Formation

Hibernate, mapping objet/relationnel

Objectifs de la formation

- Réaliser un mapping Objet/Relationnel avec Hibernate
- Créer, mettre à jour, supprimer et charger des objets persistants
- Effectuer des requêtes avec le langage HQL et l'API Criteria
- Gérer les transactions et les accès concurrents
- Configurer le cache d'Hibernate

Présentations

- Qui êtes-vous ?
- Qu’attendez-vous de cette formation ?
- Avez-vous déjà utilisé l’API JDBC, Intellij, MySQL ?

Plan

1. Introduction
2. Mise en place
3. Le mapping
4. Les associations
5. La manipulation des entités
6. Langage HQL
7. API Criteria
8. Transactions et accès concurrents
9. Le cache Hibernate
10. La génération automatique

Organisation et suivi du cours

Déroulement du cours

Demi journée	Contenu
Mardi matin	Introduction Mise en place
Mardi après-midi	Le mapping Les associations
Mercredi matin	La manipulation des entités
Mercredi après-midi	Langage HQL API Criteria
Jeudi matin	Transactions et accès concurrents
Jeudi après-midi	Le cache Hibernate La génération automatique

Horaires

Matin : 9h-12h30, avec pause à 10h30
Après-midi : 13h30-17h, avec pause à 15h30

Plan

1. *Introduction*
2. Mise en place
3. Le mapping
4. Les associations
5. La manipulation des entités
6. Langage HQL
7. API Criteria
8. Transactions et accès concurrents
9. Le cache Hibernate
10. La génération automatique

Introduction

1. *La persistance des données*
2. Correspondance entre les modèles relationnel et objet
3. Évolution des solutions de persistance avec Java
4. Le mapping O/R (objet/relationnel)
5. Les API standards
6. Qu’est-ce que Hibernate ?

La quasi-totalité des applications de gestion traitent des données dans des volumes plus ou moins importants. Dès que ce volume devient assez important, les données sont **stockées** dans une **base de données**.

Il existe plusieurs types de base de données :

- **Hiérarchique** : historiquement le type le plus ancien, ces bases de données étaient largement utilisées sur les gros systèmes de type mainframe. Les données sont organisées de façon hiérarchique grâce à des pointeurs.
- **Relationnelle** *(RDBMS / SGBDR)* : c'est le modèle le plus répandu actuellement. Les données sont organisées en tables possédant des relations entre elles grâce à des clés primaires et étrangères. Les opérations sur la base sont réalisées grâce à des requêtes SQL. *Exemple : MySQL, PosgreSQL...*
- **Objet** (ODBMS / SGBDO) : *Exemple : db4objects, mongoDb*
- **XML** (XDBMS) : *Exemple : Xindice*

- La base de données relationnelle est historiquement la plus répandue mais aussi une des moins compatibles avec la programmation orientée objet.
- C’est pourquoi ont été développées des normes, techniques et framework pour les intégrer à la POO.

Introduction

1. La persistance des données
2. *Correspondance entre les modèles relationnel et objet*
3. Évolution des solutions de persistance avec Java
4. Le mapping O/R (objet/relationnel)
5. Les API standards
6. Qu’est-ce que Hibernate ?

La correspondance entre les modèles relationnel et objet doit faire face à plusieurs problèmes :

- le modèle objet propose plus de fonctionnalités : héritage, polymorphisme...
- les relations entre les entités des deux modèles sont différentes
- un objet ne possède pas d'identifiant en standard *(hormis son adresse mémoire qui varie d'une exécution à l'autre)*. Dans le modèle relationnel, chaque occurrence devrait posséder un identifiant unique

La persistance des objets en Java présente de surcroît quelques inconvénients supplémentaires :

- de multiples choix dans les solutions et les outils *(standard, commerciaux, open source)*
- de multiples choix dans les API et leurs implémentations
- de nombreuses évolutions dans les API standard et les frameworks open source

Introduction

1. La persistance des données
2. Correspondance entre les modèles relationnel et objet
3. *Évolution des solutions de persistance avec Java*
4. Le mapping O/R (objet/relationnel)
5. Les API standards
6. Qu'est-ce que Hibernate ?

La première approche pour faire une correspondance entre ces deux modèles a été d'utiliser l'**API JDBC** fournie en standard avec le JDK.

Cependant cette approche présente plusieurs inconvénients majeurs, dont :

- nécessite l'écriture de nombreuses lignes de codes, souvent répétitives
- le mapping entre les tables et les objets est un travail de bas niveau

Tous ces facteurs réduisent la productivité mais aussi les possibilités d'évolutions et de maintenance. De plus, une grande partie de ce travail peut être automatisée.

Face à ce constat, différentes solutions sont apparues :

- des frameworks open source : le plus populaire est **Hibernate** qui utilise des POJOs *(Plain Old Java Object)*
- des frameworks commerciaux dont Toplink était le leader avant que sa base passe en open source
- des API Standards : JDO, EJB entity, JPA

Introduction

1. La persistance des données
2. Correspondance entre les modèles relationnel et objet
3. Évolution des solutions de persistance avec Java
4. ***Le mapping O/R (objet/relationnel)***
5. Les API standards
6. Qu’est-ce que Hibernate ?

Le mapping O/R *(objet/relationnel)* consiste à réaliser la correspondance entre le modèle de données relationnel et le modèle objets de la **façon la plus facile possible**.

Un outil de mapping O/R doit cependant proposer un certain nombre de fonctionnalités parmi lesquelles :

- Assurer le mapping des tables avec les classes, des champs avec les attributs, des relations et des cardinalités
- Proposer une interface qui permette de facilement mettre en oeuvre des actions de type CRUD *(Create, Read, Update, Delete)*
- Eventuellement permettre l'héritage des mappings

- Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée
- Supporter différentes formes d'identifiants générés automatiquement par les bases de données *(identity, sequence, ...)*
- Proposer un support des transactions
- Assurer une gestion des accès concurrents *(verrou, versioning, ...)*
- Fournir des fonctionnalités pour améliorer les performances *(cache, lazy loading, ...)*
- Les solutions de mapping sont donc riches en fonctionnalités, ce qui peut rendre leur mise en oeuvre plus ou moins complexe. Cette complexité est cependant différente d'un développement de toute pièce avec JDBC.

Les solutions de mapping O/R permettent de réduire la quantité de code à produire mais impliquent une partie configuration *(généralement sous la forme d'un ou plusieurs fichiers XML ou d'annotations pour les solutions reposant sur Java 5)*.

Depuis quelques années, les principales solutions mettent en oeuvre des POJOs.

Introduction

1. La persistance des données
2. Correspondance entre les modèles relationnel et objet
3. Évolution des solutions de persistance avec Java
4. Le mapping O/R (objet/relationnel)
5. *Les API standards*
6. Qu’est-ce que Hibernate ?

Les différentes évolutions de Java ont apporté plusieurs solutions pour assurer la persistance des données vers une base de données, sous forme d'API *(Application Programming Interface)* :

- JDBC
- EJB
- Java Persistence API

Une API est un ensemble de définitions et de protocoles qui facilite la création et l'intégration de logiciels d'applications.

JDBC

JDBC est l'acronyme de ***Java DataBase Connectivity***. C'est l'API standard pour permettre un accès à une base de données. Son but est de permettre de coder des accès à une base de données en laissant le code le plus indépendant de la base de données utilisée.

C'est une spécification qui définit des interfaces pour se connecter et interagir avec la base de données *(exécution de requêtes ou de procédures stockées, parcours des résultats des requêtes de sélection, ...)*

L'implémentation de ces spécifications est **fournie par des tiers**, et en particulier les fournisseurs de base de données, sous la forme de **driver**.

EJB 2.0

Les EJB *(Enterprise Java Bean)* proposent des beans de type **Entités** pour assurer la persistance des objets.

Les EJB de type Entité peuvent être de deux types :

- **BMP** *(Bean Managed Persistance)* : les développeurs doivent coder l'intégralité des transactions de base de données. Cela inclut la gestion des connexions de base de données, la détermination de la logique de sélection et le choix des éléments de données à stocker.
- **CMP** *(Container Managed Persistance)* : la persistance est assurée par le conteneur d'EJB en fonction du paramétrage fourni. Il faut donc un serveur plus sophistiqué.

Il existe de nombreuses implémentations, puisque chaque serveur d'applications certifié J2EE doit implémenter les EJB, ce qui inclut entre autres JBoss de RedHat, Geronimo d'Apache, GlassFish de Sun/Oracle, Websphere d'IBM, Weblogic de BEA...

Java Persistence API et EJB 3.0

JPA *(Java Persistence API)* est issu des travaux de la JSR-220 concernant la version 3.0 des EJB : elle remplace d'ailleurs les EJB Entités version 2. C'est une synthèse standardisée des meilleurs outils du sujet.

L'API repose sur :

- l'utilisation d'entités persistantes sous la forme de POJOs
- un gestionnaire de persistance qui assure la gestion des entités persistantes
- l'utilisation d'annotations
- la configuration via des fichiers xml
- JPA peut être utilisé avec Java EE *(avec un serveur d'application)* mais aussi avec Java SE *(application simple)*

- JPA est une spécification : il est nécessaire d'utiliser une implémentation pour la mettre en oeuvre.
  - Plusieurs implémentations existent :
    - EclipseLink
    - TopLink
    - **Hibernate**
    - OpenJPA
  - Deux formats sont utilisables
    - Les annotations *(largement utilisées)*
    - Les fichiers de configuration XML
- JPA ne peut être utilisé qu'avec des bases de données relationnelles.
- La version 3.0 des EJB utilise JPA pour la persistance des données.
- La version 3.2 d'Hibernate implémente aussi JPA.

Introduction

1. La persistance des données
2. Correspondance entre les modèles relationnel et objet
3. Évolution des solutions de persistance avec Java
4. Le mapping O/R (objet/relationnel)
5. Les API standards
6. *Qu'est-ce que Hibernate ?*

Hibernate est un **framework** open source gérant la **persistance** des objets en **base de données relationnelle**. Il est l'implémentation JPA la plus répandue.

Il est adaptable en termes d'architecture et peut donc être utilisé aussi bien dans un développement client lourd, que dans un environnement web léger de type Apache Tomcat ou dans un environnement Java EE complet : WebSphere, JBoss Application Server et Oracle WebLogic Server.

Hibernate apporte une solution aux problèmes d'adaptation entre le paradigme objet et les SGBD, en remplaçant les accès à la base de données par des appels à des méthodes objet de haut niveau.

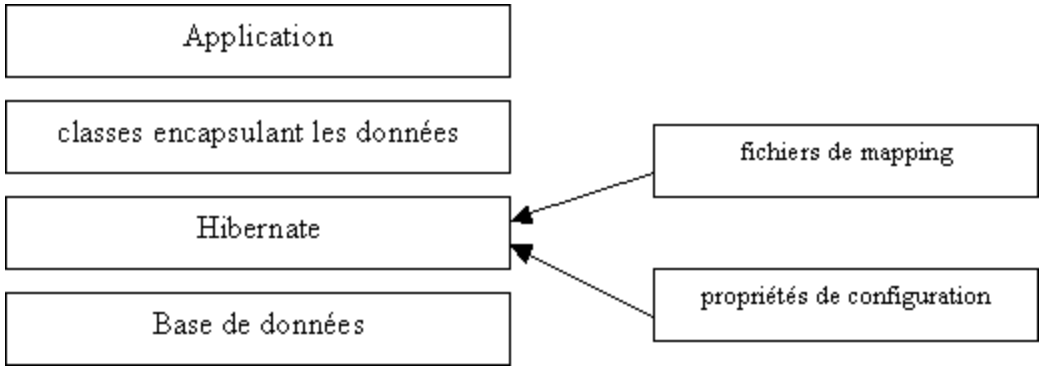
Il s'agit donc d'une **couche d'abstraction** entre la base de données et le programme Java.

Hibernate a besoin de plusieurs éléments pour fonctionner :

- une **classe de type javabean** qui encapsule les données d'une occurrence d'une table
- un **fichier de configuration** qui assure la correspondance entre la classe et la table (mapping).
- Nous travaillerons par des annotations
  des **propriétés de configuration** notamment des informations concernant la connexion à la base de données

Une fois ces éléments correctement définis, il est possible d'utiliser Hibernate dans le code des traitements à réaliser.

L'architecture d'Hibernate est donc la suivante :



Les avantages

- Hibernate est une solution de type **ORM** *(Object Relational Mapping)* qui permet de faciliter le développement de la couche persistance d'une application. En d'autres termes, il permet de représenter une base de données en objets Java et vice versa.
- Il facilite la persistance et la recherche de données dans une base de données, en réalisant lui-même la création des objets et les traitements de remplissage de ceux-ci en accédant à la base de données. La quantité de code ainsi épargnée est très importante, d'autant que ce code est généralement fastidieux et redondant.

- Hibernate est très populaire, notamment à cause de ses bonnes performances et de son ouverture à de nombreuses bases de données : DB2, Oracle, MySQL/MariaDB, PostgreSQL, SQL Server...
- Il propose, en plus de JPA, des solutions spécifiques :
  - Utiliser au maximum l'implémentation JPA
  - Utiliser les solutions spécifiques qu'en cas de besoin

Plan

1. Introduction
2. *Mise en place*
3. Le mapping
4. Les associations
5. La manipulation des entités
6. Langage HQL
7. API Criteria
8. Transactions et accès concurrents
9. Le cache Hibernate
10. La génération automatique

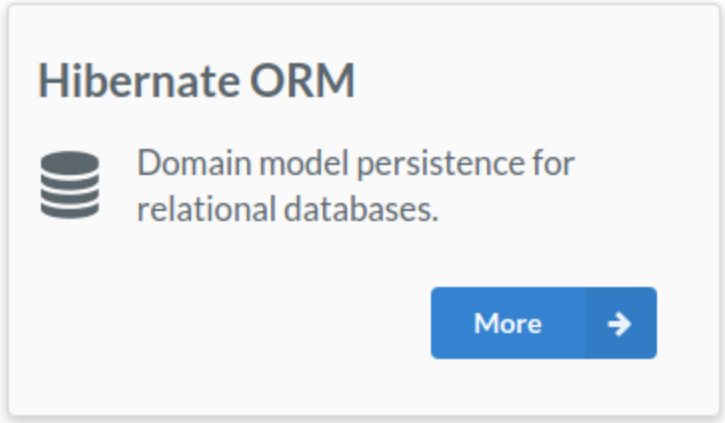
Mise en place

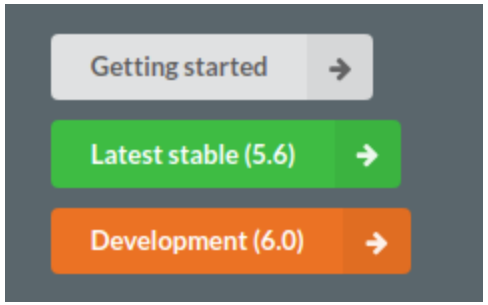
1. *Installation*
2. Configuration
3. Une simple classe persistante
4. Automatisation
5. Simple requête
6. Simple ajout
7. Implémentation pure JPA
8. Travaux pratiques

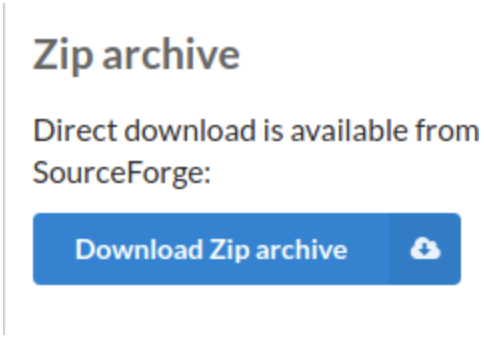
Hibernate peut être installé :

- via le fichier jar de la librairie
- via maven

La première solution consiste à télécharger manuellement la librairie Hibernate ORM *(autrefois appelée **Hibernate Core**)* depuis le site officiel (http://hibernate.org/)







Un fichier zip est alors téléchargé et il conviendra au minimum d'inclure dans le projet les jar du répertoire ***lib/required***

Installation avec Maven

L'installation via Maven est bien plus simple :

Dans le fichier `pom.xml`

```
<dependencies>
        ...
        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-core</artifactId>
                <version>5.4.4.Final</version>
        </dependency>
</dependencies>
```

Mise en place

1. Installation
2. *Configuration*
3. Une simple classe persistante
4. Automatisation
5. Simple requête
6. Simple ajout
7. Implémentation pure JPA
8. Travaux pratiques

Le fournisseur de persistance a besoin d'être configuré afin de pouvoir se connecter au SGBD utilisé.

Cette configuration comprend différentes informations essentielles :

- Nom du serveur de base de données
- Identifiant de connexion
- Mot de passe de connexion
- Nom de la base de données utilisée
- Nom du driver JDBC utilisé

Ce fichier peut contenir aussi des options destinées au fournisseur de persistance.

Pour Hibernate, ce fichier s'appelle `hibernate.cfg.xml`

Fichier de configuration minimal

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
        <session-factory>

                <!-- Propriétés de la connexion -->
                <property name="hibernate.connection.url">
                        jdbc:mysql://localhost:3306/formhibernate?useLegacyDatetimeCode=false&serverTimezone=UTC
                </property>
                <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
                <property name="hibernate.connection.username">formation</property>
                <property name="hibernate.connection.password">formation</property>
                <property name="hibernate.connection.pool_size">10</property>

                <!-- Dialect InnoDB -->
                <property name="dialect">org.hibernate.dialect.MySQL5InnoDBDialect</property>

                <property name="hibernate.show_sql">true</property>



        </session-factory>
</hibernate-configuration>
```

Dans le fichier `pom.xml`, il convient aussi d'ajouter la dépendance du driver JDBC utilisé.

Pour MySQL :

```
<dependencies>
        ...
        <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>8.0.17</version>
        </dependency>
</dependencies>
```

Mise en place

1. Installation
2. Configuration
3. *Une simple classe persistante*
4. Automatisation
5. Simple requête
6. Simple ajout
7. Implémentation pure JPA
8. Travaux pratiques

Modélisons une simple classe persistante JPA permettant de représenter un passager.

Cette classe est une représentation au format Java de la table qui contiendra les passagers.

```java
package formation.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
public class Client {
	@Id
	private int id;

	private String nom;
	private String prenom;

	public Client() { }

	public Client(String nom, String prenom) {
		this.nom = nom;
		this.prenom = prenom;
	}

	// Accesseurs et mutateurs
}
```

Dans le fichier de configuration Hibernate (`hibernate.cfg.xml`), il convient de déclarer cette classe comme persistante :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
        <session-factory>
                ...
                <mapping class="formation.entity.Client" />
        </session-factory>
</hibernate-configuration>
```

Mise en place

1. Installation
2. Configuration
3. Une simple classe persistante
4. *Automatisation*
5. Simple requête
6. Simple ajout
7. Implémentation pure JPA
8. Travaux pratiques

Hibernate peut se charger de l'automatisation du schéma de la base selon les informations fournies par les annotations via le paramètre `hbm2ddl.auto`

```
<property name="hbm2ddl.auto">create-drop</property>
```

Valeur	Résultat
validate	Valide le schéma, mais ne fait aucune modification à la base - *Hibernate uniquement*
update	Met à jour automatiquement le schéma - *Hibernate uniquement*
create	Créé le schéma en détruisant les données déjà existantes
create-drop	Créé le schéma et le supprime lorsque le `sessionFactory` est fermé explicitement
create-only	Créé le schéma uniquement - *Equivalent à* `create` *sur Hibernate*
drop	Détruit le schéma

Mise en place

1. Installation
2. Configuration
3. Une simple classe persistante
4. Automatisation
5. Simple ajout
6. Implémentation pure JPA
7. Travaux pratiques

Tout est prêt pour utiliser Hibernate avec `SessionFactory` *(requête HQL)* !

```java
package formation;

import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.query.Query;
import formation.entity.Client;

public class Hib01 {
	public static void main(String[] args) {
		StandardServiceRegistry registry = new StandardServiceRegistryBuilder().configure().build();
		MetadataSources sources = new MetadataSources(registry);
		Metadata metadata = sources.buildMetadata();
		SessionFactory sessionFactory = metadata.buildSessionFactory();

		Session session = sessionFactory.openSession();

		Query<Client> query = session.createQuery("FROM Client", Client.class);
		List<Client> resultats = query.list();
		for (Client p : resultats) {
			System.out.println(p.getNom() + " " + p.getPrenom());
		}
		session.close();
	}
}
```

Mise en place

1. Installation
2. Configuration
3. Une simple classe persistante
4. Automatisation
5. Simple requête
6. *Simple ajout*
7. Implémentation pure JPA
8. Travaux pratiques

Pour ajouter de la donnée :

```
try (Session session = sessionFactory.openSession()) {
        Transaction tx = session.beginTransaction();
        Client c = new Client("ANGELE", "Kevin");
        session.persist(c);
        tx.commit();

        Query<Client> query = session.createQuery("FROM Client", Client.class);
        List<Client> resultats = query.list();

        for (Client p : resultats) {
                System.out.println(p.getNom() + " " + p.getPrenom());
        }
}
```

Mise en place

1. Installation
2. Configuration
3. Une simple classe persistante
4. Automatisation
5. Simple requête
6. Simple ajout
7. *Implémentation pure JPA*
8. Travaux pratiques

Pour implémenter cela en pur JPA, il convient de modifier le fichier de paramètrage

`hibernate.cfg.xml` ⇒ `META-INF/persistence.xml`

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                version="2.2"
                xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
        <persistence-unit name="form-punit">
                <description>Formation Persistance Unit</description>

                <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
                <exclude-unlisted-classes>false</exclude-unlisted-classes>

                <properties>
                        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
                        <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />

                        <property name="javax.persistence.jdbc.url"
                                        value="jdbc:mysql://localhost:3306/formhibernate?useLegacyDatetimeCode=false&amp;serverTimezone=UTC"   />

                        <property name="javax.persistence.jdbc.user" value="formation" />
                        <property name="javax.persistence.jdbc.password" value="formation" />

                        <property name="hibernate.show_sql" value="true" />
                        <property name="hibernate.hbm2ddl.auto" value="create-drop" />
                </properties>
        </persistence-unit>
</persistence>
```

Le code applicatif avec `EntityManager` *(JPA et requête JPQL)*

```
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
import formation.entity.Client;

public class HibO2 {
	public static void main(String[] args) {
		EntityManagerFactory emf = Persistence.createEntityManagerFactory("form-punit");
		EntityManager em = emf.createEntityManager();

		TypedQuery<Client> query = em.createQuery("select c from Client c", Client.class);
		List<Client> resultats = query.getResultList();

		for (Client p : resultats) {
			System.out.println(p.getNom() + " " + p.getPrenom());
		}

		em.close();
	}
}
```

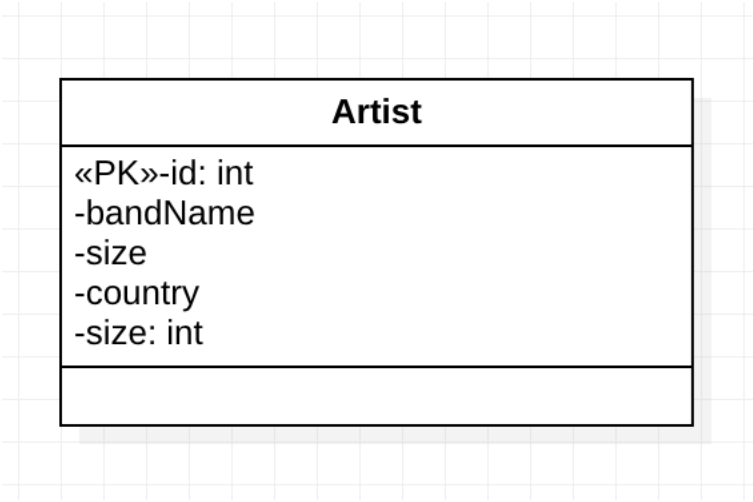
Mise en place

1. Installation
2. Configuration
3. Une simple classe persistante
4. Automatisation
5. Simple requête
6. Simple ajout
7. Implémentation pure JPA
8. *Travaux pratiques*

TP

Créer 2 nouveaux projets *'from scratch'*

- Un projet hibernate
- Un projet JPA



La base de données doit se nommer 'music-db'

Dans les deux projets. On doit pouvoir ajouter puis lister nos Artist. Il faut donc créer la classe dans vos deux projets.

Une fois que les deux projets fonctionnent, changer la valeur de hbm2ddl sur `update`

- Constater la différence

Plan

1. Introduction
2. Mise en place
3. ***Le mapping***
4. Les associations
5. La manipulation des entités
6. Langage HQL
7. API Criteria
8. Transactions et accès concurrents
9. Le cache Hibernate
10. La génération automatique

Le mapping

1. *Présentation*
2. Création d'une entité
3. L'annotation @Table
4. Les colonnes
5. Les types
6. Les clés primaires
7. La stratégie d'accès
8. Embedded / Embeddable
9. Travaux pratiques

Le mapping consiste à faire la liaison entre le modèle de la base de données et les objets Java. Pour faire cela, on utilise les annotations *(auparavant, on utilisait des fichiers XML)*.

Hibernate étant une implémentation de JPA, ses annotations respectent JPA. Cependant, elles enrichissent parfois celles de JPA en ajoutant certaines fonctionnalités.

Pour effectuer le mapping, on va partir d'une classe Java déjà écrite et y ajouter les annotations adéquates.

Lors de l'exécution, Hibernate analysera alors ces classes et y appliquera les informations de mapping :

- Ces informations permettront alors à Hibernate de comprendre de quelle façon est constitué le modèle
- Hibernate aura alors toutes les informations utiles pour faire le lien entre le code Java et la base de données

Le mapping

1. Présentation
2. *Création d'une entité*
3. L'annotation @Table
4. Les colonnes
5. Les types
6. Les clés primaires
7. La stratégie d'accès
8. Embedded / Embeddable
9. Travaux pratiques

Dans la spécification JPA, une entité est une classe dont les **instances sont persistantes**.

Pour spécifier qu'une classe est une entité, on utilise l'annotation `@Entity`.

Cette entité est utilisable en important la classe `javax.persistence.Entity`.

```
import javax.persistence.Entity;
@Entity
public class Client {
```

Lorsqu'une classe est déclarée en tant qu'entité, elle doit obligatoirement posséder un **constructeur sans argument**.

Les champs relatifs à cette entité se retrouvent dans la classe :

```
@Entity
@Table(name="clients")
public class Client {
        @Id
        private int id;

        private String nom;

        @Column(name="name2")
        private String prenom;
}
```

Grâce à l'annotation `@Table`, le mapping est fait vers une table explicitement nommée.

L'annotation `@Id` permet de spécifier la clé primaire *(voir la partie sur les clés primaires)* : il est possible de définir simplement une entité avec uniquement: `@Id` et `@Entity`

Avec `@Column`, le mapping est fait vers une colonne explicitement nommée : si l'annotation `@Column` n'est pas utilisée, alors le mapping est fait vers la colonne qui porte le même nom que l'attribut

Il est possible de ne pas mapper un attribut en utilisant l'annotation `@Transient`

Le paramétrage des constructeurs est important pour l'entité.

Le **constructeur sans paramètre** *(par défaut)* est **obligatoire** si un autre constructeur est défini dans la classe.

```
public Client() {}

public Client(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
}
```

Viennent ensuite les accesseurs *(getters)* et les mutateurs *(setters)* de l'entité :

```
public String getNom() {
    return nom;
}

public void setNom(String nom) {
    this.nom = nom;
}

public String getPrenom() {
    return prenom;
}

public void setPrenom(String prenom) {
    this.prenom = prenom;
}
```

Le mapping

1. Présentation
2. Création d'une entité
3. *L'annotation @Table*
4. Les colonnes
5. Les types
6. Les clés primaires
7. La stratégie d'accès
8. Embedded / Embeddable
9. Travaux pratiques

L'annotation `@Table` est facultative. Elle permet de :

- spécifier des options relatives à la table qui sera utilisée pour les entités
  (par défaut, le nom de la table sera le nom de l'entité)
- spécifier des attributs supplémentaires :
  - `@Table(name="clients)`
  - des index peuvent être spécifiés avec l'attribut `indexes`

```
@Table( name = "clients", indexes = {
                @Index(columnList = "id", name = "client_id_hidx"),
                @Index(columnList = "nom", name = "client_nom_hidx")
})
```

On peut aussi changer le nom de la table en écrivant `@Entity(name="clients")`

- Différences : cette notation impactera également les requêtes écrites en JPQL et HQL

Le mapping

1. Présentation
2. Création d'une entité
3. L'annotation @Table
4. *Les colonnes*
5. Les types
6. Les clés primaires
7. La stratégie d'accès
8. Embedded / Embeddable
9. Travaux pratiques

Grâce à l'annotation `@Column`, il est possible de spécifier plusieurs propriétés concernant la colonne qui représentera l'attribut :

Attribut	Utilité
name	Nom de la colonne *(par défaut le nom de la propriété)*
length	Taille de la colonne pour les **String** *(255 par défaut)*
nullable	Indique si la colonne est **nullable** *(**true** par défaut)*
unique	Indique si la colonne doit avoir une contrainte **UNIQUE** *(**false** par défaut)*
insertable	Indique si le champs est insérable - sera omis des **INSERT** si à ***false*** *(**true** par défaut)*
updatable	Comme **insertable** mais pour les **UPDATE** *(**true** par défaut)*
precision	Nombre de chiffres dans le nombre décimal
scale	Nombre de chiffres après la virgule dans le nombre décimal

Le mapping

1. Présentation
2. Création d'une entité
3. L'annotation @Table
4. Les colonnes
5. *Les types*
6. Les clés primaires
7. La stratégie d'accès
8. Embedded / Embeddable
9. Travaux pratiques

Lors de la création automatique du schéma, Hibernate choisira le type SQL qui correspondra au type Java de l'attribut :

Type de mapping	Type Java	Type SQL
integer	int ou java.lang.Integer	INTEGER
long	long ou java.lang.Long	BIGINT
short	short ou java.lang.Short	SMALLINT
float	float ou java.lang.Float	FLOAT
double	double ou java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte ou java.lang.Byte	TINYINT
boolean	boolean ou java.lang.Boolean	BIT
yes_no	boolean ou java.lang.Boolean	CHAR(1) ('Y' ou 'N')
true_false	boolean ou java.lang.Boolean	CHAR(1) ('T' ou 'F')
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

JPA 2.2, et donc aussi Hibernate, gèrent très bien les nouveaux types introduits par l'API ***Date and Time*** de Java 8

- On retrouve la liste complète sur : https://docs.jboss.org/hibernate/orm/5.0/mappingGuide/en-US/html/ch03.html

  - `YesNoConverter` encodes a boolean value as `'Y'` or `'N'`,
  - `TrueFalseConverter` encodes a boolean value as `'T'` or `'F'`, and
  - `NumericBooleanConverter` encodes the value as an integer, `1` for true, and `0` for false.

```
@Convert(converter = org.hibernate.type.NumericBooleanConverter.class)
private Boolean debug = false;
```

Le mapping

1. Présentation
2. Création d'une entité
3. L'annotation @Table
4. Les colonnes
5. Les types
6. *Les clés primaires*
7. La stratégie d'accès
8. Embedded / Embeddable
9. Travaux pratiques

Chaque entité se doit d'avoir une clé primaire, annotée avec `@Id`.

Habituellement, il s'agit d'un simple champ, même si elle peut être parfois composée *(voir la suite)*

```
import javax.persistence.Id;

@Id
private int id;
```

Par défaut, sans information supplémentaire, c'est au développeur de gérer les valeurs de clé primaire.

Cependant, l'annotation `@GeneratedValue` peut être utilisée pour spécifier la stratégie à utiliser pour la génération de clé primaire

Cette annotation accepte un attribut `strategy` qui spécifie le type de génération de clé primaire :

- `AUTO` : Hibernate choisit automatiquement selon la base de données - *Par défaut*
- `IDENTITY` : La base de données se charge de générer la clé primaire
- `SEQUENCE` : Utilisation d'une séquence
- `TABLE` : Utilisation d'une table séparée

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

Les clés primaires - séquences

Pour utiliser les séquences, il convient d'ajouter l'annotation `@SequenceGenerator`

```
@Id
@SequenceGenerator(name="seqgen1", sequenceName="CLIENT_SEQ")
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="seqgen1")
private int id;
```

- "CLIENT_SEQ" : nom de la séquence
- "seqgen1" : nom du générateur

Les clés primaires - tables

L'annotation `@TableGenerator` est utilisée pour spécifier des valeurs de clé primaire dans une table :

```
@Id
@TableGenerator(name="tablegen1",
                table="ID_TABLE",
                pkColumnName="ID",
                valueColumnName="NEXT_ID")
@GeneratedValue(strategy=GenerationType.TABLE,generator="tabgen1")
private int id;
```

L'annotation `@TableGenerator` permet de spécifier plusieurs paramètres, entre autres :

Valeur	Résultat
table	La tablecontenant les valeurs de clés primaires
pkColumnName	Identifie la colonnede la clé primaire
pkColumnValue	Identifie le nomde la clé primaire dans la colonne pkColumnName
valueColumnName	Identifiela colonne de la valeur de la clé primaire
initialValue	Valeur initiale de la clé primaire
schema	Nomdu schéma contenant la table de clés primaires

```
@TableGenerator(name="tablegen1",
                table="ID_TABLE",
                pkColumnName="ID",
                valueColumnName="NEXT_ID")
```

```
SELECT * FROM formhibernate.id_table;
```

	ID	NEXT_ID
▶	Client	100
*	NULL	NULL

Clés primaires composites

Pour créer une clé primaire composite, il convient de créer une classe représentant la clé primaire :

- Classe publique avec **constructeur par défaut** et **sérialisable**, implémentant `hashCode()` et `equals()`
- Cette classe utilisera l'annotation `@Embeddable`

La classe représentant l'entité se verra attribuer un champ avec une annotation `@EmbeddedId`.

Grâce à tout cela, la clé primaire est alors représentée par une seule propriété au sein de l'objet.

```java
@Embeddable
public class ClientNomComplet implements Serializable {
    String nom;
    String prenom;

    public ClientNomComplet() { }

    // Getters et setters

    @Override()
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof ClientNomComplet)) return false;

        ClientNomComplet cnc = (ClientNomComplet) o;
        if (this.nom.equals(cnc.nom)
                && this.prenom.equals(cnc.prenom)) {
            return true;
        }
        else {
            return false;
        }
    }

    @Override()
    public int hashCode() {
        return this.nom.hashCode() + this.prenom.hashCode();
    }
}
```

```java
@Entity
public class Client {
    @EmbeddedId
    ClientNomComplet cnc;

    @Column
    int age;

    public Client() {  }

    // Getters et setters
}
```

Autre implémentation

```
@Entity
public class Client {
        @EmbeddedId
        ClientNomComplet cnc;

        @Column
        int age;



        static class ClientNomComplet implements Serializable {
                // Même code que précédement
        }
}
```

Une autre façon de créer une clé primaire composite est d'utiliser les annotations `@Id` et `@IdClass` :

```
public class ClientId implements Serializable {
    String nom;
    String prenom;

    public ClientId() { }

    @Override()
    public boolean equals(Object o) {...}

    @Override()
    public int hashCode() {...}
}
```

```
@Entity
@IdClass(ClientId.class)
public class Client{
    @Id
    private String nom;
    @Id
    private String prenom;
    @Column
    int age;

    public Client() { }

    // Getters et setters
}
```

Cette notation a l'avantage d'être plus concise lors de l'accès aux attributs, mais elle perd la distinction des attributs de la clé primaire *(pas d'objet représentant la clé primaire)*.

Le mapping

1. Présentation
2. Création d'une entité
3. L'annotation @Table
4. Les colonnes
5. Les types
6. Les clés primaires
7. *La stratégie d'accès*
8. Embedded / Embeddable
9. Travaux pratiques

Si l'annotation est appliquée à l'accesseur du champ, alors les accesseurs et mutateurs seront utilisés par Hibernate plutôt que l'accès direct au champ :

```
import javax.persistence.Id;

@Id
public int getId() {
    return this.id;
}

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

Cette régle sera alors valable pour tous les attributs de la classe : on parle alors de ***field-based access*** ou de ***property-based access***.

Le mapping

1. Présentation
2. Création d'une entité
3. L'annotation @Table
4. Les colonnes
5. Les types
6. Les clés primaires
7. La stratégie d'accès
8. *Embedded / Embeddable*
9. Travaux pratiques

Il est possible de créer un objet embarqué.

Cet objet n'aura pas d'existence propre en base de données et utilisera la même clé primaire que l'objet contenant.

La classe à inclure doit être ***embeddable***.

Elle sera représentée dans la classe comme ***embedded***.

Cette classe ***embeddable*** doit avoir un constructeur vide.

```java
import javax.persistence.Embeddable;

@Embeddable
public class Adresse {
    ...
}
```

On incorpore Adresse dans la classe Client

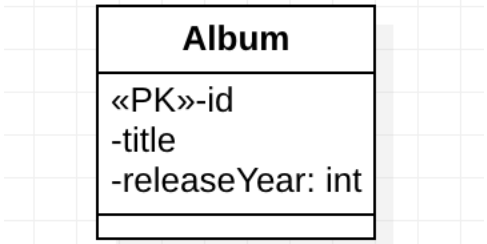
```java
import javax.persistence.Embedded;

@Embedded
public Adresse getAdresse() {
    return adresse;
}
```

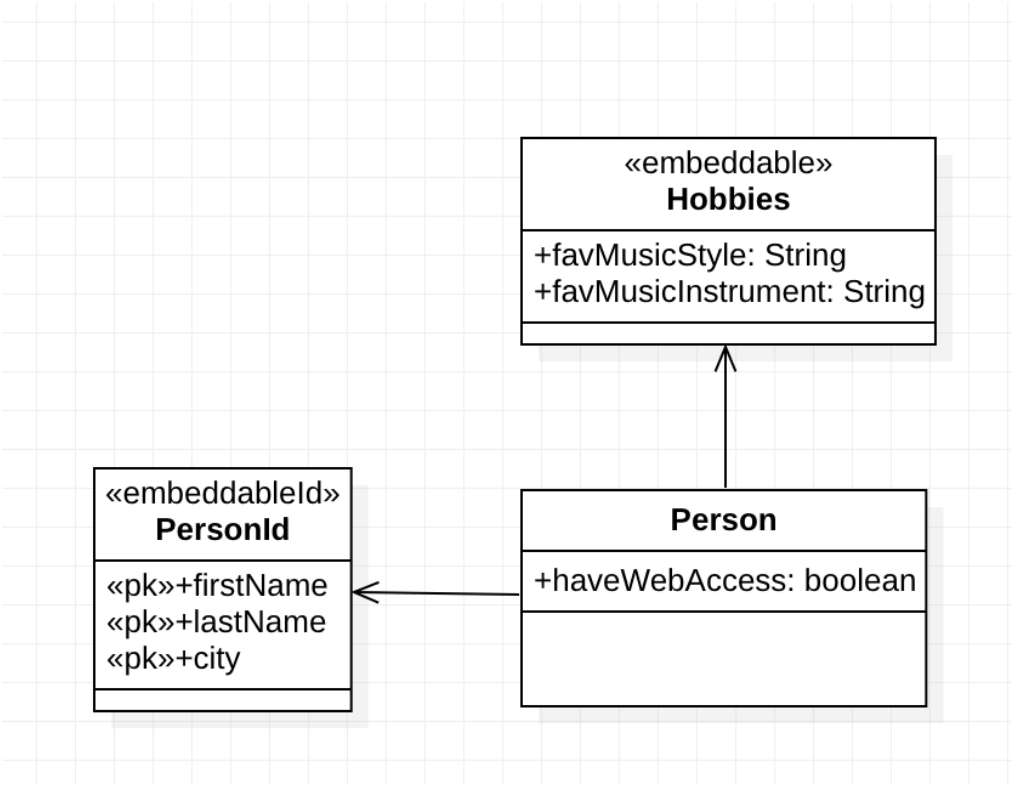
TP

- Dans le projet Hibernate
  - Créer la classe Album:
    - Elle doit avoir un id auto incrémenté.

  - Créer la classe Personne:
    - Lui associer un PersonId en tant que classe d'ID imbriquée.
    - Lui associer une Address en tant que classe imbriquée.

- Tester d'insérer ces deux nouvelles entités dans la base via votre classe Main





Le mapping

1. Présentation
2. Création d'une entité
3. L'annotation @Table
4. Les colonnes
5. Les types
6. Les clés primaires
7. La stratégie d'accès
8. Embedded / Embeddable
9. *Travaux pratiques*

Plan

1. Introduction
2. Mise en place
3. Le mapping
4. ***Les associations***
5. La manipulation des entités
6. Langage HQL
7. API Criteria
8. Transactions et accès concurrents
9. Le cache Hibernate
10. La génération automatique

Les associations

1. *Présentation*
2. Les opérations en cascade
3. Le type de récupération
4. L'association 1:1
5. Les associations 1:N et N:1
6. L'association M:N
7. L'annotation @OrderBy
8. L'héritage
9. Travaux pratiques

Une association est soit :

- Unidirectionnelle
- Bidirectionnelle

JPA propose les types classiques d'associations :

- `1:1` *(one-to-one)*
- `1:N` *(one-to-many)*
- `N:1` *(many-to-one)*
- `M:N` *(many-to-many)*

Pour indiquer une association, on utilise une annotation sur la propriété associée dans la classe.

Les associations

1. Présentation
2. *Les opérations en cascade*
3. Le type de récupération
4. L'association 1:1
5. Les associations 1:N et N:1
6. L'association M:N
7. L'annotation @OrderBy
8. L'héritage
9. Travaux pratiques

Quand une association existe entre deux entités et lorsqu'on applique une opération à une entité, il est courant de vouloir appliquer cette même opération aux entités associées

Par exemple, lorsqu'on supprime une donnée, on aimerait supprimer également les autres données associées.

Hibernate , et plus généralement JPA, permet d'appliquer des opérations en cascade aux entités liées avec l'attribut `cascade` de l'annotation décrivant la relation. Cet attribut prend des membres de l'énumération `CascadeType`

Valeur	Effet appliqué aux entités dépendantes
ALL	Applique toutes les opérations
MERGE	Applique les mises à jour *(UPDATE)*
PERSIST	Applique les ajouts *(INSERT)*
REFRESH	Applique les rafraichissements *(SELECT)*
DETACH	Enlève l'entité du contexte de persistance
REMOVE	Applique les suppressions *(DELETE)*

Par défaut, aucune opération n'est appliquée par cascade

Exemple :

```
@OneToOne(cascade = CascadeType.DELETE)
private Adresse adresse;
```

Les associations

1. Présentation
2. Les opérations en cascade
3. *Le type de récupération*
4. L'association 1:1
5. Les associations 1:N et N:1
6. L'association M:N
7. L'annotation @OrderBy
8. L'héritage
9. Travaux pratiques

Dans une association, l'attribut `fetch` indique quel est le mode de récupération

- `EAGER`
- `LAZY`

En mode `EAGER`, l'association est initialisée lors de la requête. Les objets persistants associés sont donc créés dès la récupération de l'objet ou des objets

En mode `LAZY`, l'association ne sera initialisée que si on y accède après la requête. Cette méthode implique de toujours être dans le contexte de la Session pour accéder aux associations car une requête SQL sera nécessaire

Pour des raisons de performance et de mémoire, on préfèrera le mode `LAZY`.

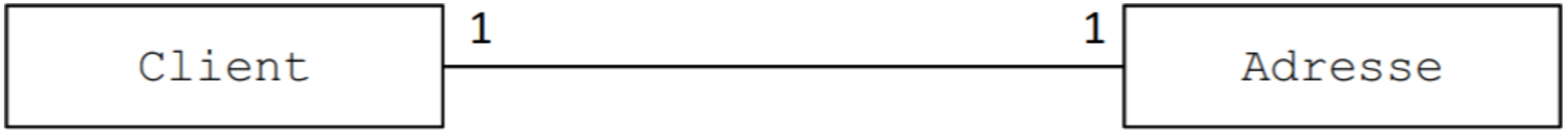
Dans certains cas d'utilisation le mode `EAGER` pourra être utilisé afin d'avoir l'association prête tout de suite :

```
@OneToOne(cascade = CascadeType.DELETE, , fetch=FetchType.EAGER)
private Adresse adresse;
```

Les associations

1. Présentation
2. Les opérations en cascade
3. Le type de récupération
4. *L'association 1:1*
5. Les associations 1:N et N:1
6. L'association M:N
7. L'annotation @OrderBy
8. L'héritage
9. Travaux pratiques

Par exemple, un client est associé à un enregistrement d'adresse.
D'un autre côté, une adresse correspond à un seul client.



On utilise l'annotation `@OneToOne`

Il existe deux types d'association ***one-to-one***

- L'association embarquée *(voir la partie Embedded / Embeddable)*
- Le mapping avec une association *one-to-one*

Pour créer ce type d'association, on utilise l'annotation `@OneToOne`

Une colonne de jointure est alors mise en place dans l'entité :

```
@OneToOne
private Adresse adresse;
```

Dans ce cas, une colonne `adresse_id` sera mise en place dans la table `Client`.
Il est possible de renommer cette colonne avec l'annotation `@JoinColumn` :

```
@OneToOne
@JoinColumn(name="add_id")
private Adresse adresse;
```

L'annotation `@OneToOne` accepte certains attributs :

- `targetEntity` : Indique la classe de l'entité utilisée pour stocker l'association. Si non indiqué, Hibernate utilisera le type de l'attribut de la classe ou le type de retour de l'accesseur
- `cascade` : Permet d'indiquer les types d'opérations en cascade associés - Par défaut ***aucune***.
- `fetch` : Type de récupération (***EAGER*** ou ***LAZY***). ***EAGER*** par défaut
- `optional` : Indique si l'association peut être nulle. ***true*** par défaut.

Pour créer une association bidirectionnelle, il convient de renseigner l'attribut `mappedBy` dans la classe correspondant à l'entité associée :

Client.java

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name="add_id")
private Adresse adresse;
```

Adresse.java

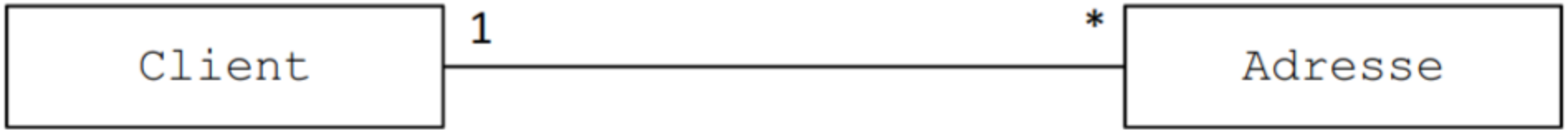
```
@OneToOne(mappedBy="adresse")
private Client client;
```

Les associations

1. Présentation
2. Les opérations en cascade
3. Le type de récupération
4. L'association 1:1
5. *Les associations 1:N et N:1*
6. L'association M:N
7. L'annotation @OrderBy
8. L'héritage
9. Travaux pratiques

Par exemple, un client peut avoir plusieurs adresses.

D'un autre côté, plusieurs adresses sont associées au même client.



On utilise les annotations `@OneToMany` et `@ManyToOne`.

Exemple :

Dans la table **Adresse** sera créée une colonne **client_id**

L'annotation `@JoinColumn` peut être utilisée pour changer le nom de la colonne de jointure.

Adresse.java

```
private Client client;

@ManyToOne
public Client getClient() {
    return this.client;
}
```

Client.java

```
private List<Adresse> adresses;

@OneToMany(cascade=CascadeType.ALL, mappedBy="client")
public List<Adresse> getAdresses(){
    return adresses;
}
```

L'annotation `@ManyToOne` accepte certains attributs :

- `targetEntity` : Indique la classe de l'entité utilisée pour stocker l'association. Si non indiquée, Hibernate utilisera le type de l'attribut de la classe ou du type de retour de l'accesseur
- `cascade` : Permet d'indiquer les types d'opérations en cascade associés. Par défaut ***aucune***.
- `fetch` : Type de récupération (***EAGER*** ou ***LAZY***). ***LAZY*** par défaut
- `optional` : Indique si l'association peut être nulle. ***true*** par défaut

Les associations 1:N

Dans le cas où on ne mappe pas des entités *(par exemple pour les **types basiques** ou pour les **Embeddable**)*, on peut utiliser l'annotation `@ElementCollection`

```
@ElementCollection
public Set<String> getVilles() {
        return this.villes;
}
```

La table générée n'aura pas de clé primaire mais seulement une clé étrangère vers l'entité propriétaire de l'association.

Les associations

1. Présentation
2. Les opérations en cascade
3. Le type de récupération
4. L'association 1:1
5. Les associations 1:N et N:1
6. *L'association M:N*
7. L'annotation @OrderBy
8. L'héritage
9. Travaux pratiques

L'association M:N est spécifiée grâce à l'annotation `@ManyToMany`

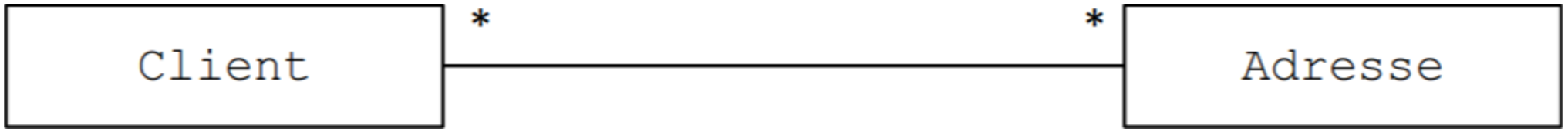
Afin de représenter ce genre d'association, une table de jointure est nécessaire.

- Par défaut, elle aura pour nom `<M>_<N>`.
- Par défaut les colonnes auront pour nom la table référencée suivie de `id`.

L'annotation `@ManyToMany` accepte les attributs suivants :

- `mappedBy` : champ qui détient la relation *(entité maître)*. Utile en cas d'association bidirectionnelle
- `targetEntity` : classe qui est la cible de l'association. Par défaut le type de la déclaration
- `cascade` : opérations en cascade à mener. Aucune par défaut
- `fetch` : mode de récupération (***LAZY*** ou ***EAGER***). ***LAZY*** par défaut

Exemple : un client a plusieurs adresses mais une même adresse est associée aussi à plusieurs clients



Client.java

```
private List<Client> clients;

@ManyToMany(mappedBy = "adresses")
public List<Client> getClients() {
    return clients;
}
```

Adresse.java

```
private List<Adresse> adresses;

@ManyToMany(cascade = CascadeType.ALL)
public List<Adresse> getAdresses() {
    return adresses;
}
```

Une table `client_adresse` est alors créée pour stocker les associations.

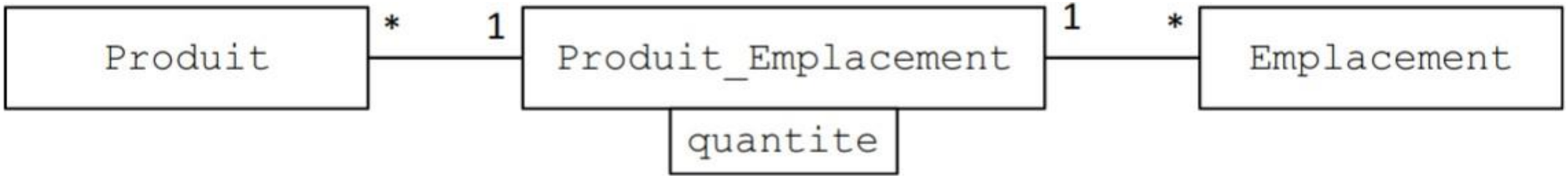
On peut utiliser l'annotation `@JoinTable` pour donner les caractéristiques de la table de jointure :

Client.java

```
private List<Adresse> adresses;

@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(
    name="Adresses_clients",
    joinColumns={@JoinColumn(name = "client_ident")},
    inverseJoinColumns={@JoinColumn(name = "adresse_ident")}
)
public List<Adresse> getAdresses() {
    return adresses;
}
```

Avec Hibernate , il n'est pas possible de créer une association `@ManyToMany` qui contiendrait des champs supplémentaires.

Pour cela, il convient de créer de toute pièce l'entité de liaison et d'utiliser des associations `@OneToMany` et `@ManyToOne`



Les associations

1. Présentation
2. Les opérations en cascade
3. Le type de récupération
4. L'association 1:1
5. Les associations 1:N et N:1
6. L'association M:N
7. *L'annotation @OrderBy*
8. L'héritage
9. Travaux pratiques

Utiliser un type respectant l'ordre *(par exemple List)* permettra d'avoir un ordre lors de la récupération des associations, et donc de ne plus utiliser les bags. On utilise pour cela l'annotation `@OrderBy`.

```
private List<Adresse> adresses;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "client")
@OrderBy("codePostal ASC")
public List<Adresse> getAdresses() {
        return adresses;
}
```

L'ordre est ASC *(croissant)* ou DESC *(décroissant)*. Par défaut ***ASC***

L'annotation `@OrderColumn` permet de spécifier une colonne dont la valeur servira de valeur de tri.

Les associations

1. Présentation
2. Les opérations en cascade
3. Le type de récupération
4. L'association 1:1
5. Les associations 1:N et N:1
6. L'association M:N
7. L'annotation @OrderBy
8. *L'héritage*
9. Travaux pratiques

Pour faire du mapping d'héritage, Hibernate propose 3 stratégies :

- *Single Table* (`SINGLE_TABLE`) : une seule table utilisée pour représenter toute la hiérarchie. Il y aura donc des éléments `NULL` dans la table
- *Joined* (`JOINED`) : Une table pour chaque sous-classe sans réplication des champs communs *(en y incluant les interfaces et les classes abstraites)*
- *Table-per-class* (`TABLE_PER_CLASS`) : Une table pour chaque classe concrète implémentée

Pour spécifier le type de mapping d'héritage, on utilise l'annotation `@Inheritance` en spécifiant l'attribut `strategy` *(par défaut* `SINGLE_TABLE` *)*.

L'héritage Single Table

Une seule et unique table sera utilisée pour l'héritage. Elle portera par défaut le nom de la classe mère.

Produit.java

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Produit {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String libelle;
    private float prix;
}
```

Chips.java

```
@Entity
public class Chips extends Produit {
        private String saveur;
```

Tomate.java

```
@Entity
public class Tomate extends Produit {
        private String saveur;
```

Une seule et unique table `Produit` est alors créée qui contient tous les champs des classes filles :

Column Name	Datatype
DTYPE	VARCHAR(31)
id	INT(11)
libelle	VARCHAR(255)
prix	FLOAT
saveur	VARCHAR(255)
calibre	INT(11)

Un champ `DTYPE` est créé. Ce champs contient, par défaut, sous forme de chaîne de caractères, le type de l'entité associé à l'enregistrement :

	DTYPE	id	libelle	prix	saveur	calibre
▶	Tomate	1	Tomate ronde	1.5	NULL	1

Il est possible de modifier ce comportement avec l'annotation `@DiscriminatorColumn` associée aux attributs suivants :

Attribut	Utilité
name	Le nom de la colonne du discriminant - `DTYPE` par défaut
discriminatorType	Le type de discriminant (Enumération `DiscriminatorType` - `STRING`, `CHAR` ou `INTEGER`) - `STRING` par défaut
length	Longueur pour lediscriminant de type `STRING-31` par défaut

Par défaut, Hibernate donnera une valeur de discriminant pour chaque entité *(le nom de la classe dans un discriminant de type* `STRING`*)*. Il est possible de spécifier une valeur précise avec l'annotation `@DiscriminatorValue`

Exemple de spécification de discriminant :

Produit.java

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
        name = "DISCRIMINANT",
        discriminatorType = INTEGER
)
@DiscriminatorValue("1")
public class Produit {
```

Chips.java

```
@Entity
@DiscriminatorValue("2")
public class Chips extends Produit {
```

Tomate.java

```
@Entity
@DiscriminatorValue("3")
public class Tomate extends Produit {
```

L'héritage Joined

Avec cette stratégie, une table sera créée pour chaque classe de la hiérarchie des classes, sans réplication des champs communs :

Produit.java

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Produit {
```

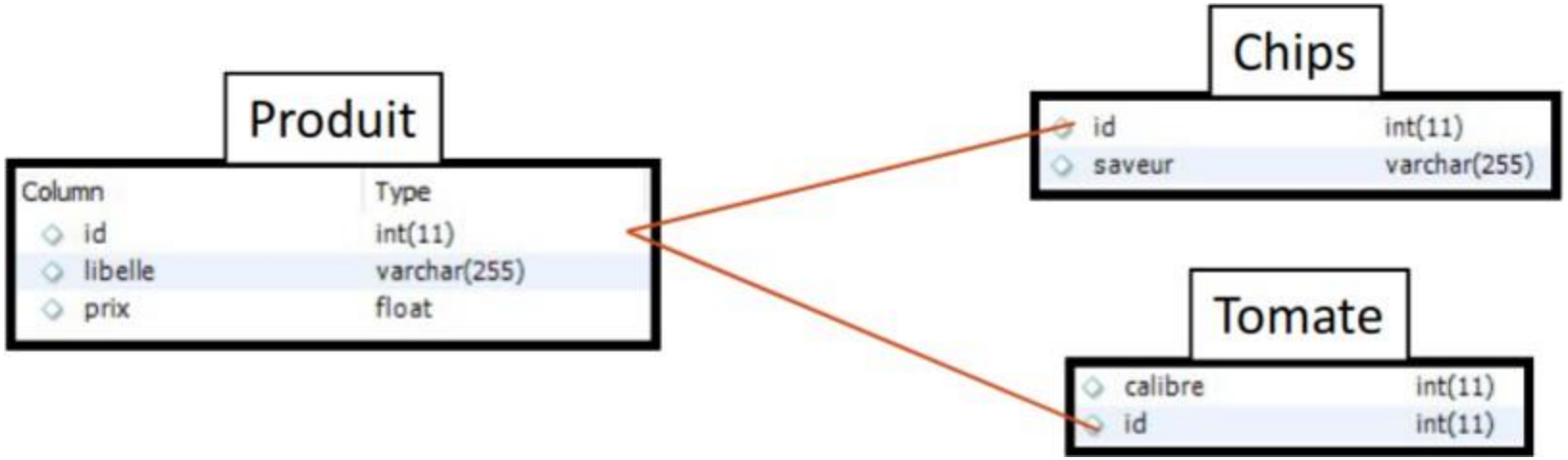
Chips.java

```
@Entity
public class Chips extends Produit {
```

Tomate.java

```
@Entity
public class Tomate extends Produit {
```

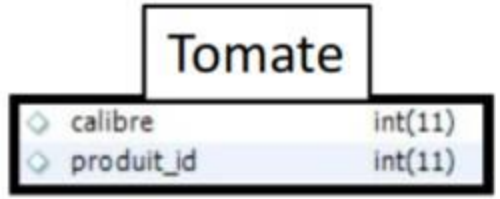
Par défaut, dans les "tables filles", Hibernate créera une colonne `id` qui aura la même valeur que l'enregistrement correspondant de la table mère.



Il est possible de modifier le nom de la colonne de jointure dans les classes fille *(`id` par défaut)* avec l'annotation `@PrimaryKeyJoinColumn`

Tomate.java

```
@Entity
@PrimaryKeyJoinColumn(name = "produit_id")
public class Chips extends Produit {
```



L'héritage table-per-class

Avec cette stratégie, une table sera créée pour chaque classe de la hiérarchie des classes sans réplication des champs communs :

- La génération des valeurs automatiques de clés primaires devra être faite avec la stratégie `TABLE - @GeneratedValue(strategy = GenerationType.TABLE)`
- Une table ne sera pas créée si une classe est abstraite

Produit.java

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Produit {

```

Chips.java

```
@Entity
public class Chips extends Produit {
```

Tomate.java

```
@Entity
public class Tomate extends Produit {
```

L'héritage - Synthèse

La stratégie `JOINED` est celle qui colle au plus près de la notion d'héritage *(besoin de jointures en lecture et plusieurs écritures pour un seul objet)*.

La stratégie `SINGLE_TABLE` est intéressante car elle ne manipule qu'une seule table *(grand nombre de colonnes et des valeurs à `NULL`)*.

La stratégie `TABLE_PER_CLASS` se détache complètement de la notion d'héritage :

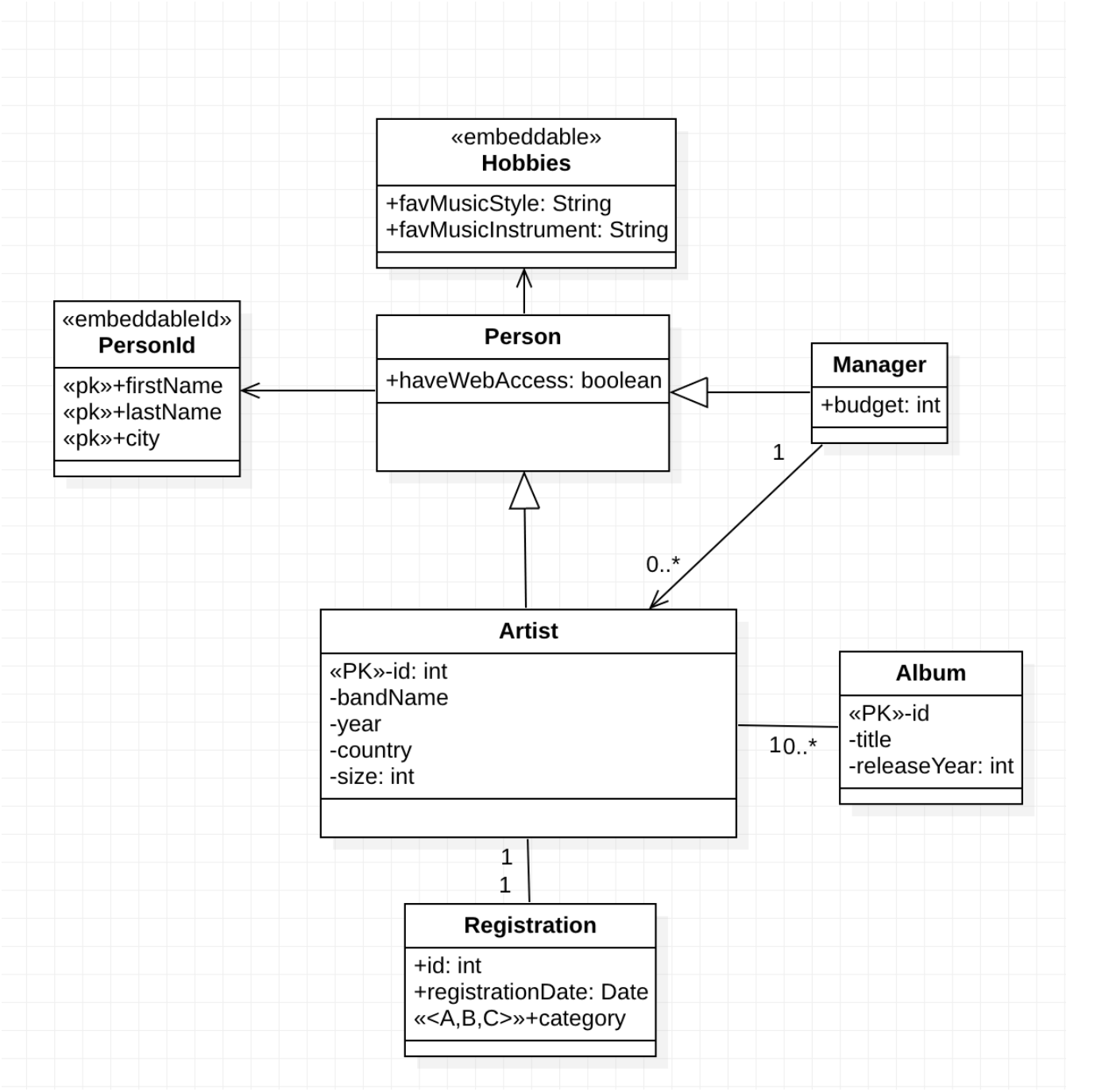
- Beaucoup de données dupliquées
- Une seule requête se traduira par l'exécution de requêtes sur plusieurs tables

Les associations

1. Présentation
2. Les opérations en cascade
3. Le type de récupération
4. L'association 1:1
5. Les associations 1:N et N:1
6. L'association M:N
7. Le conteneur
8. Les bags
9. L'annotation @OrderBy
10. L'héritage
11. *Travaux pratiques*

TP

- Dans le projet Hibernate
  Gérer les relations.
  Gérer l'héritage.



Plan

1. Introduction
2. Mise en place
3. Le mapping
4. Les associations
5. *La manipulation des entités*
6. Langage HQL
7. API Criteria
8. Transactions et accès concurrents
9. Le cache Hibernate
10. La génération automatique

Manipulation des entités

1. *Les entités*
2. Le paramétrage
3. La connexion
4. Les états d'une instance
5. L'enregistrement
6. Le chargement
7. Le merge
8. Le refresh
9. La mise à jour
10. La suppression
11. Travaux pratiques

Une entité doit respecter plusieurs règles :

- Avoir un attribut permettant de représenter la clé primaire (***@Id***)
- Avoir un constructeur sans paramètre *(ou le constructeur par défaut)*
- Ne pas être ***final***
- Ne pas être une classe interne
- Aucun champ persistant ne doit être ***final***
- Aucune méthode ne doit être ***final***
- Doit implémenter ***Serializable*** si elle est passée par valeur en paramètre d'une méthode comme un objet distant

Manipulation des entités

1. Les entités
2. *Le paramétrage*
3. La connexion
4. Les états d'une instance
5. L'enregistrement
6. Le chargement
7. Le merge
8. Le refresh
9. La mise à jour
10. La suppression
11. Travaux pratiques

En pur JPA, le paramétrage de la base de données s'effectue dans le fichier `META-INF/persistence.xml`.
Cela implique alors d'utiliser `EntityManagerFactory` et `EntityManager` pour manipuler les entités.

Hibernate propose son propre paramétrage via le fichier `hibernate.cfg.xml` à placer à la racine du projet. On utilise alors `org.hibernate.SessionFactory` et `org.hibernate.Session` pour manipuler les entités.

En pratique, quand on utilise `EntityManager`, Hibernate utilise dans les coulisses `Session` :

- Il est possible de récupérer l'objet ***Session*** associé à un ***EntityManager***.

```
Session session = entityManager.unwrap(Session.class);
```

- Pour avoir un objet `EntityManager` lié à un objet ***Session***.

```
EntityManager em = session.getEntityManagerFactory().createEntityManager()
```

Manipulation des entités

1. Les entités
2. Le paramétrage
3. *La connexion*
4. Les états d'une instance
5. L'enregistrement
6. Le chargement
7. Le merge
8. Le refresh
9. La mise à jour
10. La suppression
11. Travaux pratiques

Pour obtenir une session, il convient d'initialiser Hibernate avec le fichier de configuration puis de lui demander un objet **Session**.

```
StandardServiceRegistry registry = new StandardServiceRegistryBuilder().configure().build();

MetadataSources sources = new MetadataSources(registry);
Metadata metadata = sources.buildMetadata();
SessionFactory sessionFactory = metadata.buildSessionFactory();
```

Par défaut, c'est le fichier `hibernate.cfg.xml` qui est chargé. Il reste possible de donner un autre nom de fichier de configuration :

```
new StandardServiceRegistryBuilder().configure("ma-config.xml")
```

Ou encore de modifier la configuration dans le code :

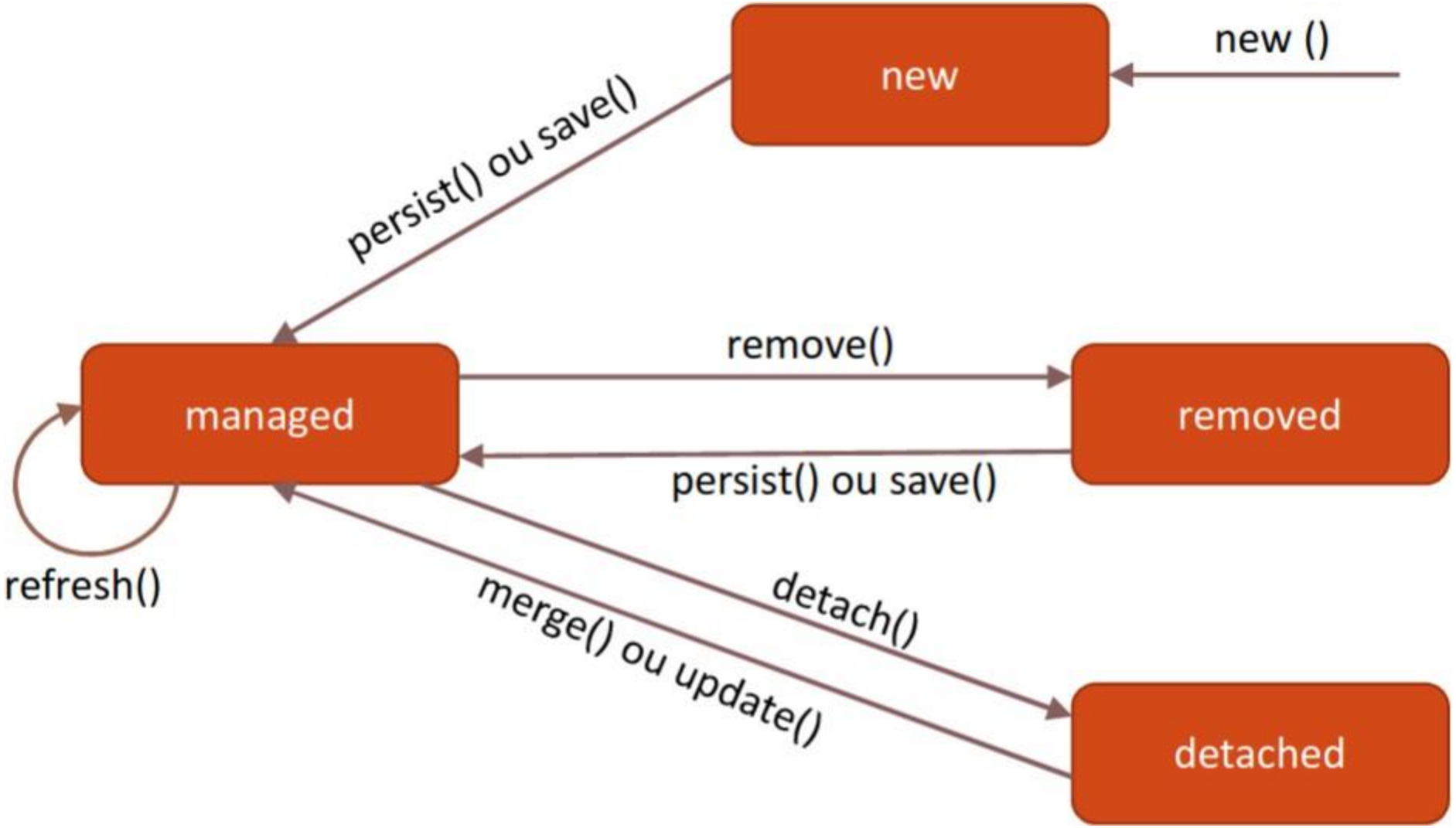
```
StandardServiceRegistryBuilder registryBuilder = new StandardServiceRegistryBuilder().configure();
registryBuilder.applySetting("hibernate.show_sql", "false");
StandardServiceRegistry registry = registryBuilder.build();
```

Manipulation des entités

1. Les entités
2. Le paramétrage
3. La connexion
4. *Les états d'une instance*
5. L'enregistrement
6. Le chargement
7. Le merge
8. Le refresh
9. La mise à jour
10. La suppression
11. Travaux pratiques

Une entité peut avoir 4 états :

- `new` ou `transient` *(nouvelle)* : l'instance existe en mémoire mais elle n'est pas associée à la base de données. Elle n'est pas associée à un contexte de persistance
- `managed` ou `persistent` *(gérée)* : l'instance est associée à la base de données. Tout changement sera répercuté.
- `detached` *(détachée)* : l'instance existe en base de données mais elle n'est plus associée au contexte de persistance
- `removed` *(supprimée)* : l'instance existe dans la base de données mais elle doit être supprimée de la base de donnée


new
new ()
persist() ou save()
managed
remove()
removed
persist() ou save()
refresh()
merge() ou update()
detach()
detached

Manipulation des entités

1. Les entités
2. Le paramétrage
3. La connexion
4. Les états d'une instance
5. *L'enregistrement*
6. Le chargement
7. Le merge
8. Le refresh
9. La mise à jour
10. La suppression
11. Travaux pratiques

Le fait de créer un nouvel objet avec une classe contenant du mapping ne fait pas automatiquement persister l'objet en base de données.

Tant que l'objet n'est pas associé à une session Hibernate valide, il se comporte comme tout autre objet Java.

Pour enregistrer un objet en base, on utilise, depuis ***Session***, la méthode `save` ou la méthode `persist`

```
Transaction tx = session.beginTransaction();
Client c = new Client();
c.setNom("ANGELE");
c.setPrenom("Kévin");
session.persist(c);
tx.commit();
```

La génération de l'identifiant est faite immédiatement avec la méthode ***save***. Ce comportement n'est pas garanti avec ***persist***.

Manipulation des entités

1. Les entités
2. Le paramétrage
3. La connexion
4. Les états d'une instance
5. L'enregistrement
6. *Le chargement*
7. Le merge
8. Le refresh
9. La mise à jour
10. La suppression
11. Travaux pratiques

La méthode `load` permet de charger une entité à partir de son identifiant :

```
public <T> T load(Class<T> theClass, Serializable id)
public Object load(String entityName, Serializable id)
public void load(Object object, Serializable id)
```

```
Client client = session.load(Client.class, 1);
```

Avec cette méthode, si l'identifiant demandé n'est pas trouvé en base, une exception `org.hibernate.ObjectNotFoundException` est lancée.

La méthode `get` agit de la même façon mais retournera `null` si l'enregistrement n'est pas trouvé en base.

```
Client client = session.get(Client.class, 1);
```

Manipulation des entités

1. Les entités
2. Le paramétrage
3. La connexion
4. Les états d'une instance
5. L'enregistrement
6. Le chargement
7. *Le merge*
8. Le refresh
9. La mise à jour
10. La suppression
11. Travaux pratiques

Le ***merge*** est utilisé lorsqu'on veut rendre une entité détachée de nouveau persistante, en mettant à jour la base de données avec les modifications apportées à l'entité :

```
Client c1;
try(Session session = sessionFactory.openSession()) {
    Transaction tx = session.beginTransaction();
    c1 = new Client();
    c1.setNom("ANGELE");
    c1.setPrenom("Kévin");
    c1.setAge(38);
    session.persist(c1);
    tx.commit();
}

c1.setAge(51);

try(Session session = sessionFactory.openSession()) {
    Transaction tx = session.beginTransaction();
    Client cm = (Client) session.merge(c1); //  ==========>
    tx.commit();
}
```

La méthode `merge` retourne l'objet chargé dans le contexte de persistance, pas l'objet donné en paramètre.

Manipulation des entités

1. Les entités
2. Le paramétrage
3. La connexion
4. Les états d'une instance
5. L'enregistrement
6. Le chargement
7. Le merge
8. *Le refresh*
9. La mise à jour
10. La suppression
11. Travaux pratiques

La méthode `refresh` permet de mettre à jour les propriétés d'un objet persistant selon les valeurs contenues dans la base de données :

- Ce n'est pas une méthode qu'il sera nécessaire d'appeler car Hibernate gère très bien le changement
- Elle pourra être utile en cas de modification externe à l'application ou en cas d'utilisation directe du SQL

```
Client c1;

try(Session session = sessionFactory.openSession()) {
    Transaction tx = session.beginTransaction();
    c1 = new Client(); c1.setNom("ANGELE"); c1.setPrenom("Kévin"); c1.setAge(38);
    session.persist(c1);
    tx.commit();
}

c1.setAge(51);

try(Session session = sessionFactory.openSession()) {
    session.refresh(c1);
}
```

Manipulation des entités

1. Les entités
2. Le paramétrage
3. La connexion
4. Les états d'une instance
5. L'enregistrement
6. Le chargement
7. Le merge
8. Le refresh
9. *La mise à jour*
10. La suppression
11. Travaux pratiques

Hibernate mettra à jour automatiquement la base de données lorsque des modifications sont apportées à des objets persistants :

```
Client c1;
try(Session session = sessionFactory.openSession()) {
        Transaction tx = session.beginTransaction();

        Client c1 = new Client();
        c1.setNom("ANGELE");
        c1.setPrenom("Kévin");
        c1.setAge(38);

        session.persist(c1);

        c1.setAge(90); // Un "UPDATE SQL" est automatiquement fait ici car l'entité c1 est persistante

        tx.commit();
}
```

La méthode `flush` permet de purger la session en synchronisant la mémoire de la ***Session*** avec la base de données *(écriture des données en base)*.

Généralement, Hibernate fait ce flush pour nous lorsqu'une transaction est ***commitée*** mais cette méthode peut être utile, par exemple, pour contrôler la taille de la session en cas d'opération en masse.

Bien entendu, faire un ***flush*** de la session n'engendre pas un ***commit*** de la transaction. Par contre, l'appel à la méthode ***commit*** engendre un ***flush***.

```
session.flush();
```

Une session est dite `dirty` si des modifications n'ont pas été *flushées*

```
session.isDirty(); //retourne un booléen
```

Il est possible de modifier la façon dont Hibernate gère le *flush* avec la méthode `setHibernate FlushMode(FlushMode flushMode)`

Manipulation des entités

1. Les entités
2. Le paramétrage
3. La connexion
4. Les états d'une instance
5. L'enregistrement
6. Le chargement
7. Le merge
8. Le refresh
9. La mise à jour
10. *La suppression*
11. Travaux pratiques

Pour supprimer une entité de la base de données, on utilise la méthode `delete` en donnant en paramètre l'objet persistant à supprimer.

Le paramètre peut aussi être un objet non persistant qui a pour identifiant l'identifiant de l'objet à supprimer de la base de données :

```
session.delete(client);
```

Dans le cas d'une entité avec des associations, Hibernate peut être configuré pour permettre la suppression en cascade.

L'attribut cascade de l'annotation définissant la relation est utilisé pour cela *(paramétrage en* `ALL` *ou* `DELETE`*)*.

On peut faire des suppressions multiples avec une requête *(voir plus tard)*

```
session.createQuery("delete from Client").executeUpdate();
```

Manipulation des entités

1. Les entités
2. Le paramétrage
3. La connexion
4. Les états d'une instance
5. L'enregistrement
6. Le chargement
7. Le merge
8. Le refresh
9. La mise à jour
10. La suppression
11. *Travaux pratiques*

TP

- Créer les DAOs associe a votre projet gérant les CRUD

Plan

1. Introduction
2. Mise en place
3. Le mapping
4. Les associations
5. La manipulation des entités
6. ***Langage HQL***
7. API Criteria
8. Transactions et accès concurrents
9. Le cache Hibernate
10. La génération automatique

Langage HQL

1. *Présentation*
2. Sélection
3. Mise à jour
4. Suppression
5. Les requêtes nommées
6. La projection
7. La pagination
8. Les associations
9. Travaux pratiques

HQL *(Hibernate Query Language)* permet d'élaborer des requêtes pour extraire de l'information ou pour mettre à jour la base de données.

Il s'inspire de SQL mais au lieu d'opérer sur de tables et des colonnes, il travaille sur les objets et sur leurs propriétés

Il s'agit d'un super-ensemble de JPQL *(Java Persistence Query Language)*

- Une requête JPQL est une requête HQL valide
- Une requête HQL n'est pas forcément une requête JPQL valide

Langage HQL

1. Présentation
2. *Sélection*
3. Mise à jour
4. Suppression
5. Les requêtes nommées
6. La projection
7. La pagination
8. Les associations
9. Travaux pratiques

Pour sélectionner de l'information *(projection)*, on utilise le mot-clé `SELECT` :

```
[SELECT [DISTINCT] property [, ...]]
FROM path [[AS] alias] [, ...] [FETCH ALL PROPERTIES]
WHERE logicalExpression
GROUP BY property [, ...]
HAVING logicalExpression
ORDER BY property [ASC | DESC] [, ...]
```

Il convient donc de créer un objet `Query` contenant la requête voulue, de donner une valeur aux paramètres de la requête et d'exécuter la méthode `list()` pour obtenir une liste des résultats.

La clause `FETCH ALL PROPERTIES` permet de forcer Hibernate à charger les associations en mode `LAZY`.

L'utilisation de l'API Criteria sera préférée pour faire des sélections…

Exemple de sélection :

```
// Création de la requête
Query<Client> query = session.createQuery("from Client where age>:age", Client.class);

// Réglage des paramètres
query.setParameter("age", 35);

// Récupération des résultats
List<Client> clients = query.list();
```

Il est possible de récupérer les résultats sous la forme d'un Stream

```
Stream<Client> clients = query.stream();

clients.map(c -> c.getNom() + " " + c.getAge())
                        .forEach(s -> System.out.println(s));
```

Langage HQL

1. Présentation
2. Sélection
3. *Mise à jour*
4. Suppression
5. Les requêtes nommées
6. La projection
7. La pagination
8. Les associations
9. Travaux pratiques

Pour la mise à jour, la requête ressemble très fortement à du SQL.

On utilise le mot clé `UPDATE` en indiquant :

- le type d'entités à mettre à jour
- les nouvelles valeurs : `SET`
- les entités ciblées par la modification : `WHERE`

En pratique, on crée un objet `Query` grâce à une chaîne de caractères contenant des paramètres repérables avec le caractère `:`

On donne ensuite une valeur à chaque paramètre avec la méthode `setParameter` de l'objet `Query`.

Enfin, on exécute la requête avec la méthode `executeUpdate()` de l'objet `Query` : la méthode retournera alors le nombre d'enregistrements impactés par la modification.

Exemple de mise à jour :

```
// Création de la requête
Query query = session.createQuery("update Client set age = :nage where nom = :nom");

// Réglage des paramètres
query.setParameter("nage", 60);
query.setParameter("nom", "ANGELE");

// Exécution de la requête
int modifications = query.executeUpdate();
```

Langage HQL

1. Présentation
2. Sélection
3. Mise à jour
4. *Suppression*
5. Les requêtes nommées
6. La projection
7. La pagination
8. Les associations
9. Travaux pratiques

La suppression va fonctionner sur le même principe que la mise à jour.

On utilise le mot clé `DELETE` en indiquant :

- le type d'entités à supprimer
- les entités ciblées par la suppression dans la clause `WHERE`

Là encore la méthode `executeUpdate()` retournera le nombre d'entités supprimées.

Exemple :

```
// Création de la requête
Query query = session.createQuery("delete from Client where nom = :nom");

// Réglage des paramètres
query.setParameter("nom", "ANGELE");

// Exécution de la requête
int suppressions = query.executeUpdate();
```

Langage HQL

1. Présentation
2. Sélection
3. Mise à jour
4. Suppression
5. *Les requêtes nommées*
6. La projection
7. La pagination
8. Les associations
9. Travaux pratiques

Grâce aux annotation `@NamedQueries` et `@NamedQuery`, il est possible de mettre des requêtes dans les entités pour les appeler plus tard.

Client.java

```
@Entity
@NamedQueries({  @NamedQuery(
                        name   = "client.findByAge",
                        query  = "from Client c where c.age = :age"
)})
public class Client {
```

Utilisation :

```
Query<Client> query = session.getNamedQuery("client.findByAge");
query.setParameter("age", 53);

Stream<Client> clients = query.stream();
```

Langage HQL

1. Présentation
2. Sélection
3. Mise à jour
4. Suppression
5. Les requêtes nommées
6. *La projection*
7. La pagination
8. Les associations
9. Travaux pratiques

Lorsqu'on utilise la projection, le type retourné par les requêtes n'est pas le type des entités mais soit le type du champs retourné, soit `Object[]` :

```
Query<String> query = session.createQuery("select nom from Client where age > :age");
query.setParameter("age", 35);
List<String> clients = query.list();    // On récupère ici une liste de String
```

On récupère ici une liste de `Object[]`. On accède au nom avec `[0]` et au prénom avec `[1]` :

```
Query<String> query = session.createQuery("select nom, prenom from Client where age>:age");
query.setParameter("age", 35);
List<Object[]> clients = query.list();
```

Langage HQL

1. Présentation
2. Sélection
3. Mise à jour
4. Suppression
5. Les requêtes nommées
6. La projection
7. *La pagination*
8. Les associations
9. Travaux pratiques

L'objet `Query` propose 2 méthodes pour gérer la pagination :

- `setFirstResult(n)` : première ligne du jeu de résultat
- `setMaxResults(n)` : nombre de résultats à récupérer

```
Query<Client> query = session.createQuery("from Client where age > :age", Client.class);

query.setParameter("age", 35);

query.setMaxResults(10);
query.setFirstResult(100);
```

Langage HQL

1. Présentation
2. Sélection
3. Mise à jour
4. Suppression
5. Les requêtes nommées
6. La projection
7. La pagination
8. *Les associations*
9. Travaux pratiques

En HQL, l'utilisation des associations reste intuitive

```
Query<Client> query = session.createQuery(
                          "select c from Client c cjoin c.adresses add where add.libelle = :lib"
                          , Client.class
                      );
```

Les jointures peuvent même être implicites dans certains cas *(dans une association de type ...**ToOne**)*

```
Query<Adresse> query = session.createQuery(
                             "select a from Adresse a where a.client.nom = :nom"
                             , Adresse.class
                         );
```

Langage HQL

1. Présentation
2. Sélection
3. Mise à jour
4. Suppression
5. Les requêtes nommées
6. La projection
7. La pagination
8. Les associations
9. *Travaux pratiques*

TP

- Créer en HQL les methodes contenant les requetes suivantes:
  - Lister tous les artistes d'un pays spécifique.
  - Lister tous les artistes avec un manager ayant un budget supérieur à un montant donné.
  - Compter le nombre d'albums sortis après une certaine année pour chaque artiste.
  - Lister les artistes qui ont un 'registration' dans une catégorie spécifique.
  - Lister les albums d'un artiste donné, triés par année de sortie décroissante.
  - Lister le nombre d'albums par artiste, triés par nombre d'albums dans l'ordre décroissant.

Plan

1. Introduction
2. Mise en place
3. Le mapping
4. Les associations
5. La manipulation des entités
6. Langage HQL
7. *API Criteria*
8. Transactions et accès concurrents
9. Le cache Hibernate
10. La génération automatique

API Criteria

1. *Présentation*
2. Restrictions
3. Résultat unique
4. Tri
5. Associations
6. Projections et agrégats
7. Synthèse
8. Travaux pratiques

L'API Criteria constitue une autre façon de récupérer de l'information.

L'idée de cette API est de construire un modèle de l'information recherchée.

Il s'agit d'une API incluse dans la spécification JPA.

Dans sa forme la plus simple :

- On va utiliser un `CriteriaBuilder`.
- A partir de ce CriteriaBuilder, on va créer une requête avec la méthode `createQuery` en indiquant le type des entités recherchées. On récupère alors un objet `CriteriaQuery`.
- A partir de ce CriteriaQuery, on va appeler sa méthode `from` pour indiquer que la requête racine correspond à une entité particulière. On récupère alors un objet `Root`.
- On exécute alors la méthode `select` du CriteriaQuery avec en paramètre l'objet ***Root*** récupéré.

Exemple :

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Client> criteria = builder.createQuery(Client.class);
Root<Client> root = criteria.from(Client.class);
criteria.select(root);

List<Client> resultats = session.createQuery(criteria).getResultList();
```

La méthode `createQuery` retournant un objet `Query`, on peut utiliser le système de pagination déjà vu plus tôt *(* `setFirstResult` *et* `setMaxResults` *)*.

API Criteria

1. Présentation
2. ***Restrictions***
3. Résultat unique
4. Tri
5. Associations
6. Projections et agrégats
7. Synthèse
8. Travaux pratiques

Pour faire des restrictions, on utilise la méthode `where` associée au `CriteriaBuilder` vu précédemment :

```
criteria.select(root);
criteria.where(  builder.gt(
                        root.get("age"),
                        builder.parameter(Integer.class, "age")
              ));

List<Client> resultats = session.createQuery(criteria)
                                .setParameter("age", 40).getResultList()
```

L'idée est donc de créer des restrictions avec le `CriteriaBuilder` en y spécifiant des paramètres nommés.

Lors de la création de l'appel à `createQuery`, on peut alors donner des valeurs aux paramètres avec la méthode `setParameter`.

Le `CriteriaBuilder` propose de nombreuses méthodes pour venir spécifier ses restrictions, par exemple :

Méthode	Utilité
equal	Egalité
gt	Supérieur à
ge	Supérieur ou égal à
lt	Inférieur à
le	Inférieur ou égal à
like	Comme l'opérateur SQL - Utile pour les chaînes de caractères
between	Comme l'opérateur SQL
isNull	Est NULL
isNotNull	Est non NULL

Enfin, il est possible de combiner les restrictions avec les opérateurs booléens, toujours fournis par le `CriteriaBuilder` :

Méthode	Utilité
or	Ou
and	Et
not	Non

En emboitant les prédicats, on peut alors créer des restrictions puissantes

```
criteria.select(root);

Predicate p1 = builder.gt(root.get("age"), builder.parameter(Integer.class, "age"));
Predicate p2 = builder.like(root.get("nom"), builder.parameter(String.class, "nom"));

criteria.where(builder.and(p1, p2));

List<Client> resultats = session.createQuery(criteria)
                                .setParameter("age", 40).setParameter("nom", "E%")
                                .getResultList();
```

API Criteria

1. Présentation
2. Restrictions
3. *Résultat unique*
4. Tri
5. Associations
6. Projections et agrégats
7. Synthèse
8. Travaux pratiques

Pour ne récupérer que les résultats uniques, on dispose de la méthode distinct :

```
criteria.select(root).distinct(true);
```

Lorsqu'une requête retourne un unique résultat, il est possible de le récupérer avec la méthode `getSingleResult` :

```
Predicate p = builder.equal(
                        root.get("nom"),
                        builder.parameter(String.class, "nom")
);
criteria.where(p);
Client c = session.createQuery(criteria).setParameter("nom", "Dubois").getSingleResult();
```

Cette méthode est très utile lorsqu'on est persuadé de ne récupérer qu'un unique résultat.
Si la requête en retourne plusieurs, alors une exception `NonUniqueResultException` sera lancée.

API Criteria

1. Présentation
2. Restrictions
3. Résultat unique
4. *Tri*
5. Associations
6. Projections et agrégats
7. Synthèse
8. Travaux pratiques

Pour faire du tri, on dispose, depuis l'objet `CriteriaQuery`, de la méthode `orderBy`.

Cette méthode accepte un nombre illimité de paramètres, obtenus grâce aux méthodes `asc` et `desc` du `CriteriaBuilder`:

```
criteria.select(root);
criteria.orderBy( builder.asc(root.get("age")) );
List<Client> resultats = session.createQuery(criteria).getResultList();
```

```
criteria.select(root);
criteria.orderBy(
            builder.asc(root.get("age")),
            builder.desc(root.get("nom"))
);
List<Client> resultats = session.createQuery(criteria).getResultList();
```

API Criteria

1. Présentation
2. Restrictions
3. Résultat unique
4. Tri
5. *Associations*
6. Projections et agrégats
7. Synthèse
8. Travaux pratiques

Avec l'API Criteria on peut travailler avec les associations, en particulier sur la restriction :

```
criteria.select(root);

criteria.where(
    builder.equal(
        root.join("adresses").get("codePostal"),
        builder.parameter(Integer.class, "code")
    )
);

List<Client> resultats = session.createQuery(criteria)
                                .setParameter("code", 76100).getResultList();
```

On accède à l'association avec la méthode `join` et on donne le champ qui nous intéresse avec la méthode `get`.

Pour ne récupérer que les associations d'une requête :

```
CriteriaQuery<Client> criteria = builder.createQuery(Client.class);
Root<Adresse> root = criteria.from(Adresse.class);    // Requête sur Adresse

criteria.select(root.join("clients")); // On récupère les clients dans la projection

criteria.where(
            builder.equal(
                        root.get("codePostal"),
                        builder.parameter(Integer.class, "code")
            )
);

List<Client> resultats = session.createQuery(criteria)
                                    .setParameter("code", 76100).getResultList();
```

API Criteria

1. Présentation
2. Restrictions
3. Résultat unique
4. Tri
5. Associations
6. *Projections et agrégats*
7. Synthèse
8. Travaux pratiques

Lorsqu'on travaille sur les projections et les agrégats, le type des résultats va changer : il est alors nécessaire de créer une classe qui viendra représenter le résultat :

ResultatNomVal.java

```
public class ResultatNomVal {
    public String nom;
    public double valeur;

    public ResultatNomVal(String nom, double valeur) {
        this.nom = nom;
        this.valeur = valeur;
    }
}
```

On peut alors créer la requête correspondante :

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<ResultatNomVal> criteria = builder.createQuery(ResultatNomVal.class);

Root<Client> root = criteria.from(Client.class);

criteria.select(
                builder.construct(
                        ResultatNomVal.class,
                        root.get("nom"),
                        builder.avg(root.get("age"))
                )
).groupBy(root.get("nom"));

List<ResultatNomVal> resultats = session.createQuery(criteria).getResultList();
```

API Criteria

1. Présentation
2. Restrictions
3. Résultat unique
4. Tri
5. Associations
6. Projections et agrégats
7. *Synthèse*
8. Travaux pratiques

En conclusion, l'API Criteria est une API très puissante et qui permet de s'affranchir complètement du SQL.

Il s'agit d'une API très type-safe, ce qui permet déjà de bien fiabiliser son code.

Cependant, elle nécessite du travail afin d'être pleinement maîtrisée et peut se révéler particulièrement verbeuse.

API Criteria

1. Présentation
2. Restrictions
3. Résultat unique
4. Tri
5. Associations
6. Projections et agrégats
7. Synthèse
8. *Travaux pratiques*

TP

- Créer en utilisant l'API criteria les méthodes contenant les requetes suivantes:
  - Lister tous les artistes d'un pays spécifique.
  - Lister tous les artistes avec un manager ayant un budget supérieur à un montant donné.
  - Compter le nombre d'albums sortis après une certaine année pour chaque artiste.
  - Lister les artistes qui ont un 'registration' dans une catégorie spécifique.
  - Lister les albums d'un artiste donné, triés par année de sortie décroissante.
  - Lister le nombre d'albums par artiste, triés par nombre d'albums dans l'ordre décroissant.

Plan

1. Introduction
2. Mise en place
3. Le mapping
4. Les associations
5. La manipulation des entités
6. Langage HQL
7. API Criteria
8. *Transactions et accès concurrents*
9. Le cache Hibernate
10. La génération automatique

Transactions et accès concurrents

1. *Transactions*
2. Accès concurrents
3. Travaux pratiques

Les transactions permettent de créer et de contrôler des opérations atomiques.

Dans le cas d'une application multi-utilisateurs, il conviendra alors de gérer la concurrence de ces transactions.

On utilise les transactions avec les deux méthodes déjà vues :

```
Transaction tx = session.beginTransaction(); // Ouverture

// Instructions

tx.commit();  // Validation
```

On peut bien entendu annuler une transaction avec la méthode rollback

```
tx.rollback();
```

Un objet Transaction propose plusieurs méthodes :

Méthode	Utilité
void begin()	Démarre une transaction
void commit()	Valide une transaction
void rollback()	Annule une transaction
void setTimeout(int seconds)	Spécifiela durée maximale d'une transaction
boolean isAlive()	Permet de savoir si une transaction est en cours
boolean wasCommited()	Permet de savoir si une transaction a été validée avec succès
boolean wasRolledBack()	Permet de savoir si une transaction a été annulée avec succès

Exemple complet :

```
Session session = null;
Transaction tx = null;
try {
        session = sessionFactory.openSession();
        tx = session.beginTransaction();

        //Instructions

        tx.commit();

} catch (Exception ex) {
        ex.printStackTrace();
        tx.rollback();
}
finally {
        session.close();
}
```

Transactions et accès concurrents

1. Transactions
2. *Accès concurrents*
3. Travaux pratiques

De manière traditionnelle, les bases de données implémentent un système de verrous, afin de prévenir qu'une transaction ne vient pas modifier des modifications en cours au sein d'une autre transaction.

Il reste donc important en premier lieu de se documenter sur la façon dont le SGBD utilisé gère les accès concurrents.

Dans le cas d'un serveur d'application JEE, il est possible d'utiliser JTA *(Java Transaction API)* afin de gérer au mieux les transactions et spécifier des niveaux d'isolation.

Hibernate, avec la spécification JPA, permet cependant de mettre en place une détection de conflits optimisée grâce à un suivi de version.

Pour activer le suivi de version sur une entité, il convient d'y ajouter une propriété annotée avec `@Version` :

```
@Version
public long getVersion() {
    return this.version;
}
```

Cette propriété sera gérée par Hibernate comme un simple compteur qui sera incrémenté à chaque fois qu'une instance sera ***dirty***.

Avec la gestion du suivi de version, Hibernate vérifiera les numéros de version lors d'une modification pour s'assurer que l'instance en cours est bien à jour.

Il s'agit d'un contrôle optimiste.

```
Client c1 = new Client();
c1.setNom("ANGELE");
c1.setPrenom("Kévin");
c1.setAge(38);

session.save(c1);

c1.setAge(90);
```

```
insert into Client (age, nom, prenom, version) values (?, ?, ?, ?)
update Client set age=?, nom=?, prenom=?, version=? where id=? and version=?
```

Si la mise à jour n'est pas réalisable car le numéro de version ne correspond pas, alors une exception `OptimisticLockExceptionest` lancée.

Il est possible aussi de faire du suivi de version à partir d'un timestamp généré manuellement ou automatiquement

En cas de gestion automatique, Hibernate utilisera la base de données pour récupérer la valeur actuelle du timestamp :

```
@Version
@Type(type = "dbtimestamp")      // Enlever l'annotation @Type pour une gestion manuelle
public Date getUpdated() {
        return this.updated;
}
```

```
select now()
insert into Client (age, nom, prenom, updated) values (?, ?, ?, ?)
select now()
update Client set age=?, nom=?, prenom=?, updated=? where id=? and updated=?
```

Une autre possibilité est de gérer les accès concurrents sans attributs complémentaires : Hibernate vérifiera alors l'entité avec la liste de toutes les colonnes.

Il faut cependant demander à Hibernate de générer le SQL à la volée. Par défaut, Hibernate génère les requêtes SQL standards au lancement via des requêtes préparées.

```
@Entity
@OptimisticLocking(type = org.hibernate.annotations.OptimisticLockType.ALL)
@DynamicUpdate
public class Client {
        ...
}
```

```
insert into Client (age, nom, prenom) values (?, ?, ?)
update Client set age=? where id=? and age=? and nom=? and prenom=?
```

Il est possible également de faire un contrôle pessimiste en verrouillant l'objet pour son usage propre. On utilise pour cela la classe `LockMode` :

Valeur	Résultat
NONE	Pas de blocage. Appliqué à tous les objets à la fin d'une transaction
READ	La version de l'entité est vérifiée à la fin de la transaction
WRITE	La version de l'entité est incrémentée automatiquement même si l'entité n'a pas changé
PESSIMISTIC_FORCE_INCREMENT	L'entité est verrouillée et sa version est incrémentée, même si l'entité n'a pas changé
PESSIMISTIC_READ	L'entité est verrouillée avec un verrou partagé, si le SGBD supporte cela. Sinon un verrou explicite est utilisé
PESSIMISTIC_WRITE, UPGRADE	L'entité est verrouillée avec un verrou explicite
UPGRADE_NOWAIT	L'acquisition du verrou échoue rapidement si la ligne est verrouillée
UPGRADE_SKIPLOCKED	L'acquisition du verrou ignore les lignes déjà verrouillées

L'utilisation du contrôle pessimiste est utilisé explicitement avec les méthodes :

- `Session.load()` en spécifiant le ***LockMode*** utilisé
- `Session.lock()`
- `Query.setLockMode()`

Transactions et accès concurrents

1. Transactions
2. Accès concurrents
3. *Travaux pratiques*

Plan

1. Introduction
2. Mise en place
3. Le mapping
4. Les associations
5. La manipulation des entités
6. Langage HQL
7. API Criteria
8. Transactions et accès concurrents
9. ***Le cache Hibernate***
10. La génération automatique

Hibernate propose un système de deux caches :

- un cache de premier niveau (L1) dans lequel toutes les requêtes doivent passer
- un cache de second niveau (L2) qui est optionnel et configurable

Le cache de premier niveau permet de s'assurer que, au cours d'une session, les requêtes pour un objet au sein de la base de données retournera toujours la même instance de l'objet.

Pour supprimer une instance du cache L1, il faut appeler, depuis la session, la méthode `evict` avec en paramètre l'objet à supprimer :

```
session.evict(c1);
```

Pour vider le cache L1 en entier, on utilise la méthode `clear` depuis la session :

```
session.clear();
```

Le cache L1 sera toujours consulté avant le cache L2 si ce dernier est utilisé

Pour utiliser un cache L2, il faut mettre en place une implémentation d'un cache L2 au sein du projet.

Cache	URL
EHCache	Cache ausein du process
Infinispan	Cache distribué
OSCache	Cache au sein du process

Mise en place de EHCache

pom.xml

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.4.4.Final</version>
</dependency>
```

Paramétrage d'Hibernate pour utiliser ce cache de niveau 2 :

hibernate.cfg.xml

```
<property name="cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory
</property>
```

Ensuite, il convient de paramétrer les entités pour lesquelles on souhaite utiliser le cache de second niveau :

```
@Entity
@Cacheable
@Cache(usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_ONLY)
public class Client {
        ...
}
```

Avec cette configuration, si une session essaie de lire un Client en base, Hibernate ira voir dans le cache L2 si ce dernier n'est pas présent plutôt que de solliciter la base de données.

Une session peut être configurée afin de lui spécifier sa politique d'accès au cache L2, grâce à la méthode `setCacheMode` qui prend en paramètre une valeur `CacheMode` :

Valeur	Effet
NORMAL	Comportement normal - Lecture et écriture quand c'est nécessaire
GET	De la donnée n'est jamais ajoutéeau cache
PUT	De la donnée n'est jamais lue depuis le cache
REFRESH	Comme `PUT` mais l'option `use_minimal_puts` sera ignorée si elle existe
IGNORE	De la donnée n'est jamais lue ni écrite dans le cache - Les données dans le cache seront cependant invalidées si elles ont été mises à jour par la session

Plan

1. Introduction
2. Mise en place
3. Le mapping
4. Les associations
5. La manipulation des entités
6. Langage HQL
7. API Criteria
8. Transactions et accès concurrents
9. Le cache Hibernate
10. ***La génération automatique***

Synthèse

Tout au long de cette formation nous avons appris les notions de base du framework Hibernate.

Maintenant vous pouvez :

- Réaliser un mapping Objet/Relationnel
- Créer, mettre à jour, supprimer et charger des objets persistants
- Effectuer des requêtes avec le langage HQL et l'API Criteria
- Gérer les transactions et les accès concurrents
- Configurer le cache d'Hibernate
- Utiliser les outils d'automatisation

Il ne vous reste plus qu'à pratiquer maintenant !

Avez-vous des questions ?

Merci à tous pour votre participation !

Sources

- La persistance des objets
- Hibernate
- Java Persistence API
- EJB : les Entity JavaBeans, clé de la persistance
- Les EJB (Entreprise Java Bean)