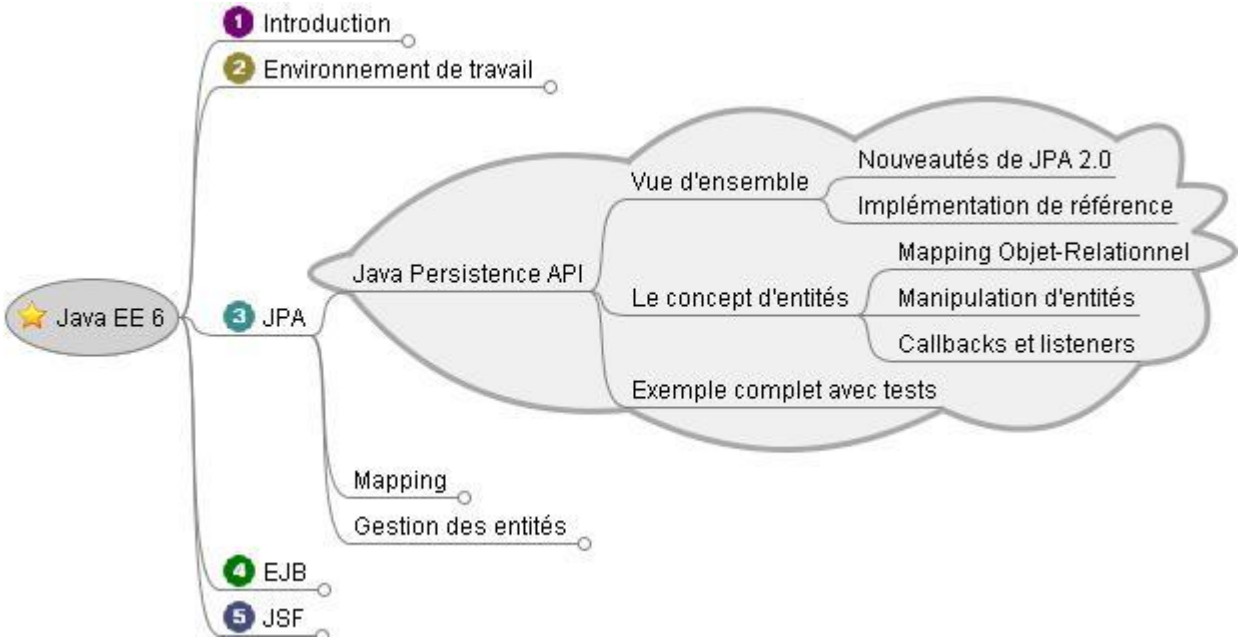
Java EE 6

Développer des applications d'entreprise

- **Chapitre 3**

  JPA – Java Persistence API

Agenda
Planning de la formation



Introduction
JPA – Java Persistence API

- Les applications ont besoin de faire persister la plupart de leurs données
- La plupart des applications utilisent des bases de données relationnelles pour le stockage
- Les bases relationnelles stockent les données dans des tables faites de lignes et de colonnes. Les données sont identifiées par des clés primaires faites de colonnes avec des contraintes d'unicité et parfois des index. Les relations sont gérées par des clés étrangères et des tables de jointures.

Introduction
JPA – Java Persistence API

- Java propose plusieurs solutions pour la persistence des données :
  - La sérialisation avec java.io.Serializable
  - Le stockage SQL avec JDBC
- Depuis quelques années et l'apparition de frameworks open source comme Hibernate, on utilise davantage les outils de conversion automatique objet-relationnel (Object-Relational Mapping tools ORM)

Introduction
JPA – Java Persistence API

- Les ORM sont des outils ou frameworks à qui on délègue l'accès à la base de données pour :
  - Présenter sous forme d'objets les données relationnelles
  - Stocker les objets sous forme de données relationnelles
- Ces outils s'occupent donc de la correspondance bi-directionnelle entre la base de données et les objets
- Plusieurs implémentations existent comme Hibernate, TopLink, Java Data Object (JDO), mais JPA est la technologie de Java EE 6

Vue d'ensemble
Historique

- JPA 1.0 est apparu avec Java EE 5 pour résoudre les problèmes de persistence objet
- JPA 2.0 de Java EE 6 suit le chemin de la simplicité et de la robustesse et ajoute de nouvelles fonctionnalités
- JPA est une abstraction de JDBC qui la rend indépendante de SQL
- Toutes les classes et annotations sont dans le package javax.persistence

Vue d'ensemble
Contenu

- Les composants principaux de JPA sont :
  - ORM : le mécanisme de transformation automatique objet-relationnel
  - Entity Manager API : interface pour réaliser des opérations liées aux bases de données comme Créer, Lire (Read), Mettre à jour (Update) ou supprimer (Delete) (CRUD). De cette façon on n'a pas besoin d'utiliser JDBC
  - Java Persistence Query Language (JPQL) : récupération de données dans un langage orienté objet
  - Transactions et vérrouillage : gère les accès concurrents
  - Callback et listeners : permet de lier la logique métier au cycle de vie des objets persistants

Vue d'ensemble
Nouveautés de JPA 2.0

- JPA 2.0 est une continuation de JPA 1.0. Elle garde l'approche orientée objet avec les annotations et des fichiers de mapping XML optionnels
- JPA 2.0 apporte de nouvelles API, étend JPQL et ajoute de nouvelles fonctionnalités :
  - Les collections de types simples (String, Integer, …) ainsi que d'objets Embeddable peuvent être mappés dans des tables séparées. Avant on ne pouvait que mapper séparément des collections d'entités JPA
  - Le support des Map a été étendu pour avoir des clés et valeurs de type simple, des entités ou des embeddable

Vue d’ensemble
Nouveautés de JPA 2.0

- Il est maintenant possible de maintenir l'ordre de persistence avec l'annotation @OrderColumn
- La suppression d'orphelins permet à des objets fils d'être supprimés d'une relation si l'objet parent est supprimé
- Le verrouillage pessimiste a été ajouté au verrouillage optimiste déjà présent
- Une nouvelle Query Definition API a été introduite pour construire des requêtes construites de façon objet
- La syntaxe de JPQL a été enrichie
- Les objets Embeddable peuvent maintenant être inclus dans d'autres objets Embeddable et avoir des relations avec les entités

Vue d’ensemble
Nouveautés de JPA 2.0

- La syntaxe de nativation par point (. ou dot) a été étendue pour gérer les embeddable avec relations et embeddables d'embeddables
- Une nouvelle API de cache a été introduite

Vue d'ensemble
Implémentation de référence

- EclipseLink 1.1 est l'implémentation de référence de JPA 2.0
- EclipseLink supporte :
  - ORM : persistence d'objets Java dans des bases
  - OXM (Object XML Mapping) : persistence d'objets Java en XML via JAXB
  - Persistence d'objets Java dans dans un EIS (Enterprise Information System) via Java EE Connector Architecture (JCA)
  - Les database web services

Vue d’ensemble
Implémentation de référence

- EclipseLink était avant Oracle TopLink, puis a été donné à la fondation Eclipse en 2006 avant de devenir l'implémentation de référence
- Le framework de persistence est aussi appelé persistence provider ou simplement provider

Le concept d'entités
Présentation

- Quand on parle d'objets mappés en base de données, le terme entités devrait être préféré au terme objet
  - Les objets sont simplement des instances qui vivent dans la mémoire de la jvm
  - Les entités sont des objets qui vivent provisoirement en mémoire et de façon permanente dans une base de données
- Les entités ont la possibilité d'être mappés avec une base, peuvent être concrets ou abstraits et supportent l'héritage, les associations et autres relations

Le concept d’entités
Présentation

- Une fois mappées, ces entités peuvent être gérées par JPA
- Une entité peut être persistée, supprimée et vous pouvez la rechercher en faisant une requête JPQL
- Vous pouvez manipuler les entités et l'ORM se charge de la base de donnée
- Une entité a également un cycle de vie. Vous pouvez brancher du code métier à ce cycle via les méthodes de callback et les listeners

Le concept d'entités
Mapping Objet-Relationnel

- L'ORM gère la correspondance (mapping) Objet-Relationnel
- La correspondance est décrite via des métadonnées
- 2 formats de métadonnées sont possibles :
  - Annotations : le code de la classe entité est directement annotée avec des annotations du package javax.persistence
  - Descripteurs XML : la correspondance est décrite dans un fichier XML externe déployé avec l'entité. Ce peut être pratique si la configuration dépend de l'environnement

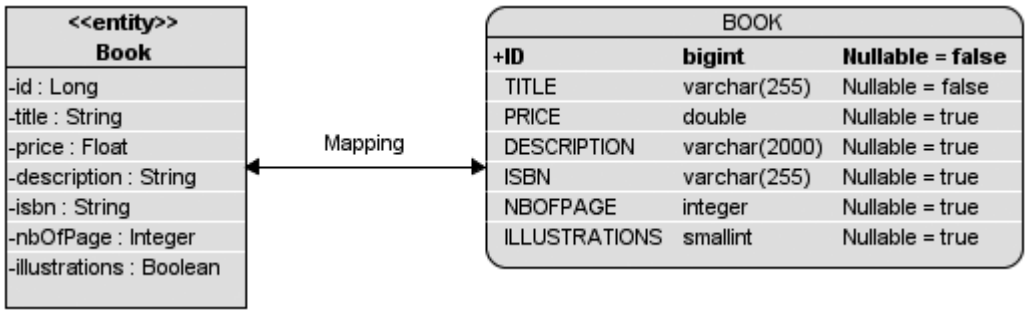
Le concept d’entités
Mapping Objet-Relationnel

- JPA comme la plupart des autres spécifications de JEE6 utilise le concept de configuration par l'exception :
  - Des règles par défaut sont définies (ex le nom de la table identique au nom de l'entité)
  - Si la configuration par défaut convient il n'y a rien à préciser
  - Sinon vous ajoutez des métadonnées
- Autrement dit, ajouter une configuration c'est ajouter une exception à la règle

Le concept d'entités
Mapping Objet-Relationnel

Book.java

```
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;
  @Column(nullable = false)
  private String title;
  private Float price;
  @Column(length = 2000)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Constructors, getters, setters
}
```



Le concept d’entités
Manipulation d’entités

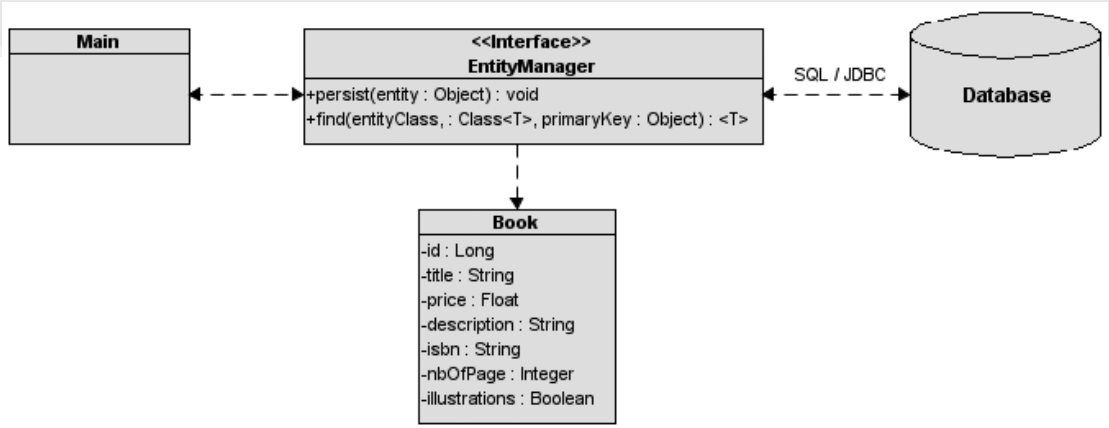
- JPA permet de faire des requêtes de façon objet sans avoir à connaîtres les détails de la structure de la base
- La pièce centrale responsable de l'orchestration des entités est l'Entity Manager. Son rôle est de gérer les entités, lire et écrire en base et permettre des opérations de type CRUD ainsi que des requêtes complexes JPQL
- D'un point de vue technique, l'Entity Manager est l'interface d'accès au persistence provider

Le concept d'entités
Manipulation d'entités

- Le code suivant permet de créer un Entity Manager et de persister une entité

MyDAO.java

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter02PU");
EntityManager em = emf.createEntityManager();
em.persist(book);
```



Le concept d'entités
Manipulation d'entités

- L'entity manager permet aussi de faire des recherches dans une syntaxe JPQL proche de SQL utilisant la syntaxe point (dot .)
- Pour retrouver tous les livres avec le titre H2G2 on écrira :

MyDAO.java

```
SELECT b FROM Book b WHERE b.title = 'H2G2'
```

Le concept d'entités
Manipulation d'entités

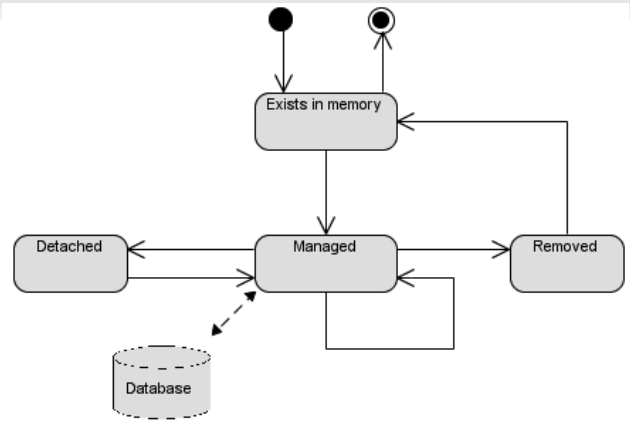
- Les requêtes JPQL peuvent être :
  - Dynamiques : créées dynamiquement à l'exécution
  - Statiques : créées statiquement à la compilation. On les appelle aussi requêtes nommées
  - Natives SQL
- On peut définir ainsi une requête statique :

MyDAO.java

```
@Entity
@NamedQuery(name = "findBookByTitle", query = "SELECT b FROM Book b WHERE b.title
='H2G2'")
public class Book {
    @Id @GeneratedValue
    private Long id;
    ...
```

Le concept d’entités
Callbacks et listeners

- Les entités sont de simples POJOs qui peuvent être managées ou non par l'entity manager
- Quand elles ne sont sont pas managées, c'est à dire quand elles sont détachées, elles peuvent être utilisées comme toute classe Java
- Les entités ont donc un cycle de vie



Le concept d’entités
Callbacks et listeners

- A chaque changement d'état correspond un événement Pre et Post (sauf pour le chargement avec seulement un Post)
- On peut demander à l'entity manager d'appeler une méthode au déclenchement d'un des événements. Il suffit de positionner une annotation, par exemple @PrePersist sur une méthode de l'entité

Exemple complet avec tests
Présentation

- L'objectif est d'écrire une entité Book et une classe Main qui persiste le livre en base
- L'ensemble sera compilé avec Maven 2
- EclipseLink et Derby seront utilisés
- Enfin un test unitaire sera créé

Exemple complet avec tests
Arborescence

- On suit la convention Maven :
  - src/main/java : classes Book et Main
  - src/main/resources : persistence.xml utilisé par Main
  - src/test/java : classe BookTest
  - src/test/resources : persistence.xml utilisé par les tests
  - pom.xml : décrit le projet et ses dépendances

Exemple complet avec tests
Entité Book

Book.java

```
@Entity
@NamedQuery(name = "findAllBooks", query = "SELECT b FROM Book b")
public class Book {
    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    // Constructors, getters, setters
}
```

Exemple complet avec tests
Classe Main

Main.java

```
public class Main {
    public static void main(String[] args) {
        // Creates an instance of book
        Book book = new Book();
        book.setTitle("The Hitchhiker's Guide to the Galaxy");
        book.setPrice(12.5F);
        book.setDescription("Science fiction comedy book");
        book.setIsbn("1-84023-742-2");
        book.setNbOfPage(354);
        book.setIllustrations(false);
        // Gets an entity manager and a transaction
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter02PU");
        EntityManager em = emf.createEntityManager();
        // Persists the book to the database
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        em.persist(book);
        tx.commit();
        em.close();
        emf.close();
    }
}
```

Exemple complet avec tests
Persistence Unit du Main

- Dans la classe Main, l'EntityManagerFactory a besoin d'un ”persistence unit” appelé ”chapter02PU”
- Ce ”persistence unit” doit être défini dans un fichier persistence.xml dans le dossier src/main/resources/META-INF
- Il contient tous les détails pour se connecter à la base et définir le mode de génération (create-tables, ...)

Exemple complet avec tests
Entité Book

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
  <persistence-unit name="chapter02PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>com.apress.javaee6.chapter02.Book</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="INFO"/>

      <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.url"
value="jdbc:derby://localhost:1527/chapter02DB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
  </persistence-unit>
</persistence>
```

Exemple complet avec tests
Compilation avec Maven

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ➥
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"➥
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 ➥
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.javaee6</groupId>
    <artifactId>chapter02</artifactId>
    <version>1.0</version>
    <name>chapter02</name>
    <dependencies>
        <dependency>
            <groupId>org.eclipse.persistence</groupId>
            <artifactId>javax.persistence</artifactId>
            <version>2.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.eclipse.persistence</groupId>
            <artifactId>eclipselink</artifactId>
            <version>2.0.1</version>
            <scope>provided</scope>
        </dependency>
```

Exemple complet avec tests
Compilation avec Maven

pom.xml

```
<dependency>
            <groupId>org.apache.derby</groupId>
                  <artifactId>derbyclient</artifactId>
                  <version>10.5.3.0_1</version>
        <scope>provided</scope>
           </dependency>
           <dependency>
                  <groupId>org.apache.derby</groupId>
                  <artifactId>derby</artifactId>
                  <version>10.5.3.0_1</version>
                  <scope>test</scope>
           </dependency>
           <dependency>
                  <groupId>junit</groupId>
                  <artifactId>junit</artifactId>
                  <version>4.5</version>
                  <scope>test</scope>
           </dependency>
      </dependencies>
```

Exemple complet avec tests
Compilation avec Maven

pom.xml

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <inherited>true</inherited>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Exemple complet avec tests
Lancement avec Derby

- Il faut démarrer Derby avant de lancer la classe Main :

```
%DERBY_HOME%\bin\startNetworkServer.bat
```

- Lancement de la classe Main avec Maven :

```
mvn exec:java -Dexec.mainClass="com.apress.javaee6.chapter02.Main"
```

- Vérification de la base :

```
ij
ij> connect 'jdbc:derby://localhost:1527/chapter02DB';
ij> show tables;
ij> describe book;
```

Exemple complet avec tests
Ecriture de la classe de test

- Tester un entity sans base n'a pas grand intérêt
- Pour tester un entity sans changer le code, on utilise une base en mémoire seulement et sans transactions. C'est le mode embedded de Derby. De cette façon la base est dans le même process et les tests s'exécutent rapidement

Exemple complet avec tests
Ecriture de la classe de test

BookTest.java

```
public class BookTest {
      private static EntityManagerFactory emf;
      private static EntityManager em;
      private static EntityTransaction tx;

      @BeforeClass
      public static void initEntityManager() throws Exception {
            emf = Persistence.createEntityManagerFactory(
                        "chapter02PU");
            em = emf.createEntityManager();
      }
      @AfterClass
      public static void closeEntityManager()
                  throws SQLException {
            em.close();
            emf.close();
      }
      @Before
      public void initTransaction() {
            tx = em.getTransaction();
      }
```

Exemple complet avec tests
Ecriture de la classe de test

BookTest.java

```
    @Test
    public void createBook() throws Exception {
        // Creates an instance of book
        Book book = new Book();
        book.setTitle("The Hitchhiker's Guide to the Galaxy");
        book.setPrice(12.5F);
        book.setDescription("Science fiction comedy book");
        book.setIsbn("1-84023-742-2");
        book.setNbOfPage(354);
        book.setIllustrations(false);
        // Persists the book to the database
        tx.begin();
        em.persist(book);
        tx.commit();
        assertNotNull("ID should not be null", book.getId());
        // Retrieves all the books from the database
        List<Book> books = ➥
           em.createNamedQuery("findAllBooks").getResultList();
        assertNotNull(books);
    }
}
```

Exemple complet avec tests
Persistence Unit du Test

- Le test utilise Derby en mode embarqué
- Pour cela il faut définir un autre persistence.xml dans src/test/resources/META-INF

Exemple complet avec tests
Persistence Unit du Test

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
  <persistence-unit name="chapter02PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>com.apress.javaee6.chapter02.Book</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY"/>
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
      <property name="eclipselink.logging.level" value="FINE"/>

      <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="javax.persistence.jdbc.url"
value="jdbc:derby:chapter02DB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
  </persistence-unit>
</persistence>
```

Exemple complet avec tests
Lancement des tests

- Avec Maven :

```
mvn test


Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 9.415 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------------------------
       BUILD SUCCESSFUL
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] Total time: 19 seconds
[INFO] Finished
[INFO] Final Memory: 4M/14M
[INFO] ------------------------------------------------------------------
```

Conclusion
Ce qu'il faut retenir

- L'écriture d'entité managées par l'Entity Manager JPA est très simple, c'est un simple POJO avec l'annotation @Entity et un attribut @Id
- L'Entity Manager fournit les principales méthodes CRUD
- JPA 2.0 permet d'écrire des tests unitaires avec une base séparée