Université Nord Américaine privée
Institut International Technologie

Génie Logiciel Avancée
Architecture logicielle évoluée : Framework Spring

05- Persistance avec Spring

Mohamed ZAYANI

2025/2026

Plan

- Introduction à la persistance
- ORM (Object Relational Mapping)
- La persistance en JAVA
- JPA (Java Persistence API)
- Les deux approches de persistance avec Spring
- Mise en œuvre avec Spring Boot
- Personnalisation des requêtes avec Spring Data JPA
- Mapping des relations et d'héritage entre les entités

Introduction à la persistance

• Définition

- ✓ La persistance, en informatique, désigne la capacité à conserver les données même après l'arrêt du programme.
- ✓ Contrairement aux variables en mémoire vive (RAM), qui disparaissent après l'arrêt du programme, les données persistantes restent accessibles.
- ✓ La persistance désigne le mécanisme permettant de stocker et récupérer les données de façon durable (BD relationnelle, NoSQL, fichiers, etc.).

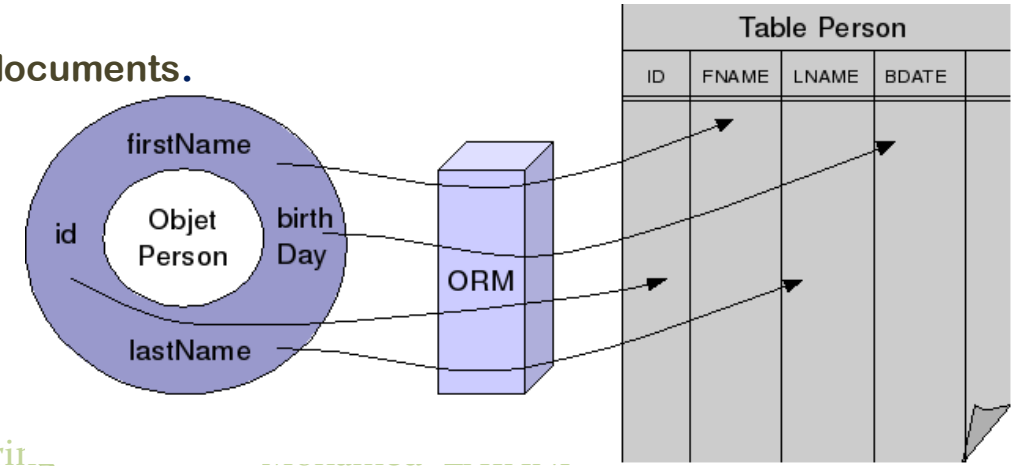
• Principe

On distingue deux couches principales :

- Domaine (modèle objet) : des objets Java, C#, Python, etc.
- Stockage (modèle relationnel ou autre) : tables, collections, documents.

→ Le défi majeur : faire le pont entre objet et relationnel

→ C'est le fameux Object Relational Mapping (ORM).



ORM

- **ORM est l'acronyme de Object/Relational Mapping**
- **ORM est un mapping objet-relationnel: type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet.**
- **Le but de l'ORM est de faciliter la manipulation de données stockées dans un Système de Gestion de Base de Données Relationnelles (SGBDR) au sein des langages de programmation objet.**
- **ORM offre une couche d'abstraction pour traduire des données extraites de la base de données vers un objet propre au langage de programmation et vice-versa.**

→ Le développeur travaille ainsi uniquement avec des objets sans se soucier du stockage sous-jacent des données.

Avantages du concept ORM

- **Simplifier le code**

  On ne fait que "parler objet" de la couche la plus haute à celle la plus basse.

- **Augmenter la maintenabilité**

  Il n'y a plus de code SQL à maintenir au sein du code objet.

- **Augmenter la portabilité**

  La base de données étant masquée il est possible de passer d'un SGBD à un autre.

La persistance en Java

- ## Avant l'arrivée des ORM

✓ Au départ, la persistance en Java était essentiellement via l'API standard JDBC (Java Database Connectivity).

✓ JDBC une API standard qui permet à une application Java de communiquer avec n'importe quelle BD relationnelle via un driver selon le principe suivant:

- Charger le driver (n'est plus obligatoire depuis la version JDBC4.0).
- Créer une connexion (Connection).
- Écrire des requêtes SQL (Statement / PreparedStatement).
- Exécuter la requête.
- Récupérer les résultats (ResultSet).
- Fermer les ressources.

```
// Charger le driver JDBC
Class.forName("com.mysql.jdbc.Driver");
// Créer une connexion
Connection con = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/ma_base", "root", "pwd");

// Insérer un client
PreparedStatement ps = con.prepareStatement(
"INSERT INTO client (nom, ville) VALUES (?, ?)");
ps.setString(1, "Ali");
ps.setString(2, "Tunis");
ps.executeUpdate();

// Lire les clients
ResultSet rs = con.createStatement().executeQuery("SELECT * FROM client");
while (rs.next()) {
System.out.println(rs.getString("nom") + " - " + rs.getString("ville"));
}

con.close();
```

✓ Problèmes du modèle JDBC:

- Beaucoup de code répétitif (connexion, fermeture, gestion exceptions).
- Dépendance forte au SQL:

  Toute modification de la BD impose de réécrire du code.
- Mapping manuel entre objets Java et lignes SQL.

  Exemple : un Client en Java doit être construit à partir d'un ResultSet.
- Transactions à gérer manuellement.
- Difficulté à gérer les relations

La persistance en Java

- **Transition vers ORM**
  - ✓ Comme JDBC restait bas-niveau, il fallait une solution pour :
    - ❑ Automatiser le mapping Objet ↔ Table.
    - ❑ Gérer les relations complexes.
    - ❑ Éviter d'écrire trop de SQL.
  - ✓ C'est là que sont apparus les ORM (Object Relational Mapping):
    - avec Hibernate en tête,
    - puis JPA comme standard.
  - ✓ Avec Spring, on a simplifié les intégrations (à travers les modules : Spring JDBC et Spring ORM)
  - ✓ Avec le module Spring Data, on a atteint un niveau ultra-abstrait, où l'on se concentre sur le métier et non plus sur l'implémentation technique.

JPA ?

- JPA est l'acronyme de Java Persistence API
- JPA est une solution standard de JEE pour adopter le concept ORM
- JPA une spécification JEE qui définit un ensemble de règles permettant la gestion de la correspondance entre des objets Java et une base de données (gestion de la persistance).
- JPA ne contient aucune implémentation, mais plutôt:
  - des interfaces (EntityManager, Query, Repository, etc.)
  - et des annotations (@Entity, @Id, @OneToMany, etc.).
- Pour que ça marche, il faut une implémentation concrète de JPA comme:
  - EclipseLink
  - OpenJPA
  - Hibernate (utilisé, par défaut, par Spring Boot et c'est le plus utilisé)

→ Quand utiliser spring-boot-starter-data-jpa, par défaut Spring Boot choisit Hibernate comme implémentation JPA

Concepts clés de JPA

Concept	Description
Entity	Classe Java persistante (@Entity).
Id	Identifiant unique (@Id).
EntityManager	Interface principale pour manipuler les entités (CRUD, requêtes JPQL).
Persistence Context	Générer l'artéfact et le déployer dans le dépôt local
Transaction	Ensemble cohérent d'opérations sur les entités (@Transactional).
JPQL	Java Persistence Query Language : langage de requêtes orienté objets.

Entité JPA

- Plain Old Java Object (POJO)

  Une simple classe JAVA

- Peut contenir des attributs persistants ou non

  l'état non persistant est spécifié grâce à l'annotation:

  @Transient

- Peut étendre d'autres entités ou des classes qui ne sont pas des entités
- Sérializable

  pas besoin de s'occuper des transferts d'objets

- Déclarée avec le mot clé :

  @Entity

Classe persistance

- Pour qu'une classe puisse être persistante, il est nécessaire:
  - qu'elle soit identifiée comme une entité (entity) en utilisant l'annotation @Entity
  - qu'elle possède un attribut identifiant en utilisant l'annotation @Id
  - qu'elle ait un constructeur sans argument

```
@Entity
public class Client {
@Id
private Long id;
private String prénom;
private String nom;
private String téléphone;
private String email;
private Integer age;
private Date dateNaissance;
// constructeurs/setter/getter
}
```

Entité JPA

- Une entité JPA, déclarée par l'annotation @Entity définit une classe Java comme étant persistante (associée à une table dans la base de données).

- Cette classe doit être implantée selon les normes des beans:
  - Les propriétés sont déclarées non publiques (de préférence de type « private »)
  - Chaque propriété possède deux accesseurs selon les conventions habituelles:
    - Un accesseur en lecture (getter) permet de lire la valeur d'un attribut.
    - Un accesseur en écriture (mutateur ou setter) permet de modifier la valeur d'un attribut.
  - Une entité doit comporter un constructeur sans argument

Les annotations

- Une annotation peut marquer:
  - soit le champ de la classe concernée
  - soit le getter de la propriété.
- Exemple:

```
@Column (name ="name")
private String nom;
```

Ou bien :

```
@Column (name ="name")
public String getNom()
{
return nom;
}
```

Définition d'une entité

```java
@Entity
public class Personne
{
    private String nom;
    private String prenom;
    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;
    }
    public Personne () {}
}
```

- **Remarques:**
- Toute entité doit avoir une propriété déclarée comme étant l'identifiant de la ligne dans la table correspondante.
- L'identifiant est indiqué avec l'annotation @Id

```java
@Id
private int matricule;
```

- La propriété annotée par « @Id » est traduite en une colonne constituant la clé primaire de la table correspondante dans la BD

Identifiant d'une entité

- L'identifiant d'une entité JPA est indiqué avec l'annotation @Id.
- Pour produire automatiquement les valeurs d'identifiant (en insérant dans le BD), on ajoute une annotation @GeneratedValue avec un paramètre strategy.
- Le paramètre strategy peut avoir plusieurs valeurs :
  - strategy = GenerationType.AUTO: La génération de la clé primaire est laissée à l'implémentation. C'est hibernate qui s'en charge et qui crée une séquence unique sur tout le schéma via la table hibernate_sequence
  - strategy = GenerationType.IDENTITY : La génération de la clé primaire se fera à partir d'une Identité propre au SGBD. Il utilise un type de colonne spéciale à la base de données. (Exemple pour MySQL, il s'agit d'un AUTO_INCREMENT)
  - strategy = GenerationType.TABLE : La génération de la clé primaire se fera en utilisant une table dédiée hibernate_sequence qui stocke les noms et les valeurs des séquences.Cette stratégie doit être utilisée avec une autre annotation qui est @TableGenerator.
  - strategy = GenerationType.SEQUENCE : La génération de la clé primaire se fera par une séquence définie dans le SGBD, auquel on ajoutera l'attribut generator. Cette stratégie doit être utilisée avec une autre annotation qui est @SequenceGenerator.
- Généralement, on utilise la première ou bien la deuxième stratégie. Exemple:

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

Annotation @Table

- Par défaut, une entité est associée à la table portant le même nom que la classe.
- Il est possible d'indiquer le nom de la table par une annotation @Table. (annotation optionnelle)
- Exemple:

```
@Entity
@Table(name="Person")
public class Personne
{
   .......
}
```

Annotation @Column

- Par défaut, toutes les propriétés non-statiques et non-finales d'une classe-entité sont persistantes (à être stockées dans la BD)
- Pour indiquer des options à une colonne dans la BD, on utilise le plus souvent l'annotation @Column.
- L'annotation @Column présente les principaux attributs suivants
  - name: indique le nom de la colonne dans la table
  - length: indique la taille maximale de la valeur de la propriété
  - nullable: (avec les valeurs false ou true) indique si la colonne accepte ou non des valeurs à NULL
  - unique: indique que la valeur de la colonne est unique.
- Exemple:

```
@Column (name ="name", nullable =false ,length = 50)
private String nom;
@Column (unique =true)
private int cin;
```

Annotation @Transient

- Un objet métier peut avoir des propriétés que l'on ne souhaite pas rendre persistantes dans la BD. Il faut alors impérativement les marquer avec l'annotation @Transient.
- L'annotation @Transient permet d'indiquer au gestionnaire de persistance d'ignorer cette propriété.
- Exemple:

```
@Transient
private String nom_prenom;
```

Annotation @Temporal

- L'annotation @Temporal permet de fournir des informations complémentaires sur la façon dont les propriétés encapsulant des données temporelles (Date et Calendar) sont associées aux colonnes dans la table (date, time ou timestamp).
- La valeur par défaut est timestamp.
- Exemple:

```
//prendre uniquement l'information du temps (heure:minute:seconde)
@Temporal(TemporalType.TIME)
private java.util.Date heureCapteur;
```

14:11:37

```
//prendre uniquement l'information du jour (année-mois-jour)
@Temporal(TemporalType.DATE)
private java.util.Date jourCapteur;
```

2025-03-06

```
//prendre l'information totale de la date (année-mois-jour  heure:minute:seconde)
@Temporal(TemporalType.TIMESTAMP)
private java.util.Date dateCapteur;
```

2025-03-06 14:11:37

Deux approches de persistance avec Spring

1. Approche classique avec EntityManager (DAO)

- ✓ Créer une classe DAO annotée @Repository.
- ✓ L'EntityManager est injecté par Spring (@PersistenceContext).
- ✓ Manipuler directement les entités et les transactions.

```
@Repository
public class ClientDao implements  ClientDaoInterface
{
      @PersistenceContext
      private EntityManager em;

      // persister l'entité
      @Transactional
      public void save(Client client) {
      em.persist(client);
      }

      // rechercher par clé primaire
      public Client find(Long id) {
      return em.find(Client.class, id);
      }

      public List<Client> findAll() {
      return em.createQuery("SELECT c FROM Client c",
      Client.class).getResultList();
      }
}
```

❑ Avantage:

contrôle total

❑ Inconvénients:

- beaucoup de code répétitif,
- moins pratique pour les grandes applications.

Deux approches de persistance avec Spring

2. Approche moderne avec Spring Data JPA (Repository)

- ✓ **Spring Data JPA** automatise les DAO.
- ✓ créer uniquement une interface,
- ✓ Spring génère l'implémentation à la compilation.
- ✓ Possibilité de définir des méthodes basées sur le nom ou avec **@Query**.

```
public interface ClientRepository extends
JpaRepository<Client, Long>
{
//méthode de recherche (optionnelle selon le besoin)
List<Client> findByNom(String nom);
}
```

❑ Avantages:

- très rapide,
- moins de code,
- pagination et tri intégrés

❑ Inconvénients:

- Moins de contrôle que EntityManager pour les requêtes complexes,
- mais @Query permet de l'étendre.

```
@Service
public class ClientService {
@Autowired
private ClientRepository clientRepo;

public void ajouterClient(String nom) {
    clientRepo.save(new Client(nom));
}
public List<Client> listerClients() {
    return clientRepo.findAll();
}
}
```

Approche1- EntityManager

- Toutes les actions de persistance sur les entités JPA sont réalisées grâce à un objet dédié de l'API : Il s'agit de EntityManager.
- Une instance de « EntityManager » est réalisée par injection de dépendance en spécifiant l'annotation @PersistenceContext
- Exemple:

```
@Repository
@Transactional
public class PersonneDaoImpl implements IPersonneDao{
//déclarer un objet « EntityManager »
@PersistenceContext
private EntityManager em;
........
}
```

- Un contexte de persistance (persistence context) est un ensemble d'entités géré par EntityManager.
- Exemple:

```
// référencer le contexte
ApplicationContext contexte=SpringApplication.run(JpaSpringBootApplication.class, args);
// Récupérer une implémentation de l'interface "IProduitDao" par injection de dépendance
IProduitDao daoProduit =  contexte.getBean(IProduitDao.class);
```

Approche1- Fonctionnalités de « EntityManager »

- EntityManager est donc au cœur de toutes les actions de persistance.
- EntityManager permet de réaliser des opérations CRUD (create, read, update, delete) sur les données.
- Elle permet aussi de rechercher des données (find).
- La méthode contains() de l'EntityManager permet de savoir si une instance fournie en paramètre est gérée par le contexte. Dans ce cas, elle renvoie true, sinon elle renvoie false.
- La méthode clear() de l'EntityManager permet de détacher toutes les entités gérées par le contexte.
- L'appel des méthodes de mise à jour persist(), merge() et remove() ne réalise pas d'actions immédiates dans la base de données sous-jacente,
- Il est possible de forcer l'enregistrement des mises à jour dans la base de données en utilisant la méthode flush() de l'EntityManager

Approche1- Utilisation de « EntityManager »

```java
@PersistenceContext
private EntityManager em;

public Personne save(Personne p)
{
     em.persist(p);
     return p;
}

public Personne findOne(Long id)
{
     Personne p = em.find(Personne.class, id);
     return p;
}

public Personne update(Personne p)
{
     em.merge(p);
     return p;
}

public void delete(Long id)
{
     Personne p = em.find(Personne.class, id);
     em.remove(p);
}
```

insertion

Recherche par clé primaire

Mise à jour

suppression

Approche1- Recherche par requête

- La recherche par requête repose sur des méthodes dédiées de la classe EntityManager (Exemple :createQuery) et sur un langage de requête spécifique nommé HQL (implémenté par Hibernate)
- HQL est un langage d'interrogation extrêmement puissant qui ressemble au SQL.
- HQL est totalement orienté objet, cernant des notions comme l'héritage, le polymorphisme et les associations.
- Les requêtes sont insensibles à la casse, à l'exception des noms de classes Java et des propriétés.

- Exemple:

Nom de l'attribut de la classe Java et non de la colonne de la table

```
public List<Personne> findAll()
{
     Query query= em.createQuery("select p from Personne p order by p.nom");
     return query.getResultList();
}
```

P est un alias pour référer à Personne dans la requête

Nom de la classe Java et non de la table

Approche1- Recherche par requête paramétrée

- L'objet Query gère aussi des paramètres nommés dans la requête.
- Le nom de chaque paramètre est préfixé par « : » dans la requête.
- La méthode setParameter() permet de fournir une valeur à chaque paramètre.
- Query fournit une méthode getResultList() qui renvoie une collection contenant les éventuelles occurrences retournées par la requête.
- Il est possible d'utiliser la méthode getSingleResult() pour obtenir un objet unique retourné par la requête.
- Exemple :

Paramètre nommé « x » préfixé par « : »

```
public List<Produit> findByDesignation(String mc)
{
Query query=
    em.createQuery("select p from Produit p where p.designation like :x");
    query.setParameter("x", "%"+mc+"%");

    return query.getResultList();
}
```

Affecter la valeur du paramètre « x »

Approche1- Mettre à jour une entité JPA

- Pour modifier une entité existante dans la base de données, il faut :
  - Obtenir une instance de l'entité à modifier (ou bien à travers une recherche sur la clé primaire ou l'exécution d'une requête) (find)
  - Modifier les propriétés de l'entité
  - Utiliser la méthode merge

```
public Produit update(Produit p, String nom)
{
    p.setNom(nom);
    em.merge(p);
    return p;
}
```

Mettre à jour l'entité

Appeler la méthode « merge »

```
public Personne update(int id , String nom)
{
    Personne p =(Personne) em.find(Personne.class, id);
    p.setNom(nom);
    em.merge(p);
    return p;
}
```

Recherche par clé primaire

Approche1- Supprimer une entité JPA

- Pour supprimer une entité existante dans la base de données, il faut :
  - ❖ Obtenir une instance de l'entité à modifier (ou bien à travers une recherche sur la clé primaire ou l'exécution d'une requête) (find)
  - ❖ Utiliser la méthode remove

```
public void delete(int id)
{
   Personne p = em.find(Personne.class, id);
   em.remove(p);
}
```

Recherche par clé primaire

Appeler la méthode « remove »

Approche1- Rafraîchir une entité JPA

- Pour rafraîchir une entité existante dans la base de données, il faut :
  - Obtenir une instance de l'entité à modifier (ou bien à travers une recherche sur la clé primaire ou l'exécution d'une requête) (find)
  - Utiliser la méthode refresh()

```
public void delete(int id)
{
   Personne p = em.find(Personne.class, id);
   em.refresh(p);
}
```

Recherche par clé primaire

Appeler la méthode « refresh »

Approche2- Principe de Spring Data JPA

- Spring Data JPA n'est pas une implémentation de JPA.
- Mais, c'est une abstraction fournie par Spring au-dessus de JPA (surcouche) pour simplifier encore plus la persistance.
- Spring Data JPA est une solution Spring qui utilise JPA (et donc Hibernate), mais en rajoutant une surcouche pour automatiser et simplifier.
- Elle permet d'éviter le code répétitif (EntityManager, DAO, Query) en introduisant le concept de Repository avec génération automatique des méthodes.
- Exemple avec Spring Data JPA :

```
public interface UserRepository extends JpaRepository<User, Long>
{
    List<User> findByEmail(String email);
}
```

→ Pas besoin d'écrire EntityManager ni de SQL ni de DAO manuellement: Il suffit de déclarer une interface Repositroy.

→ Spring Data JPA génère la requête automatiquement en utilisant JPA + Hibernate en dessous.

Approche2- Mise en œuvre avec Spring Boot

- **Pour utiliser Spring Data JPA, il est nécessaire de:**

1. Déclarer les dépendances : spring-boot-starter-data-jpa + driver (selon le type de la BD).
2. Configurer une datasource (dans application.properties).
3. Définir une entité JPA (classe annotée par @Entity).
4. Définir un Repository (interface qui étend JpaRepository ou CrudRepository).
5. Définir un service ou un contrôleur ou une classe principale qui consomme le repository.

Approche2- Mise en œuvre avec Spring Boot

1. Dépendances Maven

```
<!– dépendance de persistance Spring Data JPA->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<!– driver pour la BD MySQL-->
<dependency>
   <groupId>com.mysql</groupId>
   <artifactId>mysql-connector-j</artifactId>
   <scope>runtime</scope>
</dependency>
```

Approche2- Mise en œuvre avec Spring Boot

2. Configuration du fichier « application.properties »

```
#database Configuration:
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/spring-jpa?createDatabaseIfNotExist=true&useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=

#Hibernate Configuration:
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

- **Pour la base de données, spécifier :**
  - Le nom du driver (pilote de la BD)
  - L'URL de connexion à la base de données
  - Les paramètres d'accès à la BD (username et password)
- **Pour Hibernate, plusieurs valeurs sont possible pour la propriété « spring.jpa.hibernate.ddl-auto » :**
  - create : supprime et recrée les tables à chaque démarrage.
  - create-drop : comme create mais supprime en plus à l'arrêt de l'application.
  - update : met à jour le schéma sans détruire les données (le plus utilisé en dev).
  - validate : vérifie seulement la correspondance avec les entités

Approche2- Mise en œuvre avec Spring Boot

3. Définition d'une classe Persistante (entité)

```
@Entity
public class Produit implements Serializable {
    @Id
    @GeneratedValue
    private Long id;

    @Column (length=50)
    private String designation;


    private double prix;
    private int quantite;

      // setters et getters
}
```

Approche2- Mise en œuvre avec Spring Boot

4. Définition d'un repository

```
public interface ProduitRepository extends
JpaRepository<Produit, Long>
{
   // méthode personnalisée (optionnelle)
   List<Produit> findByDesignation(String designation);
}
```

- Pas besoin d'implémentation: Spring Data JPA génère tout seul les méthodes:
  - save(),
  - findById(),
  - findAll(),
  - deleteById(),
  - etc.
- Il est possible de déclarer des méthodes personnalisées comme findByDesignation (basée sur la designation).

Approche2- Mise en œuvre avec Spring Boot

5. Définition d'une classe de consommation

```
@SpringBootApplication
public class MainApp {
     // Déclarer une référence de l'interface " ProduitRepository"
     static ProduitRepository produitRepos ;
     public static void main(String[] args) {
     // référencer le contexte
     ApplicationContext contexte = SpringApplication.run(MainApp.class, args);
     // Récupérer une implémentation de l'interface "ProduitRepository" par injection de dépendance
     produitRepos =contexte.getBean(ProduitRepository.class);
     // Insérer 3 produits
     Produit p1 =new Produit("Yaourt", 0.500, 20);
     Produit p2 =new Produit("Chocolat", 2000.0, 5);
     Produit p3 =new Produit("Panier", 1.200, 30);
     produitRepos.save(p1);
     produitRepos.save(p2);
     produitRepos.save(p3);
     //Afficher la liste des produits
     List<Produit> lp = produitRepos.findAll();
     for (Produit p : lp) {
          System.out.println(p);
     }
     }
}
```

Personnalisation des requêtes avec Spring Data JPA

- Spring Data JPA permet de gérer la persistance des données à partir d'interfaces Repository.
- Par défaut, il fournit les opérations CRUD (Create, Read, Update, Delete).
- Mais dans les projets réels, on a souvent besoin de requêtes plus spécifiques (exemple : trouver tous les étudiants dont le nom commence par "Mo").
- Pour cela, Spring Data JPA offre plusieurs moyens de personnaliser les requêtes :
  - Utilisation directe de l'API EntityManager : à travers la méthode « createQuery »).
  - Requêtes JPQL: Java Persistence Query Language (ou HQL: Hibernate Query Language) avec @Query: langage orienté entités qui garde l'avantage d'une syntaxe orientée objet.
  - Requêtes SQL natives: possibilité d'utiliser des fonctions propres au SGBD, vues, procédures stockées.
  - Méthodes requêts (query methods) : génèrer automatiquement les requêtes à partir du nom des méthodes.

Exemples de personnalisation des requêtes

- Avec EntityManager :

```
@Repository
public class StudentDAO
{
    @PersistenceContext
    private EntityManager em;

    // récupérer la liste des étudiants dont le nom commence par la valeur passée en argument
    public List<Student> searchByLastname(String lastname)
    {
     Query query=em.createQuery("SELECT s FROM Student s WHERE s.lastname = :lastname",
    Student.class);

    query.setParameter("lastname", lastname);

    return query.getResultList();
    }
}
```

Exemples de personnalisation des requêtes

- Avec JPQL (Java Persistence Query Language) en utilisant @Query :

```
public interface StudentRepository extends JpaRepository<Student, Long> {
@Query("SELECT s FROM Student s WHERE s.lastname = :lastname")
List<Student> findStudentsByLastname(@Param("lastname") String lastname);
}
```

- Avec SQL natif en utilisant @Query :

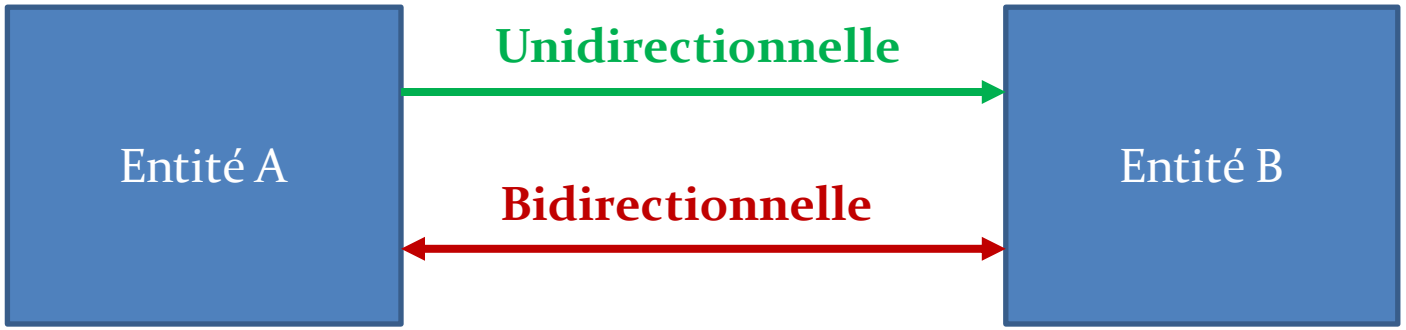
```
public interface StudentRepository extends JpaRepository<Student, Long> {
@Query(value = "SELECT * FROM student WHERE firstname LIKE %:prefix%", nativeQuery = true)
List<Student> findStudentsNative(@Param("prefix") String prefix);
}
```

- Avec Query Method:

```
public interface StudentRepository extends JpaRepository<Student, Long> {
List<Student> findByLastname(String lastname);
List<Student> findByFirstnameAndLastname(String firstname, String lastname);
}
```

Gestion des relations entre les entités

- Dans le modèle des bases de données relationnelles, les tables peuvent être liées entre elles grâce à des relations (ou associations)
- Les relations peuvent avoir différentes cardinalities :
  - 1-1 (one-to-one)
  - 1-N (one-to-many)
  - N-1 (many-to-one)
  - N-N (many-to-many)
- Chacune de ces relations peut être unidirectionnelle ou bidirectionnelle sauf one-to-many et many-to-one qui sont par définition bidirectionnelles.
- Dans le cas unidirectionnel, l'une des deux entités doit être maître et l'autre esclave,
- Dans les deux cas « 1-N » et « N-1 », l'entité du côté 1 est l'entité esclave.



Relation 1-1 unidirectionnelle

- Une « Personne » possède une seule « Identite »et une « Identite » ne peut être relative qu'à une seule « Personne ».
- Une « Personne » peut consulter son idendité et le sens inverse n'est pas permis.
- Nous avons donc bien une relation 1-1 unidirectionnelle

L'entité « Identite » ne peut pas connaître la personne y associée

- Dans ce cas, « Personne » est l'entité maître, donc elle maintient la relation:

```
@Entity
@Table(name="Person")
public class Personne
{
	@Id
	@GeneratedValue (strategy = GenerationType.AUTO)
	private int matricule;
	private String nom;
	private String prenom;

	@Transient
	private String nom_prenom;

	@OneToOne
	private Identite identite;
	……
```

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

#	Nom	Type	Interclassement
1	matricule	int(11)	
2	nom	varchar(255)	latin1_swedish_ci
3	prenom	varchar(255)	latin1_swedish_ci
4	identite_id	int(11)	

Clé étrangère

L'annotation **@OneToOne** associé à l'attribut **identite** sera converti dans la table « **Person** » en une clé étrangère « **identite_id** » référençant la colonne **id**(clé primaire) de la table « **Identite** »

Utilisation d'une relation 1-1 unidirectionnelle

```
public Personne save (String nom, String prenom, int cin,  String adresse)
{
    Identite i = new Identite ( cin, adresse);
    em.persist(i);
    Personne p = new Personne(nom, prenom);
    p.setIdentite(i);
    em.persist(p);
    return p;
}

public Personne updateAdressePersonne(int matriculePersonne, String adresse)
{
    Personne p = em.find(Personne.class, matriculePersonne);
    p.getIdentite().setAdresse(adresse);
    return p;
}
public Identite updateAdresseIdentite(int idIdentite, String adresse)
{
Identite i= em.find(Identite.class, idIdentite);
i.setAdresse(adresse);
em.merge(i);
return i;
}
```

Insérer tout d'abord l'entité « esclave »

Affecter la valeur de l'entité « esclave »

Référencer l'entité « maître » par clé primaire puis appeler l'entité « esclave » pour réaliser la modification

Référencer l'entité « esclave » par clé primaire puis réaliser la modification et appeler la méthode « merge »

Relation 1-1 bidirectionnelle

- Chaque entité peut accéder à l'autre (à travers un accesseur getter)

- Toujours l'entité « maitre » est la propriétaire de la relation, elle présente un attribut transformé en une clé étrangère dans la BD
- L'entité esclave doit préciser un champ retour par une annotation @OneToOne et un attribut mappedBy qui doit référencer le champ qui porte la relation côté maître.
- Ce champ ne génère pas une clé étrangère, mais permet de une requête est lancée sur la base pour réaliser une jointure

```java
@Entity
@Table(name="Person")
public class Personne
{
	@Id
	@GeneratedValue (strategy = GenerationType.AUTO)
	private int matricule;
	private String nom;
	private String prenom;

	@Transient
	private String nom_prenom;

	@OneToOne
	private Identite identite;
	……
```

```java
@Entity
public class Identite {

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

private int cin;

private String adresse;

@OneToOne (mappedBy = "identite")
private Personne personne;
```

référencer la relation dans la classe « Personne »

Relation 1-N et N-1

- Une « Personne » possède un ou plusieurs « Compte » et un « Compte » ne peut être relatif qu'à une seule « Personne ». Nous avons donc bien une relation 1-N

- Dans ce cas, « Personne » est l'entité « esclave » et présente l'annotation « @OneToMany ».
- De l'autre côté, la classe « Compte » est l'entité « maître » qui maintient la relation avec l'annotation « @ManyToOne » et qui contient un champ transformé en clé étrangère dans la BD.
- L'entité « Personne » présente une « Collection » de type « Compte »

```java
@Entity
@Table(name="Person")
public class Personne
{
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private int matricule;
private String nom;
private String prenom;

@OneToMany (mappedBy = "personne")
private Collection <Compte> comptes = new
ArrayList<Compte>();
```

référencer la relation dans la classe « Compte »

```java
@Entity
public class Compte
	{@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int id;
	private int code;
	private double solde;
	@ManyToOne
	private Personne personne;
```

Clé étrangère

#	Nom	Type
1	id	int(11)
2	code	int(11)
3	solde	double
4	personne_matricule	int(11)

Relation N-N unidirectionnelle

- Une « Personne » réalise un ou plusieurs « Vol » et un « Vol » regroupe une ou plusieurs « Personne ». Nous avons donc bien une relation N-N
- Dans le cas d'une relation unidirectionnelle (@ManyToMany) maintenue par l'entité « Personne », Une « Personne » peut accéder à ses vols mais un « Vol » ne peut déterminer les personnes associées.

- La façon classique d'enregistrer ce modèle en base consiste à créer une table de jointure « personne_vols » qui comporte deux clés étrangères:
  - personne_matricule : référence la table « Personne »
  - vols_id : référence la table « Vol »

```
@Entity
public class Personne
{
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private int matricule;
private String nom;
private String prenom;


@ManyToMany
private Collection<Vol> vols = new ArrayList<Vol>();
```

Etablir une relation (N-N) dans la classe « Vol »

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

Relation N-N bidirectionnelle

- Une « Personne » réalise un ou plusieurs « Vol » et un « Vol » regroupe une ou plusieurs « Personne ». Nous avons donc bien une relation N-N

- Dans le cas d'une relation bidirectionnelle (@ManyToMany) maintenue par l'entité « Personne », Une « Personne » peut accéder à ses vols et un « Vol » peut déterminer les personnes associées.
- On ajoute une autre annotation (@ManyToMany) dans l'entité « Vol » sur une collection de « Personne » et en utilisant l'attribut « mappedBy » pour référence la relation dans l'entité « Personne »

```java
@Entity
public class Personne
{
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private int matricule;
private String nom;
private String prenom;


@ManyToMany
private Collection<Vol> vols = new ArrayList<Vol>();
```

Référencer la relation dans la classe « Personne »

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

@ManyToMany (mappedBy = "vols")
private Collection<Personne> presonnes = new
ArrayList<Personne>();
```

Comportement en cascade

- Le comportement cascade consiste à spécifier ce qui se passe pour une entité en relation d'une entité mère lorsque cette entité mère subit une des opérations définies ci-dessus.
- Le comportement cascade est précisé par l'attribut cascade, disponible sur les annotations : @OneToOne, @OneToMany et @ManyToMany (et non pour @ManyToOne)
- La valeur de l'attribut est une énumération de type CascadeType ayant les principales valeurs suivantes:
  - MERGE : cascade en cas de « merge »
  - PERSIST : cascade en cas de « persist »
  - REMOVE: cascade en cas de « remove »
  - ALL : correspond à toutes les valeurs à la fois.

```
@Entity
public class Personne
{
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private int matricule;
private String nom;
private String prenom;

@OneToOne (cascade ={CascadeType.PERSIST,
CascadeType.REMOVE, CascadeType.MERGE})
private Identite identite;
```

```
@Entity
public class Personne
{
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private int matricule;
private String nom;
private String prenom;

@OneToOne (cascade =CascadeType.ALL)
private Identite identite;
```

Mode de récupération d'une collection

- Pour récupérer les élément d'une collection contenu dans une entité , Nous disposons de deux modes:
  - EAGER: effectuer la récupération des éléments de la collection, dès que l'on récupère l'objet et donc on initialise la collection. C'est le Fetch Type "eager" (fetch=FetchType.EAGER).
  - LAZY: (par défaut) effectuer la récupération des éléments de la collection à la demande, c'est à dire dès que l'on aura besoin de la collection. C'est le Fetch Type "lazy" (fetch=FetchType.LAZY).
  - Le mode « LAZY » est le mode recommandé pour ne pas faire des requêtes inutiles vers la base de données surtout en cas de non besoin d'utiliser la collection

```java
@Entity
@Table(name="Person")
public class Personne
{
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private int matricule;
private String nom;
private String prenom;

@OneToMany (mappedBy = "personne« , fetch=FetchType.EAGER)
private Collection <Compte> comptes = new ArrayList<Compte>();
```

Charger la la collection des comptes au moment du chargement de l'entité « Personne »

Héritage et JPA

- Les bases de données relationnelles n'ont pas une fonctionnalité permettant de traduire ou mapper une hiérarchie de classe en tables de bases de données
- L'héritage est un des concepts clefs en java en particulier et de la programmation orientée objet en général. JPA et ses différentes implémentations avec doivent donc fournir un moyen pour traduire ce concept clef en un concept compréhensible par les bases de données relationnelles.
- JPA propose plusieurs stratégies :
  - **MappedSuperclass** : les classes parentes ne sont pas des entités.
  - **Single Table** : les entités de la hiérarchie de classe sont placées dans une seule table.
  - **Joined Table** : chaque classe a sa table et effectuer une requête sur une sous-classe de la hiérarchie implique de faire une jointure sur les tables.
  - **Table-Per-Class** : une table par classe.
- Chacune des stratégies implique une structure différente de la base de données.

MappedSuperclass

- Cette stratégie permet de partager les propriétés entre plusieurs entités.
- Elle permet de mapper chaque classe vers une table dédiée.
- La classe mère sur laquelle est définie la stratégie **MappedSuperclass** n'est pas une entité et aucune table ne sera créée dans la BD pour elle..

```
@MappedSuperclass
public class Personne {

@Id
protected long Id;
protected String nom;
protected String prenom;

// constructeur, getters, setters
}
```

```
@Entity
public class Formateur extends Personne {

private String matiere;

// constructeur, getters, setters
}
```

```
@Entity
public class Etudiant extends Personne {

private double note;

// constructeur, getters, setters
}
```

- Dans la BD, on aura une table **Etudiant** et une table **Formateur** qui en plus de leurs propriétés auront les propriétés de la classe mère comme champs.
- Avec la stratégie **MappedSuperclass**, les classes mères ne peuvent pas définir de relations avec d'autres entités.

Single Table

- C'est la stratégie par défaut utilisée par JPA lorsqu'aucune stratégie n'est implicitement définie et que la classe mère de la hiérarchie est une entité.
- Avec cette stratégie, une seule table est créée et partage par toutes les classes de la hiérarchie.
- L'annotation @Inheritance est utilisée sur la classe mère pour préciser à JPA la stratégie d'héritage à utiliser.
- Dans cette stratégie, toutes les classes entités sont mappées dans une unique table.
- JPA a besoin de faire la différence entre les différentes lignes de la table ainsi mappée afin de pouvoir convertir chaque enregistrement vers la classe entité correspondante.
- Pour ce faire, JPA utilise un mécanisme permettant de faire cette différence en créant une colonne appelée discriminator qui ne fait pas partie des attributs de l'entité mappée.

```
@Entity
@Inheritance(strategy=
InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_PERSONNE")
public class Personne {

@Id
protected long Id;
protected String nom;
protected String prenom;

// constructeur, getters, setters
}
```

Single Table

- Une colonne sera créé dans la table générée ayant pour valeur TYPE_PERSONNE pour différencier les enregistrements des entités Etudiant et Formateur.
- Par défaut, les valeurs de la colonne TYPE_PERSONNE seront les noms des classes filles.
- Pour préciser le nom à enregistrer, on ajoute l'annotation @DiscriminatorValue sur les classes filles.

```
@Entity
@DiscriminatorValue("etu")
public class Etudiant extends Personne
{

private double note;

// constructeur, getters, setters
}
```

```
@Entity
@DiscriminatorValue("form")
public class Formateur extends Personne
{

private String matiere;

// constructeur, getters, setters
}
```

JOINED

- Cette stratégie consiste à enregistrer les champs de chaque entité dans une table propre à cette classe.
- On a donc autant de tables que de classes dans notre modèle, abstraites ou concrètes.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "TYPE_PERSONNE")
public class Personne {

@Id
protected long Id;

protected String nom;

protected String prenom;

// constructeur, getters, setters
}
```

JOINED

```
@Entity
@DiscriminatorValue("etu")
public class Etudiant extends Personne {

private double note;

// constructeur, getters, setters
}
```

```
@Entity
@DiscriminatorValue("form")
public class Formateur extends Personne {

private String matiere;

// constructeur, getters, setters
}
```

- Enfin on remarque que chacune des deux tables Etudiant et Formateur comporte une clé primaire: id.
- Cette clé primaire est aussi une clé étrangère qui référence la clé primaire de la table Personne.
- Les trois tables partagent en fait la même clé primaire, ce qui est logique puisque toutes les lignes de la tables Etudiant ont une partie de leurs champs dans la table Personne et de même pour la table Formateur

TABLE_PER_CLASS

- Cette stratégie fonctionne à l'inverse de la stratégie SINGLE_TABLE. Plutôt que d'envoyer tous les champs de toutes les entités vers une table unique, on les envoie vers autant de tables qu'il y a de classes concrètes annotées @Entity dans la hiérarchie..

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@DiscriminatorColumn(name = "TYPE_PERSONNE")
public class Personne {

@Id
protected long Id;
protected String nom;
protected String prenom;
// constructeur, getters, setters
}
```

TABLE_PER_CLASS

```java
@Entity
@DiscriminatorValue("etu")
public class Etudiant extends Personne {

private double note;

// constructeur, getters, setters
}
```

```java
@Entity
@DiscriminatorValue("form")
public class Formateur extends Personne {

private String matiere;

// constructeur, getters, setters
}
```

- On a donc autant de tables dans notre schéma que de classes concrètes annotées Entity dans la hiérarchie
- On remarque que les tables Personne, Etudiant et Formateur sont toujours présentes, c'est leur contenu qui change.
- Chacune des deux tables Etudiant et Formateur comporte maintenant deux colonnes nom et prenom, elle deviennent indépendantes de la table Personne. De fait, sa clé primaire n'est plus une clé étrangère.