MODULE WS

Présentation des services Web

1.

Présentation des architectures distribuées

Architectures ?

- Architecture centralisée : « client/serveur »
  - 1 serveur / N clients
- Architecture distribuée : « peer to peer »
  - N serveurs / N clients

Rappel sur l'architecture centralisée

- Client/serveur : distinction stricte entre le rôle de client et le rôle de serveur
  - 1 client effectue une requête pour un service donné sur un serveur et attend une réponse
  - 1 serveur reçoit une demande de service, la traite et retourne une réponse au client

Rappel sur l'architecture centralisée

- Caractéristiques du client :
  - Actif
  - Connecté à un serveur
  - Envoie des requêtes à un serveur
  - Attend et traite les réponses du serveur
  - Interagit avec un utilisateur final (par exemple avec une IHM)

Rappel sur l'architecture centralisée

- Caractéristiques du serveur :
  - Passif
  - A l'écoute des requêtes clients
  - Traite les requêtes et fournit une réponse
  - Pas d'interaction directe avec les utilisateurs finaux

Rappel sur l'architecture centralisée

- Exemples d'architecture centralisée client/serveur :
  - Consultation de pages web (envoi de requêtes HTTP depuis un navigateur à un serveur pour consulter les pages)
  - Gestion des mails (client pour envoyer et recevoir les mail, serveur pour la gestion : SMTP, POP, IMAP)

Rappel sur l'architecture centralisée

- Découpage en couches :
  - **Présentation** : affichage, dialogue avec un utilisateur final
  - **Service** : traitements, règles de gestion et logique applicative
  - **Données** : DAO, persistance des données

Rappel sur l'architecture centralisée

- Découpage en couches :
  - La **répartition** de ces couches entre les rôles de client et de serveur permet de distinguer entre les différents types d'architecture client/serveur
    - 2 tiers
    - 3 tiers
    - N tiers

Architectures distribuées

- Relations d’égal à égal :
  - Pas de connaissance globale du réseau
  - Pas de coordination globale des nœuds
  - Chaque nœud ne connaît que les nœuds constituant son voisinage
  - Toutes les données sont accessibles depuis n’importe quel nœud
  - Les nœuds sont volatiles

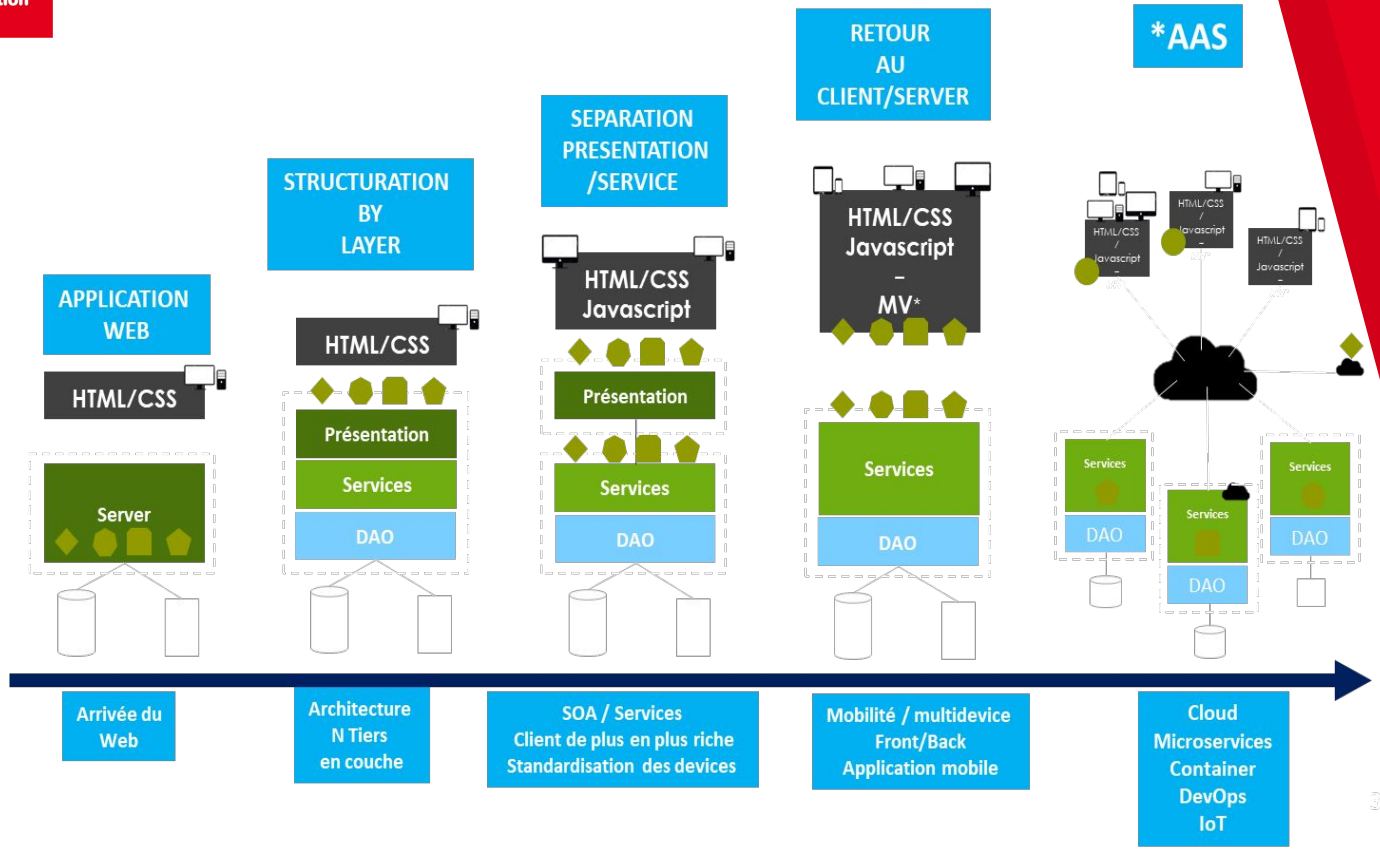
Architectures distribuées

- Avantages :
  - Plus adapté à la montée en charge (scalabilité)
  - Meilleure robustesse en cas de panne (réplication, pas de SPOF : « single point of failure »)

Architectures distribuées

- Inconvénients :
  - Problématiques spécifiques
    - Concurrence
    - Fragmentation des données
    - Gestion de la réplication
    - ...

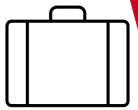
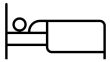
Evolution des architectures au cours du temps



2.

Positionnement des Web services

Les Services Web

- Exemple d'une agence de voyage :
  - Un produit « voyage » = une combinaison de plusieurs produits 
    - Réservation des billets de transport  
    - Réservation des nuits d'hôtel 
    - Réservation des locations de voiture 
    - …

Les Services Web

- Exemple d'une agence de voyage :
  - La construction d'un produit « voyage » est le résultat d'informations récupérées auprès de différents fournisseurs :
    - Compagnies aériennes
    - Compagnies ferroviaires
    - Loueurs de voiture
    - Chaînes hôtelières
    - …

Les Services Web

- Exemple d'une agence de voyage :
  - Une application de réservation de voyage sollicite d'autres applications réparties pour satisfaire la demande !
  - 2 types de sollicitations :
    - Transformation : adaptation du dialogue en fonction du profil utilisateur
    - Agrégation : appel à des applications proposées par des partenaires ou fournisseurs

Les Services Web

- Exemple d’une agence de voyage :
  - Pour réaliser cela, on s’appuie des Services Web !

Les Services Web

- Un Service Web, c'est quoi ?
  - Fonction distante mise à disposition sur un réseau accessibilité
  - Infrastructure souple pour des échanges entres des systèmes distribués hétérogènes
  - Localisable à partir de registres
  - Couplage faible

Les Services Web

- Un Service Web, c'est quoi ?
  - Répond à la problématique B2B (SOA -> architecture orientée service)
  - Un service résout un problème donnée
  - Combinaison possible pour résoudre des problèmes complexes

Les Services Web

- Idée générale
  - Un client a un besoin
  - Pour un besoin, plusieurs services et donc fournisseurs peuvent exister (avec ses propres caractéristiques)
  - Le client choisit un fournisseur pour pouvoir utiliser son service (celui qui correspond à son besoin et qui est compatible avec ses exigences (coût, performance, …)

Les Services Web

- Pourquoi peut-on avoir besoin de Services Web ?
  - Besoin d'interopérabilité dans des environnements applicatifs distribués
  - Echanges sur des protocoles standards (HTTP, SMTP, …)
  - Échanges entre des systèmes hétérogènes (environnements différents, langages différents)

Les Services Web

- Les usages
  - Assemblage de composants faiblement couplés
  - Définition indépendantes mais interaction
  - Adapté pour les applications orientées messages
  - Asynchronisme

Les Services Web

- Les acteurs
  - Le client : celui qui invoque le service web
  - Le fournisseur : celui qui fournit et met à disposition le service web
  - L'annuaire : celui qui détient et partage les informations sur les services web

Les Services Web

- Les acteurs
  - Le fournisseur :
    - Serveur d'application (par exemple JEE)
    - Expose un ou plusieurs services (EJB, servlets, enveloppés d'une couche « service »)

Les Services Web

- Les acteurs
  - L'annuaire :
    - Déclaration dans un annuaire = publication

3.

Approches SOAP et REST

Les types de service web

- Services web de type **SOAP**

- Services web de type **REST**

Les types de service web

- Services web de type **SOAP**
  - SOAP est un protocole
  - SOAP = Simple Object Access Protocol
  - Initialement conçu pour que des applications développées avec différents langages sur différentes plateformes puissent communiquer

Les types de service web

- Services web de type **SOAP**
  - Protocole = règles imposées qui augmentent la complexité et les coûts
  - Mais, standards qui assurent la conformité et sont privilégiés pour certaines applications en entreprise

Les types de service web

- Services web de type **SOAP**
  - Les standards de conformité intégrés incluent la sécurité, l'atomicité, la cohérence, l'isolement et la durabilité (ACID), et un ensemble de propriétés qui permet d'assurer des transactions de base de données fiables

Les types de service web

- Services web de type **SOAP**
  - Les principales spécifications :
    - WS-Security : standardise la manière dont les messages sont sécurisés et transférés via des identifiants uniques appelés jetons
    - WS-ReliableMessaging : standardise la gestion des erreurs entre les messages transférés par le biais d'une infrastructure informatique non fiable

Les types de service web

- Services web de type **SOAP**
  - Les principales spécifications :
    - WS-Adressing : ajoute les informations de routage des paquets en tant que métadonnées dans des en-têtes SOAP, au lieu de les conserver plus en profondeur dans le réseau
    - WSDL (Web Services Description Language) : décrit la fonction d'un service web ainsi que ses limites

Les types de service web

- Services web de type **SOAP**
  - Lorsqu'une requête de données est envoyée à une API SOAP, elle peut être gérée par n'importe quel protocole de couches de l'application : HTTP, SMTP, TCP, ...

Les types de service web

- Services web de type **SOAP**
  - Les messages SOAP doivent être envoyés sous la forme d'un document XML
  - Une fois finalisée, une requête destinée à une API SOAP ne peut pas être mise en cache par un navigateur (pas possible d'y accéder plus tard sans la renvoyer vers l'API)

Les types de service web

- Services web de type **REST**
  - REST **n'est pas un protocole**
  - REST est un ensemble de principes architecturaux adapté aux besoins des services web et applications mobiles légers
  - La mise en place de ces recommandations est laissée à l'appréciation des développeurs

Les types de service web

- Services web de type **REST**
  - L'envoi d'une requête à une API REST se fait généralement par le protocole HTTP
  - À la réception de la requête, les API développées selon les principes REST (appelées API ou services web RESTful) peuvent renvoyer des messages dans différents formats : HTML, XML, texte brut, JSON

Les types de service web

- Services web de type **REST**
  - Le format JSON (JavaScript Object Notation) est le plus utilisé pour les messages : léger, lisible par tous les langages de programmation et les humains
  - Les API REST sont plus flexibles et plus faciles à mettre en place

Le format JSON (exemple)

```
{
    "listAuteurs": {
        "count": "3",
        "0": {
            "id": 2,
            "firstName": "Joe",
            "lastName": "Goncalves",
            "phone": "0102030405",
            "email": "gonzalves@gmail.c",
        },
        "0": {
            "id": 4,
            "firstName": "Claude",
            "lastName": "Delannoy",
            "phone": "0677889900",
            "email": "claude@delanooy.com",
        },
        "0": {
            "id": 10,
            "firstName": "Stefan",
            "lastName": "Zweig",
            "phone": "0660606060",
            "email": "stefan.zweig@lejoueurdechecs.de",
        },
    }
}
```

MODULE WS

REST

API Rest en JAVA

- Beaucoup de manières différentes :
  - Manuellement (Servlet) 
  - Framework dédié (Exemple : Eclipse Vert.x, …) 
  - Le Framework Spring 

MODULE JPA

JPA avec Hibernate

1.

Introduction

Introduction JPA

- **JPA** : Java Persistence API
  - Framework de **persistance** en Java
    - ORM : Object-Relational Mapping (exemple : Hibernate)
  - **Principes** de base
    - Définition de la correspondance entre le structure des classes Java et le schéma relationnel de la Base de données
    - Manipulation directe des objets dans le code Java

Introduction JPA

- **JPA** : Java Persistence API
  - Le Framework s'occupe de la **transformation**
    - Plus besoin de requêtes SQL !
    - Langage de requêtage propre mis à disposition

2.

Problématique de la persistance

Problématique de la persistance

- **Limites** de JDBC
  - Nécessite l'utilisation de **requêtes SQL**
  - Représentation **différente** des données
    - Langage SQL pour le requêtage
    - Classes Java pour les entités

Problématique de la persistance

- **Limites** de JDBC
  - Ça marche, **mais…**
    - Beaucoup de code à produire
    - Si un grand nombre de références entre les classes, nécessité de charger beaucoup de choses (problématique des **ressources**)
    - Problème de **cohérence** entre les objets et la Base de données
      - C'est au développeur de gérer la cohérence entre le contenu des objets et la Base de données

Problématique de la persistance

- **De JDBC à JPA**
  - Avec JPA, on définit des **correspondances** entre des classes (POJO : Plain Old Java Object) et des tables
  - JPA gère la **cohérence** entre les objets et les données en Base de données
  - Beaucoup **moins de code** technique à produire !

Problématique de la persistance

- **Fonctionnement de JPA**
  - Manipulation **d'objets** métier Java **uniquement**
  - API proposant des fonctionnalités :
    - Récupération d'objets à partir des données de la Base
      - Langage propre : JPQL
    - Persistance des objets en base
      - Insertion, modification, suppression
      - Avec gestion des transactions

Problématique de la persistance

- **Fonctionnement de JPA**
  - Nécessite une **implémentation**
    - Hibernate, TopLink, EclipseLink, ...
    - **Attention** : éviter d'utiliser les fonctionnalités spécifiques d'une implémentation particulière pour ne pas être dépendant
      -

3.

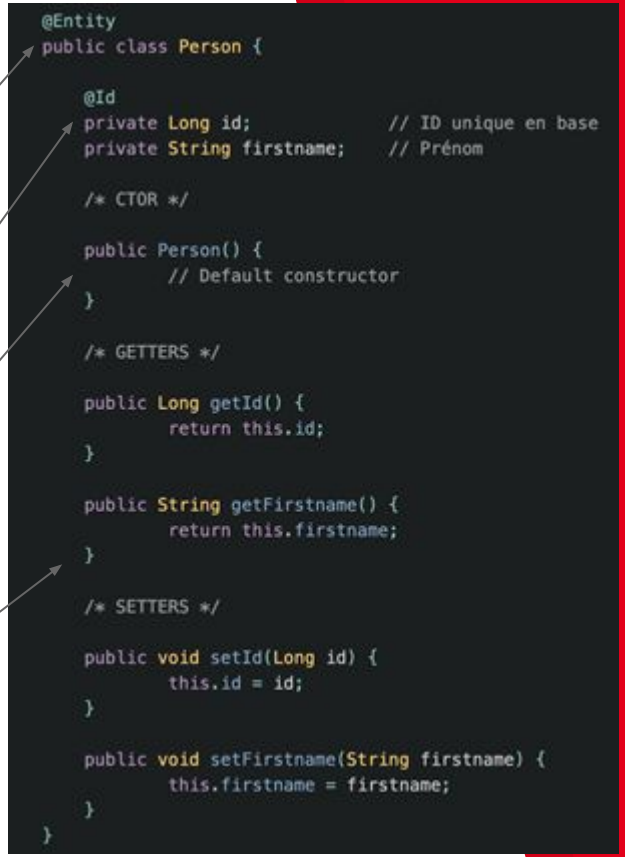
Modèle de persistance

Modèle de persistance

- **Mapping sur les entités Java**
  - Une **entité** = un objet **métier**
  - Une classe Java est **mappée** sur une table SQL
    - **Correspondance** entre les attributs de la classe et les colonnes de la table
    - Mapping réalisé avec des **annotations** dans la classe

Modèle de persistance

- **Mapping sur les entités Java**
  - Au minimum :
    - **@Entity** : défini comme entité
    - **@Id** : pour l'identifiant de l'entité
    - Un constructeur (pour instancier)
    - Un getter et un setter pour chaque attribut



```java
@Entity
public class Person {

    @Id
    private Long id;              // ID unique en base
    private String firstname;     // Prénom

    /* CTOR */

    public Person() {
            // Default constructor
    }

    /* GETTERS */

    public Long getId() {
            return this.id;
    }

    public String getFirstname() {
            return this.firstname;
    }

    /* SETTERS */

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

    public void setFirstname(String firstname) {
            this.firstname = firstname;
    }
}
```

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - Les annotations **@Entity** et **@Id** **ne suffisent pas** à faire comprendre à l'ORM comment manipuler cette entité
  - Il faut ajouter **@Table** pour décrire plus spécifiquement les détails à propos de la base de données (ex: nom, schéma)

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - **@GeneratedValue** est utilisé conjointement avec @Id pour les valeurs de clé générée :
    - **IDENTITY** : pour spécifier une colonne d'identité de la base de données
    - **AUTO** : pour choisir automatiquement une implémentation basée sur la base de données utilisée
    - **SEQUENCE** : pour utiliser une séquence (si la base de données la supporte)(Voir @SequenceGenerator)
    - **TABLE** : pour spécifier qu'une base de données utilisera une table et une colonne d'identité pour s'assurer son caractère unique (voir @TableGenerator)

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - **Exemple :**

```
@Entity
@Table(name = "person")
public class Person {

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

    private String firstname;
```

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - En l'absence d'information précise, Hibernate va assumer certaines configurations par défaut
    - L'utilisation de **@Column** permet de définir les noms de colonnes souhaités, ainsi que des informations à leur propos

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - L'utilisation de **@Column** permet de définir les noms de colonnes souhaités, ainsi que des informations à leur propos :
      - columnDefinition
      - insertable
      - length
      - name
      - nullable
      - precision
      - scale
      - table
      - unique
      - updatable

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
  - Sans les informations de l'annotation, Hibernate aurait considéré l'attribut « firstname » comme :
    - Étant mappé à une colonne nommée "firstname" (et pas "first_name")
    - Pouvant avoir une valeur à NULL
    - Ayant une longueur à 255 caractères

```java
@Entity
@Table(name = "person")
public class Person {

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

    @Column(
        name = "first_name",
        nullable = false,
        length = 100
    )
    private String firstname;
```

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - En dehors de « simplement décrire » la base de données dans le code, Hibernate possède aussi des annotations permettant d'écrire du code plus simple :
    - Exemple : l'utilisation de **@Enumerated** permet de dire à Hibernate de « mapper » des chaînes de caractères, des nombres en base vers des « **Enums** » Java (plus pratique)
    - Exemple : l'utilisation de **@Temporal** permet de dire à Hibernate de « mapper » des dates SQL (TIMESTAMP, DATETIME, etc.) vers des « **Date** » Java (plus pratique)

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Il existe 4 types de relation entre entités :
    - Relation « one to one » : **relation 1 -> 1**
    - Relation « one to many » : **relation 1 -> n**
    - Relation « many to one » : **relation n -> 1**
    - Relation « many to many » : **relation n -> n**

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - S'ajoutent à ces types de relation « des **configurations** »
    - Notion de **sens** :
      - Relation **unidirectionnelle**
      - Relation **bidirectionnelle**
    - Notion de **cascade** : que faire de B, lié à A,
      - Lors d'une **mise à jour** de A ?
      - Lors d'une **suppression** de A ?

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **1 -> 1** : annotation **@OneToOne**
    - Pour définir une relation **forte**, **bidirectionnelle** entre 2 entités
    - Exemple : « Un navire est gouverné par un capitaine »

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **1 -> N** : annotation **@OneToMany**
    - Pour définir une relation entre 1 entité et une liste d'entités
    - Exemple : « Une personne possède un ou plusieurs téléphones »

```
// Person.java

@OneToMany(mappedBy = "person")
private List<Phone> phones;
```

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **N -> 1** : annotation **@ManyToOne**
    - Pour définir une relation **contraire** à @OneToMany (plusieurs entités liées à une autre)
    - Exemple : « Un ou plusieurs téléphones peuvent être détenus par une personne »

```
// Phone.java

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "person_id")
private Person person;
```

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **N -> N** : annotation **@ManyToMany**
    - Pour définir des relations entres entités dont on ne peut faire les liens qu'à travers des **tables de jointure**
    - Exemple : « Un ou plusieurs employés travaillent sur un ou plusieurs projets »

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **N -> N** : annotation **@ManyToMany**

```java
// Employee.java

@ManyToMany(cascade = { CascadeType.ALL })
@JoinTable(
    name = "Employee_Project",
    joinColumns        = { @JoinColumn(name = "employee_id") },
    inverseJoinColumns = { @JoinColumn(name = "project_id") }
)
List<Project> projects;
```

```java
// Project.java

@ManyToMany(mappedBy = "projects")
private List<Employee> employees;
```

Modèle de persistance

```java
// Phone.java

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "person_id")
private Person person;
```

- Concepts **ORM** : mode de récupération d'**entités** liées entre elles
  - Il existe **2 modes** de récupération d'entités
    - **LAZY** : interroge la base de données seulement quand la propriété est appelée (exemple: appel à un getter)
    - **EAGER** : interroge la base de données dès que l'objet original est créé

Modèle de persistance

- Concepts **ORM** : types d'effet « **cascade** » entre entités
  - Les relations entre entités **dépendent** souvent de l'existence d'une autre
  - Exemple : relation **Personne** <-> **Adresse**
    - Sans la personne, **l'adresse n'aurait pas de signification métier**
    - En **supprimant** la personne, on souhaiterait que son adresse soit **supprimée** aussi

Modèle de persistance

- Concepts **ORM** : types d'effet « **cascade** » entre entités

```java
@Entity
public class Person {

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

    private String name;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<Address> addresses;
}
```

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private int zipCode;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;
}
```

Modèle de persistance

- Bien réfléchir à la **cardinalité** des entités
- Ne pas partir tête baissée pour **écrire les entités**
- Attention à positionner le “fetch type” en LAZY sur les collections d’objets qui ne sont **pas nécessaires** au premier abord
  - Syndrome de « ramener la terre entière »
- Penser aux effets « **cascade** » entre entités
  - Souhaite-t-on garder une carte d’identité en base si l’on a supprimé la personne qui la possédait ?

4.

Manipulation des entités

Manipulation des entités

- Configuration d'une **unité de persistance**
  - **Fichier XML** définissant la liste des classes correspondant à des entités, la connexion à la base de données, des paramètres, …
  - Dans l'application **Java**
    - Récupération d'un **entity manager** pour manipuler les entités
    - Les modifications se font via une **transaction**

Manipulation des entités

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_2.xsd">
        <persistence-unit name="crm" transaction-type="JTA">
                <class>fr.m2i.crm.model.Customer</class>
                <class>fr.m2i.crm.model.Order</class>
                <properties>
                        <!-- database connection -->
                        <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
                        <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/crm" />
                        <property name="javax.persistence.jdbc.user" value="crm" />
                        <property name="javax.persistence.jdbc.password" value="crm" />
                </properties>
        </persistence-unit>
</persistence>
```

Manipulation des entités

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <persistence-unit ...>
      - Définit le nom de l'unité de persistance et le type de transaction utilisée

Manipulation des entités

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <class>
      - Définit qu'une classe Java sera une entité dont les instances seront persistantes en base de données

Manipulation des entités

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <properties>
      - Ensemble de propriétés de configuration
      - Par exemple, les paramètres de connexion à la base (URL, driver, utilisateur et mot de passe)

Manipulation des entités

- **Entity Manager**
  - Récupéré à partir de la fabrique de gestionnaire d'entité et via le nom donné à l'unité de persistance

```
private EntityManager em = null;
...
EntityManagerFactory emf = Persistence.createEntityManagerFactory("monPu");
em = emf.createEntityManager();
...
```

Manipulation des entités

- **Etat d'un objet persistant**
  - Plusieurs états pour l'instance d'une classe entité
    - **Persistant** : l'entité a une correspondance de contenu en BDD
      - **Gérée** : état synchronisé par le gestionnaire d'entité avec le contenu en BDD
      - **Détachée** : état non géré, les modifications ne sont plus synchronisées avec la BDD
    - **Transient** : objet java classique avec existence uniquement en mémoire de la JVM
      - Cas de l'instanciation d'un objet

Manipulation des entités

- **Etat d'un objet persistant**
  - Plusieurs états pour l'instance d'une classe entité
    - **Supprimé** : instance persistante dont on a supprimé le contenu associé en BDD
      - L'objet existe toujours en mémoire de la JVM mais n'a plus de correspondance en base

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Récupération d'une instance d'une entité en précisant sa classe et son identifiant
    - <T> T find(Class<T> entityClass, Object id)
      - Customer customer = em.find(Customer.class, 3) ;

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Modification du contenu de la BDD en mode transactionnel

```
EntityTransaction trans = null;
try {
        trans = em.getTransaction();
        trans.begin();
                … ici les actions ...
        trans.commit();
} catch(Exception e) {
 if (trans != null) trans.rollback();
}
```

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Rendre persistant en BDD un objet qui devient géré par le gestionnaire d'entités
    - void persist(Object entity)

```
trans.begin();
Customer customer = new Customer(...);
em.persist(customer);
trans.commit();
```

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Récupérer une copie gérée par le gestionnaire d'entité de l'objet passé en paramètre
    - <T> T merge(T entity)

```
Customer newCustomer = em.merge(customer);
// Modifications effectuées sur l'objet retourné par le merge, pas l'initial
newCustomer.setLastname("Dupont");
trans.commit();
```

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Détacher un objet du gestionnaire d'entité (les modifications sur l'objet ne sont alors plus reportées sur la BDD)
    - void detach(Object entity)

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Détacher tous les objets du gestionnaire d'entité
    - void clear()

Manipulation des entités

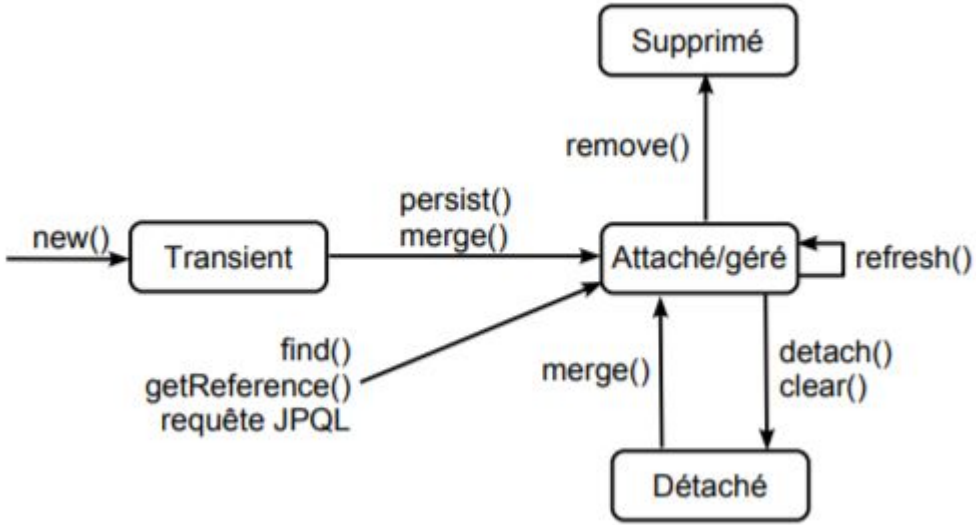
- **Opérations sur les instances d'entité**
  - Supprimer un objet : effacer ses données en base
    - void remove(Object entity)

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Remettre à jour le contenu de l'objet par rapport au contenu en base
    - void refresh(Object entity)

Manipulation des entités

- **Cycle de vie d'un objet persistant**



Manipulation des entités

- **JPQL et Query**
  - **JPQL** n'est pas du SQL
  - Langage **d'interrogation** centré sur les **objets Java**

```
private EntityManager em;
// ...

public List<Person> getAll() {
    Query query = em.createQuery("SELECT p FROM Person p");
    List<Person> results = query.getResultList();
}
```

Manipulation des entités

- **JPQL et Query**
  - **Query** avec paramètres
    - Liaison par **nom de paramètre** (“name parameter binding”)

Requête SELECT p FROM Person WHERE p.name = :searched

query.setParameter(“searched”, “Harry”);

Manipulation des entités

- **JPQL et Query**
  - **Query** avec paramètres
    - Liaison par **position de paramètre** (“positionnal parameter binding”)

Requête SELECT p FROM Person WHERE p.name = ?1

query.setParameter(1, “Harry”);

Manipulation des entités

- **JPQL et Query**
  - **Native Query**
    - Pour **reprendre** la main sur le **SQL**

```
List<Object[]> persons = session
    .createNativeQuery("SELECT id, name FROM PERSON" )
    .list();

for(Object[] person : persons) {
        Number id = (Number) person[0];
        String name = (String) person[1];
}
```

```
List<Phone> phones = session.createNativeQuery(
        "SELECT id, phone_number, phone_type, person_id " +
        "FROM Phone" )
    .addEntity( Phone.class )
    .list();
```