,
32.     primary key (id)
33. ) ENGINE=InnoDB;
34.
35. create table jpa10_hb_restaurant (
36.     id bigint not null auto_increment,
37.     nom varchar(30) not null unique,
38.     adresse_id bigint,
39.     primary key (id)
40. ) ENGINE=InnoDB;
41.
42. create table jpa10_hb_restaurant_plat (
43.     jpa10_hb_restaurant_id bigint not null,
44.     plat_id bigint not null,
45.     primary key (jpa10_hb_restaurant_id, plat_id)
46. ) ENGINE=InnoDB;
47.
48. alter table jpa10_hb_restaurant
49.     add index FK3E8E4F5D5FE379D0 (adresse_id),
50.     add constraint FK3E8E4F5D5FE379D0
51.         foreign key (adresse_id)
52.             references jpa10_hb_adresse (id);
53.
54. alter table jpa10_hb_restaurant_plat
55.     add index FK1D2D06D11F0F78A4 (plat_id),
56.     add constraint FK1D2D06D11F0F78A4
57.         foreign key (plat_id)
58.             references jpa10_hb_plat (id);
59.
60. alter table jpa10_hb_restaurant_plat
61.     add index FK1D2D06D1AFAC3E44 (jpa10_hb_restaurant_id),
62.     add constraint FK1D2D06D1AFAC3E44
63.         foreign key (jpa10_hb_restaurant_id)
64.             references jpa10_hb_restaurant (id);

```

- lignes 21-26 : la table [adresse]
- lignes 28-33 : la table [plat]
- lignes 35-40 : la table [restaurant]
- lignes 42-46 : la table de jointure [restaurant\_plat]. On notera la clé composite (ligne 45)
- lignes 48-52 : la clé étrangère de la table [restaurant] vers la table [adresse]
- lignes 54-58 : la clé étrangère de la table [restaurant\_plat] vers la table [plat]
- lignes 60-64 : la clé étrangère de la table [restaurant\_plat] vers la table [restaurant]

Cette DDL correspond au schéma déjà présenté :



Dans la perspective SQL Explorer, la base se présente de la façon suivante :

Table	Attribut	Type	Attribut	Type
1	id	BIGINT(19)	NUMERO_RUE	INT(10)
1			NOM_RUE	VARCHAR(30)
2	id	BIGINT(19)	nom	VARCHAR(50)
2			vegetarien	BIT
3	id	BIGINT(19)	nom	VARCHAR(50)
3			vegetarien	BIT
4	id	BIGINT(19)	jpa10_hb_restaurant_id	BIGINT(19)
4			plat_id	BIGINT(19)

- en [1] : les 4 tables de la base
- en [2] : les adresses
- en [3] : les plats
- en [4] : les restaurants. [adresse\_id] référence les adresses de [2].
- en [5] : la table de jointure [restaurant,plat]. [jpa10\_hb\_restaurant\_id] référence les restaurants de [4] et [plat\_id] les plats de [3]. Ainsi [1,1] signifie que le restaurant " Burger Barn " sert le plat " CheeseBurger ".

Pour obtenir les données ci-dessus, le programme [QueryDB] du projet Eclipse a été exécuté.

## 2.7.5 Requêtes JPQL avec une console Hibernate

Nous créons une console Hibernate liée au projet Eclipse précédent. On suivra la démarche déjà exposée à deux reprises notamment au paragraphe 2.1.12, page 34.

- en [1] et [2] : la configuration de la console Hibernate

- en [3] : une requête JPQL et en [4] le résultat.
- en [5] : l'ordre SQL équivalent

Nous présentons maintenant une série de requêtes JPQL. Le lecteur est invité à les jouer et à découvrir l'ordre SQL généré par Hibernate pour l'exécuter.

Obtenir tous les restaurants avec leurs plats :

Obtenir les restaurants servant au moins un plat végétarien :

Obtenir les noms des restaurants qui ne servent que des plats végétariens :

Obtenir les restaurants qui servent des burgers :

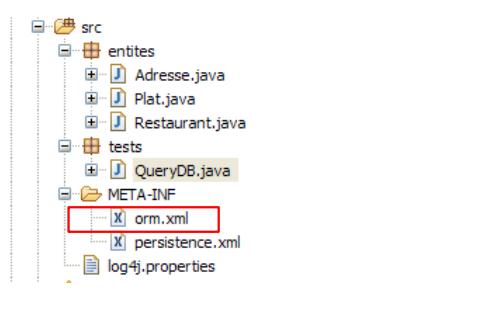
## 2.7.6 QueryDB

Nous nous intéressons maintenant au programme [QueryDB] du projet Eclipse qui :

- remplit la base

Persistance Java 5 par la pratique

- émet dessus un certain nombre de requêtes JPQL. Celles-ci sont enregistrées dans le fichier [META-INF/orm.xml] du projet Eclipse :



Le fichier [orm.xml] peut être utilisé pour configurer la couche JPA en lieu et place des annotations Java. Cela amène de la souplesse dans la configuration de la couche JPA. On peut la modifier sans recompilation des codes Java. On peut utiliser les deux méthodes simultanément : annotations Java et fichier [orm.xml]. La configuration JPA est d'abord faite avec les annotations Java puis avec le fichier [orm.xml]. Si donc on veut modifier une configuration faite par une annotation Java sans recompiler, il suffit de mettre cette configuration dans [orm.xml]. C'est elle qui aura le dernier mot.

Dans notre exemple, le fichier [orm.xml] est utilisé pour enregistrer des textes de requêtes JPQL. Son contenu est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
   http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
4.   <description>Restaurants</description>
5.   <named-query name="supprimer le contenu de la table restaurant">
6.     <query>delete from Restaurant</query>
7.   </named-query>
8.   <named-query name="supprimer le contenu de la table plat">
9.     <query>delete from Plat</query>
10.  </named-query>
11.  <named-query name="obtenir tous les restaurants">
12.    <query>select r from Restaurant r order by r.nom asc</query>
13.  </named-query>
14.  <named-query name="obtenir toutes les adresses">
15.    <query>select a from Adresse a order by a.nomRue asc</query>
16.  </named-query>
17.  <named-query name="obtenir tous les plats">
18.    <query>select p from Plat p order by p.nom asc</query>
19.  </named-query>
20.  <named-query name="obtenir tous les restaurants avec leurs plats">
21.    <query>select r.nom,p.nom from Restaurant r join r.plats p</query>
22.  </named-query>
23.  <named-query name="obtenir les restaurants ayant au moins un plat vegetarien">
24.    <query>select distinct r from Restaurant r join r.plats p where p.vegetarien=true</query>
25.  </named-query>
26.  <named-query name="obtenir les restaurants avec uniquement des plats vegetariens">
27.    <query>
28.      select distinct r1.nom from Restaurant r1 where not exists (select p1 from Restaurant r2
join r2.plats p1 where r2.id=r1.id and
29.          p1.vegetarien=false)
30.    </query>
31.  </named-query>
32.  <named-query name="obtenir les restaurants d'une certaine rue">
33.    <query>select r from Restaurant r where r.adresse.nomRue=:nomRue</query>
34.  </named-query>
35.  <named-query name="obtenir les restaurants qui servent des burgers">
36.    <query>select r.nom,r.adresse.numeroRue, r.adresse.nomRue, p.nom from Restaurant r join
r.plats p where p.nom like '%burger'</query>
37.  </named-query>
38.  <named-query name="obtenir les plats du restaurant untel">
39.    <query>select p.nom from Restaurant r join r.plats p where r.nom=:nomRestaurant</query>
40.  </named-query>
41.</entity-mappings>
```

- la racine du fichier [orm.xml] est <entity-mappings> (ligne 2).
- lignes 5-7 : les requêtes JPQL nommées font l'objet de balises <named-query name="...">...</namedquery>.
  - l'attribut *name* de la balise est le nom de la requête.
  - le contenu *texte* de la balise est le texte de la requête.

QueryDB va exécuter les requêtes précédentes. Son code est le suivant :

```
1. package tests;
```

```

2.
3. ...
4. public class QueryDB {
5.
6.     // Contexte de persistance
7.     private static EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
8.
9.     private static EntityManager em = emf.createEntityManager();
10.
11.    public static void main(String[] args) {
12.        // début transaction
13.        EntityTransaction tx = em.getTransaction();
14.        tx.begin();
15.        // supprimer les éléments de la table [restaurant]
16.        em.createNamedQuery("supprimer le contenu de la table restaurant").executeUpdate();
17.        // supprimer les éléments de la table [plat]
18.        em.createNamedQuery("supprimer le contenu de la table plat").executeUpdate();
19.        // création d'objets Address
20.        Adresse adr1 = new Adresse(10, "Main Street");
21.        Adresse adr2 = new Adresse(20, "Main Street");
22.        Adresse adr3 = new Adresse(123, "Dover Street");
23.        // création d'objets Entrée
24.        Plat ent1 = new Plat("Hamburger", false);
25.        Plat ent2 = new Plat("Cheeseburger", false);
26.        Plat ent3 = new Plat("Tofu Stir Fry", true);
27.        Plat ent4 = new Plat("Vegetable Soup", true);
28.        // création d'objets Restaurant
29.        Restaurant restaurant1 = new Restaurant();
30.        restaurant1.setNom("Burger Barn");
31.        restaurant1.setAdresse(adr1);
32.        restaurant1.getPlats().add(ent1);
33.        restaurant1.getPlats().add(ent2);
34.        Restaurant restaurant2 = new Restaurant();
35.        restaurant2.setNom("Veggie Village");
36.        restaurant2.setAdresse(adr2);
37.        restaurant2.getPlats().add(ent3);
38.        restaurant2.getPlats().add(ent4);
39.        Restaurant restaurant3 = new Restaurant();
40.        restaurant3.setNom("Dover Diner");
41.        restaurant3.setAdresse(adr3);
42.        restaurant3.getPlats().add(ent1);
43.        restaurant3.getPlats().add(ent2);
44.        restaurant3.getPlats().add(ent4);
45.        // persistance des objets Restaurant (et des autres objets par cascade)
46.        em.persist(restaurant1);
47.        em.persist(restaurant2);
48.        em.persist(restaurant3);
49.        // fin transaction
50.        tx.commit();
51.        // dump base
52.        dumpDataBase();
53.        // fin EntityManager
54.        em.close();
55.        // fin EntityManagerFactory
56.        emf.close();
57.    }
58.
59.    // affichage contenu de la base
60.    @SuppressWarnings("unchecked")
61.    private static void dumpDataBase() {
62.        // test2
63.        log("données de la base");
64.        // début transaction
65.        EntityTransaction tx = em.getTransaction();
66.        tx.begin();
67.        // affichages restaurants
68.        log("[restaurants]");
69.        for (Object restaurant : em.createNamedQuery("obtenir tous les restaurants").getResultList())
    {
70.            System.out.println(restaurant);
71.        }
72.        // affichages adresses
73.        log("[adresses]");
74.        for (Object adresse : em.createNamedQuery("obtenir toutes les adresses").getResultList()) {
75.            System.out.println(adresse);
76.        }
77.        // affichages plats
78.        log("[plats]");
79.        for (Object plat : em.createNamedQuery("obtenir tous les plats").getResultList()) {
80.            System.out.println(plat);
81.        }
82.        // affichages liens restaurants <--> plats
83.        log("[restaurants/plats]");

```

```

84.     Iterator record = em.createNamedQuery("obtenir tous les restaurants avec leurs
85.     plats").getResultList().iterator();
86.     while (record.hasNext()) {
87.         Object[] currentRecord = (Object[]) record.next();
88.         System.out.format("[%s,%s]%n", currentRecord[0], currentRecord[1]);
89.     }
90.     log("[Liste des restaurants avec au moins un plat végétarien]");
91.     for (Object r : em.createNamedQuery("obtenir les restaurants ayant au moins un plat
92.     vegetarien").getResultList()) {
93.         System.out.println(r);
94.     }
95.     // query
96.     log("[Liste des restaurants avec seulement des plats végétariens]");
97.     for (Object r : em.createNamedQuery("obtenir les restaurants avec uniquement des plats
98.     vegetariens").getResultList()) {
99.         System.out.println(r);
100.    }
101.   // query
102.   log("[Liste des restaurants dans Dover Street]");
103.   for (Object r : em.createNamedQuery("obtenir les restaurants d'une certaine
104.   rue").setParameter("nomRue", "Dover Street").getResultList()) {
105.       System.out.println(r);
106.   }
107.   // query
108.   log("[Liste des restaurants ayant un plat de type burger]");
109.   record = em.createNamedQuery("obtenir les restaurants qui servent des
110.   burgers").getResultList().iterator();
111.   while (record.hasNext()) {
112.       Object[] currentRecord = (Object[]) record.next();
113.       System.out.format("[%s,%d,%s,%s]%n", currentRecord[0], currentRecord[1], currentRecord[2],
114.       currentRecord[3]);
115.   }
116.   // fin transaction
117.   tx.commit();
118.
119. // logs
120. private static void log(String message) {
121.     System.out.println(" -----" + message);
122. }
123.
124.

```

Le résultat de l'exécution de [QueryDB] est le suivant :

```

1.-----données de la base
2. -----[restaurants]
3.R[Burger Barn,A[10,Main Street],E[Cheeseburger,false],E[Hamburger,false]]
4.R[Dover Diner,A[123,Dover Street],E[Cheeseburger,false],E[Hamburger,false],E[Vegetable Soup,true]]
5.R[Veggie Village,A[20,Main Street],E[Tofu Stir Fry,true],E[Vegetable Soup,true]]
6. -----[adresses]
7.A[123,Dover Street]
8.A[10,Main Street]
9.A[20,Main Street]
10. -----[plats]
11.E[Cheeseburger,false]
12.E[Hamburger,false]
13.E[Tofu Stir Fry,true]
14.E[Vegetable Soup,true]
15. -----[restaurants/plats]
16.[Burger Barn,Cheeseburger]
17.[Burger Barn,Hamburger]
18.[Dover Diner,Cheeseburger]
19.[Dover Diner,Hamburger]
20.[Dover Diner,Vegetable Soup]
21.[Veggie Village,Tofu Stir Fry]
22.[Veggie Village,Vegetable Soup]
23. -----[Liste des restaurants avec au moins un plat végétarien]
24.R[Veggie Village,A[20,Main Street],E[Tofu Stir Fry,true],E[Vegetable Soup,true]]
25.R[Dover Diner,A[123,Dover Street],E[Cheeseburger,false],E[Hamburger,false],E[Vegetable Soup,true]]
26. -----[Liste des restaurants avec seulement des plats végétariens]
27.Veggie Village
28. -----[Liste des restaurants dans Dover Street]
29.R[Dover Diner,A[123,Dover Street],E[Cheeseburger,false],E[Hamburger,false],E[Vegetable Soup,true]]
30. -----[Liste des restaurants ayant un plat de type burger]
31.[Burger Barn,10,Main Street,Cheeseburger]
32.[Burger Barn,10,Main Street,Hamburger]
33.[Dover Diner,123,Dover Street,Cheeseburger]

```

```

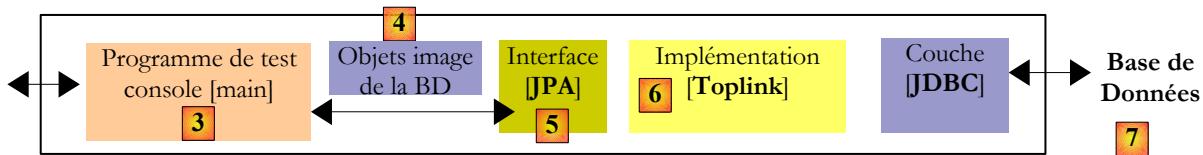
34. [Dover Diner,123,Dover Street,Hamburger]
35. -----[Plats de Veggie Village]
36.Tofu Stir Fry
37.Vegetable Soup

```

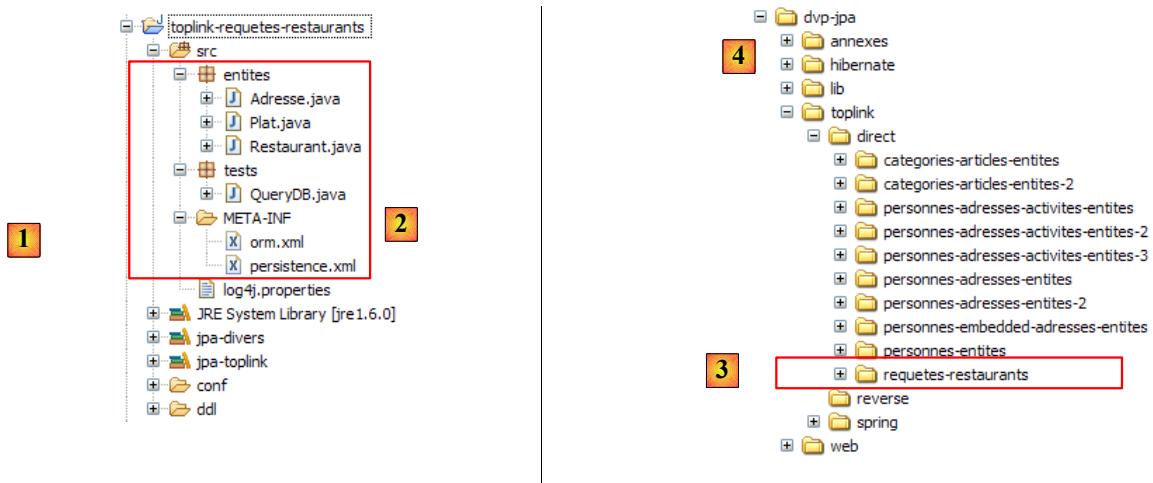
Nous laissons au lecteur, le soin de faire le lien entre le code et les résultats. Pour cela, nous lui conseillons de jouer les requêtes JPQL dans la console Hibernate et d'examiner le code SQL qui va avec.

## 2.7.7 Le projet Eclipse / Toplink

Le lecteur intéressé trouvera dans les exemples téléchargeables avec ce tutoriel le projet précédent implémenté avec Toplink :



Le projet Eclipse avec Toplink est une copie du projet Eclipse avec Hibernate :



Le fichier <persistence.xml> [2] déclare les entités gérées :

```

1.      <!-- provider -->
2.      <provider>oracle.toplink.essentials.PersistenceProvider</provider>
3.          <!-- classes persistantes -->
4.          <class>entites.Restaurnt</class>
5.          <class>entites.Adresse</class>
6.          <class>entites.Plat</class>
7.
8. ...

```

- lignes 4-6 : les entités gérées

Les requêtes JPQL enregistrées dans [orm.xml] sont correctement exécutées par Toplink. Pour cela, dans le projet précédent on avait pris soin de ne pas utiliser de requêtes HQL (Hibernate Query Language) qui est de fait un sur-ensemble de JPQL et dont certaines syntaxes ne sont pas acceptées par JPQL.

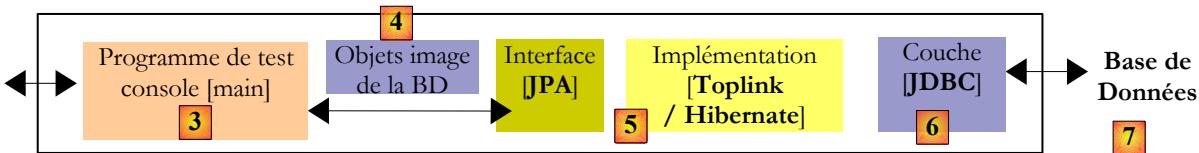
## 2.8 Conclusion

Nous terminons ici notre étude des entités JPA. Ce fut long et pourtant des choses importantes (pour le développeur avancé) n'ont pas été faites. De nouveau, il est conseillé de lire un livre de référence tel que celui qui a été utilisé pour ce tutoriel :

[refl] : **Java Persistence with Hibernate**, de Christian Bauer et Gavin King, chez Manning.

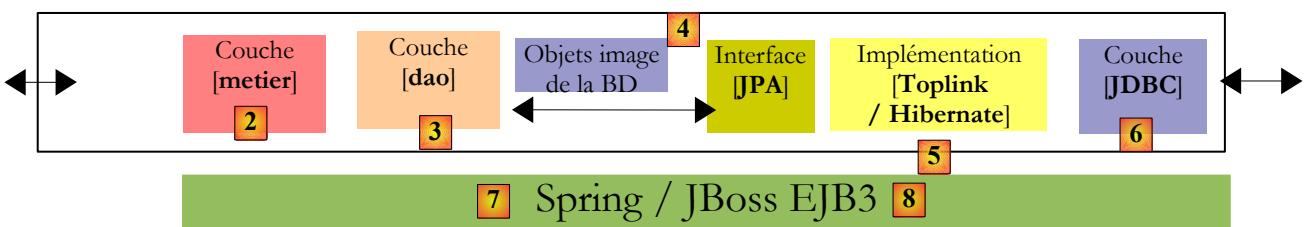
### 3 JPA dans une architecture multicouches

Pour étudier l'API JPA, nous avons utilisé l'architecture de test suivante :



Nos programmes de tests étaient des applications console qui interrogaient directement la couche JPA. Nous avons découvert à cette occasion les principales méthodes de la couche JPA. Nous étions dans un environnement dit "Java SE" (Standard Edition). JPA fonctionne à la fois dans un environnement Java SE et Java EE5 (Edition Entreprise).

Maintenant que nous avons une certaine maîtrise à la fois de la configuration du pont relationnel / objet et de l'utilisation des méthodes de la couche JPA, nous revenons à une architecture multi-couches plus classique :



La couche [JPA] sera accédée via une architecture à 2 couches [metier] et [dao]. Le framework Spring [7], puis le conteneur EJB3 de JBoss [8] seront utilisés pour lier ces couches entre-elles.

Nous avons dit plus haut que JPA était disponible dans les environnements SE et EE5. L'environnement Java EE5 délivre de nombreux services dans le domaine de l'accès aux données persistantes notamment les pools de connexion, les gestionnaire de transactions, ... Il peut être intéressant pour un développeur de profiter de ces services. L'environnement Java EE5 n'est pas encore très répandu (mai 2007). On le trouve actuellement sur le serveurs d'application *Sun Application Server 9.x (Glassfish)*. Un serveur d'application est essentiellement un serveur d'applications web. Si on construit une application graphique autonome de type Swing, on ne peut disposer de l'environnement EE et des services qu'il apporte. C'est un problème. On commence à voir des environnements EE "stand-alone", c.a.d. pouvant être utilisés en-dehors d'un serveur d'applications. C'est le cas de JBos EJB3 que nous allons utiliser dans ce document.

Dans un environnement EE5, les couches sont implémentées par des objets appelés EJB (Enterprise Java Bean). Dans les précédentes versions d'EE, les EJB (EJB 2.x) sont réputés difficiles à mettre en oeuvre, à tester et parfois peu-performants. On distingue les EJB2.x "entity" et les EJB2.x "session". Pour faire court, un EJB2.x "entity" est l'image d'une ligne de table de base de données et EJB2.x "session" un objet utilisé pour implémenter les couches [metier], [dao] d'une architecture multi-couches. L'un des principaux reproches faits aux couches implémentées avec des EJB est qu'elles ne sont utilisables qu'au sein de conteneurs EJB, un service délivré par l'environnement EE. Cela rend problématiques les tests unitaires. Ainsi dans le schéma ci-dessus, les tests unitaires des couches [metier] et [dao] construits avec des EJB nécessiteraient la mise en place d'un serveur d'application, une opération assez lourde qui n'incite pas vraiment le développeur à faire fréquemment des tests.

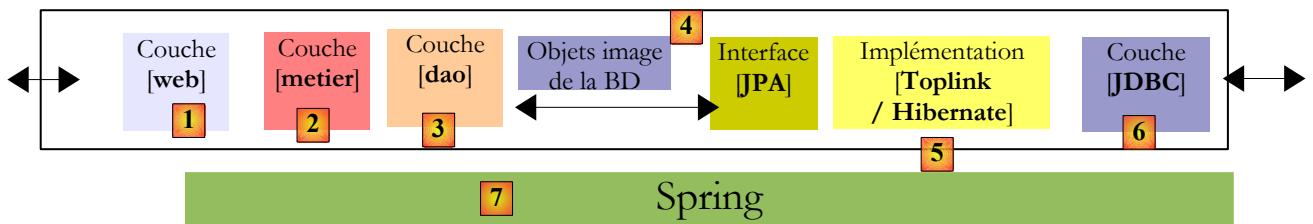
Le framework **Spring** est né en réaction à la complexité des EJB2. Spring fournit dans un environnement SE un nombre important des services habituellement fournis par les environnements EE. Ainsi dans la partie "Persistance de données" qui nous intéresse ici, Spring fournit les pools de connexion et les gestionnaires de transactions dont ont besoin les applications. L'émergence de Spring a favorisé la culture des tests unitaires, devenus d'un seul coup beaucoup plus faciles à mettre en oeuvre. Spring permet l'implémentation des couches d'une application par des objets Java classiques (POJO, Plain Old/Ordinary Java Object), permettant la réutilisation de ceux-ci dans un autre contexte. Enfin, il intègre de nombreux outils tiers de façon assez transparente, notamment des outils de persistance tels que Hibernate, Ibatis, ...

Java EE5 a été conçu pour corriger les lacunes de la précédente spécification EE. Les EJB 2.x sont devenus les EJB3. Ceux-ci sont des POJOs tagués par des annotations qui en font des objets particuliers lorsqu'ils sont au sein d'un conteneur EJB3. Dans celui-ci, l'EJB3 va pouvoir bénéficier des services du conteneur (pool de connexions, gestionnaire de transactions, ...). En-dehors du conteneur EJB3, l'EJB3 devient un objet Java normal. Ses annotations EJB sont ignorées.

Ci-dessus, nous avons représenté Spring et JBoss EJB3 comme infrastructure (framework) possible de notre architecture multi-couches. C'est cette infrastructure qui délivrera les services dont nous avons besoin : un pool de connexions et un gestionnaire de transactions.

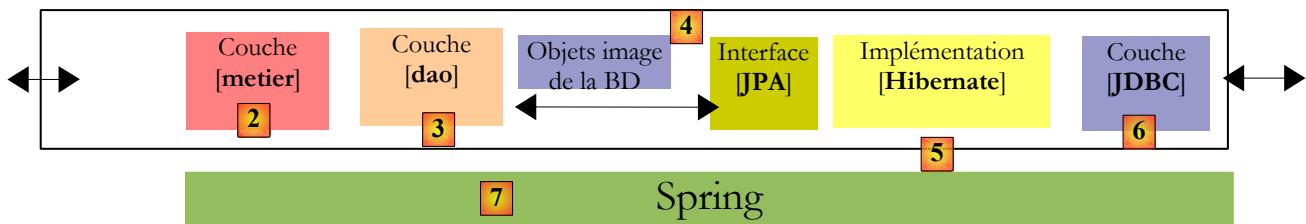
- avec Spring, les couches seront implémentées avec des POJOs. Ceux-ci auront accès aux services de Spring (pool de connexions, gestionnaire de transaction) par injection de dépendances dans ces POJOs : lors de la construction de ceux-ci, Spring leur injecte des références sur les services dont il vont avoir besoin.
- JBoss EJB3 est un conteneur EJB pouvant fonctionner en-dehors d'un serveur d'application. Son principe de fonctionnement (pour le développeur) est analogue à celui décrit pour Spring. Nous trouverons peu de différences.

Nous terminerons le document avec un exemple d'application web à trois couches, basique mais néanmoins représentative :



### 3.1 Exemple 1 : Spring / JPA avec entité Personne

Nous prenons l'entité *Personne* étudiée au paragraphe 2.1, page 9 et nous l'intégrons dans une architecture multi-couches où l'intégration des couches est faite avec Spring et la couche de persistance est implémentée par Hibernate.

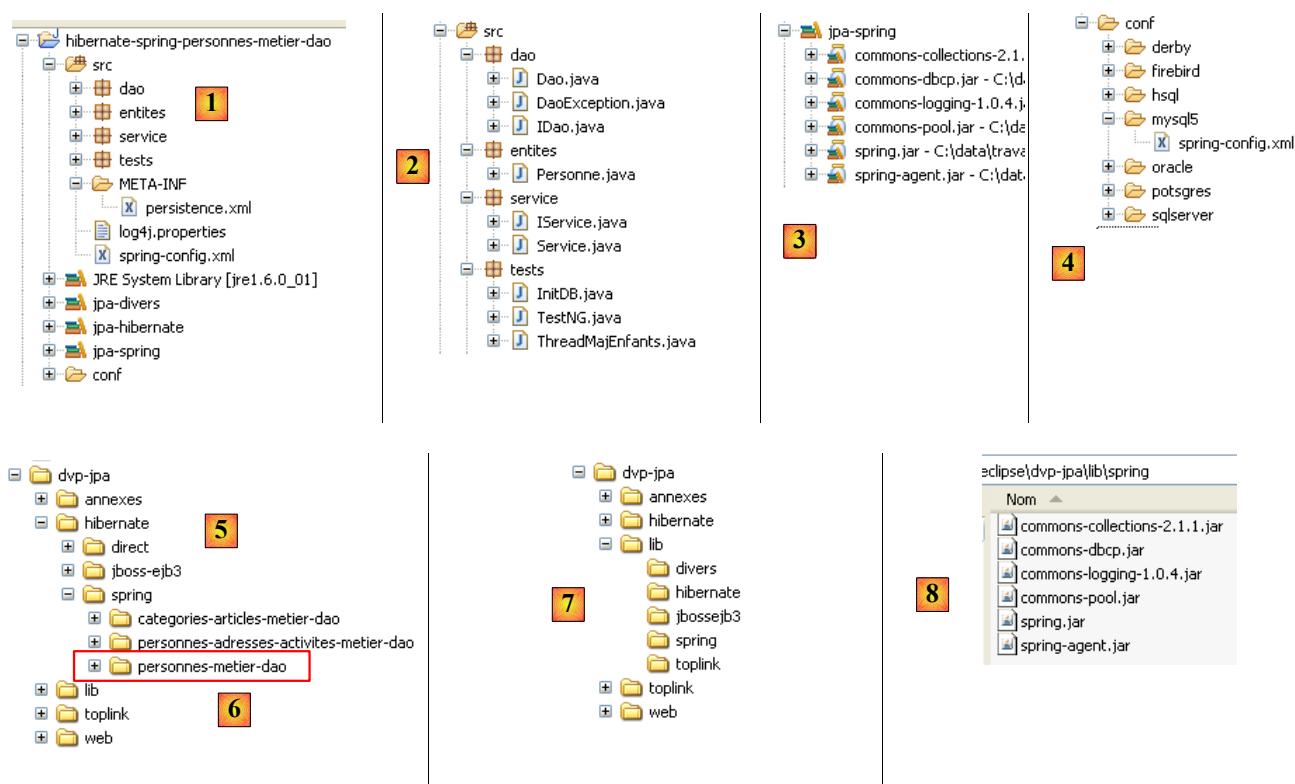


Le lecteur est ici supposé avoir des connaissances de base sur Spring. Si ce n'était pas le cas, on pourra lire le document suivant qui explique la notion d'**injection de dépendances** qui est au coeur de Spring :

[ref3] : **Spring Ioc** (Inversion Of Control) [<http://tahe.developpez.com/java/springioc>].

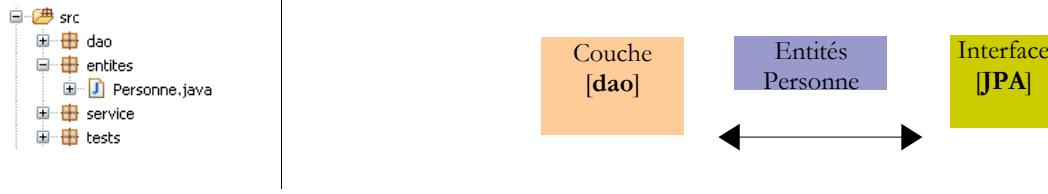
#### 3.1.1 Le projet Eclipse / Spring / Hibernate

Le projet Eclipse est le suivant :



- en [1] : le projet Eclipse. Il sera trouvé en [6] dans les exemples du tutoriel [5]. On l'importera.
- en [2] : les codes Java des couches présentés en paquetages :
  - [entités] : le paquetage des entités JPA
  - [dao] : la couche d'accès aux données - s'appuie sur la couche JPA
  - [service] : une couche de services plus que de métier. On y utilisera le service de transactions des conteneurs.
  - [tests] : regroupe les programmes de tests.
- en [3] : la bibliothèque [jpa-spring] regroupe les jars nécessaires à Spring (voir aussi [7] et [8]).
- en [4] : le dossier [conf] rassemble les fichiers de configuration de Spring pour chacun des SGBD utilisés dans ce tutoriel.

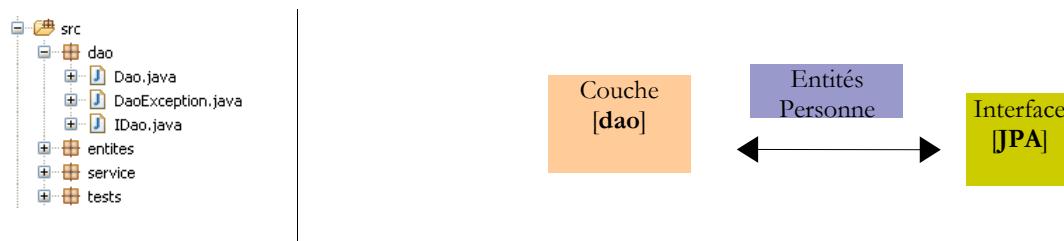
### 3.1.2 Les entités JPA



Il n'y a qu'une entité gérée ici, l'entité *Personne* étudiée au paragraphe 2.1, page 9, et dont nous rappelons ci-dessous la configuration :

```
1. package entites;
2.
3. ...
4. @Entity
5. @Table(name="jpa01_hb_personne")
6. public class Personne {
7.
8.     @Id
9.     @Column(name = "ID", nullable = false)
10.    @GeneratedValue(strategy = GenerationType.AUTO)
11.    private Integer id;
12.
13.    @Column(name = "VERSION", nullable = false)
14.    @Version
15.    private int version;
16.
17.    @Column(name = "NOM", length = 30, nullable = false, unique = true)
18.    private String nom;
19.
20.    @Column(name = "PRENOM", length = 30, nullable = false)
21.    private String prenom;
22.
23.    @Column(name = "DATENAISSEANCE", nullable = false)
24.    @Temporal(TemporalType.DATE)
25.    private Date datenaissance;
26.
27.    @Column(name = "MARIE", nullable = false)
28.    private boolean marie;
29.
30.    @Column(name = "NBENFANTS", nullable = false)
31.    private int nbefants;
32.
33.    // constructeurs
34.    public Personne() {
35.    }
36.
37.    public Personne(String nom, String prenom, Date datenaissance, boolean marie,
38.        int nbefants) {
39.    ...
40.    }
41.
42.    // toString
43.    public String toString() {
44.        return String.format("[%d,%d,%s,%s,%s,%d]", getId(), getVersion(),
45.            getNom(), getPrenom(), new SimpleDateFormat("dd/MM/yyyy")
46.                .format(getDatenaissance()), isMarie(), getNbefants());
47.    }
48.
49.    // getters and setters
50.    ...
51. }
```

### 3.1.3 La couche [dao]



La couche [dao] présente l'interface IDao suivante :

```
1. package dao;
2.
3. import java.util.List;
4.
5. import entites.Personne;
6.
7. public interface IDao {
8.     // obtenir une personne via son identifiant
9.     public Personne getOne(Integer id);
10.
11.    // obtenir toutes les personnes
12.    public List<Personne> getAll();
13.
14.    // sauvegarder une personne
15.    public Personne saveOne(Personne personne);
16.
17.    // mettre à jour une personne
18.    public Personne updateOne(Personne personne);
19.
20.    // supprimer une personne via son identifiant
21.    public void deleteOne(Integer id);
22.
23.    // obtenir les personnes dont le nom correspond à un modèle
24.    public List<Personne> getAllLike(String modele);
25.
26. }
```

L'implémentation [Dao] de cette interface est la suivante :

```
1. package dao;
2.
3. import java.util.List;
4.
5. import javax.persistence.EntityManager;
6. import javax.persistence.PersistenceContext;
7.
8. import entites.Personne;
9.
10. public class Dao implements IDao {
11.
12.     @PersistenceContext
13.     private EntityManager em;
14.
15.     // supprimer une personne via son identifiant
16.     public void deleteOne(Integer id) {
17.         Personne personne = em.find(Personne.class, id);
18.         if (personne == null) {
19.             throw new DaoException(2);
20.         }
21.         em.remove(personne);
22.     }
23.
24.     @SuppressWarnings("unchecked")
25.     // obtenir toutes les personnes
26.     public List<Personne> getAll() {
27.         return em.createQuery("select p from Personne p").getResultList();
28.     }
29.
30.     @SuppressWarnings("unchecked")
31.     // obtenir les personnes dont le nom correspond à un modèle
32.     public List<Personne> getAllLike(String modele) {
33.         return em.createQuery("select p from Personne p where p.nom like :modele")
34.             .setParameter("modele", modele).getResultList();
35.     }
36.
37.     // obtenir une personne via son identifiant
38.     public Personne getOne(Integer id) {
39.         return em.find(Personne.class, id);
40.     }
41.
42.     // sauvegarder une personne
43.     public Personne saveOne(Personne personne) {
44.         em.persist(personne);
45.         return personne;
46.     }
47.
48.     // mettre à jour une personne
49.     public Personne updateOne(Personne personne) {
50.         return em.merge(personne);
51.     }
```

```
52.  
53. }
```

- tout d'abord, on notera la simplicité de l'implémentation [Dao]. Celle-ci est due à l'utilisation de la couche JPA qui fait l'essentiel du travail d'accès aux données.
- ligne 10 : la classe [Dao] implémente l'interface [IDao]
- ligne 13 : l'objet de type [EntityManager] qui va être utilisé pour manipuler le contexte de persistance JPA. Par abus de langage, nous le confondrons parfois avec le contexte de persistance lui-même. Le contexte de persistance contiendra des entités *Personne*.
- ligne 12 : nulle part dans le code, le champ [EntityManager em] n'est initialisé. Il le sera au démarrage de l'application par Spring. C'est l'annotation JPA **@PersistenceContext** de la ligne 12 qui demande à Spring d'injecter dans *em*, un gestionnaire de contexte de persistance.
- lignes 26-28 : la liste de toutes les personnes est obtenue par une requête JPQL.
- lignes 32-35 : la liste de toutes les personnes ayant un nom correspondant à un certain modèle est obtenue par une requête JPQL.
- lignes 38-40 : la personne ayant tel identifiant est obtenue par la méthode **find** de l'API JPA. Rend un pointeur *null* si la personne n'existe pas.
- lignes 43-46 : une personne est rendue persistante par la méthode **persist** de l'API JPA. La méthode rend la personne persistante.
- lignes 49-51 : la mise à jour d'une personne est réalisée par la méthode **merge** de l'API JPA. Cette méthode n'a de sens que si la personne ainsi mise à jour était auparavant détachée. La méthode rend la personne persistante ainsi créée.
- lignes 16-22 : la suppression de la personne dont on nous passe l'identifiant en paramètre se fait en deux temps :
  - ligne 17 : elle est cherchée dans le contexte de persistance
  - lignes 18-20 : si on ne la trouve pas, on lance une exception avec un code erreur 2
  - ligne 21 : si on l'a trouvée on l'enlève du contexte de persistance avec la méthode **remove** de l'API JPA.
- ce qui n'est pas visible pour l'instant est que chaque méthode sera exécutée au sein d'une transaction démarrée par la couche [service].

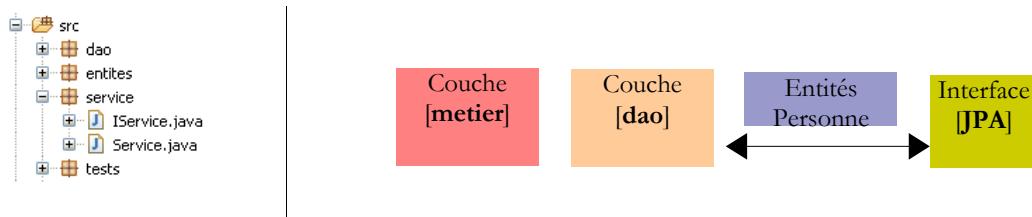
L'application a son propre type d'exception nommé [DaoException] :

```
1. package dao;  
2.  
3. @SuppressWarnings("serial")  
4. public class DaoException extends RuntimeException {  
5.  
6.     // code d'erreur  
7.     private int code;  
8.  
9.     public DaoException(int code) {  
10.         super();  
11.         this.code = code;  
12.     }  
13.  
14.    public DaoException(String message, int code) {  
15.        super(message);  
16.        this.code = code;  
17.    }  
18.  
19.    public DaoException(Throwable cause, int code) {  
20.        super(cause);  
21.        this.code = code;  
22.    }  
23.  
24.    public DaoException(String message, Throwable cause, int code) {  
25.        super(message, cause);  
26.        this.code = code;  
27.    }  
28.  
29.    // getter et setter  
30.  
31.    public int getCode() {  
32.        return code;  
33.    }  
34.  
35.    public void setCode(int code) {  
36.        this.code = code;  
37.    }  
38.  
39.}
```

- ligne 4 : [DaoException] dérive de [RuntimeException]. C'est donc un type d'exceptions que le compilateur ne nous oblige pas à gérer par un try / catch ou à mettre dans la signature des méthodes. C'est pour cette raison, que [DaoException] n'est pas dans la signature de la méthode [deleteOne] de l'interface [IDao]. Cela permet à cette interface d'être implémentée par une classe lançant un autre type d'exceptions pourvu que celui-ci dérive également de [RuntimeException].

- pour différencier les erreurs qui peuvent se produire, on utilise le code erreur de la ligne 7. Les trois constructeurs des lignes 14, 19 et 24 sont ceux de la classe parente [RuntimeException] auxquels on a rajouté un paramètre : celui du code d'erreur qu'on veut donner à l'exception.

### 3.1.4 La couche [metier / service]



La couche [service] présente l'interface [IService] suivante :

```

1. package service;
2.
3. import java.util.List;
4.
5. import entites.Personne;
6.
7. public interface IService {
8.     // obtenir une personne via son identifiant
9.     public Personne getOne(Integer id);
10.
11.    // obtenir toutes les personnes
12.    public List<Personne> getAll();
13.
14.    // sauvegarder une personne
15.    public Personne saveOne(Personne personne);
16.
17.    // mettre à jour une personne
18.    public Personne updateOne(Personne personne);
19.
20.    // supprimer une personne via son identifiant
21.    public void deleteOne(Integer id);
22.
23.    // obtenir les personnes dont le nom correspond à un modèle
24.    public List<Personne> getAllLike(String modele);
25.
26.    // supprimer plusieurs personnes à la fois
27.    public void deleteArray(Personne[] personnes);
28.
29.    // sauvegarder plusieurs personnes à la fois
30.    public Personne[] saveArray(Personne[] personnes);
31.
32.    // mettre à jour plusieurs personnes à la fois
33.    public Personne[] updateArray(Personne[] personnes);
34.
35. }

```

- lignes 8-24 : l'interface [IService] reprend les méthodes de l'interface [IDao]
- ligne 27 : la méthode [deleteArray] permet de supprimer un ensemble de personnes au sein d'une transaction : toutes les personnes sont supprimées ou aucune.
- lignes 30 et 33 : des méthodes analogues à [deleteArray] pour sauvegarder (ligne 30) ou mettre à jour (ligne 33) un ensemble de personnes au sein d'une transaction.

L'implémentation [Service] de l'interface [IService] est la suivante :

```

1. package service;
2.
3. ...
4.
5. // toutes les méthodes de la classe se déroulent dans une transaction
6. @Transactional
7. public class Service implements IService {
8.
9.     // couche [dao]
10.    private IDao dao;
11.
12.    public IDao getDao() {
13.        return dao;
14.    }

```

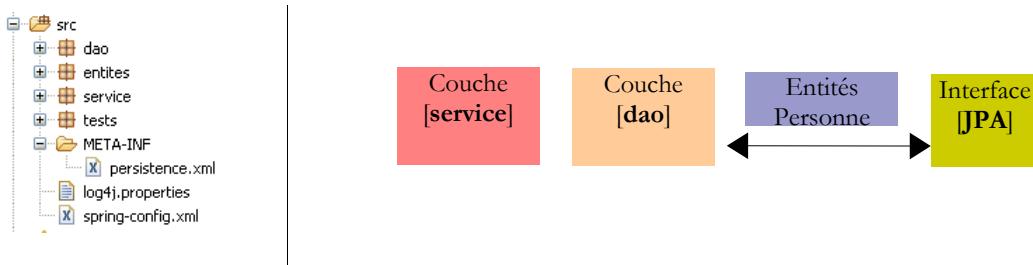
```

15.
16. public void setDao(IDao dao) {
17.     this.dao = dao;
18. }
19.
20. // supprimer plusieurs personnes à la fois
21. public void deleteArray(Personne[] personnes) {
22.     for (Personne p : personnes) {
23.         dao.deleteOne(p.getId());
24.     }
25. }
26.
27. // supprimer une personne via son identifiant
28. public void deleteOne(Integer id) {
29.     dao.deleteOne(id);
30. }
31.
32. // obtenir toutes les personnes
33. public List<Personne> getAll() {
34.     return dao.getAll();
35. }
36.
37. // obtenir les personnes dont le nom correspond à un modèle
38. public List<Personne> getAllLike(String modele) {
39.     return dao.getAllLike(modele);
40. }
41.
42. // obtenir une personne via son identifiant
43. public Personne getOne(Integer id) {
44.     return dao.getOne(id);
45. }
46.
47. // sauvegarder plusieurs personnes à la fois
48. public Personne[] saveArray(Personne[] personnes) {
49.     Personne[] personnes2 = new Personne[personnes.length];
50.     for (int i = 0; i < personnes.length; i++) {
51.         personnes2[i] = dao.saveOne(personnes[i]);
52.     }
53.     return personnes2;
54. }
55.
56. // sauvegarder une personne
57. public Personne saveOne(Personne personne) {
58.     return dao.saveOne(personne);
59. }
60.
61. // mettre à jour plusieurs personnes à la fois
62. public Personne[] updateArray(Personne[] personnes) {
63.     Personne[] personnes2 = new Personne[personnes.length];
64.     for (int i = 0; i < personnes.length; i++) {
65.         personnes2[i] = dao.updateOne(personnes[i]);
66.     }
67.     return personnes2;
68. }
69.
70. // mettre à jour une personne
71. public Personne updateOne(Personne personne) {
72.     return dao.updateOne(personne);
73. }
74.
75. }

```

- ligne 6 : l'annotation Spring **@Transactional** indique que toutes les méthodes de la classe doivent s'exécuter au sein d'une transaction. Une transaction sera commencée avant le début d'exécution de la méthode et fermée après exécution. Si une exception de type [RuntimeException] ou dérivé se produit au cours de l'exécution de la méthode, un **rollback** automatique annule toute la transaction, sinon un **commit** automatique la valide. On retiendra que le code Java n'a pas besoin de se soucier des transactions. Elles sont gérées par Spring.
- ligne 10 : une référence sur la couche [dao]. Nous verrons ultérieurement que cette référence est initialisée par Spring au démarrage de l'application.
- les méthodes de [Service] se contentent d'appeler les méthodes de l'interface [IDao dao] de la ligne 10. Nous laissons le lecteur prendre connaissance du code. Il n'y a pas de difficultés particulières.
- nous avons dit précédemment que chaque méthode de [Service] s'exécutait dans une transaction. Celle-ci est attachée au thread d'exécution de la méthode. Dans ce thread, sont exécutées des méthodes de la couche [dao]. Celles-ci seront automatiquement rattachées à la transaction du thread d'exécution. La méthode [deleteArray] (ligne 21), par exemple, est amenée à exécuter N fois la méthode [deleteOne] de la couche [dao]. Ces N exécutions se feront au sein du thread d'exécution de la méthode [deleteArray], donc au sein de la même transaction. Aussi seront-elles soit toutes validées (commit) si les choses se passent bien ou toutes annulées (rollback) si une exception se produit dans l'une des N exécutions de la méthode [deleteOne] de la couche [dao].

### 3.1.5 Configuration des couches



La configuration des couches [service], [dao] et [JPA] est assurée par deux fichiers ci-dessus : [META-INF/**persistence.xml**] et [**spring-config.xml**]. Les deux fichiers doivent être dans le *classpath* de l'application, ce qui explique qu'ils soient dans le dossier [src] du projet Eclipse. Le nom du fichier [spring-config.xml] est libre.

#### persistence.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence_1_0.xsd">
4.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL" />
5. </persistence>

```

- ligne 4 : le fichier déclare une unité de persistance appelée *jpa* qui utilise des transactions "locales", c.a.d. non fournies par un conteneur EJB3. Ces transactions sont créées et gérées par Spring et font l'objet de configurations dans le fichier [spring-config.xml].

#### spring-config.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.   xmlns:tx="http://www.springframework.org/schema/tx"
5.   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
6.
7.   <!-- couches applicatives -->
8.   <bean id="dao" class="dao.Dao" />
9.   <bean id="service" class="service.Service">
10.    <property name="dao" ref="dao" />
11.   </bean>
12.
13.   <!-- couche de persistance JPA -->
14.   <bean id="entityManagerFactory"
15.     class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
16.     <property name="dataSource" ref="dataSource" />
17.     <property name="jpaVendorAdapter">
18.       <bean
19.         class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
20.           <!--
21.             <property name="showSql" value="true" />
22.           -->
23.             <property name="databasePlatform"
24.               value="org.hibernate.dialect.MySQL5InnoDBDialect" />
25.             <property name="generateDdl" value="true" />
26.           </bean>
27.         </property>
28.         <property name="loadTimeWeaver">
29.           <bean
30.             class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
31.           </property>
32.         </bean>
33.
34.   <!-- la source de données DBCP -->
35.   <bean id="dataSource"
36.     class="org.apache.commons.dbcp.BasicDataSource"
37.     destroy-method="close">
38.     <property name="driverClassName" value="com.mysql.jdbc.Driver" />
39.     <property name="url" value="jdbc:mysql://localhost:3306/jpa" />
40.     <property name="username" value="jpa" />
41.     <property name="password" value="jpa" />

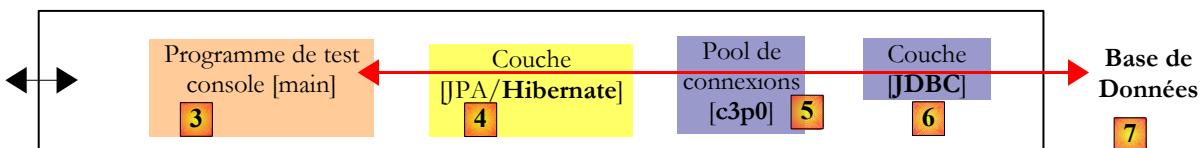
```

```

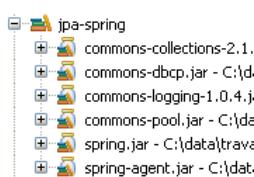
42. </bean>
43.
44. <!-- le gestionnaire de transactions -->
45. <tx:annotation-driven transaction-manager="txManager" />
46. <bean id="txManager"
47.   class="org.springframework.orm.jpa.JpaTransactionManager">
48.   <property name="entityManagerFactory"
49.     ref="entityManagerFactory" />
50. </bean>
51.
52. <!-- traduction des exceptions -->
53. <bean
54.   class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
55.
56. <!-- annotations de persistance -->
57. <bean
58.   class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
59.
60. </beans>

```

- lignes 2-5 : la balise racine `<beans>` du fichier de configuration. Nous ne commentons pas les divers attributs de cette balise. On prendra soin de faire un copier / coller parce que se tromper dans l'un de ces attributs provoque des erreurs parfois difficiles à comprendre.
- ligne 8 : le bean "dao" est une référence sur une instance de la classe [dao.Dao]. Une instance unique sera créée (singleton) et implémentera la couche [dao] de l'application.
- lignes 9-11 : instantiation de la couche [service]. Le bean "service" est une référence sur une instance de la classe [service.Service]. Une instance unique sera créée (singleton) et implémentera la couche [service] de l'application. Nous avons vu que la classe [service.Service] avait un champ privé [IDao dao]. Ce champ est initialisé ligne 10 par le bean "dao" défini ligne 8.
- au final les lignes 8-11 ont configuré les couches [dao] et [service]. Nous verrons plus loin à quel moment et comment elles seront instanciées.
- lignes 35-42 : une source de données est définie. Nous avons déjà rencontré la notion de source de données lors de l'étude des entités JPA avec Hibernate :



Ci-dessus, [c3p0] appelé "pool de connexions" aurait pu être appelé "source de données". Une source de données fournit le service de "pool de connexions". Avec Spring, nous utiliserons une source de données autre que [c3p0]. C'est [DBCP] du projet *Apache commons DBCP* [<http://jakarta.apache.org/commons/dbcp/>]. Les archives de [DBCP] ont été placées dans la bibliothèque utilisateur [jpa-spring] :



- lignes 38-41 : pour créer des connexions avec la base de données cible, la source de données a besoin de connaître le pilote Jdbc utilisé (ligne 38), l'url de la base de données (ligne 39), l'utilisateur de la connexion et son mot de passe (lignes 40-41).
- lignes 14-32 : configurent la couche JPA
- lignes 14-15 : définissent un bean de type [EntityManagerFactory] capable de créer des objets de type [EntityManager] pour gérer les contextes de persistance. La classe instanciée [LocalContainerEntityManagerFactoryBean] est fournie par Spring. Elle a besoin d'un certain nombre de paramètres pour s'instancier, définis lignes 16-31.
- ligne 16 : la source de données à utiliser pour obtenir des connexions au SGBD. C'est la source [DBCP] définie aux lignes 35-42.
- lignes 17-27 : l'implémentation JPA à utiliser
- lignes 18-26 : définissent Hibernate (ligne 19) comme implémentation JPA à utiliser
- lignes 23-24 : le dialecte SQL qu'Hibernate doit utiliser avec le SGBD cible, ici MySQL5.
- ligne 25 : demande qu'au démarrage de l'application, la base de données soit générée (drop et create).
- lignes 28-31 : définissent un "chargeur de classes". Je ne saurai pas expliquer de façon claire le rôle de ce bean utilisé par l'EntityManagerFactory de la couche JPA. Toujours est-il, qu'il implique de passer à la JVM qui exécute l'application, le nom d'une archive dont le contenu va gérer le chargement des classes au démarrage de l'application. Ici, cette archive est

[spring-agent.jar] placée dans la bibliothèque utilisateur [jpa-spring] (voir plus haut). Nous verrons qu'Hibernate n'a pas besoin de cet agent mais que Toplink lui en a besoin.

- lignes 45-50 : définissent le gestionnaire de transactions à utiliser
- ligne 45 : indique que les transactions sont gérées avec des annotations Java (elles auraient pu être également déclarées dans *spring-config.xml*). C'est en particulier l'annotation **@Transactional** rencontrée dans la classe [Service] (ligne 6).
- lignes 46-50 : le gestionnaire de transactions
- ligne 47 : le gestionnaire de transactions est une classe fournie par Spring
- lignes 48-49 : le gestionnaire de transactions de Spring a besoin de connaître l'EntityManagerFactory qui gère la couche JPA. C'est celui défini aux lignes 14-32.
- lignes 57-58 : définissent la classe qui gère les annotations de persistance Spring trouvées dans le code Java, telles l'annotation **@PersistenceContext** de la classe [dao.Dao] (ligne 12).
- lignes 53-54 : définissent la classe Spring qui gère notamment l'annotation **@Repository** qui rend une classe ainsi annotée, éligible pour la traduction des exceptions natives du pilote Jdbc du SGBD en exceptions génériques Spring de type [DataAccessException]. Cette traduction encapsule l'exception Jdbc native dans un type [DataAccessException] ayant diverses sous-classes :

#### Direct Known Subclasses:

```
CleanupFailureDataAccessException, ConcurrencyFailureException,
DataAccessResourceFailureException,
DataIntegrityViolationException, DataRetrievalFailureException,
DataSourceLookupFailureException,
InvalidDataAccessApiUsageException,
InvalidDataAccessResourceUsageException,
PermissionDeniedDataAccessException,
UncategorizedDataAccessException
```

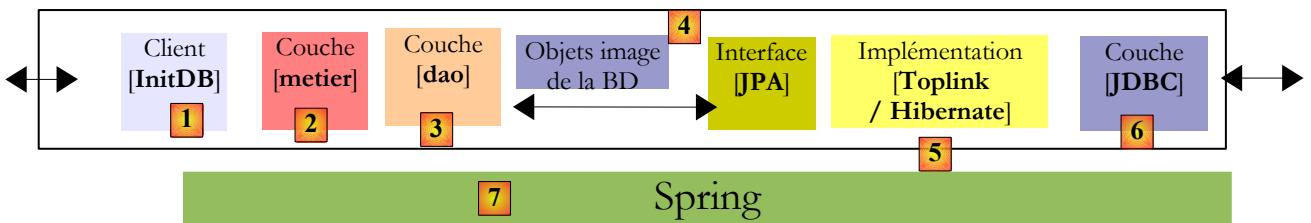
Cette traduction permet au programme client de gérer les exceptions de façon générique quelque soit le SGBD cible. Nous n'avons pas utilisé l'annotation **@Repository** dans notre code Java. Aussi les lignes 53-54 sont-elles inutiles. Nous les avons laissées par simple souci d'information.

Nous en avons fini avec le fichier de configuration de Spring. Il est complexe et bien des choses restent obscures. Il a été tiré de la documentation Spring. Heureusement, son adaptation à diverses situations se résume souvent à deux modifications :

- celle de la base de données cible : lignes 38-41. Nous donnerons un exemple Oracle.
- celle de l'implémentation JPA : lignes 14-32. Nous donnerons un exemple Toplink.

### 3.1.6 Programme client [InitDB]

Nous abordons l'écriture d'un premier client de l'architecture décrite précédemment :



Le code de [InitDB] est le suivant :

```
1. package tests;
2.
3. ...
4. public class InitDB {
5.
6.     // couche service
7.     private static IService service;
8.
9.     // constructeur
10.    public static void main(String[] args) throws ParseException {
11.        // configuration de l'application
12.        ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config.xml");
13.        // couche service
14.        service = (IService) ctx.getBean("service");
15.        // on vide la base
16.        clean();
17.        // on la remplit
```

```

18.     fill();
19.     // on vérifie visuellement
20.     dumpPersonnes();
21. }
22.
23. // affichage contenu table
24. private static void dumpPersonnes() {
25.     System.out.format("[personnes]%n");
26.     for (Personne p : service.getAll()) {
27.         System.out.println(p);
28.     }
29. }
30.
31. // remplissage table
32. public static void fill() throws ParseException {
33.     // création personnes
34.     Personne p1 = new Personne("p1", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
35.         true, 2);
36.     Personne p2 = new Personne("p2", "Sylvie", new
37.         SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
38.     // qu'on sauvegarde
39.     service.saveArray(new Personne[] { p1, p2 });
40. }
41. // suppression éléments de la table
42. public static void clean() {
43.     for (Personne p : service.getAll()) {
44.         service.deleteOne(p.getId());
45.     }
46. }

```

- ligne 12 : le fichier [spring-config.xml] est exploité pour créer un objet [ApplicationContext ctx] qui est une image mémoire du fichier. Les beans définis dans [spring-config.xml] sont instanciés à cette occasion.
- ligne 14 : on demande au contexte d'application **ctx** une référence sur la couche [service]. On sait que celle-ci est représentée par un bean s'appelant "service".
- ligne 16 : la base est vidée au moyen de la méthode *clean* des lignes 41-45 :
  - lignes 42-44 : on demande la liste de toutes les personnes au contexte de persistance et on boucle sur elles pour les supprimer une à une. On se rappelle peut-être que [spring-config.xml] précise que la base de données doit être générée au démarrage de l'application. Aussi dans notre cas, l'appel de la méthode *clean* est inutile puisqu'on part d'une base vide.
- ligne 18 : la méthode *fill* remplit la base. Celle-ci est définie lignes 32-38 :
  - lignes 34-35 : deux personnes sont créées
  - ligne 37 : on demande à la couche [service] de les rendre persistantes.
- ligne 20 : la méthode *dumpPersonnes* affiche les personnes persistantes. Elle est définie aux lignes 24-29
  - lignes 26-28 : on demande la liste de toutes les personnes persistantes à la couche [service] et on les affiche sur la console.

L'exécution de [InitDB] donne le résultat suivant :

```

1. [personnes]
2. [72,0,p1,Paul,31/01/2000,true,2]
3. [73,0,p2,Sylvie,05/07/2001,false,0]

```

### 3.1.7 Tests unitaires [TestNG]

L'installation du plugin [TestNG] est décrite au paragraphe 5.2.4, page 217. Le code du programme [TestNG] est le suivant :

```

1. package tests;
2.
3. ....
4. public class TestNG {
5.
6.     // couche service
7.     private IService service;
8.
9.     @BeforeClass
10.    public void init() {
11.        // log
12.        log("init");
13.        // configuration de l'application
14.        ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config.xml");
15.        // couche service
16.        service = (IService) ctx.getBean("service");
17.    }

```

```

18.
19.    @BeforeMethod
20.    public void setUp() throws ParseException {
21.        // on vide la base
22.        clean();
23.        // on la remplit
24.        fill();
25.    }
26.
27.    // logs
28.    private void log(String message) {
29.        System.out.println("----- " + message);
30.    }
31.
32.    // affichage contenu table
33.    private void dump() {
34.        log("dump");
35.        System.out.format("[personnes]%n");
36.        for (Personne p : service.getAll()) {
37.            System.out.println(p);
38.        }
39.    }
40.
41.    // remplissage table
42.    public void fill() throws ParseException {
43.        log("fill");
44.        // création personnes
45.        Personne p1 = new Personne("p1", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
46.                                    true, 2);
47.        Personne p2 = new Personne("p2", "Sylvie", new
48.                                    SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
49.        // qu'on sauvegarde
50.        service.saveArray(new Personne[] { p1, p2 });
51.    }
52.
53.    // suppression éléments de la table
54.    public void clean() {
55.        log("clean");
56.        for (Personne p : service.getAll()) {
57.            service.deleteOne(p.getId());
58.        }
59.    }
60.
61.    @Test()
62.    public void test01() {
63.    }
64. }
```

- ligne 9 : l'annotation **@BeforeClass** désigne la méthode à exécuter pour initialiser la configuration nécessaire aux tests. Elle est exécutée avant que le premier test ne soit exécuté. L'annotation **@AfterClass** non utilisée ici, désigne la méthode à exécuter une fois que tous les tests ont été exécutés.
- lignes 10-17 : la méthode **init** annotée par **@BeforeClass** exploite le fichier de configuration de Spring pour instancier les différentes couches de l'application et avoir une référence sur la couche [service]. Tous les tests utilisent ensuite cette référence.
- ligne 19 : l'annotation **@BeforeMethod** désigne la méthode à exécuter avant chaque test. L'annotation **@AfterMethod**, non utilisée ici, désigne la méthode à exécuter après chaque test.
- lignes 20-25 : la méthode **setUp** annotée par **@BeforeMethod** vide la base (clean lignes 52-56) puis la remplit avec deux personnes (fill lignes 42-49).
- ligne 59 : l'annotation **@Test** désigne une méthode de test à exécuter. Nous décrivons maintenant ces tests.

```

1. @Test()
2. public void test01() {
3.     log("test1");
4.     dump();
5.     // liste des personnes
6.     List<Personne> personnes = service.getAll();
7.     assert 2 == personnes.size();
8. }
9.
10. @Test()
11. public void test02() {
12.     log("test2");
13.     // recherche de personnes par leur nom
14.     List<Personne> personnes = service.getAllLike("p1%");
15.     assert 1 == personnes.size();
16.     Personne p1 = personnes.get(0);
17.     assert "Paul".equals(p1.getPrenom());
18. }
```

```

20.  @Test()
21.  public void test03() throws ParseException {
22.      log("test3");
23.      // création d'une nouvelle personne
24.      Personne p3 = new Personne("p3", "x", new SimpleDateFormat("dd/MM/yy").parse("05/07/2001"),
25.          false, 0);
26.      // on la persiste
27.      service.saveOne(p3);
28.      // on la redemande
29.      Personne loadedp3 = service.getOne(p3.getId());
30.      // on l'affiche
31.      System.out.println(loadedp3);
32.      // vérification
33.      assert "p3".equals(loadedp3.getNom());
33.  }

```

- lignes 2-8 : le test 01. Il faut se rappeler qu'au départ de chaque test, la base contient deux personnes de noms respectifs **p1** et **p2**.
- ligne 6 : on demande la liste des personnes
- lignes 7 : on vérifie que le nombre de personnes de la liste obtenue est 2
- ligne 14 : on demande la liste des personnes ayant un nom commençant par **p1**
- on vérifie que la liste obtenue n'a qu'un élément (ligne 15) et que le prénom de l'unique personne obtenue est "Paul" (ligne 17)
- ligne 24 : on crée une personne nommée **p3**
- ligne 25 : on la persiste
- ligne 28 : on la redemande au contexte de persistance pour vérification
- ligne 32 : on vérifie que la personne obtenue a bien le nom **p3**.

```

1. @Test()
2. public void test04() throws ParseException {
3.     log("test4");
4.     // on charge la personne p1
5.     List<Personne> personnes = service.getAllLike("p1%");
6.     Personne p1 = personnes.get(0);
7.     // on l'affiche
8.     System.out.println(p1);
9.     // on vérifie
10.    assert "p1".equals(p1.getNom());
11.    int version1 = p1.getVersion();
12.    // on modifie le prénom
13.    p1.setPrenom("x");
14.    // on sauvegarde
15.    service.updateOne(p1);
16.    // on recharge
17.    p1 = service.getOne(p1.getId());
18.    // on l'affiche
19.    System.out.println(p1);
20.    // on vérifie que la version a été incrémentée
21.    assert (version1 + 1) == p1.getVersion();
22.
23. }

```

- ligne 5 : on demande la personne **p1**
- ligne 10 : on vérifie son nom
- ligne 11 : on note son n° de version
- ligne 13 : on modifie son prénom
- ligne 15 : on sauvegarde la modification
- ligne 17 : on redemande la personne **p1**
- ligne 21 : on vérifie que son n° de version a augmenté de 1

```

1. @Test()
2. public void test05() {
3.     log("test5");
4.     // on charge la personne p2
5.     List<Personne> personnes = service.getAllLike("p2%");
6.     Personne p2 = personnes.get(0);
7.     // on l'affiche
8.     System.out.println(p2);
9.     // on vérifie
10.    assert "p2".equals(p2.getNom());
11.    // on supprime la personne p2
12.    service.deleteOne(p2.getId());
13.    // on la recharge
14.    p2 = service.getOne(p2.getId());
15.    // on vérifie qu'on a obtenu un pointeur null
16.    assert null == p2;

```

```

17.     // on affiche la table
18.     dump();
19. }

```

- ligne 5 : on demande la personne **p2**
- ligne 10 : on vérifie son nom
- ligne 12 : on la supprime
- ligne 14 : on la redemande
- ligne 16 : on vérifie qu'on ne l'a pas trouvée

```

1. @Test()
2. public void test06() throws ParseException {
3.     log("test6");
4.     // on crée un tableau de 2 personnes de même nom (enfreint la règle d'unicité du nom)
5.     Personne[] personnes = { new Personne("p3", "x", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"), true, 2),
6.         new Personne("p4", "x", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"), true, 2),
7.         new Personne("p4", "x", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"), true, 2) };
8.     // on sauvegarde ce tableau - on doit obtenir une exception et un rollback
9.     boolean erreur = false;
10.    try {
11.        service.saveArray(personnes);
12.    } catch (RuntimeException e) {
13.        erreur = true;
14.    }
15.    // dump
16.    dump();
17.    // vérifications
18.    assert erreur;
19.    // recherche personne de nom p3
20.    List<Personne> personnesp3 = service.getAllLike("p3%");
21.    assert 0 == personnesp3.size();
22.    // dump
23.    dump();
24. }

```

- ligne 5 : on crée un tableau de trois personnes dont deux ont le même nom "p4". Cela enfreint la règle d'unicité du nom de l'`@Entity Personne`:

```

1. @Column(name = "NOM", length = 30, nullable = false, unique = true)
2. private String nom;

```

- ligne 11 : le tableau des trois personnes est mis dans le contexte de persistance. L'ajout de la seconde personne **p4** devrait échouer. Comme la méthode [saveArray] se déroule dans une transaction, toutes les insertions qui ont pu être faites avant seront annulées. Au final, aucun ajout ne sera fait.
- ligne 18 : on vérifie que [saveArray] a bien lancé une exception
- lignes 20-21 : on vérifie que la personne **p3** qui aurait pu être ajoutée ne l'a pas été.

```

1. @Test()
2. public void test07() {
3.     log("test7");
4.     // test optimistic locking
5.     // on charge la personne p1
6.     List<Personne> personnes = service.getAllLike("p1%");
7.     Personne p1 = personnes.get(0);
8.     // on l'affiche
9.     System.out.println(p1);
10.    // on augmente son nbre d'enfants
11.    int nbEnfants1 = p1.getNbEnfants();
12.    p1.setNbEnfants(nbEnfants1 + 1);
13.    // on sauvegarde p1
14.    Personne newp1 = service.updateOne(p1);
15.    assert (nbEnfants1 + 1) == newp1.getNbEnfants();
16.    System.out.println(newp1);
17.    // on sauvegarde une deuxième fois - on doit avoir une exception car p1 n'a plus la bonne version
18.    // c'est newp1 qui l'a
19.    boolean erreur = false;
20.    try {
21.        service.updateOne(p1);
22.    } catch (RuntimeException e) {
23.        erreur = true;
24.    }
25.    // vérification
26.    assert erreur;
27.    // on augmente le nbre d'enfants de newp1
28.    int nbEnfants2 = newp1.getNbEnfants();
29.    newp1.setNbEnfants(nbEnfants2 + 1);
30.    // on sauvegarde newp1
31.    service.updateOne(newp1);

```

```

32.     // on recharge
33.     p1 = service.getOne(p1.getId());
34.     // on vérifie
35.     assert (nbEnfants1 + 2) == p1.getNbenfants();
36.     System.out.println(p1);
37. }

```

- ligne 6 : on demande la personne **p1**
- ligne 12 : on augmente de 1 son nombre d'enfants
- ligne 14 : on met à jour la personne **p1** dans le contexte de persistance. La méthode [updateOne] rend la nouvelle version **newp1** persistante de **p1**. Elle diffère de **p1** par son n° de version qui a du être incrémenté.
- ligne 15 : on vérifie le nombre d'enfants de **newp1**.
- ligne 21 : on redemande une mise à jour de la personne **p1** à partir de l'ancienne version **p1**. On doit avoir une exception car **p1** n'est pas la dernière version de la personne **p1**. Cette dernière version est **newp1**.
- ligne 23 : on vérifie que l'erreur a bien eu lieu
- lignes 27-35 : on vérifie que si une mise à jour est faite à partir de la dernière version **newp1**, alors les choses se passent bien.

```

1. @Test()
2. public void test08() {
3.     log("test8");
4.     // test rollback sur updateArray
5.     // on charge la personne p1
6.     List<Personne> personnes = service.getAllLike("p1%");
7.     Personne p1 = personnes.get(0);
8.     // on l'affiche
9.     System.out.println(p1);
10.    // on augmente son nbre d'enfants
11.    int nbEnfants1 = p1.getNbenfants();
12.    p1.setNbenfants(nbEnfants1 + 1);
13.    // on sauvegarde 2 modifications dont la 2ième doit échouer (personne mal initialisée)
14.    // à cause de la transaction, les deux doivent alors être annulées
15.    boolean erreur = false;
16.    try {
17.        service.updateArray(new Personne[] { p1, new Personne() });
18.    } catch (RuntimeException e) {
19.        erreur = true;
20.    }
21.    // vérifications
22.    assert erreur;
23.    // on recharge la personne p1
24.    personnes = service.getAllLike("p1%");
25.    p1 = personnes.get(0);
26.    // son nbre d'enfants n'a pas du changer
27.    assert nbEnfants1 == p1.getNbenfants();
28. }

```

- le test 8 est similaire au test 6 : il vérifie le *rollback* sur un **updateArray** opérant sur un tableau de deux personnes où la deuxième n'a pas été initialisée correctement. D'un point de vue JPA, l'opération **merge** sur la seconde personne qui n'existe pas déjà va générer un ordre SQL *insert* qui va échouer à cause des contraintes *nullable=false* qui existe sur certains des champs de l'entité *Personne*.

```

1. @Test()
2. public void test09() {
3.     log("test9");
4.     // test rollback sur deleteArray
5.     // dump
6.     dump();
7.     // on charge la personne p1
8.     List<Personne> personnes = service.getAllLike("p1%");
9.     Personne p1 = personnes.get(0);
10.    // on l'affiche
11.    System.out.println(p1);
12.    // on fait 2 suppressions dont la 2ième doit échouer (personne inconnue)
13.    // à cause de la transaction, les deux doivent alors être annulées
14.    boolean erreur = false;
15.    try {
16.        service.deleteArray(new Personne[] { p1, new Personne() });
17.    } catch (RuntimeException e) {
18.        erreur = true;
19.    }
20.    // vérifications
21.    assert erreur;
22.    // on recharge la personne p1
23.    personnes = service.getAllLike("p1%");
24.    // vérification
25.    assert 1 == personnes.size();
26.    // dump
27.    dump();

```

- le test 9 est similaire au précédent : il vérifie le *rollback* sur un **deleteArray** opérant sur un tableau de deux personnes où la deuxième n'existe pas. Or dans ce cas, la méthode [deleteOne] de la couche [dao] lance une exception.

```

1.// optimistic locking - accès multi-threads
2.@Test()
3.public void test10() throws Exception {
4.    // ajout d'une personne
5.    Personne p3 = new Personne("X", "X", new SimpleDateFormat("dd/MM/yyyy").parse("01/02/2006"), true,
6.    0);
7.    service.saveOne(p3);
8.    int id3 = p3.getId();
9.    // création de N threads de mise à jour du nombre d'enfants
10.   final int N = 20;
11.   Thread[] taches = new Thread[N];
12.   for (int i = 0; i < taches.length; i++) {
13.       taches[i] = new ThreadMajEnfants("thread n° " + i, service, id3);
14.       taches[i].start();
15.   }
16.   // on attend la fin des threads
17.   for (int i = 0; i < taches.length; i++) {
18.       taches[i].join();
19.   }
20.   // on récupère la personne
21.   p3 = service.getOne(id3);
22.   // elle doit avoir N enfants
23.   assertEquals(N == p3.getNbenfants());
24.   // suppression personne p3
25.   service.deleteOne(p3.getId());
26.   // vérification
27.   p3 = service.getOne(p3.getId());
28.   // on doit avoir un pointeur null
29.   assertEquals(p3 == null);
29. }
```

- l'idée du test 10 est de lancer N threads (ligne 9) pour incrémenter en parallèle le nombre d'enfants d'une personne. On veut vérifier que le système du n° de version résiste bien à ce cas de figure. Il a été créé pour cela.
- lignes 5-6 : une personne nommée **p3** est créée puis persistée. Elle a 0 enfant au départ.
- ligne 7 : on note son identifiant.
- lignes 9-14 : on lance N threads en parallèle, tous chargés d'incrémenter de 1 le nombre d'enfants de **p3**.
- lignes 16-18 : on attend la fin de tous les threads
- ligne 20 : on demande à voir la personne **p3**
- ligne 22 : on vérifie qu'elle a maintenant N enfants
- ligne 24 : la personne **p3** est supprimée.

Le thread [ThreadMajEnfants] est le suivant :

```

1. package tests;
2.
3. ...
4. public class ThreadMajEnfants extends Thread {
5.     // nom du thread
6.     private String name;
7.
8.     // référence sur la couche [service]
9.     private IService service;
10.
11.    // l'id de la personne sur qui on va travailler
12.    private int idPersonne;
13.
14.    // constructeur
15.    public ThreadMajEnfants(String name, IService service, int idPersonne) {
16.        this.name = name;
17.        this.service = service;
18.        this.idPersonne = idPersonne;
19.    }
20.
21.    // coeur du thread
22.    public void run() {
23.        // suivi
24.        suivi("lancé");
25.        // on boucle tant qu'on n'a pas réussi à incrémenter de 1
26.        // le nbre d'enfants de la personne idPersonne
27.        boolean fini = false;
28.        int nbEnfants = 0;
29.        while (!fini) {
30.            // on récupère une copie de la personne d'idPersonne
31.            Personne personne = service.getOne(idPersonne);
```

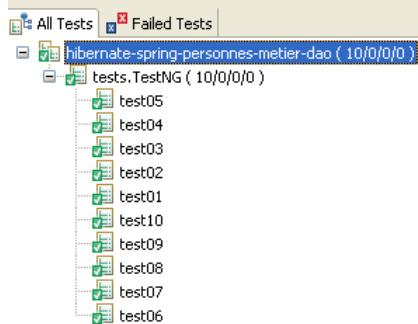
```

32.     nbEnfants = personne.getNbEnfants();
33.     // suivi
34.     suivi("'" + nbEnfants + " -> " + (nbEnfants + 1) + " pour la version " +
personne.getVersion());
35.     // incrémenté de 1 le nbre d'enfants de la personne
36.     personne.setNbEnfants(nbEnfants + 1);
37.     // attente de 10 ms pour abandonner le processeur
38.     try {
39.         // suivi
40.         suivi("début attente");
41.         // on s'interrompt pour laisser le processeur
42.         Thread.sleep(10);
43.         // suivi
44.         suivi("fin attente");
45.     } catch (Exception ex) {
46.         throw new RuntimeException(ex.toString());
47.     }
48.     // attente terminée - on essaie de valider la copie
49.     // entre-temps d'autres threads ont pu modifier l'original
50.     try {
51.         // on essaie de modifier l'original
52.         service.updateOne(personne);
53.         // on est passé - l'original a été modifié
54.         fini = true;
55.     } catch (javax.persistence.OptimisticLockException e) {
56.         // version de l'objet incorrecte : on ignore l'exception pour recommencer
57.     } catch (org.springframework.transaction.UnexpectedRollbackException e2) {
58.         // exception Spring qui surgit de temps en temps
59.     } catch (RuntimeException e3) {
60.         // autre type d'exception - on la remonte
61.         throw e3;
62.     }
63. }
64. // suivi
65. suivi("a terminé et passé le nombre d'enfants à " + (nbEnfants + 1));
66. }
67.
68. // suivi
69. private void suivi(String message) {
70.     System.out.println(name + " [" + new Date().getTime() + "] : " + message);
71. }
72. }

```

- lignes 15-19 : le constructeur mémorise les informations dont il a besoin pour travailler : son nom (ligne 16), la référence sur la couche [service] qu'il doit utiliser (ligne 17) et l'identifiant de la personne **p** dont il doit incrémenter le nombre d'enfants (ligne 18).
- lignes 22-66 : la méthode [run] exécutée par tous les threads en parallèle.
- ligne 29 : le thread essaie de façon répétée d'incrémenter le nombre d'enfants de la personne **p**. Il ne s'arrête que lorsqu'il a réussi.
- ligne 31 : la personne **p** est demandée
- ligne 36 : son nombre d'enfants est incrémenté en mémoire
- lignes 38-47 : on fait une pause de 10 ms. Cela va permettre à d'autres threads d'obtenir la même version de la personne **p**. On aura donc au même moment plusieurs threads détenant la même version de la personne **p** et voulant la modifier. C'est ce qui est désiré.
- ligne 52 : une fois la pause terminée, le thread demande à la couche [service] de persister la modification. On sait qu'il y aura de temps en temps des exceptions, aussi a-t-on entouré l'opération d'un try / catch.
- ligne 55 : les tests montrent qu'on a des exceptions de type [javax.persistence.OptimisticLockException]. C'est normal : c'est l'exception lancée par la couche JPA lorsqu'un thread veut modifier la personne **p** sans avoir la dernière version de celle-ci. Cette exception est ignorée pour laisser le thread tenter de nouveau l'opération jusqu'à ce qu'il y arrive.
- ligne 57 : les tests montrent qu'on a également des exceptions de type [org.springframework.transaction.UnexpectedRollbackException]. C'est ennuyeux et inattendu. Je n'ai pas d'explications à donner. Nous voilà dépendants de Spring alors qu'on aurait voulu éviter cela. Cela signifie que si on exécute notre application dans JBoss Ejb3 par exemple, le code du thread devra être changé. L'exception Spring est ici aussi ignorée pour laisser le thread tenter de nouveau l'opération d'incrémentation.
- ligne 59 : les autres types d'exception sont remontés à l'application.

Lorsque [TestNG] est exécuté on obtient les résultats suivants :



Les 10 tests ont été passés avec succès.

Le test 10 mérite des explications complémentaires parce que le fait qu'il ait réussi a un côté magique. Revenons tout d'abord sur la configuration de la couche [dao] :

```

1. public class Dao implements IDao {
2.
3.     @PersistenceContext
4.     private EntityManager em;
5.

```

- ligne 4 : un objet [EntityManager] est injecté dans le champ **em** grâce à l'annotation JPA **@PersistenceContext**. La couche [dao] est instanciée une unique fois. C'est un singleton utilisé par tous les threads utilisant la couche JPA. Ainsi donc l'EntityManager **em** est-il commun à tous les threads. On peut le vérifier en affichant la valeur de **em** dans la méthode [updateOne] utilisée par les threads [ThreadMajEnfants] : on a la même valeur pour tous les threads.

Du coup, on peut se demander si les objets persistants des différents threads manipulés par l'EntityManager **em** qui est le même pour tous les threads, ne vont pas se mélanger et créer des conflits entre-eux. Un exemple de ce qui pourrait se passer se trouve dans [ThreadMajEnfants] :

```

1.     while (!fini) {
2.         // on récupère une copie de la personne d'idPersonne
3.         Personne personne = service.getOne(idPersonne);
4.         nbEnfants = personne.getNbEnfants();
5.         // suivi
6.         suivi(" " + nbEnfants + " -> " + (nbEnfants + 1) + " pour la version " +
personne.getVersion());
7.         // incrémente de 1 le nbre d'enfants de la personne
8.         personne.setNbEnfants(nbEnfants + 1);
9.         // attente de 10 ms pour abandonner le processeur
10.        try {
11.            // suivi
12.            suivi("début attente");
13.            // on s'interrompt pour laisser le processeur
14.            Thread.sleep(10);
15.            // suivi
16.            suivi("fin attente");
17.        } catch (Exception ex) {
18.            throw new RuntimeException(ex.toString());
19.        }

```

- ligne 3 : un thread T1 récupère la personne **p**
- ligne 8 : elle incrémente le nombre d'enfants de **p**
- ligne 14 : le thread T1 fait une pause

Un thread T2 prend la main et exécute lui aussi la ligne 3 : il demande la même personne **p** que T1. Si le contexte de persistance des threads était le même, la personne **p** étant déjà dans le contexte grâce à T1 devrait être rendue à T2. En effet, la méthode [getOne] utilise la méthode [EntityManager].find de l'API JPA et cette méthode ne fait un accès à la base que si l'objet demandé ne fait pas partie du contexte de persistance, sinon elle rend l'objet du contexte de persistance. Si tel était le cas, T1 et T2 détiendraient la même personne **p**. T2 incrémenterait alors le nombre d'enfants de **p** de 1 de nouveau (ligne 8). Si l'un des threads réussit sa mise à jour après la pause, alors le nombre d'enfants de **p** aura été augmenté de 2 et non de 1 comme prévu. On pourrait alors s'attendre à ce que les N threads passent le nombre d'enfants non pas à N mais à davantage. Or ce n'est pas le cas. On peut alors conclure que T1 et T2 n'ont pas la même référence **p**. On le vérifie en faisant afficher l'adresse de **p** par les threads : elle est différente pour chacun d'eux.

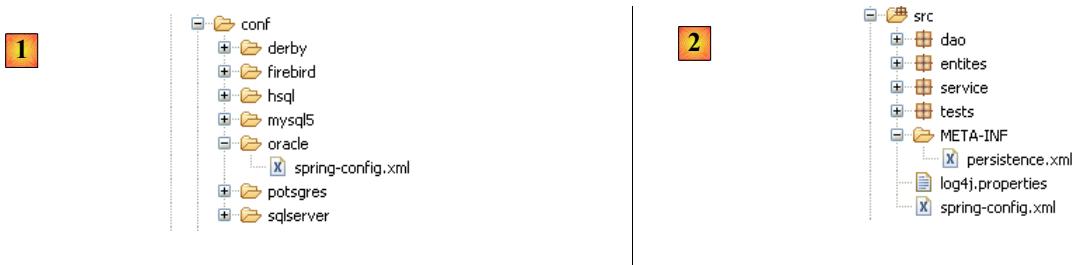
Il semblerait donc que les threads :

- partagent le même gestionnaire de contexte de persistance (EntityManager)

- mais ont chacun un contexte de persistance qui leur est propre.

Ce ne sont que des suppositions et l'avis d'un expert serait utile ici.

### 3.1.8 Changer de SGBD



Pour changer de SGBD, il suffit de remplacer le fichier [src/spring-config.xml] [2] par le fichier [spring-config.xml] du SGBD concerné du dossier [conf] [1].

Le fichier [spring-config.xml] d'Oracle est, par exemple, le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xmlns:tx="http://www.springframework.org/schema/tx"
4.   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-
2.0.xsd">
5.
6. ...
7.   <bean id="entityManagerFactory"
       class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
8.     <property name="dataSource" ref="dataSource" />
9.     <property name="jpaVendorAdapter">
10.      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
11.        <!--
12.          <property name="showSql" value="true" />
13.        -->
14.        <property name="databasePlatform" value="org.hibernate.dialect.OracleDialect" />
15.        <property name="generateDdl" value="true" />
16.      </bean>
17.    </property>
18.    <property name="loadTimeWeaver">
19.      <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
20.    </property>
21.  </bean>
22.
23.  <!-- la source de données DBCP -->
24.  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
25.    <property name="driverClassName" value="oracle.jdbc.OracleDriver" />
26.    <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
27.    <property name="username" value="jpa" />
28.    <property name="password" value="jpa" />
29.  </bean>
30. ...
31. </beans>
```

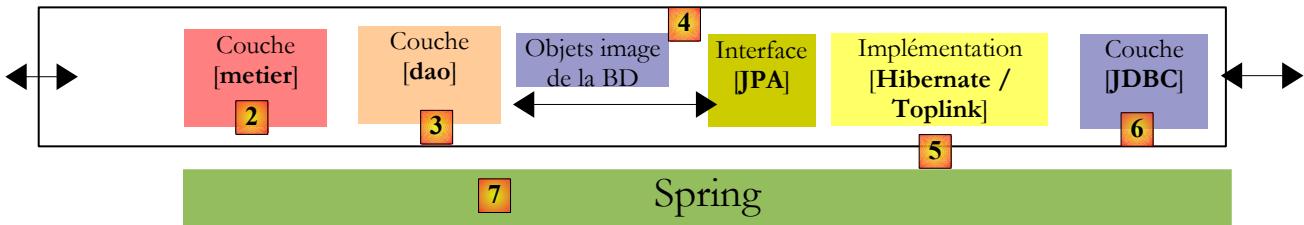
Seules certaines lignes changent vis à vis du même fichier utilisé précédemment pour MySQL5 :

- ligne 14 : le dialecte SQL qu'Hibernate doit utiliser
- lignes 25-28 : les caractéristiques de la connexion Jdbc avec le SGBD

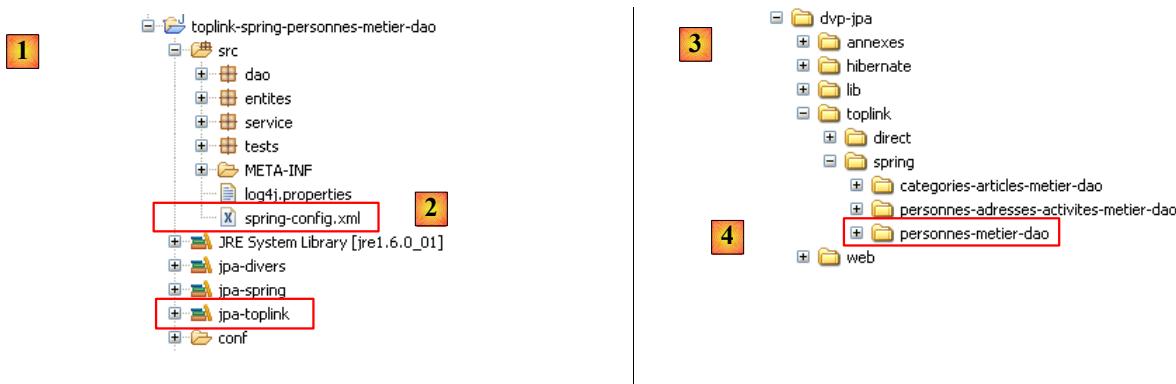
Le lecteur est invité à répéter les tests décrits pour MySQL5 avec d'autres SGBD.

### 3.1.9 Changer d'implémentation JPA

Revenons à l'architecture des tests précédents :



Nous remplaçons l'implémentation JPA / Hibernate par une implémentation JPA / Toplink. Toplink n'utilisant pas les mêmes bibliothèques qu'Hibernate, nous utilisons un nouveau projet Eclipse :



- en [1] : le projet Eclipse. Il est identique au précédent. Seuls changent le fichier de configuration [spring-config.xml] [2] et la bibliothèque [jpa-toplink] qui remplace la bibliothèque [jpa-hibernate].
- en [3] : le dossier des exemples de ce tutoriel. En [4] le projet Eclipse à importer.

Le fichier de configuration [spring-config.xml] pour Toplink devient le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2.
3. <!-- la JVM doit être lancée avec l'argument -javaagent:C:\data\2006-2007\eclipse\dvp-
   jpa\lib\spring\spring-agent.jar
4.   (à remplacer par le chemin exact de spring-agent.jar)-->
5.
6. <beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7.   xmlns:tx="http://www.springframework.org/schema/tx"
8.   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-
   2.0.xsd">
9.
10. <!-- couches applicatives -->
11. <bean id="dao" class="dao.Dao" />
12. <bean id="service" class="service.Service">
13.   <property name="dao" ref="dao" />
14. </bean>
15.
16. <bean id="entityManagerFactory"
   class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
17.   <property name="dataSource" ref="dataSource" />
18.   <property name="jpaVendorAdapter">
19.     <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter">
20.       <!--
21.         <property name="showSql" value="true" />
22.       -->
23.       <property name="databasePlatform"
24.         value="oracle.toplink.essential.platform.database.MySQL4Platform" />
25.         <property name="generateDdl" value="true" />
26.       </bean>
27.     </property>
28.     <property name="loadTimeWeaver">
29.       <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
30.     </property>
31.   </bean>
32. <!-- la source de données DBCP -->
33. <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
34.   <property name="driverClassName" value="com.mysql.jdbc.Driver" />
35.   <property name="url" value="jdbc:mysql://localhost:3306/jpa" />
```

```

36.      <property name="username" value="jpa" />
37.      <property name="password" value="jpa" />
38.    </bean>
39.
40.    <!-- le gestionnaire de transactions -->
41.    <tx:annotation-driven transaction-manager="txManager" />
42.    <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
43.      <property name="entityManagerFactory" ref="entityManagerFactory" />
44.    </bean>
45.
46.    <!-- traduction des exceptions -->
47.    <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
48.
49.    <!-- persistance -->
50.    <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
51.
52.  </beans>

```

Peu de lignes doivent être changées pour passer d'Hibernate à Toplink :

- ligne 19 : l'implémentation JPA est désormais faite par Toplink
- ligne 23 : la propriété [databasePlatform] a une autre valeur qu'avec Hibernate : le nom d'une classe propre à Toplink. Où trouver ce nom a été expliqué au paragraphe 2.1.15.2, page 58.

C'est tout. On notera la facilité avec laquelle on peut changer de SGBD ou d'implémentation JPA avec Spring.

On n'a quand même pas tout *a* fait fini. Lorsqu'on exécute [InitDB] par exemple, on a une exception pas simple à comprendre :

```

1. Exception in thread "main" org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'entityManagerFactory' defined in class path resource [spring-config.xml]:
Invocation of init method failed; nested exception is java.lang.IllegalStateException: Must start with Java agent to use InstrumentationLoadTimeWeaver. See Spring documentation.
2. Caused by: java.lang.IllegalStateException: Must start with Java agent to use
3.

```

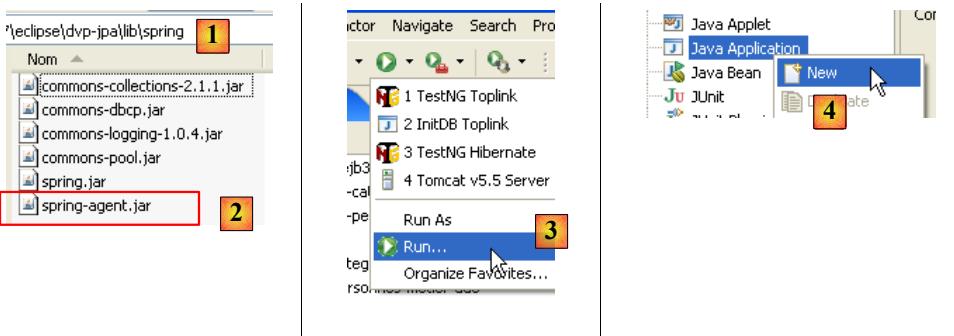
Le message d'erreur de la ligne 1 incite à lire la documentation Spring. On y découvre alors un peu plus le rôle joué par une déclaration obscure du fichier [spring-config.xml] :

```

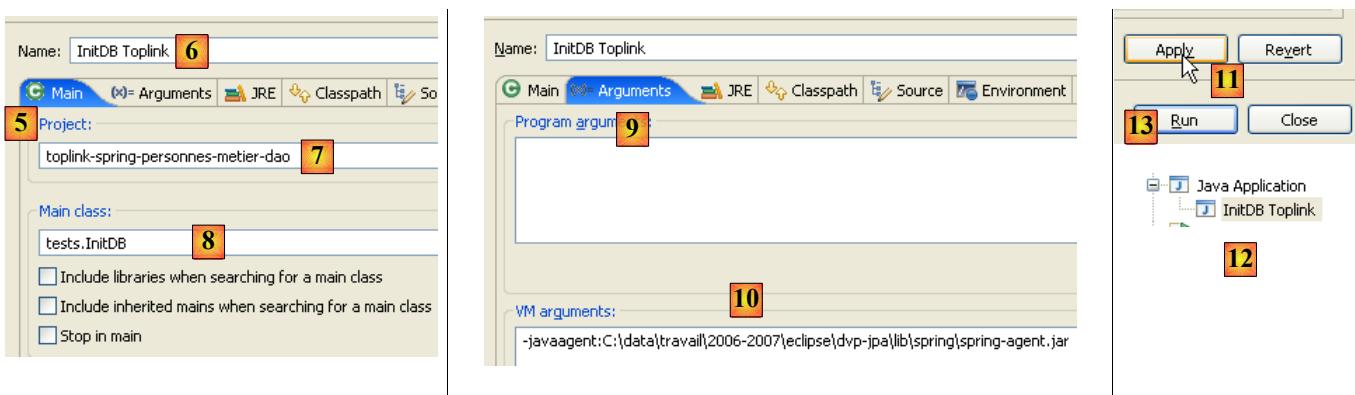
1. <bean id="entityManagerFactory"
2.   class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
3.     <property name="dataSource" ref="dataSource" />
4.     <property name="jpaVendorAdapter">
5.       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
6.         <!--
7.           <property name="showSql" value="true" />
8.         -->
9.         <property name="databasePlatform" value="org.hibernate.dialect.OracleDialect" />
10.        <property name="generateDdl" value="true" />
11.      </bean>
12.    </property>
13.    <property name="loadTimeWeaver">
14.      <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
15.    </property>
</bean>

```

La ligne 1 de l'exception fait référence à une classe nommée [InstrumentationLoadTimeWeaver], classe que l'on retrouve ligne 13 du fichier de configuration Spring. La documentation Spring explique que cette classe est nécessaire dans certains cas pour charger les classes de l'application et que pour qu'elle soit exploitée, la JVM doit être lancée avec un agent. Cet agent est fourni par Spring et s'appelle [spring-agent] :

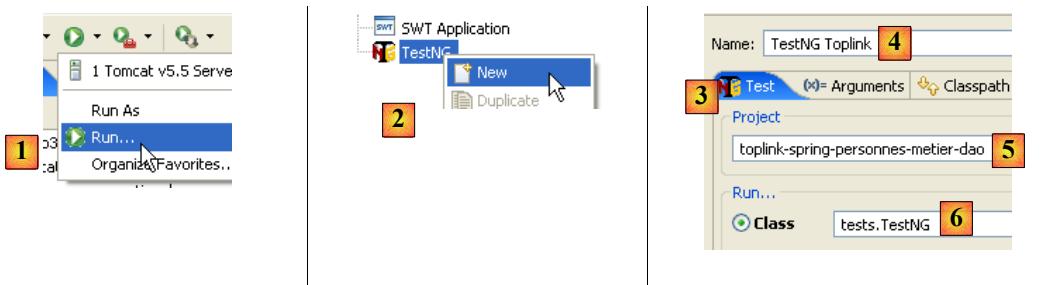


- le fichier [spring-agent.jar] est dans le dossier <exemples>/lib [1]. Il est fourni avec la distribution Spring 2.x (cf paragraphe 5.11, page 299).
- en [3], on crée une configuration d'exécution [Run/Run...]
- en [4], on crée une configuration d'exécution Java (il y a diverses sortes de configurations d'exécution)



- en [5], on choisit l'onglet [Main]
- en [6], on donne un nom à la configuration
- en [7], on nomme le projet Eclipse concerné par cette configuration (utiliser le bouton Browse)
- en [8], on nomme la classe Java qui contient la méthode [main] (utiliser le bouton Browse)
- en [9], on passe dans l'onglet [Arguments]. Dans celui-ci on peut préciser deux types d'arguments :
  - en [9], ceux passés à la méthode [main]
  - en [10], ceux passés à la JVM qui va exécuter le code. L'agent Spring est défini à l'aide du paramètre **-javaagent: valeur** de la JVM. La valeur est le chemin du fichier [spring-agent.jar].
- en [11] : on valide la configuration
- en [12] : la configuration est créée
- en [13] : on l'exécute

Ceci fait, [InitDB] s'exécute et donne les mêmes résultats qu'avec Hibernate. Pour [TestNG], il faut procéder de la même façon :



- en [1], on crée une configuration d'exécution [Run/Run...]
- en [2], on crée une configuration d'exécution **TestNG**
- en [3], on choisit l'onglet [Test]
- en [4], on donne un nom à la configuration
- en [5], on nomme le projet Eclipse concerné par cette configuration (utiliser le bouton Browse)
- en [6], on nomme la classe des tests (utiliser le bouton Browse)

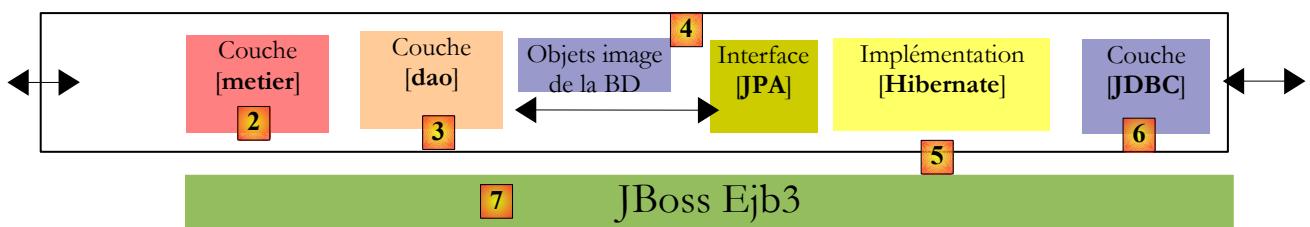


- en [7], on passe dans l'onglet [Arguments].
- en [8] : on fixe l'argument **-javaagent** de la JVM.
- en [9] : on valide la configuration
- en [10] : la configuration est créée
- en [11] : on l'exécute

Ceci fait, [TestNG] s'exécute et donne les mêmes résultats qu'avec Hibernate.

## 3.2 Exemple 2 : JBoss EJB3 / JPA avec entité Personne

Nous reprenons le même exemple que précédemment, mais nous l'exécutons dans un conteneur EJB3, celui de JBoss :

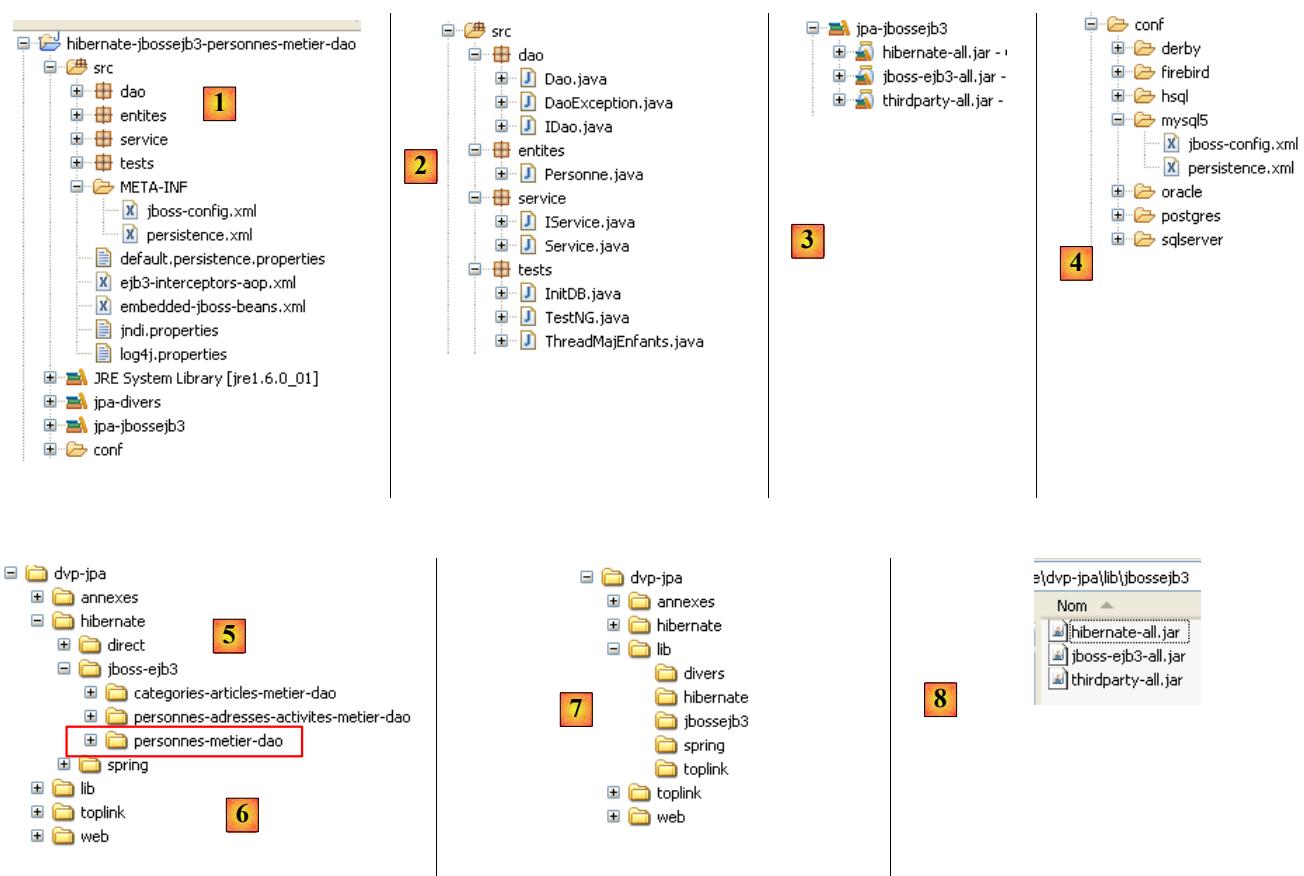


Un conteneur Ejb3 est normalement intégré à un serveur d'application. JBoss délivre un conteneur Ejb3 "standalone" utilisable hors d'un serveur d'application. Nous allons découvrir qu'il délivre des services analogues à ceux délivrés par Spring. Nous essaierons de voir lequel de ces conteneurs se montre le plus pratique.

L'installation du conteneur JBoss Ejb3 est décrite au paragraphe 5.12, page 301.

### 3.2.1 Le projet Eclipse / Jboss Ejb3 / Hibernate

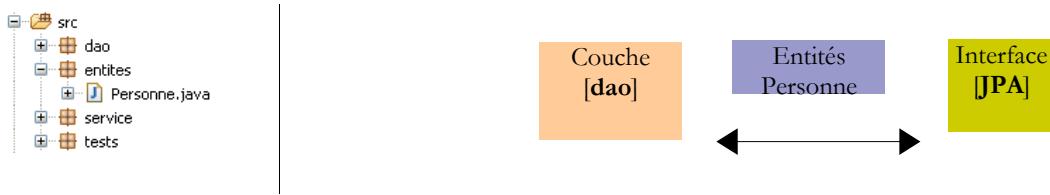
Le projet Eclipse est le suivant :



- en [1] : le projet Eclipse. Il sera trouvé en [6] dans les exemples du tutoriel [5]. On l'importera.
- en [2] : les codes Java des couches présentés en paquetages :
  - [entites] : le paquetage des entités JPA
  - [dao] : la couche d'accès aux données - s'appuie sur la couche JPA
  - [service] : une couche de services plus que de métier. On y utilisera le service de transactions du conteneur Ejb3.
  - [tests] : regroupe les programmes de tests.

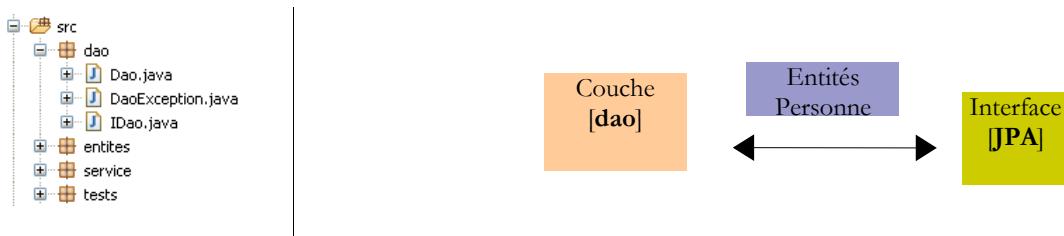
- en [3] : la bibliothèque [jpa-jbossejb3] regroupe les jars nécessaires à Jboss Ejb3 (voir aussi [7] et [8]).
- en [4] : le dossier [conf] rassemble les fichiers de configuration pour chacun des SGBD utilisés dans ce tutoriel. Il y en a à chaque fois deux : [persistence.xml] qui configure la couche JPA et [jboss-config.xml] qui configure le conteneur Ejb3.

### 3.2.2 Les entités JPA



Il n'y a qu'une entité gérée ici, l'entité *Personne* étudiée précédemment au paragraphe 3.1.2, page 146.

### 3.2.3 La couche [dao]



La couche [dao] présente l'interface [IDao] décrite précédemment au paragraphe 3.1.3, page 146.

L'implémentation [Dao] de cette interface est la suivante :

```

1. package dao;
2.
3. ...
4. @Stateless
5. public class Dao implements IDao {
6.
7.     @PersistenceContext
8.     private EntityManager em;
9.
10.    // supprimer une personne via son identifiant
11.    @TransactionAttribute(TransactionAttributeType.REQUIRED)
12.    public void deleteOne(Integer id) {
13.        Personne personne = em.find(Personne.class, id);
14.        if (personne == null) {
15.            throw new DaoException(2);
16.        }
17.        em.remove(personne);
18.    }
19.
20.    // obtenir toutes les personnes
21.    @TransactionAttribute(TransactionAttributeType.REQUIRED)
22.    public List<Personne> getAll() {
23.        return em.createQuery("select p from Personne p").getResultList();
24.    }
25.
26.    // obtenir les personnes dont le nom correspond à un modèle
27.    @TransactionAttribute(TransactionAttributeType.REQUIRED)
28.    public List<Personne> getAllLike(String modele) {
29.        return em.createQuery("select p from Personne p where p.nom like :modele")
30.            .setParameter("modele", modele).getResultList();
31.    }
32.
33.    // obtenir une personne via son identifiant
34.    @TransactionAttribute(TransactionAttributeType.REQUIRED)
35.    public Personne getOne(Integer id) {
36.        return em.find(Personne.class, id);
37.    }
38.
39.    // sauvegarder une personne
40.    @TransactionAttribute(TransactionAttributeType.REQUIRED)
41.    public Personne saveOne(Personne personne) {
42.        em.persist(personne);

```

```

43.     return personne;
44. }
45.
46. // mettre à jour une personne
47. @TransactionAttribute(TransactionAttributeType.REQUIRED)
48. public Personne updateOne(Personne personne) {
49.     return em.merge(personne);
50. }
51.
52. }

```

- ce code est en tout point identique à celui qu'on avait avec Spring. Seules les annotations Java changent et c'est cela que nous commentons.
- ligne 4 : l'annotation **@Stateless** fait de la classe [Dao] un **Ejb sans état**. L'annotation **@Stateful** fait d'une classe un Ejb **avec état**. Un Ejb avec état a des champs privés dont il faut conserver la valeur au fil du temps. Un exemple classique est celui d'une classe qui contient des informations liées à l'utilisateur web d'une application. Une instance de cette classe est liée à un utilisateur précis et lorsque le thread d'exécution d'une requête de cet utilisateur est terminé, l'instance doit être conservée pour être disponible lors de la prochaine requête du même client. Un Ejb **@Stateless** n'a pas d'état. Si on reprend le même exemple, à la fin du thread d'exécution d'une requête d'un utilisateur, l'Ejb **@Stateless** va rejoindre un pool d'Ejb **@Stateless** et devient disponible pour le thread d'exécution d'une requête d'un autre utilisateur.
- pour le développeur, la notion d'Ejb3 **@Stateless** est proche de celle du **singleton** de Spring. Il l'utilisera dans les mêmes cas.
- ligne 7 : l'annotation **@PersistenceContext** est la même que celle rencontrée dans la version Spring de la couche [dao]. Elle désigne le champ qui va recevoir l'EntityManager qui va permettre à la couche [dao] de manipuler le contexte de persistance.
- ligne 11 : l'annotation **@TransactionAttribute** appliquée à une méthode sert à configurer la transaction dans laquelle va s'exécuter la méthode. Voici quelques valeurs possibles de cette annotation :
  - TransactionAttributeType.REQUIRED** : la méthode doit s'exécuter dans une transaction. Si une transaction a déjà démarré, les opérations de persistance de la méthode prennent place dans celle-ci. Sinon, une transaction est créée et démarrée.
  - TransactionAttributeType.REQUIRES\_NEW** : la méthode doit s'exécuter dans une transaction neuve. Celle-ci est créée et démarrée.
  - TransactionAttributeType.MANDATORY** : la méthode doit s'exécuter dans une transaction existante. Si celle-ci n'existe pas, une exception est lancée.
  - TransactionAttributeType.NEVER** : la méthode ne s'exécute jamais dans une transaction.
  - ...

L'annotation aurait pu être placée sur la classe elle-même :

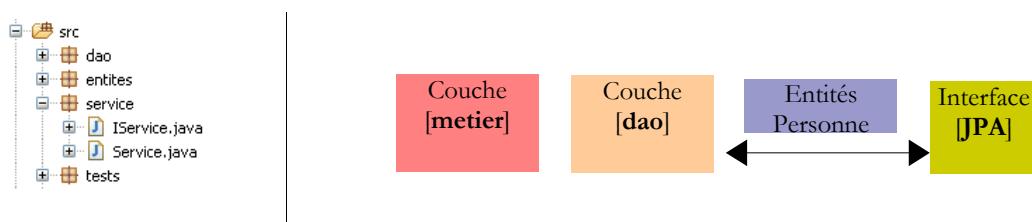
```

1. @Stateless
2. @TransactionAttribute(TransactionAttributeType.REQUIRED)
3. public class Dao implements IDao {

```

L'attribut est alors appliqué à toutes les méthodes de la classe.

### 3.2.4 La couche [metier / service]



La couche [service] présente l'interface [IService] étudiée précédemment au paragraphe 3.1.4, page 149. L'implémentation [Service] de l'interface [IService] est identique à l'implémentation étudiée précédemment au paragraphe 3.1.4, page 149 à trois détails près :

```

1. @Stateless
2. @TransactionAttribute(TransactionAttributeType.REQUIRED)
3. public class Service implements IService {
4.
5.
6.     // couche [dao]
7.     @EJB
8.     private IDao dao;
9.

```

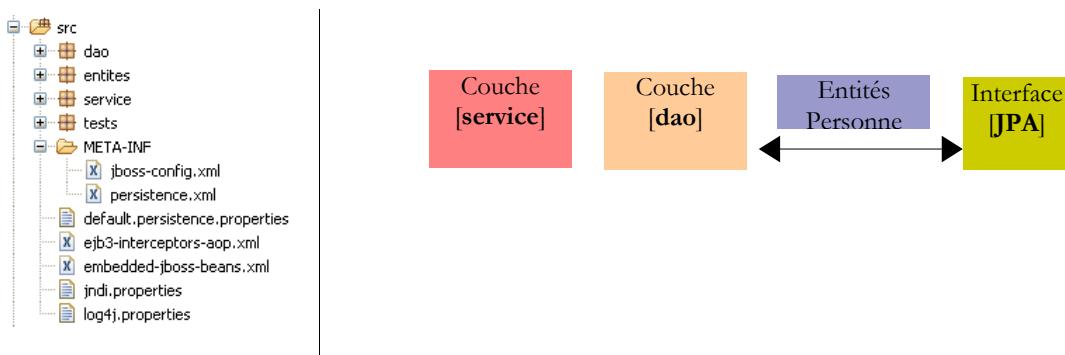
```

10. public IDao getDao() {
11.     return dao;
12. }
13.
14. public void setDao(IDao dao) {
15.     this.dao = dao;
16. }
17.

```

- ligne 2 : la classe [Service] est un Ejb sans état
- ligne 3 : toutes les méthodes de la classe [Service] doivent se dérouler dans une transaction
- lignes 7-8 : une référence sur l'Ejb de la couche [dao] sera injectée par le conteneur Ejb dans le champ [IDao dao] de la ligne 8. C'est l'annotation **@EJB** de la ligne 7 qui demande cette injection. L'objet injecté doit être un Ejb. C'est une différence importante avec Spring où tout type d'objet peut être injecté dans un autre objet.

### 3.2.5 Configuration des couches



La configuration des couches [service], [dao] et [JPA] est assurée par les fichiers suivants :

- [META-INF/**persistence.xml**] configure la couche JPA
- [**jboss-config.xml**] configure le conteneur Ejb3. Il utilise lui-même les fichiers [default.persistence.properties, ejb3-interceptors-aop.xml, embedded-jboss-beans.xml, jndi.properties]. Ces derniers fichiers sont livrés avec Jboss Ejb3 et assure une configuration par défaut à laquelle on ne touche normalement pas. Le développeur ne s'intéresse qu'au fichier [jboss-config.xml]

Examinons les deux fichiers de configuration :

#### **persistence.xml**

```

1. <persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
3.     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
4.
5. <persistence-unit name="jpa">
6.
7.   <!-- le fournisseur JPA est Hibernate -->
8.   <provider>org.hibernate.ejb.HibernatePersistence</provider>
9.
10.  <!-- la DataSource JTA gérée par l'environnement Java EE5 -->
11.  <jta-data-source>java:/datasource</jta-data-source>
12.
13.  <properties>
14.    <!-- recherche des entités de la couche JBA -->
15.    <property name="hibernate.archive.autodetection" value="class, hbm" />
16.
17.    <!-- logs SQL Hibernate
18.    <property name="hibernate.show_sql" value="true"/>
19.    <property name="hibernate.format_sql" value="true"/>
20.    <property name="use_sql_comments" value="true"/>
21.    -->
22.
23.    <!-- le type de SGBD géré -->
24.    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLInnoDBDialect" />
25.
26.    <!-- recréation de toutes les tables (drop+create) au déploiement de l'unité de persistance
-->
27.    <property name="hibernate.hbm2ddl.auto" value="create" />
28.

```

```

29.      </properties>
30.    </persistence-unit>
31.
32.</persistence>
```

Ce fichier ressemble à ceux que nous avons déjà rencontrés dans l'étude des entités JPA. Il configure une couche JPA Hibernate. Les nouveautés sont les suivantes :

- ligne 5 : l'unité de persistance *jpa* n'a pas l'attribut *transaction-type* qu'on avait toujours jusqu'ici :

```
1.<persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL" />
```

En l'absence de valeur, l'attribut *transaction-type* a la valeur par défaut "JTA" (pour Java Transaction Api) qui indique que le gestionnaire de transactions est fourni par un conteneur Ejb3. Un gestionnaire "JTA" peut faire davantage qu'un gestionnaire "RESOURCE\_LOCAL" : il peut gérer des transactions qui couvrent plusieurs connexions. Avec JTA, on peut ouvrir une transaction *t1* sur une connexion *c1* sur un SGBD 1, une transaction *t2* sur une connexion *c2* avec un SGBD 2 et être capable de considérer (*t1,t2*) comme une unique transaction dans laquelle soit toutes les opérations réussissent (commit) soit aucune (rollback).

ici, nous fonctionnons avec le gestionnaire JTA du conteneur Jboss Ejb3.

- ligne 11 : déclare la source de données que doit utiliser le gestionnaire JTA. Celle-ci est donnée sous la forme d'un nom JNDI (Java Naming and Directory Interface). Cette source de données est définie dans *jboss-config.xml*.

### jboss-config.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2.
3. <deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
   xmlns="urn:jboss:bean-deployer:2.0">
4.
5.
6.  <!-- factory de la DataSource -->
7.  <bean name="datasourceFactory" class="org.jboss.resource.adapter.jdbc.local.LocalTxDataSource">
8.    <!-- nom JNDI de la DataSource -->
9.    <property name="jndiName">java:/datasource</property>
10.
11.   <!-- base de données générée -->
12.   <property name="driverClass">com.mysql.jdbc.Driver</property>
13.   <property name="connectionURL">jdbc:mysql://localhost:3306/jpa</property>
14.   <property name="userName">jpa</property>
15.   <property name="password">jpa</property>
16.
17.   <!-- propriétés pool de connexions -->
18.   <property name="minSize">0</property>
19.   <property name="maxSize">10</property>
20.   <property name="blockingTimeout">1000</property>
21.   <property name="idleTimeout">100000</property>
22.
23.   <!-- gestionnaire de transactions, ici JTA -->
24.   <property name="transactionManager">
25.     <inject bean="TransactionManager" />
26.   </property>
27.   <!-- gestionnaire du cache Hibernate -->
28.   <property name="cachedConnectionManager">
29.     <inject bean="CachedConnectionManager" />
30.   </property>
31.   <!-- propriétés instanciation JNDI ? -->
32.   <property name="initialContextProperties">
33.     <inject bean="InitialContextProperties" />
34.   </property>
35. </bean>
36.
37. <!-- la DataSource est demandée à une factory -->
38. <bean name="datasource" class="java.lang.Object">
39.   <constructor factoryMethod="getDatasource">
40.     <factory bean="datasourceFactory" />
41.   </constructor>
42. </bean>
43.
44.</deployment>
```

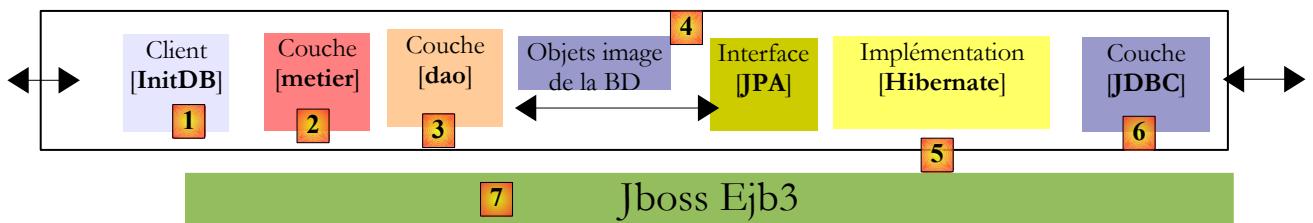
- ligne 3 : la balise racine du fichier est **<deployment>**. Ce fichier de déploiement vise essentiellement à configurer la source de données **java:/datasource** qui a été déclarée dans **persistence.xml**.

- la source de données est définie par le bean "datasource" de la ligne 38. On voit que la source de données est obtenue (ligne 40) auprès d'une "factory" définie par le bean "datasourceFactory" de la ligne 7. Pour obtenir la source de données de l'application, le client devra appeler la méthode [getDatasource] de la factory (ligne 39).
- ligne 7 : la factory qui délivre la source de données est une classe Jboss.
- ligne 9 : le nom JNDI de la source de données. Ce doit être le même nom que celui déclaré dans la balise <jta-datasource> du fichier **persistence.xml**. En effet, la couche JPA va utiliser ce nom JNDI pour demander la source de données.
- lignes 12-15 : quelque chose de plus classique : les caractéristiques Jdbc de la connexion au SGBD
- lignes 18-21 : configuration du pool de connexions interne du conteneur Jboss Ejb3.
- lignes 24-26 : le gestionnaire JTA. La classe [TransactionManager] injectée ligne 25 est définie dans le fichier [embedded-jboss-beans.xml].
- lignes 28-30 : le cache Hibernate, une notion que nous n'avons pas abordée. La classe [CachedConnectionManager] injectée ligne 29 est définie dans le fichier [embedded-jboss-beans.xml]. On notera que la configuration est maintenant dépendante d'Hibernate, ce qui nous posera problème lorsqu'on voudra migrer vers Toplink.
- lignes 32-34 : configuration du service JNDI.

Nous en avons fini avec le fichier de configuration de Jboss Ejb3. Il est complexe et bien des choses restent obscures. Il a été tiré de **[ref1]**. Nous serons cependant capables de l'adapter à un autre SGBD ( lignes 12-15 de **jboss-config.xml**, ligne 24 de **persistence.xml**). La migration vers Toplink n'a pas été possible faute d'exemples.

### 3.2.6 Programme client [InitDB]

Nous abordons l'écriture d'un premier client de l'architecture décrite précédemment :



Le code de [InitDB] est le suivant :

```

1. package tests;
2.
3. ...
4. public class InitDB {
5.
6.     // couche service
7.     private static IService service;
8.
9.     // constructeur
10.    public static void main(String[] args) throws ParseException, NamingException {
11.        // on démarre le conteneur EJB3 JBoss
12.        // les fichiers de configuration ejb3-interceptors-aop.xml et embedded-jboss-beans.xml sont
13.        // exploités
14.        EJB3StandaloneBootstrap.boot(null);
15.        // Création des beans propres à l'application
16.        EJB3StandaloneBootstrap.deployXmlResource("META-INF/jboss-config.xml");
17.        // Deploy all EJBs found on classpath (slow, scans all)
18.        // EJB3StandaloneBootstrap.scanClasspath();
19.        // on déploie tous les EJB trouvés dans le classpath de l'application
20.        EJB3StandaloneBootstrap.scanClasspath("bin".replace("/", File.separator));
21.        // On initialise le contexte JNDI. Le fichier jndi.properties est exploité
22.        InitialContext initialContext = new InitialContext();
23.
24.        // instanciation couche service
25.        service = (IService) initialContext.lookup("Service/local");
26.        // on vide la base
27.        clean();
28.        // on la remplit
29.        fill();
30.        // on vérifie visuellement
31.        dumpPersonnes();
32.        // on arrête le conteneur Ejb
33.        EJB3StandaloneBootstrap.shutdown();
34.    }
35.
36.

```

```

37.
38. }
39.
40. // affichage contenu table
41. private static void dumpPersonnes() {
42.
    System.out.format("[personnes]-----\n");
43.     for (Personne p : service.getAll()) {
44.         System.out.println(p);
45.     }
46. }
47.
48. // remplissage table
49. public static void fill() throws ParseException {
50.     // création personnes
51.     Personne p1 = new Personne("p1", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
52.         true, 2);
53.     Personne p2 = new Personne("p2", "Sylvie", new
54.         SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
55.         // qu'on sauvegarde
56.         service.saveArray(new Personne[] { p1, p2 });
57. }
58. // suppression éléments de la table
59. public static void clean() {
60.     for (Personne p : service.getAll()) {
61.         service.deleteOne(p.getId());
62.     }
63. }

```

- la façon de lancer le conteneur Jboss Ejb3 a été trouvée dans [ref1].
- ligne 13 : le conteneur est lancé. [EJB3StandaloneBootstrap] est une classe du conteneur.
- ligne 16 : l'unité de déploiement configurée par [jboss-config.xml] est déployée dans le conteneur : gestionnaire JTA, source de données, pool de connexions, cache Hibernate, service JNDI sont mis en place.
- ligne 22 : on demande au conteneur de scanner le dossier **bin** du projet Eclipse pour y trouver les Ejb. Les Ejb des couches [service] et [dao] vont être trouvées et gérées par le conteneur.
- ligne 25 : un contexte JNDI est initialisé. Il va nous servir à localiser les Ejb.
- ligne 28 : l'Ejb correspondant à la classe [Service] de la couche [service] est demandé au service JNDI. Un Ejb peut être accédé localement (local) ou via le réseau (remote). Ici le nom "Service/local" de l'Ejb cherché désigne la classe [Service] de la couche [service] pour un accès local.
- maintenant, l'application est déployée et on détient une référence sur la couche [service]. On est dans la même situation qu'après la ligne 11 ci-dessous du code [InitDB] de la version Spring. On retrouve alors le même code dans les deux versions.

```

1. public class InitDB {
2.
3.     // couche service
4.     private static IService service;
5.
6.     // constructeur
7.     public static void main(String[] args) throws ParseException {
8.         // configuration de l'application
9.         ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config.xml");
10.        // couche service
11.        service = (IService) ctx.getBean("service");
12.        // on vide la base
13.        clean();
14.        // on la remplit
15.        fill();
16.        // on vérifie visuellement
17.        dumpPersonnes();
18.    }
19. ...

```

- ligne 36 (Jboss Ejb3) : on arrête le conteneur Ejb3.

L'exécution de [InitDB] donne les résultats suivants :

```

1. 16:07:00,781  INFO LocalTxDataSource:117 - Bound datasource to JNDI name 'java:/datasource'
2. ...
3. 16:07:01,171  INFO Version:94 - Hibernate EntityManager 3.2.0.CR1
4. ...
5. 16:07:01,296  INFO Ejb3Configuration:94 - Processing PersistenceUnitInfo [
6.   name: jpa
7.   ...
8. 16:07:01,312  INFO Ejb3Configuration:94 - found EJB3 Entity bean: entites.Personne
9. ...

```

```

10. 16:07:01,375 INFO Configuration:94 - Reading mappings from resource: META-INF/orm.xml
11. 16:07:01,375 INFO Ejb3Configuration:94 - [PersistenceUnit: jpa] no META-INF/orm.xml found
12. 16:07:01,421 INFO AnnotationBinder:94 - Binding entity from annotated class: entites.Personne
13. 16:07:01,468 INFO EntityBinder:94 - Bind entity entites.Personne on table jpa01_hb_personne
14. ...
15. 16:07:01,859 INFO SettingsFactory:94 - RDBMS: MySQL, version: 5.0.41-community-nt
16. 16:07:01,859 INFO SettingsFactory:94 - JDBC driver: MySQL-AB JDBC Driver, version: mysql-
    connector-java-5.0.5 ( $Date: 2007-03-01 00:01:06 +0100 (Thu, 01 Mar 2007) $, $Revision: 6329 $ )
17. 16:07:01,890 INFO Dialect:94 - Using dialect: org.hibernate.dialect.MySQLInnoDBDialect
18. 16:07:01,890 INFO TransactionFactoryFactory:94 - Transaction strategy:
    org.hibernate.ejb.transaction.JoinableCMTTransactionFactory
19. ...
20. 16:07:02,234 INFO SchemaExport:94 - Running hbm2ddl schema export
21. 16:07:02,234 INFO SchemaExport:94 - exporting generated schema to database
22. 16:07:02,343 INFO SchemaExport:94 - schema export complete
23. ...
24. 16:07:02,562 INFO EJBContainer:479 - STARTED EJB: dao.Dao ejbName: Dao
25. ...
26. 16:07:02,593 INFO EJBContainer:479 - STARTED EJB: service.Service ejbName: Service
27. ...
28. [personnes]-----
29. [1,0,p1,Paul,31/01/2000,true,2]
30. [2,0,p2,Sylvie,05/07/2001,false,0]

```

Le lecteur est invité à lire ces logs. On y trouve des informations intéressantes sur ce que fait le conteneur Ejb3.

### 3.2.7 Tests unitaires [TestNG]

Le code du programme [TestNG] est le suivant :

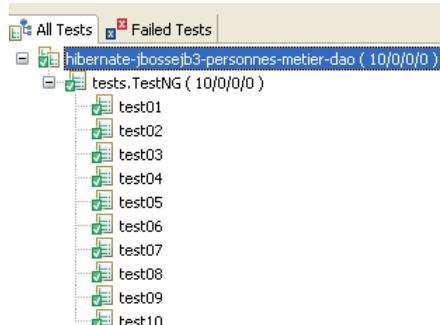
```

1. package tests;
2.
3. ...
4. public class TestNG {
5.
6.     // couche service
7.     private IService service = null;
8.
9.     @BeforeClass
10.    public void init() throws NamingException, ParseException {
11.         // log
12.         log("init");
13.         // on démarre le conteneur EJB3 JBoss
14.         // les fichiers de configuration ejb3-interceptors-aop.xml et embedded-jboss-beans.xml sont
            exploités
15.         EJB3StandaloneBootstrap.boot(null);
16.
17.         // Création des beans propres à l'application
18.         EJB3StandaloneBootstrap.deployXmlResource("META-INF/jboss-config.xml");
19.
20.         // Deploy all EJBs found on classpath (slow, scans all)
21.         // EJB3StandaloneBootstrap.scanClasspath();
22.
23.         // on déploie tous les EJB trouvés dans le classpath de l'application
24.         EJB3StandaloneBootstrap.scanClasspath("bin".replace("/", File.separator));
25.
26.         // On initialise le contexte JNDI. Le fichier jndi.properties est exploité
27.         InitialContext initialContext = new InitialContext();
28.
29.         // instanciation couche service
30.         service = (IService) initialContext.lookup("Service/local");
31.         // on vide la base
32.         clean();
33.         // on la remplit
34.         fill();
35.         // on vérifie visuellement
36.         dumpPersonnes();
37.     }
38.
39.     @AfterClass
40.     public void terminate() {
41.         // log
42.         log("terminate");
43.         // Shutdown EJB container
44.         EJB3StandaloneBootstrap.shutdown();
45.     }
46.
47.     @BeforeMethod
48.     public void setUp() throws ParseException {
49.     ...
50.     }
51.
52. ...

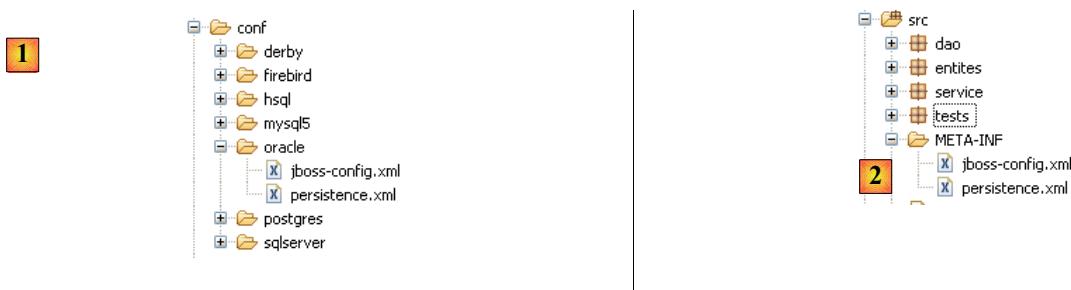
```

- la méthode **init** (lignes 10-37) qui sert à mettre en place l'environnement nécessaire aux tests reprend le code expliqué précédemment dans [InitDB].
- la méthode **terminate** (lignes 40-45) qui est exécutée à la fin des tests (présence de l'annotation **@AfterClass**) arrête le conteneur Ejb3 (ligne 44).
- tout le reste est identique à ce qu'il était dans la version Spring.

Les tests réussissent :



### 3.2.8 Changer de SGBD



Pour changer de SGBD, il suffit de remplacer le contenu du dossier [META-INF] [2] par celui du dossier du SGBD dans le dossier [conf] [1]. Prenons l'exemple de **SQL Server** :

Le fichier [persistence.xml] est le suivant :

```

1. <persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
3.     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
4.
5.   <persistence-unit name="jpa">
6.
7.     <!-- le fournisseur JPA est Hibernate -->
8.     <provider>org.hibernate.ejb.HibernatePersistence</provider>
9.
10.    <!-- la DataSource JTA gérée par l'environnement Java EE5 -->
11.    <jta-data-source>java:/datasource</jta-data-source>
12.
13.    <properties>
14.      <!-- recherche des entités de la couche JBA -->
15.      <property name="hibernate.archive.autodetection" value="class, hbm" />
16.
17.      <!-- logs SQL Hibernate
18.          <property name="hibernate.show_sql" value="true"/>
19.          <property name="hibernate.format_sql" value="true"/>
20.          <property name="use_sql_comments" value="true"/>
21.      -->
22.
23.      <!-- le type de SGBD géré -->
24.      <property name="hibernate.dialect" value="org.hibernate.dialect.SQLServerDialect" />
25.
26.      <!-- recréation de toutes les tables (drop+create) au déploiement de l'unité de persistance
-->
27.      <property name="hibernate.hbm2ddl.auto" value="create" />
28.
29.    </properties>
30.  </persistence-unit>
```

```
31.  
32. </persistence>
```

Seule une ligne a changé :

- ligne 24 : le dialecte SQL qu'Hibernate doit utiliser

Le fichier [jboss-config.xml] de SQL Server est lui, le suivant :

```
1. <?xml version="1.0" encoding="UTF-8"?>  
2.  
3. <deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
   xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"  
   xmlns="urn:jboss:bean-deployer:2.0">  
5.  
6.   <!-- factory de la DataSource -->  
7.   <bean name="datasourceFactory" class="org.jboss.resource.adapter.jdbc.local.LocalTxDataSource">  
8.     <!-- nom JNDI de la DataSource -->  
9.     <property name="jndiName">java:/datasource</property>  
10.    <!-- base de données générée -->  
12.    <property name="driverClass">com.microsoft.sqlserver.jdbc.SQLServerDriver</property>  
13.    <property  
        name="connectionURL">jdbc:sqlserver://localhost\\SQLEXPRESS:1246;databaseName=jpa</property>  
14.    <property name="userName">jpa</property>  
15.    <property name="password">jpa</property>  
16.  
17.      <!-- propriétés pool de connexions -->  
18.      ...  
19.  </bean>  
20.  
21. </deployment>
```

Seules les lignes 12-15 ont été changées : elles donnent les caractéristiques de la nouvelle connexion Jdbc.

Le lecteur est invité à répéter avec d'autres SGBD les tests décrits pour MySQL5.

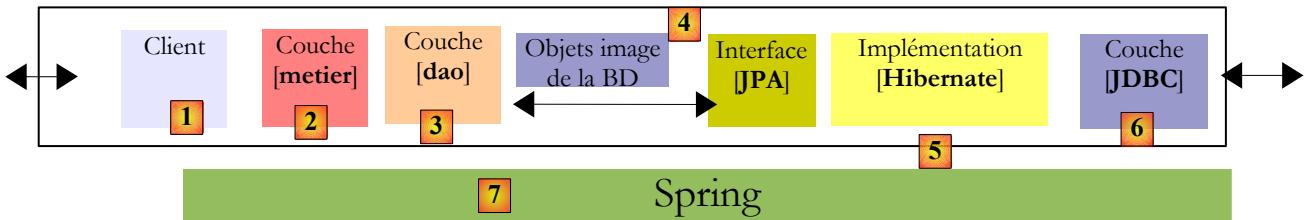
### 3.2.9 Changer d'implémentation JPA

Comme il a été indiqué plus haut, nous n'avons pas trouvé d'exemple d'utilisation du conteneur Jboss Ejb3 avec Toplink. A ce jour (juin 2007), je ne sais toujours pas si cette configuration est possible.

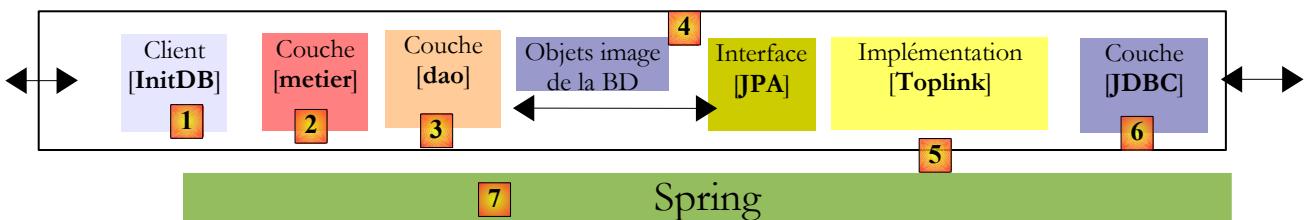
### 3.3 Autres exemples

Résumons ce qui a été fait avec l'entité *Personne*. Nous avons construit trois architectures pour conduire les mêmes tests :

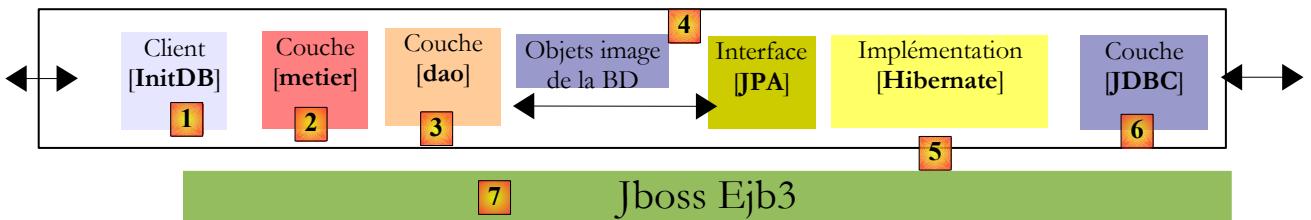
1 - une implémentation **Spring / Hibernate**



2 - une implémentation **Spring / Toplink**

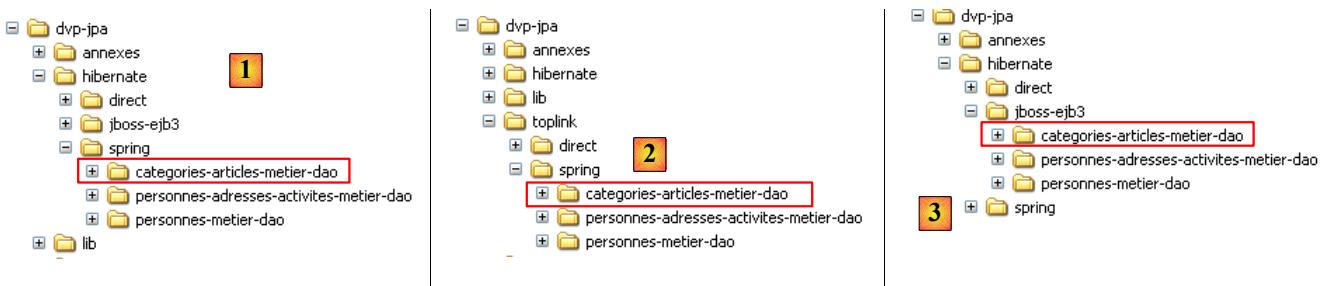


3 - une implémentation **Jboss Ejb3 / Hibernate**



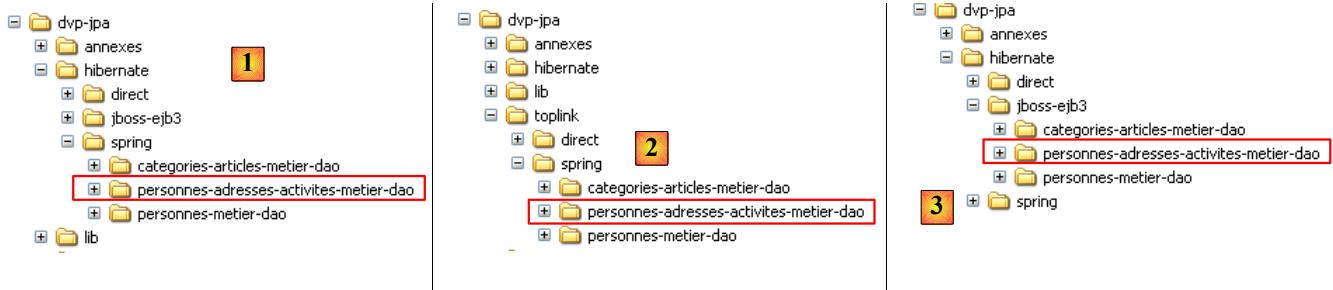
Les exemples du tutoriel reprennent ces trois architectures avec d'autres entités étudiées dans la première partie du tutoriel :

#### Categorie - Article



- en [1] : la version Spring / Hibernate
- en [2] : la version Spring / Toplink
- en [3] : la version Jboss Ejb3 / Hibernate

## Personne- Adresse - Activite



- en [1] : la version Spring / Hibernate
- en [2] : la version Spring / Toplink
- en [3] : la version Jboss Ejb3 / Hibernate

Ces exemples n'amènent pas de nouveautés quant à l'architecture. Ils se placent simplement dans un contexte où il y a plusieurs entités à gérer avec des relations **un-à-plusieurs** ou **plusieurs-à-plusieurs** entre elles, ce que n'avaient pas les exemples avec l'entité *Personne*.

## 3.4 Exemple 3 : Spring / JPA dans une application web

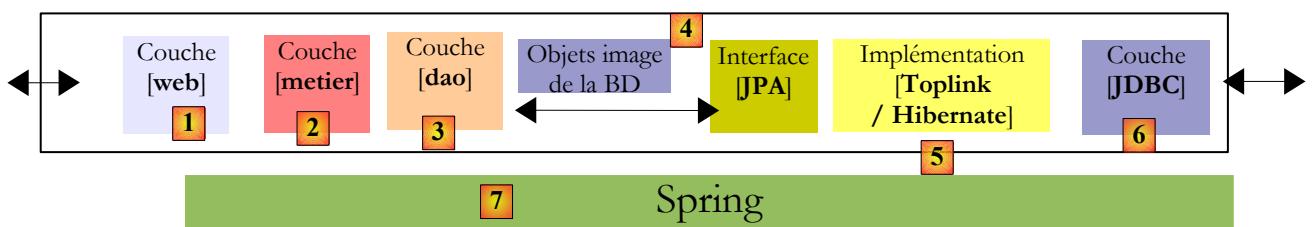
### 3.4.1 Présentation

Nous reprenons ici une application présentée dans le document suivant :

[ref4] : Les bases du développement web MVC en Java [<http://tahe.developpez.com/java/baseswebmvc/>].

Ce document présente les bases du développement web MVC en Java. Pour comprendre l'exemple qui suit, le lecteur doit avoir ces bases. L'application web utilisera le serveur Tomcat. L'installation de celui-ci et son utilisation au sein d'Eclipse sont présentées au paragraphe 5.3, page 221.

L'application avait été développée avec une couche [dao] s'appuyant sur l'outil Ibatis / SqlMap [<http://ibatis.apache.org/>] qui assurait le pont relationnel / objet. Nous nous contentons de remplacer Ibatis par JPA. L'architecture de l'application sera la suivante :



L'application web que nous allons écrire va permettre de gérer un groupe de personnes avec quatre opérations :

- liste des personnes du groupe
- ajout d'une personne au groupe
- modification d'une personne du groupe
- suppression d'une personne du groupe

On reconnaîtra les quatre opérations de base sur une table de base de données. Les copies d'écran qui suivent montrent les pages que l'application échange avec l'utilisateur.

MVC - Personnes

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list;jsessionid=AE7D07121E395B01791;\_

Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

Ajout

Une liste initiale de personnes est tout d'abord présentée à l'utilisateur. Il peut ajouter une personne ->

MVC - Personnes

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/edit?id=1

Ajout/Modification d'une personne

Id	-1
Version	-1
Prénom	Sophie
Nom	p3
Date de naissance (JJ/MM/AAAA)	13/03/1946
Marié	<input checked="" type="radio"/> Oui <input type="radio"/> Non
Nombre d'enfants	4

Valider Annuler

L'utilisateur a créé une nouvelle personne qu'il valide avec le bouton [Valider] ->

MVC - Personnes

<http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list>

### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
15	0	Sophie	p3	13/03/1946	true	4	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

La nouvelle personne a été ajoutée. On la modifie maintenant ->

MVC - Personnes

<http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/edit?id=15>

### Ajout/Modification d'une personne

Id	15
Version	0
Prénom	Sophie
Nom	p3
Date de naissance (JJ/MM/AAAA)	13/03/1956
Marié	<input type="radio"/> Oui <input checked="" type="radio"/> Non
Nombre d'enfants	2

[Valider](#) [Annuler](#)

On modifie la date de naissance, l'état marital, le nombre d'enfants et on valide ->

MVC - Personnes

<http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list>

### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
15	1	Sophie	p3	13/03/1956	false	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

On retrouve la personne telle qu'elle a été modifiée. On la supprime maintenant ->

MVC - Personnes

<http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list>

### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

Elle n'est plus là. On ajoute une personne ->

MVC - Personnes

<http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/edit?id=-1>

### Ajout/Modification d'une personne

Id	-1
Version	-1
Prénom	<input type="text"/>
Nom	<input type="text"/>
Date de naissance (JJ/MM/AAAA)	<input type="text"/>
Marié	<input type="radio"/> Oui <input checked="" type="radio"/> Non
Nombre d'enfants	xx

[Valider](#) [Annuler](#)

Les erreurs de saisie sont signalées ->

MVC - Personnes

<http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/validate>

### Ajout/Modification d'une personne

Id	-1	
Version	-1	
Prénom	<input type="text"/>	Le prénom est obligatoire
Nom	<input type="text"/>	Le nom est obligatoire
Date de naissance (JJ/MM/AAAA)	<input type="text"/>	Date incorrecte
Marié	<input type="radio"/> Oui <input checked="" type="radio"/> Non	
Nombre d'enfants	xx	Nombre d'enfants incorrect

[Valider](#) [Annuler](#)

On notera que le formulaire a été renvoyé tel qu'il a été saisi (Nombre d'enfants). Le lien [Annuler] permet de revenir à la liste des personnes ->

MVC - Personnes

<http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list>

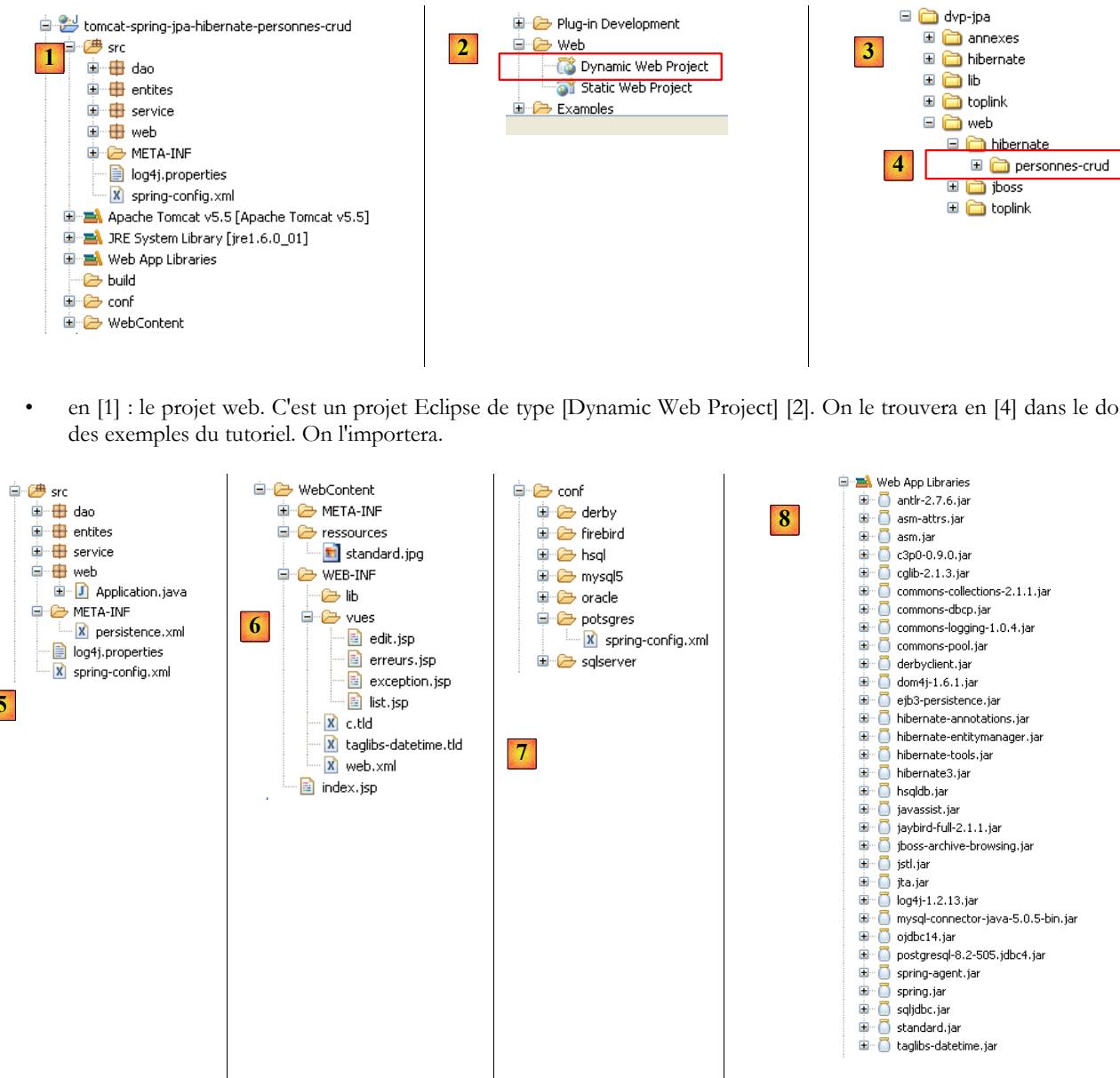
### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

### 3.4.2 Le projet Eclipse

Le projet Eclipse de l'application est le suivant :



- en [1] : le projet web. C'est un projet Eclipse de type [Dynamic Web Project] [2]. On le trouvera en [4] dans le dossier [3] des exemples du tutoriel. On l'importera.

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. ...
4.   <bean id="entityManagerFactory"
5.     class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
6.     <property name="dataSource" ref="dataSource" />
7.     <property name="jpaVendorAdapter">
8.       <property name="databasePlatform" value="org.hibernate.dialect.PostgreSQLDialect" />
9.     ...
10.    </property>
11.  ...
12. </bean>
13.
14.  <!-- la source de données DBCP -->
15. <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
16.   <property name="driverClassName" value="org.postgresql.Driver" />
17.   <property name="url" value="jdbc:postgresql:jpa" />
18.   <property name="username" value="jpa" />

```

```

19.    <property name="password" value="jpa" />
20.  </bean>
21. ....
22.</beans>

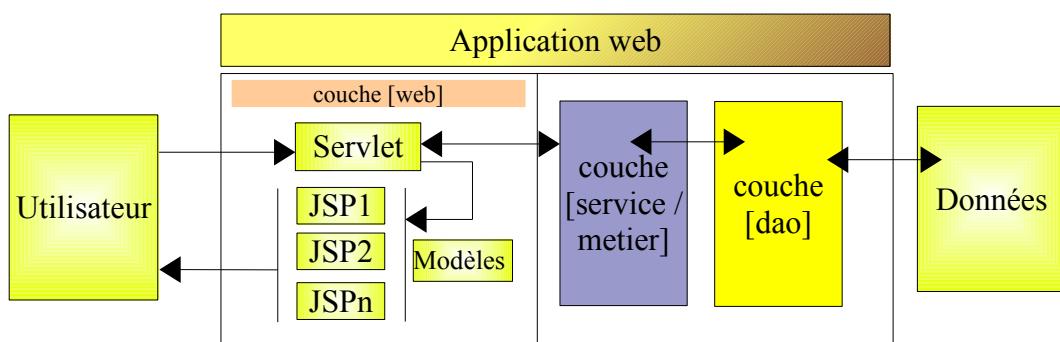
```

Les lignes 8 et 16-19 ont été adaptées à Postgres.

- en [6] : le dossier [WebContent] contient les pages JSP du projet ainsi que les bibliothèques nécessaires. Ces dernières sont présentées en [8]
- l'application est utilisable avec divers SGBD. Il suffit de changer le fichier [spring-config.xml]. Le dossier [conf] [7] contient le fichier [spring-config.xml] adapté à divers SGBD.

### 3.4.3 La couche [web]

Notre application a l'architecture multi-couches suivante :



La couche [web] va offrir des écrans à l'utilisateur pour lui permettre de gérer le groupe de personnes :

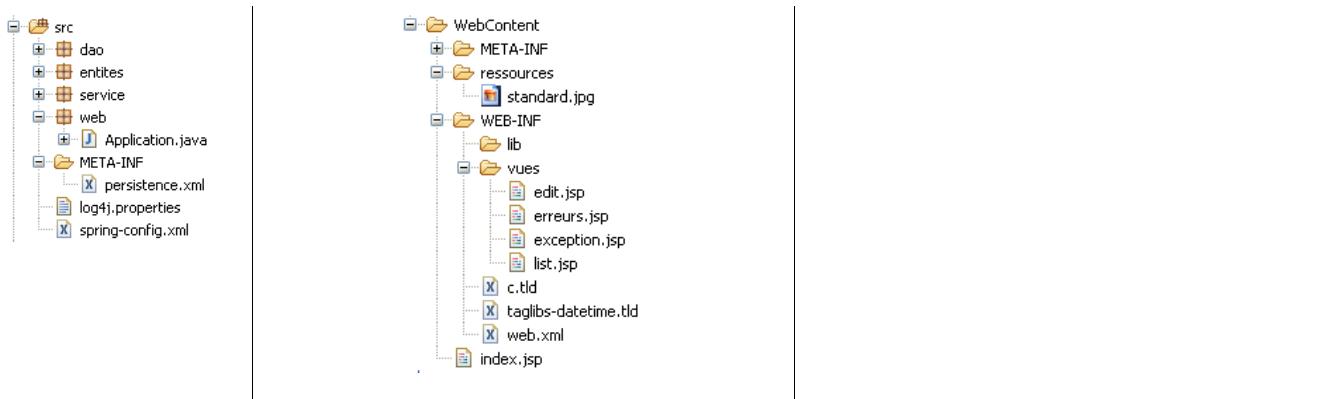
- liste des personnes du groupe
- ajout d'une personne au groupe
- modification d'une personne du groupe
- suppression d'une personne du groupe

Pour cela, elle va s'appuyer sur la couche [service] qui elle-même fera appel à la couche [dao]. Nous avons déjà présenté les écrans générés par la couche [web] (page 179). Pour décrire la couche web, nous allons présenter successivement :

- sa configuration
- ses vues
- son contrôleur
- quelques tests

#### 3.4.3.1 Configuration de l'application web

Revenons sur l'architecture du projet Eclipse :



- dans le paquetage [web], on trouve le contrôleur de l'application web : la classe [Application].

- les pages JSP / JSTL de l'application sont dans [WEB-INF/vues].
- le dossier [WEB-INF/lib] contient les archives tierces nécessaires à l'application. Elles sont visibles dans le dossier [Web App Libraries].

---

### [web.xml]

Le fichier [web.xml] est le fichier exploité par le serveur web pour charger l'application. Son contenu est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
   app_2_4.xsd">
4.   <display-name>spring-jpa-hibernate-personnes-crud</display-name>
5.   <!-- ServletPersonne -->
6.   <servlet>
7.     <servlet-name>personnes</servlet-name>
8.     <servlet-class>web.Application</servlet-class>
9.     <init-param>
10.       <param-name>urlEdit</param-name>
11.       <param-value>/WEB-INF/vues/edit.jsp</param-value>
12.     </init-param>
13.     <init-param>
14.       <param-name>urlErreurs</param-name>
15.       <param-value>/WEB-INF/vues/erreurs.jsp</param-value>
16.     </init-param>
17.     <init-param>
18.       <param-name>urlList</param-name>
19.       <param-value>/WEB-INF/vues/list.jsp</param-value>
20.     </init-param>
21.   </servlet>
22.   <!-- Mapping ServletPersonne-->
23.   <servlet-mapping>
24.     <servlet-name>personnes</servlet-name>
25.     <url-pattern>/do/*</url-pattern>
26.   </servlet-mapping>
27.   <!-- fichiers d'accueil -->
28.   <welcome-file-list>
29.     <welcome-file>index.jsp</welcome-file>
30.   </welcome-file-list>
31.   <!-- Page d'erreur inattendue -->
32.   <error-page>
33.     <exception-type>java.lang.Exception</exception-type>
34.     <location>/WEB-INF/vues/exception.jsp</location>
35.   </error-page>
36. </web-app>
```

- lignes 23-26 : les url [/do/\*] seront traitées par la servlet [personnes]
- lignes 7-8 : la servlet [personnes] est une instance de la classe [Application], une classe que nous allons construire.
- lignes 9-20 : définissent trois paramètres [urlList, urlEdit, urlErreurs] identifiant les Url des pages JSP des vues [list, edit, erreurs].
- lignes 28-30 : l'application a une page d'entrée par défaut [index.jsp] qui se trouve à la racine du dossier de l'application web.
- lignes 32-35 : l'application a une page d'erreurs par défaut qui est affichée lorsque le serveur web récupère une exception non gérée par l'application.
  - ligne 37 : la balise <exception-type> indique le type d'exception gérée par la directive <error-page>, ici le type [java.lang.Exception] et dérivé, donc toutes les exceptions.
  - ligne 38 : la balise <location> indique la page JSP à afficher lorsqu'une exception du type défini par <exception-type> se produit. L'exception e survenue est disponible à cette page dans un objet nommé exception si la page a la directive :

```
<%@ page isErrorPage="true" %>
```

- si <exception-type> précise un type T1 et qu'une exception de type T2 non dérivé de T1 remonte jusqu'au serveur web, celui-ci envoie au client une page d'exception propriétaire généralement peu conviviale. D'où l'intérêt de la balise <error-page> dans le fichier [web.xml].

---

### [index.jsp]

Cette page est présentée si un utilisateur demande directement le contexte de l'application sans préciser d'url, c.a.d. ici [/spring-jpa-hibernate-personnes-crud]. Son contenu est le suivant :

```

1. <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html; charset=ISO-8859-1%>
2. <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.
```

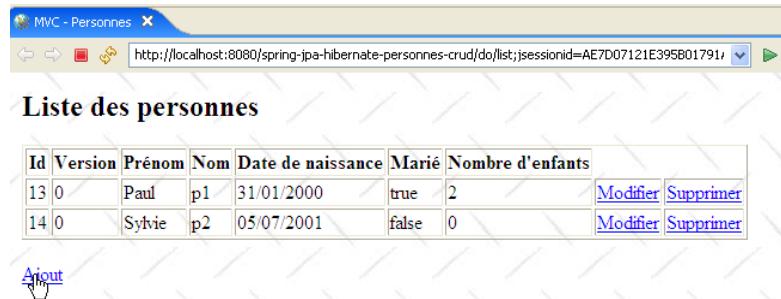
```
|4. <c:redirect url="/do/list"/>
```

[index.jsp] redirige (ligne 4) le client vers l'url [/do/list]. Cette url affiche la liste des personnes du groupe.

### 3.4.3.2 Les pages JSP / JSTL de l'application

La vue [list.jsp]

Elle sert à afficher la liste des personnes :



Son code est le suivant :

```

1. <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html; charset=ISO-8859-1"%>
2. <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3. <%@ taglib uri="/WEB-INF/taglibs-datetime.tld" prefix="dt" %>
4.
5. <html>
6.   <head>
7.     <title>MVC - Personnes</title>
8.   </head>
9.   <body background="">
10.    <c:if test="${erreurs!=null}">
11.      <h3>Les erreurs suivantes se sont produites :</h3>
12.      <ul>
13.        <c:forEach items="${erreurs}" var="erreur">
14.          <li><c:out value="${erreur}" /></li>
15.        </c:forEach>
16.      </ul>
17.      <hr>
18.    </c:if>
19.    <h2>Liste des personnes</h2>
20.    <table border="1">
21.      <tr>
22.        <th>Id</th>
23.        <th>Version</th>
24.        <th>Pr&eacute;nom</th>
25.        <th>Nom</th>
26.        <th>Date de naissance</th>
27.        <th>Mari eacute;</th>
28.        <th>Nombre d'enfants</th>
29.        <th></th>
30.      </tr>
31.      <c:forEach var="personne" items="${personnes}">
32.        <tr>
33.          <td><c:out value="${personne.id}" /></td>
34.          <td><c:out value="${personne.version}" /></td>
35.          <td><c:out value="${personne.prenom}" /></td>
36.          <td><c:out value="${personne.nom}" /></td>
37.          <td><dt:format pattern="dd/MM/yyyy">${personne.datenaissance.time}</dt:format></td>
38.          <td><c:out value="${personne.marie}" /></td>
39.          <td><c:out value="${personne.nbenfants}" /></td>
40.          <td><a href="

```

- cette vue reçoit deux éléments dans son modèle :
  - l'élément **[personnes]** associé à un objet de type [List] d'objets de type [Personne] : une liste de personnes.
  - l'élément facultatif **[erreurs]** associé à un objet de type [List] d'objets de type [String] : une liste de messages d'erreur.

- lignes 31-43 : on parcourt la liste \${personnes} pour afficher un tableau HTML contenant les personnes du groupe.
- ligne 40 : l'url pointée par le lien [Modifier] est paramétrée par le champ [id] de la personne courante afin que le contrôleur associé à l'url [/do/edit] sache quelle est la personne à modifier.
- ligne 41 : il est fait de même pour le lien [Supprimer].
- ligne 37 : pour afficher la date de naissance de la personne sous la forme JJ/MM/AAAA, on utilise la balise <dt> de la bibliothèque de balise [DateTime] du projet Apache [Jakarta Taglibs] :



[Home](#)  
[Jakarta Taglibs](#)  
**JCP**  
**Standardized**  
**Tag Libraries**

## Date Time Tag Library

The DateTime custom tag library contains tags which can be used to handle date and time related functions. Tags are provided for formatting a Date for output, generating a Date from HTML form input, using time zones, and localization.

Le fichier de description de cette bibliothèque de balises est défini ligne 3.

- ligne 46 : le lien [Ajout] d'ajout d'une nouvelle personne a pour cible l'url [/do/edit] comme le lien [Modifier] de la ligne 40. C'est la valeur -1 du paramètre [id] qui indique qu'on a affaire à un ajout plutôt qu'une modification.
- lignes 10-18 : si l'élément \${erreurs} est dans le modèle, alors on affiche les messages d'erreurs qu'il contient.

La vue [edit.jsp]

Elle sert à afficher le formulaire d'ajout d'une nouvelle personne ou de modification d'une personne existante :

**Liste des personnes**

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
15	0	Sophie	p3	13/03/1946	true	4	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

La nouvelle personne a été ajoutée. On la modifie maintenant ->

**Ajout/Modification d'une personne**

Id	15
Version	0
Prénom	Sophie
Nom	p3
Date de naissance (JJ/MM/AAAA)	13/03/1956
Marié	<input type="radio"/> Oui <input checked="" type="radio"/> Non
Nombre d'enfants	2

[Valider](#) [Annuler](#)

On modifie la date de naissance, l'état marital, le nombre d'enfants et on valide ->

Le code de la vue [edit.jsp] est le suivant :

```

1. <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html; charset=ISO-8859-1"%>
2. <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3. <%@ taglib uri="/WEB-INF/taglibs-datetime.tld" prefix="dt" %>
4.
5. <html>
6.   <head>
7.     <title>MVC - Personnes</title>
8.   </head>
9.   <body background="../ressources/standard.jpg">
10.    <h2>Ajout/Modification d'une personne</h2>
11.    <c:if test="${erreurEdit!=''}">
12.      <h3>Echec de la mise à jour :</h3>
13.      L'erreur suivante s'est produite : ${erreurEdit}

```

```

14.      <hr>
15.    </c:if>
16.    <form method="post" action="Annuler</a>
75.    </form>
76.  </body>
77. </html>

```

Cette vue présente un formulaire d'ajout d'une nouvelle personne ou de mise à jour d'une personne existante. Par la suite et pour simplifier l'écriture, nous utiliserons l'unique terme de [mise à jour]. Le bouton [Valider] (ligne 73) provoque le POST du formulaire à l'url [/do/validate] (ligne 16). Si le POST échoue, la vue [edit.jsp] est réaffichée avec la ou les erreurs qui se sont produites, sinon la vue [list.jsp] est affichée.

- la vue [edit.jsp] affichée aussi bien sur un GET que sur un POST qui échoue, reçoit les éléments suivants dans son modèle :

attribut	GET	POST
id	identifiant de la personne mise à jour	idem
version	sa version	idem
prenom	son prénom	prénom saisi
nom	son nom	nom saisi

attribut	GET	POST
datenaissance	sa date de naissance	date de naissance saisie
marie	son état marital	état marital saisie
nbenfants	son nombre d'enfants	nombre d'enfants saisie
erreurEdit	vide	un message d'erreur signalant un échec de l'ajout ou de la modification au moment du POST provoqué par le bouton [Envoyer]. Vide si pas d'erreur.
erreurPrenom	vide	signale un prénom erroné – vide sinon
erreurNom	vide	signale un nom erroné – vide sinon
erreurDateNaissance	vide	signale une date de naissance erronée – vide sinon
erreurNbEnfants	vide	signale un nombre d'enfants erroné – vide sinon

- lignes 11-15 : si le POST du formulaire se passe mal, on aura [erreurEdit!=""] et un message d'erreur sera affiché.
- ligne 16 : le formulaire sera posté à l'url [/do/validate]
- ligne 20 : l'élément [id] du modèle est affiché
- ligne 24 : l'élément [version] du modèle est affiché
- lignes 26-32 : saisie du prénom de la personne :
  - lors de l'affichage initial du formulaire (GET), \${prenom} affiche la valeur actuelle du champ [prenom] de l'objet [Personne] mis à jour et \${erreurPrenom} est vide.
  - en cas d'erreur après le POST, on réaffiche la valeur saisie \${prenom} ainsi que le message d'erreur éventuel \${erreurPrenom}
- lignes 33-39 : saisie du nom de la personne
- lignes 40-46 : saisie de la date de naissance de la personne
- lignes 47-61 : saisie de l'état marié ou non de la personne avec un bouton radio. On utilise la valeur du champ [marie] de l'objet [Personne] pour savoir lequel des deux boutons radio doit être coché.
- lignes 62-68 : saisie du nombre d'enfants de la personne
- ligne 71 : un champ HTML caché nommé [id] et ayant pour valeur le champ [id] de la personne en cours de mise à jour, -1 pour un ajout, autre chose pour une modification.
- ligne 72 : un champ HTML caché nommé [version] et ayant pour valeur le champ [id] de la personne en cours de mise à jour.
- ligne 73 : le bouton [Valider] de type [Submit] du formulaire
- ligne 74 : un lien permettant de revenir à la liste des personnes. Il a été libellé [Annuler] parce qu'il permet de quitter le formulaire sans le valider.

## La vue [exception.jsp]

Elle sert à afficher une page signalant qu'il s'est produit une exception non gérée par l'application et qui est remontée jusqu'à l'serveur web.

Par exemple, supprimons une personne qui n'existe pas dans le groupe :

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Actions
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a> <a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a> <a href="#">Supprimer</a>

Ajout

la vue [list.jsp] - il n'y a pas de personne d'id=7

la vue [exception.jsp] – on a demandé la suppression de la personne d'id=7 en tapant à la main l'url dans le navigateur.

Le code de la vue [exception.jsp] est le suivant :

```

1. <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html; charset=ISO-8859-1"%>
2. <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3. <%@ page isErrorPage="true" %>
4.
5. <%
6.     response.setStatus(200);
7. %>
8.

```

```

9. <html>
10.  <head>
11.    <title>MVC - Personnes</title>
12.  </head>
13. <body background="Retour à la liste</a>
19. </body>
20. </html>

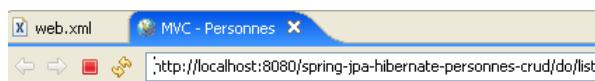
```

- cette vue reçoit une clé dans son modèle l'élément [exception] qui est l'exception qui a été interceptée par le serveur web. Pour que cet élément soit inclus dans le modèle de la page JSP par le serveur web, il faut que la page ait défini la balise de la ligne 3.
- ligne 6 : on fixe à 200 le code d'état HTTP de la réponse. C'est le premier entête HTTP de la réponse. Le code 200 signifie au client que sa demande a été honorée. Généralement un document HTML a été intégré dans la réponse du serveur. C'est le cas ici. Si on ne fixe pas à 200 le code d'état HTTP de la réponse, il aura ici la valeur 500 qui signifie qu'il s'est produit une erreur. En effet, le serveur web ayant intercepté une exception non gérée trouve cette situation anormale et le signale par le code 500. La réaction au code HTTP 500 diffère selon les navigateurs : Firefox affiche le document HTML qui peut accompagner cette réponse alors qu'IE ignore ce document et affiche sa propre page. C'est pour cette raison que nous avons remplacé le code 500 par le code 200.
- ligne 16 : le texte de l'exception est affiché
- ligne 18 : on propose à l'utilisateur un lien pour revenir à la liste des personnes

---

#### La vue [erreurs.jsp]

Elle sert à afficher une page signalant les erreurs d'initialisation de l'application, c.a.d. les erreurs détectées lors de l'exécution de la méthode [init] de la servlet du contrôleur. Ce peut être par exemple l'absence d'un paramètre dans le fichier [web.xml] comme le montre l'exemple ci-dessous :



#### Les erreurs suivantes se sont produites

- Le paramètre [urlList] n'a pas été initialisé

Le code de la page [erreurs.jsp] est le suivant :

```

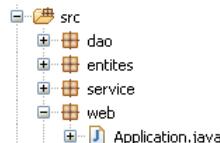
1. <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2.   pageEncoding="ISO-8859-1"%>
3. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
4. <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
5.
6. <html>
7.  <head>
8.    <title>MVC - Personnes</title>
9.  </head>
10. <body>
11.   <h2>Les erreurs suivantes se sont produites</h2>
12.   <ul>
13.     <c:forEach var="erreur" items="${erreurs}">
14.       <li>${erreur}</li>
15.     </c:forEach>
16.   </ul>
17. </body>
18. </html>

```

La page reçoit dans son modèle un élément [erreurs] qui est un objet de type [ArrayList] d'objets [String], ces derniers étant des messages d'erreurs. Ils sont affichés par la boucle des lignes 13-15.

### 3.4.3.3 Le contrôleur de l'application

Le contrôleur [Application] est défini dans le paquetage [web] :



## Structure et initialisation du contrôleur

Le squelette du contrôleur [Application] est le suivant :

```

1. package web;
2.
3. ...
4.
5.
6. @SuppressWarnings("serial")
7. public class Application extends HttpServlet {
8.     // paramètres d'instance
9.     private String urlErreurs = null;
10.    private ArrayList<String> erreursInitialisation = new ArrayList<String>();
11.    private String[] paramètres = { "urlList", "urlEdit", "urlErreurs" };
12.    private Map params = new HashMap<String, String>();
13.
14.    // service
15.    private IService service = null;
16.
17.    // init
18.    @SuppressWarnings("unchecked")
19.    public void init() throws ServletException {
20.        // on récupère les paramètres d'initialisation de la servlet
21.        ServletConfig config = getServletConfig();
22.        // on traite les autres paramètres d'initialisation
23.        String valeur = null;
24.        for (int i = 0; i < paramètres.length; i++) {
25.            // valeur du paramètre
26.            valeur = config.getInitParameter(paramètres[i]);
27.            // paramètre présent ?
28.            if (valeur == null) {
29.                // on note l'erreur
30.                erreursInitialisation.add("Le paramètre [" + paramètres[i] + "] n'a pas été
initialisé");
31.            } else {
32.                // on mémorise la valeur du paramètre
33.                params.put(paramètres[i], valeur);
34.            }
35.        }
36.        // l'url de la vue [erreurs] a un traitement particulier
37.        urlErreurs = config.getInitParameter("urlErreurs");
38.        if (urlErreurs == null)
39.            throw new ServletException("Le paramètre [urlErreurs] n'a pas été initialisé");
40.        // configuration de l'application
41.        ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config.xml");
42.        // couche service
43.        service = (IService) ctx.getBean("service");
44.        // on vide la base
45.        clean();
46.        // on la remplit
47.        try {
48.            fill();
49.        } catch (ParseException e) {
50.            throw new ServletException(e);
51.        }
52.    }
53.
54.    // remplissage table
55.    public void fill() throws ParseException {
56.        // création personnes
57.        Personne p1 = new Personne("p1", "Paul", new SimpleDateFormat("dd/MM/yy").parse("31/01/2000"),
true, 2);
58.        Personne p2 = new Personne("p2", "Sylvie", new
SimpleDateFormat("dd/MM/yy").parse("05/07/2001"), false, 0);
59.        // qu'on sauvegarde
60.        service.saveArray(new Personne[] { p1, p2 });
61.    }
62.
63.    // suppression éléments de la table
64.    public void clean() {
65.        for (Personne p : service.getAll()) {
66.            service.deleteOne(p.getId());
67.        }
68.    }

```

```

69.
70. // GET
71. @SuppressWarnings("unchecked")
72. public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException,
    servletException {
73. ...
74. }
75.
76. // affichage liste des personnes
77. private void doListPersonnes(HttpServletRequest request, HttpServletResponse response) throws
    servletException, IOException {
78. ...
79. }
80.
81. // modification / ajout d'une personne
82. private void doEditPersonne(HttpServletRequest request, HttpServletResponse response) throws
    servletException, IOException {
83. ...
84. }
85.
86. // suppression d'une personne
87. private void doDeletePersonne(HttpServletRequest request, HttpServletResponse response) throws
    servletException, IOException {
88. ...
89. }
90.
91. // validation modification / ajout d'une personne
92. public void doValidatePersonne(HttpServletRequest request, HttpServletResponse response) throws
    servletException, IOException {
93. ...
94. }
95.
96. // affichage formulaire pré-rempli
97. private void showFormulaire(HttpServletRequest request, HttpServletResponse response, String
    erreurEdit) throws ServletException, IOException {
98. ...
99. }
100.
101.// post
102. public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException,
    ServletException {
103.     // on passe la main au GET
104.     doGet(request, response);
105. }
106.
107.

```

- lignes 21-34 : on récupère les paramètres attendus dans le fichier [web.xml].
- lignes 37-39 : le paramètre [urlErreurs] doit être obligatoirement présent car il désigne l'url de la vue [erreurs] capable d'afficher les éventuelles erreurs d'initialisation. S'il n'existe pas, on interrompt l'application en lançant une [ServletException] (ligne 39). Cette exception va remonter au serveur web et être gérée par la balise <error-page> du fichier [web.xml]. La vue [exception.jsp] est donc affichée :



Le lien [Retour à la liste] ci-dessus est inopérant. L'utiliser redonne la même réponse tant que l'application n'a pas été modifiée et rechargée. Il est utile pour d'autres types d'exceptions comme nous l'avons déjà vu.

- lignes 40-43 : exploitent le fichier de configuration Spring pour récupérer une référence sur la couche [service]. Après l'initialisation du contrôleur, les méthodes de celui-ci disposent d'une référence [service] sur la couche [service] (ligne 15) qu'elles vont utiliser pour exécuter les actions demandées par l'utilisateur. Celles-ci vont être interceptées par la méthode [doGet] qui va les faire traiter par une méthode particulière du contrôleur :

Url	Méthode HTTP	méthode contrôleur
/do/list	GET	doListPersonnes
/do/edit	GET	doEditPersonne
/do/validate	POST	doValidatePersonne
/do/delete	GET	doDeletePersonne

---

#### La méthode [doGet]

---

Cette méthode a pour but d'orienter le traitement des actions demandées par l'utilisateur vers la bonne méthode. Son code est le suivant :

```

1.// GET
2.@SuppressWarnings("unchecked")
3.public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException,
   ServletException {
4.
5.
6.// on vérifie comment s'est passée l'initialisation de la servlet
7. if (erreursInitialisation.size() != 0) {
8.     // on passe la main à la page d'erreurs
9.     request.setAttribute("erreurs", erreursInitialisation);
10.    getServletContext().getRequestDispatcher(urlErreurs).forward(request, response);
11.    // fin
12.    return;
13. }
14. // on récupère la méthode d'envoi de la requête
15. String méthode = request.getMethod().toLowerCase();
16. // on récupère l'action à exécuter
17. String action = request.getPathInfo();
18. // action ?
19. if (action == null) {
20.     action = "/list";
21. }
22. // exécution action
23. if (méthode.equals("get") && action.equals("/list")) {
24.     // liste des personnes
25.     doListPersonnes(request, response);
26.     return;
27. }
28. if (méthode.equals("get") && action.equals("/delete")) {
29.     // suppression d'une personne
30.     doDeletePersonne(request, response);
31.     return;
32. }
33. if (méthode.equals("get") && action.equals("/edit")) {
34.     // présentation formulaire ajout / modification d'une personne
35.     doEditPersonne(request, response);
36.     return;
37. }
38. if (méthode.equals("post") && action.equals("/validate")) {
39.     // validation formulaire ajout / modification d'une personne
40.     doValidatePersonne(request, response);
41.     return;
42. }
43. // autres cas
44. doListPersonnes(request, response);
45. }
```

- lignes 7-13 : on vérifie que la liste des erreurs d'initialisation est vide. Si ce n'est pas le cas, on fait afficher la vue [erreurs(erreurs)] qui va signaler la ou les erreurs.
- ligne 15 : on récupère la méthode [get] ou [post] que le client a utilisée pour faire sa requête.
- ligne 17 : on récupère la valeur du paramètre [action] de la requête.
- lignes 23-27 : traitement de la requête [GET /do/list] qui demande la liste des personnes.
- lignes 28-32 : traitement de la requête [GET /do/delete] qui demande la suppression d'une personne.
- lignes 33-37 : traitement de la requête [GET /do/edit] qui demande le formulaire de mise à jour d'une personne.
- lignes 38-42 : traitement de la requête [POST /do/validate] qui demande la validation de la personne mise à jour.
- ligne 44 : si l'action demandée n'est pas l'une des cinq précédentes, alors on fait comme si c'était [GET /do/list].

---

#### La méthode [doListPersonnes]

---

Cette méthode traite la requête [GET /do/list] qui demande la liste des personnes :

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a> <a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a> <a href="#">Supprimer</a>

Ajout

Son code est le suivant :

```
1. // affichage liste des personnes
2. private void doListPersonnes(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException {
3.     // le modèle de la vue [list]
4.     request.setAttribute("personnes", service.getAll());
5.     // affichage de la vue [list]
6.     getServletContext().getRequestDispatcher((String) params.get("urlList")).forward(request, response);
7. }
```

- ligne 4 : on demande à la couche [service] la liste des personnes du groupe et on met celle-ci dans le modèle sous la clé "personnes".
- ligne 6 : on fait afficher la vue [list.jsp] décrite page 184.

---

#### La méthode [doDeletePersonne]

---

Cette méthode traite la requête [GET /do/delete?id=XX] qui demande la suppression de la personne d'id=XX. L'url [/do/delete?id=XX] est celle des liens [Supprimer] de la vue [list.jsp] :

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	
13	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a> <a href="#">Supprimer</a>
14	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a> <a href="#">Supprimer</a>
15	1	Sophie	p3	13/03/1956	false	2	<a href="#">Modifier</a> <a href="#">Supprimer</a>

Ajout

dont le code est le suivant :

```
1. ...
2. <html>
3.   <head>
4.     <title>MVC - Personnes</title>
5.   </head>
6.   <body background=<c:url value="/ressources/standard.jpg"/>><br/>
7.   ...
8.     <c:forEach var="personne" items="${personnes}">
9.       <tr>
10.      ...
11.        <td><a href="

```

Ligne 12, on voit l'url [/do/delete?id=XX] du lien [Supprimer]. La méthode [doDeletePersonne] qui doit traiter cette url doit supprimer la personne d'id=XX puis faire afficher la nouvelle liste des personnes du groupe. Son code est le suivant :

```
1.// suppression d'une personne
2. private void doDeletePersonne(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException {
```

```

3.    // on récupère l'id de la personne
4.    int id = Integer.parseInt(request.getParameter("id"));
5.    // on supprime la personne
6.    service.deleteOne(id);
7.    // on redirige vers la liste des personnes
8.    response.sendRedirect("list");
9. }

```

- ligne 4 : l'url traitée est de la forme [/do/delete?id=XX]. On récupère la valeur [XX] du paramètre [id].
- ligne 6 : on demande à la couche [service] la suppression de la personne ayant l'id obtenu. Nous ne faisons aucune vérification. Si la personne qu'on cherche à supprimer n'existe pas, la couche [dao] lance une exception que laisse remonter la couche [service]. Nous ne la gérons pas non plus ici, dans le contrôleur. Elle remontera donc jusqu'au serveur web qui par configuration fera afficher la page [exception.jsp], décrite page 187 :



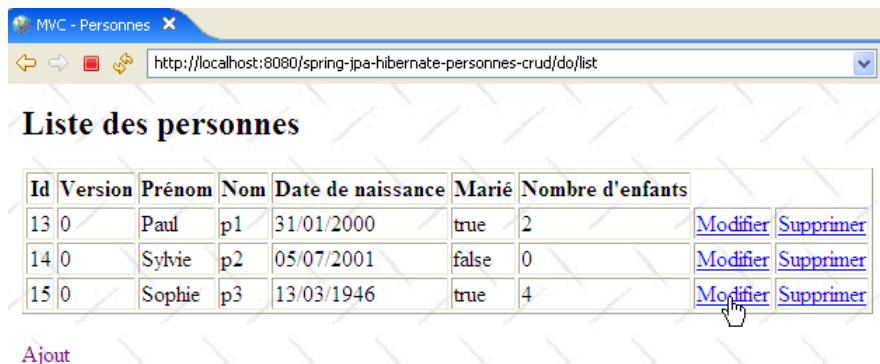
- ligne 9 : si la suppression a eu lieu (pas d'exception), on demande au client de se rediriger vers l'Url relative [list]. Comme celle qui vient d'être traitée est [/do/delete], l'Url de redirection sera [/do/list]. Le navigateur sera donc amené à faire un [GET /do/list] qui provoquera l'affichage de la liste des personnes.

---

La méthode [doEditPersonne]

---

Cette méthode traite la requête [GET /do/edit?id=XX] qui demande le formulaire de mise à jour de la personne d'id=XX. L'url [/do/edit?id=XX] est celle des liens [Modifier] et celui du lien [Ajout] de la vue [list.jsp] :



dont le code est le suivant :

```

1. ...
2. <html>
3.   <head>
4.     <title>MVC - Personnes</title>
5.   </head>
6.   <body background=<c:url value="/ressources/standard.jpg"/>><br/>
7. ...
8.     <c:forEach var="personne" items="${personnes}">
9.       <tr>
10. ...
11.         <td><a href="

```

Ligne 11, on voit l'url [/do/edit?id=XX] du lien [Modifier] et ligne 17, l'url [/do/edit?id=-1] du lien [Ajout]. La méthode [doEditPersonne] doit faire afficher le formulaire d'édition de la personne d'id=XX ou s'il s'agit d'un ajout présenter un formulaire vide.

- en [1] ci-dessus, le formulaire d'ajout et en [2] le formulaire de modification.

Le code de la méthode [doEditPersonne] est le suivant :

```

1.// modification / ajout d'une personne
2.private void doEditPersonne(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException {
3.    // on récupère l'id de la personne
4.    int id = Integer.parseInt(request.getParameter("id"));
5.    // ajout ou modification ?
6.    Personne personne = null;
7.    if (id != -1) {
8.        // modification - on récupère la personne à modifier
9.        personne = service.getOne(id);
10.       request.setAttribute("id", personne.getId());
11.       request.setAttribute("version", personne.getVersion());
12.    } else {
13.        // ajout - on crée une personne vide
14.        personne = new Personne();
15.        request.setAttribute("id", -1);
16.        request.setAttribute("version", -1);
17.    }
18.    // on la met l'objet [Personne] dans la session de l'utilisateur
19.    request.getSession().setAttribute("personne", personne);
20.    // et dans le modèle de la vue [edit]
21.    request.setAttribute("erreurEdit", "");
22.    request.setAttribute("prenom", personne.getPrenom());
23.    request.setAttribute("nom", personne.getNom());
24.    Date dateNaissance = personne.getDateNaissance();
25.    if (dateNaissance != null) {
26.        request.setAttribute("datenaissance", new
           SimpleDateFormat("dd/MM/yyyy").format(dateNaissance));
27.    } else {
28.        request.setAttribute("datenaissance", "");
29.    }
30.    request.setAttribute("marie", personne.isMarie());
31.    request.setAttribute("nbenfants", personne.getNbenfants());
32.    // affichage de la vue [edit]
33.    getServletContext().getRequestDispatcher((String) params.get("urlEdit")).forward(request,
   response);
34. }
```

- le GET a pour cible une url du type [/do/edit?id=XX]. Ligne 4, nous récupérons la valeur de [id]. Ensuite il y a deux cas :
  - id est différent de -1. Alors il s'agit d'une modification et il faut afficher un formulaire pré-rempli avec les informations de la personne à modifier. Ligne 9, cette personne est demandée à la couche [service].
  - id est égal à -1. Alors il s'agit d'un ajout et il faut afficher un formulaire vide. Pour cela, une personne vide est créée ligne 14.
  - dans les deux cas, les éléments [id, version] du modèle de la page [edit.jsp] décrite page 185 sont initialisés.
- l'objet [Personne] obtenu est placé dans le modèle de la page [edit.jsp]. Ce modèle comprend les éléments suivants [erreurEdit, id, version, prenom, erreurPrenom, nom, erreurNom, datenaissance, erreurDateNaissance, marie, nbenfants, erreurNbEnfants]. Ces éléments sont initialisés lignes 19-31 à l'exception de ceux dont la valeur est la chaîne vide [erreurPrenom, erreurNom, erreurDateNaissance, erreurNbEnfants]. On sait qu'en leur absence dans le modèle, la bibliothèque JSTL affichera une chaîne vide pour leur valeur. Bien que l'élément [erreurEdit] ait également pour valeur une chaîne vide, il est néanmoins initialisé car un test est fait sur sa valeur dans la page [edit.jsp].

- une fois le modèle prêt, le contrôle est passé à la page [edit.jsp], ligne 33, qui va générer la vue [edit].

#### La méthode [doValidatePersonne]

Cette méthode traite la requête [POST /do/validate] qui valide le formulaire de mise à jour. Ce POST est déclenché par le bouton [Valider] :

Id	-1
Version	-1
Prénom	Sophie
Nom	p3
Date de naissance (JJ/MM/AAAA)	13/03/1946
Marié	<input checked="" type="radio"/> Oui <input type="radio"/> Non
Nombre d'enfants	4

**Valider** **Annuler**

Rappelons les éléments de saisie du formulaire HTML de la vue ci-dessus :

```

1. <form method="post" action="Annuler</a>
24. </form>
```

La requête POST contient les paramètres [prenom, nom, datenaissance, marie, nbenfants, id] et est postée à l'url [/do/validate] (ligne 1). Elle est traitée par la méthode [doValidatePersonne] suivante :

```

1.// validation modification / ajout d'une personne
2.public void doValidatePersonne(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException {
3.    // on récupère les éléments postés
4.    boolean formulaireErroné = false;
5.    boolean erreur;
6.    // le prénom
7.    String prenom = request.getParameter("prenom").trim();
8.    // prénom valide ?
9.    if (prenom.length() == 0) {
10.        // on note l'erreur
11.        request.setAttribute("erreurPrenom", "Le prénom est obligatoire");
12.        formulaireErroné = true;
13.    }
14.    // le nom
15.    String nom = request.getParameter("nom").trim();
16.    // prénom valide ?
17.    if (nom.length() == 0) {
18.        // on note l'erreur

```

```

19.     request.setAttribute("erreurNom", "Le nom est obligatoire");
20.     formulaireErroné = true;
21. }
22. // la date de naissance
23. Date datenaissance = null;
24. try {
25.     datenaissance = new
SimpleDateFormat("dd/MM/yyyy").parse(request.getParameter("datenaissance").trim());
26. } catch (ParseException e) {
27.     // on note l'erreur
28.     request.setAttribute("erreurDateNaissance", "Date incorrecte");
29.     formulaireErroné = true;
30. }
31. // état marital
32. boolean marie = Boolean.parseBoolean(request.getParameter("marie").trim());
33. // nombre d'enfants
34. int nbenfants = 0;
35. erreur = false;
36. try {
37.     nbenfants = Integer.parseInt(request.getParameter("nbefants").trim());
38.     if (nbefants < 0) {
39.         erreur = true;
40.     }
41. } catch (NumberFormatException ex) {
42.     // on note l'erreur
43.     erreur = true;
44. }
45. // nombre d'enfants erroné ?
46. if (erreur) {
47.     // on signale l'erreur
48.     request.setAttribute("erreurNbEnfants", "Nombre d'enfants incorrect");
49.     formulaireErroné = true;
50. }
51. // id de la personne
52. int id = Integer.parseInt(request.getParameter("id"));
53. // le formulaire est-il erroné ?
54. if (formulaireErroné) {
55.     // on réaffiche le formulaire avec les messages d'erreurs
56.     showFormulaire(request, response, "");
57.     // fini
58.     return;
59. }
60. // le formulaire est correct - on met à jour la personne qui a été placée dans la session
61. // avec les informations envoyées par le client
62. Personne personne = (Personne)request.getSession().getAttribute("personne");
63. personne.setDatenaissance(datenaissance);
64. personne.setMarie(marie);
65. personne.setNbenfants(nbenfants);
66. personne.setNom(nom);
67. personne.setPrenom(prenom);
68. // persistance
69. try {
70.     if (id == -1) {
71.         // création
72.         service.saveOne(personne);
73.     } else {
74.         // mise à jour
75.         service.updateOne(personne);
76.     }
77. } catch (DaoException ex) {
78.     // on réaffiche le formulaire avec le message de l'erreur survenue
79.     showFormulaire(request, response, ex.getMessage());
80.     // fini
81.     return;
82. }
83. // on redirige vers la liste des personnes
84. response.sendRedirect("list");
85. }
86.
87. // affichage formulaire pré-rempli
88. private void showFormulaire(HttpServletRequest request, HttpServletResponse response, String
erreurEdit) throws ServletException, IOException {
89.     // on prépare le modèle de la vue [edit]
90.     request.setAttribute("erreurEdit", erreurEdit);
91.     request.setAttribute("id", request.getParameter("id"));
92.     request.setAttribute("version", request.getParameter("version"));
93.     request.setAttribute("prenom", request.getParameter("prenom").trim());
94.     request.setAttribute("nom", request.getParameter("nom").trim());
95.     request.setAttribute("datenaissance", request.getParameter("datenaissance").trim());
96.     request.setAttribute("marie", request.getParameter("marie"));
97.     request.setAttribute("nbefants", request.getParameter("nbefants").trim());
98.     // affichage de la vue [edit]
99.     getServletContext().getRequestDispatcher((String) params.get("urlEdit")).forward(request,
response);
100. }

```

- lignes 7-13 : le paramètre [prenom] de la requête POST est récupéré et sa validité vérifiée. S'il s'avère incorrect, l'élément [erreurPrenom] est initialisé avec un message d'erreur et placé dans les attributs de la requête.
- lignes 15-21 : on opère de façon similaire pour le paramètre [nom]
- lignes 23-30 : on opère de façon similaire pour le paramètre [datenaissance]
- ligne 32 : on récupère le paramètre [marie]. On ne fait pas de vérification sur sa validité parce qu'à priori il provient de la valeur d'un bouton radio. Ceci dit, rien n'empêche un programme de faire un [POST /.../do/validate] accompagné d'un paramètre [marie] fantaisiste. Nous devrions donc tester la validité de ce paramètre. Ici, on se repose sur notre gestion des exceptions qui provoquent l'affichage de la page [exception.jsp] si le contrôleur ne les gère pas lui-même. Si donc, la conversion du paramètre [marie] en booléen échoue ligne 32, une exception en sortira qui aboutira à l'envoi de la page [exception.jsp] au client. Ce fonctionnement nous convient.
- lignes 34-50 : on récupère le paramètre [nbenfants] et on vérifie sa valeur
- ligne 52 : on récupère le paramètre [id] sans vérifier sa valeur
- lignes 54-59 : si le formulaire est erroné, il est réaffiché avec les messages d'erreurs construits précédemment
- lignes 62-67 : s'il est valide, on construit un nouvel objet [Personne] avec les éléments du formulaire
- lignes 69-82 : la personne est sauvegardée. La sauvegarde peut échouer. Dans un cadre multi-utilisateurs, la personne à modifier a pu être supprimée ou bien déjà modifiée par quelqu'un d'autre. Dans ce cas, la couche [dao] va lancer une exception qu'on gère ici.
- ligne 84 : s'il n'y a pas eu d'exception, on redirige le client vers l'url [/do/list] pour lui présenter le nouvel état du groupe.
- ligne 79 : s'il y a eu exception lors de la sauvegarde, on redemande le réaffichage du formulaire initial en lui passant le message d'erreur de l'exception (3ième paramètre).

La méthode [showFormulaire] (lignes 88-97) construit le modèle nécessaire à la page [edit.jsp] avec les valeurs saisies (`request.getParameter("...")`). On se rappelle que les messages d'erreurs ont déjà été placés dans le modèle par la méthode [doValidatePersonne]. La page [edit.jsp] est affichée ligne 99.

### 3.4.4 Les tests de l'application web

Un certain nombre de tests ont été présentés au paragraphe 3.4.1, page 179. Nous invitons le lecteur à les rejouer. Nous montrons ici d'autres copies d'écran qui illustrent les cas de conflits d'accès aux données dans un cadre multi-utilisateurs :

[Firefox] sera le navigateur de l'utilisateur U1. Celui-ci demande l'url [<http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list>] :

<b>Id</b>	<b>Version</b>	<b>Prénom</b>	<b>Nom</b>	<b>Date de naissance</b>	<b>Marié</b>	<b>Nombre d'enfants</b>	
22	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a> <a href="#">Supprimer</a>
23	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a> <a href="#">Supprimer</a>

[Ajout](#)

[IE7] sera le navigateur de l'utilisateur U2. Celui-ci demande la même Url :

**MVC - Personnes - Windows Internet Explorer**

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list

MVC - Personnes

### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
22	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
23	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

L'utilisateur U1 entre en modification de la personne [p2] :

**MVC - Personnes - Mozilla Firefox**

Eichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/edit

MVC - Personnes

### Ajout/Modification d'une personne

Id	23
Version	0
Prénom	Sylvie
Nom	p2
Date de naissance (JJ/MM/AAAA)	05/07/2001
Marié	<input type="radio"/> Oui <input checked="" type="radio"/> Non
Nombre d'enfants	0

[Valider](#) [Annuler](#)

L'utilisateur U2 fait de même :

**MVC - Personnes - Windows Internet Explorer**

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/edit

MVC - Personnes

### Ajout/Modification d'une personne

Id	23
Version	0
Prénom	Sylvie
Nom	p2
Date de naissance (JJ/MM/AAAA)	05/07/2001
Marié	<input type="radio"/> Oui <input checked="" type="radio"/> Non
Nombre d'enfants	0

[Valider](#) [Annuler](#)

L'utilisateur U1 fait des modifications et valide :

MVC - Personnes - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/ed

MVC - Personnes

### Ajout/Modification d'une personne

Id	23
Version	0
Prénom	Sylvie
Nom	p2
Date de naissance (JJ/MM/AAAA)	05/07/2001
Marié	<input checked="" type="radio"/> Oui <input type="radio"/> Non
Nombre d'enfants	2

[Valider](#) [Annuler](#)

validation

L'utilisateur U2 fait de même :

MVC - Personnes - Windows Internet Explorer

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/ed

MVC - Personnes

### Ajout/Modification d'une personne

Id	23
Version	0
Prénom	Sylvie
Nom	LEMARCHAND
Date de naissance (JJ/MM/AAAA)	05/07/2001
Marié	<input type="radio"/> Oui <input checked="" type="radio"/> Non
Nombre d'enfants	0

[Valider](#) [Annuler](#)

validation

L'utilisateur U2 revient à la liste des personnes avec le lien [Retour à la liste] du formulaire :

MVC - Personnes - Mozilla Firefox

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list

MVC - Personnes

### Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
22	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
23	1	Sylvie	p2	05/07/2001	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

réponse du serveur

MVC - Personnes - Windows Internet Explorer

http://localhost:8080/

MVC - Personnes

### MVC - personnes

L'exception suivante s'est produite : org.hibernate.StaleObjectStateException:  
Row was updated or deleted by another transaction (or unsaved-value mapping  
was incorrect): [entites.Personne#23]

[Retour à la liste](#)

le conflit de version a été détecté



MVC - Personnes - Windows Internet Explorer

http://localhost:8080/spring-jpa-hibernate-personnes

MVC - Personnes

### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
22	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
23	1	Sylvie	p2	05/07/2001	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

Il trouve la personne [Lemarchand] telle que U1 l'a modifiée (mariée, 2 enfants). Le n° de version de p2 a changé. Maintenant U2 supprime [p2] :

MVC - Personnes - Windows Internet Explorer

http://localhost:8080/spring-jpa-hibernate-personnes

MVC - Personnes

### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
22	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
23	1	Sylvie	p2	05/07/2001	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

- Suppression

MVC - Personnes - Windows Internet Explorer

http://localhost:8080/spring-jpa-hibernate-personnes

MVC - Personnes

### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
22	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

- réponse du serveur

U1 a toujours sa propre liste et veut modifier [p2] de nouveau :

MVC - Personnes - Mozilla Firefox

Eichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list

MVC - Personnes

### Liste des personnes

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
22	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
23	1	Sylvie	p2	05/07/2001	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

- modification

MVC - Personnes - Mozilla Firefox

Eichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/edit?id=23

MVC - Personnes

L'exception suivante s'est produite : La personne n° [23] n'existe pas

[Retour à la liste](#)

- réponse du serveur : il n'a pas trouvé [p2]

U1 utilise le lien [Retour à la liste] pour voir de quoi il retourne :

MVC - Personnes - Mozilla Firefox

Eichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/spring-jpa-hibernate-personnes-crud/do/list

MVC - Personnes

### Liste des personnes

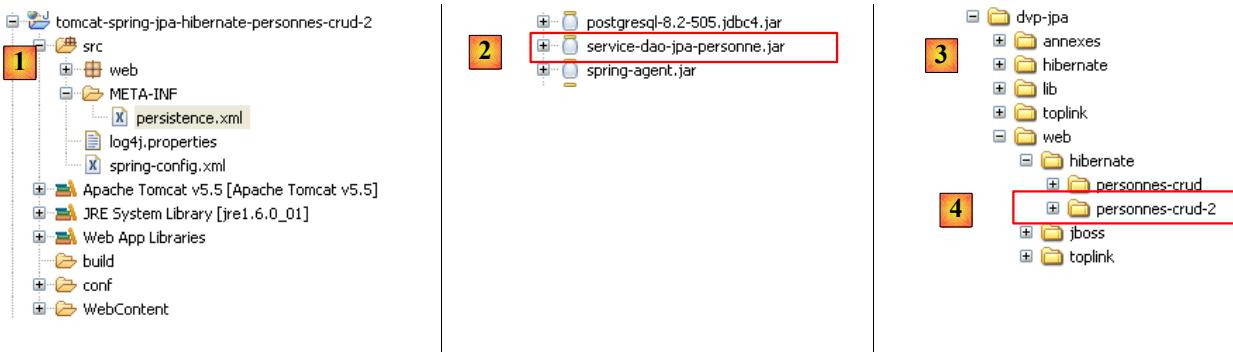
ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
22	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

Il découvre qu'effectivement [p2] ne fait plus partie de la liste...

### 3.4.5 Version 2

Nous modifions légèrement la version précédente pour utiliser les archives des couches [service, dao, jpa] et non plus leurs codes source :



- en [1] : le nouveau projet Eclipse. On notera la disparition des paquetages [service, dao, entites]. Ceux-ci ont été encapsulés dans l'archive [service-dao-jpa-personne.jar] [2] placée dans [WEB-INF/lib].
- le dossier du projet est en [4]. On l'importera.

Il n'y a rien de plus à faire. Lorsque la nouvelle application web est lancée et qu'on demande la liste des personnes, on reçoit la réponse suivante :

MVC - personnes

L'exception suivante s'est produite : org.hibernate.hql.ast.QuerySyntaxException: Personne is not mapped  
[select p from Personne p]

[Retour à la liste](#)

Hibernate ne trouve pas l'entité [Personne]. Pour résoudre ce problème, on est obligés de déclarer explicitement dans [persistence.xml] les entités gérées :

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0"
3.   xmlns="http://java.sun.com/xml/ns/persistence"
4.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
6.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
7.     <class>entites.Personne</class>
8.   </persistence-unit>
9. </persistence>
```

- ligne 7 : l'entité *Personne* est déclarée.

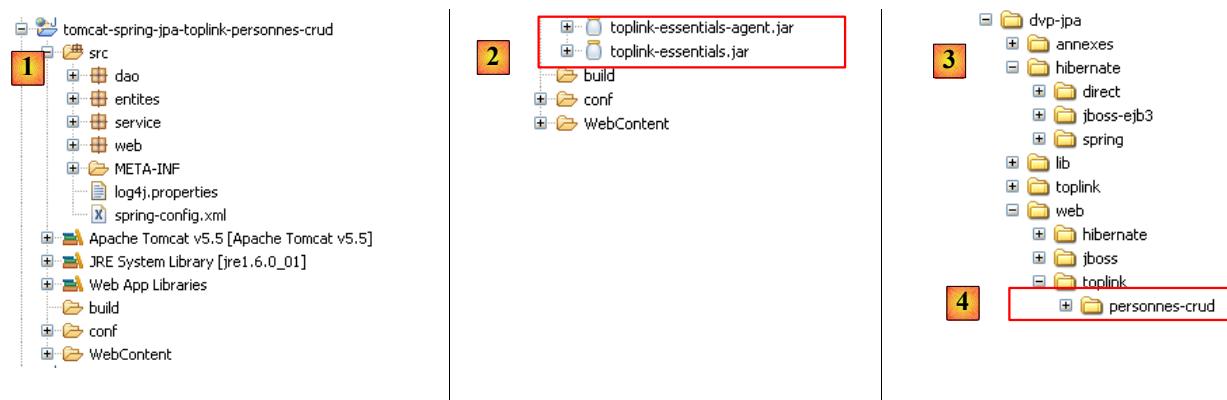
Ceci fait, l'exception disparaît :

**Liste des personnes**

ID	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
32	0	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
33	0	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

### 3.4.6 Changer d'implémentation JPA



- en [1] : le nouveau projet Eclipse
- en [2] : les bibliothèques Toplink ont remplacé les bibliothèques Hibernate
- le dossier du projet est en [4]. On l'importera.

Changer d'implémentation JPA n'implique que quelques changements dans le fichier [spring-config.xml]. Rien d'autre ne change. Les changements amenés dans le fichier [spring-config.xml] ont été expliqués au paragraphe 3.1.9, page 162 :

```
|1.  <?xml version="1.0" encoding="UTF-8"?>
```

```

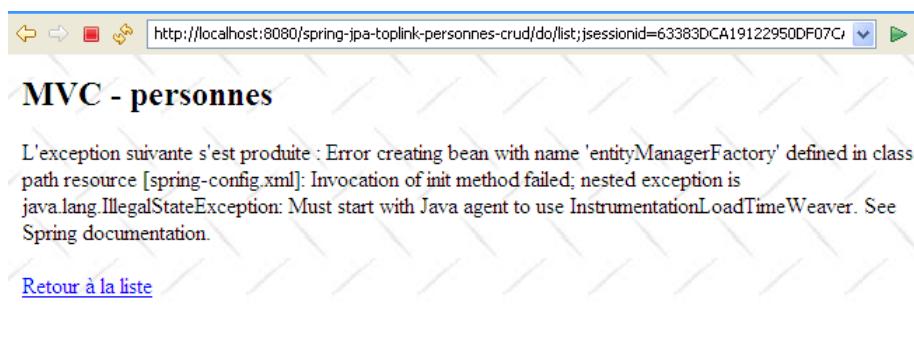
2.
3. <!-- la JVM doit être lancée avec l'argument -javaagent:C:\data\2006-2007\eclipse\dvp-
   jpa\lib\spring\spring-agent.jar
4.   (à remplacer par le chemin exact de spring-agent.jar) -->
5.
6. <beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7. ...
8.   <bean id="entityManagerFactory"
   class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
9.     <property name="dataSource" ref="dataSource" />
10.    <property name="jpaVendorAdapter">
11.      <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter">
12. ...
13.      <property name="databasePlatform"
   value="oracle.toplink.essentials.platform.database.MySQL4Platform" />
14. ...
15. </bean>
16. ...
17.</beans>

```

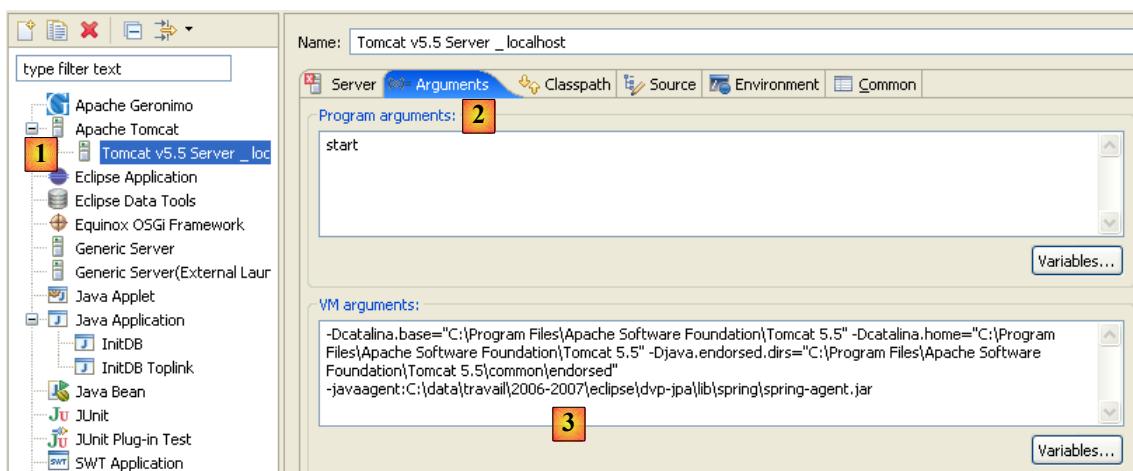
Peu de lignes doivent être changées pour passer d'Hibernate à Toplink :

- ligne 11 : l'implémentation JPA est désormais faite par Toplink
- ligne 13 : la propriété [databasePlatform] a une autre valeur qu'avec Hibernate : le nom d'une classe propre à Toplink. Où trouver ce nom a été expliqué au paragraphe 2.1.15.2, page 58.

C'est tout. On notera la facilité avec laquelle on peut changer de SGBD ou d'implémentation JPA avec Spring. On n'a quand même pas tout à fait fini. Lorsqu'on exécute l'application, on a une exception :



On reconnaîtra là un problème rencontré et décrit au paragraphe 3.1.9, page 162. Il est résolu en lançant la JVM avec un agent Spring. Pour cela, on modifie la configuration de lancement de Tomcat :



- en [1] : on a pris l'option [Run / Run...] pour modifier la configuration de Tomcat
- en [2] : on a sélectionné l'onglet [Arguments]
- en [3] : on a rajouté le paramètre **-javaagent** tel qu'il a été décrit au paragraphe 3.1.9, page 162.

Ceci fait, on peut demander la liste des personnes :

The screenshot shows a Java development environment with several tabs open: Application.java, MVC - Personnes (highlighted), Dao.java, Service.java, and spring-config.xml. Below the tabs, a browser window is displayed with the URL <http://localhost:8080/spring-jpa-toplink-personnes-crud/do/list;jsessionid=63383DCA19122950DF07C>. The browser title bar also shows the MVC - Personnes tab.

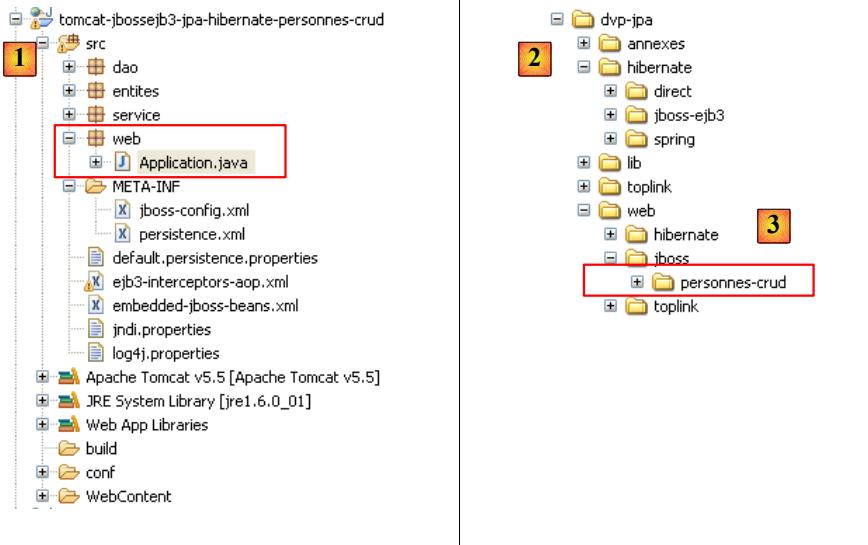
**Liste des personnes**

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	Modifier	Supprimer
52	1	Paul	p1	31/01/2000	true	2	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
53	1	Sylvie	p2	05/07/2001	false	0	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajout](#)

### 3.5 Autres exemples

Nous aurions voulu montrer un exemple web où le conteneur Spring était remplacé par le conteneur Jboss Ejb3 étudié au paragraphe 3.2, page 167 :



- en [1] : le projet Eclipse
- en [3] : son emplacement dans le dossier des exemples. On l'importera.

Nous avons repris la configuration [jboss-config.xml, persistence.xml] décrite au paragraphe 3.2, page 167, puis avons modifié la méthode [init] du contrôleur [Application.java] de la façon suivante :

```

1.// init
2.@SuppressWarnings("unchecked")
3.public void init() throws ServletException {
4.    try {
5.        // on récupère les paramètres d'initialisation de la servlet
6.        ServletConfig config = getServletConfig();
7.        // on traite les autres paramètres d'initialisation
8.        String valeur = null;
9.        for (int i = 0; i < paramètres.length; i++) {
10.            // valeur du paramètre
11.            valeur = config.getInitParameter(paramètres[i]);
12.            // paramètre présent ?
13.            if (valeur == null) {
14.                // on note l'erreur
15.                erreursInitialisation.add("Le paramètre [" + paramètres[i] + "] n'a pas été
initialisé");
16.            } else {
17.                // on mémorise la valeur du paramètre
18.                params.put(paramètres[i], valeur);
19.            }
20.        }
21.        // l'url de la vue [erreurs] a un traitement particulier
22.        urlErreurs = config.getInitParameter("urlErreurs");
23.        if (urlErreurs == null)
24.            throw new ServletException("Le paramètre [urlErreurs] n'a pas été initialisé");
25.        // configuration de l'application
26.        // on démarre le conteneur EJB3 JBoss
27.        // les fichiers de configuration ejb3-interceptors-aop.xml et embedded-jboss-beans.xml sont
exploités
28.        EJB3StandaloneBootstrap.boot(null);
29.
30.        // Crération des beans propres à l'application
31.        EJB3StandaloneBootstrap.deployXmlResource("META-INF/jboss-config.xml");
32.
33.        // on déploie tous les EJB trouvés dans le classpath de l'application
34.        //EJB3StandaloneBootstrap.scanClasspath("WEB-INF/classes".replace("/", File.separator));
35.        EJB3StandaloneBootstrap.scanClasspath();
36.
37.        // On initialise le contexte JNDI. Le fichier jndi.properties est exploité
38.        InitialContext initialContext = new InitialContext();
39.
40.        // instanciation couche service
41.        service = (IService) initialContext.lookup("Service/local");
42.        // on vide la base

```

```

43.     clean();
44.     // on la remplit
45.     fill();
46. } catch (Exception e) {
47.     throw new ServletException(e);
48. }
49. }

```

- lignes 28-38 : on démarre le conteneur Ejb3. Celui-ci remplace le conteneur Spring.
- ligne 41 : on demande une référence sur la couche [service] de l'application.

A priori, ce sont les seules modifications à apporter. A l'exécution, on a l'erreur suivante :



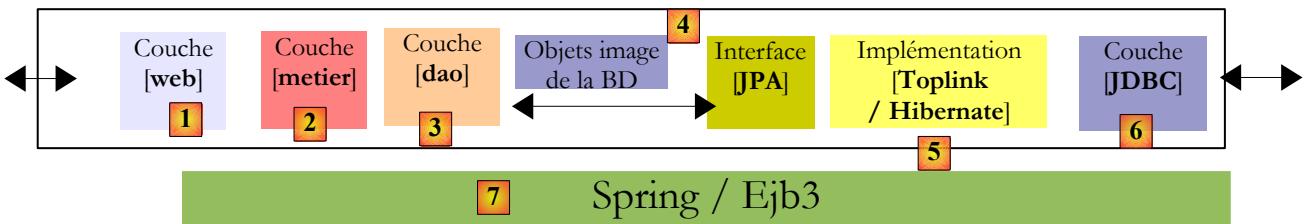
Je n'ai pas été capable de comprendre où était exactement le problème. L'exception rapportée par Tomcat semble dire que l'objet nommé "TransactionManager" a été demandé au service JNDI et que celui-ci ne le connaissait pas. Je laisse aux lecteurs le soin d'apporter une solution à ce problème. Si une solution est trouvée, elle sera intégrée au document.

## 4 Conclusion

Ce tutoriel "Persistance Java 5 par la pratique" a apporté des exemples pour la construction d'architectures multi-couches où

- l'accès aux données est contrôlé par une couche JPA
- les couches [service] et [dao] s'exécutent ausein d'un conteneur Spring ou Ejb3

Nous avons terminé par un exemple d'application web rassemblant toutes les couches :



Bien que ce tutoriel soit fort long, il n'est cependant pas exhaustif :

- des points JPA n'ont pas été abordés (cache des entités, traduction de la notion objet d'héritage dans un schéma de SGBD, ...)
- des difficultés ont été rencontrées, signalées mais pas résolues, notamment dans l'utilisation du conteneur Ejb3.

Le lecteur considérera donc ce document seulement comme un point de départ pour ses premiers pas avec la persistance Java 5.

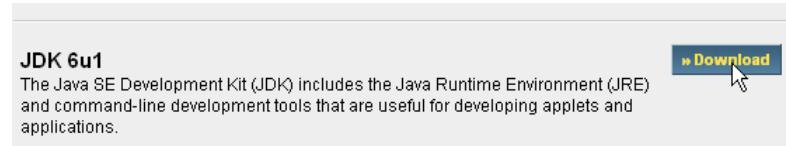


## 5 ANNEXES

Nous décrivons ici l'installation et une utilisation basique des outils utilisés dans le document "Persistance Java 5 par la pratique". Les informations données ci-dessous sont celles disponibles en mai 2007. Elles seront rapidement obsolètes. Lorsque ce sera le cas, le lecteur sera invité à suivre des démarches analogues mais qui ne seront pas identiques. Les installations ont été faites sur une machine Windows XP Professionnel.

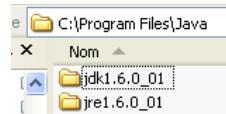
### 5.1 Java

Nous utiliserons la dernière version de Java disponible chez Sun [<http://www.sun.com>]. Les téléchargements sont accessibles via l'url [<http://java.sun.com/javase/downloads/index.jsp>] :



Windows Platform - Java(TM) SE Development Kit 6 Update 1			
<input checked="" type="checkbox"/>			
<input type="checkbox"/>	<a href="#">Windows Offline Installation, Multi-language</a>	jdk-6u1-windows-i586-p.exe	56.06 MB
<input type="checkbox"/>	<a href="#">Windows Online Installation, Multi-language</a>	jdk-6u1-windows-i586-p-iflw.exe	361.65 KB

Lancer l'installation du JDK à partir du fichier téléchargé. Par défaut, Java est installé dans [C:\Program Files\Java] :



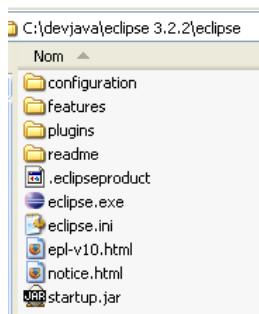
### 5.2 Eclipse

#### 5.2.1 Installation de base

Eclipse est un IDE Jdisponible à l'url [<http://www.eclipse.org/>] et peut être téléchargé à l'url [<http://www.eclipse.org/downloads/>]. Nous téléchargeons ci-dessous Eclipse 3.2.2 :



Une fois le zip téléchargé, on le décomprime dans un dossier du disque :



Nous appellerons par la suite <eclipse>, le dossier d'installation d'Eclipse, ci-dessus [C:\devjava\ eclipse 3.2.2\ eclipse]. [eclipse.exe] est l'exécutable et [eclipse.ini] le fichier de configuration de celui-ci. Regardons le contenu de celui-ci :

- ```
1. -vmargs  
2. -Xms40m  
3. -Xmx256m
```

Ces arguments sont utilisés lors du lancement d'Eclipse de la façon suivante :

```
eclipse.exe -vmargs -Xms40m -Xmx256m
```

On arrive au même résultat que celui obtenu avec le fichier .ini, en créant un raccourci qui lancerait Eclipse avec ces mêmes arguments. Explicitons ceux-ci :

1. **-vmargs** : indique que les arguments qui suivent sont destinés à la machine virtuelle Java qui va exécuter Eclipse. Eclipse est une application Java.
2. **-Xms40m** : ?
3. **-Xmx256m** : fixe la taille mémoire en Mo allouée à la machine virtuelle Java (JVM) qui exécute Eclipse. Par défaut, cette taille est de 256 Mo comme montré ici. Si la machine le permet, 512 Mo est préférable.

Ces arguments sont passés à la JVM qui va exécuter Eclipse. La JVM est représentée par un fichier [java.exe] ou [javaw.exe]. Comment celui-ci est-il trouvé ? En fait, il est cherché de différentes façons :

- dans le PATH de l'OS
- dans le dossier <JAVA\_HOME>/jre/bin où JAVA\_HOME est une variable système définissant le dossier racine d'un JDK.
- à un emplacement passé en argument à Eclipse sous la forme -vm <chemin>\javaw.exe

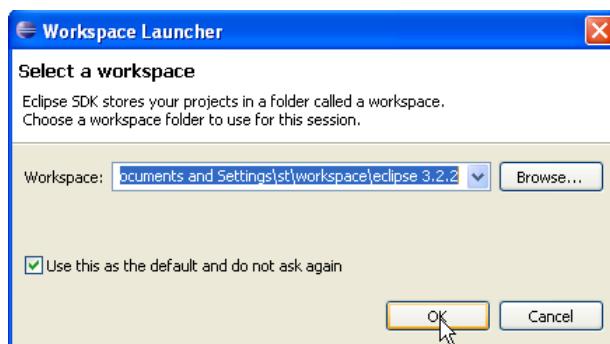
Cette dernière solution est préférable car les deux autres sont sujettes aux aléas d'installations ultérieures d'applications qui peuvent soit changer le PATH de l'OS, soit changer la variable JAVA\_HOME.

Nous créons donc le raccourci suivant :



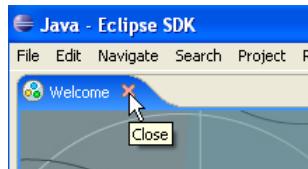
cible      |<eclipse>\eclipse.exe" -vm "C:\Program Files\Java\jre1.6.0\_01\bin\javaw.exe" -vmargs -Xms40m -Xmx512m  
Démarrer dans | dossier <eclipse> d'installation d'Eclipse

Ceci fait, lançons Eclipse via ce raccourci. On obtient une première boîte de dialogue :



Un [workspace] est un espace de travail. Acceptons les valeurs par défaut proposées. Par défaut, les projets Eclipse créés le seront dans le dossier <workspace> spécifié dans cette boîte de dialogue. Il y a moyen de contourner ce comportement. C'est ce que nous ferons systématiquement. Aussi la réponse donnée à cette boîte de dialogue n'est-elle pas importante.

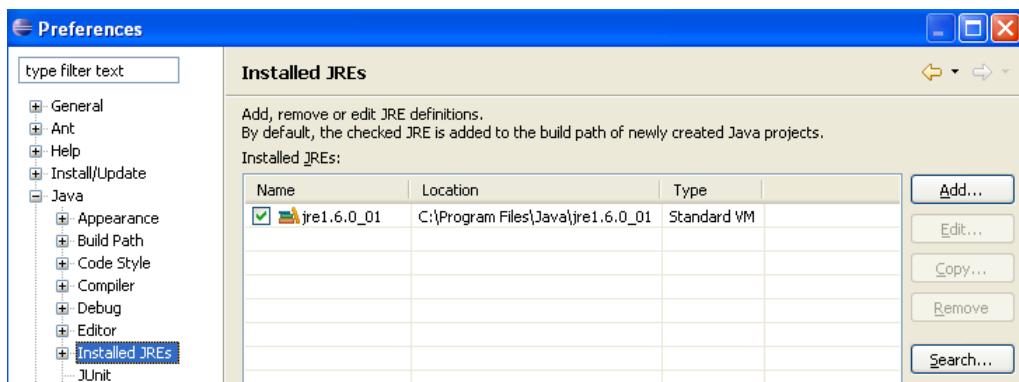
Passée cette étape, l'environnement de développement Eclipse est affiché :



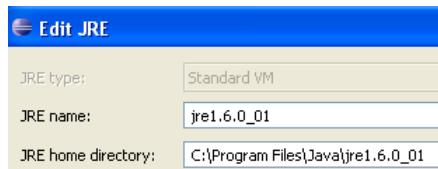
Nous fermons la vue [Welcome] comme suggéré ci-dessus :



Avant de créer un projet Java, nous allons configurer Eclipse pour indiquer le JDK à utiliser pour compiler les projets Java. Pour cela, nous prenons l'option [Window / Preferences / Java / Installed JREs] :



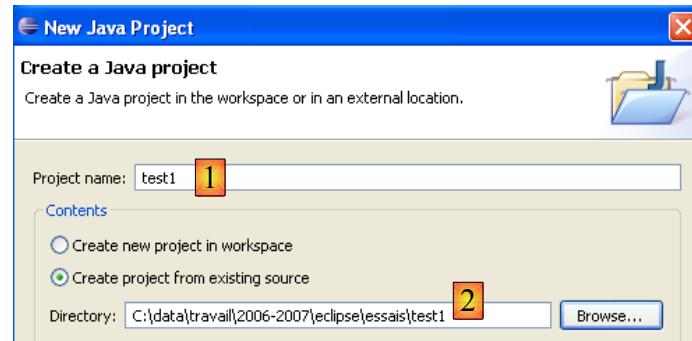
Normalement, le JRE (Java Runtime Environment) qui a été utilisé pour lancer Eclipse lui-même doit être présent dans la liste des JRE. Ce sera le seul normalement. Il est possible d'ajouter des JRE avec le bouton [Add]. Il faut alors désigner la racine du JRE. Le bouton [Search] va lui, lancer une recherche de JREs sur le disque. C'est un bon moyen de savoir où on en est dans les JREs qu'on installe puis qu'on oublie de désinstaller lorsqu'on passe à une version plus récente. Ci-dessus, le JRE coché est celui qui sera utilisé pour compiler et exécuter les projets Java. C'est celui qui a été installé au paragraphe 5.1, page 209 et qui a également servi à lancer Eclipse. Un double clic dessus donne accès à ses propriétés :



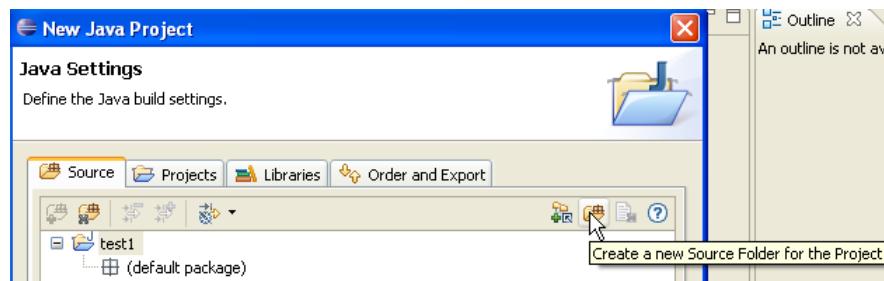
Maintenant, créons un projet Java [File / New / Project] :

A screenshot of the Eclipse interface. On the left, the "File" menu is open with "New" selected, and the "Project..." option is highlighted. On the right, a "New Project" dialog is open with "Select a wizard" set to "Create a Java project". Under "Wizards:", "Java Project" is selected and highlighted. Other options include "Java Project from Existing Ant Buildfile" and "Plug-in Project".

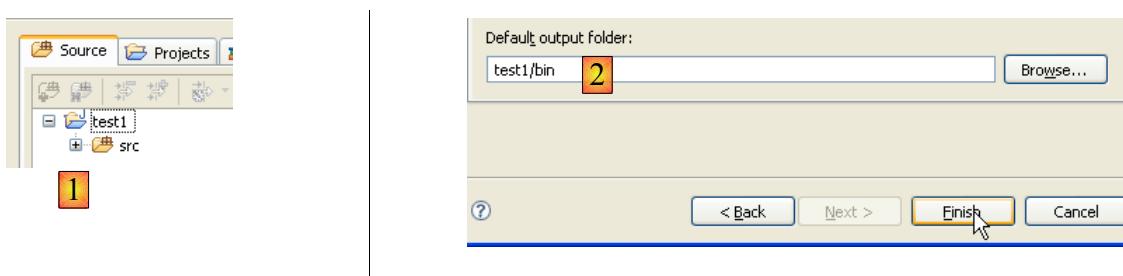
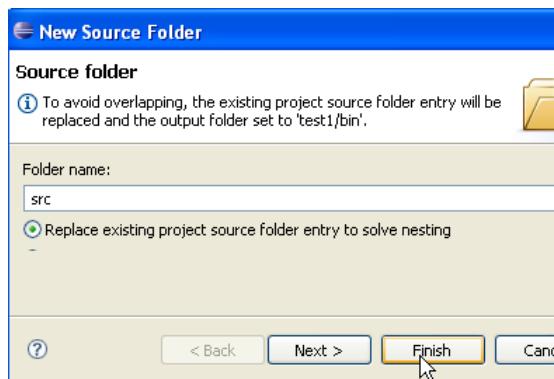
Choisir [Java Project], puis [Next] ->



Dans [2], nous indiquons un dossier vide dans lequel sera installé le projet Java. Dans [1], nous donnons un nom au projet. Il n'a pas à porter le nom de son dossier comme pourrait le laisser croire l'exemple ci-dessus. Ceci fait, on utilise le bouton [Next] pour passer à la page suivante de l'assistant de création :

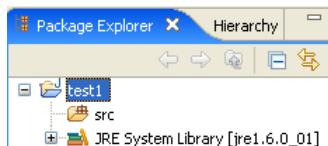


Ci-dessus, nous créons un dossier spécial dans le projet pour y stocker les fichiers source (.java) :

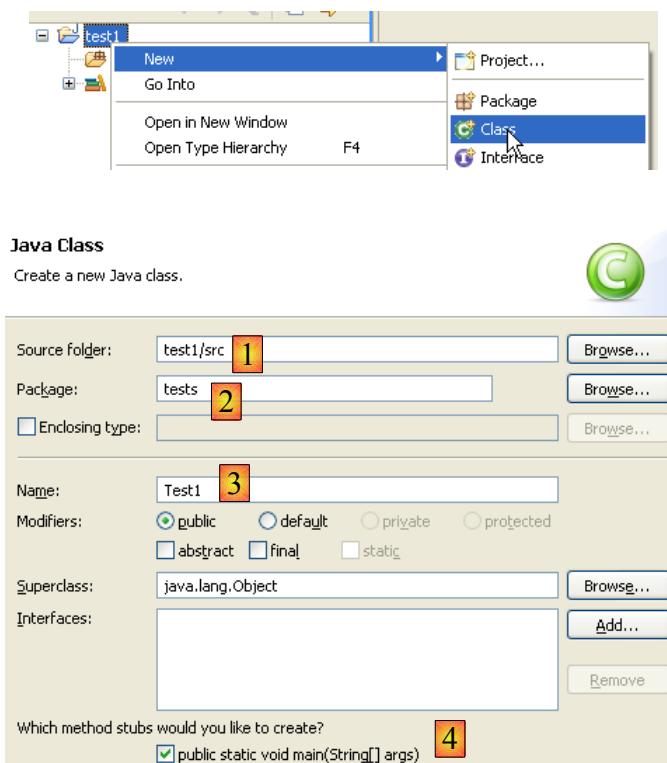


- en [1], nous voyons le dossier [src] dans lequel seront rangés les sources .java
- en [2], nous voyons le dossier [bin] dans lequel seront rangés les fichiers compilés .class

Nous terminons l'assistant avec [Finish]. Nous avons alors un squelette de projet Java :



Cliquons droit sur le projet [test1] pour créer une classe Java :



1. dans [1], le dossier où sera créée la classe. Eclipse propose par défaut le dossier du projet courant.
2. dans [2], le paquetage dans lequel sera placée la classe
3. dans [3], le nom de la classe
4. dans [4], nous demandons à ce que la méthode statique [main] soit générée

Nous validons l'assistant par [Finish]. Le projet est alors enrichi d'une classe :



Eclipse a généré le squelette de la classe. Celui-ci est obtenu en double-cliquant sur [Test1.java] ci-dessus :

```

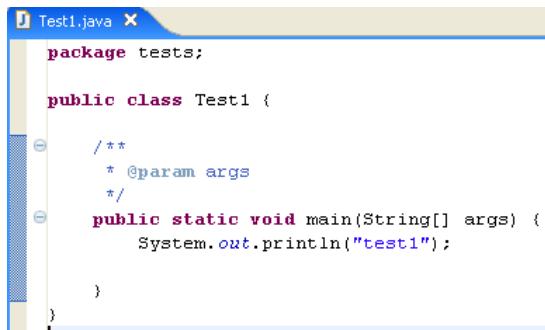
Test1.java x
package tests;

public class Test1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}

```

Nous modifions le code ci-dessus de la façon suivante :

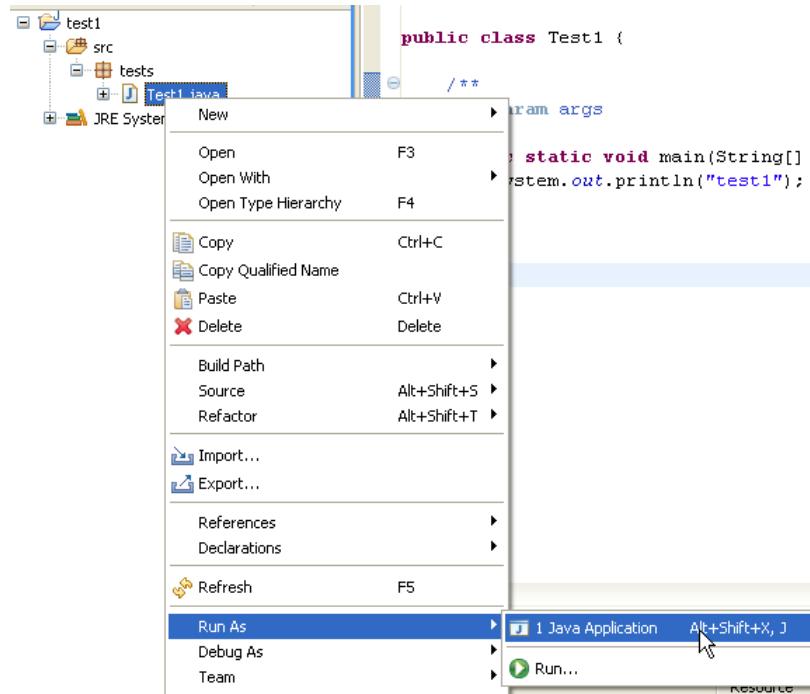


```
package tests;

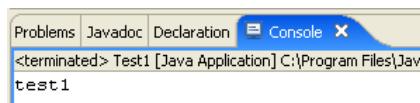
public class Test1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("test1");
    }
}
```

Nous exécutons le programme [Test1.java] : [clic droit sur Test1.java -> Run As -> Java Application]

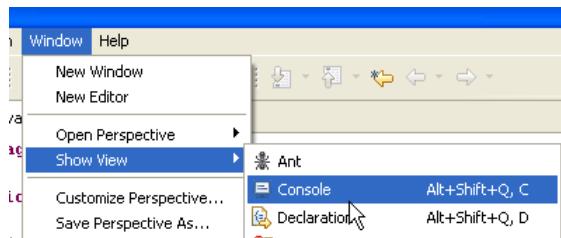


Le résultat de l'exécution est obtenu dans la fenêtre [Console] :



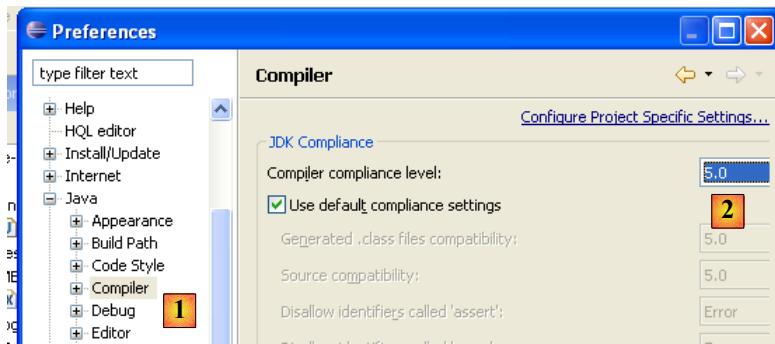
```
<terminated> Test1 [Java Application] C:\Program Files\Java\test1
```

La fenêtre [Console] doit apparaître par défaut. Si ce n'était pas le cas, on peut demander son affichage par [Window>Show View/Console] :



## 5.2.2 Choix du compilateur

Eclipse permet de générer du code compatible Java 1.4, Java 1.5, Java 1.6. Par défaut, il est configuré pour générer du code compatible Java 1.4. L'API JPA nécessite du code Java 1.5. Nous changeons la nature du code généré par [Window / Preferences / Java / Compiler] :

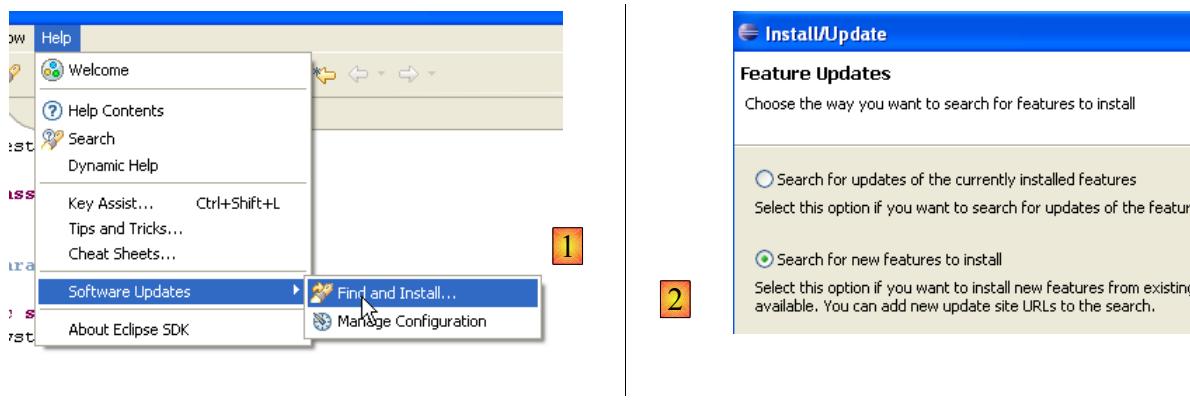


- en [1] : choix de l'option [Java / Compiler]
- en [2] : choix de la compatibilité Java 5.0

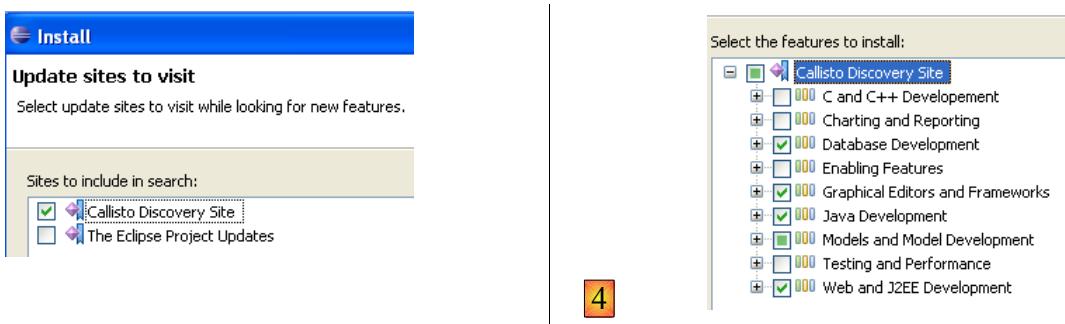
### 5.2.3 Installation des plugins Callisto

La version de base installée ci-dessus permet de construire des applications Java console mais pas des applications Java de type web ou Swing, ou alors il faut tout faire soi-même. Nous allons installer divers plugins :

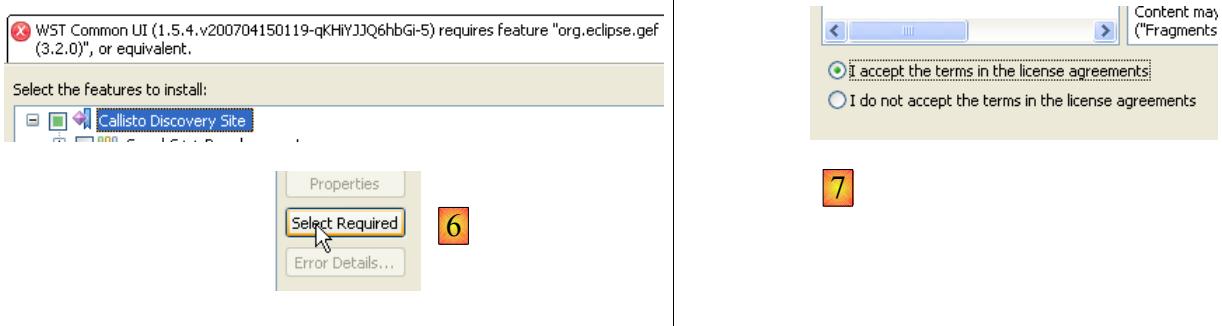
Procédons ainsi [Help/Software Updates/Find and Install] :



- en [2], on indique qu'on veut installer de nouveaux plugins



- en [3], on indique les sites à explorer pour trouver les plugins
- en [4], on coche les plugins désirés



- en [5], Eclipse signale qu'on a choisi un plugin qui dépend d'autres plugins qui n'ont pas été sélectionnés
- en [6], on utilise le bouton [Select Required] pour sélectionner automatiquement les plugins manquants
- en [7], on accepte les conditions des licences de ces divers plugins

Features to install:

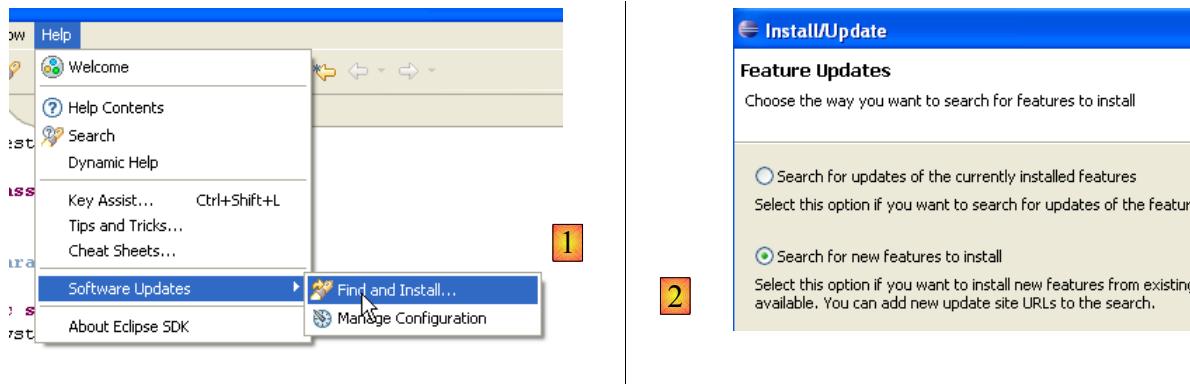
- Feature Name
  - Data Tools Platform Connectivity
  - Data Tools Platform Enablement
  - Data Tools Platform Model Base**
  - Data Tools Platform Open Data Access Designer
  - Data Tools Platform Open Data Access Runtime
  - Data Tools Platform SQL Development Tools
  - Eclipse Modeling Framework (EMF) Runtime + End-User Tools
  - EMF Service Data Objects (SDO) Runtime + End-User Tools
  - Graphical Editing Framework
  - J2EE Standard Tools (JST) Project
  - J2EE Standard Tools (JST) Project
  - Visual Editor
  - Visual Editor
  - Web Standard Tools (WST) Project
  - XML Schema InfoSet Model (XSD) Runtime + End-User Tools

- en [8], on a la liste de tous les plugins qui vont être installés
- en [9], on lance le téléchargement de ces plugins
- en [10], une fois ceux-ci téléchargés, on les installe tous sans vérifier leur signature

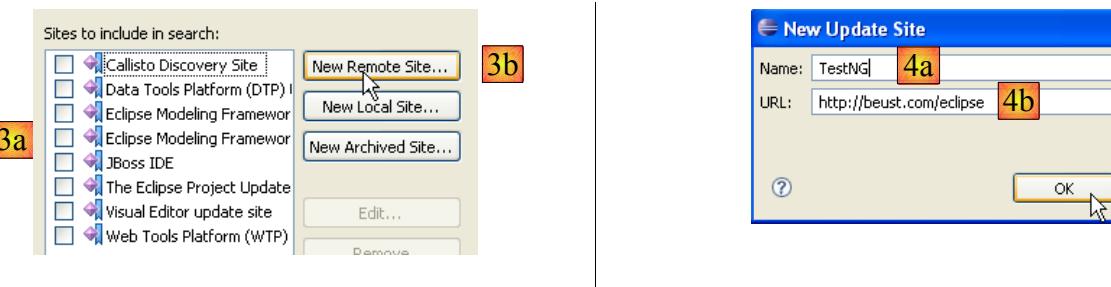
- en [11], à la fin de l'installation des plugins, on laisse Eclipse se relancer
- en [12], si on fait [File/New/Project], on découvre qu'on peut maintenant créer des applications web, ce qui n'était pas possible initialement.

## 5.2.4 Installation du plugin [TestNG]

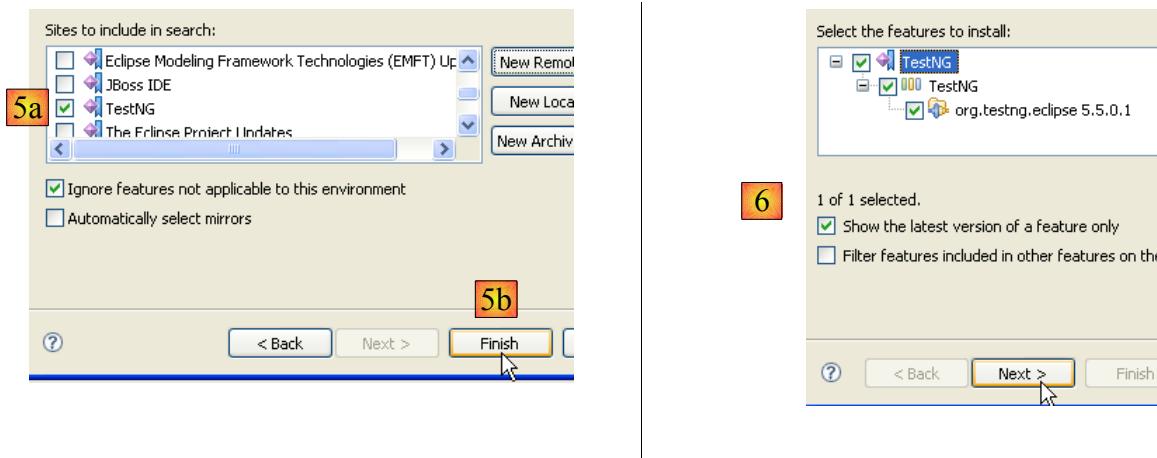
TestNG (Test Next Generation) est un outil de tests unitaires semblable dans son esprit à JUnit. Il apporte cependant des améliorations qui nous le font préférer ici à JUnit. Nous procéderons comme précédemment : [Help/Software Updates/Find and Install] :



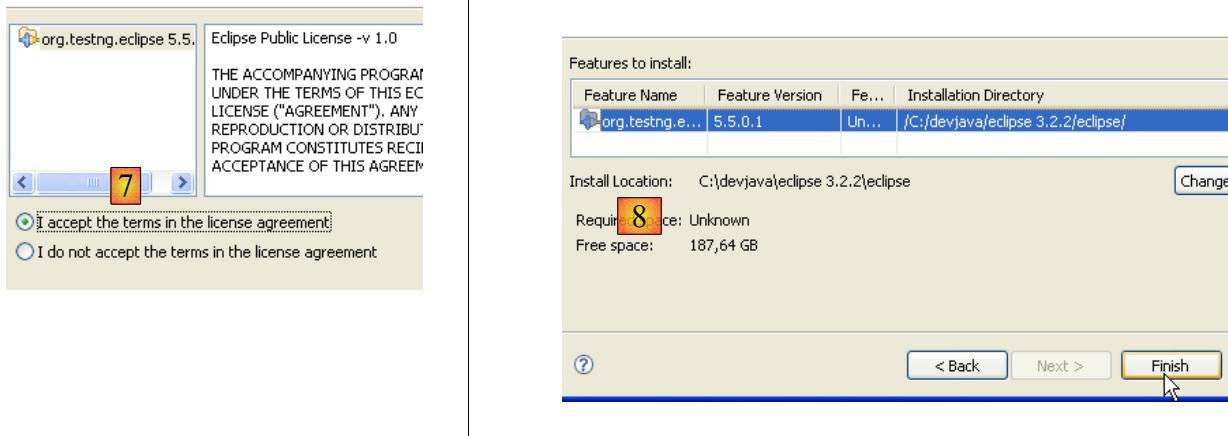
- en [2], on indique qu'on veut installer de nouveaux plugins



- en [3a], le site de téléchargement de [TestNG] n'est pas présent. Nous l'ajoutons avec [3b]
- en [4b] : le site du plugin est [http://beust.com/eclipse]. En [4a], on met ce qu'on veut.

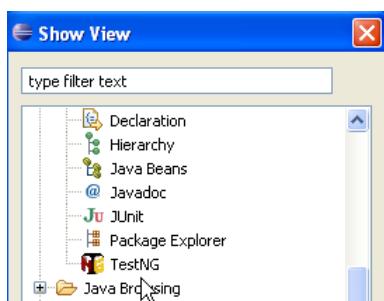


- en [5a], le plugin [TestNG] est sélectionné pour la mise à jour. En [5b], on lance celle-ci.
- en [6], la connexion avec le site du plugin a été faite. Nous sommes présentés tous les plugins disponibles sur le site. Un seul, ici, que nous sélectionnons avant de passer à l'étape suivante.



- en [7], on accepte les conditions des licences du plugin
- en [8], on a la liste de tous les plugins qui vont être installés, un ici. On lance le téléchargement. Ensuite tout se déroule comme décrit ci-dessus, pour les plugins Callisto.

Une fois Eclipse relancé, on peut constater la présence du nouveau plugin en demandant, par exemple, à voir les vues disponibles [Window / show View / Other] :



Nous voyons ci-dessus, l'existence d'une vue [TestNG] qui n'existe pas auparavant.

## 5.2.5 Installation du plugin [Hibernate Tools]

Hibernate est un fournisseur JPA et le plugin [Hibernate Tools] pour Eclipse est utile dans la construction d'applications JPA. En mai 2007, seule sa dernière version (3.2.0beta9) permet de travailler avec Hibernate/JPA et elle n'est pas disponible via le mécanisme qui vient d'être décrit. Seules des versions plus anciennes le sont. On va donc procéder différemment.

Le plugin est disponible sur le site d'Hibernate Tools : <http://tools.hibernate.org/>.

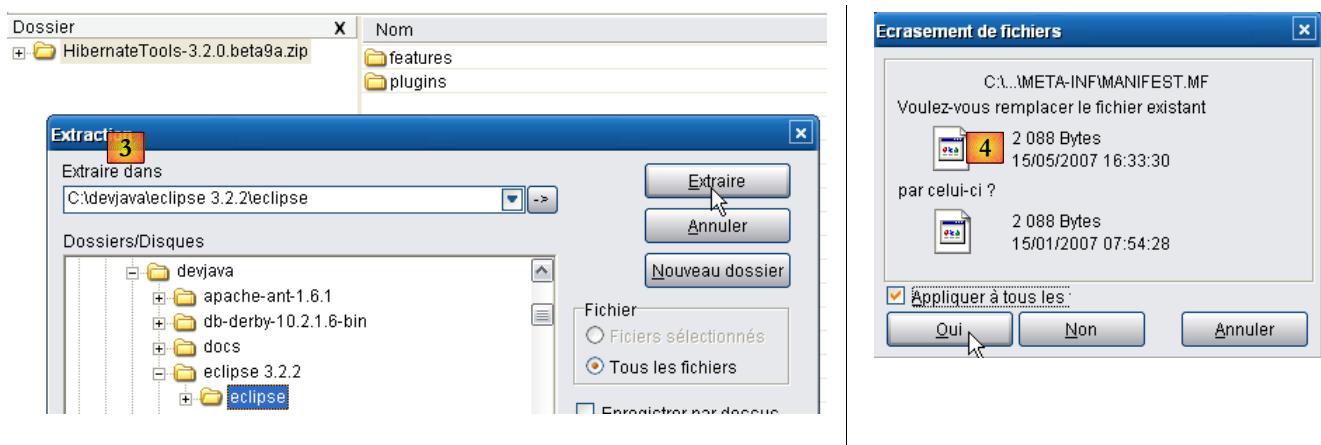
| Release (date)   | Filename                           | Size (bytes) |
|------------------|------------------------------------|--------------|
| 2007-01-13 05:28 | HibernateTools-3.2.0.beta9.zip     | 32689290     |
|                  | HibernateTools-3.2.0.beta9.zip.MD5 | 32           |

| Release (date)   | Filename                           | Size (bytes) |
|------------------|------------------------------------|--------------|
| 2007-01-13 05:28 | HibernateTools-3.2.0.beta9.zip     | 32689290     |
|                  | HibernateTools-3.2.0.beta9.zip.MD5 | 32           |

- en [1], on sélectionne la dernière version d'Hibernate Tools

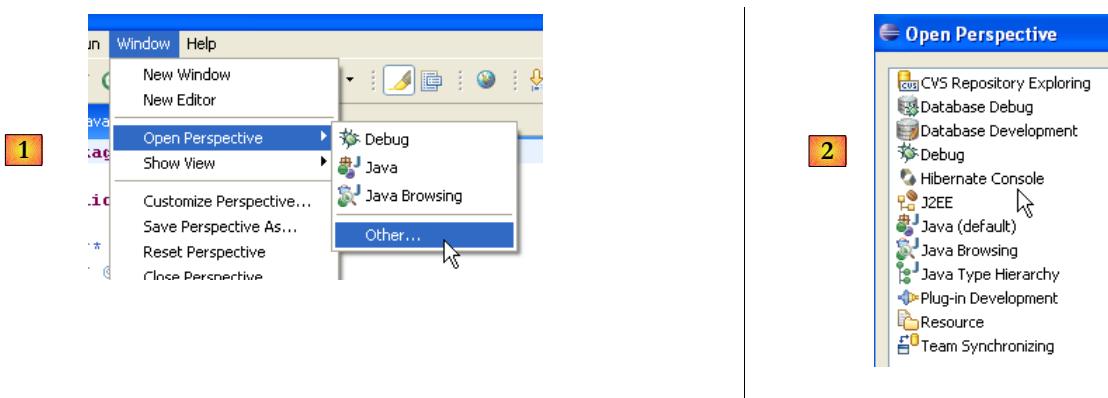
Persistance Java 5 par la pratique

- en [2], on la télécharge



- en [3], avec un dézippeur, on décomprime dans le dossier <clipse> le fichier zip téléchargé (il est préférable qu'Eclipse ne soit pas actif)
- en [4], on accepte que certains fichiers soient écrasés dans l'opération

On relance Eclipse :



- en [1] : on ouvre une perspective
- en [2] : il existe maintenant une perspective [Hibernate Console]

Nous n'irons pas plus loin avec le plugin [Hibernate Tools] (Cancel en [2]). Son mode d'utilisation est expliqué dans les exemples du tutoriel.

Parfois Eclipse ne détecte pas la présence de nouveaux plugins. On peut le forcer à rescanner tous ses plugins avec l'option **-clean**. Ainsi l'exécutable du raccourci d'Eclipse serait modifié de la façon suivante :

```
"<clipse>\clipse.exe" -clean -vm "C:\Program Files\Java\jre1.6.0_01\bin\javaw.exe" -vmargs -Xms40m -Xmx512m
```

Une fois les nouveaux plugins détectés par Eclipse, on enlèvera l'option **-clean** ci-dessus.

## 5.2.6 Installation du plugin [SQL Explorer]

Nous allons maintenant installer un plugin qui nous permettra d'explorer le contenu d'une base de données directement à partir d'Eclipse. Les plugins disponibles pour eclipse peuvent être trouvés sur le site [<http://eclipse-plugins.2y.net/eclipse/plugins.jsp>] :

**Plugin categories**

- All (1472)
- Ant (11)
- AspectJ (6)
- Bug Tracker (12)
- Business Process Tools (20)
- Code Generation (36)
- Code Generation/Modelling (19)
- Code mgmt (40)
- Com,Corba,Idl,... (10)
- Database (45) **[2]**
- Database Persistence (25)

Languages - Java  
Languages - Ma...  
Languages - oth...  
LDAP (8)  
Logging (12)  
Misc (35)  
Mobile/PDA (15)  
Modelling (34)  
Network (11)  
Obsolete (54)

**Plugins in Database**  
Database (45) sort on rating  
45 plugins pages: 1 2 3 4 5 Next > **[3]**

**QuantumDB** **[4]**  
( Free / Database )  
updated:30 avr. 2007

**SQLExplorer** **[5]**  
( LGPL / Database )  
updated:31 août 2006  
Version 3.0  
SQLExplorer is a database query/schema to ma...  
<http://sourceforge.net/projects/eclipsesql>  
[details](#) | [plugin-homepage](#) | [rating \(8.0/10\)](#)

- en [1] : le site des plugins Eclipse
- en [2] : choisir la catégorie [Database]
- en [3] : dans la catégorie [Database], choisir un affichage par classement (peu fiable vu le faible nombre de gens qui votent)
- en [4] : QuantumDB arrive en 1ère position
- en [5] : nous choisissons SQLExplorer, plus ancien, moins bien côté (3ième) mais très bien quand même. Nous allons sur le site du plugin [plugin-homepage]

for other databases.

[Download Eclipse SQL Explorer \[RCP/Plugin\]](#) **[6]**



- en [6] et [7] : on procède au téléchargement du plugin.

New Open Add Extract Install View

Folder

+ [sqlexplorer\\_plugin\\_3.0.0.20060901.zip](#)

**Extract**

Extract to: C:\dev\java\eclipse 3.2\eclipse

Folders / Drives

- db derby-10.2.2.0-bin
- eclipse
- eclipse 3.2
  - eclipse** **[8]**
- eclipse 3.3
- eclipse callisto-060701

- en [8] : décompresser le fichier zip du plugin dans le dossier d'Eclipse.

Pour vérifier, relancer Eclipse avec éventuellement l'option -clean :



- en [1] : ouvrir une nouvelle perspective
- en [2] : on voit qu'une perspective [SQL Explorer] est disponible. Nous y reviendrons ultérieurement.

## 5.3 Le conteneur de servlets Tomcat 5.5

### 5.3.1 Installation

Pour exécuter des servlets, il nous faut un conteneur de servlets. Nous présentons ici l'un d'eux, Tomcat 5.5, disponible à l'url <http://tomcat.apache.org/>. Nous indiquons la démarche (mai 2007) pour l'installer. Si une précédente version de Tomcat est déjà installée, il est préférable de la supprimer auparavant.

**Download**

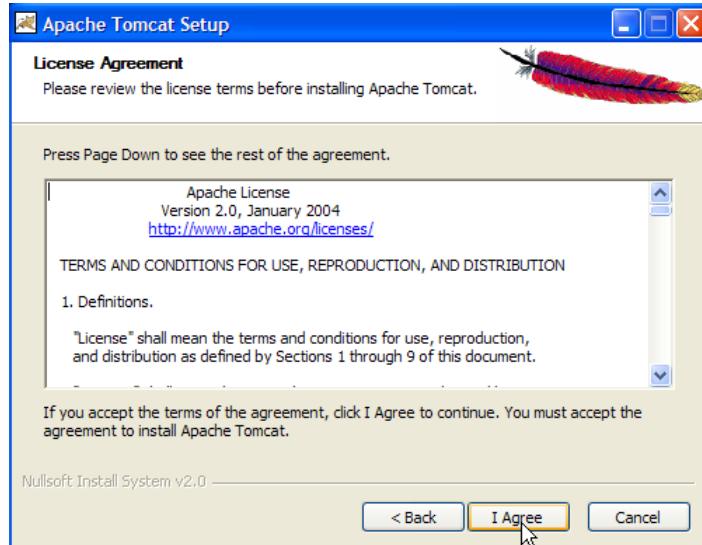
- ◆ [Which version?](#)
- ◆ [Tomcat 6.x](#)
- ◆ [Tomcat 5.x](#)
- ◆ [Tomcat 4.x](#)
- ◆ [Tomcat 3.3](#)

Pour télécharger le produit, on suivra le lien [Tomcat 5.x] ci-dessus :

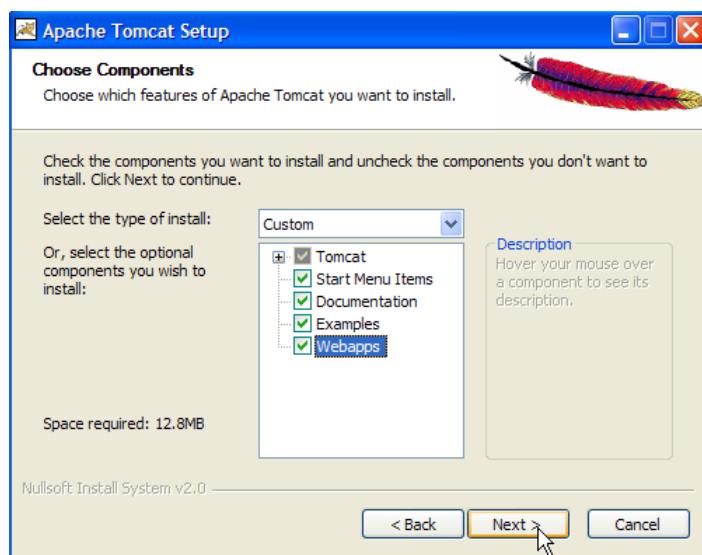
**Binary Distributions**

- ◆ Core:
  - [zip \(pgp, md5\)](#)
  - [tar.gz \(pgp, md5\)](#)
  - [Windows Service Installer \(pgp, md5\)](#)
- ◆ Deployer:

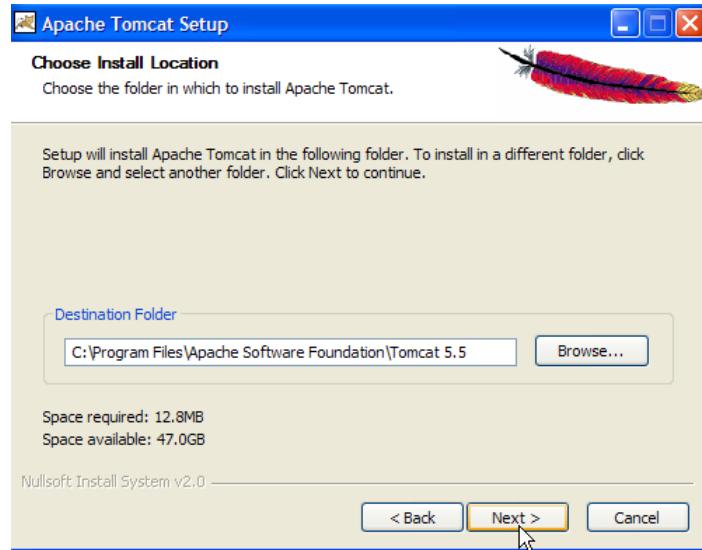
On pourra prendre le .exe destiné à la plate-forme windows. Une fois celui-ci téléchargé, on lance l'installation de Tomcat en double-cliquant dessus :



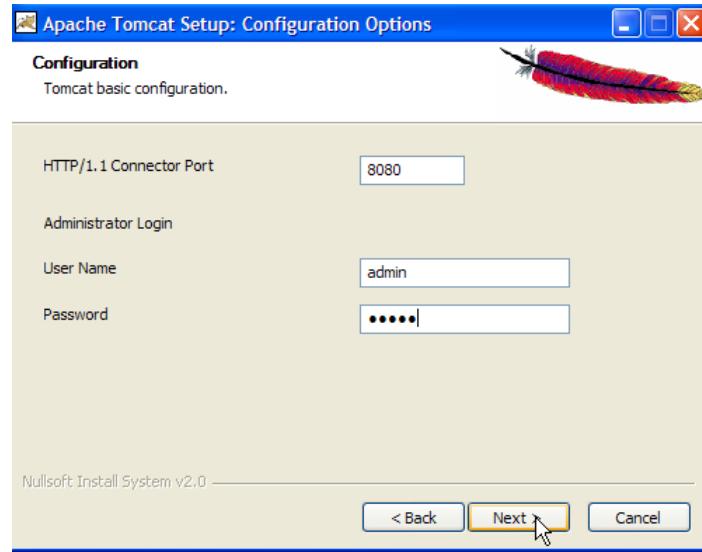
On accepte les conditions de la licence ->



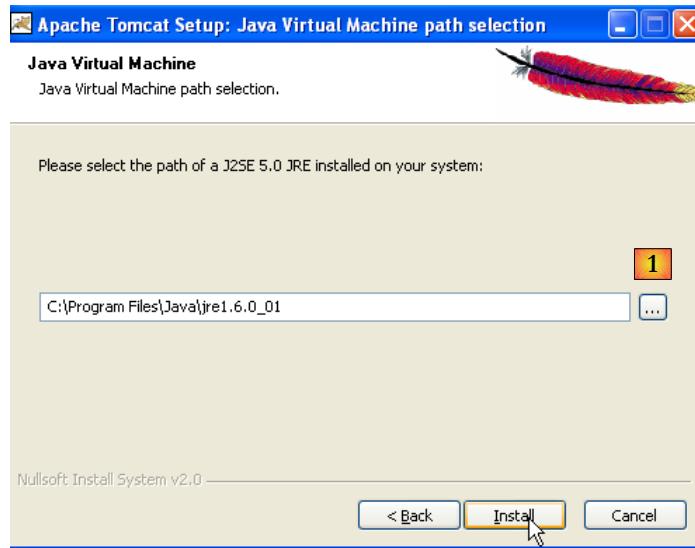
Faire [next] ->



Accepter le dossier d'installation proposé ou le changer avec [Browse] ->



Fixer le login / mot de passe de l'administrateur du serveur Tomcat. Ici on a mis [admin / admin] ->



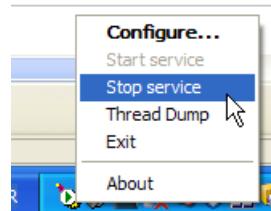
Tomcat 5.x a besoin d'un JRE 1.5. Il doit normalement trouver celui qui est installé sur votre machine. Ci-dessus, le chemin désigné est celui du JRE 1.6 téléchargé au paragraphe 5.1, page 209. Si aucun JRE n'est trouvé, désignez sa racine en utilisant le bouton [1]. Ceci fait, utilisez le bouton [Install] pour installer Tomcat 5.x ->



Le bouton [Finish] termine l'installation. La présence de Tomcat est signalée par une icône à droite dans la barre des tâches de windows :



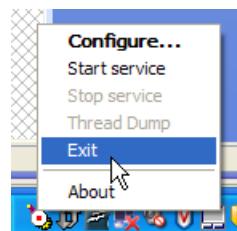
Un clic droit sur cette icône donne accès aux commandes Marche – Arrêt du serveur :



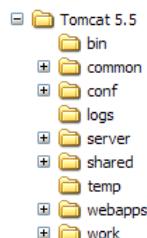
Nous utilisons l'option [Stop service] pour arrêter maintenant le serveur web :



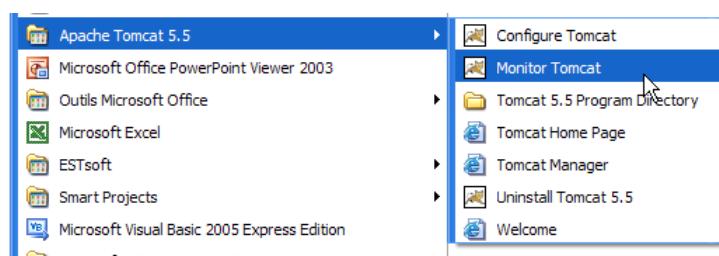
On notera le changement d'état de l'icône. Cette dernière peut être enlevée de la barre des tâches :



L'installation de Tomcat s'est faite dans le dossier choisi par l'utilisateur, que nous appellerons désormais <tomcat>. L'arborescence de ce dossier pour la version Tomcat 5.5.23 téléchargée est la suivante :



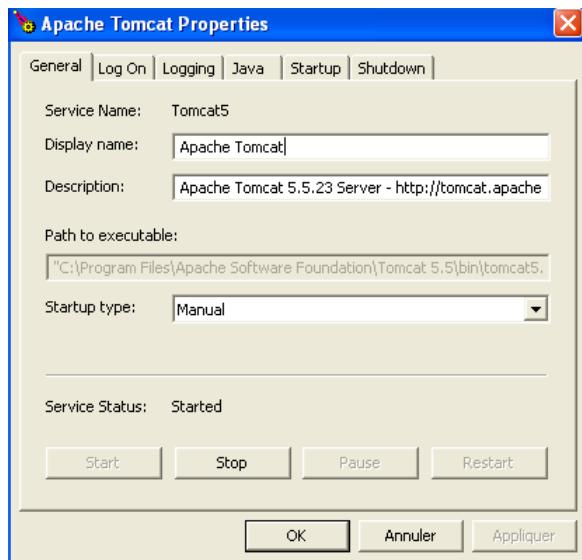
L'installation de Tomcat a amené un certain nombre de raccourcis dans le menu [Démarrer]. Nous utilisons le lien [Monitor] ci-dessous pour lancer l'outil d'arrêt/démarrage de Tomcat :



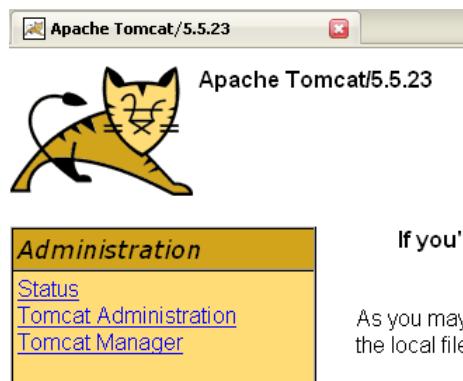
Nous retrouvons alors l'icône présentée précédemment :



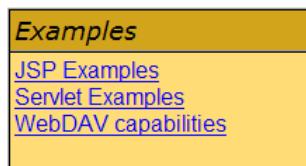
Le moniteur de Tomcat peut être activé par un double-clic sur cette icône :



Les boutons [Start - Stop - Pause] - Restart nous permettent de lancer - arrêter - relancer le serveur. Nous lançons le serveur par [Start] puis, avec un navigateur nous demandons l'url <http://localhost:8080>. Nous devons obtenir une page analogue à la suivante :



On pourra suivre les liens ci-dessous pour vérifier la correcte installation de Tomcat :



Tous les liens de la page [<http://localhost:8080>] présentent un intérêt et le lecteur est invité à les explorer. Nous aurons l'occasion de revenir sur les liens permettant de gérer les applications web déployées au sein du serveur :



## 5.3.2 Déploiement d'une application web au sein du serveur Tomcat

### 5.3.3 Déploiement

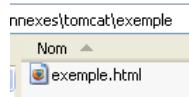
Une application web doit suivre certaines règles pour être déployée au sein d'un conteneur de servlets. Soit <webapp> le dossier d'une application web. Une application web est composée de :

|                                     |                                               |
|-------------------------------------|-----------------------------------------------|
| classes                             | dans le dossier <webapp>\WEB-INF\classes      |
| archives java                       | dans le dossier <webapp>\WEB-INF\lib          |
| vues, ressources (.jsp, .html, ...) | dans le dossier <webapp> ou des sous-dossiers |

L'application web est configurée par un fichier XML : <webapp>\WEB-INF\web.xml. Ce fichier n'est pas nécessaire dans les cas simples, notamment le cas où l'application web ne contient que des fichiers statiques. Construisons le fichier HTML suivant :

```
<html>
  <head>
    <title>Application exemple</title>
  </head>
  <body>
    Application exemple active ....
  </body>
</html>
```

et sauvegardons-le dans un dossier :

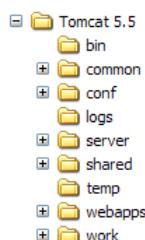


Si on charge ce fichier dans un navigateur, on obtient la page suivante :

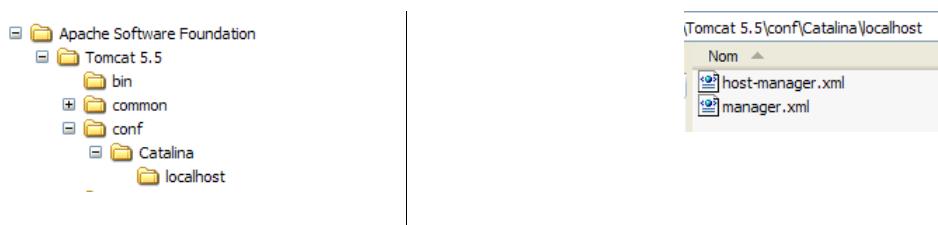


L'URL affichée par le navigateur montre que la page n'a pas été délivrée par un serveur web mais directement chargée par le navigateur. Nous voulons maintenant qu'elle soit disponible via le serveur web Tomcat.

Revenons sur l'arborescence du répertoire <tomcat> :



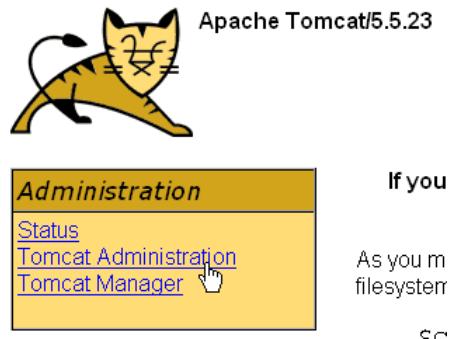
La configuration des applications web déployées au sein du serveur Tomcat se fait à l'aide de fichiers XML placés dans le dossier [<tomcat>\conf\Catalina\localhost] :



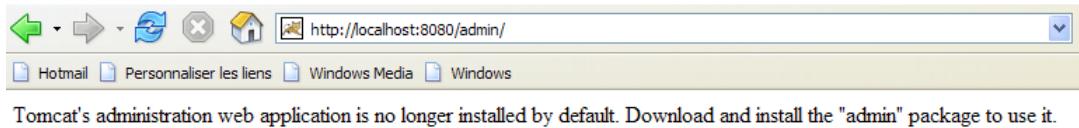
Ces fichiers XML peuvent être créés à la main car leur structure est simple. Plutôt que d'adopter cette démarche, nous allons utiliser les outils web que nous offre Tomcat.

### 5.3.4 Administration de Tomcat

Sur sa page d'entrée <http://localhost:8080>, le serveur nous offre des liens pour l'administrer :



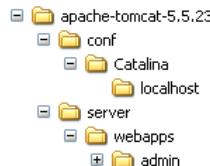
Le lien [Tomcat Administration] nous offre la possibilité de configurer les ressources mises par Tomcat à la disposition des applications web déployées en son sein, par exemple un pool de connexions à une base de données. Suivons le lien :



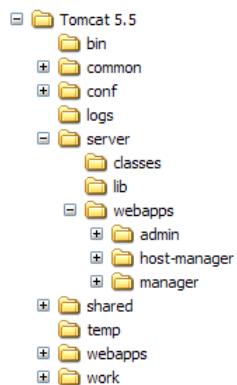
La page obtenue nous indique que l'administration de Tomcat 5.x nécessite un paquetage particulier appelé " admin ". Retournons sur le site de Tomcat [<http://tomcat.apache.org/download-55.cgi>] :

- Administration Web Application:
  - [zip \(pgp, md5\)](#)
  - [tar.gz \(pgp, md5\)](#)

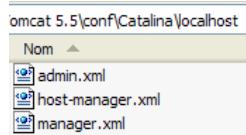
Téléchargeons le zip libellé [Administration Web Application] puis décompressons-le. Son contenu est le suivant :



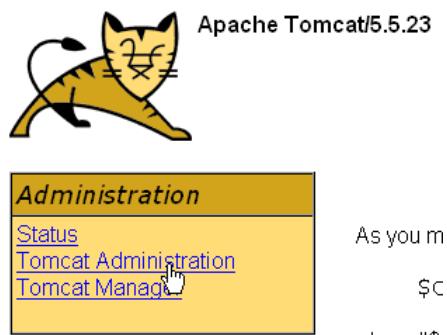
Le dossier [admin] doit être copié dans [`<tomcat>\server\webapps`] où `<tomcat>` est le dossier où a été installé tomcat 5.x :



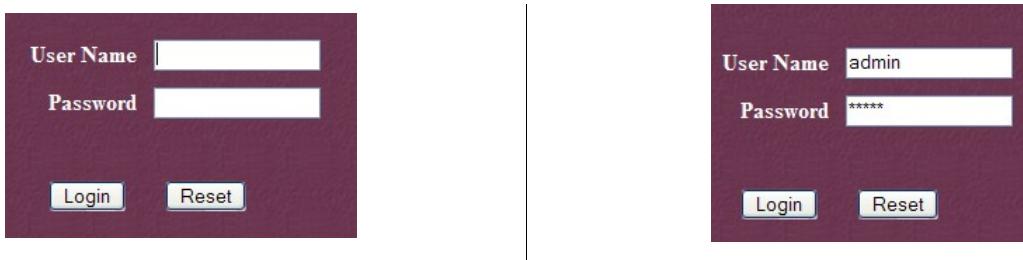
Le dossier [localhost] contient un fichier [admin.xml] qui doit être copié dans [`<tomcat>\conf\Catalina\localhost`] :



Arrêtons puis relançons Tomcat s'il était actif. Puis avec un navigateur, redemandons la page d'entrée du serveur web :



Suivons le lien [Tomcat Administration]. Nous obtenons une page d'identification (pour l'obtenir, on peut être amené à faire un "reload/refresh" de la page) :



Ici, il faut redonner les informations que nous avons données au cours de l'installation de Tomcat. Dans notre cas, nous donnons le couple *admin / admin*. Le bouton [Login] nous amène à la page suivante :



Cette page permet à l'administrateur de Tomcat de définir

- des sources de données (Data Sources),
- les informations nécessaires à l'envoi de courrier (Mail Sessions),
- des données d'environnement accessibles à toutes les applications (Environment Entries),
- de gérer les utilisateurs/administrateurs de Tomcat (Users),
- de gérer des groupes d'utilisateurs (Groups),
- de définir des rôles (= ce que peut faire ou non un utilisateur),
- de définir les caractéristiques des applications web déployées par le serveur (Service Catalina)

Suivons le lien [Roles] ci-dessus :

Roles List		Role Actions
Role Name	Description	...Available Actions....
<a href="#">admin</a>		---Available Actions---
<a href="#">manager</a>		Create New Role Delete Existing Roles List Existing Roles
<a href="#">role1</a>		
<a href="#">tomcat</a>		

Un rôle permet de définir ce que peut faire ou ne pas faire un utilisateur ou un groupe d'utilisateurs. On associe à un rôle certains droits. Chaque utilisateur est associé à un ou plusieurs rôles et dispose des droits de ceux-ci. Le rôle [manager] ci-dessous donne le droit de gérer les applications web déployées dans Tomcat (déploiement, démarrage, arrêt, déchargement). Nous allons créer un utilisateur [manager] qu'on associera au rôle [manager] afin de lui permettre de gérer les applications de Tomcat. Pour cela, nous suivons le lien [Users] de la page d'administration :

Users List		User Actions
User Name	Full Name	...Available Actions....
<a href="#">admin</a>		---Available Actions---
<a href="#">both</a>		Create New User Delete Existing Users List Existing Users
<a href="#">role1</a>		
<a href="#">tomcat</a>		

Nous voyons qu'un certain nombre d'utilisateurs existent déjà. Nous utilisons l'option [Create New User] pour créer un nouvel utilisateur :

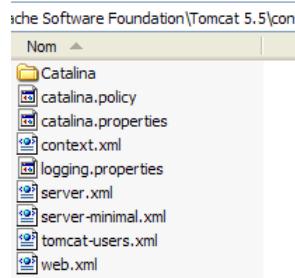
User Properties	
User Name:	<input type="text" value="manager"/>
Password:	<input type="password" value="*****"/>
Full Name:	<input type="text"/>
Group Name	Description
Role Name	Description
<input type="checkbox"/>	<a href="#">admin</a>
<input checked="" type="checkbox"/>	<a href="#">manager</a>
<input type="checkbox"/>	<a href="#">role1</a>
<input type="checkbox"/>	<a href="#">tomcat</a>

[Save](#) | [Reset](#)

Nous donnons à l'utilisateur **manager** le mot de passe **manager** et nous lui attribuons le rôle **manager**. Nous utilisons le bouton [Save] pour valider cet ajout. Le nouvel utilisateur apparaît dans la liste des utilisateurs :

User Name	Full Name
<a href="#">admin</a>	
<a href="#">both</a>	
<a href="#">manager</a>	
<a href="#">role1</a>	
<a href="#">tomcat</a>	

Ce nouvel utilisateur va être ajouté dans le fichier [`<tomcat>\conf\tomcat-users.xml`] :



dont le contenu est le suivant :

```

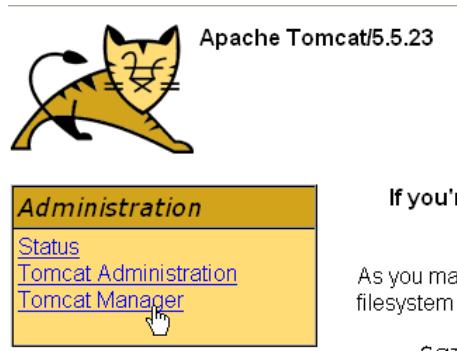
-  <?xml version='1.0' encoding='utf-8'?>
-  <tomcat-users>
-    <role rolename="tomcat"/>
-    <role rolename="role1"/>
-    <role rolename="manager"/>
-    <role rolename="admin"/>
-    <user username="tomcat" password="tomcat" roles="tomcat"/>
-    <user username="role1" password="tomcat" roles="role1"/>
-    <user username="both" password="tomcat" roles="tomcat,role1"/>
-    <user username="manager" password="manager" fullName="" roles="manager"/>
-    <user username="admin" password="admin" roles="admin,manager"/>
-  </tomcat-users>
```

- ligne 10 : l'utilisateur [manager] qui a été créé

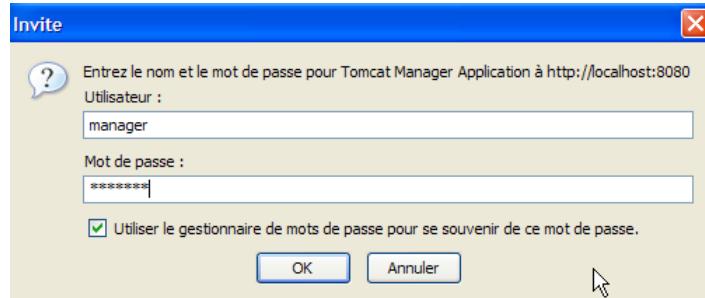
Une autre façon d'ajouter des utilisateurs est de modifier directement ce fichier. C'est notamment ainsi qu'il faut procéder si, d'aventure, on a oublié le mot de passe de l'administrateur **admin** ou de **manager**.

### 5.3.5 Gestion des applications web déployées

Revenons maintenant à la page d'entrée [`http://localhost:8080`] et suivons le lien [Tomcat Manager] :



Nous obtenons alors une page d'authentification. Nous nous identifions comme *manager / manager*, c.a.d. l'utilisateur de rôle [manager] que nous venons de créer. En effet, seul un utilisateur ayant ce rôle peut utiliser ce lien. Ligne 11 de [tomcat-users.xml], nous voyons que l'utilisateur [admin] a également le rôle [manager]. Nous pourrions donc utiliser également l'authentification [admin / admin].



Nous obtenons une page listant les applications actuellement déployées dans Tomcat :

Manager					
List Applications		HTML Manager Help		Manager Help	
Applications					
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands	
/	Welcome to Tomcat	true	0	Démarrer	Arrêter Recharger Undeploy
/admin	Tomcat Administration Application	true	1	Démarrer	Arrêter Recharger Undeploy
/balancer	Tomcat Simple Load Balancer Example App	true	0	Démarrer	Arrêter Recharger Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer	Arrêter Recharger Undeploy
/jsp-examples	JSP 2.0 Examples	true	0	Démarrer	Arrêter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer	Arrêter Recharger Undeploy
/servlets-examples	Servlet 2.4 Examples	true	0	Démarrer	Arrêter Recharger Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer	Arrêter Recharger Undeploy
/webdav	Webdav Content Management	true	0	Démarrer	Arrêter Recharger Undeploy

Nous pouvons ajouter une nouvelle application grâce à des formulaires placés en bas de la page :

Deploy	
Deploy directory or WAR file located on server	
Context Path (optional):	<input type="text"/>
XML Configuration file URL:	<input type="text"/>
WAR or Directory URL:	<input type="text"/>
<input type="button" value="Deploy"/>	
WAR file to deploy	
Select WAR file to upload	<input type="text"/> <input type="button" value="Parcourir..."/>
<input type="button" value="Deploy"/>	

Ici, nous voulons déployer au sein de Tomcat, l'application **exemple** que nous avons construite précédemment. Nous le faisons de la façon suivante :

Deploy	
Deploy directory or WAR file located on server	
Context Path (optional):	<input type="text" value="/exemple"/>
XML Configuration file URL:	<input type="text"/>
WAR or Directory URL:	<input type="text" value="C:\data\travail\2006-2007\eclipse\dvp-jpa\ar\exemp"/>
<input type="button" value="Deploy"/>	

Context Path	/exemple	le nom utilisé pour désigner l'application web à déployer
Directory URL	C:\data\travail\2006-2007\eclipse\dvp-jpa\annexes\tomcat\exemple	le dossier de l'application web

Pour obtenir le fichier [C:\data\travail\2006-2007\eclipse\dvp-jpa\annexes\tomcat\exemple\exemple.html], nous demanderons à Tomcat l'URL [<http://localhost:8080/exemple/exemple.html>]. Le **contexte** sert à donner un nom à la racine de l'arborescence de l'application web déployée. Nous utilisons le bouton [Deploy] pour effectuer le déploiement de l'application. Si tout se passe bien, nous obtenons la page réponse suivante :

## Gestionnaire d'applications WEB Tomcat

<b>Message:</b>	OK - Application déployée pour le chemin de contexte /exemple
-----------------	---------------------------------------------------------------

et la nouvelle application apparaît dans la liste des applications déployées :

<b>Applications</b>				
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arrêter Recharger Undeploy
/admin	Tomcat Administration Application	true	1	Démarrer Arrêter Recharger Undeploy
/balancer	Tomcat Simple Load Balancer Example App	true	0	Démarrer Arrêter Recharger Undeploy
/exemple		true	0	Démarrer Arrêter Recharger Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy
/jsp-examples	JSP 2.0 Examples	true	0	Démarrer Arrêter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy
/servlets-examples	Servlet 2.4 Examples	true	0	Démarrer Arrêter Recharger Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer Arrêter Recharger Undeploy
/webdav	Webdav Content Management	true	0	Démarrer Arrêter Recharger Undeploy

Commentons la ligne du contexte **/exemple** ci-dessus :

/exemple	lien sur <a href="http://localhost:8080/exemple">http://localhost:8080/exemple</a>
Démarrer	permet de démarrer l'application
Arrêter	permet d'arrêter l'application
Recharger	permet de recharger l'application. C'est nécessaire par exemple lorsqu'on a ajouté, modifié ou supprimé certaines classes de l'application.
Undeploy	suppression du contexte [/exemple]. L'application disparaît de la liste des applications disponibles.

Maintenant que notre application **/exemple** est déployée, nous pouvons faire quelques tests. Nous demandons la page [exemple.html] via l'url [<http://localhost:8080/exemple/vues/exemple.html>] :



Une autre façon de déployer une application web au sein du serveur Tomcat est de donner les renseignements que nous avons donnés via l'interface web, dans un fichier [contexte].xml placé dans le dossier [<tomcat>\conf\Catalina\localhost], où [contexte] est le nom de l'application web.

Revenons à l'interface d'administration de Tomcat :

<b>Applications</b>				
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arrêter Recharger Undeploy
/admin	Tomcat Administration Application	true	1	Démarrer Arrêter Recharger Undeploy
/balancer	Tomcat Simple Load Balancer Example App	true	0	Démarrer Arrêter Recharger Undeploy
/exemple		true	0	Démarrer Arrêter Recharger Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy

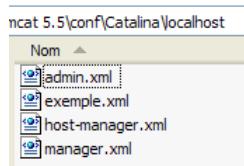
Supprimons l'application [/exemple] avec son lien [Undeploy] :

Applications	
Chemin	Nom d'affichage
/	Welcome to Tomcat
/admin	Tomcat Administration Application
/balancer	Tomcat Simple Load Balancer Example App
/host-manager	Tomcat Manager Application
/jsp-examples	JSP 2.0 Examples
/manager	Tomcat Manager Application
/servlets-examples	Servlet 2.4 Examples
/tomcat-docs	Tomcat Documentation
/webdav	Webdav Content Management

L'application [/exemple] ne fait plus partie de la liste des applications actives. Maintenant définissons le fichier [exemple.xml] suivant :

```
1. <Context docBase="C:/data/travail/2006-2007/eclipse/dvp-jpa/annexes/tomcat/exemple">
2. </Context>
```

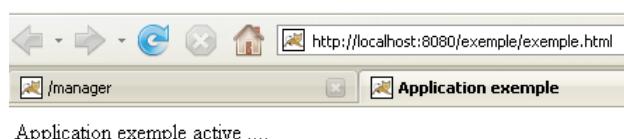
Le fichier XML est constitué d'une unique balise <Context> dont l'attribut **docBase** définit le dossier contenant l'application web à déployer. Plaçons ce fichier dans <tomcat>\conf\ Catalina\localhost :



Arrêtons et relançons Tomcat si besoin est, puis visualisons la liste des applications actives avec l'administrateur de Tomcat :

Applications	
Chemin	Nom d'affichage
/	Welcome to Tomcat
/admin	Tomcat Administration Application
/balancer	Tomcat Simple Load Balancer Example App
/exemple	
/host-manager	Tomcat Manager Application

L'application [/exemple] est bien présente. Demandons, avec un navigateur, l'url : [http://localhost:8080/exemple/exemple.html] :



Une application web ainsi déployée peut être supprimée de la liste des applications déployées, de la même façon que précédemment, avec le lien [Undeploy] :

Applications					
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands	
/	Welcome to Tomcat	true	0	Démarrer	Arrêter
/admin	Tomcat Administration Application	true	1	Démarrer	Arrêter
/balancer	Tomcat Simple Load Balancer Example App	true	0	Démarrer	Arrêter
/exemple		true	0	Démarrer	Arrêter
/host-manager	Tomcat Manager Application	true	0	Démarrer	Arrêter

Dans ce cas, le fichier [exemple.xml] est automatiquement supprimé du dossier [<tomcat>\conf\Catalina\localhost].

Enfin, pour déployer une application web au sein de Tomcat, on peut également définir son contexte dans le fichier [<tomcat>\conf\server.xml]. Nous ne développerons pas ce point ici.

### 5.3.6 Application web avec page d'accueil

Lorsque nous demandons l'url [<http://localhost:8080/exemple/>], nous obtenons la réponse suivante :



Avec certaines versions précédentes de Tomcat, nous aurions obtenu le contenu du dossier physique de l'application [/exemple].

On peut faire en sorte que, lorsque le contexte est demandé, on affiche une page dite d'accueil. Pour cela, nous créons un fichier [web.xml] que nous plaçons dans le dossier <exemple>\WEB-INF, où <exemple> est le dossier physique de l'application web [/exemple]. Ce fichier est le suivant :

```

1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <web-app xmlns="http://java.sun.com/xml/ns/j2ee"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
    app_2_4.xsd"
5.      version="2.4">
6.
7.      <display-name>Application Exemple</display-name>
8.      <description>Application web minimale</description>
9.      <welcome-file-list>
10.         <welcome-file>/exemple.html</welcome-file>
11.     </welcome-file-list>
12. </web-app>
```

- lignes 2-5 : la balise racine <web-app> avec des attributs obtenus par copier / coller du fichier [web.xml] de l'application [/admin] de Tomcat (<tomcat>/server/webapps/admin/WEB-INF/web.xml).
- ligne 7 : le nom d'affichage de l'application web. C'est un nom libre qui a moins de contraintes que le nom de contexte de l'application. On peut y mettre des espaces par exemple, ce qui n'est pas possible avec le nom de contexte. Ce nom est affiché par exemple par l'administrateur Tomcat :

Applications	
Chemin	Nom d'affichage
/	Welcome to Tomcat
/admin	Tomcat Administration Application
/balancer	Tomcat Simple Load Balancer Example App
/exemple	Application Exemple

- ligne 8 : description de l'application web. Ce texte peut ensuite être obtenu par programmation.
- lignes 9-11 : la liste des fichiers d'accueil. La balise <welcome-file-list> sert à définir la liste des vues à présenter lorsqu'un client demande le contexte de l'application. Il peut y avoir plusieurs vues. La première trouvée est présentée au client. Ici nous n'en avons qu'une : [/exemple.html]. Ainsi, lorsqu'un client demandera l'url [/exemple], ce sera en fait l'url [/exemple/exemple.html] qui lui sera délivrée.

Sauvegardons ce fichier [web.xml] dans <exemple>\WEB-INF :

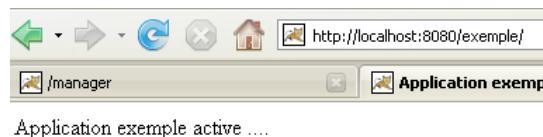


Si Tomcat est toujours actif, il est possible de le forcer à recharger l'application web [/exemple] avec le lien [Recharger] :

/balancer	Tomcat Simple Load Balancer Example App	true	0	Démarrer	Arrêter	<b>Recharger</b>	Undeploy
/exemple	Application Exemple	true	0	Démarrer	Arrêter	<b>Recharger</b>	Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer	Arrêter	<b>Recharger</b>	Undeploy

Lors de cette opération de " recharge ", Tomcat relit le fichier [web.xml] contenu dans [<exemple>\WEB-INF] s'il existe. Ce sera le cas ici. Si Tomcat était arrêté, le relancer.

Avec un navigateur, demandons l'URL [http://localhost:8080/exemple/] :



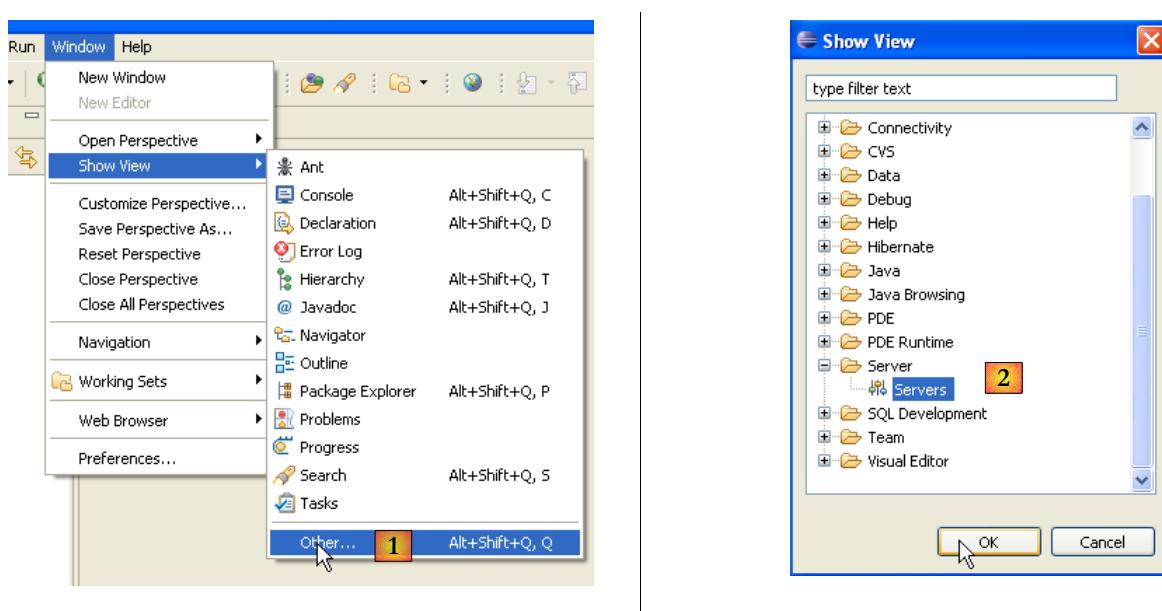
Le mécanisme des fichiers d'accueil a fonctionné.

### 5.3.7 Intégration de Tomcat dans Eclipse

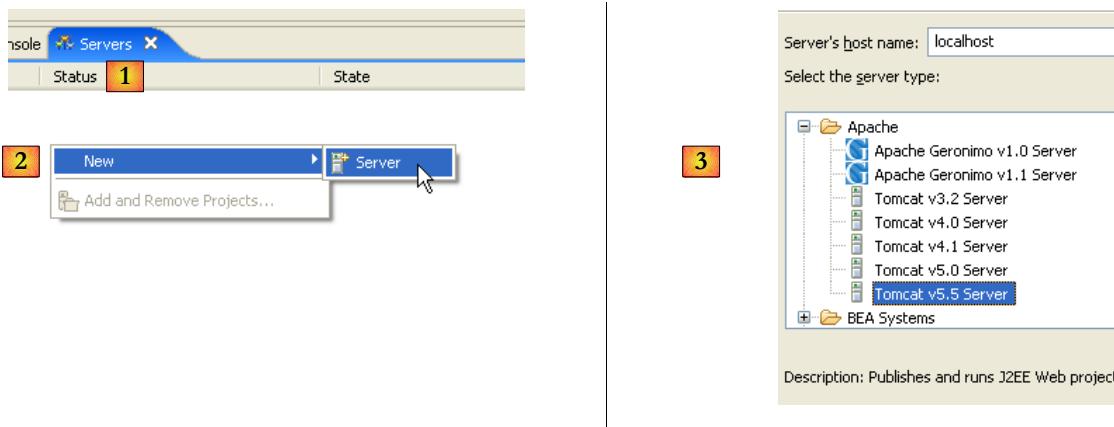
Nous allons maintenant intégrer Tomcat dans Eclipse. Cette intégration permet de :

- lancer / arrêter Tomcat à partir d'Eclipse
- développer des applications web Java et les exécuter sur Tomcat. L'intégration Eclipse / Tomcat permet de tracer (déboguer) l'exécution de l'application, y compris l'exécution des classes Java (servlets) exécutées par Tomcat.

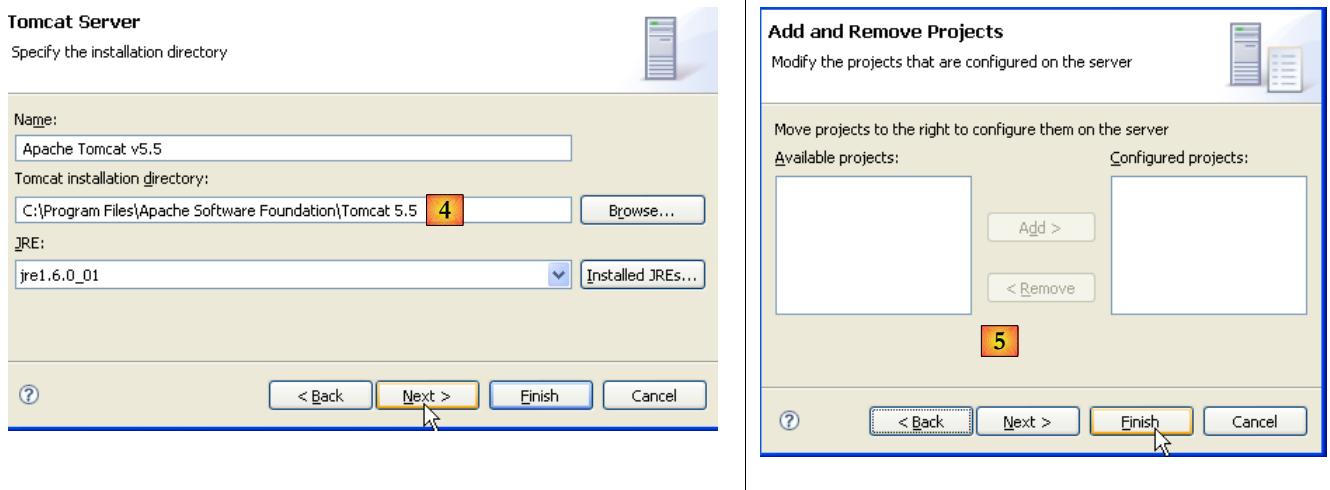
Launchons Eclipse, puis demandons à voir la vue [Servers] :



- en [1] : Window/Show View/Other
- en [2] : sélectionner la vue [Servers] et faire [OK]



- en [1], on a une nouvelle vue [Servers]
- en [2], on clique droit sur la vue et on demande à créer un nouveau serveur [New/Server]
- en [3], on choisit le serveur [Tomcat 5.5] puis on fait [Next]

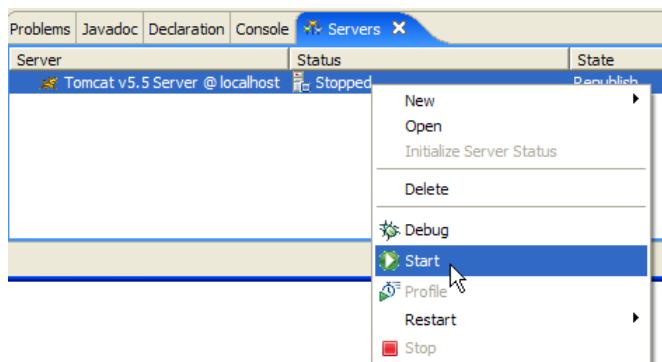


- en [4], on désigne le dossier d'installation de Tomcat 5.5
- en [5], on indique qu'il n'y a pas de projets Eclipse/Tomcat pour le moment. on fait [Finish]

L'ajout du serveur se concrétise par celui d'un dossier dans l'explorateur de projets d'Eclipse [6] et l'apparition d'un serveur dans la vue [servers] [7] :



Dans la vue [Servers] apparaissent tous les serveurs déclarés, ici uniquement le serveur Tomcat 5.5 que nous venons d'enregistrer. Un clic droit dessus donne accès aux commandes permettant de démarrer – arrêter – relancer le serveur :



Ci-dessus, nous lançons le serveur. Lors de son démarrage, un certain nombre de logs sont écrits dans la vue [Console] :

```

1. 16 mai 2007 09:51:57 org.apache.catalina.core.AprLifecycleListener lifecycleEvent
2. ...
3. 16 mai 2007 09:51:57 org.apache.coyote.http11.Http11BaseProtocol init
4. INFO: Initialisation de Coyote HTTP/1.1 sur http-8080
5. ...
6. INFO: Find registry server-registry.xml at classpath resource
7. 16 mai 2007 09:51:58 org.apache.catalina.startup.Catalina start
8. INFO: Server startup in 828 ms

```

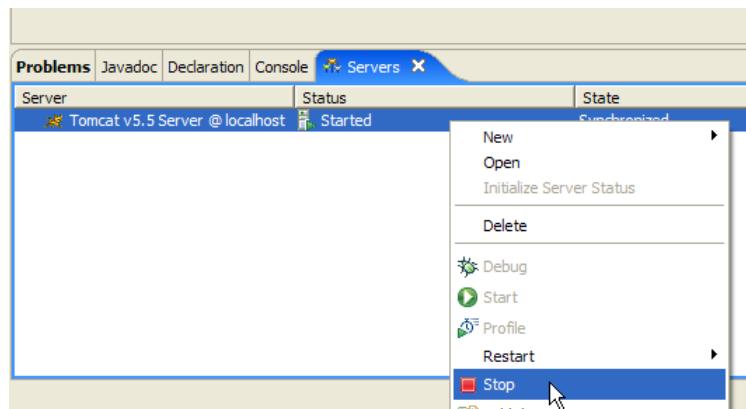
La compréhension de ces logs demande une certaine habitude. Nous ne nous apesantirons pas dessus pour le moment. Il est cependant important de vérifier qu'ils ne signalent pas d'erreurs de chargement de contextes. En effet, lorsqu'il est lancé, le serveur Tomcat / Eclipse va chercher à charger le contexte des applications qu'il gère. Charger le contexte d'une application implique d'exploiter son fichier [web.xml] et de charger une ou plusieurs classes l'initialisant. Plusieurs types d'erreurs peuvent alors se produire :

- le fichier [web.xml] est syntaxiquement incorrect. C'est l'erreur la plus fréquente. Il est conseillé d'utiliser un outil capable de vérifier la validité d'un document XML lors de sa construction.
- certaines classes à charger n'ont pas été trouvées. Elles sont cherchées dans [WEB-INF/classes] et [WEB-INF/lib]. Il faut en général vérifier la présence des classes nécessaires et l'orthographe de celles déclarées dans le fichier [web.xml].

Le serveur lancé à partir d'Eclipse n'a pas la même configuration que celui installé au paragraphe 5.3, page 221. Pour nous en assurer, demandons l'url [<http://localhost:8080>] avec un navigateur :



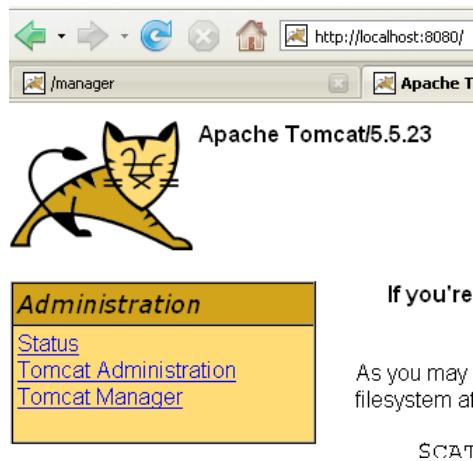
Cette réponse n'indique pas que le serveur ne fonctionne pas mais que la ressource / qui lui est demandée n'est pas disponible. Avec le serveur Tomcat intégré à Eclipse, ces ressources vont être des projets web. Nous le verrons ultérieurement. Pour le moment, arrêtons Tomcat :



Le mode de fonctionnement précédent peut être changé. Revenons dans la vue [Servers] et double-cliquons sur le serveur Tomcat pour avoir accès à ses propriétés :

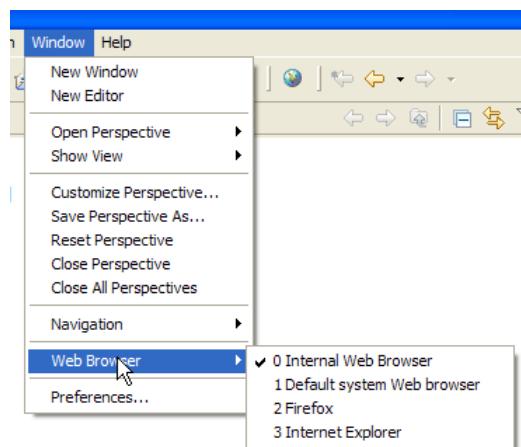
La case à cocher [1] est responsable du fonctionnement précédent. Lorsqu'elle est cochée, les applications web développées sous Eclipse ne sont pas déclarées dans les fichiers de configuration du serveur Tomcat associé mais dans des fichiers de configuration séparés. Ce faisant, on ne dispose pas des applications définies par défaut au sein du serveur Tomcat : [admin] et [manager] qui sont deux applications utiles. Aussi, allons-nous décocher [1] et relancer Tomcat :

Ceci fait, demandons l'url [<http://localhost:8080>] avec un navigateur :



Nous retrouvons le fonctionnement décrit au paragraphe 5.3.4, page 227.

Nous avons, dans nos exemples précédents, utilisé un navigateur extérieur à Eclipse. On peut également utiliser un navigateur interne à Eclipse :



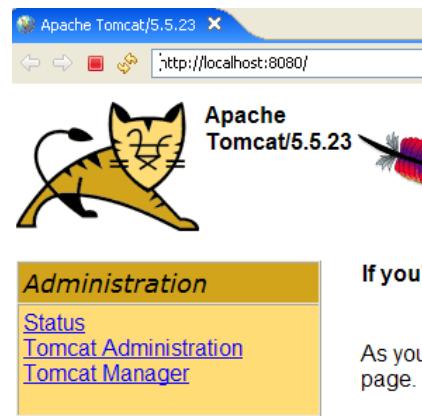
Nous sélectionnons ci-dessus, le navigateur interne. Pour le lancer à partir d'Eclipse on peut utiliser l'icône suivante :



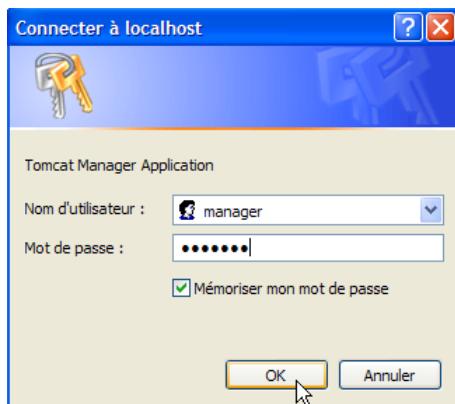
Le navigateur réellement lancé sera celui sélectionné par l'option [Window -> Web Browser]. Ici, nous obtenons le navigateur interne :



Lançons si besoin est Tomcat à partir d'Eclipse et demandons dans [1] l'url [<http://localhost:8080>] :



Suivons le lien [Tomcat Manager] :



Le couple [login / mot de passe] nécessaire pour accéder à l'application [manager] est demandé. D'après la configuration de Tomcat que nous avons faite précédemment, on peut saisir [admin / admin] ou [manager / manager]. On obtient alors la liste des applications déployées :

/manager		
http://localhost:8080/manager/html		
/admin	Tomcat Administration Application	true
/balancer	Tomcat Simple Load Balancer Example App	true
/exemple	Application Exemple	true
/host-manager	Tomcat Manager Application	true

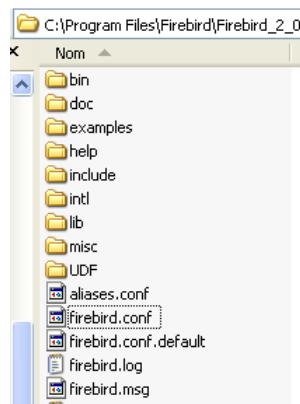
## 5.4 Le SGBD Firebird

### 5.4.1 SGBD Firebird

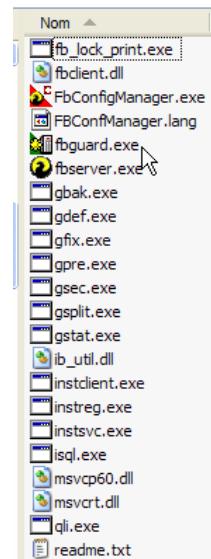
Le SGBD Firebird est disponible à l'url [<http://www.firebirdsql.org/>] :

- en [1] : on utilise l'option [Download.Firebird Relational Database]
- en [2] : on désigne la version de Firebird désirée
- en [3] : on télécharge le binaire de l'installation

Une fois le fichier [3] téléchargé, on double-clique dessus pour installer le SGBD Firebird. Le SGBD est installé dans un dossier dont le contenu est analogue au suivant :

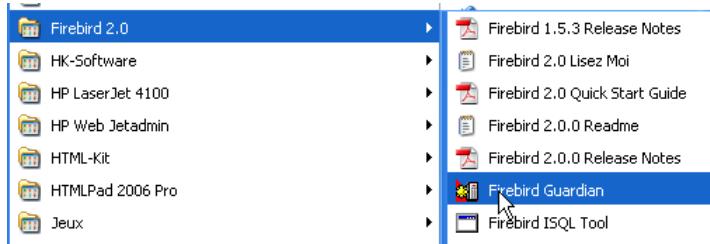


Les binaires sont dans le dossier [bin] :

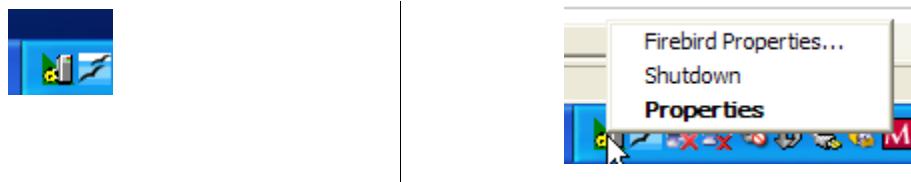


**fbguard.exe** permet de lancer/arrêter le SGBD  
**isql.exe** client ligne permettant de gérer des bases de données

On notera que par défaut, l'administrateur du SGBD s'appelle [SYSDBA] et son mot de passe est [masterkey]. Des menus ont été installés dans [Démarrer] :



L'option [Firebird Guardian] permet de lancer/arrêter le SGBD. Après le lancement, l'icône du SGBD reste dans la barre des tâches de windows :

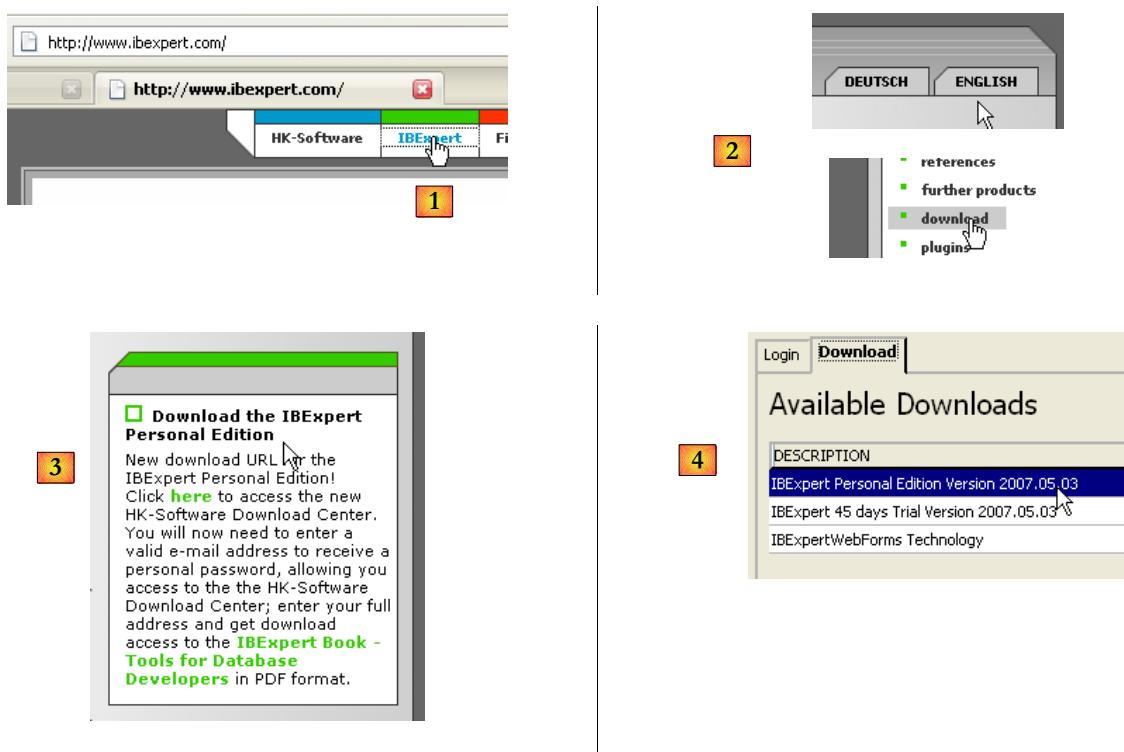


Pour créer et exploiter des bases de données Firebird avec le client ligne [sql.exe], il est nécessaire de lire la documentation livrée avec le produit et qui est accessible via des raccourcis de Firebird dans [Démarrer/Programmes/Firebird 2.0].

Une façon rapide de travailler avec Firebird et d'apprendre le langage SQL est d'utiliser un client graphique. Un tel client est IB-Expert décrit au paragraphe suivant.

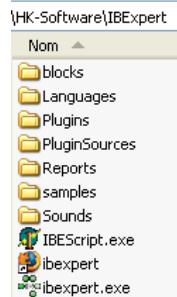
## 5.4.2 Travailler avec le SGBD Firebird avec IB-Expert

Le site principal de IB-Expert est [<http://www.ibexpert.com/>].



- en [1], on sélectionne IBExpert
- en [2], on sélectionne le téléchargement après avoir éventuellement choisi la langue de son choix
- en [3], on sélectionne la version dite "personnelle" car elle est gratuite. Il faut néanmoins s'inscrire auprès du site.
- en [4], on télécharge IBExpert

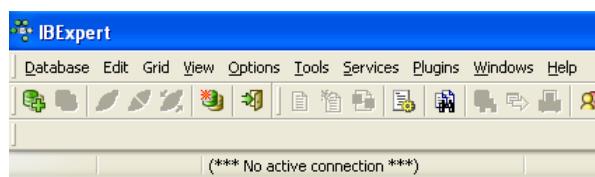
IBExpert est installé dans un dossier analogue au suivant :



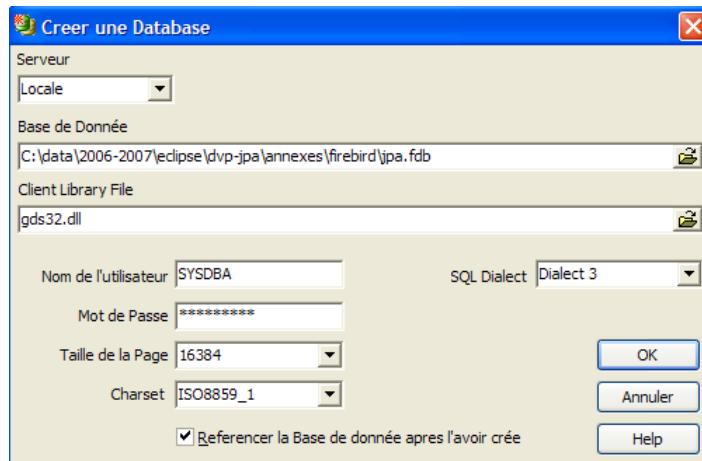
L'exécutable est [ibexpert.exe]. Un raccourci est normalement disponible dans le menu [Démarrer] :



Une fois lancé, IBExpert affiche la fenêtre suivante :



Utilisons l'option [Database/Create Database] pour créer une base de données :



Server (Serveur)
Database (Base de données)
Username
Password (Nom de l'utilisateur)
Dialect
Register Database (Référencer la base de données)

peut être [local] ou [remote]. Ici notre serveur est sur la même machine que [IBExpert]. On choisit donc [local]

utiliser le bouton de type [dossier] du combo pour désigner le fichier de la base. Firebird met toute la base dans un unique fichier. C'est l'un de ses atouts. On transporte la base d'un poste à l'autre par simple copie du fichier. Le suffixe [.fdb] est ajouté automatiquement.

**SYSDBA** est l'administrateur par défaut des distributions actuelles de Firebird

**masterkey** est le mot de passe de l'administrateur SYSDBA des distributions actuelles de Firebird

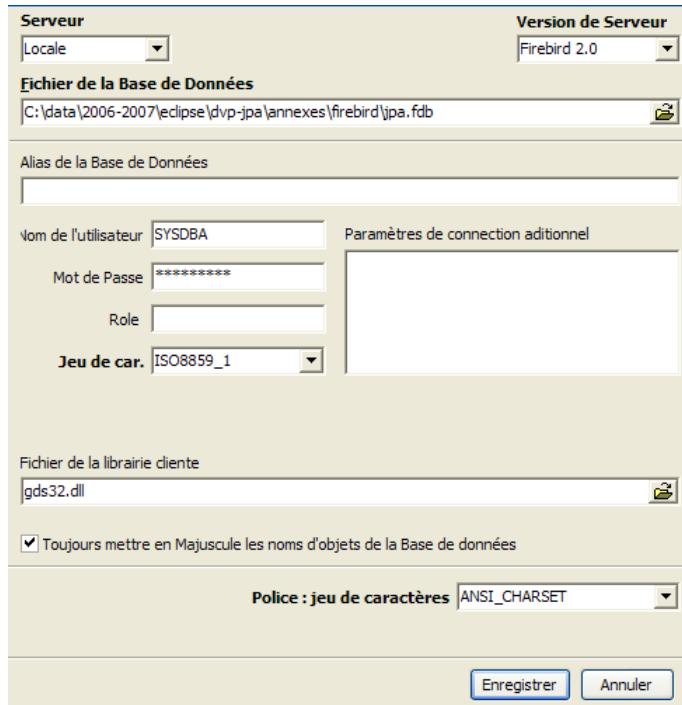
le dialecte SQL à utiliser

si la case est cochée, IBExpert présentera un lien vers la base créée après avoir créé celle-ci

Si en cliquant le bouton [OK] de création, vous obtenez l'avertissement suivant :



c'est que vous n'avez pas lancé Firebird. Lancez-le. On obtient une nouvelle fenêtre :



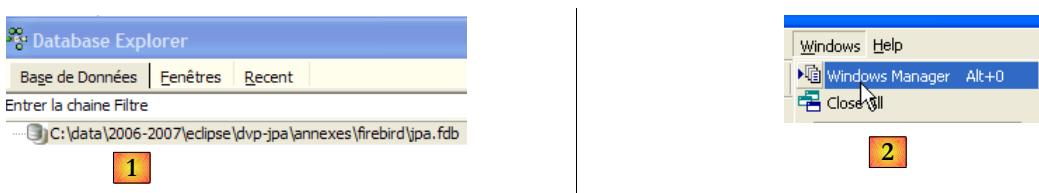
Charset  
(Jeu de caractères)

Famille de caractères à utiliser. Il est conseillé de prendre dans la liste déroulante la famille [ISO-8859-1] qui permet d'utiliser les caractères latins accentués.

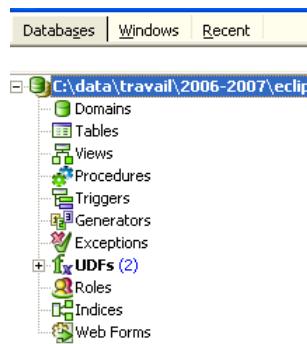
Server version  
(Version du serveur)

[IBExpert] est capable de gérer différents SGBD dérivés d'Interbase. Prendre la version de Firebird que vous avez installée.

Une fois cette nouvelle fenêtre validée par [Register], on a le résultat [1] dans la fenêtre [Database Explorer]. Cette fenêtre peut être fermée malencontreusement. Pour l'obtenir de nouveau faire [2] :

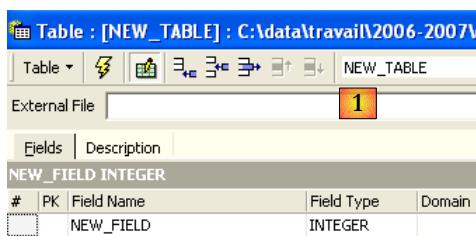


Pour avoir accès à la base créée, il suffit de double-cliquer sur son lien. IBExpert expose alors une arborescence donnant accès aux propriétés de la base :

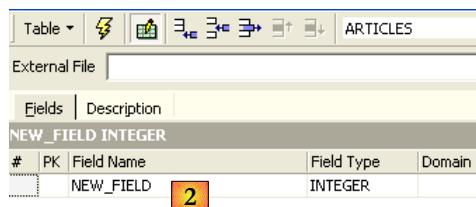


### 5.4.3 Crédation d'une table de données

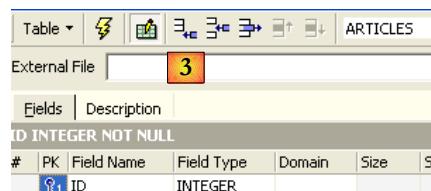
Créons une table. On clique droit sur [Tables] (cf fenêtre ci-dessus) et on prend l'option [New Table]. On obtient la fenêtre de définition des propriétés de la table :



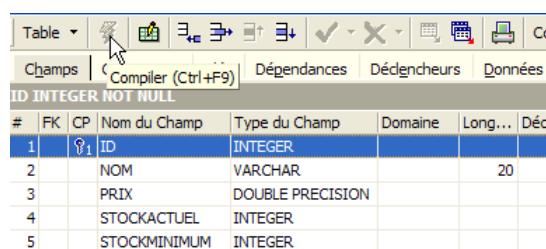
Commençons par donner le nom [ARTICLES] à la table en utilisant la zone de saisie [1] :



Utilisons la zone de saisie [2] pour définir une clé primaire [ID] :



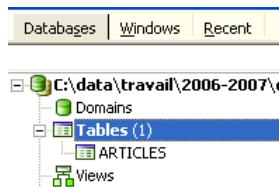
Un champ est fait clé primaire par un double-clic sur la zone [PK] (Primary Key) du champ. Ajoutons des champs avec le bouton situé au-dessus de [3] :



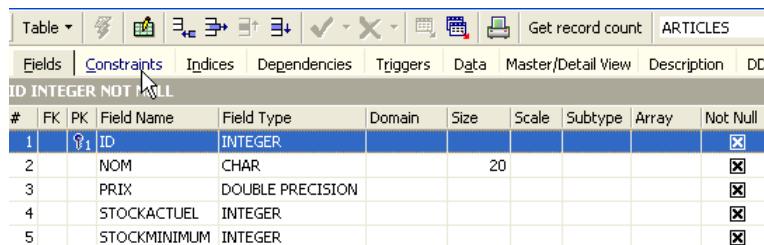
Tant qu'on n'a pas " compilé " notre définition, la table n'est pas créée. Utilisons le bouton [Compile] ci-dessus pour terminer la définition de la table. IBExpert prépare les requêtes SQL de génération de la table et demande confirmation :



De façon intéressante, IBExpert affiche les requêtes SQL qu'il a exécutées. Cela permet un apprentissage à la fois du langage SQL mais également du dialecte SQL éventuellement propriétaire utilisé. Le bouton [Commit] permet de valider la transaction en cours, [Rollback] de l'annuler. Ici on l'accepte par [Commit]. Ceci fait, IBExpert ajoute la table créée, à l'arborescence de notre base de données :



En double-cliquant sur la table, on a accès à ses propriétés :



Le panneau [Constraints] nous permet d'ajouter de nouvelles contraintes d'intégrité à la table. Ouvrons-le :



On retrouve la contrainte de clé primaire que nous avons créée. On peut ajouter d'autres contraintes :

- des clés étrangères [Foreign Keys]
- des contraintes d'intégrité de champs [Checks]
- des contraintes d'unicité de champs [Uniques]

Indiquons que :

- les champs [ID, PRIX, STOCKACTUEL, STOKMINIMUM] doivent être  $>0$
- le champ [NOM] doit être non vide et unique

Ouvrons le panneau [Checks] et cliquons droit dans son espace de définition des contraintes pour ajouter une nouvelle contrainte :

1.Primary key	2.Foreign keys	3.Checks	4.Uniques
Constraint Name   Source			



Définissons les contraintes souhaitées :

Constraint Name	Source
CHK_ID	ID>0
CHK_PRIX	PRIX>0
CHK_STOCKACTUEL	STOCKACTUEL>0
CHK_STOCKMINIMUM	STOCKMINIMUM>0
CHK_NOM	NOM<>"

On notera ci-dessus, que la contrainte [NOM<>"] utilise deux apostrophes et non des guillemets. Compilons ces contraintes avec le bouton [Compile] ci-dessus :

Operation	Result	Copy
Creating CHECK Constraint for ARTICLES ...	Committed	<input checked="" type="checkbox"/>
Creating CHECK Constraint for ARTICLES ...	Committed	<input checked="" type="checkbox"/>
Creating CHECK Constraint for ARTICLES ...	Committed	<input checked="" type="checkbox"/>
Creating CHECK Constraint for ARTICLES ...	Committed	<input checked="" type="checkbox"/>
Creating CHECK Constraint for ARTICLES ...	Committed	<input checked="" type="checkbox"/>

Statement

```
alter table ARTICLES
add constraint CHK_NOM
check (NOM<>")
```

Là encore, IBExpert fait preuve de pédagogie en indiquant les requêtes SQL qu'il a exécutées. Passons maintenant au panneau [Constraints/Uniques] pour indiquer que le nom doit être unique. Cela signifie qu'on ne peut pas avoir deux fois le même nom dans la table.

Constraint Name	On Field
UNQ_NOM	NOM

Définissons la contrainte :

Constraint Name	On Field	Index Name	Index Sorting
UNQ_NOM	NOM		

Puis compilons-la. Ceci fait, ouvrons le panneau [DDL] (Data Definition Language) de la table [ARTICLES] :

```
V*****
***** Generated by IBExpert 16/05/2007 11:17:55 *****
*****
```

Celui-ci donne le code SQL de génération de la table avec toutes ses contraintes. On peut sauvegarder ce code dans un script afin de le rejouer ultérieurement :

```
SET SQL DIALECT 3;
```

```

SET NAMES ISO8859_1;
CREATE TABLE ARTICLES (
    ID          INTEGER NOT NULL,
    NOM         VARCHAR(20) NOT NULL,
    PRIX        DOUBLE PRECISION NOT NULL,
    STOCKACTUEL INTEGER NOT NULL,
    STOCKMINIMUM INTEGER NOT NULL
);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_ID check (ID>0);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX check (PRIX>0);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKACTUEL check (STOCKACTUEL>0);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKMINIMUM check (STOCKMINIMUM>0);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM check (NOM<>'');
ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM UNIQUE (NOM);
ALTER TABLE ARTICLES ADD CONSTRAINT PK_ARTICLES PRIMARY KEY (ID);

```

## 5.4.4 Insertion de données dans une table

Il est maintenant temps de mettre des données dans la table [ARTICLES]. Pour cela, utilisons son panneau [Data] :

The screenshot shows the 'Table : [ARTICLES]' window. The table has columns: ID, NOM, PRIX, STOCKACTUEL, and STOCKMINIMUM. Row 1 contains values: 2, article2, 200,000, 20, 2. Row 2 contains values: 1, article1, 100,000, 10, 1. A cursor is visible over the 'NOM' column of the second row.

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
2	article2	200,000	20	2
1	article1	100,000	10	1

Les données sont entrées par un double-clic sur les champs de saisie de chaque ligne de la table. Une nouvelle ligne est ajoutée avec le bouton [+], une ligne supprimée avec le bouton [-]. Ces opérations se font dans une transaction qui est validée par le bouton [Commit Transaction] (cf ci-dessus). Sans cette validation, les données seront perdues.

## 5.4.5 L'éditeur SQL de [IB-Expert]

Le langage SQL (Structured Query Language) permet à un utilisateur de :

1. créer des tables en précisant le type de données qu'elle va stocker, les contraintes que ces données doivent vérifier
2. d'y insérer des données
3. d'en modifier certaines
4. d'en supprimer d'autres
5. d'en exploiter le contenu pour obtenir des informations
6. ...

IBExpert permet à un utilisateur de faire les opérations 1 à 4 de façon graphique. Nous venons de le voir. Lorsque la base contient de nombreuses tables avec chacune des centaines de lignes, on a besoin de renseignements difficiles à obtenir visuellement. Supposons par exemple qu'un magasin virtuel sur le web ait des milliers d'acheteurs par mois. Tous les achats sont enregistrés dans une base de données. Au bout de six mois, on découvre qu'un produit « X » est défaillant. On souhaite contacter toutes les personnes qui l'ont acheté afin qu'elles renvoient le produit pour un échange gratuit. Comment trouver les adresses de ces acheteurs ?

1. On peut consulter visuellement toutes les tables et chercher ces acheteurs. Cela prendra quelques heures.
2. On peut émettre un ordre SQL qui va donner la liste de ces personnes en quelques secondes

Le langage SQL est utile dès

- que la quantité de données dans les tables est importante
- qu'il y a beaucoup de tables liées entre-elles
- que l'information à obtenir est répartie sur plusieurs tables
- ...

Nous présentons maintenant l'éditeur SQL d'IBExpert. Celui-ci est accessible via l'option [Tools/SQL Editor] ou [F12] :



On a alors accès à un éditeur de requêtes SQL évolué avec lequel on peut jouer des requêtes. Tapons une requête :



On exécute la requête SQL avec le bouton [Execute] ci-dessus. On obtient le résultat suivant :

SQL Editor	
Edit	Results
History	Plan Analyzer
Performance Analysis	Logs
Execute (F9)	

NOM	PRIX
article1	100,000

Ci-dessus, l'onglet [Results] présente la table résultat de l'ordre SQL [Select]. Pour émettre une nouvelle commande SQL, il suffit de revenir sur l'onglet [Edit]. On retrouve alors l'ordre SQL qui a été joué.



Plusieurs boutons de la barre d'outils sont utiles :

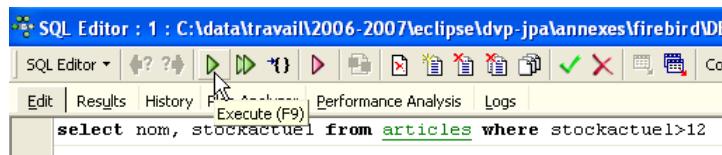
- le bouton [New Query] permet de passer à une nouvelle requête SQL :



On obtient alors une page d'édition vierge :

SQL Editor	
Edit	Results
History	Plan Analyzer
Performance Analysis	Logs
New Query (Ctrl+N)	

On peut alors saisir un nouvel ordre SQL :



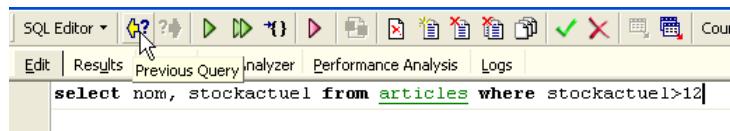
et l'exécuter :

SQL Editor	
Edit	Results
History	Plan Analyzer
Performance Analysis	Logs
Execute (F9)	

NOM	STOCKACTUEL
article2	20

Revenons sur l'onglet [Edit]. Les différents ordres SQL émis sont mémorisés par [IBExpert]. Le bouton [Previous Query] permet de revenir à un ordre SQL émis antérieurement :



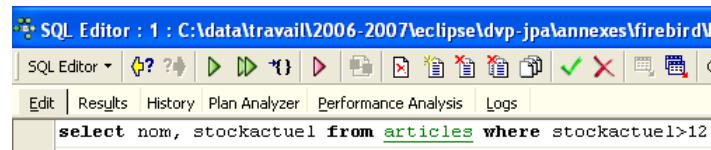
On revient alors à la requête précédente :



Le bouton [Next Query] permet lui d'aller à l'ordre SQL suivant :



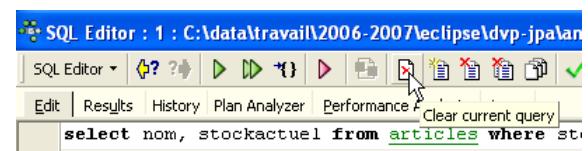
On retrouve alors l'ordre SQL qui suit dans la liste des ordres SQL mémorisés :



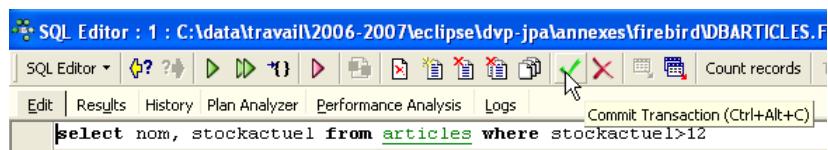
Le bouton [Delete Query] permet de supprimer un ordre SQL de la liste des ordres mémorisés :



Le bouton [Clear Current Query] permet d'effacer le contenu de l'éditeur pour l'ordre SQL affiché :



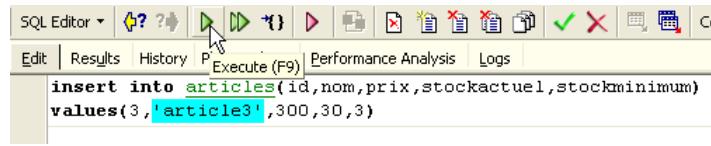
Le bouton [Commit] permet de valider définitivement les modifications faites à la base de données :



Le bouton [RollBack] permet d'annuler les modifications faites sur la base depuis le dernier [Commit]. Si aucun [Commit] n'a été fait depuis la connexion à la base, alors ce sont les modifications faites depuis cette connexion qui sont annulées.



Prenons un exemple. Insérons une nouvelle ligne dans la table :



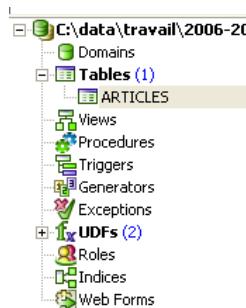
L'ordre SQL est exécuté mais aucun affichage ne se produit. On ne sait pas si l'insertion a eu lieu. Pour le savoir, exécutons l'ordre SQL suivant [New Query] :



On obtient [Execute] le résultat suivant :

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100,000	10	1
2	article2	200,000	20	2
3	article3	300,000	30	3

La ligne a donc bien été insérée. Examinons le contenu de la table d'une autre façon maintenant. Double-cliquons sur la table [ARTICLES] dans l'explorateur de bases :



On obtient la table suivante :

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100,000	10	1
2	article2	200,000	20	2

Le bouton fléché ci-dessus permet de rafraîchir la table. Après rafraîchissement, la table ci-dessus ne change pas. On a l'impression que la nouvelle ligne n'a pas été insérée. Revenons à l'éditeur SQL (F12) puis validons l'ordre SQL émis avec le bouton [Commit] :

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100,000	10	1
2	article2	200,000	20	2
3	article3	300,000	30	3

Ceci fait, revenons sur la table [ARTICLES]. Nous pouvons constater que rien n'a changé même en utilisant le bouton [Refresh] :

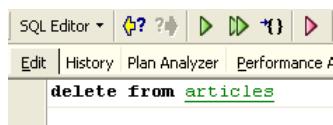
ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100,000	10	1
2	article2	200,000	20	2

Ci-dessus, ouvrons l'onglet [Fields] puis revenons sur l'onglet [Data]. Cette fois-ci la ligne insérée apparaît correctement :

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100,000	10	1
2	article2	200,000	20	2
3	article3	300,000	30	3

Quand commence l'émission des différents ordres SQL, l'éditeur ouvre ce qu'on appelle une **transaction** sur la base. Les modifications faites par ces ordres SQL de l'éditeur SQL ne seront visibles que tant qu'on reste dans le même éditeur SQL (on peut en ouvrir plusieurs). Tout se passe comme si l'éditeur SQL travaillait non pas sur la base réelle mais sur une copie qui lui est propre. Dans la réalité, ce n'est pas exactement de cette façon que cela se passe mais cette image peut nous aider à comprendre la notion de transaction. Toutes les modifications apportées à la copie au cours d'une transaction ne seront visibles dans la base réelle que lorsqu'elles auront été validées par un [Commit Transaction]. La transaction courante est alors terminée et une nouvelle transaction commence.

Les modifications apportées au cours d'une transaction peuvent être annulées par une opération appelée [Rollback]. Faisons l'expérience suivante. Commençons une nouvelle transaction (il suffit de faire [Commit] sur la transaction courante) avec l'ordre SQL suivant :



Exécutons cet ordre qui supprime toutes les lignes de la table [ARTICLES], puis exécutons [New Query] le nouvel ordre SQL suivant :



Nous obtenons le résultat suivant :

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
<null>	<null>	<null>	<null>	<null>

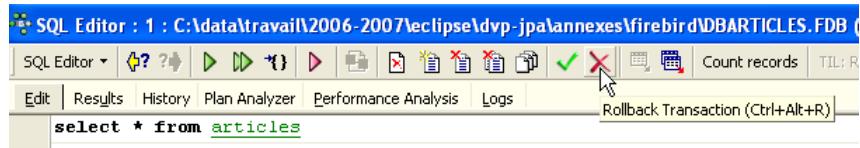
Toutes les lignes ont été détruites. Rappelons-nous que cela a été fait sur une copie de la table [ARTICLES]. Pour le vérifier, double-cliquons sur la table [ARTICLES] ci-dessous :



et visualisons l'onglet [Data] :

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100,000	10	1
2	article2	200,000	20	2
3	article3	300,000	30	3

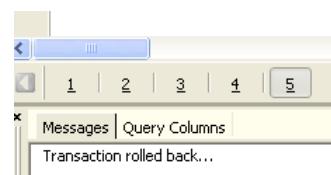
Même en utilisant le bouton [Refresh] ou en passant à l'onglet [Fields] pour revenir ensuite à l'onglet [Data], le contenu ci-dessus ne bouge pas. Ceci a été expliqué. Nous sommes dans une autre transaction qui travaille sur sa propre copie. Maintenant revenons à l'éditeur SQL (F12) et utilisons le bouton [RollBack] pour annuler les suppressions de lignes qui ont été faites :



Confirmation nous est demandée :



Confirmons. L'éditeur SQL confirme que les modifications ont été annulées :



Rejouons la requête SQL ci-dessus pour vérifier. On retrouve les lignes qui avaient été supprimées :

A screenshot of the SQL Editor displaying the results of a SELECT query. The table has columns: ID, NOM, PRIX, STOCKACTUEL, and STOCKMINIMUM. The data shows three rows: article1 (ID 1), article2 (ID 2), and article3 (ID 3). The 'STOCKACTUEL' column shows values 10, 20, and 30 respectively, while the 'STOCKMINIMUM' column shows values 1, 2, and 3.

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100,000	10	1
2	article2	200,000	20	2
3	article3	300,000	30	3

L'opération [Rollback] a ramené la copie sur laquelle travaille l'éditeur SQL, dans l'état où elle était au début de la transaction.

#### 5.4.6 Exportation d'une base Firebird vers un script SQL

Lorsqu'on travaille avec divers SGBD, comme c'est le cas dans le tutoriel " Persistance Java 5 par la pratique ", il est intéressant de pouvoir exporter une base à partir d'un SGBD 1 vers un script SQL pour importer ensuite ce dernier dans un SGBD 2. Cela évite un certain nombre d'opérations manuelles. Ce n'est cependant pas toujours possible, les SGBD ayant souvent des extensions SQL propriétaires.

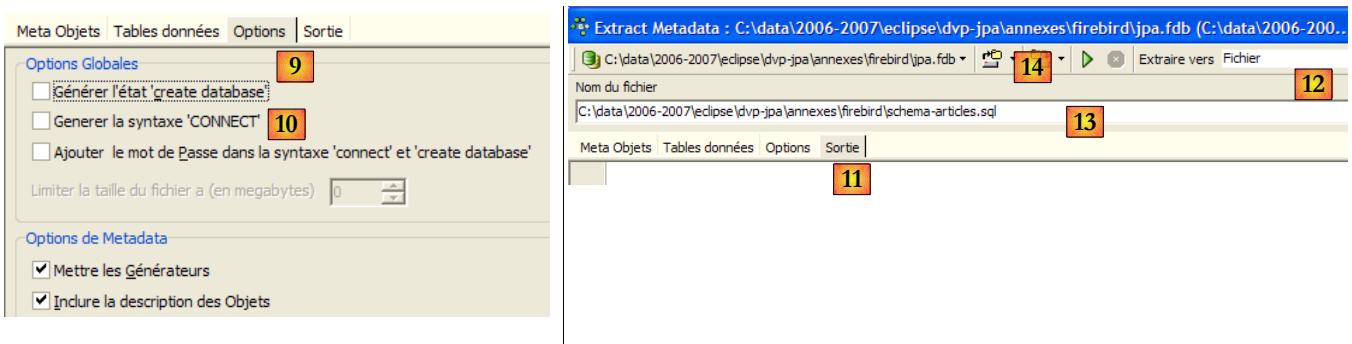
Montrons comment exporter la base [dbarticles] précédente vers un script SQL :

This screenshot shows the Eclipse IDE interface with the 'SQL Editor' open. On the left, the toolbar includes 'Outils', 'Services', 'Plugins', 'Fenêtres', and 'Aide'. Below these are various tools like 'Editeur SQL', 'Nouvel Editeur SQL', 'Créateur de requête', etc. The 'Extract Metadata' dialog is open on the right, showing the path 'C:\data\2006-2007\eclipse\dvp-jpa\annexes\firebird\DBARTICLES.FDB'. The 'Tables données' tab is selected, showing a list of objects: 'Domaines', 'Tables (1)', 'Vues', 'Procédures', 'Déclencheurs', 'Générateurs', 'Exceptions', 'UDFs', and 'Roles'. The 'Tables (1)' item is highlighted. The 'Objets Sélectionnés' panel on the right lists the selected objects: 'Tables', 'Vues', 'Procédures', 'Déclencheurs', 'Générateurs', 'Exceptions', 'UDFs', and 'Roles'. A blue box labeled '1' is over the 'Extract Metadata' button in the toolbar, and another blue box labeled '2' is over the 'Tables (1)' entry in the tree view.

- en [1] : Outils / Extraire la MetaData, pour extraire les méta-données
- en [2] : onglet Méta Objets
- en [3] : sélectionner la table [Articles] dont on veut extraire la structure (méta-données)
- en [4] : pour transférer à droite l'objet sélectionné à gauche



- en [5] : la table [ARTICLES] fera partie des méta-données extraites
- en [6] : l'onglet [Table de données] sert à sélectionner les tables dont on veut extraire le contenu (dans l'étape précédente, c'était la structure de la table qui était exportée)
- en [7] : pour transférer à droite l'objet sélectionné à gauche
- en [8] : le résultat obtenu



- en [9] : l'onglet [Options] permet de configurer certains paramètres de l'extraction
- en [10] : on décoche les options liées à la génération des ordres SQL permettant de se connecter à la base. Ils sont propriétaires à Firebird et de ce fait ne nous intéressent pas.
- en [11] : l'onglet [Sortie] permet de préciser où sera généré le script SQL
- en [12] : on précise que le script doit être généré dans un fichier
- en [13] : on précise l'emplacement de ce fichier
- en [14] : on lance la génération du script SQL

Le script généré, débarrassé de ses commentaires est le suivant :

```

1. SET SQL DIALECT 3;
2. SET NAMES ISO8859_1;
3.
4. CREATE TABLE ARTICLES (
5.     ID          INTEGER NOT NULL,
6.     NOM         VARCHAR(20) NOT NULL,
7.     PRIX        DOUBLE PRECISION NOT NULL,
8.     STOCKACTUEL INTEGER NOT NULL,
9.     STOCKMINIMUM INTEGER NOT NULL
10. );
11.
12. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (1, 'article1', 100, 10,
13. 1);
13. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (2, 'article2', 200, 20,
14. 2);
14. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (3, 'article3', 300, 30,
15. 3);
15.
16. COMMIT WORK;
17.
18. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_ID check (ID>0);
19. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX check (PRI>0);

```

```

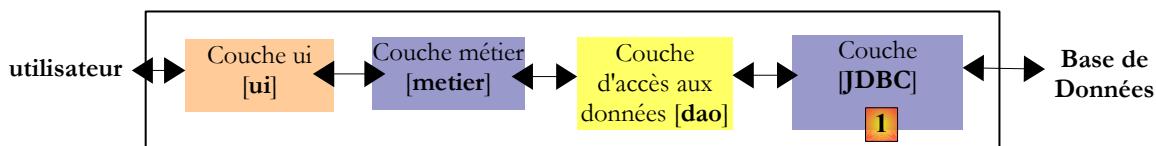
20. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKACTUEL check (STOCKACTUEL>0);
21. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKMINIMUM check (STOCKMINIMUM>0);
22. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM check (NOM<>'');
23. ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM UNIQUE (NOM);
24. ALTER TABLE ARTICLES ADD CONSTRAINT PK_ARTICLES PRIMARY KEY (ID);

```

Note : les lignes 1-2 sont propres à Firebird. Elles doivent être supprimées du script généré afin d'avoir du SQL générique.

### 5.4.7 Pilote JDBC de Firebird

Un programme Java accède aux données d'une base de données via un pilote JDBC propre au SGBD utilisé :



Dans une architecture multi-couches comme celle ci-dessus, le pilote JDBC [1] est utilisé par la couche [dao] (Data Access Object) pour accéder aux données d'une base de données.

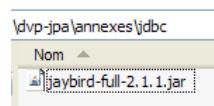
Le pilote JDBC de Firebird est disponible à l'url où Firebird a été téléchargé :

Two screenshots illustrating the download process:

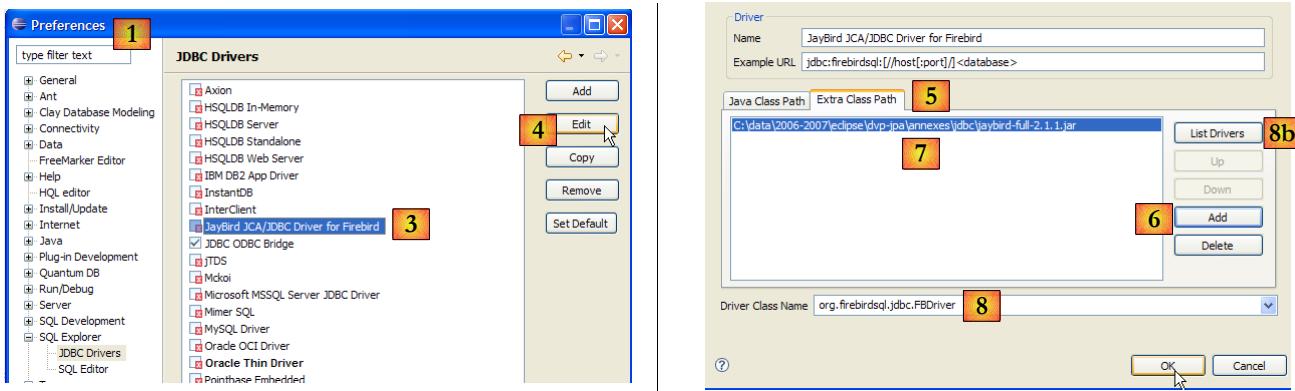
- Left Screenshot (Firebird Download Page):** Shows the "Download" section with "Firebird class 4 JCA-JDBC Driver" highlighted (marked with a red box labeled 1). Below it is a link to "All released packages (SourceForge)".
- Right Screenshot (Jaybird 2.1.1 Release Page):** Shows the "Downloads" section for version 2.1.1. It lists three JDBC drivers for JDK 1.3, 1.4, and 1.5, each with a download link and file size. The "Jaybird 2.1.1 for JDK 1.5 (.zip) (2.9 Mb)" link is highlighted (marked with a red box labeled 2).
- Bottom Screenshot (File Explorer):** Shows the downloaded files in a Windows File Explorer window. The "jaybird-full-2.1.1.jar" file is selected (marked with a red box labeled 3).

- en [1] : on choisit de télécharger le pilote JDBC
- en [2] : on choisit un pilote JDBC compatible JDK 1.5
- en [3] : l'archive contenant le pilote JDBC est [jaybird-full-2.1.1.jar]. On extraiera ce fichier. Il sera utilisé pour tous les exemples JPA avec Firebird.

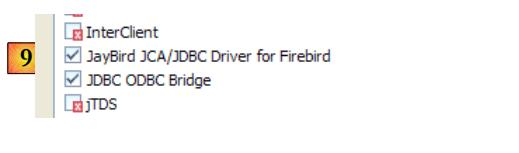
Nous le plaçons dans un dossier que nous appellerons par la suite <jdbc> :



Pour vérifier ce pilote JDBC, nous allons utiliser Eclipse et le plugin SQL Explorer (paragraphe 5.2.6, page 219). Nous commençons par déclarer le pilote JDBC de Firebird :



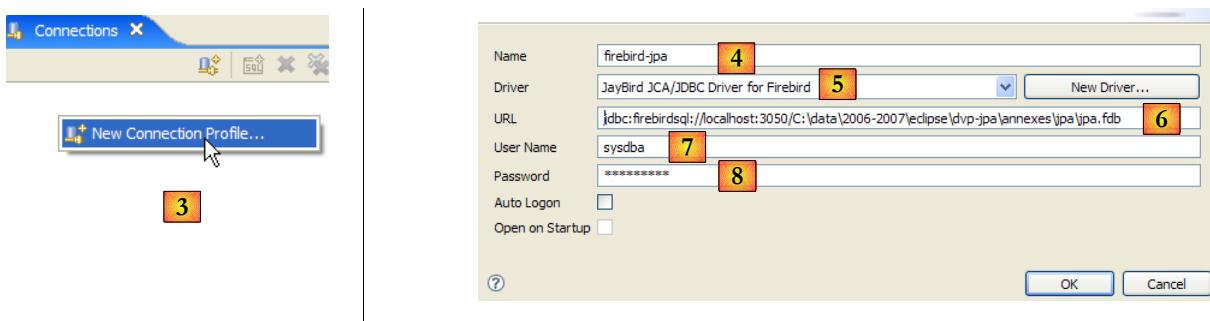
- en [1] : faire Window / Preferences
- en [2] : choisir l'option SQL Explorer / JDBC Drivers
- en [3] : choisir le pilote JDBC pour Firebird
- en [4] : passer en phase de configuration
- en [5] : passer dans l'onglet [Extra Class Path]
- avec [6], désigner le fichier du pilote JDBC. Ceci fait, celui-ci apparaît en [7]. On choisira ici le pilote placé précédemment dans le dossier <jdbc>
- dans [8] : le nom de la classe Java du pilote JDBC. Il peut être obtenu par le bouton [8b].
- on fait [OK] pour valider la configuration



- en [9] : le pilote JDBC de Firebird est désormais configuré. On peut passer à son utilisation.



- en [1] : ouvrir une nouvelle perspective
- en [2] : choisir la perspective [SQL Explorer]



- en [3] : créer une nouvelle connexion
- en [4] : lui donner un nom
- en [5] : choisir dans la liste déroulante le pilote JDBC de Firebird
- en [6] : préciser l'Url de la base à laquelle on veut se connecter, ici : [jdbc:firebirdsql:localhost:3050:C:\data\2006-2007\eclipse\dvp-jpa\annexes\jpa\fdb]. [jpa.fdb] est la base créée précédemment avec IBExpert.
- en [7] : le nom de l'utilisateur de la connexion, ici [sysdba], l'administrateur de Firebird
- en [8] : son mot de passe [masterkey]
- on valide la configuration de la connexion par [OK]

The screenshot shows the IBExpert interface divided into two main sections. On the left, the 'Connections' window (1) displays a single connection named 'firebird-jpa'. Below it, the 'Connection Profile firebird-jpa' dialog (2) shows the configuration details: Driver (JayBird JCA/JDBC Driver for Firebird), Url (jdbc:firebirdsql://localhost:3050/C:\data\2006-2007\eclipse\dvp-jpa\annexes\jpa\fdb), User (sysdba), Password (\*\*\*\*\*), AutoCommit checked, Commit On Close checked. On the right, the 'Database Structure' window (3) shows the database hierarchy for 'firebird-jpa'. It includes 'NoCatalog', 'System Table', 'Tables' (with 'ARTICLES' selected, 4), 'Columns' (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM), and 'Indexes'. A 'Views' section is also present.

- en [1] : on double-clique sur le nom de la connexion qu'on veut ouvrir
- en [2] : on s'identifie (sysdba, masterkey)
- en [3] : la connexion est ouverte
- en [4] : on a la structure de la base. On y voit la table [ARTICLES]. On la sélectionne.

The screenshot shows the 'SQL Results' window (5) and 'Database Detail' window (6). The 'Database Detail' window shows the columns of the 'ARTICLES' table: ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM. The 'SQL Results' window shows the preview of the 'ARTICLES' table (7), containing three rows of data: article1 (ID 1, PRIX 100, STOCKACTUEL 10, STOCKMINIMUM 1), article2 (ID 2, PRIX 200, STOCKACTUEL 20, STOCKMINIMUM 2), and article3 (ID 3, PRIX 300, STOCKACTUEL 30, STOCKMINIMUM 3).

- en [5] : dans la fenêtre [Database Detail], on a les détails de l'objet sélectionné en [4], ici la table [ARTICLES]
- en [6] : l'onglet [Columns] donne la structure de la table
- en [7] : l'onglet [Preview] donne la structure de la table

On peut émettre des requêtes SQL dans la fenêtre [SQL Editor] :

The screenshot shows the 'SQL Editor' window (1) with the query 'select nom, prix from ARTICLES where prix<233' entered. The 'SQL Results' window (2) shows the output of the query, which is '2'. The 'Database Detail' window (4) shows the results of the query: article1 (NOM article1, PRIX 100) and article2 (NOM article2, PRIX 200). The 'SQL Results' window (5) also shows the count of 2.

- en [1] : choisir une connexion ouverte
- en [2] : taper l'ordre SQL à exécuter
- en [3] : l'exécuter

- en [4] : rappel de l'ordre exécuté
- en [5] : son résultat

## 5.5 Le SGBD MySQL5

### 5.5.1 Installation

Le SGBD MySQL5 est disponible à l'url [<http://dev.mysql.com/downloads/>] :

The screenshot shows the MySQL Community Server download page. A mouse cursor is hovering over the 'Windows' link in the 'Windows' section of the dropdown menu. The menu also lists 'Windows x64', 'Linux (non RPM packages)', 'Linux (non RPM, Intel C/C++ compiled, glibc-2.3)', 'Red Hat Enterprise Linux 3 RPM (x86)', 'Red Hat Enterprise Linux 3 RPM (AMD64 / Intel EM64T)', and 'Red Hat Enterprise Linux 3 RPM (Intel IA-32)'. Below the dropdown, a link to 'Windows ZIP/Setup.EXE' is highlighted with a red box labeled '3'.

- en [1] : choisir la version désirée
- en [2] : choisir une version Windows

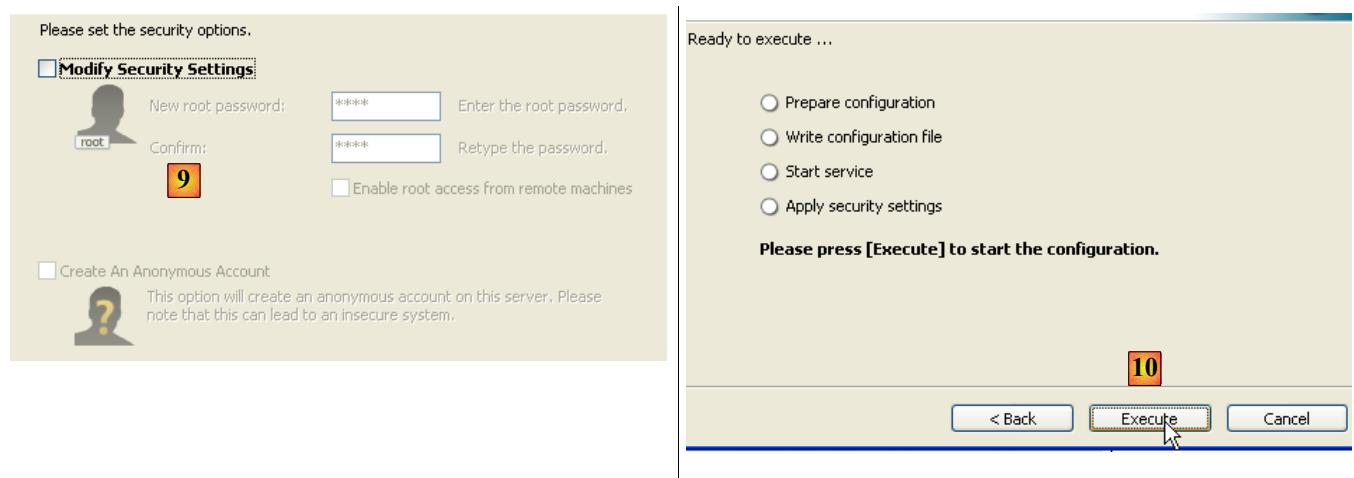
- en [3] : choisir la version windows désirée
- en [4] : le zip téléchargé contient un exécutable [Setup.exe] [4b] qu'il faut extraire et exécuter pour installer MySQL5

The screenshot shows two windows of the MySQL Server 5.0 - Setup Wizard. The left window, titled 'Setup Type', shows the 'Typical' radio button selected (indicated by a red box labeled '5'). The right window, titled 'Wizard Completed', shows the message 'Setup has finished installing MySQL Server 5.0. Click Finish to exit the wizard.' and a checked checkbox for 'Configure the MySQL Server now' (indicated by a red box labeled '6').

- en [5] : choisir une installation typique
- en [6] : une fois l'installation terminée, on peut configurer le serveur MySQL5



- en [7] : choisir une configuration standard, celle qui pose le moins de questions
- en [8] : le serveur MySQL5 sera un service windows



- en [9] : par défaut l'administrateur du serveur est **root** sans mot de passe. On peut garder cette configuration ou donner un nouveau mot de passe à **root**. Si l'installation de MySQL5 vient derrière une désinstallation d'une version précédente, cette opération peut échouer. Il y a moins moyen d'y revenir.
- en [10] : on demande la configuration du serveur

L'installation de MySQL5 donne naissance à un dossier dans [Démarrer / Programmes] :

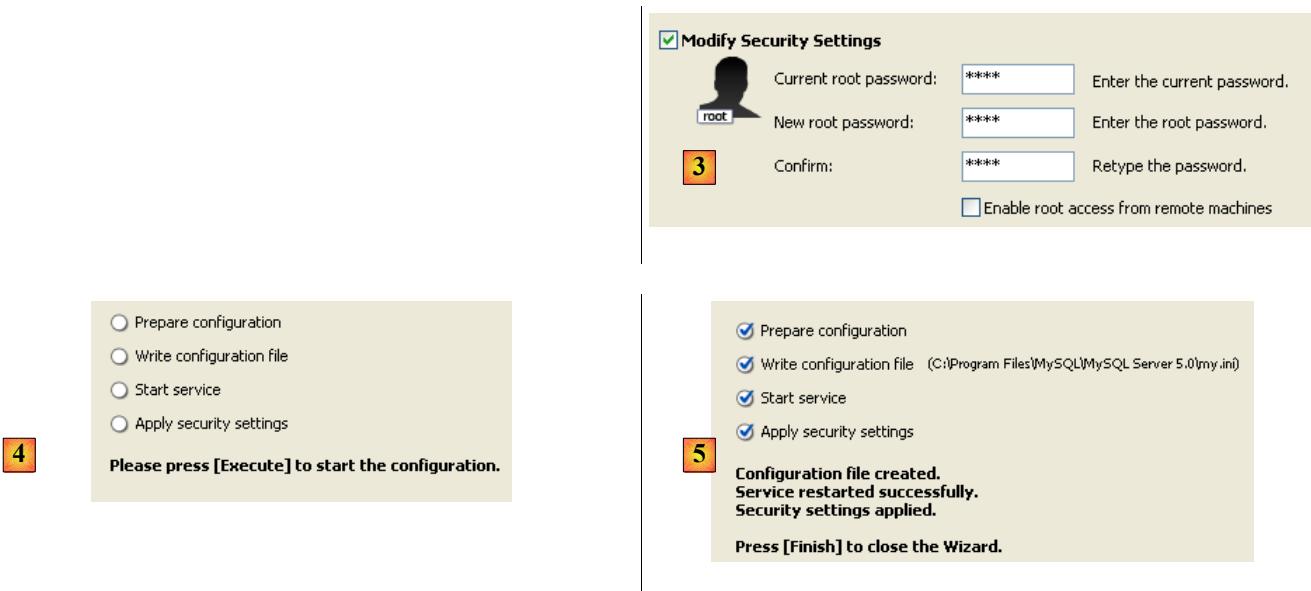


On peut utiliser [MySQL Server Instance Config Wizard] pour reconfigurer le serveur :

#### Welcome to the MySQL Server Instance Configuration Wizard 1.0.8

The Configuration Wizard will allow you to configure the MySQL Server 5.0 server instance. To Continue, click Next.

**Reconfigure Instance**  
Select this option to create a new configuration for the instance. This will replace the current configuration and restart the service if it is currently running.

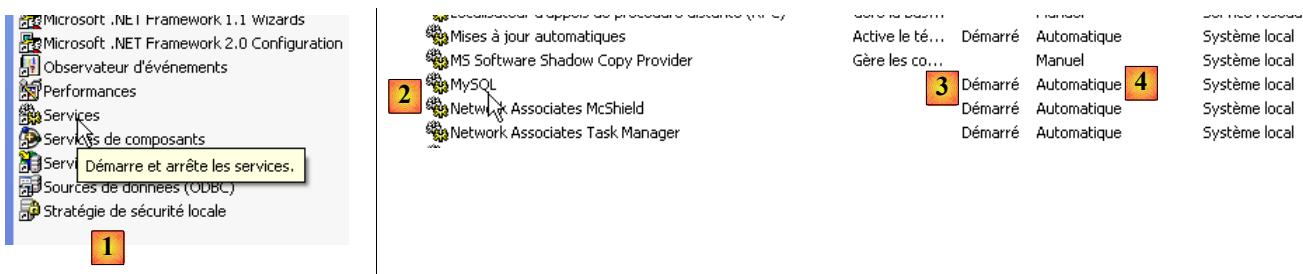


- en [3] : nous changeons le mot de passe de root (ici root/root)

### 5.5.2 Lancer / Arrêter MySQL5

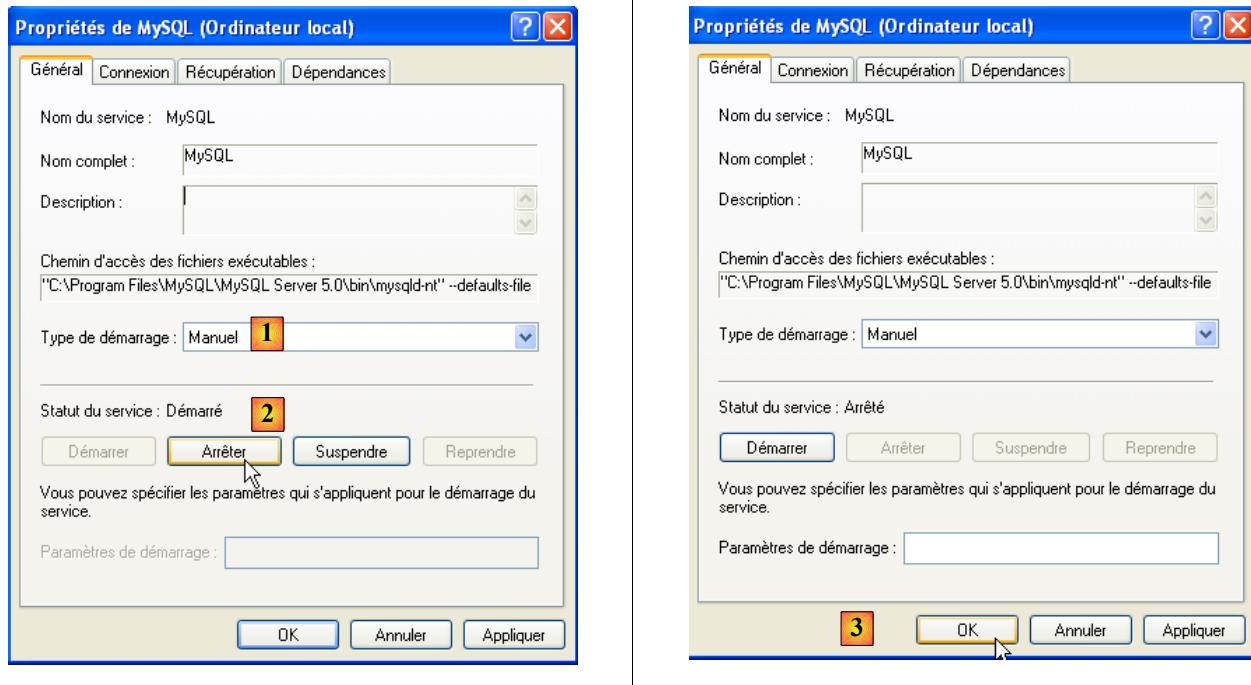
Le serveur MySQL5 a été installé comme un service windows à démarrage automatique, c.a.d lancé dès le démarrage de windows. Ce mode de fonctionnement est peu pratique. Nous allons le changer :

[Démarrer / Panneau de configuration / Performances et maintenance / Outils d'administration / Services] :



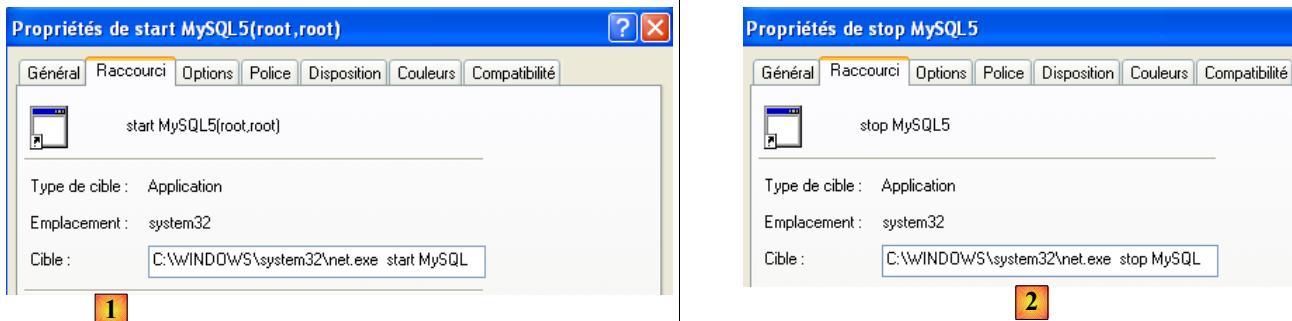
- en [1] : nous double-cliquons sur [Services]
- en [2] : on voit qu'un service appelé [MySQL] est présent, qu'il est démarré [3] et que son démarrage est automatique [4].

Pour modifier ce fonctionnement, nous double-cliquons sur le service [MySQL] :



- en [1] : on met le service en démarrage manuel
- en [2] : on l'arrête
- en [3] : on valide la nouvelle configuration du service

Pour lancer et arrêter manuellement le service MySQL, on pourra créer deux raccourcis :



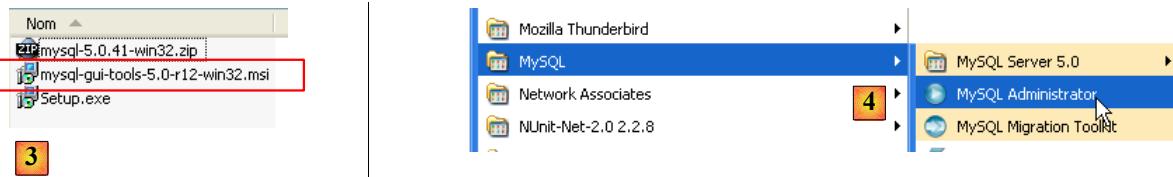
- en [1] : le raccourci pour lancer MySQL5
- en [2] : le raccourci pour l'arrêter

### 5.5.3 Clients d'administration MySQL

Sur le site de MySQL, on peut trouver des clients d'administration du SGBD :

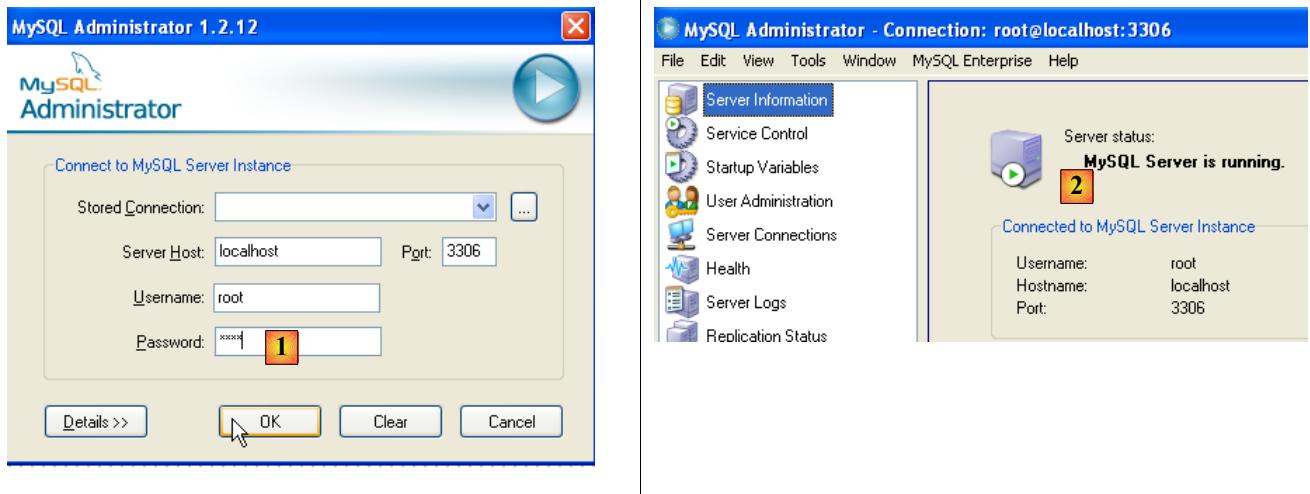
Version	Taille	Mirror
Windows (x86)	5.0-r12 MD5: 7d6d546069554900e6d9a324b6840336	Pick a mirror
Without Installer (unzip in C:\)	5.0-r12 MD5: 9f396065bc095ff73dbd6e478554ee62	Pick a mirror

- en [1] : choisir [MySQL GUI Tools] qui rassemble divers clients graphiques permettant soit d'administrer le SGBD, soit de l'exploiter
- en [2] : prendre la version Windows qui convient



- en [3] : on récupère un fichier .msi à exécuter
- en [4] : une fois l'installation faite, de nouveaux raccourcis apparaissent dans le dossier [Menu Démarrer / Programmes / MySQL].

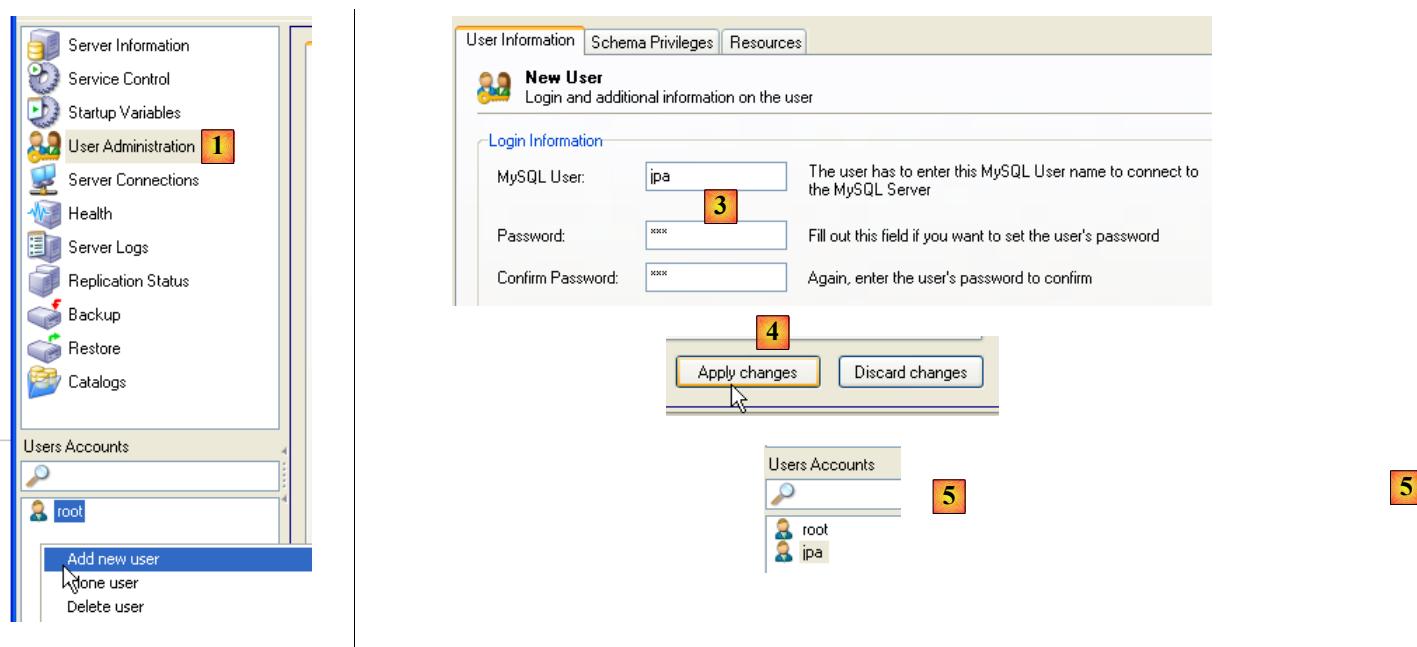
Lançons MySQL (via les raccourcis que vous avez créés), puis lançons [MySQL Administrator] via le menu ci-dessus :



- en [1] : mettre le mot de passe de l'utilisateur root (root ici)
- en [2] : on est connecté et on voit que MySQL est actif

#### 5.5.4 Crédation d'un utilisateur jpa et d'une base de données jpa

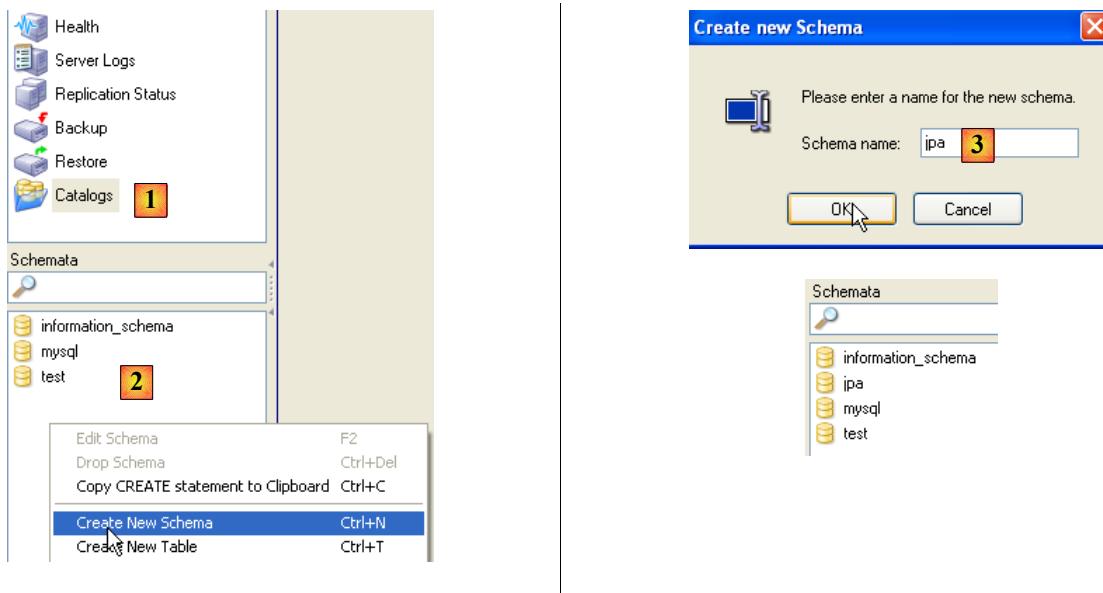
Le tutoriel utilise MySQL5 avec une base de données appelée **jpa** et un utilisateur de même nom. Nous les créons maintenant. D'abord l'utilisateur :



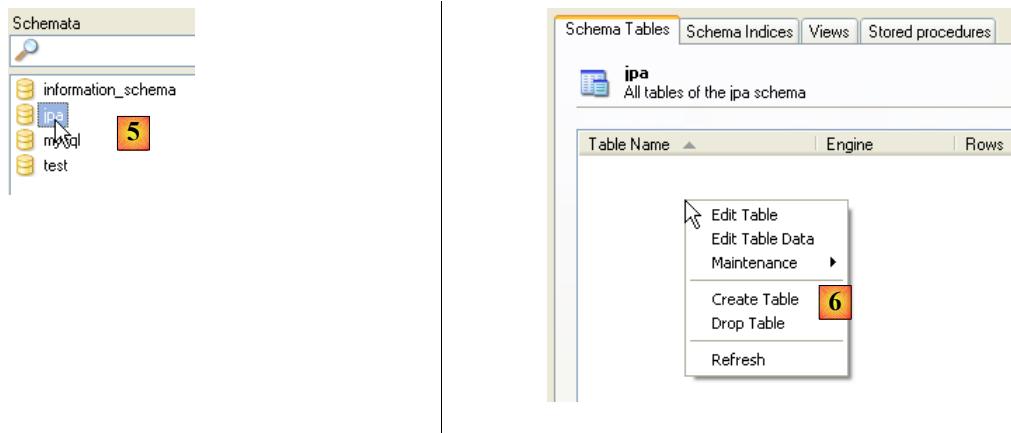
- en [1] : on sélectionne [User Administration]
- en [2] : on clique droit dans la partie [User accounts] pour créer un nouvel utilisateur

- en [3] : l'utilisateur s'appelle **jpa** et son mot de passe est **jpa**
- en [4] : on valide la création
- en [5] : l'utilisateur [jpa] apparaît dans la fenêtre [User Accounts]

La base de données maintenant :



- en [1] : choix de l'option [Catalogs]
- en [2] : clic droit sur la fenêtre [Schemata] pour créer un nouveau schéma (désigne une base de données)
- en [3] : on nomme le nouveau schéma
- en [4] : il apparaît dans la fenêtre [Schemata]



- en [5] : on sélectionne le schéma [jpa]
- en [6] : les objets du schéma [jpa] apparaissent, notamment les tables. Il n'y en a pas encore. Un clic droit permettrait d'en créer. Nous laissons le lecteur le faire.

Revenons à l'utilisateur [jpa] afin de lui donner tous les droits sur le schéma [jpa] :

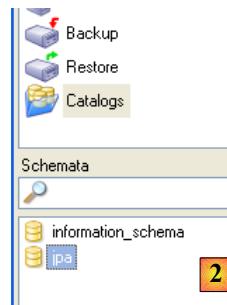
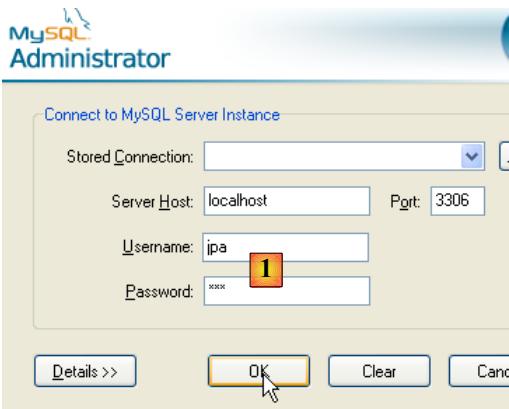
The screenshot shows the MySQL Workbench User Administration interface. On the left, under 'User Administration' [1], there are various tabs like Server Connections, Health, Server Logs, Replication Status, Backup, Restore, Catalogs, and Users Accounts. Under 'Users Accounts' [2], the 'root' and 'ipa' users are listed. The main window shows the 'Schema Privileges' tab [3] for the 'ipa' user. It has three panels: 'Schemata' [4] containing 'information\\_\schema', 'ipa', 'mysql', and 'test'; 'Assigned Privileges' which is currently empty; and 'Available Privileges' listing various MySQL privileges with icons. A cursor [5] is hovering over the '<<' button between the Assigned and Available panels.

- en [1], puis [2] : on sélectionne l'utilisateur [jpa]
- en [3] : on sélectionne l'onglet [Schema Privileges]
- en [4] : on sélectionne le schéma [jpa]
- en [5] : on va donner à l'utilisateur [jpa] tous les privilèges sur le schéma [jpa]

This screenshot shows the same MySQL Workbench interface after step 6. The 'Assigned Privileges' panel [6] now contains all the privileges listed in the 'Available Privileges' panel, indicating that the changes have been applied. The 'Apply changes' button at the bottom right is highlighted.

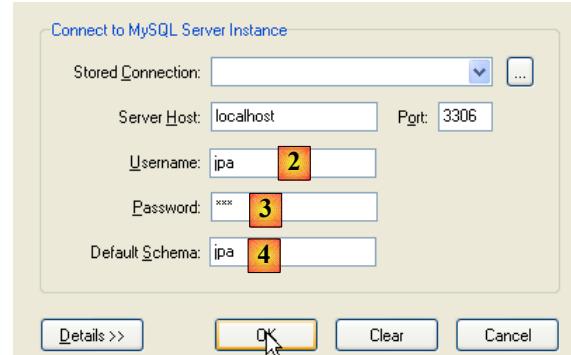
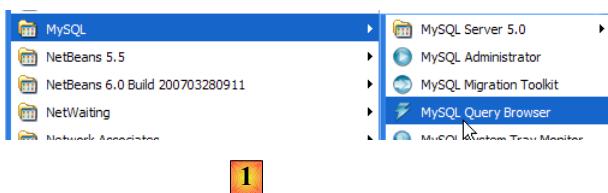
- en [6] : on valide les changements faits

Pour vérifier que l'utilisateur [jpa] peut travailler avec le schéma [jpa], on ferme l'administrateur MySQL. On le relance et on se connecte cette fois sous le nom [jpa/jpa] :

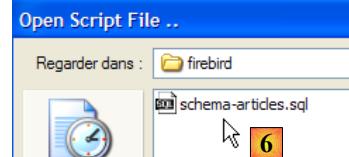
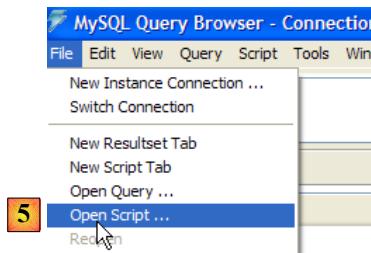


- en [1] : on s'identifie (ipa/jpa)
- en [2] : la connexion a réussi et dans [Schemata], on voit les schémas sur lesquels on a des droits. On voit le schéma [ipa].

Nous allons maintenant créer la même table [ARTICLES] qu'avec le SGBD Firebird en utilisant le script SQL [schema-articles.sql] généré au paragraphe 5.4.6, page 253.



- en [1] : utiliser l'application [MySQL Query Browser]
- en [2], [3], [4] : s'identifier (ipa / jpa / jpa)



- en [5] : ouvrir un script SQL afin de l'exécuter
- en [6] : désigner le script [schema-articles.sql] créé au paragraphe 5.4.6, page 253.

The screenshot shows two panels of MySQL Workbench. The left panel is a 'Script 1' editor containing a SQL script for creating a 'ARTICLES' table and inserting data. The right panel is a 'Schemata' browser showing the database structure of 'ipa'. A yellow box labeled '7' highlights the number '100' in the first row of the 'ARTICLES' table data. A yellow box labeled '8' highlights the 'Execute' button in the top toolbar. A yellow box labeled '9' highlights the 'articles' table in the schema browser.

```

1  /*
2   *****
3   * Tables
4   *****
5
6
7
8 CREATE TABLE ARTICLES (
9     ID          INTEGER NOT NULL,
10    NOM         VARCHAR(20) NOT NULL,
11    PRIX        DOUBLE PRECISION NOT NULL,
12    STOCKACTUEL INTEGER NOT NULL,
13    STOCKMINIMUM INTEGER NOT NULL
14 );
15
16 INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (1, 'article1', 100, 10, 1);
17 INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (2, 'article2', 200, 20, 2);
18 INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (3, 'article3', 300, 30, 3);
19
20 COMMIT WORK;
21
22
23 /* Check constraints definition */
24
25 ALTER TABLE ARTICLES ADD CONSTRAINT CHK_ID_check (ID>0);
26 ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX_check (PRIXT>0);
27 ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKACTUEL_check (STOCKACTUEL>0);
28 ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKMINIMUM_check (STOCKMINIMUM>0);
29 ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM_check (NOM<>'');
30
31
32 /**
33  *****
34  * Unique Constraints
35  *****
36
37 ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM UNIQUE (NOM);
38
39
40 /**
41  *****
42  * Primary Keys
43  *****
44 ALTER TABLE ARTICLES ADD CONSTRAINT PK_ARTICLES PRIMARY KEY (ID);

```

- en [7] : le script chargé
- en [8] : on l'exécute
- en [9] : la table [ARTICLES] a été créée

## 5.5.5 Pilote JDBC de MySQL5

Le pilote JDBC de MySQL est téléchargeable au même endroit que le SGBD :

The screenshot shows two parts. On the left, a web browser displays the MySQL Connector/J download page on SourceForge.net, with a yellow box labeled '1' highlighting the 'MySQL Connector/J 5.1 -- Alpha release' link. On the right, a file manager window shows a folder structure for 'mysql-connector-java-5.0.5.zip', with a yellow box labeled '2' pointing to the 'Source and Binaries (zip)' download link and a yellow box labeled '3' pointing to the 'mysql-connector-java-5.0.5-bin.jar' file in the contents list.

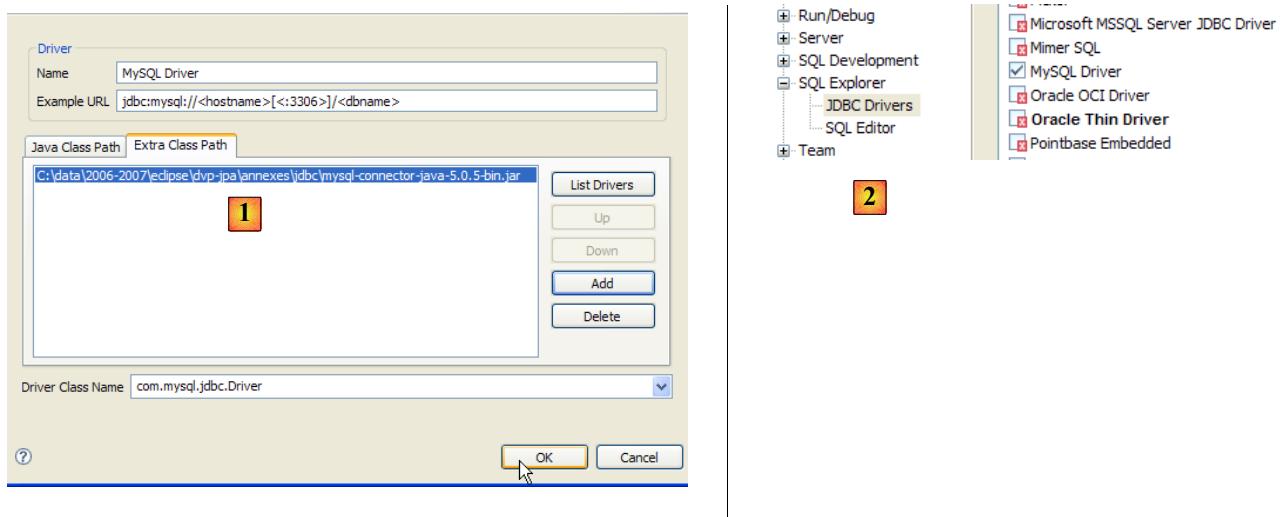
- en [1] : choisir le pilote JDBC qui convient
- en [2] : prendre la version windows qui convient

- en [3] : dans le zip récupéré, l'archive Java contenant le pilote JDBC est [mysql-connector-java-5.0.5-bin.jar]. On l'extraiera afin de l'utiliser dans les exemples du tutoriel JPA.

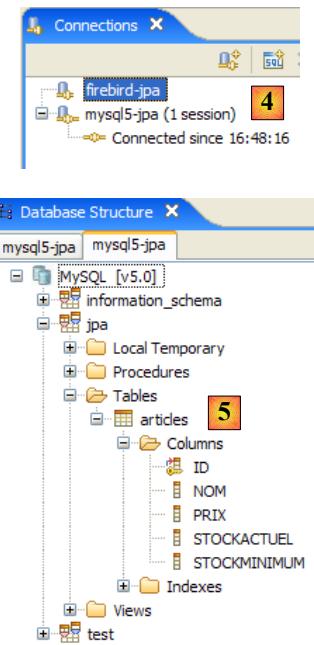
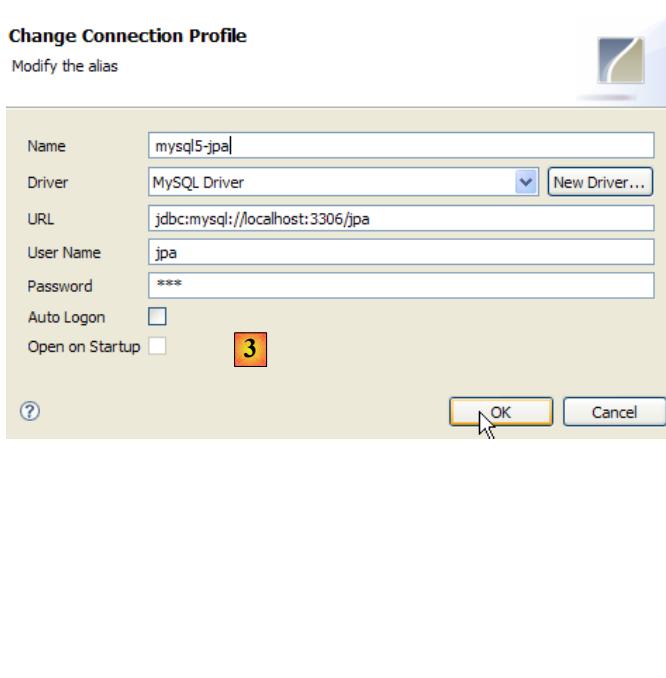
Nous le plaçons comme le précédent (paragraphe 5.4.7, page 255) dans le dossier <jdbc> :



Pour tester ce pilote JDBC, nous allons utiliser Eclipse et le plugin SQL Explorer. Le lecteur est invité à suivre la démarche expliquée au paragraphe 5.4.7, page 255. Nous présentons quelques copies d'écran significatives :



- en [1] : on a désigné l'archive du pilote JDBC de MySQL5
- en [2] : le pilote JDBC de MySQL5 est disponible



- en [3] : définition de la connexion (user, password)=(jpa, jpa)
- en [4] : la connexion est active
- en [5] : la base connectée

## 5.6 Le SGBD PostgreSQL

### 5.6.1 Installation

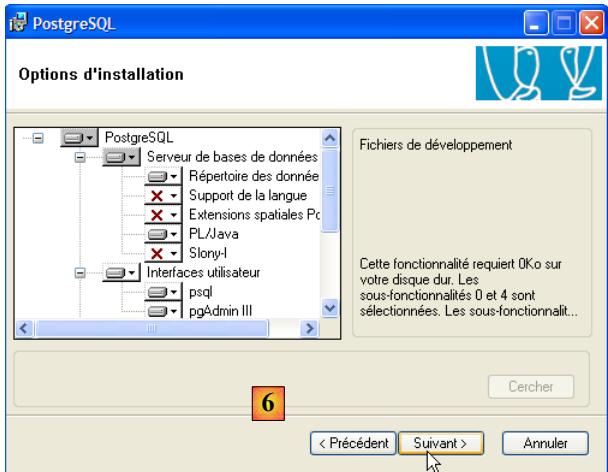
Le SGBD PostgreSQL est disponible à l'url [<http://www.postgresql.org/download/>] :

The screenshot shows the PostgreSQL download page at <http://www.postgresql.org/download/>. The 'Downloads' section is highlighted. A file list on the right shows several zip files, with the 'win32' folder expanded, showing 'v8.2.4' and '2' (highlighted with a red box). A file named 'postgresql-8.2.4-1-win.zip' is selected, highlighted with a red box. A mouse cursor is hovering over it.

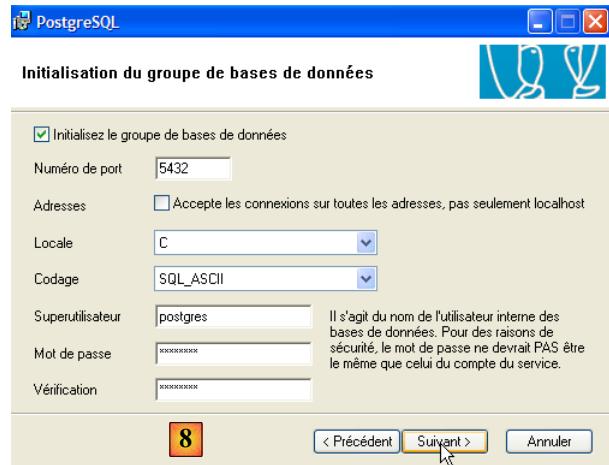
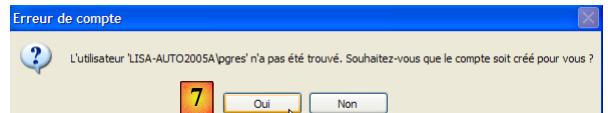
- en [1] : les site de téléchargement de PostgreSQL
- en [2] : choisir une version Windows
- en [3] : choisir une version avec installateur

The screenshot shows the PostgreSQL Installation Wizard. It asks to select the language for installation. The 'French / Français' option is selected (highlighted with a red box). A checkbox for 'Write detailed installation log to postgresql-8.2.log in the current directory' is present. At the bottom are 'Start >' and 'Cancel' buttons. A mouse cursor is hovering over the 'Start >' button.

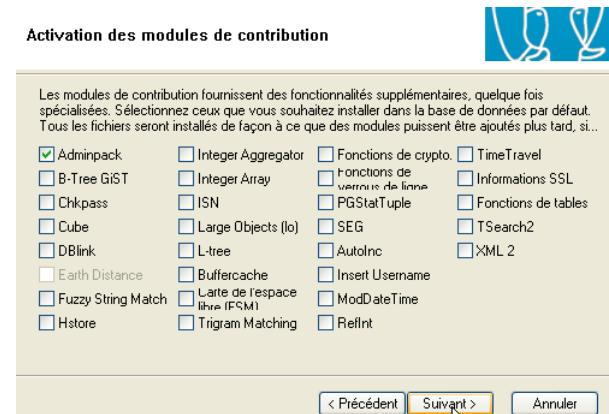
- en [4] : le contenu du fichier zip téléchargé. Double-cliquer sur le fichier [postgresql-8.2.msi]
- en [5] : la première page de l'assistant d'installation



- en [6] : choisir une installation typique en acceptant les valeurs par défaut
- en [6b] : création du compte windows qui lancera le service PostgreSQL, ici le compte **pgres** avec le mot de passe **pgres**.

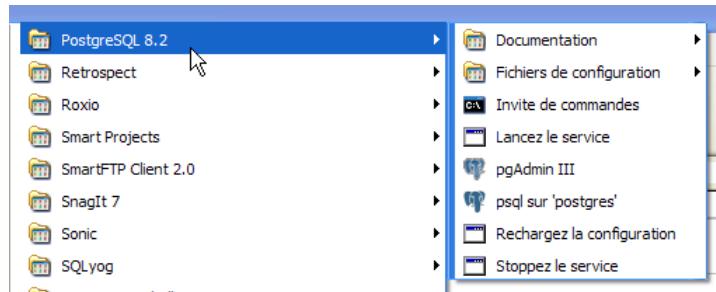


- en [7] : laisser PostgreSQL créer le compte [pgres] si celui-ci n'existe pas déjà
- en [8] : définir le compte administrateur du SGBD, ici **postgres** avec le mot de passe **postgres**



- en [9] et [10] : accepter les valeurs par défaut jusqu'à la fin de l'assistant. PostgreSQL va être installé.

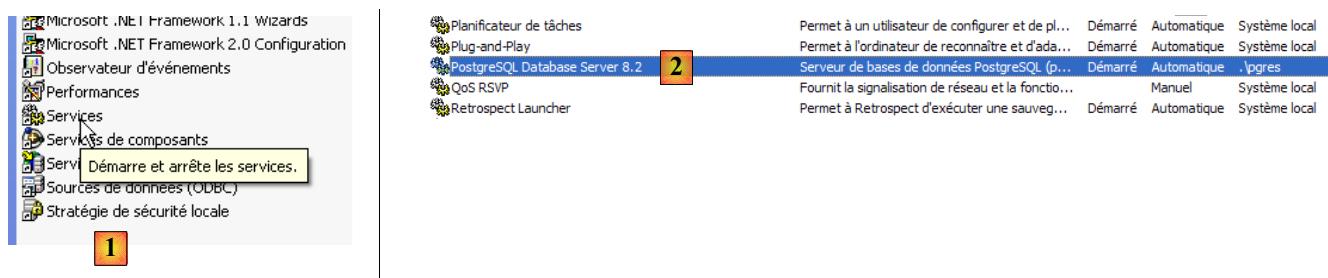
L'installation de PostgreSQL donne naissance à un dossier dans [Démarrer / Programmes] :



## 5.6.2 Lancer / Arrêter PostgreSQL

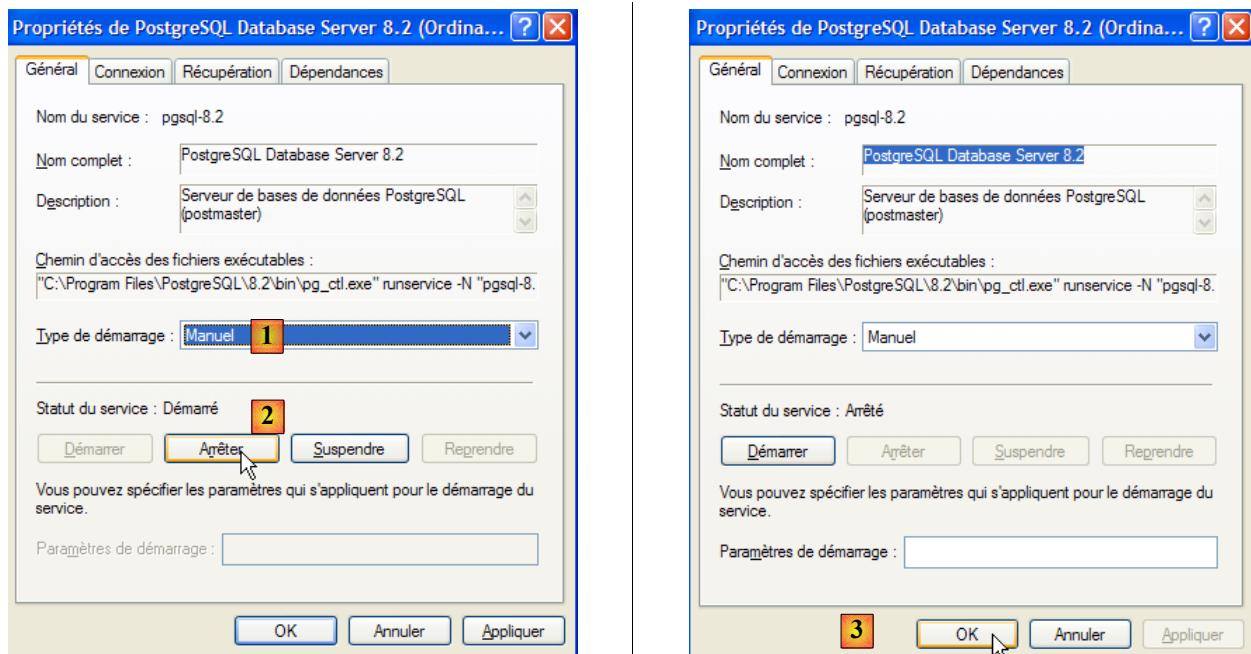
Le serveur PostgreSQL a été installé comme un service windows à démarrage automatique, c.a.d lancé dès le démarrage de windows. Ce mode de fonctionnement est peu pratique. Nous allons le changer :

[Démarrer / Panneau de configuration / Performances et maintenance / Outils d'administration / Services] :



- en [1] : nous double-cliquons sur [Services]
- en [2] : on voit qu'un service appelé [PostgreSQL] est présent, qu'il est démarré [3] et que son démarrage est automatique [4].

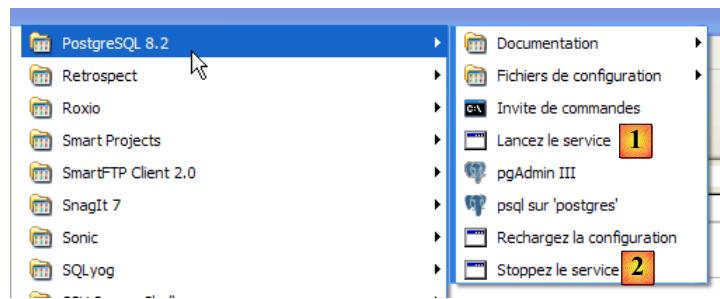
Pour modifier ce fonctionnement, nous double-cliquons sur le service [PostgreSQL] :



- en [1] : on met le service en démarrage manuel
- en [2] : on l'arrête

- en [3] : on valide la nouvelle configuration du service

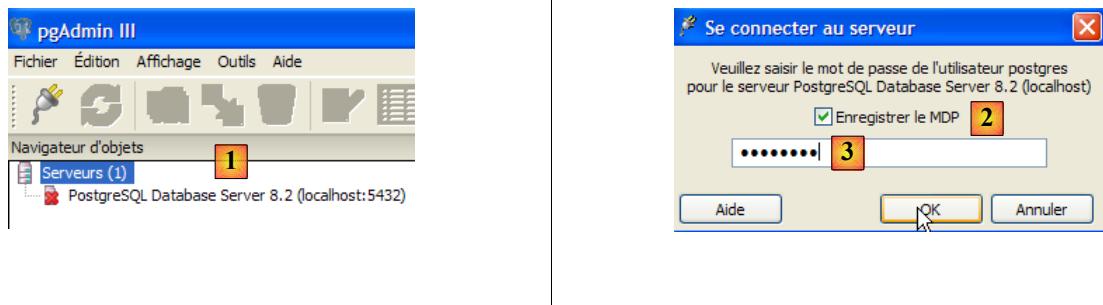
Pour lancer et arrêter manuellement le service PostgreSQL, on pourra utiliser les raccourcis du dossier [PostgreSQL] :



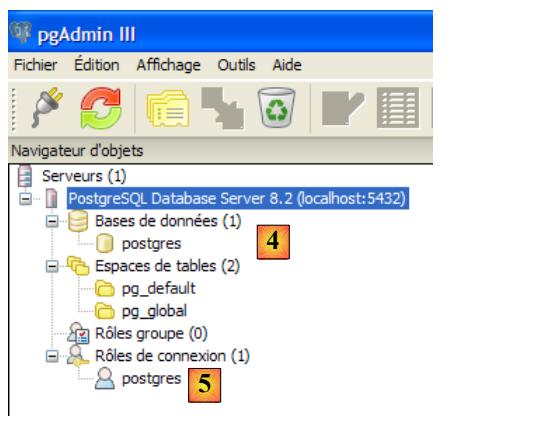
- en [1] : le raccourci pour lancer PostgreSQL
- en [2] : le raccourci pour l'arrêter

### 5.6.3 Administrer PostgreSQL

Sur la copie d'écran ci-dessus, l'application [pgAdmin III] (3) permet d'administrer le SGBD PostgreSQL. Lançons le SGBD, puis [pgAdmin III] via le menu ci-dessus :



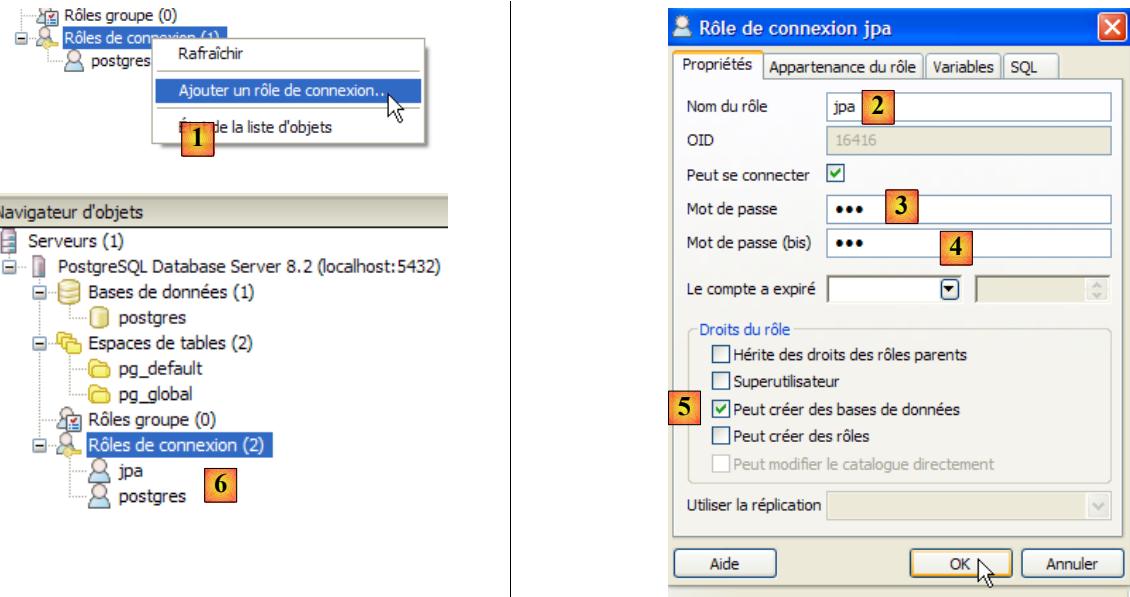
- en [1] : double-cliquer sur le serveur PostgreSQL pour s'y connecter
- en [2,3] : s'identifier comme administrateur du SGBD, ici (postgres / postgres)



- en [4] : l'unique base existante
- en [5] : l'unique utilisateur existant

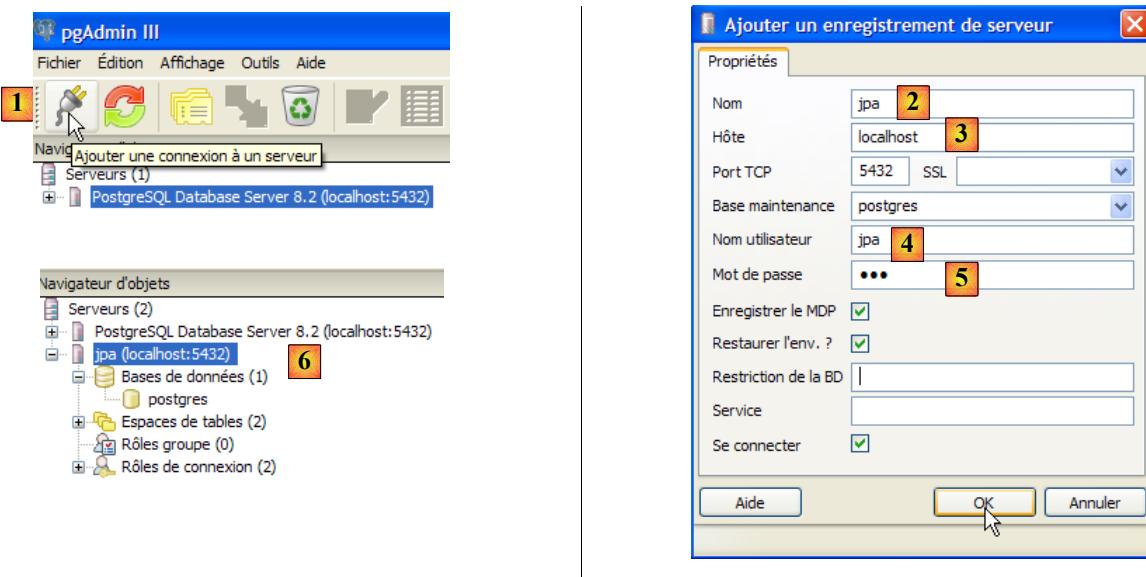
### 5.6.4 Crédation d'un utilisateur jpa et d'une base de données jpa

Le tutoriel utilise PostgreSQL avec une base de données appelée **jpa** et un utilisateur de même nom. Nous les créons maintenant. D'abord l'utilisateur :



- en [1] : on crée un nouveau rôle (~utilisateur)
- en [2] : création de l'utilisateur **jpa**
- en [3] : son mot de passe est **jpa**
- en [4] : on répète le mot de passe
- en [5] : on autorise l'utilisateur à créer des bases de données
- en [6] : l'utilisateur [jpa] apparaît parmi les rôles de connexion

La base de données maintenant :



- en [1] : on crée une nouvelle connexion au serveur
- en [2] : elle s'appellera **jpa**
- en [3] : machine à laquelle on veut se connecter
- en [4] : l'utilisateur qui se connecte
- en [5] : son mot de passe. On valide la configuration de la connexion par [OK]
- en [6] : la nouvelle connexion a été créée. Elle appartient à l'utilisateur **jpa**. Celui-ci va maintenant créer une nouvelle base de données :

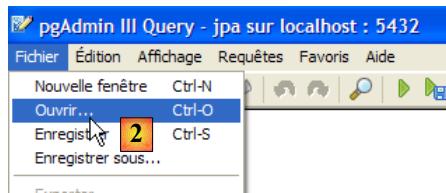
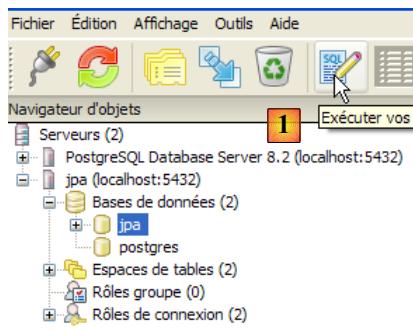
The screenshot shows two windows from pgAdmin III. The left window is the 'Navigateur d'objets' (Object Navigator) showing a tree structure of servers, databases, schemas, and roles. A new database named 'jpa' has been created under the 'Bases de données' node of the 'jpa' server. The right window is a dialog box titled 'Ajouter une base de données...' (Add a database...). It contains fields for 'Nom' (Name) set to 'jpa' (highlighted with a red box), 'OID' (Object ID) set to '2', 'Propriétaire' (Owner) set to 'jpa' (highlighted with a red box), 'Codage' (Encoding) set to 'LATIN1', and 'Modèle' (Template) set to 'template0'. The 'Tablespace' and 'Restriction schéma' fields are empty. The 'Commentaires' (Comments) field is also empty. At the bottom are 'Aide' (Help), 'OK' (highlighted with a red box), and 'Annuler' (Cancel) buttons.

- en [1] : on ajoute une nouvelle base
- en [2] : son nom est **jpa**
- en [3] : son propriétaire est l'utilisateur **jpa** créé précédemment. On valide par [OK]
- en [4] : la base **jpa** a été créée. Un simple clic dessus nous connecte à elle et nous fait découvrir sa structure :

The screenshot shows the 'Navigateur d'objets' window again. The 'jpa' database is selected, revealing its internal structure. Under the 'Tables' node (highlighted with a red box and labeled '5'), a context menu is open, with the option 'Ajouter une table...' (Add a table...) highlighted with a red box. Other options in the menu include 'Rafraîchir' (Refresh), 'Rapports' (Reports), and 'Assistant de gestion des droits' (Rights Management Assistant).

- en [5] : les objets du schéma [jpa] apparaissent, notamment les tables. Il n'y en a pas encore. Un clic droit permettrait d'en créer. Nous laissons le lecteur le faire.

Nous allons maintenant créer la même table [ARTICLES] qu'avec les SGBD précédents en utilisant le script SQL [schema-articles.sql] généré au paragraphe 5.4.6, page 253.



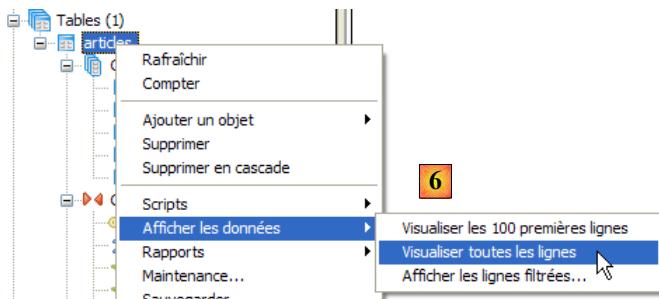
- en [1] : ouvrir l'éditeur SQL
- en [2] : ouvrir un script SQL
- en [3]: désigner le script [schema-articles.sql] créé au paragraphe 5.4.6, page 253.

```

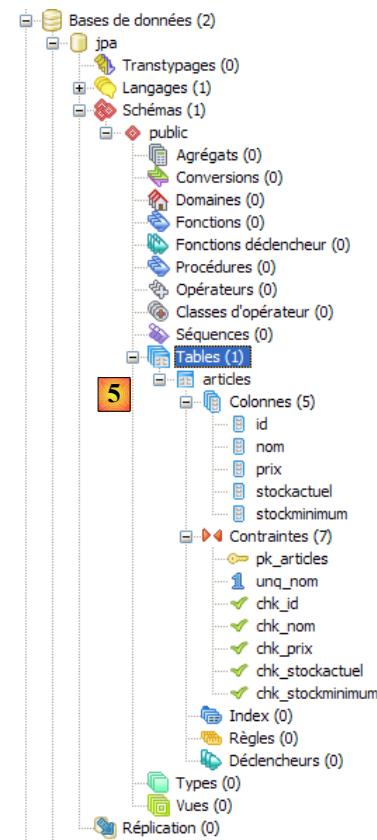
/*
***** Tables *****
*/
CREATE TABLE ARTICLES (
    ID      INTEGER NOT NULL,
    NOM     VARCHAR(20) NOT NULL,
    PRIX    DOUBLE PRECISION NOT NULL,
    STOCKACTUEL  INTEGER NOT NULL,
    STOCKMINIMUM  INTEGER NOT NULL
);

INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM)
VALUES (1, 'article1', 100, 100, 1);
INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM)
VALUES (2, 'article2', 200, 200, 2);
INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM)
VALUES (3, 'article3', 300, 300, 3);

```



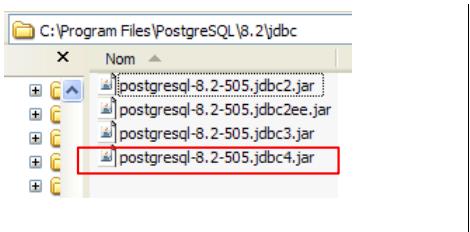
	<b>id</b> [PK] integer	<b>nom</b> character varying	<b>prix</b> double precision	<b>stockactuel</b> integer	<b>stockminimum</b> integer
1	1	article1	100	100	1
2	2	article2	200	200	2
3	3	article3	300	300	3
*					



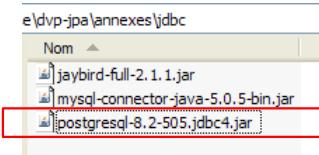
- en [4] : le script chargé. On l'exécute.
- en [5] : la table [ARTICLES] a été créée.
- en [6, 7] : son contenu

## 5.6.5 Pilote JDBC de PostgreSQL

Le pilote JDBC de MySQL est disponible dans le dossier [jdbc] du dossier d'installation de PostgreSQL :



Nous le plaçons l'archive Jdbc comme les précédentes (paragraphe 5.4.7, page 255) dans le dossier <jdbc> :



Pour tester ce pilote JDBC, nous allons utiliser Eclipse et le plugin SQL Explorer. Le lecteur est invité à suivre la démarche expliquée au paragraphe 5.4.7, page 255. Nous présentons quelques copies d'écran significatives :

The left screenshot shows the 'Driver' configuration dialog in Eclipse. It includes fields for 'Name' (PostgreSQL), 'Example URL' (jdbc:postgresql:[</host>[:<5432>]] <database>), and 'Driver Class Name' (org.postgresql.Driver). The 'Extra Class Path' tab is selected, and the path C:\data\2006-2007\ eclipse\dvp-jpa\annexes\jdbc\postgresql-8.2-505.jdbc4.jar is listed. A yellow box labeled '1' highlights this path.  
The right screenshot shows the Eclipse interface with the 'SQL Explorer' perspective selected. The 'JDBC Drivers' node is expanded, showing a list of drivers: Oracle OCI Driver, Oracle Thin Driver, Pointbase Embedded, Pointbase Server, PostgreSQL (which is checked), and Progress OpenEdge 10. A yellow box labeled '2' highlights the 'PostgreSQL' entry.

- en [1] : on a désigné l'archive du pilote JDBC de PostgreSQL
- en [2] : le pilote JDBC de PostgreSQL est disponible

The screenshot shows the SQL Developer interface. At the top left is the 'Create New Connection Profile' dialog with fields for Name (postgresql-jpa), Driver (PostgreSQL), URL (jdbc:postgresql://localhost:5432/jpa), User Name (jpa), Password (\*\*\*), and Auto Logon checked (marked with a red box). At the bottom right of the dialog is the 'OK' button. To the right of the dialog is the 'Connections' window showing three connections: firebird-jpa, mysql5-jpa, and postgresql-jpa (1 session) (marked with a red box). A small red box labeled '4' is in the top right corner of the Connections window. Below the connections is the 'Database Structure' window for the postgresql-jpa database, showing the PostgreSQL schema with tables like information\_schema, pg\_catalog, and public, and a detailed view of the articles table with columns id, nom, prix, stockactuel, and stockminimum (marked with a red box). A small red box labeled '5' is in the top right corner of the Database Structure window. At the bottom left is the 'SQL Results' window showing a table with data (marked with a red box). A small red box labeled '6' is in the top right corner of the SQL Results window.

- en [3] : définition de la connexion (user, password)=(jpa, jpa)
- en [4] : la connexion est active
- en [5] : la base connectée
- en [6] : le contenu de la table [ARTICLES]

## 5.7 Le SGBD Oracle 10g Express

### 5.7.1 Installation

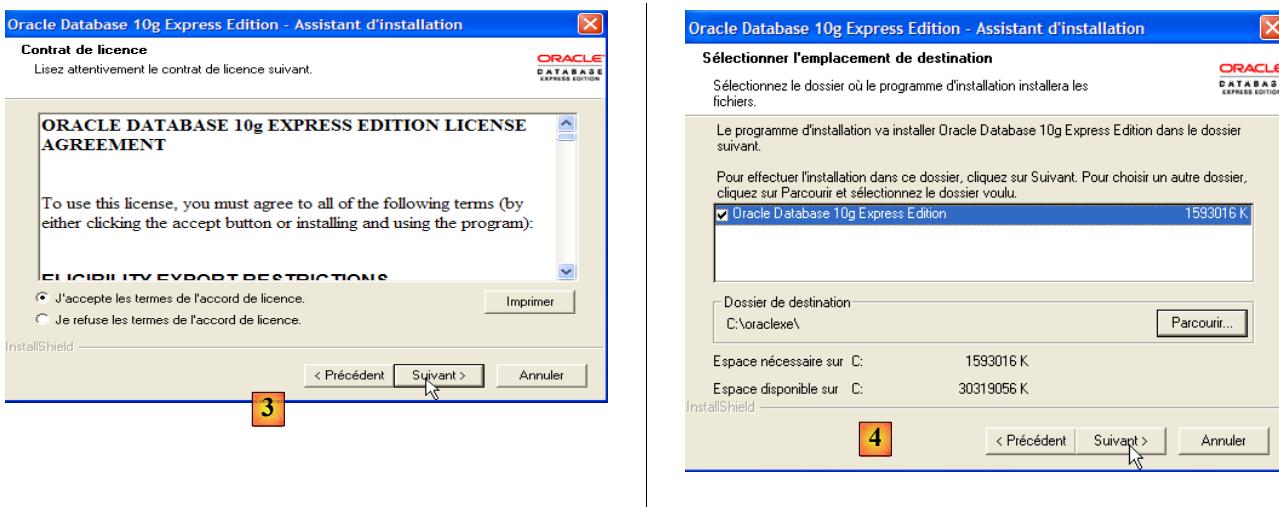
Le SGBD Oracle 10g Express est disponible à l'url  
[\[http://www.oracle.com/technology/software/products/database/xe/index.html\]](http://www.oracle.com/technology/software/products/database/xe/index.html) :

The screenshot shows the 'Oracle Database 10g Express Edition Downloads' page. At the top left is the URL http://www.oracle.com/technology/software/products/database/xe/index.html. Below the URL is a navigation bar with links for Getting Started, Downloads, Documentation, and Feedback (marked with a red box). In the center is a large download area for Oracle Database 10g Express Edition (Western European). It includes a link to OracleXE.exe (165,332,312 bytes), a note about it being a Single-byte LATIN1 database for Western European language storage, and a note about the Database Homepage user interface in English only. A small red box labeled '1' is at the top center of the page. A larger red box labeled '2' is at the bottom right of the download area.

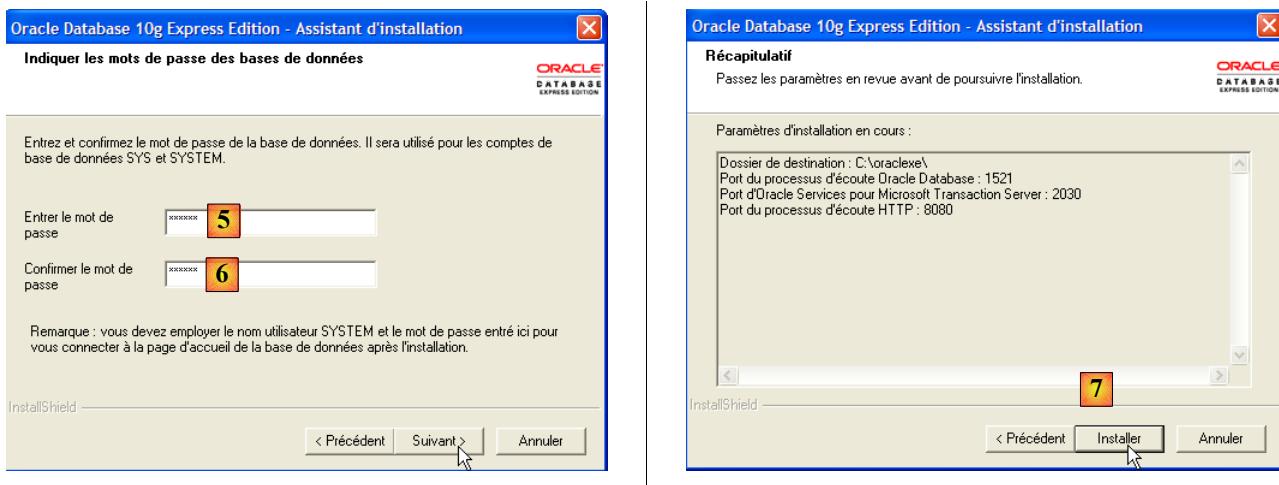
- en [1] : le site de téléchargement d'Oracle 10g Express
- en [2] : choisir une version Windows. Une fois téléchargé le fichier, l'exécuter :



- en [1] : double-cliquer sur le fichier [OracleXE.exe]
- en [2] : la première page de l'assistant d'installation

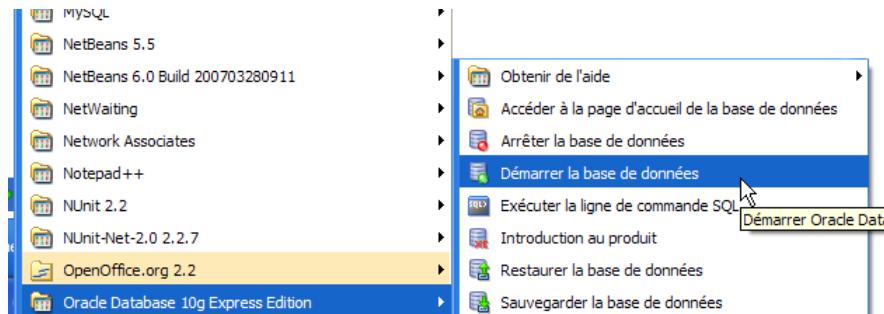


- en [3] : accepter la licence
- en [4] : accepter les valeurs par défaut.



- en [5,6] : l'utilisateur SYSTEM aura le mot de passe **system**.
- en [7] : on lance l'installation

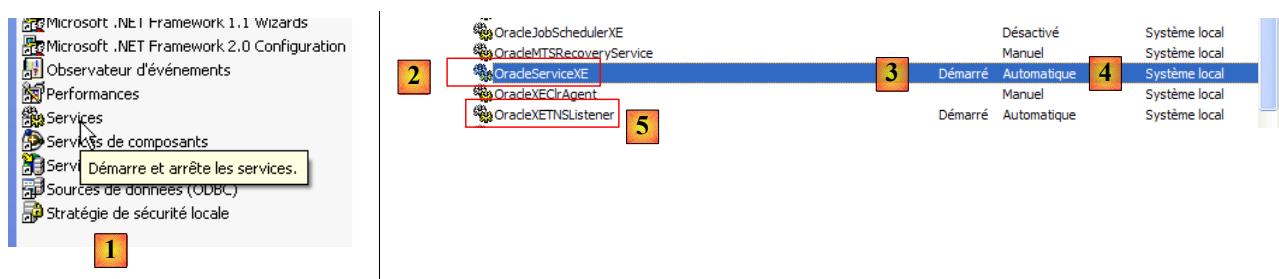
L'installation de Oracle 10g Express donne naissance à un dossier dans [Démarrer / Programmes] :



## 5.7.2 Lancer / Arrêter Oracle 10g

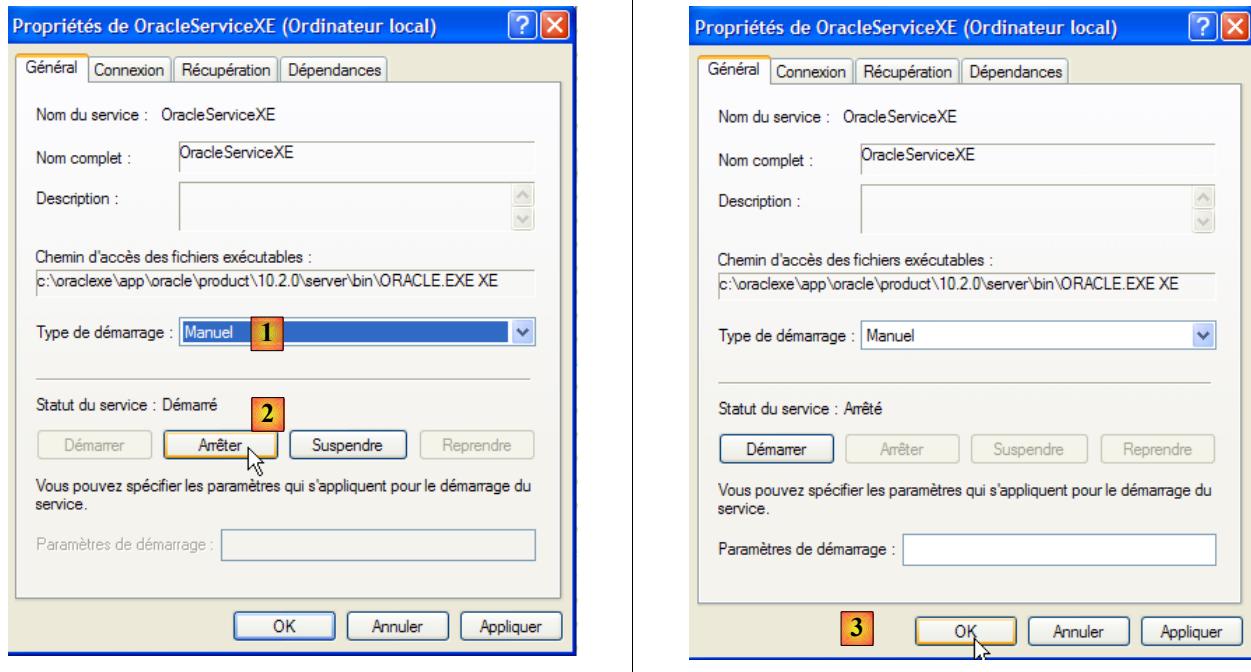
Comme pour les SGBD précédents, Oracle 10g a été installé comme un service windows à démarrage automatique. Nous changeons cette configuration :

[Démarrer / Panneau de configuration / Performances et maintenance / Outils d'administration / Services] :



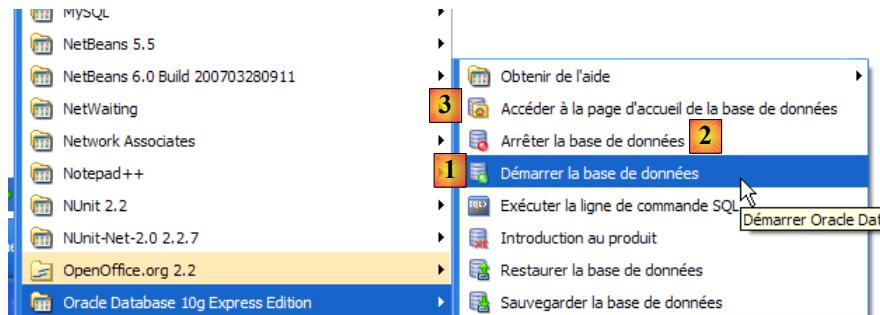
- en [1] : nous double-cliquons sur [Services]
- en [2] : on voit qu'un service appelé [OracleServiceXE] est présent, qu'il est démarré [3] et que son démarrage est automatique [4].
- en [5] : un autre service d'Oracle, appelé " Listener " est également actif et à démarrage automatique.

Pour modifier ce fonctionnement, nous double-cliquons sur le service [OracleServiceXE] :



- en [1] : on met le service en démarrage manuel
- en [2] : on l'arrête
- en [3] : on valide la nouvelle configuration du service

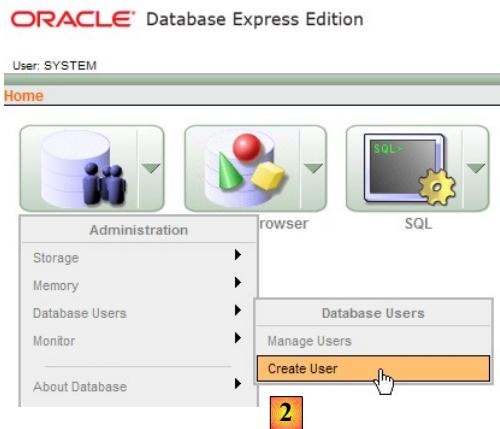
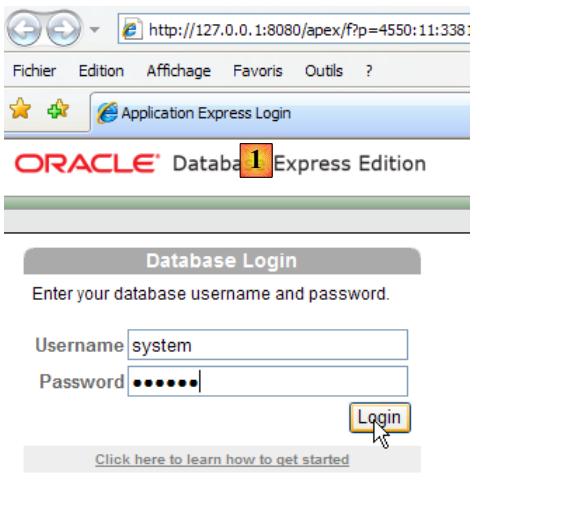
On procèdera de même avec le service [OracleXETNSListener] (cf [5] plus haut). Pour lancer et arrêter manuellement le service OracleServiceXE, on pourra utiliser les raccourcis du dossier [Oracle] :



- en [1] : pour démarrer le SGBD
- en [2] : pour l'arrêter
- en [3] : pour l'administrer (ce qui le lance s'il ne l'est pas déjà)

### 5.7.3 Crédation d'un utilisateur jpa et d'une base de données jpa

Sur la copie d'écran ci-dessus, l'application [3] permet d'administrer le SGBD Oracle 10g Express. Lançons le SGBD [1], puis l'application d'administration [3] via le menu ci-dessus :



- en [1] : s'identifier comme administrateur du SGBD, ici (system / system)
- en [2] : on crée un nouvel utilisateur

User: SYSTEM  
Home > Administration > Manage Database Users > Create Database User

Create Database User

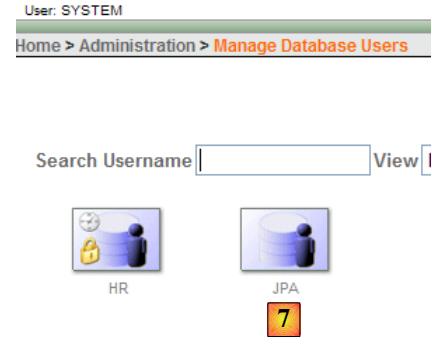
* Username: <b>jpa</b>	[4]
* Password: <b>***</b>	[5]
* Confirm Password: <b>***</b>	[6]
Expire Password:	<input type="checkbox"/>
Account Status:	Unlocked
Default Tablespace:	USERS
Temporary Tablespace:	TEMP

**Create** [7]

User Privileges

Roles:

CONNECT  RESOURCE  DBA



- en [4] : nom de l'utilisateur
- en [5, 6] : son mot de passe, ici **jpa**
- en [7] : l'utilisateur **jpa** a été créé

Sous Oracle, un utilisateur est automatiquement associé à une base de données de même nom. La base de données **jpa** existe donc en même temps que l'utilisateur **jpa**.

## 5.7.4 Crédation de la table [ARTICLES] de la base de données jpa

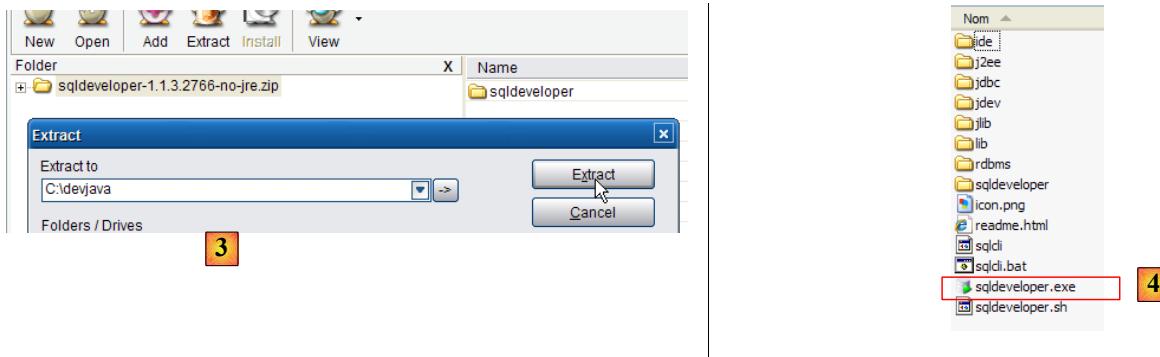
OracleXE a été installé avec un client SQL travaillant en mode ligne. On peut travailler plus confortablement avec SQL Developer également fourni par Oracle. On le trouve sur le site :  
[\[http://www.oracle.com/technology/products/database/sql\\_developer/index.html\]](http://www.oracle.com/technology/products/database/sql_developer/index.html)



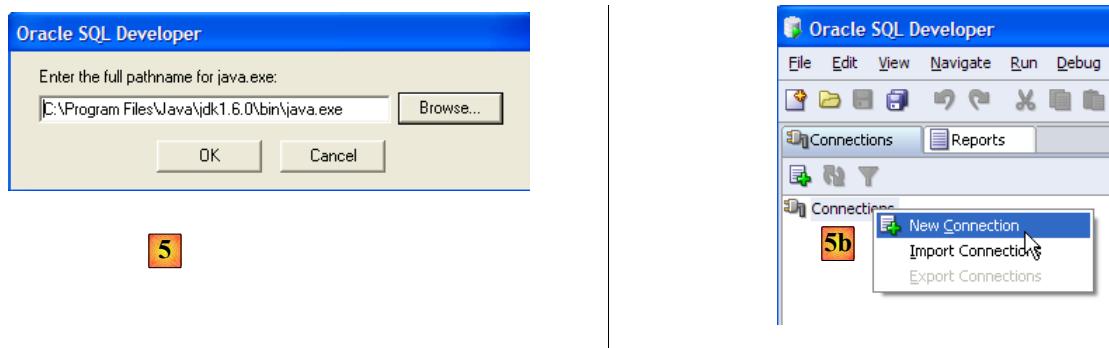
Oracle SQL Developer for Windows, with JDK 1.5 already installed (43 M)  
For developers with JDK 1.5 already installed  
To install and run:  
 - Download the file above [2]  
 - Extract sqldeveloper.zip into any folder, using folder names  
 - Within that folder, open the sqldeveloper folder  
 - Double-click sqldeveloper.exe

- en [1] : le site de téléchargement

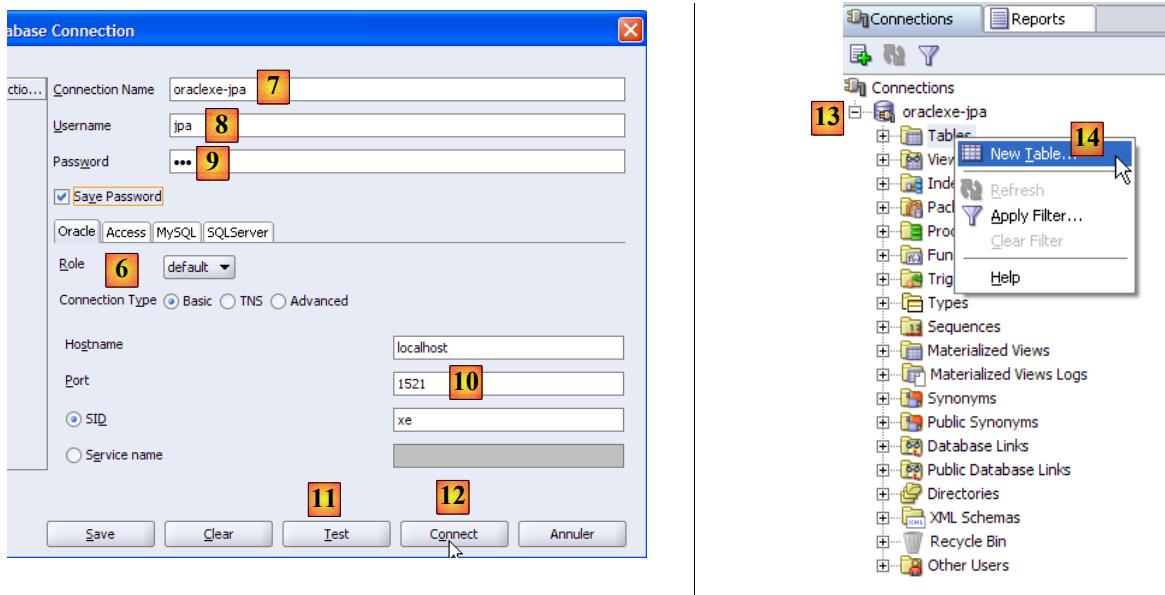
- en [2] : prendre une version windows sans Jre si celui-ci est déjà installé (le cas ici), [SQL Developer] étant une application Java.



- en [3] : décompresser le zip téléchargé
- en [4] : lancer l'exécutable [sqldeveloper.exe]



- en [5] : au 1er lancement de [SQL Developer], indiquer le chemin du Jre installé sur la machine
- en [5b] : créer une nouvelle connexion



- en [6] : SQL Developer permet de se connecter à divers SGBD. Choisir Oracle.
- en [7] : nom donné à la connexion qu'on est en train de créer
- en [8] : propriétaire de la connexion
- en [9] : son mot de passe (jpa)
- en [10] : garder les valeurs par défaut
- en [11] : pour tester la connexion (Oracle doit être lancé)
- en [12] : pour terminer la configuration de la connexion
- en [13] : les objets de la base jpa

- en [14] : on peut créer des tables. Comme dans les cas précédents, nous allons créer la table [ARTICLES] à partir du script créé au paragraphe 5.4.6, page 253.

```

CREATE TABLE ARTICLES (
    ID          INTEGER NOT NULL,
    NOM         VARCHAR(20) NOT NULL,
    PRIX        DOUBLE PRECISION NOT NULL,
    STOCKACTUEL INTEGER NOT NULL,
    STOCKMINIMUM INTEGER NOT NULL
);

INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM)
VALUES (1, 'article1', 100, 10, 1);
INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM)
VALUES (2, 'article2', 200, 20, 2);
INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM)
VALUES (3, 'article3', 300, 30, 3);

```

- en [15] : on ouvre un script SQL
- en [16] : on désigne le script SQL créé au paragraphe 5.4.6, page 253.
- en [17] : le script qui va être exécuté

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100	10	1
2	article2	200	20	2
3	article3	300	30	3

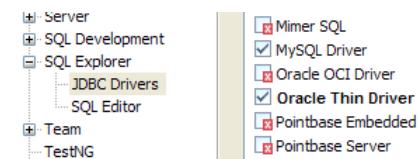
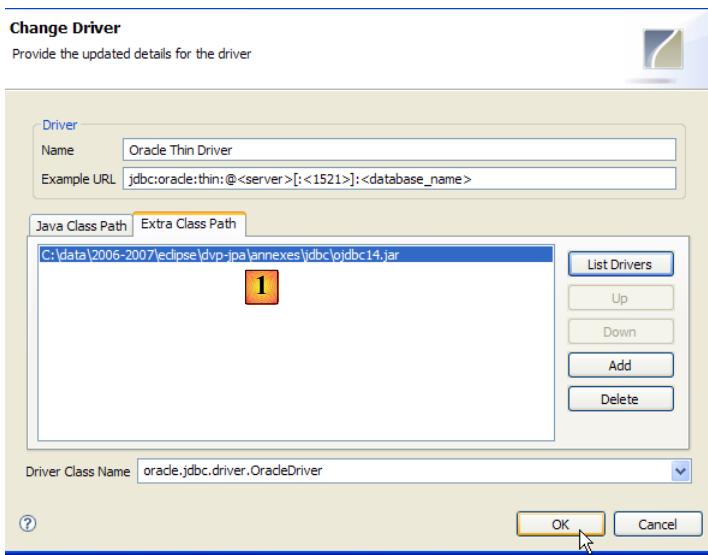
- en [18] : le résultat de l'exécution : la table [ARTICLES] a été créée. On double-clique dessus pour avoir accès à ses propriétés.
- en [19] : le contenu de la table.

## 5.7.5 Pilote JDBC de OracleXE

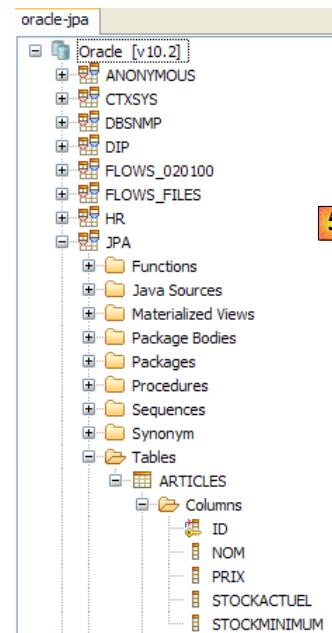
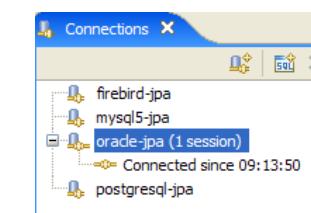
Le pilote JDBC de OracleXE est disponible dans le dossier [jdbc/lib] du dossier d'installation de OracleXE [1] :

Nous le plaçons l'archive Jdbc [ojdbc14.jar] comme les précédentes (paragraphe 5.4.7, page 255) dans le dossier <jdbc> [2] :

Pour tester ce pilote JDBC, nous allons utiliser Eclipse et le plugin SQL Explorer. Le lecteur est invité à suivre la démarche expliquée au paragraphe 5.4.7, page 255. Nous présentons quelques copies d'écran significatives :



- en [1] : on a désigné l'archive du pilote JDBC de OracleXE
- en [2] : le pilote JDBC de OracleXE est disponible



- en [3] : définition de la connexion (user, password)=(jpa, jpa)
- en [4] : la connexion est active
- en [5] : la base connectée
- en [6] : le contenu de la table [ARTICLES]

## 5.8 Le SGBD SQL Server Express 2005

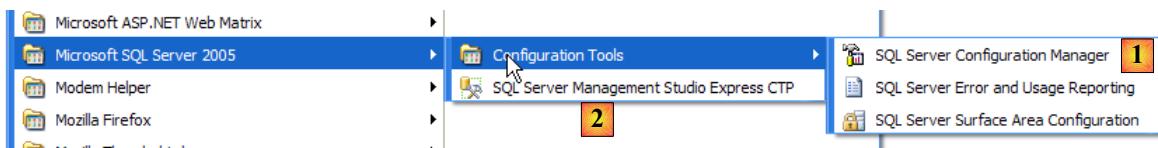
### 5.8.1 Installation

Le SGBD SQL Server Express 2005 est disponible à l'url [<http://msdn.microsoft.com/vstudio/express/sql/download/>] :

The screenshot shows the Microsoft SQL Server Express download page. At the top, there's a search bar with the query "sqlserver express download". Below it, a large orange banner says "Download Now!". Underneath, there's a section titled "Download the Microsoft .NET Framework 2.0" with a note about installing .NET Framework 2.0 first. Then, there's a section titled "Uninstall beta versions" with a note about uninstalling previous versions. Finally, there's a "Download and install" section with a note about customizing the installation. On the left, there's a sidebar with links like "Top of page", "Contact us", and "All rights reserved". The main content area has two columns. The left column contains a link to "SQL Server 2005 Express Edition SP2" (marked with a red box [3]), which includes a note about getting started quickly and a link to "Install Microsoft SQL Server 2005 Express Edition (more...)" (marked with a red box [2]). The right column contains a link to "SQL Server 2005 Express Edition with Advanced Services SP2" (marked with a red box [1]), which includes a note about additional development tools and a link to "Install Microsoft SQL Server 2005 Express Edition with Advanced Services (more...)". Both columns have "Download" links with file sizes: 36.5 MB and 234 MB respectively.

- en [1] : d'abord télécharger et installer la plate-forme .NET 2.0
- en [2] : puis installer et télécharger SQL Server Express 2005
- en [3] : puis installer et télécharger SQL Server Management Studio Express qui permet d'administrer SQL Server

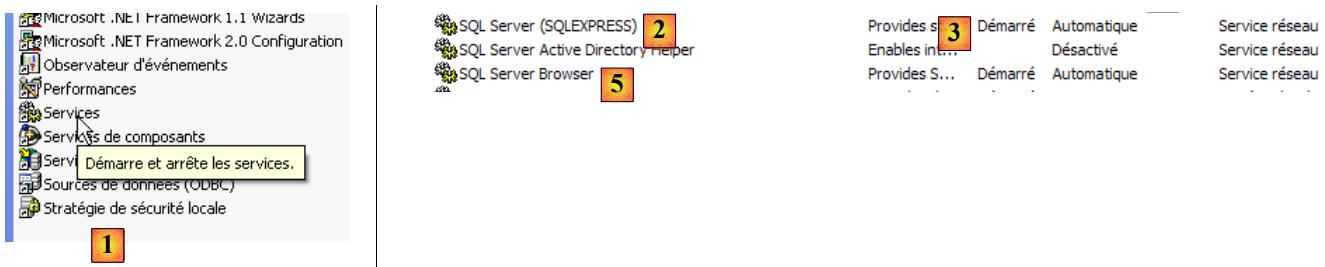
L'installation de SQL Server Express donne naissance à un dossier dans [Démarrer / Programmes] :



- en [1] : l'application de configuration de SQL Server. Permet également de lancer / arrêter le serveur
- en [2] : l'application d'administration du serveur

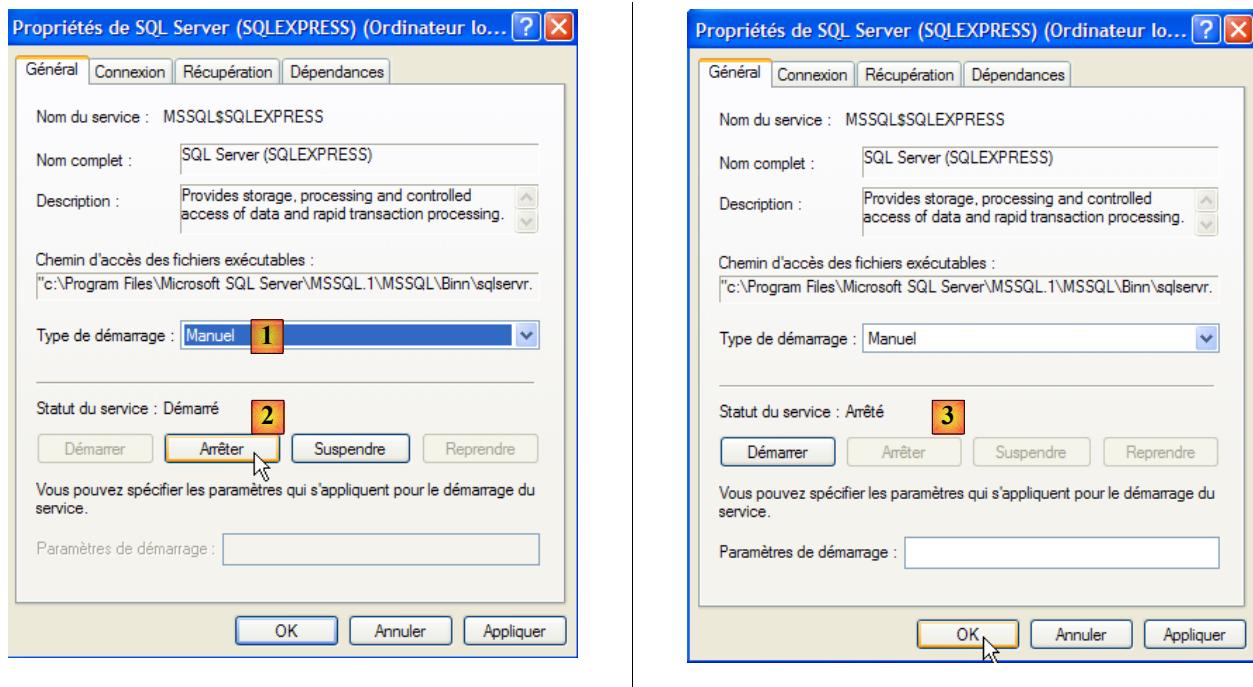
### 5.8.2 Lancer / Arrêter SQL Server

Comme pour les SGBD précédents, SQL server Express a été installé comme un service windows à démarrage automatique. Nous changeons cette configuration :



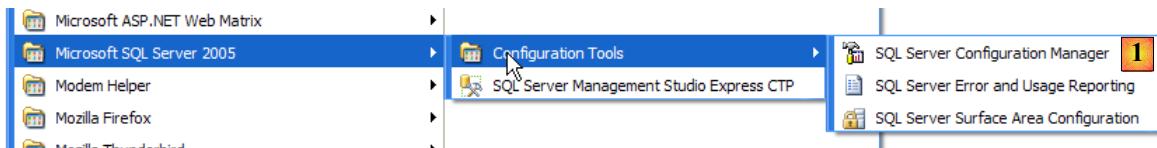
- en [1] : nous double-cliquons sur [Services]
- en [2] : on voit qu'un service appelé [SQL Server] est présent, qu'il est démarré [3] et que son démarrage est automatique [4].
- en [5] : un autre service lié à SQL Server, appelé "SQL Server Browser" est également actif et à démarrage automatique.

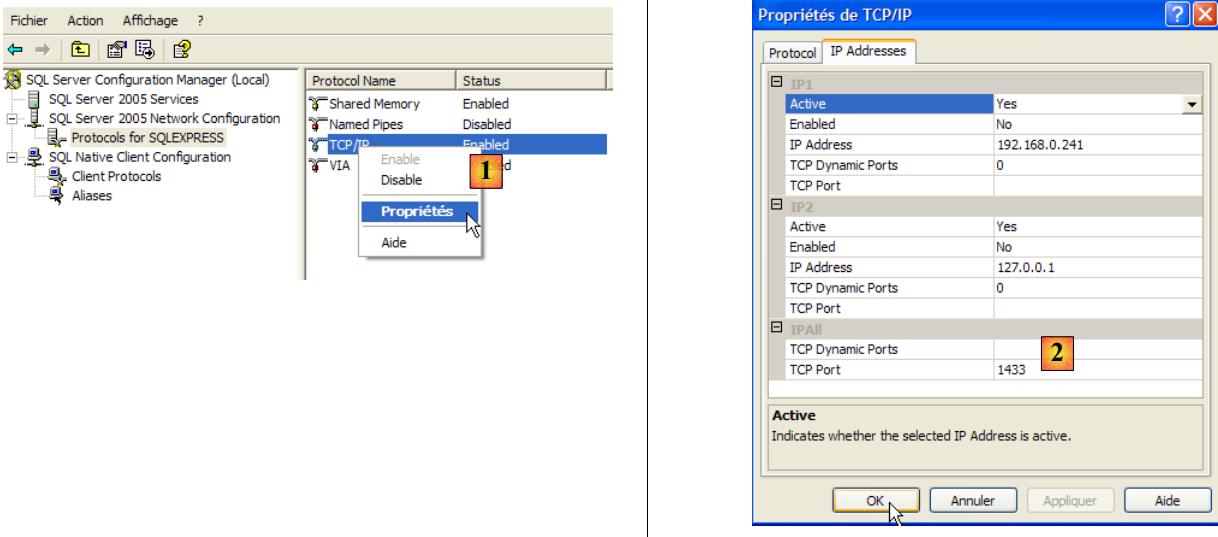
Pour modifier ce fonctionnement, nous double-cliquons sur le service [SQL Server] :



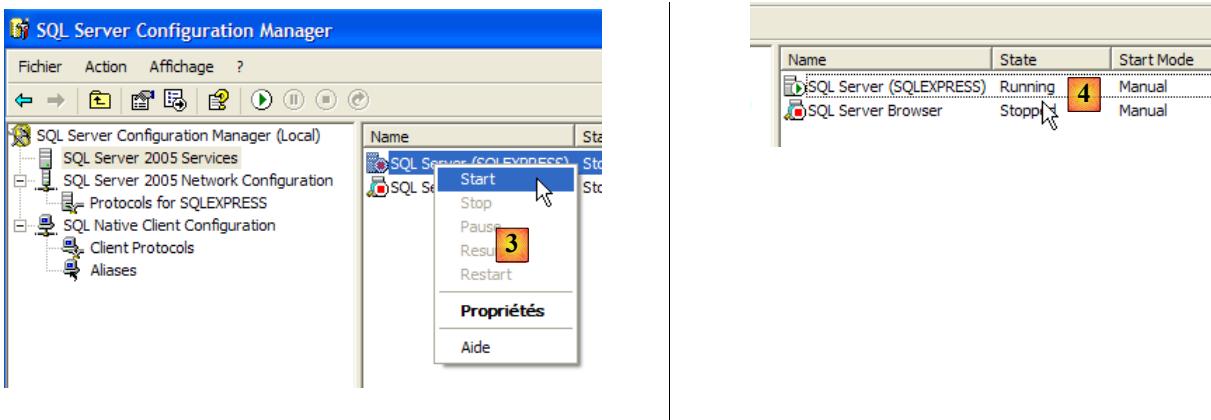
- en [1] : on met le service en démarrage manuel
- en [2] : on l'arrête
- en [3] : on valide la nouvelle configuration du service

On procèdera de même avec le service [SQL Server Browser] (cf [5] plus haut). Pour lancer et arrêter manuellement le service OracleServiceXE, on pourra utiliser l'application [1] du dossier [SQL server] :





- en [1] : s'assurer que le protocole TCP/IP est actif (enabled) puis passer aux propriétés du protocole.
- en [2] : dans l'onglet [IP Addresses], option [IPAll] :
  - le champ [TCP Dynamic ports] est laissé vide
  - le port d'écoute du serveur est fixé à 1433 dans [TCP Port]

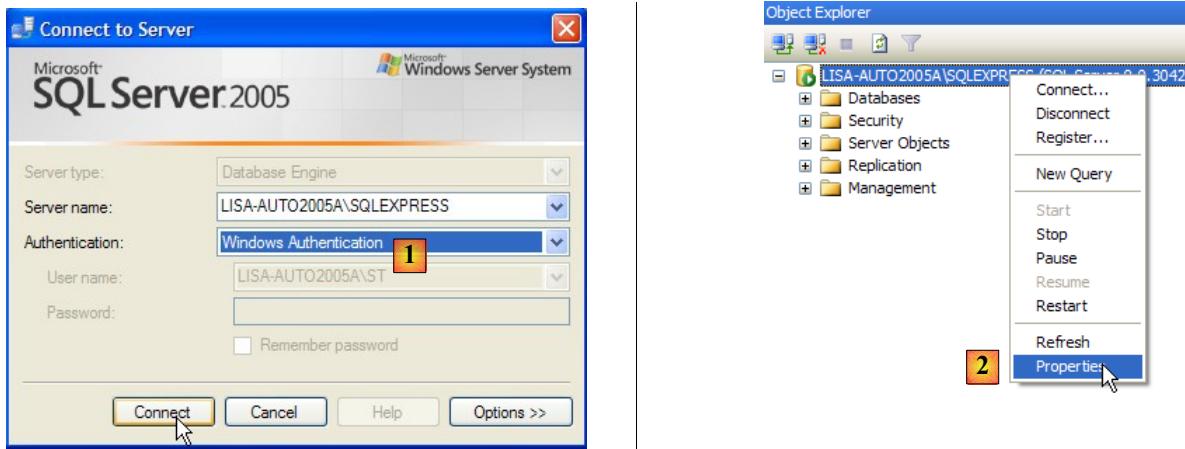


- en [3] : un clic droit sur le service [SQL Server] donne accès aux options de démarrage / arrêt du serveur. Ici, on le lance.
- en [4] : SQL Server est lancé

### 5.8.3 Crédation d'un utilisateur jpa et d'une base de données jpa

Lançons le SGBD comme indiqué ci-dessus, puis l'application d'administration [1] via le menu ci-dessous :



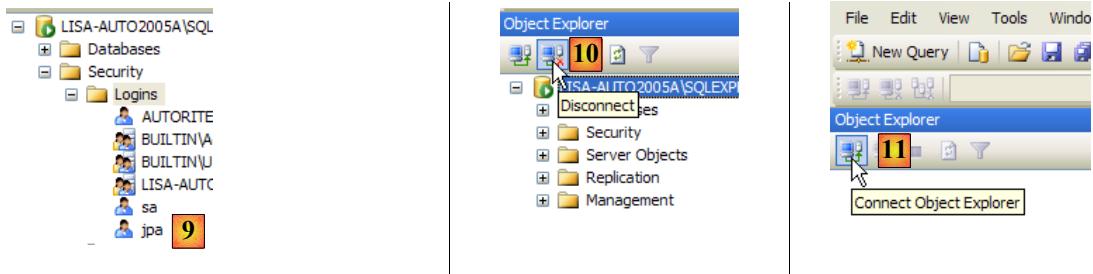


- en [1] : on se connecte à SQL Server en tant qu'**administrateur Windows**
- en [2] : on configure les propriétés de la connexion

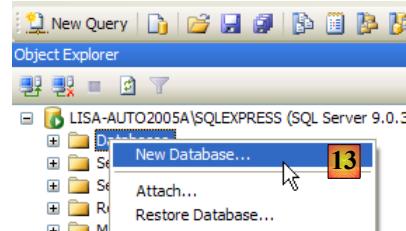
- en [3] : on autorise un mode mixte de connexion au serveur : soit avec un login windows (un utilisateur windows), soit avec un login SQL Server (compte défini au sein de SQL Server, indépendant de tout compte windows).
- en [3b] : on crée un utilisateur SQL Server

- en [4] : option [General]
- en [5] : le login
- en [6] : le mot de passe (**jpa** ici)
- en [7] : option [Server Roles]
- en [8] : l'utilisateur **jpa** aura le droit de créer des bases de données

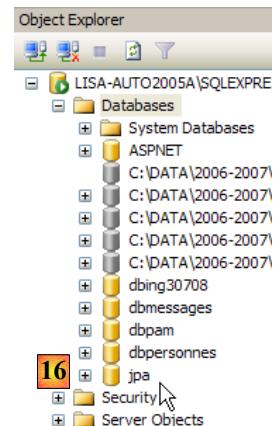
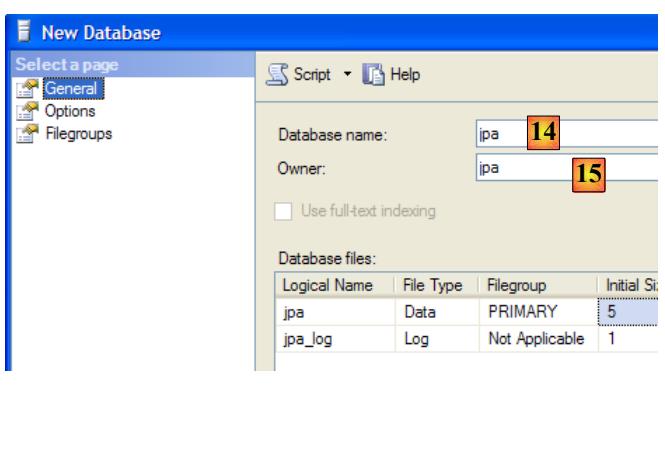
On valide cette configuration :



- en [9] : l'utilisateur **jpa** a été créé
- en [10] : on se déconnecte
- en [11] : on se reconnecte



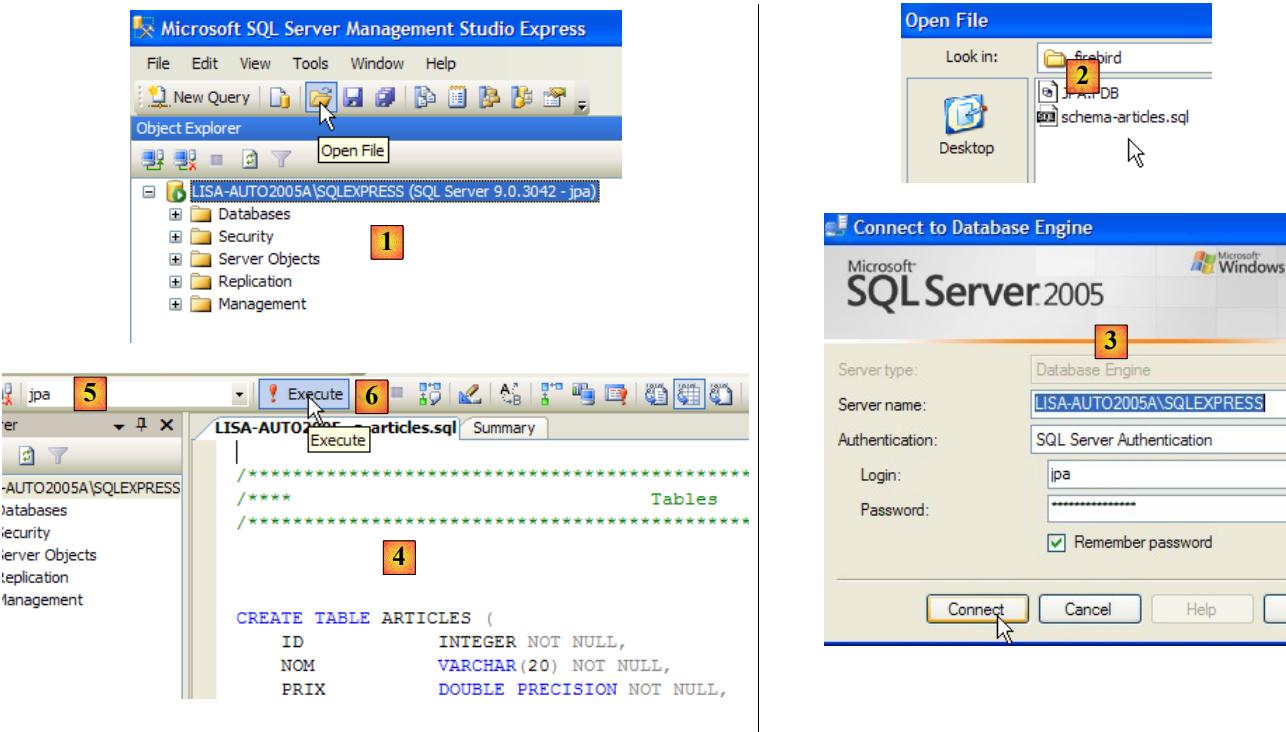
- en [12] : on se connecte en tant qu'utilisateur **jpa/jpa**
- en [13] : une fois connecté, l'utilisateur **jpa** crée une base de données



- en [14] : la base s'appellera **jpa**
- en [15] : et appartiendra à l'utilisateur **jpa**
- en [16] : la base **jpa** a été créée

## 5.8.4 Crédation de la table [ARTICLES] de la base de données jpa

Comme dans les exemples précédents, nous allons créer la table [ARTICLES] à partir du script créé au paragraphe 5.4.6, page 253.



- en [1] : on ouvre un script SQL
- en [2] : on désigne le script SQL créé au paragraphe 5.4.6, page 253.
- en [3] : on doit s'identifier de nouveau (jpa/jpa)
- en [4] : le script qui va être exécuté
- en [5] : sélectionner la base dans laquelle le script va être exécuté
- en [6] : l'exécuter

The screenshot shows the Microsoft SQL Server Management Studio Express interface. On the left, the Object Explorer pane shows a tree structure for 'jpa'. A red box labeled [7] highlights the 'Tables' node under 'dbo'. In the center, a table named 'dbo.ARTICLES' is displayed with the following data:

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100	10	1
2	article2	200	20	2
3	article3	300	30	3
*	NULL	NULL	NULL	NULL

A red box labeled [9] highlights the first row of the table. On the right, a context menu is open over the 'ARTICLES' table in the Object Explorer. The 'Open Table' option is highlighted with a red box [8].

- en [7] : le résultat de l'exécution : la table [ARTICLES] a été créée.
- en [8] : on demande à voir son contenu
- en [9] : le contenu de la table.

## 5.8.5 Pilote JDBC de SQL Server Express

## Microsoft SQL Server 2005 JDBC Driver

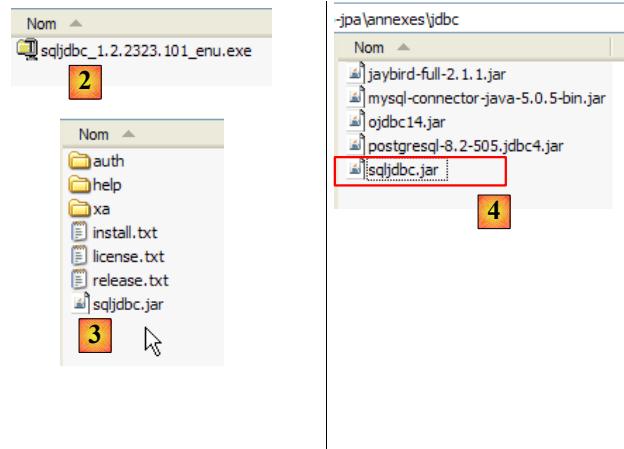
In its continued commitment to interoperability, Microsoft provides . Server 2005 JDBC Driver is available to all SQL Server users at no cost. any Java application, application server, or Java-enabled applet. The standard JDBC application program interfaces (APIs) available in J2EE. The SQL Server 2005 JDBC Driver is JDBC 3.0 compliant and runs on all major application servers, including BEA WebLogic, IBM WebSphere, and Oracle GlassFish.



[Download SQL Server 2005 JDBC Driver 1.2 CTP](#)

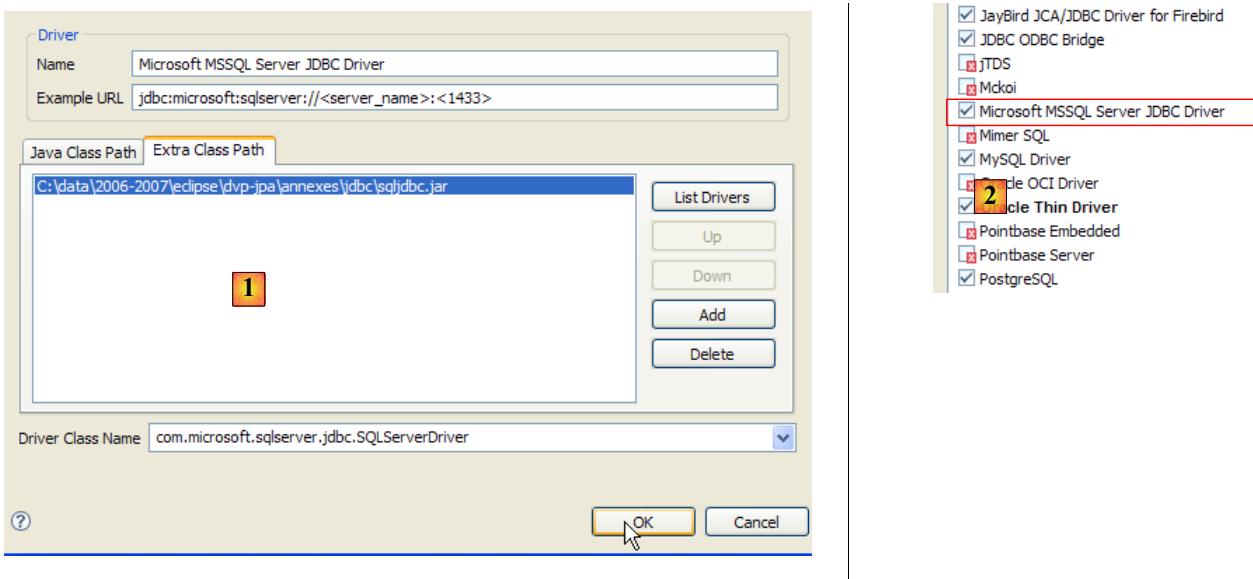
This latest version of the JDBC driver provides an adaptive cursor. In addition, this release includes new st

1



- en [1] : une recherche sur Google avec le texte [Microsoft SQL Server 2005 JDBC Driver] nous amène à la page de téléchargement du pilote JDBC. Nous sélectionnons la version la plus récente
- en [2] : le fichier téléchargé. On double-clique dessus. Une décompression a lieu et donne naissance à un dossier dans lequel on trouve le pilote Jdbc [3]
- en [4] : nous plaçons l'archive Jdbc [sqljdbc.jar] comme les précédentes (paragraphe 5.4.7, page 255) dans le dossier <jdbc>

Pour tester ce pilote JDBC, nous allons utiliser Eclipse et le plugin SQL Explorer. Le lecteur est invité à suivre la démarche expliquée au paragraphe 5.4.7, page 255. Nous présentons quelques copies d'écran significatives :



- en [1] : on a désigné l'archive du pilote JDBC de SQL Server
- en [2] : le pilote JDBC de SQL Server est disponible

The screenshot shows the SSMS interface. On the left, the 'Connections' window [4] lists several connections, with 'sqlserver-jpa' selected and a note 'Connected since 12:24:34'. On the right, the 'Database Structure' window [5] displays the schema of the 'sqlserver-jpa' database, specifically the 'ARTICLES' table with columns ID, NOM, PRIX, STOCKACTUEL, and STOCKMINIMUM.

**Connections**

- Name: sqlserver-jpa
- Driver: Microsoft MSSQL Server JDBC Driver
- URL: jdbc:sqlserver://localhost\SQLEXPRESS:1433;databaseName=jpa
- User Name: jpa [3]
- Password: \*\*\*
- Auto Logon:
- Open on Startup:

**OK Cancel**

**SQL Results Database Detail**

ID	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
1	article1	100	10	1 [6]
2	article2	200	20	2
3	article3	300	30	3

**Database Structure**

```

sqlserver-jpa
  Microsoft SQL Server [v9.0]
    ASPNET
    C:\DATA\2006-2007\VBNET\PAM\DATAE
    C:\DATA\2006-2007\VBNET\PAM\DATAE
    C:\DATA\2006-2007\VBNET\PAM\PAM-II
    C:\DATA\2006-2007\VBNET\PAM\PAM-II
    C:\DATA\2006-2007\VBNET\PAM\PAM-S
    dbng30708
    dbmessages
    dbpam
    dbpersonnes
    jpa
      System Table
      Tables
        ARTICLES
          Columns
            ID
            NOM
            PRIX
            STOCKACTUEL
            STOCKMINIMUM
          Indexes
        Views

```

- en [3] : définition de la connexion (user, password)=(jpa, jpa)
- en [4] : la connexion est active
- en [5] : la base connectée
- en [6] : le contenu de la table [ARTICLES]

## 5.9 Le SGBD HSQLDB

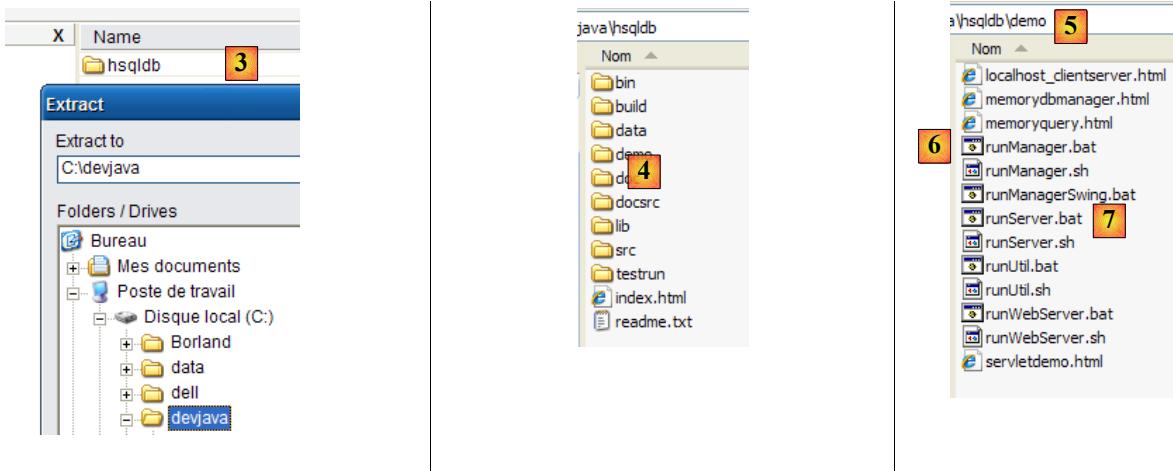
### 5.9.1 Installation

Le SGBD HSQLDB est disponible à l'url [<http://sourceforge.net/projects/hsqldb>]. C'est un SGBD écrit en Java, très léger en mémoire, qui gère des bases de données en mémoire et non sur disque. Le résultat en est une très grande rapidité d'exécution des requêtes. C'est son principal intérêt. Les bases de données ainsi créées en mémoire peuvent être retrouvées lorsque le serveur est arrêté puis relancé. En effet, les ordres SQL émis pour créer les bases sont mémorisés dans un fichier de logs pour être rejoués au démarrage suivant du serveur. On a ainsi une persistance des bases dans le temps.

La méthode a ses limites et HSQLDB n'est pas un SGBD à vocation commerciale. Son principal intérêt réside dans les tests ou les applications de démonstration. Par exemple, le fait que HSQLDB soit écrit en Java permet de l'inclure dans des tâches Ant (Another Neat Tool) un outil Java d'automatisation de tâches. Ainsi des tests journaliers de codes en cours de développement, automatisés par Ant, vont pouvoir intégrer des tests de bases de données gérées par le SGBD HSQLDB. Le serveur sera lancé, arrêté, géré par des tâches Java.

The screenshot shows the SourceForge project page for HSQLDB. At the top, there's a browser header with navigation icons and the URL <http://sourceforge.net/projects/hsqldb>. Below the header, a message states: 'HSQLDB is a relational database engine written in Java, with a JDBC driver, supporting a large subset of ANSI-92 SQL. A small, fast engine with both in memory and disk based tables. This product is the continuation of HypersonicSQL. Active since 2001.' A green button labeled 'Download HSQL Database Engine' [1] is visible. To the right, the 'Latest' section [2] shows the 'hsqldb\_1\_8\_0' release from September 27, 2006, at 12:59. It lists seven zip files: hsqldb\_1\_8\_0\_1.zip, hsqldb\_1\_8\_0\_2.zip, hsqldb\_1\_8\_0\_4.zip, hsqldb\_1\_8\_0\_5.zip, and hsqldb\_1\_8\_0\_7.zip.

- en [1] : le site de téléchargement
- en [2] : prendre la version la plus récente



- en [3] : décompresser le fichier zip téléchargé
- en [4] : le dossier [hsqldb] issu de la décompression
- en [5] : le dossier [demo] qui contient le script permettant de lancer le serveur [hsq] [6] et en [7], celui permettant de lancer un outil rustique d'administration du serveur.

## 5.9.2 Lancer / Arrêter HSQLDB

Pour lancer le serveur HSQLDB on double-clique sur l'application [runManager.bat] [6] ci-dessus :

A screenshot of a Windows command prompt window titled "start HSQL". The window contains the following text:

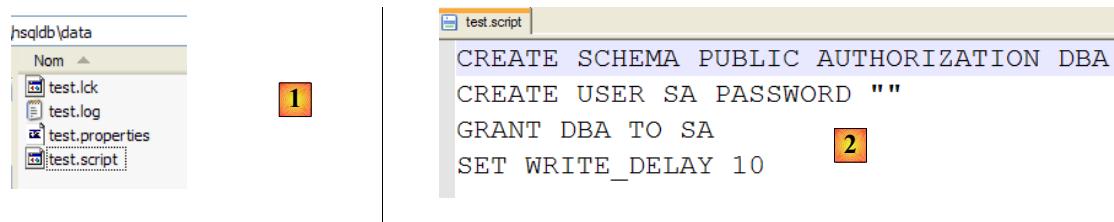
```
C:\devjava\hsqldb\demo>cd ..\data
[Server@1d4c61c]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@1d4c61c]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@1d4c61c]: Startup sequence initiated from main() method
[Server@1d4c61c]: Loaded properties from [C:\devjava\hsqldb\data\server.properties]
[Server@1d4c61c]: Initiating startup sequence...
[Server@1d4c61c]: Server socket opened successfully in 16 ms.
[Server@1d4c61c]: Database [index=0, id=0, db=file:test, alias=1] opened sucessfully in 406 ms.
[Server@1d4c61c]: Startup sequence completed in 422 ms.
[Server@1d4c61c]: 2007-05-19 16:59:43.609 HSQLDB server 1.8.0 is online
[Server@1d4c61c]: To close normally, connect and execute SHUTDOWN SQL
[Server@1d4c61c]: From command line, use [Ctrl]+[C] to abort abruptly
```

The line "From command line, use [Ctrl]+[C] to abort abruptly" is highlighted with a red box [1].

- en [1] : on voit que pour arrêter le serveur, il suffira de faire Ctrl-C dans la fenêtre.

## 5.9.3 La base de données [test]

La base de données gérée par défaut se trouve dans le dossier [data] :



- en [1] : au démarrage, le SGBD HSQL exécute le script appelé [test.script]

```
1. CREATE SCHEMA PUBLIC AUTHORIZATION DBA
2. CREATE USER SA PASSWORD ""
3. GRANT DBA TO SA
4. SET WRITE_DELAY 10
```

- ligne 1 : un schéma [public] est créé
- ligne 2 : un utilisateur [sa] avec un mot de passe vide est créé
- ligne 3 : l'utilisateur [sa] reçoit les droits d'administration

Au final, un utilisateur ayant des droits d'administration a été créé. C'est cet utilisateur que nous utiliserons par la suite.

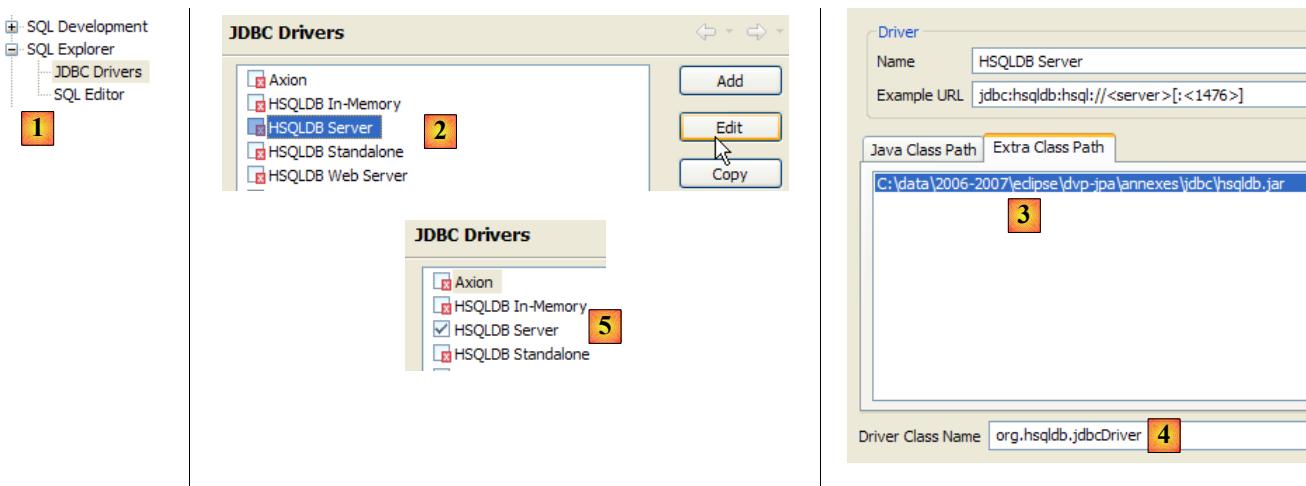
## 5.9.4 Pilote JDBC de HSQL

Le pilote Jdbc du SGBD HSQL se trouve dans le dossier [lib] :



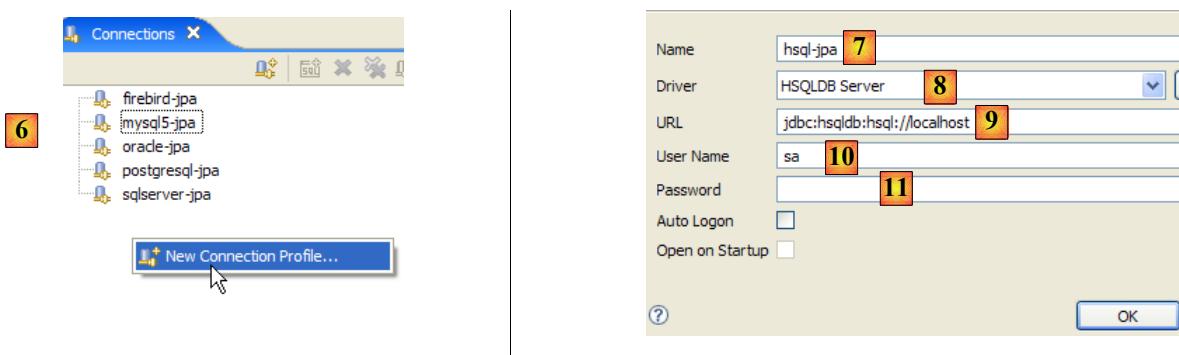
- en [1] : l'archive [hsqldb.jar] contient le pilote Jdbc du SGBD HSQL
- en [2] : nous plaçons cette archive comme les précédentes (paragraphe 5.4.7, page 255) dans le dossier <jdbc>

Pour vérifier ce pilote JDBC, nous allons utiliser Eclipse et le plugin SQL Explorer. Le lecteur est invité à suivre la démarche expliquée au paragraphe 5.4.7, page 255. Nous présentons quelques copies d'écran significatives :



- en [1] : [window / preferences / SQL Explorer / JDBC Drivers]
- en [2] : on configure le server [HSQLDB]
- en [3] : on désigne l'archive [hsqldb.jar] contenant le pilote Jdbc
- en [4] : le nom de la classe Java du pilote Jdbc
- en [5] : le pilote Jdbc est configuré

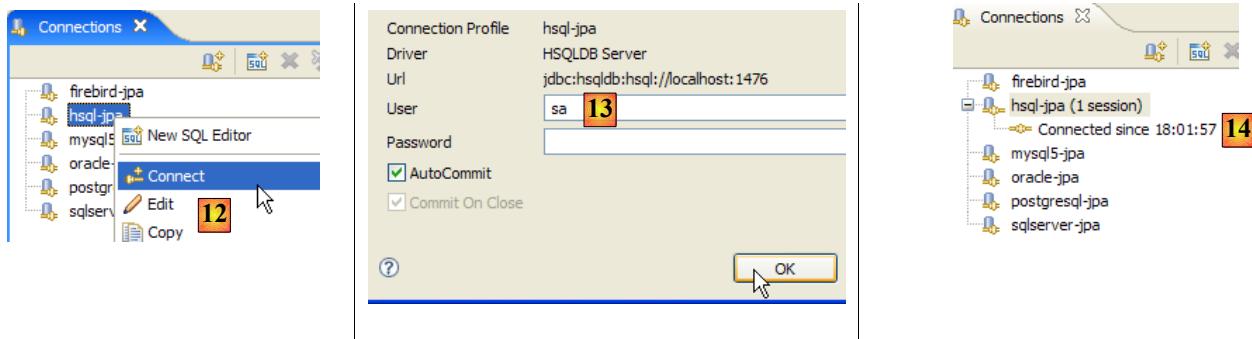
Ceci fait, on se connecte au serveur HSQL. On lance celui-ci auparavant.



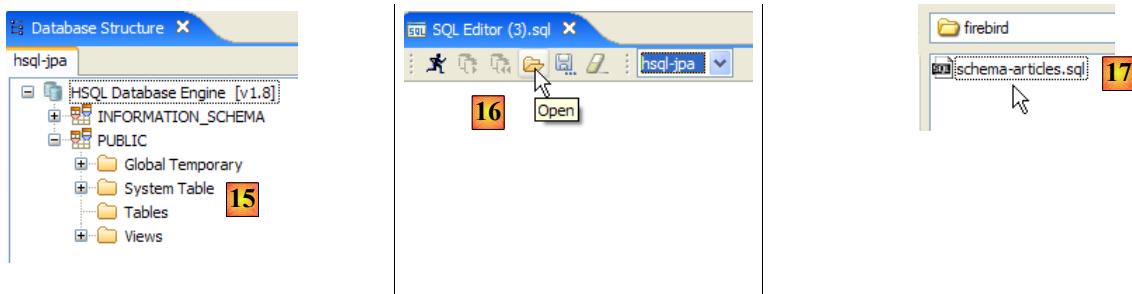
- en [6] : on crée une nouvelle connexion
- en [7] : on lui donne un nom
- en [8] : on veut se connecter au serveur HSQLDB
- en [9] : l'url de la base de données à laquelle on veut se connecter. Ce sera la base [test] vue précédemment.
- en [10] : on se connecte en tant qu'utilisateur [sa]. On a vu qu'il était administrateur du SGBD.

- en [11] : l'utilisateur [sa] n'a pas de mot de passe.

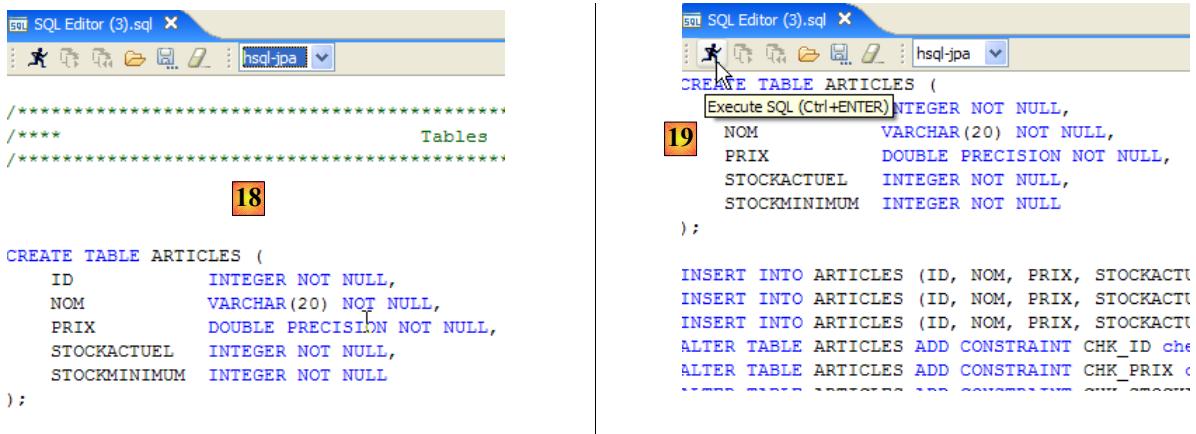
On valide la configuration de la connexion.



- en [12] : on se connecte
- en [13] : on s'identifie
- en [14] : on est connecté



- en [15] : le schéma [PUBLIC] n'a pas encore de table
- en [16] : on va créer la table [ARTICLES] à partir du script [schema-articles.sql] créé au paragraphe 5.4.6, page 253.
- en [17] : on sélectionne le script



- en [18] : le script à exécuter
- en [19] : on l'exécute après l'avoir débarrassé de tous ses commentaires car HSQLB n'accepte pas.

- une fois l'exécution du script faite, on rafraîchit en [20] l'affichage de la base
- en [21] : la table [ARTICLES] est bien là
- en [22] : son contenu

Arrêtons, puis relançons le serveur HSQLDB. Ceci fait, examinons le fichier [test.script] :

```

1. CREATE SCHEMA PUBLIC AUTHORIZATION DBA
2. CREATE MEMORY TABLE ARTICLES(ID INTEGER NOT NULL,NOM VARCHAR(20) NOT NULL,PRIX DOUBLE NOT
NULL,STOCKACTUEL INTEGER NOT NULL,STOCKMINIMUM INTEGER NOT NULL,CONSTRAINT PK_ARTICLES PRIMARY
KEY (ID),CONSTRAINT CHK_ID CHECK(ARTICLES.ID>0),CONSTRAINT CHK_PRIX
CHECK(ARTICLES.PRIX>0),CONSTRAINT CHK_STOCKACTUEL CHECK(ARTICLES.STOCKACTUEL>0),CONSTRAINT
CHK_STOCKMINIMUM CHECK(ARTICLES.STOCKMINIMUM>0),CONSTRAINT CHK_NOM
CHECK(ARTICLES.NOM!=''),CONSTRAINT UNQ_NOM UNIQUE(NOM))
3. CREATE USER SA PASSWORD ""
4. GRANT DBA TO SA
5. SET WRITE_DELAY 10
6. SET SCHEMA PUBLIC
7. INSERT INTO ARTICLES VALUES(1,'article1',100.0E0,10,1)
8. INSERT INTO ARTICLES VALUES(2,'article2',200.0E0,20,2)
9. INSERT INTO ARTICLES VALUES(3,'article3',300.0E0,30,3)

```

On voit que le SGBD a mémorisé les différents ordres SQL joués lors de la précédente session et qu'il les rejoue au démarrage de la nouvelle session. On voit par ailleurs (ligne 2) que la table [ARTICLES] est créée en mémoire (MEMORY). A chaque session, les ordres SQL émis sont mémorisés dans [test.log] pour être recopierés en début de session suivante dans [test.script] et rejoués en début de session.

## 5.10 Le SGBD Apache Derby

### 5.10.1 Installation

Le SGBD Apache Derby est disponible à l'url [<http://db.apache.org/derby/>]. C'est un SGBD écrit lui aussi en Java et également très léger en mémoire. Il présente des avantages analogues à HSQLDB. Lui aussi peut être embarqué dans des applications Java, c.a.d. être partie intégrante de l'application et fonctionner dans la même JVM.

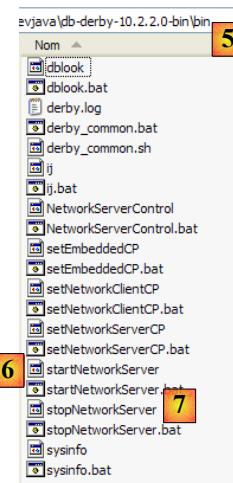
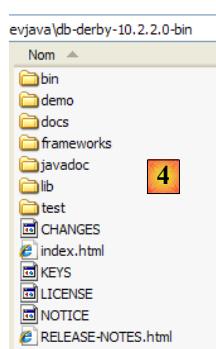
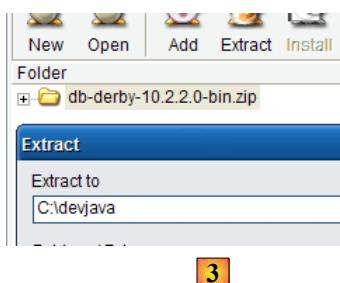
### What is Apache Derby?

Apache Derby, an [Apache DB subproject](#), is an open source relational database implemented entirely in Java and available under the [Apache License, Version 2.0](#). Some key advantages include:

- Derby has a small footprint -- about 2 megabytes for the base engine and embedded JDBC driver.
- Derby is based on the Java, JDBC, and [SQL](#) standard.
- Derby provides an embedded JDBC driver that lets you embed Derby in any Java-based solution.
- Derby also supports the more familiar client/server mode with the [Derby Network Client JDBC driver and Derby Network Server](#).
- Derby is easy to install, deploy, and use.

[db-derby-10.2.2.0-bin.zip](#) [PGF] [3] [MD5 ↗]  
[db-derby-10.2.2.0-bin.tar.gz](#) [3] [MD5 ↗]

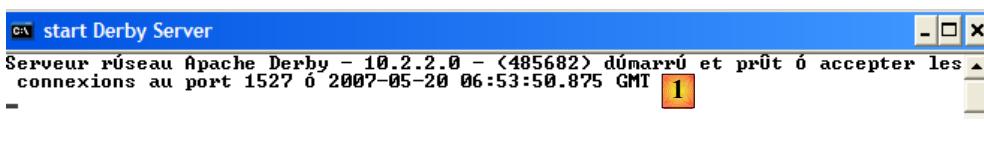
- en [1] : le site de téléchargement
- en [2,3] : prendre la version la plus récente



- en [3] : décompresser le fichier zip téléchargé
- en [4] : le dossier [db-derby-\*-bin] issu de la décompression
- en [5] : le dossier [bin] qui contient le script permettant de lancer le serveur [db derby] [6] et en [7], celui de l'arrêter.

## 5.10.2 Lancer / Arrêter Apache Derby (Db Derby)

Pour lancer le serveur Db Derby on double-clique sur l'application [startNetworkServer] [6] ci-dessus :



- en [1] : le serveur est démarré. On l'arrêtera avec l'application [stopNetworkServer] [7] ci-dessus.

## 5.10.3 Pilote JDBC de Db Derby

Le pilote Jdbc du SGBD Db Derby se trouve dans le dossier [lib] du dossier d'installation :

Download Community Documentation

### Apache Derby: Downloads

- [Latest Official Release](#)
- [Archived Official Releases](#)
- [Change History](#)

#### Latest Official Release

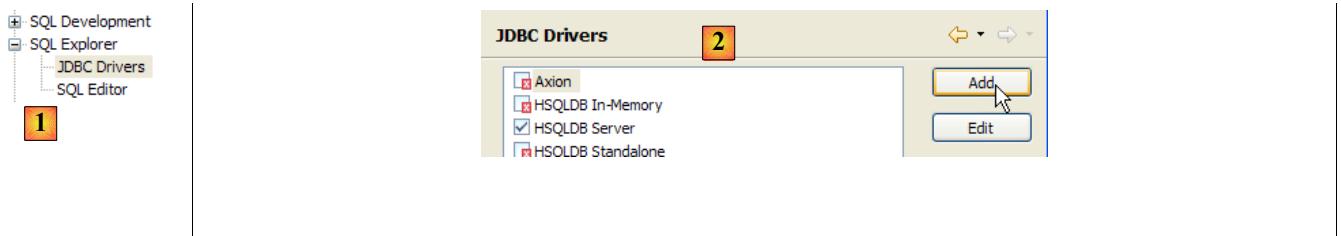
- [10.2.2.0](#) (Dec 12, 2006 / SVN 485682)

#### Archived Official Releases

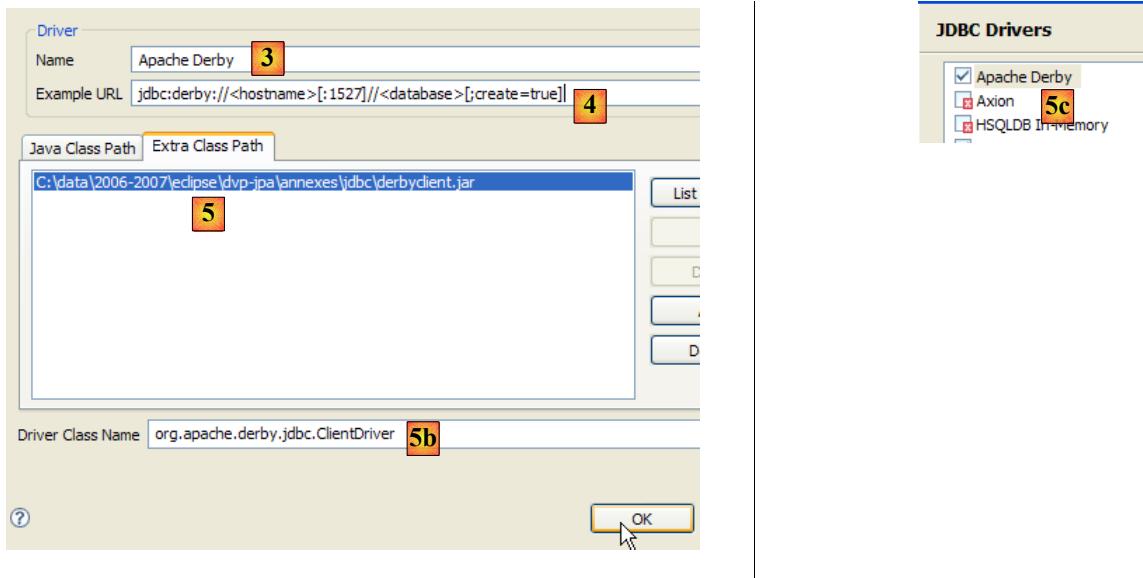


- en [1] : l'archive [derbyclient.jar] contient le pilote Jdbc du SGBD Db Derby
- en [2] : nous plaçons cette archive comme les précédentes (paragraphe 5.4.7, page 255) dans le dossier <jdbc>

Pour tester ce pilote JDBC, nous allons utiliser Eclipse et le plugin SQL Explorer. Le lecteur est invité à suivre la démarche expliquée au paragraphe 5.4.7, page 255. Nous présentons quelques copies d'écran significatives :

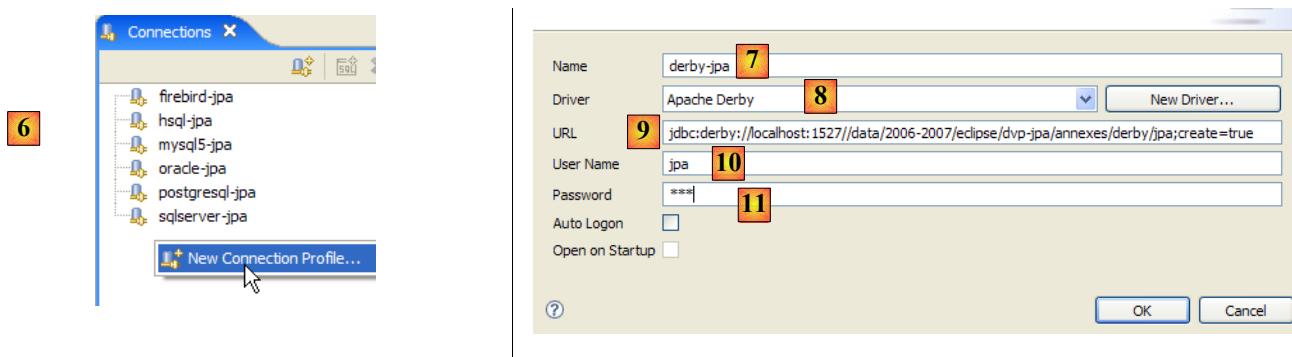


- en [1] : [window / preferences / SQL Explorer / JDBC Drivers]
- en [2] : le pilote Jdbc d'Apache Derby n'est pas dans la liste. On l'ajoute.



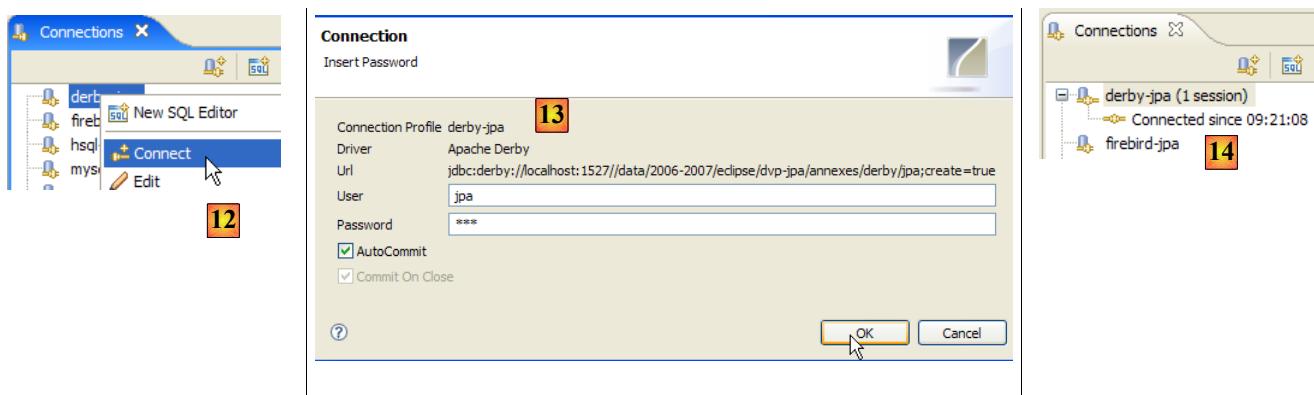
- en [3] : on donne un nom au nouveau pilote
- en [4] : on précise la forme des Url gérées par le pilote Jdbc
- en [5] : on a désigné l'archive .jar du pilote Jdbc
- en [5b] : le nom de la classe Java du pilote Jdbc
- en [5c] : le pilote Jdbc est configuré

Ceci fait, on se connecte au serveur Apache Derby. On lance celui-ci auparavant.

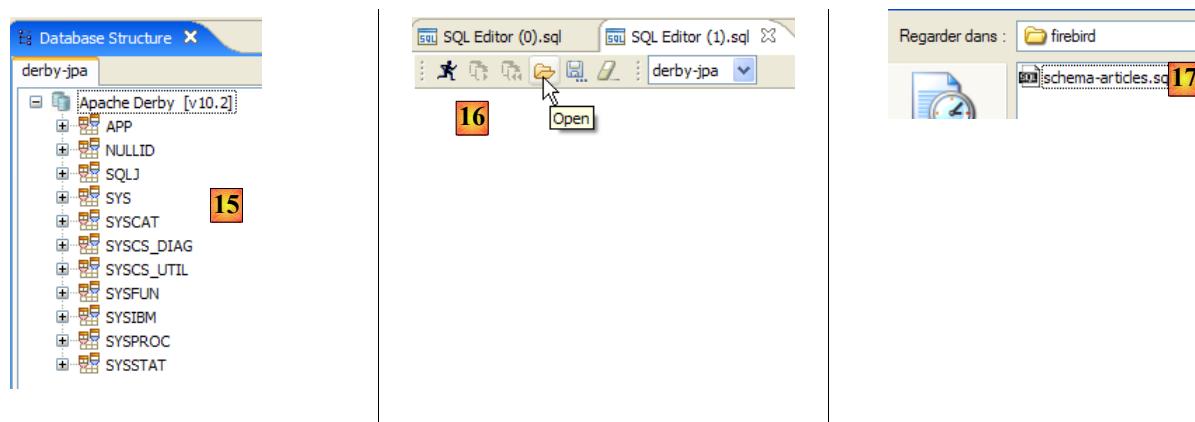


- en [6] : on crée une nouvelle connexion
- en [7] : on lui donne un nom
- en [8] : on veut se connecter au serveur Apache Derby
- en [9] : l'url de la base de données à laquelle on veut se connecter. Derrière le début standard [jdbc:derby://localhost:1527], on mettra le chemin d'un dossier du disque contenant une base de données Derby. L'option [create=true] permet de créer ce dossier s'il n'existe pas encore.
- en [10,11] : on se connecte en tant qu'utilisateur [jpa/jpa]. Je n'ai pas creusé la question mais il semble qu'on puisse mettre ce qu'on veut comme login / mot de passe. On déclare ici le propriétaire de la base, si create=true.

On valide la configuration de la connexion.



- en [12] : on se connecte
- en [13] : on s'identifie (jpa/jpa)
- en [14] : on est connecté



- en [15] : le schéma [jpa] n'apparaît pas encore.
- en [16] : on va créer la table [ARTICLES] à partir du script [schema-articles.sql] créé au paragraphe 5.4.6, page 253.
- en [17] : on sélectionne le script

A screenshot of the Derby SQL Editor interface. The title bar shows 'SQL Editor (0).sql' and 'SQL Editor (1).sql x'. A dropdown menu next to the title bar shows 'derby-jpa'. The main area contains a toolbar with icons for file operations, a search bar with 'Execute SQL (Ctrl+ENTER)', and a text area with a multi-line SQL command. The text area has a red border around the first line and the number '18' in a red box at the bottom left.

```
CREATE TABLE ARTICLES (
    ID          INTEGER NOT NULL,
    NOM         VARCHAR(20) NOT NULL,
    PRIX        DOUBLE PRECISION NOT NULL,
```

SQL Editor (3).sql

CREATE TABLE ARTICLES (

NOM VARCHAR(20) NOT NULL,  
 PRIX DOUBLE PRECISION NOT NULL,  
 STOCKACTUEL INTEGER NOT NULL,  
 STOCKMINIMUM INTEGER NOT NULL

);

INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL)

INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL)

INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL)

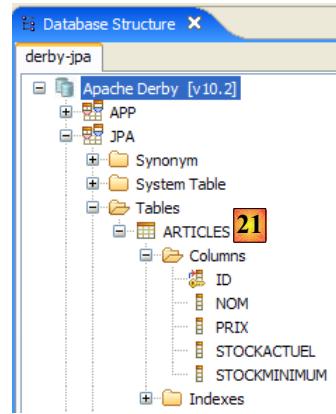
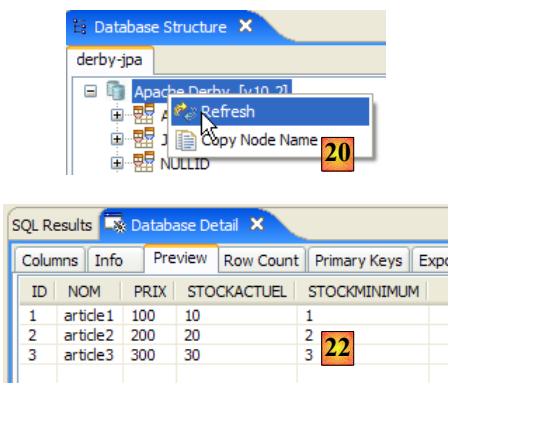
ALTER TABLE ARTICLES ADD CONSTRAINT CHK\_ID CHECK (ID > 0)

ALTER TABLE ARTICLES ADD CONSTRAINT CHK\_PRIX CHECK (PRIX > 0)

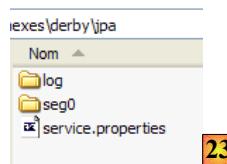
ALTER TABLE ARTICLES ADD CONSTRAINT CHK\_STOCKACTUEL CHECK (STOCKACTUEL > 0)

ALTER TABLE ARTICLES ADD CONSTRAINT CHK\_STOCKMINIMUM CHECK (STOCKMINIMUM > 0)

- en [18] : le script à exécuter
  - en [19] : on l'exécute après l'avoir débarrassé de tous ses commentaires car Apache Derby comme HSQLB ne les accepte pas.



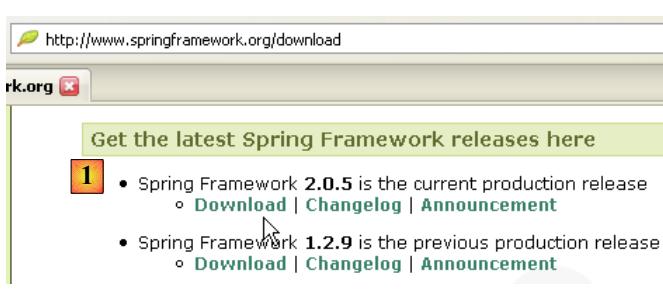
- une fois l'exécution du script faite, on rafraîchit en [20] l'affichage de la base en [21] : le schéma [jpa] et la table [ARTICLES] sont bien là
  - en [22] : le contenu de la table [ARTICLES]



- en [23] : le contenu du dossier [derby\jpa] dans lequel a été créée la base de données.

## 5.11 Le framework Spring 2

Le framework Spring 2 est disponible à l'url [<http://www.springframework.org/download>] :



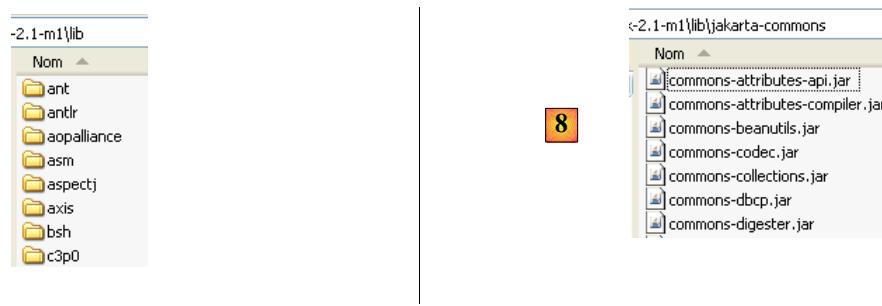
- en [1] : on télécharge la dernière version
- en [2] : on télécharge la version dite "avec dépendances" car elle contient les archives .jar des outils tierces que Spring intègre et dont on a tout le temps besoin.



- en [3] : on décomprime l'archive téléchargée
- en [4] : le dossier d'installation de Spring 2.1



- en [5] : dans le dossier <dist>, on trouve les archives de Spring. L'archive [spring.jar] réunit toutes les classes du framework Spring. Celles-ci sont également disponibles par modules dans le dossier <modules> en [6]. Si l'on connaît les modules dont on a besoin, on peut les trouver ici. On évite ainsi d'inclure dans l'application des archives dont elle n'a pas besoin.



- en [7] : le dossier <lib> contient les archives des outils tiers utilisés par Spring
- en [8] : quelques archives du projet [jakarta-commons]

Lorsque le tutoriel utilise des archives de Spring, il faut aller les chercher soit dans le dossier <dist>, soit dans le dossier <lib> du dossier d'installation de Spring.

## 5.12 Le conteneur EJB3 de JBoss

Le conteneur EJB3 de JBoss est disponible à l'url [<http://labs.jboss.com/jbossejb3/downloads/embeddableEJB3>] :

### JBoss EJB3 Downloads

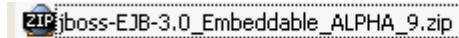
[JBoss EJB3](#) > [Downloads](#) > [Embeddable EJB3](#)

A release of EJB3 that can be used in any JEE application server.

Files:

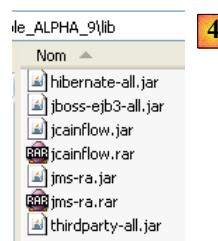
Name	Description	Size	Released	License	Download	Release Notes
<b>JBoss EJB 3.0 Preview RC9</b>	Alpha - Final Draft	15.4 MB	2006-09-14	LGPL		<a href="#">Release Notes</a>
--	--	--	--	--	--	--

2



15 719 Ko

- en [1] : on télécharge JBoss EJB3. On peut remarquer la date du produit (septembre 2006), alors qu'on télécharge en mai 2007. On peut se demander si ce produit évolue encore.
- en [2] : le fichier téléchargé



- en [3] : le fichier zip décompressé
- en [4] : les archives [hibernate-all.jar, jboss-ejb3-all.jar, thirdparty-all.jar] forment le conteneur EJB3 de JBoss. Il faut les mettre dans le classpath de l'application utilisant ce conteneur.

# Table des matières

<b>1INTRODUCTION.....</b>	<b>2</b>
<b>1.1OBJECTIFS.....</b>	<b>2</b>
<b>1.2RÉFÉRENCES.....</b>	<b>4</b>
<b>1.3OUTILS UTILISÉS.....</b>	<b>5</b>
<b>1.4TÉLÉCHARGEMENT DES EXEMPLES.....</b>	<b>5</b>
<b>1.5CONFIGURATION DES PROJETS ECLIPSE DES EXEMPLES.....</b>	<b>7</b>
<b>2LES ENTITÉS JPA.....</b>	<b>9</b>
<b>2.1EXEMPLE 1 - REPRÉSENTATION OBJET D'UNE TABLE UNIQUE.....</b>	<b>9</b>
<b>2.1.1La TABLE [PERSONNE].....</b>	<b>9</b>
<b>2.1.2L'ENTITÉ [PERSONNE].....</b>	<b>9</b>
<b>2.1.3Le projet ECLIPSE DES TESTS.....</b>	<b>12</b>
<b>2.1.4L'ENTITÉ [PERSONNE] (2).....</b>	<b>13</b>
<b>2.1.5CONFIGURATION DE LA COUCHE D'ACCÈS AUX DONNÉES.....</b>	<b>14</b>
<b>2.1.6GÉNÉRATION DE LA BASE DE DONNÉES AVEC UN SCRIPT ANT.....</b>	<b>17</b>
<b>2.1.7Exécution de la tâche ANT DDL.....</b>	<b>20</b>
<b>2.1.8Exécution de la tâche ANT BD.....</b>	<b>25</b>
<b>2.1.9Le contexte de persistance d'une application.....</b>	<b>26</b>
<b>2.1.10Un premier client JPA.....</b>	<b>29</b>
<b>2.1.10.1Le code.....</b>	<b>29</b>
<b>2.1.10.2L'exécution du code.....</b>	<b>30</b>
<b>2.1.11METTRE EN OEUVRE LES LOGS D'HIBERNATE.....</b>	<b>31</b>
<b>2.1.12DÉCOUVRIR LE LANGAGE JPQL / HQL AVEC LA CONSOLE HIBERNATE.....</b>	<b>34</b>
<b>2.1.13Un second client JPA.....</b>	<b>37</b>
<b>2.1.13.1La structure du code.....</b>	<b>38</b>
<b>2.1.13.2Test 1.....</b>	<b>40</b>
<b>2.1.13.3Test 2.....</b>	<b>41</b>
<b>2.1.13.4Test 3.....</b>	<b>41</b>
<b>2.1.13.5Test 4.....</b>	<b>42</b>
<b>2.1.13.6Test 5.....</b>	<b>43</b>
<b>2.1.13.7Test 6.....</b>	<b>44</b>
<b>2.1.13.8Test 7.....</b>	<b>44</b>
<b>2.1.13.9Test 8.....</b>	<b>45</b>
<b>2.1.13.10Test 9.....</b>	<b>46</b>
<b>2.1.13.11Test 10.....</b>	<b>47</b>
<b>2.1.13.12Test 11.....</b>	<b>49</b>
<b>2.1.13.13Test 12.....</b>	<b>50</b>
<b>2.1.14CHANGER DE SGBD.....</b>	<b>51</b>
<b>2.1.14.1Oracle 10g Express.....</b>	<b>51</b>
<b>2.1.14.2PostgreSQL 8.2.....</b>	<b>53</b>
<b>2.1.14.3SQL Server Express 2005.....</b>	<b>53</b>
<b>2.1.14.4Firebird 2.0.....</b>	<b>54</b>
<b>2.1.14.5Apache Derby.....</b>	<b>55</b>
<b>2.1.14.6HSQLDB.....</b>	<b>56</b>
<b>2.1.15CHANGER D'IMPLÉMENTATION JPA.....</b>	<b>57</b>
<b>2.1.15.1Le projet Eclipse.....</b>	<b>57</b>
<b>2.1.15.2La configuration de la couche JPA / Toplink.....</b>	<b>58</b>
<b>2.1.15.3Test [InitDB].....</b>	<b>59</b>
<b>2.1.15.4Test [Main].....</b>	<b>63</b>
<b>2.1.16CHANGER DE SGBD DANS L'IMPLÉMENTATION JPA / TOPLINK.....</b>	<b>64</b>
<b>2.1.16.1Oracle 10g Express.....</b>	<b>64</b>
<b>2.1.16.2Les autres SGBD.....</b>	<b>66</b>
<b>2.1.17CONCLUSION.....</b>	<b>67</b>
<b>2.2EXEMPLE 2 : RELATION UN-À-UN VIA UNE INCLUSION.....</b>	<b>68</b>
<b>2.2.1Le schéma de la base de données.....</b>	<b>68</b>
<b>2.2.2Les objets @Entity représentant la base de données.....</b>	<b>68</b>
<b>2.2.3L'environnement des tests.....</b>	<b>69</b>
<b>2.2.4Génération de la DDL de la base de données.....</b>	<b>70</b>
<b>2.2.5InitDB.....</b>	<b>70</b>
<b>2.2.6Main.....</b>	<b>72</b>

<u><a href="#">2.2.7</a></u> IMPLÉMENTATION JPA / TOPLINK.....	74
<u><a href="#">2.3</a></u> EXEMPLE 2 : RELATION UN-À-UN VIA UNE CLÉ ÉTRANGÈRE.....	76
<u><a href="#">2.3.1</a></u> LE SCHÉMA DE LA BASE DE DONNÉES.....	76
<u><a href="#">2.3.2</a></u> LES OBJETS @ENTITY REPRÉSENTANT LA BASE DE DONNÉES.....	76
<u><a href="#">2.3.3</a></u> LE PROJET ECLIPSE / HIBERNATE 1.....	78
<u><a href="#">2.3.4</a></u> GÉNÉRATION DE LA DDL DE LA BASE DE DONNÉES.....	79
<u><a href="#">2.3.5</a></u> INITDB.....	79
<u><a href="#">2.3.6</a></u> MAIN.....	80
<u><a href="#">2.3.6.1</a></u> Test1.....	80
<u><a href="#">2.3.6.2</a></u> Test2.....	81
<u><a href="#">2.3.6.3</a></u> Test4.....	81
<u><a href="#">2.3.6.4</a></u> Test5.....	82
<u><a href="#">2.3.6.5</a></u> Test6.....	83
<u><a href="#">2.3.6.6</a></u> Test7.....	83
<u><a href="#">2.3.7</a></u> PROJET ECLIPSE / HIBERNATE 2.....	85
<u><a href="#">2.3.8</a></u> CONSOLE HIBERNATE.....	87
<u><a href="#">2.3.9</a></u> IMPLÉMENTATION JPA / TOPLINK.....	91
<u><a href="#">2.4</a></u> EXEMPLE 3 : RELATION UN-À-PLUSIEURS.....	94
<u><a href="#">2.4.1</a></u> LE SCHÉMA DE LA BASE DE DONNÉES.....	94
<u><a href="#">2.4.2</a></u> LES OBJETS @ENTITY REPRÉSENTANT LA BASE DE DONNÉES.....	94
<u><a href="#">2.4.3</a></u> LE PROJET ECLIPSE / HIBERNATE 1.....	96
<u><a href="#">2.4.4</a></u> GÉNÉRATION DE LA DDL DE LA BASE DE DONNÉES.....	96
<u><a href="#">2.4.5</a></u> INITDB.....	96
<u><a href="#">2.4.6</a></u> MAIN.....	98
<u><a href="#">2.4.6.1</a></u> Test3.....	98
<u><a href="#">2.4.6.2</a></u> Test4.....	99
<u><a href="#">2.4.6.3</a></u> Test5.....	99
<u><a href="#">2.4.6.4</a></u> Test6.....	100
<u><a href="#">2.4.6.5</a></u> Test7.....	101
<u><a href="#">2.4.6.6</a></u> Test8.....	101
<u><a href="#">2.4.7</a></u> PROJET ECLIPSE / HIBERNATE 2.....	102
<u><a href="#">2.4.8</a></u> IMPLÉMENTATION JPA / TOPLINK - 1.....	104
<u><a href="#">2.4.9</a></u> IMPLÉMENTATION JPA / TOPLINK - 2.....	105
<u><a href="#">2.5</a></u> EXEMPLE 4 : RELATION PLUSIEURS-À-PLUSIEURS AVEC UNE TABLE DE JOINTURE EXPLICITE.....	108
<u><a href="#">2.5.1</a></u> LE SCHÉMA DE LA BASE DE DONNÉES.....	108
<u><a href="#">2.5.2</a></u> LES OBJETS @ENTITY REPRÉSENTANT LA BASE DE DONNÉES.....	108
<u><a href="#">2.5.3</a></u> LE PROJET ECLIPSE / HIBERNATE.....	112
<u><a href="#">2.5.4</a></u> GÉNÉRATION DE LA DDL DE LA BASE DE DONNÉES.....	112
<u><a href="#">2.5.5</a></u> INITDB.....	113
<u><a href="#">2.5.6</a></u> MAIN.....	116
<u><a href="#">2.5.6.1</a></u> Test2.....	116
<u><a href="#">2.5.6.2</a></u> Test3.....	118
<u><a href="#">2.5.6.3</a></u> Test4.....	119
<u><a href="#">2.5.6.4</a></u> Test5.....	119
<u><a href="#">2.5.7</a></u> IMPLÉMENTATION JPA / TOPLINK.....	120
<u><a href="#">2.6</a></u> EXEMPLE 5 : RELATION PLUSIEURS-À-PLUSIEURS AVEC UNE TABLE DE JOINTURE IMPLICITE.....	122
<u><a href="#">2.6.1</a></u> LE SCHÉMA DE LA BASE DE DONNÉES.....	122
<u><a href="#">2.6.2</a></u> LES OBJETS @ENTITY REPRÉSENTANT LA BASE DE DONNÉES.....	122
<u><a href="#">2.6.3</a></u> LE PROJET ECLIPSE / HIBERNATE.....	124
<u><a href="#">2.6.4</a></u> GÉNÉRATION DE LA DDL DE LA BASE DE DONNÉES.....	125
<u><a href="#">2.6.5</a></u> INITDB.....	126
<u><a href="#">2.6.6</a></u> MAIN.....	127
<u><a href="#">2.6.6.1</a></u> Test3.....	127
<u><a href="#">2.6.6.2</a></u> Test6.....	128
<u><a href="#">2.6.7</a></u> IMPLÉMENTATION JPA / TOPLINK.....	129
<u><a href="#">2.6.8</a></u> LE PROJET ECLIPSE / HIBERNATE 2.....	130
<u><a href="#">2.6.9</a></u> LE PROJET ECLIPSE / TOPLINK 2.....	131
<u><a href="#">2.7</a></u> EXEMPLE 6 : UTILISER DES REQUÊTES NOMMÉES.....	133
<u><a href="#">2.7.1</a></u> LA BASE DE DONNÉES EXEMPLE.....	133
<u><a href="#">2.7.2</a></u> LES OBJETS @ENTITY REPRÉSENTANT LA BASE DE DONNÉES.....	133
<u><a href="#">2.7.3</a></u> LE PROJET ECLIPSE / HIBERNATE.....	135
<u><a href="#">2.7.4</a></u> GÉNÉRATION DE LA DDL DE LA BASE DE DONNÉES.....	136
<u><a href="#">2.7.5</a></u> REQUÊTES JPQL AVEC UNE CONSOLE HIBERNATE.....	137
<u><a href="#">2.7.6</a></u> QUERYDB.....	138
<u><a href="#">2.7.7</a></u> LE PROJET ECLIPSE / TOPLINK.....	142

<b>2.8 CONCLUSION.....</b>	<b>142</b>
<b>3 JPA DANS UNE ARCHITECTURE MULTICOUCHES.....</b>	<b>143</b>
<b>  3.1 EXEMPLE 1 : SPRING / JPA AVEC ENTITÉ PERSONNE.....</b>	<b>145</b>
3.1.1 LE PROJET ECLIPSE / SPRING / HIBERNATE.....	145
3.1.2 LES ENTITÉS JPA.....	146
3.1.3 LA COUCHE [DAO].....	146
3.1.4 LA COUCHE [METIER / SERVICE].....	149
3.1.5 CONFIGURATION DES COUCHES.....	151
3.1.6 PROGRAMME CLIENT [INITDB].....	153
3.1.7 TESTS UNITAIRES [TESTNG].....	154
3.1.8 CHANGER DE SGBD.....	162
3.1.9 CHANGER D'IMPLÉMENTATION JPA.....	162
<b>  3.2 EXEMPLE 2 : JBOSS EJB3 / JPA AVEC ENTITÉ PERSONNE.....</b>	<b>167</b>
3.2.1 LE PROJET ECLIPSE / JBOSS EJB3 / HIBERNATE.....	167
3.2.2 LES ENTITÉS JPA.....	168
3.2.3 LA COUCHE [DAO].....	168
3.2.4 LA COUCHE [METIER / SERVICE].....	169
3.2.5 CONFIGURATION DES COUCHES.....	170
3.2.6 PROGRAMME CLIENT [INITDB].....	172
3.2.7 TESTS UNITAIRES [TESTNG].....	174
3.2.8 CHANGER DE SGBD.....	175
3.2.9 CHANGER D'IMPLÉMENTATION JPA.....	176
<b>  3.3 AUTRES EXEMPLES.....</b>	<b>177</b>
<b>  3.4 EXEMPLE 3 : SPRING / JPA DANS UNE APPLICATION WEB.....</b>	<b>179</b>
3.4.1 PRÉSENTATION.....	179
3.4.2 LE PROJET ECLIPSE.....	181
3.4.3 LA COUCHE [WEB].....	182
3.4.3.1 Configuration de l'application web.....	182
3.4.3.2 Les pages JSP / JSTL de l'application.....	184
3.4.3.3 Le contrôleur de l'application.....	188
3.4.4 LES TESTS DE L'APPLICATION WEB.....	197
3.4.5 VERSION 2.....	202
3.4.6 CHANGER D'IMPLÉMENTATION JPA.....	203
<b>  3.5 AUTRES EXEMPLES.....</b>	<b>206</b>
<b>4 CONCLUSION.....</b>	<b>207</b>
<b>5 ANNEXES.....</b>	<b>209</b>
<b>  5.1 JAVA.....</b>	<b>209</b>
<b>  5.2 ECLIPSE.....</b>	<b>209</b>
5.2.1 INSTALLATION DE BASE.....	209
5.2.2 CHOIX DU COMPILATEUR.....	214
5.2.3 INSTALLATION DES PLUGINS CALLISTO.....	215
5.2.4 INSTALLATION DU PLUGIN [TESTNG].....	217
5.2.5 INSTALLATION DU PLUGIN [HIBERNATE TOOLS].....	218
5.2.6 INSTALLATION DU PLUGIN [SQL EXPLORER].....	219
<b>  5.3 LE CONTENEUR DE SERVLETS TOMCAT 5.5.....</b>	<b>221</b>
5.3.1 INSTALLATION.....	221
5.3.2 DÉPLOIEMENT D'UNE APPLICATION WEB AU SEIN DU SERVEUR TOMCAT.....	226
5.3.3 DÉPLOIEMENT.....	226
5.3.4 ADMINISTRATION DE TOMCAT.....	227
5.3.5 GESTION DES APPLICATIONS WEB DÉPLOYÉES.....	230
5.3.6 APPLICATION WEB AVEC PAGE D'ACCUEIL.....	234
5.3.7 INTÉGRATION DE TOMCAT DANS ECLIPSE.....	235
<b>  5.4 LE SGBD FIREBIRD.....</b>	<b>240</b>
5.4.1 SGBD FIREBIRD.....	240
5.4.2 TRAVAILLER AVEC LE SGBD FIREBIRD AVEC IB-EXPERT.....	242
5.4.3 CRÉATION D'UNE TABLE DE DONNÉES.....	245
5.4.4 INSERTION DE DONNÉES DANS UNE TABLE.....	248
5.4.5 L'ÉDITEUR SQL DE [IB-EXPERT].....	248
5.4.6 EXPORTATION D'UNE BASE FIREBIRD VERS UN SCRIPT SQL.....	253
5.4.7 PILOTE JDBC DE FIREBIRD.....	255
<b>  5.5 LE SGBD MySQL5.....</b>	<b>258</b>
5.5.1 INSTALLATION.....	258

<b>5.5.2</b> LANCER / ARRÊTER MySQL.....	260
<b>5.5.3</b> CLIENTS D'ADMINISTRATION MySQL.....	261
<b>5.5.4</b> CRÉATION D'UN UTILISATEUR JPA ET D'UNE BASE DE DONNÉES JPA.....	262
<b>5.5.5</b> PILOTE JDBC DE MySQL.....	266
<b>5.6</b> Le SGBD POSTGRESQL.....	<b>268</b>
<b>5.6.1</b> INSTALLATION.....	268
<b>5.6.2</b> LANCER / ARRÊTER POSTGRESQL.....	270
<b>5.6.3</b> ADMINISTRER POSTGRESQL.....	271
<b>5.6.4</b> CRÉATION D'UN UTILISATEUR JPA ET D'UNE BASE DE DONNÉES JPA.....	271
<b>5.6.5</b> PILOTE JDBC DE POSTGRESQL.....	275
<b>5.7</b> Le SGBD ORACLE 10G EXPRESS.....	<b>276</b>
<b>5.7.1</b> INSTALLATION.....	276
<b>5.7.2</b> LANCER / ARRÊTER ORACLE 10G.....	278
<b>5.7.3</b> CRÉATION D'UN UTILISATEUR JPA ET D'UNE BASE DE DONNÉES JPA.....	279
<b>5.7.4</b> CRÉATION DE LA TABLE [ARTICLES] DE LA BASE DE DONNÉES JPA.....	280
<b>5.7.5</b> PILOTE JDBC DE ORACLEXE.....	282
<b>5.8</b> Le SGBD SQL SERVER EXPRESS 2005.....	<b>284</b>
<b>5.8.1</b> INSTALLATION.....	284
<b>5.8.2</b> LANCER / ARRÊTER SQL SERVER.....	284
<b>5.8.3</b> CRÉATION D'UN UTILISATEUR JPA ET D'UNE BASE DE DONNÉES JPA.....	286
<b>5.8.4</b> CRÉATION DE LA TABLE [ARTICLES] DE LA BASE DE DONNÉES JPA.....	288
<b>5.8.5</b> PILOTE JDBC DE SQL SERVER EXPRESS.....	289
<b>5.9</b> Le SGBD HSQLDB.....	<b>291</b>
<b>5.9.1</b> INSTALLATION.....	291
<b>5.9.2</b> LANCER / ARRÊTER HSQLDB.....	292
<b>5.9.3</b> LA BASE DE DONNÉES [TEST].....	292
<b>5.9.4</b> PILOTE JDBC DE HSQL.....	293
<b>5.10</b> Le SGBD APACHE DERBY.....	<b>295</b>
<b>5.10.1</b> INSTALLATION.....	295
<b>5.10.2</b> LANCER / ARRÊTER APACHE DERBY (Db DERBY).....	296
<b>5.10.3</b> PILOTE JDBC DE Db DERBY.....	296
<b>5.11</b> Le FRAMEWORK SPRING 2.....	<b>299</b>
<b>5.12</b> Le CONTENEUR EJB3 DE JBOSS.....	<b>301</b>