

PERSISTANCE ET MAPPING OBJET-RELATIONNEL

Programmation 3
GCR2


I.AZAIEZ

2024

1

Vue d'ensemble

2



- Persistance et ORM
- Les entités
- Gestionnaire des entités

2

Chapitre 1

3

PERSISTENCE ET ORM

3

ORM: Object/Relational Mapping

4

- Beaucoup d'applications ont besoin d'une couche de persistance :
 - ▣ Accès à des données persistantes stockées dans un SGBD
 - ▣ Gestion des données persistantes dans le SGBD
- ORM résout la différence entre modèle Objet et modèle Relationnel
 - ▣ Les données sont dans la BDD relationnelle: orientée tables, N-uplets ou colonnes
 - ▣ L'application travaille avec des classes et des objets, pas avec des nuplets ou des colonnes
- Plusieurs frameworks qui ont exploré les meilleures pratiques pour la réalisation d'un système ORM : JPA, Hibernate, EclipseLink, etc.

4

Persistence des données

5

- A la fin d'une session d'utilisation d'une application orientée objet toutes les données des objets existant dans la mémoire vive de l'ordinateur seront perdues.
- Rendre **persistant un objet** c'est sauvegarder ses données sur un support non volatile de telle sorte qu'un objet identique à cet objet pourra être recréé lors d'une session ultérieure.
- La **persistance des données** des objets peut se faire via une **base de données relationnelle**.

5

Persistence des données: pourquoi?

6

- La gestion de la persistance des objets peut parfois être complexe :
 - A quel moment se connecte-t-on à la BDD ?
 - Doit-on recréer tous les objets ou seulement les objets modifiés ?
 - Quelle transformation (ou mapping) utiliser pour adapter la structure des données relationnelles au modèle objet ?
 - Faut-il monter en mémoire l'ensemble des données utilisées par tous les objets impliqués dans une transaction en une seule fois ?
 - Ou extraire les données de la base au fur et à mesure que les objets sont exécutés ?
- D'où l'utilité d'utiliser un Framework se chargeant d'une partie de la gestion de la persistance.

6

Persistence des données: les Besoins

- Création d'un modèle de persistance léger
- Supporter la modélisation complexe
 - ▣ héritage, polymorphisme
- Utiliser un ORM standard et efficace
 - ▣ Optimisé pour les SGBD relationnels
 - ▣ Utilise des annotations standardisées
- Gérer les requêtes
- Comporte plusieurs supports de persistance: mécanisme responsable de la sauvegarde et de la restauration des données. (persistence.xml)

7

Présentation de JPA : API

- JPA (Java Persistence API) est une API qui concerne la persistance des objets dans une base de données relationnelle
- JPA peut être utilisé par toutes les applications Java, Java SE ou Java EE, Spring
- C'est l'API ORM (Object-Relational Mapping) standard pour la persistance des objets Java

8

Principales propriétés de JPA

9

Une classe = Une table (en première approche)

Une variable de classe = Une colonne de la table

Mapping des types de classes avec les types disponibles dans la base de données

- problème de portabilité : JDBC ne suffit pas

Une annotation : **@Entity**

Une annotation obligatoire : **@Id**

- La clé primaire

Un gestionnaire de persistance

- Permet de manipuler les Entity
- Utilise dans un bean Session (pattern Facade) ou dans un client géré par le container

9

Fournisseur de persistance

10

- Comme pour JDBC, l'utilisation de **JPA** nécessite un fournisseur de persistance qui implémente les classes et les méthodes de l'API :
 - ▣ GlassFish est l'implémentation de référence de la spécification EJB 3
 - ▣ EclipseLink est l'implémentation de référence de la spécification JPA 2.0
 - ▣ D'autres implémentations : **Hibernate Entity Manager**, OpenJPA, BEA Kodo.

10

Chapitre 2

11

LES ENTITÉS (ENTITY)

11

Les entités

12

- Les classes dont les instances peuvent être persistantes sont appelées des entités dans la spécification de JPA
- Le développeur indique qu'une classe est une entité en lui associant l'annotation `@Entity`
- Ne pas oublier d'importer:

```
import javax.persistence.Entity;
```
- Dans les classes entités (importations semblables pour toutes les annotations)

12

Les entités : @Entity

13

- Cette annotation peut avoir un attribut **name** qui donne le nom de l'entité (rarement utilisé)
- Par défaut, le nom de l'entité est le nom de la classe (sans le nom du paquetage)
- Exemple :

```
@Entity
public class Etudiant {
```

Ou bien

```
@Entity (name="etud")
public class Etudiant {
```

13

Les entités : Exemple (1)

14

```
@Entity
public class Etudiant {
    @Id
    private int cin;
    private String nom;
    private String prenom;
    private Groupe groupe;

    //constructeur / setter / getter
}
```

Java

```
@Entity
public class Groupe {
    @Id
    private int idgp;
    private String nomgp;

    //constructeur / setter / getter
}
```

Etudiant

```
cin: int Primary key
nom: varchar(50)
prenom: varchar(50)
idgp: int
```

Relationnelle (BDD)

14

Les entités : les règles

15

- Pour qu'une classe puisse être persistante, il faut:
 - ▣ Qu'elle soit identifiée comme une entité (*Entity*) en utilisant l'annotation **@Entity**
 - ▣ Elle doit posséder un attribut qui représente la clé primaire dans la BDD : **@Id**
 - ▣ Qu'elle ait un constructeur sans argument *protected* ou *public*.

15

Les entités : clé primaire (1/2)

16

- Une entité doit avoir un attribut qui correspond à la clé primaire de la table associée
- La valeur de cet attribut ne doit jamais être modifiée.
- L'attribut clé primaire est désigné par l'annotation **@Id**
- Le type de la clé primaire doit être d'un des types suivants:
 - ▣ type primitif Java
 - ▣ classe qui enveloppe un type primitif
 - ▣ `java.lang.String`
 - ▣ `java.util.Date`
 - ▣ `java.sql.Date`

16

Les entités : clé primaire (2/2)

17

- Possibilité de générer automatiquement les clés
- Si la clé est de type numérique
 - ▣ **@GeneratedValue** indique que la clé sera automatiquement générée par le SGBD
 - ▣ l'annotation peut avoir un attribut **strategy** qui indique comment la clé sera générée:
 - **AUTO** : le SGBD choisit (valeur par défaut)
 - **SEQUENCE** : il utilise une séquence SQL
 - **IDENTITY** : il utilise un générateur de type IDENTITY (auto increment dans MySQL par exemple)
 - **TABLE** : il utilise une table qui contient la prochaine valeur de l'identificateur

17

Les entités : attribut

18

- Paramétrage en utilisant l'annotation **@Column**
- Les attributs de **@Column(...)** :
 - ▣ **name** : nom de l'attribut
 - ▣ **unique** : la valeur est-elle unique ?
 - ▣ **nullable** : accepte une valeur nulle ?
 - ▣ **insertable** : autorise ou non l'attribut à être mis à jour
 - ▣ **columnDefinition** : définition DDL de l'attribut
 - ▣ **table** : lorsque l'attribut est utilisé dans plusieurs tables
 - ▣ **length** : longueur max
 - ▣ **precision** : précision pour les valeurs numériques

18

Les entités : Exemple (2)

```
@Entity
public class Etudiant {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int cin;
    @Column(nullable=false) private String nom;
    @Column(nullable=false, length=45) private String prenom;
    @Column(name="gp") private String groupe;
    //constructeur / setter / getter
}
```

Table: etudiant

cin	nom	prenom	gp
1	ali	moh	gcp
2	sami	rjeb	gcr
3	ali	moh	gea
4	rami	benali	gcr
5	rami	benali	gcr

19

Association entre les entités

- Comment représenter l'association entre deux entités?
 - ▣ lier un *etudiant* a son *groupe*
 - ▣ lier un *etudiant* a des *cours*
- Trois possibilités en fonction des cardinalités
 - ▣ combien d'instances peuvent se trouver de chaque côté de l'association?
 - 1 *etudiant* est relié a 1 *groupe* G
 - 1 *groupe* G est relié a plusieurs *etudiants*
 - 1 *etudiant* est relié a plusieurs *cours*
 - 1 *cours* est relié a plusieurs *etudiants*

Comment ?

20

Associations

21

- Une association peut être uni- ou bidirectionnelle
- Cardinalités: 1-1, 1-N, N-1, M-N
- Elles sont définies grâce à une annotation sur la propriété correspondante
 - ▣ Annotations dédiées :

Annotation	Annote un(e)
@OneToOne	Object
@OneToMany	List<>
@ManyToOne	Object
@ManyToMany	List<>

21

Association 1-1

22

- Annotation **OneToOne**
 - ▣ Représentée par une clé étrangère ajoutée dans la table qui correspond au côté propriétaire.
 - ▣ L'annotation **OneToOne** est optionnelle
 - Exemple :

```
@Entity
public class Etudiant {
    @Id
    @OneToOne
    private Compte c;
    //...
}
```

```
@Entity
public class Compte{
    @Id
    private int idCpte;
    private String user;
    private String pswd;
}
```

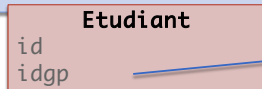
22

Association I-N et N-I

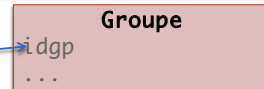
23

- Annotations `@OneToOne` et `@ManyToOne`
 - Représentée par une clé étrangère dans la table qui correspond au côté propriétaire (obligatoirement le côté Many)

```
@Entity
public class Etudiant {
    @Id
    private int cin;
    //...
    @ManyToOne
    @JoinColumn(name="idgp")
    private Groupe groupe;
}
```



```
@Entity
public class Groupe {
    @Id
    private int idgp;
    private String descgp;
    @OneToMany
    List<Etudiant> etudiants;
}
```



23

Association M-N

24

- Nous en venons maintenant aux associations de type "plusieurs-à-plusieurs" annotées par `@ManyToMany`
- Exemple :
 - un film a plusieurs acteurs;
 - un acteur joue dans plusieurs films;
 - un internaute note plusieurs films;
 - un film est noté par plusieurs internautes.

24

Association M-N

25

- Quand on représente une association **plusieurs-plusieurs** en relationnel, l'association devient une table dont la clé est la concaténation des clés des deux entités de l'association.
 - ▣ Nous obtenons alors: une table **Role** représente la première association avec une clé composée (id_film, id_acteur) et une table **Notation** représente la seconde avec une clé (id_film, email).
- Et il faut noter de plus que chaque partie de la clé est elle-même une clé étrangère.
- La recommandation dans ce cas est de promouvoir l'association en entité, en lui donnant un identifiant qui lui est propre.

25

Association M-N

26

□ Exemple :

```
@Entity
public class Film{
    @Id
    private int idf;
    //...
    @ManyToMany
    List<Film> acteurs;
}
```

```
@Entity
public class Acteur{
    @Id
    private int ida;
    //...
    @ManyToMany(mappedBy="acteurs")
    List<Film> films;
}
```

26

Association M-N

27

- Si le *mapping* par défaut ne convient pas, on peut le surcharger avec l'annotation **@JoinTable**
 - ▣ L'autre côté doit toujours comporter l'attribut **mappedBy**

```
@Entity
public class Film{
    @Id
    private int idf;
    //...
    @ManyToMany
    @JoinTable(name="Role",
        joinColumns=@JoinColumn(name="films_id"),
        inverseJoinColumns=@JoinColumn(name="acteurs_id"))
    List<Film> Acteurs;
}
```

27

Association M-N

28

- **@JoinTable**
 - ▣ donne des informations sur la table association qui va représenter l'association
 - ▣ attribut **name** donne le nom de la table
 - ▣ attribut **joinColumns** donne les noms des attributs de la table qui référencent les clés primaires du côté propriétaire de l'association
 - ▣ attribut **inverseJoinColumns** donne les noms des attributs de la table qui référencent les clés primaires du côté qui n'est pas propriétaire de l'association

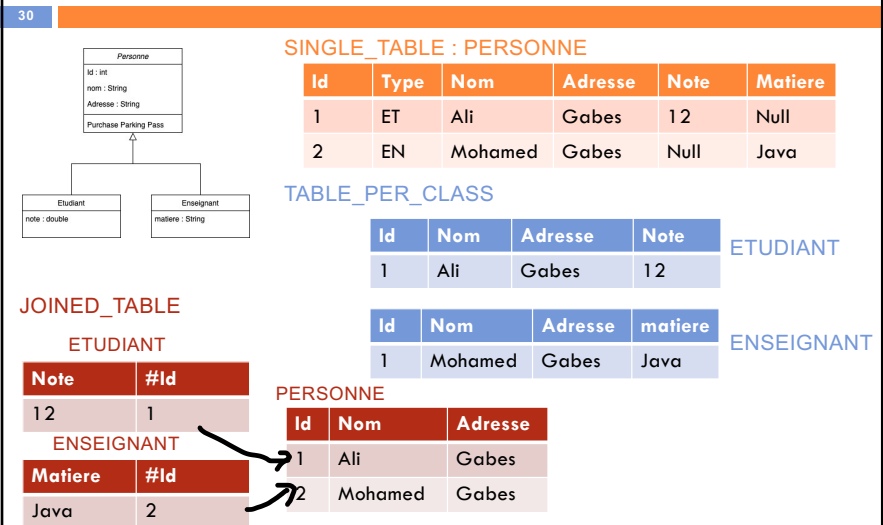
28

Mapping de l'héritage

- Différente stratégie de mapping de l'héritage
 - ▣ **SINGLE_TABLE** : l'héritage est représenté par une seule table en base de données
 - ▣ **JOINED_TABLE** : l'héritage est représenté par une jointure entre la table de l'entité parente et la table de l'entité enfant
 - ▣ **TABLE_PER_CLASS** : l'héritage est représenté par une table par entité

29

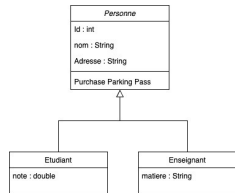
Mapping de l'héritage



30

Mapping de l'héritage : Stratégie **SINGLE_TABLE**

31

SINGLE_TABLE : PERSONNE

type	id	adresse	nom	matiere	note
ETD	1	Gabes	Ali	NULL	12
ENS	2	Gabes	Mohamed	Java	NULL

```

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "Type", length = 4)
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long ID;

    private String nom;
    private String adresse;
  
```

```

@Entity
@DiscriminatorValue("ENS")
public class Enseignant extends Personne {
  
```

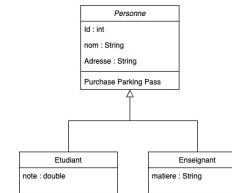
```

@Entity
@DiscriminatorValue("ETD")
public class Etudiant extends Personne {
    double note;
  
```

31

Mapping de l'héritage : Stratégie **TABLE_PER_CLASS**

32



id	adresse	nom	matiere
1	Gabes	Mohamed	Java

ENSEIGNANT

id	adresse	nom	note
2	Gabes	Ali	12

ETUDIANT

```

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private Long ID;

    private String nom;
    private String adresse;
  
```

```

@Entity
public class Enseignant extends Personne {
    String matiere;
  
```

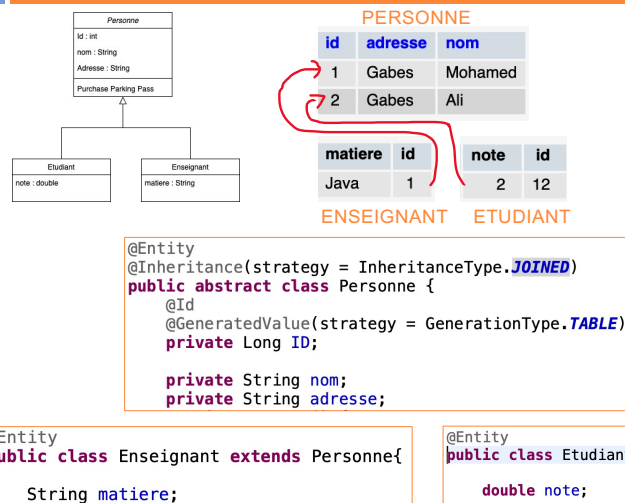
```

@Entity
public class Etudiant extends Personne {
    double note;
  
```

32

Mapping de l'héritage : Stratégie JOINED_TABLE

33



33

Exemple : gestion des soutenances

34

- On veut créer une application Web JEE, en se basant sur le mapping JPA, de planification des soutenances de PFE dédiée au département GCR. Pour planifier une soutenance, on doit mentionner le nom de l'étudiant et les membres de jury (encadrant, président de jury et le membre de jury), ainsi que le titre du PFE et la date et l'heure de la soutenance.
- De plus, on doit satisfaire les règles suivantes :
 - ▣ Une soutenance est planifiée une et une seule fois pour un étudiant.
 - ▣ Pour une date donnée, on ne peut planifier qu'une soutenance par heure (les soutenances se déroulent entre 8h et 16h).
 - ▣ Le jury est composé de trois membres différents.

34

Chapitre 3

35

IMPLÉMENTATION DE JPA EclipseLink V. 2

35

Gestionnaire des entités

36

□ Principe de base:

- Une instance d'entité ne devient persistante que lorsque l'application appelle la méthode appropriée du gestionnaire d'entités (**persist** ou **merge**)
- C'est une configuration nommée qui contient les informations nécessaires à l'utilisation d'une base de données.
- Les informations sur une unité de persistance sont données dans un fichier **persistence.xml** situé dans un sous-répertoire **META-INF**.

36

Gestionnaire des entités

37

- Il est nécessaire d'indiquer au fournisseur de persistance comment il peut se connecter à la base de données:
 - ▣ Les informations peuvent être données dans un fichier **persistence.xml** situé dans un répertoire **META-INF** dans le classpath
 - ▣ Ce fichier peut aussi comporter d'autres informations :

37

Gestionnaire des entités

38

- Exemple d'un fichier **persistence.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

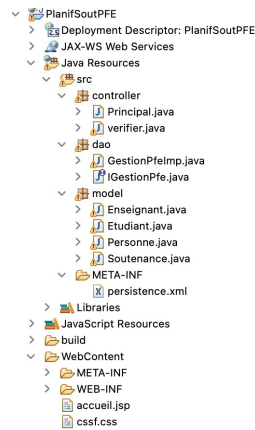
  <persistence-unit name="test1" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>model.Etudiant</class>
    <properties>
      <property name="javax.persistence.jdbc.password" value="root" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/Agenda" />
      <property name="eclipselink.logging.level" value="INFO" />
    </properties>
  </persistence-unit>
</persistence>
```

38

Gestionnaire des entités

39

Structure
de projet :
EclipseLink



39

Gestionnaire des entités

40

- **EntityManager** est donc la classe qui va permettre au développeur de manipuler ses objets Java devenus des entités et ainsi lui permettre de les persister.
- Il est donc nécessaire d'obtenir une référence vers un objet **EntityManager**; cela s'effectue par l'appel à la méthode **factory** de la classe **EntityManagerFactory**, comme montré ci-dessous:

```

EntityManagerFactory factory =
Persistence.createEntityManagerFactory("test1");
EntityManager em = factory.createEntityManager();

//test1 c'est l'unité de persistance
  
```

40

Gestionnaire des entités

41

- **EntityManager** permet de créer et exécuter les requêtes grâce à ses méthodes :

Méthodes	Descriptions
persist	insère une entité dans la base
merge	synchronise l'état des entités détachées
remove	supprime une entité de la BD
find	trouve une instance dans la BDD à partir par sa PK et le met dans le contexte
createQuery	Créer une requête JPQL

41

Gestionnaire des entités

42

- On peut ensuite utiliser cette instance de l'EntityManager "em" pour réaliser les opérations suivantes:
- **Exemple 1**: recherche d'un objet par son id

```
Etudiant e = em.find(Etudiant.class, id);
```

- On peut aussi définir nos fonctions de recherche en utilisant des requêtes SQL avec **createQuery** et **createNativeQuery**. Le résultat récupérer par:
 - **getResultList()** : permet de récupérer le résultat sous forme d'une liste
 - **getSingleResult()** : permet de récupérer un seul élément de la sélection
 - **getFirstResult()** : permet de récupérer le premier élément de la sélection

42

Gestionnaire des entités

43

- **Exemple 2:** recherche par requête avec `getResultList()`

```
Query q = em.createQuery("SELECT e FROM Etudiant e");
List<Etudiant> etdList = q.getResultList();
```

- **Exemple 3:** recherche par requête avec `getSingleResult()` et `createNativeQuery()`

```
Query q = em.createNativeQuery("select * from
Etudiant where num=?", model.Etudiant.class);
q.setParameter(1,102);
Etudiant e = (Etudiant) q.getSingleResult();
```

43

Gestionnaire des entités

44

- **Exemple 4:** Ajouter un objet Etudiant e

```
em.getTransaction().begin();
em.persist(e);
em.getTransaction().commit();
```

- **Exemple 5:** Modifier un objet Etudiant e

```
em.getTransaction().begin();
Etudiant e1 = em.find(Etudiant.class,
e.getCin());
e1.setNom("Dupond");
em.merge(e1);
em.getTransaction().commit();
```

44

Gestionnaire des entités

45

```
public void addSoutenance(Soutenance s) {  
    em.getTransaction().begin();  
    try {  
        em.persist(s);  
        em.getTransaction().commit();  
    }  
    catch(Exception e){  
        em.getTransaction().rollback();  
        e.printStackTrace();  
    }  
}
```

45

Gestionnaire des entités

46

Exemple 6: Supprimer un objet par son id

```
em.getTransaction().begin();  
Etudiant e = em.find(Etudiant.class, cin);  
em.remove(e);  
em.getTransaction().commit();
```

46

Chapitre 4

47

IMPLÉMENTATION DE JPA

Hibernate

47

Gestionnaire des entités

48

- **Hibernate** est une solution open source de type ORM (Object Relational Mapping) qui permet de faciliter le développement de la couche persistance d'une application.
- **Hibernate** permet donc de représenter une base de données en objets Java et vice versa.

48

Gestionnaire des entités

49

- Pour assurer le **mapping**, **Hibernate** a besoin d'un fichier de correspondance (mapping file) au format XML qui va contenir des informations sur la correspondance entre la classe définie et la table de la base de données.
- Il est donc nécessaire d'indiquer au fournisseur de persistance comment il peut se connecter à la base de données:
 - ▣ Les informations peuvent être données dans un fichier **hibernate.cfg.xml** situé dans le répertoire **SCR**
 - ▣ Ce fichier peut aussi comporter d'autres informations :

49

Gestionnaire des entités

50

- **Fichier hibernate.cfg.xml (fichier de configuration d'Hibernate)**

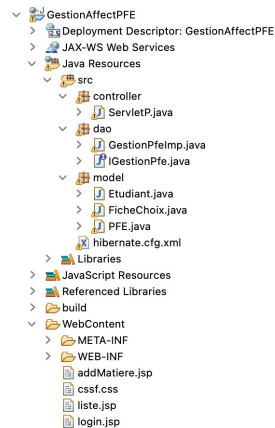
```
<?xml version="1.0" encoding="UTF-8"?>
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost:3306/GestionLivre</property>
<property name="connection.username">root</property>
<property name="connection.password">root</property>
<property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
<property name="current_session_context_class">thread</property>
<property name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider</property>
<property name="show_sql">true</property>
<!-- Drop and re-create the database schema on startup -->
<property name="hbm2ddl.auto">update</property>
<!-- mapping des entites -->
<mapping class="tn.enig.model.Livre" />
</session-factory>
</hibernate-configuration>
```

50

Gestionnaire des entités

51

Structure
de projet :
Hibernate



51

Gestionnaire des entités

52

- Construire l'usine de **gestionnaire d'entité** à partir de l'objet de configuration qui inclut toutes les métadonnées de la cartographie d'**Objet/Relationnel**

```

SessionFactory sessionFactory ;
ServiceRegistry serviceRegistry =
    new StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
Metadata metadata = new MetadataSources(serviceRegistry).getMetadataBuilder().build();
sessionFactory=metadata.getSessionFactoryBuilder().build();
  
```

- Une fois que la fabrique de session (**SessionFactory**) est créée, une connexion est créée et par la méthode **openSession()** on ouvre une session :

```
Session session=sessionFactory.openSession();
```

52

Gestionnaire des entités

53

- **Session** permet de créer et exécuter les requêtes grâce à ses méthodes :

Méthodes	Descriptions
save	Insère une entité dans la base
merge	Synchronise l'état des entités détachées
delete	Supprime une entité de la BD
load	Trouve une instance dans la BDD à partir par sa PK et le met dans le contexte
createQuery	Créer une requête JPQL

53

Gestionnaire des entités

54

- On peut ensuite utiliser cette instance de Session pour réaliser les opérations suivantes:

- **Exemple1**: recherche d'un objet par son **id**

```
Livre lv = this.session.load(Livre.class, id);
```

- On peut aussi définir nos fonctions de recherche en utilisant des requêtes SQL avec **createQuery**. Le résultat récupérer par:
 - **getResultList()** : permet de récupérer le résultat sous forme d'une liste
 - **getSingleResult()** : permet de récupérer un seul élément de la sélection

54

Gestionnaire des entités

55

- Exemple 2 : recherche par requête avec `getSingleResult()`

```
this.session.getTransaction().begin();
String sql = "Select d from Livre d Where d.id = :id";
Query<Livre> query = this.session.createQuery(sql);
query.setParameter("id", id);
Livre liv=query.getSingleResult();
this.session.getTransaction().commit();
```

- Exemple 3 : recherche par requête avec `getResultList()`

```
Query<Livre> query = this.session.createQuery("Select e
from Livre e ");
List<Livre> livs = query.getResultList();
```

55

Gestionnaire des entités

56

- Exemple 4: Ajouter un objet Livre lv

```
this.session.getTransaction().begin();
this.session.save(lv);
this.session.getTransaction().commit();
```

- Exemple 5: Modifier un objet Livre lv

```
this.session.getTransaction().begin();
this.session.merge(lv);
this.session.getTransaction().commit();
```

56

Gestionnaire des entités

57

Exemple 6: Supprimer un objet par son id

```
Livre lv = this.session.load(Livre.class, id);  
this.session.getTransaction().begin();  
this.session.delete(lv);  
this.session.getTransaction().commit();
```

57

58

JPQL

58

JPQL

59

- JPQL est un langage de requêtes adapté à la spécification JPA
- Inspiré du langage SQL (et HQL : Hibernate Query Language) mais adapté aux entités JPA
- Permet de manipuler les entités JPA et pas les tables d'une base de données
- Supporte des requêtes de type **select, update et delete**

59

JPQL : Exemple

60

Exemple : Recherche avec la clause **where**

```
public List<Etudiant> getAllEtudiantByGp(int idgp) {  
    em.getTransaction().begin();  
    Query q = (Query) em.createQuery  
        ("select e from Etudiant e where e.idgp=:idgp");  
    q.setParameter("idgp", idgp);  
    List<Etudiant> le=q.getResultList();  
    em.getTransaction().commit();  
    return le;  
}
```

60

61

Introduction à Maven

61

Maven

62

- De façon simplifier
 - ▣ Gère et va chercher les dépendances versionnées
 - ▣ Compile le code source puis les tests
 - ▣ Exécute les tests (unitaires et intégrations)
 - ▣ Génère un jar (librarie ou exécutable)

62

POM - Project Object Model

63

Le fichier pom.xml décrit un projet sous forme d'objet eux même décrit en XML

Anatomie

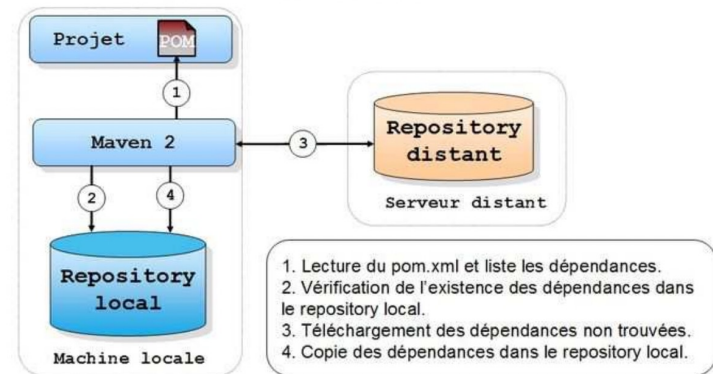
- Information sur le module
`<groupId>`, `<artifactId>`, `<version>`
- Le type de pom (simple module ou multi-module)
`<packaging>` (jar ou pom)
- Metadata
`<name>`, `<description>`, `<url>`, `<licenses>`, `<organization>`, `<developers>`, `<contributors>`
- Propriété
`<properties>`
- Dépendences
`<dependencies>`
- Build (configuration des plugins)
`<build>`

63

POM - Project Object Model

64

Gestion des dépendances par Maven 2



64

POM - Project Object Model

65

- La racine du projet : `<project>`
- La version du modèle de pom (`<modelVersion>`) : 4.0.0 pour Maven 2.x
- L'identifiant du groupe auquel appartient le projet : `<groupId>`
 - Généralement commun à tous les modules d'un projet
- L'identifiant de l'artefact à construire: `<artifactId>`
 - Généralement le nom du module du projet sans espace en minuscules.
- La version de l'artefact à construire `<version>` : Souvent SNAPSHOT sauf lors de la release
- Le type d'artefact à construire: `<packaging>` : pom, jar, war, ear

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>tn.enig</groupId>
  <artifactId>gestionColis</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>gestionColis Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>[]
  <build>
    <finalName>gestionColis</finalName>
  </build>
</project>
```

65

POM - Dépendances

66

```
<dependencies>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.8</version>
</dependency>
</dependencies>
```

66

Coordonnées Maven

67

- Maven utilise un système de 3 valeurs pour définir un module de façon universelle :

<groupid> ; <artifactId> ; <version>

- <groupid>
 - Nom de l'organisation, exp : tn.enig
- <artifactId>
 - Nom du projet
- <version>

67

Création d'un projet Maven

68

New Maven Project

Select an Archetype

Catalog: Internal

Filter:

Group Id	Artifact Id	Version
org.apache.maven.archetypes	maven-archetype-archetype	1.0
org.apache.maven.archetypes	maven-archetype-200-sample	1.0
org.apache.maven.archetypes	maven-archetype-plugin	1.2
org.apache.maven.archetypes	maven-archetype-plugin-sile	1.1
org.apache.maven.archetypes	maven-archetype-profile	1.0.1
org.apache.maven.archetypes	maven-archetype-profile	1.0-alpha-4
org.apache.maven.archetypes	maven-archetype-quickstart	1.1
org.apache.maven.archetypes	maven-archetype-sile	1.1
org.apache.maven.archetypes	maven-archetype-sile-sample	1.1
org.apache.maven.archetypes	maven-archetype-sile-sample	1.2

An archetype which contains a sample Maven Webapp project.

☒ Show the last version of Archetype only ☐ Include snapshot archetypes [Add Archetype...](#)

New Maven project

Specify Archetype parameters

Group Id: tn.enig

Artifact Id: MooFrogger

Version: 0.0.1-SNAPSHOT

Package: tn.enig.MooFrogger

Properties available from archetype:

Name	Value

[Add...](#) [Remove](#)

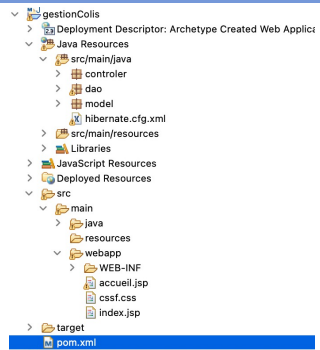
☒ Advanced

68

Structure d'un projet Maven

69

Projet Maven JEE



Descriptions

- **src/main/java** :
 - Contient les sources Java de l'application
- **src/main/resources**
 - Contient les ressources de l'application
- **src/main/webapp**
 - Contient les fichiers de l'application Web
- **src/test/java**
 - Contient les sources Java pour les tests unitaires
- **src/test/resources**
 - Contient les ressources pour les tests unitaires
- **src/site**
 - Contient les fichiers pour le site
- **target**
 - Répertoire de destination de tous les traitements Maven

69

70



QUESTIONS !!!

70