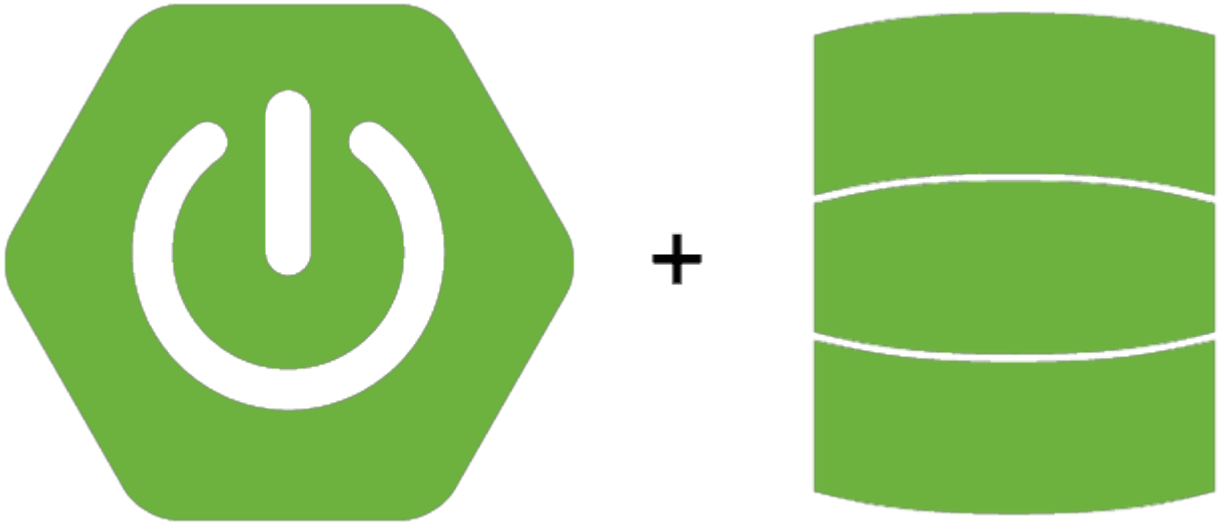
Spring Boot

semifir



Spring boot data

`Spring boot data` permet d'accéder à des bases de données de manière simple.

Cette dépendance est basée sur `Spring Data`.

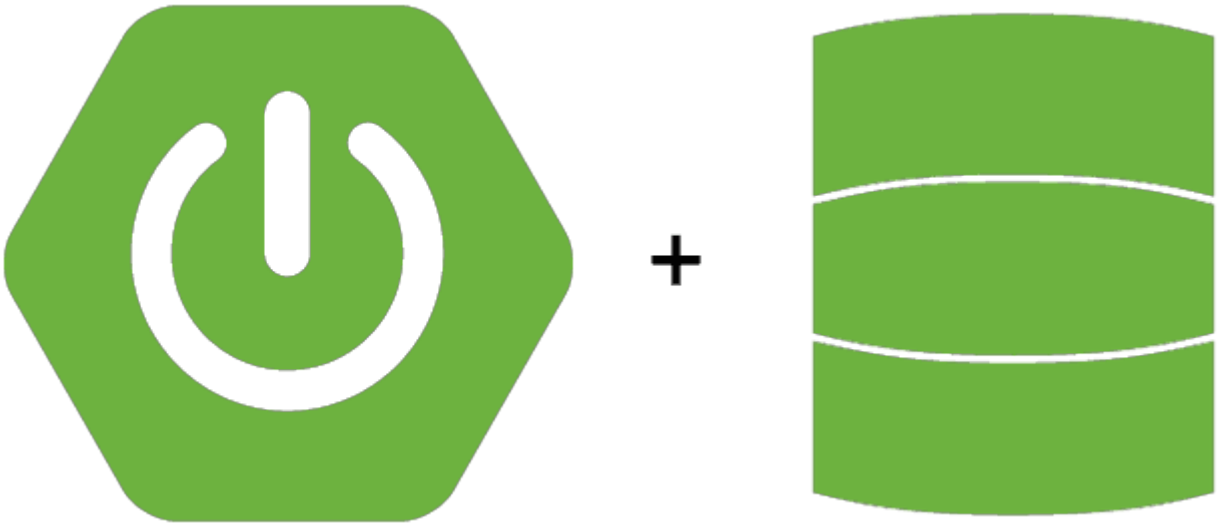
Les entités

`Spring boot data` permet de créer des entités de manière simple.

> Ici un exemple avec une entité SQL

```
@Entity // entité JPA
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id; // Primary key du nom d'id
    private String name;
    private String email;
    private String phone;
    private String address;
    private String city;
    private String zipCode;
    private String country;
}
```

Spring Data JPA



Spring Data JPA

Spring Data JPA est un module de Spring Data qui permet d'accéder à des bases de données relationnelles en utilisant la norme JPA de JEE.

Qu'est ce que JPA ?

Java Persistence API (JPA) est une norme Java qui permet de gérer les entités.

Ce n'est qu'une norme, il faut donc un framework pour l'implémenter.

Le framework le plus connu est `Hibernate`.

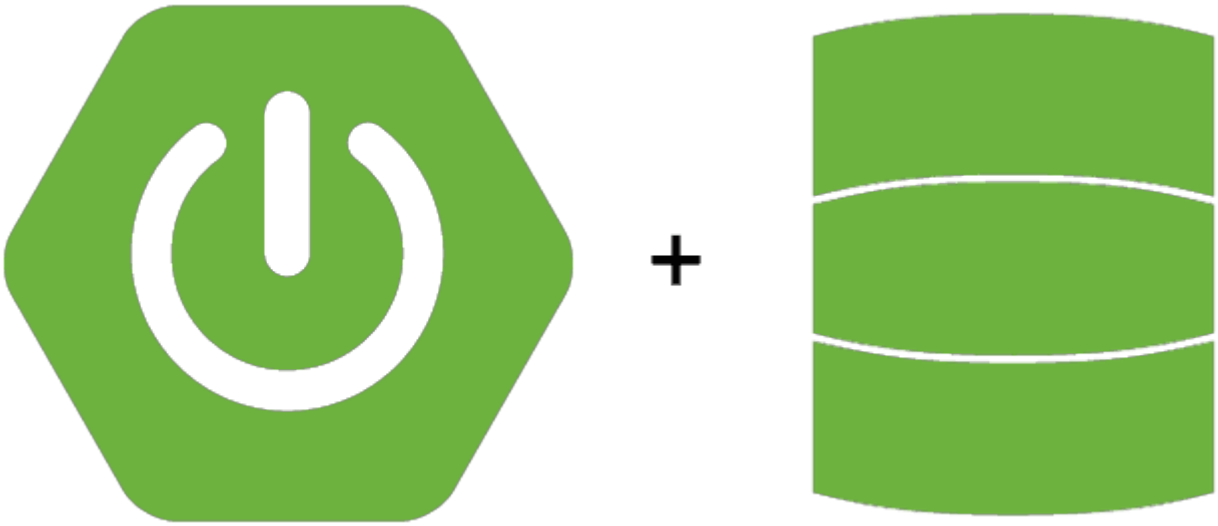
Qu'est ce que Hibernate ?

`Hibernate` est un ORM (Object Relational Mapping) qui permet de gérer les entités.

Spring Boot Data JPA

Spring Boot + JPA + Hibernate = Spring Boot Data JPA

Installation de Spring Boot Data JPA



Spring Data JPA dans maven

Pour utiliser Spring Data JPA, il faut ajouter la dépendance suivante dans le fichier `pom.xml`.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Il est nécessaire d'ajouter le driver JDBC de votre base de données

Exemple avec le connecteur JDBC de `MySQL`.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Installation avec Gradle

Pour vos projets en gradle, il faut ajouter la dépendance suivante dans le fichier `build.gradle`.

```
// Spring Data JPA
compile('org.springframework.boot:spring-boot-starter-data-jpa')
// MySQL
```

```
compile('mysql:mysql-connector-java')
```

Configuration

Pour communiquer avec la Base de données, il faut configurer les paramètres de connexion dans un nouveau fichier `application.properties` ou `application.yml`.

Ces deux types de fichiers servent à configurer l'application, la différence est le format.

Application.properties

C'est un fichier de configuration au format `clé=valeur`.

```
spring.datasource.url= mon url de base de données
spring.datasource.username= l'utilisateur de la base de données
spring.datasource.password= le mot de passe de la base de données
spring.datasource.driver-class-name= nom du Driver utilisé
# pas obligatoire car Spring peux le deviner grace au pom.xml
...
```

Application.yml

C'est un fichier de configuration au format `YAML`.

```
spring:
  datasource:
    url: mon url de base de données
    username: l'utilisateur de la base de données
    password: le mot de passe de la base de données
    driver-class-name: nom du Driver utilisé
    ...
```

Configuration

Il existe beaucoup d'autre paramètres de configuration, vous pouvez les trouver dans la documentation de Spring Boot

---

Configuration MySQL

Dans notre cas nous devons configurer le driver JDBC dans le fichier `application.properties`

```
spring.datasource.url=jdbc:mysql://localhost:3306/spring_data_jpa?
connectionTimeZone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Configuration MySQL

- `url` sert à donner le lien vers la base de donnée
- `username` & `password` permettent de stocker les variable d'environnements pour l'utilisateur
- `driver-class-name` est le nom du driver utilisé pour communiquer avec la base de données
- `ddl-auto` permet de choisir le mode de génération de la base de données
- `show-sql` permet d'afficher les requêtes SQL dans la console

Configuration PostGreSQL

```
spring.datasource.url=jdbc:postgresql://localhost:5432/spring_data_jpa
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=update
```

Configuration en YML

Si vous utiliser le format YML, il faut configurer le driver JDBC de la manière suivante.

> Vous pouvez juste renommé le fichier `application.properties` en `application.yml`

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/spring_data_jpa?connectionTimeZone=UTC
    username: root
    password: ****
    driver-class-name: com.mysql.cj.jdbc.Driver
```

Configuration en YML pour PostGreSQL

```
spring:
  datasource:
```

```
    url: jdbc:postgresql://localhost:5432/spring_data_jpa
    username: root
    password: root
    driver-class-name: org.postgresql.Driver
```

Configuration de Spring Data JPA

`Spring Data JPA` utilise `Hibernate` pour la gestion des entités.

Il est possible de configurer `Hibernate` pour choisir le mode de génération de la base de données.

```
spring:
  ...
  jpa:
    hibernate:
      ddl-auto: create
```

Ddl-auto

La configuration `ddl-auto` permet de choisir le mode de génération de la base de données.

- `create` : la recrée à chaque démarrage de l'application
- `create-drop` : supprime la base de données et la recrée à chaque démarrage de l'application
- `update` : met à jour la base de données à chaque démarrage de l'application
- `validate` : valide la base de données à chaque démarrage de l'application
- `none` : aucune action

Création d'une entité

Une `entité` est une classe qui représente une table de la base de données.

Elle DOIT avoir un identifiant unique (id) qui est une clé primaire.

Création d'une entité

Pour définir une entité, il faut utiliser l'annotation `@Entity`.

```
@Entity
public class User {
    // Reste de la classe
}
```

@Id

L'annotation `@Id` permet de définir le champs qui représente l'identifiant de l'entité.

Pour que l'identifiant soit généré automatiquement, il faut utiliser l'annotation `@GeneratedValue`.

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    // autres champs
    // getters and setters
}
```

@GeneratedValue

L'annotation `@GeneratedValue` permet de définir la stratégie de génération de l'identifiant.

- `AUTO` : utilise le type de base de données
- `IDENTITY` : utilise l'auto-increment de la base de données
- `SEQUENCE` : utilise une séquence de la base de données
- `TABLE` : utilise une table de la base de données

@Table

L'annotation `@Table` permet de définir le nom de la table.

> Si l'annotation `@Table` n'est pas utilisée, le nom de la table sera le nom de la classe en minuscule.

```
@Entity
@Table(name = "users")
public class User {
    // ...
}
```

@Column

L'annotation `@Column` permet de définir les propriétés de la colonne:

- `name` : le nom de la colonne
- `nullable` : si la colonne peut être null
- `unique` : si la colonne doit être unique

```
@Entity
public class User {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "email", nullable = false, unique = true)
    private String email;
    @Column(name = "name")
    private String name;
    @Column(name = "date")
    private Date date;
    // getters and setters
}
```

Attention !!

Il est important que votre classe d'entité soit un `POJO` (Plain Old Java Object):

- Constructeur sans paramètre
- Getters et Setters

> Sans les getters et setters, `Spring Data JPA` ne pourra pas fonctionner.

> Sans le constructeur sans paramètre, `Hibernate` ne pourra pas générer les entités.

@Transient

L'annotation `@Transient` permet de définir un champs qui ne sera pas persisté.

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "email")
    private String email;
    @Column(name = "name")
    private String name;
    @Column(name = "date")
    private Date date;
    @Transient
    private String password;
    // getters and setters
}
```

@Temporal

L'annotation `@Temporal` permet de définir le type de la date.

```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "email")
    private String email;
    @Column(name = "name")
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name = "date")
    private Date date;
    // getters and setters
}
```

Les jointures

Spring Data JPA permet de faire des jointures entre les entités.

Il existe 3 types de jointures :

- `@OneToOne`
- `@OneToMany`
- `@ManyToOne`

@OneToOne

L'annotation `@OneToOne` permet de faire une jointure entre deux entités.

```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "email")
    private String email;
    @Column(name = "name")
    private String name;
    @OneToOne
    private Address address;
    // getters and setters
}

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

```
    @Column(name = "street")
    private String street;
    @Column(name = "city")
    private String city;
    @Column(name = "country")
    private String country;
    // getters and setters
}
```

@OneToMany

L'annotation `@OneToMany` permet de faire une jointure entre deux entités.

```
@Entity
public class Musicien {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @OneToMany
    private List<Instrument> instruments;
    // getters and setters
}

@Entity
public class Instrument {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    // getters and setters
}
```

OneToMany

Ici un `Musicien` peut avoir plusieurs `Instrument`.

@ManyToOne

L'annotation `@ManyToOne` permet de faire une jointure entre deux entités.

```
@Entity
public class Voiture {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @ManyToOne
    private Marque marque;
    // getters and setters
}

@Entity
public class Marque {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    // getters and setters
}
```

ManyToOne

Ici une `Voiture` peut avoir une seule `Marque`.

Une `Marque` peut avoir plusieurs `Voiture`.

@ManyToMany

L'annotation `@ManyToMany` permet de faire une jointure entre deux entités.

```
@Entity
public class Formateur {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @ManyToMany
    private List<Formation> formations;
    // getters and setters
}

@Entity
public class Formation {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    // getters and setters
}
```

ManyToMany

Ici un `Formateur` peut avoir plusieurs `Formation`.

Une `Formation` peut avoir plusieurs `Formateur`.

@JoinTable

L'annotation `@JoinTable` permet de définir la table de jointure pour les jointures `@ManyToMany`.

```
@Entity
public class Formateur {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @ManyToMany
    @JoinTable(name = "formateur_formation")
    private List<Formation> formations;
    // getters and setters
}
```

@JoinTable

L'annotation `@JoinTable` permet de définir la colonne de jointure pour les jointures `@ManyToMany`.

Elle possède 3 paramètres :

- `name` : le nom de la table de jointure
- `joinColumns` : la colonne de jointure de la table courante
- `inverseJoinColumns` : la colonne de jointure de la table liée

@JoinTable Exemple

```
@Entity
public class Voiture {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @ManyToMany
    @JoinTable(
```

```
        name = "voiture_personne",
        joinColumns = @JoinColumn(name = "voiture_id"),
        inverseJoinColumns = @JoinColumn(name = "personne_id")
    )
    private List<Personne> personnes;
    // getters and setters
}
```

Bi-directional ManyToMany

Il est possible de faire une jointure bidirectionnelle avec `@ManyToMany`.

```
@Entity
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @ManyToMany
    @JoinTable(
        name = "personne_voiture",
        joinColumns = @JoinColumn(name = "personne_id"),
        inverseJoinColumns = @JoinColumn(name = "voiture_id")
    )
    private List<Voiture> voitures;
    // getters and setters
}

@Entity
public class Voiture {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @ManyToMany(mappedBy = "voitures")
    private List<Personne> proprietaires;
    // getters and setters
}
```

Attention!!

Les relations bidirectionnelles sont à utiliser avec précaution.

Elles peuvent être source de problème de performance et de complexité.

Elle ne sont pas réellement bidirectionnel car pour la persistance, il faut toujours utiliser l'entité persisté qui possède la relation (Celle avec `@JoinTable`).

Attention!!

Dans le cas des voitures et des propriétaires, pour sauvegarder la relation, il faut sauvegarder le propriétaire.

Sauvegarder la voiture ne sauvegardera pas la relation.

Exemples de problème lié à la bidirectionnalité

Si vous avez une relation bidirectionnelle représentant les "amis" d'une personne, quand vous récupérez une personne, vous récupérez aussi tous ses amis.

Ainsi que tous les amis de ses amis.

... et leurs amis.

Problème de sérialisation

Si vous avec une relation bidirectionnelle entre les voitures et leurs propriétaires, vous aurez un problème de sérialisation.

```
{
    "id": 1,
    "name": "Renault",
    "proprietaires":[
```

```
        {
            "id": 1,
            "name": "Jean",
            "voitures":[
                {
                    "id": 1,
                    "name": "Renault",
                    "proprietaires":[
                        {
                            "id": 1,
                            "name": "Jean",
                            "voitures":[
                                {
                                    "id": 1,
                                    "name": "Renault",
                                    "proprietaires":[
                                        {
                                            "id": 1,
                                            "name": "Jean",
                                            "voitures":[
                                                {
                                                    "id": 1,
                                                    "name": "Renault",
                                                    "proprietaires":[
                                                        {
                                                            "id": 1,
                                                            "name": "Jean",
                                                            "voitures":[
                                                                ...
```

@OrderBy

L'annotation `@OrderBy` permet de définir l'ordre des résultats.

```java
@Entity
public class Musicien {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @OneToMany
    @OrderBy("name")
    private List<Instrument> instruments;
    // getters and setters
}
```

@Cascade

L'annotation `@Cascade` permet de définir le comportement de cascade.

```
@Entity
public class Musicien {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @OneToMany
    @Cascade(CascadeType.ALL)
    private List<Instrument> instruments;
    // getters and setters
}
```

@Cascade

Ici, si on supprime un `Musicien`, tous les `Instrument` associés seront supprimés.

Il existe 4 types de cascade :

- `CascadeType.ALL` : toutes les opérations
- `CascadeType.PERSIST` : création
- `CascadeType.MERGE` : mise à jour
- `CascadeType.REMOVE` : suppression

Exercice 3

Les repositories

Un `repository` est une interface qui permet d'effectuer des opérations sur une base de données.

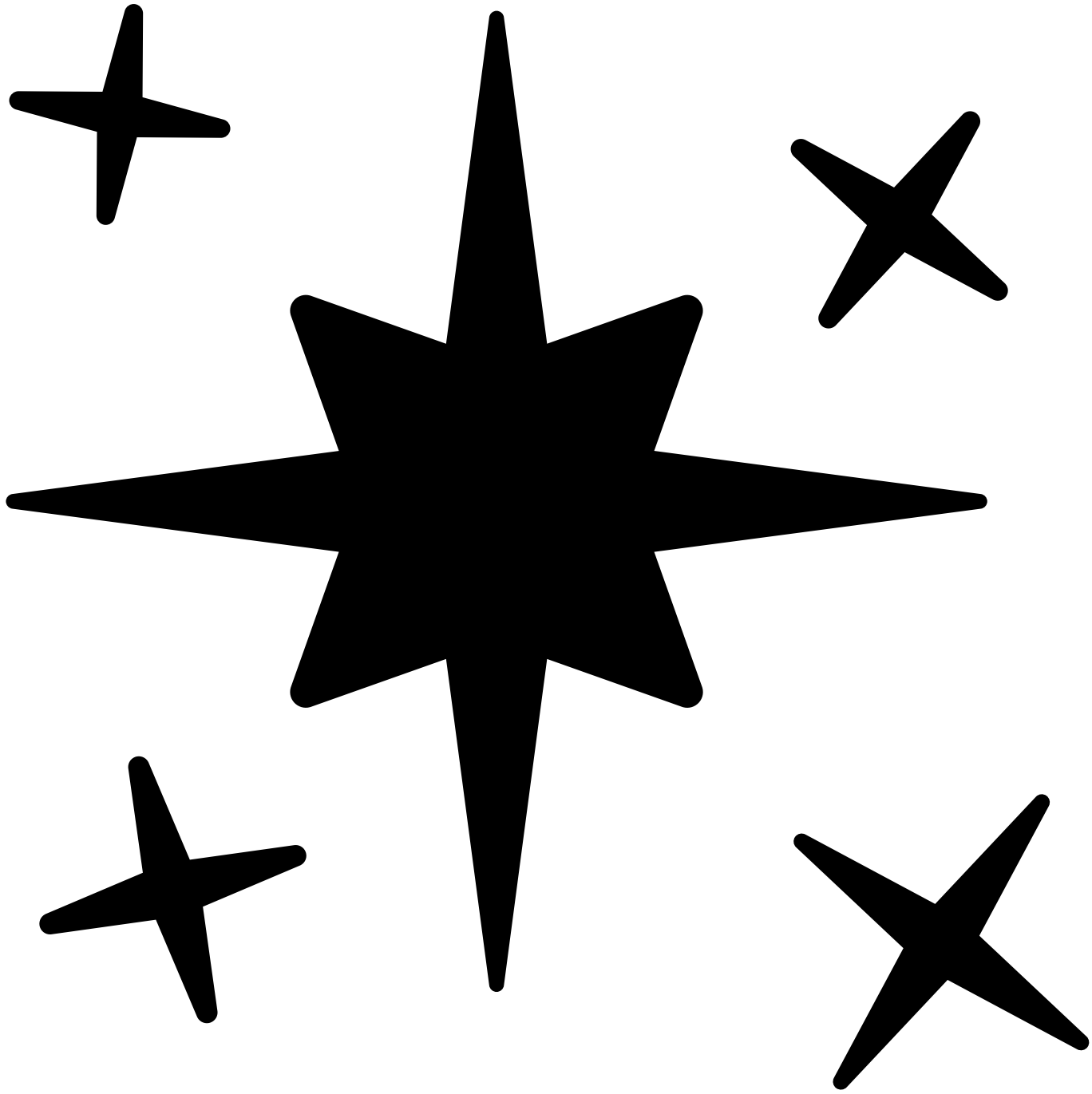
C'est elle qui dialogue avec la base de données.

Les repositories en Spring boot

En spring boot, il est possible de créer des `repositories` de plusieurs façons:

- En créant une classe ayant une annotation `@Repository`
- En créant une interface qui étend une interface de `Spring Data`

Les repositories de Spring Data



Les repositories de Spring Data

`Spring Data` fournit des interfaces qui permettent d'effectuer des opérations sur une base de données.

Voici quelques interfaces :

- `CrudRepository` : pour effectuer des opérations CRUD
- `PagingAndSortingRepository` : pour effectuer des opérations de pagination et de tri
- `JpaRepository` : pour effectuer des opérations CRUD et de pagination

CrudRepository

`CrudRepository` permet d'effectuer des opérations CRUD.

Il prend deux paramètres génériques :

- `T` : le type de l'entité
- `ID` : le type de l'id de l'entité

```
public interface PersonRepository extends CrudRepository<Person, Long> {}
```

CrudRepository

ici le repository contient les méthodes suivantes :

- `insert` : pour insérer une entité (`POST`)
- `save` : pour sauvegarder une entité (`PUT`)
- `findAll` : pour récupérer toutes les entités (`GET`)
- `findById` : pour récupérer une entité par son id (`GET`)
- `deleteById` : pour supprimer une entité par son id (`DELETE`)
- `deleteAll` : pour supprimer toutes les entités (`DELETE`)

JpaRepository

`JpaRepository` permet d'effectuer des opérations CRUD avec quelques méthodes supplémentaires.

> C'est une interface qui étend `CRUDRepository`

```
public interface PersonRepository extends JpaRepository<Person, Long> {
}
```

PagingAndSortingRepository

`PagingAndSortingRepository` permet d'effectuer des opérations de pagination et de tri.

> C'est une interface qui étend `JpaRepository`

```
public interface PersonRepository extends PagingAndSortingRepository<Person, Long>
{
}
```

PagingAndSortingRepository

ici le repository contient les méthodes du CRUD plus les méthodes suivantes :

- `findAll(Pageable pageable)` : pour récupérer toutes les entités avec pagination (`GET`)
- `findAll(Sort sort)` : pour récupérer toutes les entités avec tri (`GET`)

Exemple avec une entité `User`

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
    private String password;
    // Getters and setters
}
```

Exemple du repository

```
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Injection d'un repository

Dans une autre classe, comme un service, il est possible d'injecter un repository en utilisant l'injection de dépendances.

Spring va automatiquement générer une instance du repository.

```
@Service
public class UserService {
    private UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // ...
}
```

Les méthodes du repository

Spring génère automatiquement des méthodes pour effectuer des opérations sur la base de données.

- `findAll()` : pour récupérer tous les enregistrements
- `findById()` : pour récupérer un enregistrement par son id
- `save()` : pour enregistrer un enregistrement

- `deleteById()` : pour supprimer un enregistrement par son id
- `delete()` : pour supprimer un enregistrement

> Chaque interface de `Spring Data` contient ces méthodes.

Ajouter ses propres méthodes

Il est possible d'ajouter ses propres méthodes dans un repository en respectant la convention de nommage.

> `<verbe><*champ>by<champ>(<type> <champ>)`

Exemple

```
findByEmail(String email);
findByEmailAndName(String email, String name);
findByEmailOrName(String email, String name);
findByEmailOrderByDate(String email);
findByEmailOrderByDateDesc(String email);
findByEmailNot(String email);
findByEmailIn(List<String> emails);
findByEmailNotIn(List<String> emails);
findByEmailLike(String email);
```

Les verbes:

- `findAll` : pour récupérer tous les enregistrements
- `findOne` : pour récupérer un enregistrement
- `find` : pour récupérer un ou plusieurs enregistrements
- `count` : pour compter le nombre d'enregistrements
- `exists` : pour vérifier l'existence d'un enregistrement
- `delete` : pour supprimer un ou plusieurs enregistrements

---

filtres

Pour choisir comment filtrer les résultats, il est possible d'utiliser le champs au format `camelCase`.

```
findByEmail(String email);
```

Assemblage des filtres

Vous pouvez combiner plusieurs filtres en utilisant les mots clés `And`, `Or` et `Not`.

```
findByEmailAndName(String email, String name);
findByEmailOrName(String email, String name);
```

Tri

Pour trier les résultats, il est possible d'utiliser le champs au format `camelCase`.

```
findByEmailOrderByDate(String email);
findByEmailOrderByDateDesc(String email);
```

Négation

Pour négation, il est possible d'utiliser le mot clé `Not`.

```
findByEmailNot(String email);
```

In

Pour filtrer les résultats par rapport à une liste, il est possible d'utiliser le mot clé `In`.

```
findByEmailIn(List<String> emails);
findByEmailNotIn(List<String> emails);
```

Like

Pour filtrer les résultats par rapport à une expression régulière, il est possible d'utiliser le mot clé `Like`.

```
findByEmailLike(String email);
```

Exemple

```
public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmail(String email);
    List<User> findByEmailAndName(String email, String name);
    List<User> findByEmailOrName(String email, String name);
    List<User> findByEmailOrderByDate(String email);
```

```
    List<User> findByEmailOrderByDateDesc(String email);
    List<User> findByEmailNot(String email);
    List<User> findByEmailIn(List<String> emails);
    List<User> findByEmailNotIn(List<String> emails);
    List<User> findByEmailLike(String email);
}
```

Les requêtes personnalisées

Il est possible d'écrire des requêtes personnalisées en utilisant l'annotation `@Query`.

> Attention: Ce n'est pas du SQL, mais du `JPQL`.

```
@Query("SELECT u FROM User u WHERE u.email = ?1")
User findByEmail(String email);
```

Les requêtes personnalisées

Pour utiliser du SQL, il est possible d'utiliser l'attribut `native` de l'annotation `@Query`.

```
@Query(native = "SELECT u FROM User u WHERE u.email = ?1")
User findByEmail(String email);
```

Les requêtes personnalisées

Il est possible de spécifier les paramètres de la requête en utilisant l'annotation `@Param`.

> Cela permet de spécifier le nom du paramètre dans la requête.

```
@Query("SELECT u FROM User u WHERE u.email = :email AND u.name = :name")
List<User> findByEmailAndName(@Param("email") String email, @Param("name") String
name);
```

Exercice 4

Les transactions avec Spring Data JPA

Les transactions

Les transactions sont des opérations essentielles dans de nombreuses applications. Elles permettent de garantir l'intégrité des données.

Spring data JPA permet de gérer les transactions de manière automatique.

Rappel sur la notion de transaction

Une transaction est un ensemble d'opérations qui doivent être effectuées de manière atomique.

Si l'une des opérations échoue, toutes les opérations doivent être annulées.

Elles permettent aussi de garantir la cohérence des données en bloquant l'accès aux données pendant l'exécution de la transaction.

ACID

Les transactions doivent respecter les propriétés ACID.

- `Atomicité` : l'ensemble des opérations doit être effectué ou annulé.
- `Consistance` : les données doivent être cohérentes.
- `Isolation` : les opérations doivent être isolées les unes des autres.
- `Durabilité` : les opérations doivent être persistantes.

Les transactions

Pour gérer les transactions, il est possible d'utiliser l'annotation `@Transactional`.

```
@Transactional
public void save(User user) {
    userRepository.save(user);
}
```

> Cette annotation peut être utilisée sur les méthodes des services.

@Transactional

Quand l'annotation `@Transactional` est utilisée sur une méthode, toutes les opérations effectuées dans cette méthode sont considérées comme une transaction.

Si une exception est levée, toutes les opérations sont annulées.

Exemple

Ici si une exception est levée dans la méthode updateStock, la transaction est annulée et la commande n'est pas créée.

```
@Service
public class OrderService {
    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private StockService stockService;

    @Transactional(rollbackFor = StockException.class)
    public void createOrder(long productId, int quantity) throws StockException {
        Order order = new Order();
        //remplir les champs de order
        orderRepository.save(order);

        stockService.updateStock(productId, -quantity);
    }
}
```

Placement de l'annotation

L'annotation `@Transactional` peut être placée sur plusieurs éléments:

- `Méthode` : pour définir une transaction sur une méthode
- `Classe` : pour définir une transaction sur toutes les méthodes de la classe

---

Placement de l'annotation sur la Classe

```
@Service
@Transactional
public class OrderService {

    private OrderRepository orderRepository;
    private StockService stockService;

    public void createOrder(long productId, int quantity) throws StockException {
        Order order = new Order();
```

```
        orderRepository.save(order);
        stockService.updateStock(productId, -quantity);
    }
}
```

Placement de l'annotation sur la Classe

- Si l'on fait le choix de place l'annotation directement sur la classe, toutes les méthodes de la classe seront transactionnelles et avec les mêmes paramètres.

Placement de l'annotation sur les Méthodes

```
@Service
public class OrderService {

    private OrderRepository orderRepository;
    private StockService stockService;

    @Transactional
    public void createOrder(long productId, int quantity) throws StockException {
        Order order = new Order();
        orderRepository.save(order);
        stockService.updateStock(productId, -quantity);
    }
}
```

Placement de l'annotation sur les Méthodes

- Si l'on fait le choix de placer l'annotation sur les méthodes, il est possible de spécifier les paramètres de la transaction pour chaque méthode.
- Cela permet de personnaliser les transactions pour chaque cas d'utilisation.

Les paramètres de @Transactional

Il est possible de spécifier les paramètres de l'annotation `@Transactional`:

- `propagation` : pour spécifier le type de propagation de la transaction
- `isolation` : pour spécifier le niveau d'isolation de la transaction
- `timeout` : pour spécifier le temps d'attente avant d'annuler la transaction
- `readOnly` : pour spécifier si la transaction est en lecture seule
- `rollbackFor` : pour spécifier les exceptions qui doivent annuler la transaction

---

Rollback

Par défaut, toutes les exceptions annulent la transaction.

Vous pouvez spécifier les exceptions qui doivent annuler la transaction en utilisant l'attribut `rollbackFor`.

```
@Transactional(rollbackFor = StockException.class)
public void createOrder(long productId, int quantity) throws StockException {
    Order order = new Order();
    //remplir les champs de order
    orderRepository.save(order);

    stockService.updateStock(productId, -quantity);
}
```

Read Only

Il est possible de spécifier que la transaction est en lecture seule en utilisant l'attribut `readOnly`.

```
@Transactional(readOnly = true)
public void createOrder(long productId, int quantity) {
    Order order = new Order();
    //remplir les champs de order
    orderRepository.save(order);
}
```

> Vous ne pouvez pas modifier les données dans une transaction en lecture seule.

Propagation

Il est possible de spécifier le type de propagation de la transaction en utilisant l'attribut `propagation`.

La propagation de la transaction permet de spécifier comment les transactions sont gérées quand une méthode est appelée depuis une autre méthode.

Propagation: Exemple

Ici la méthode `parent` appel la méthode `enfant`.

```
@Transactional
public void parent(params) {
    // du code
    this.enfant(params);
}

@Transactional
public void enfant(params) {
```

```
    // du code
}
```

Propagation: Les types

Il existe différents types de propagation de transaction:

- `REQUIRED`
- `REQUIRES_NEW`
- `SUPPORTS`
- `MANDATORY`
- `NESSTED`
- `NOT_SUPPORTED`
- `NEVER`

Propagation: REQUIRED

Si la méthode appelante est déjà dans une transaction, la méthode appelée utilise la même transaction. Sinon, une nouvelle transaction est démarrée pour la méthode appelée.

> Le type `REQUIRED` est le type par défaut.

Propagation: REQUIRES_NEW

Une nouvelle transaction est toujours démarrée pour la méthode appelée, indépendamment de l'état de la transaction de la méthode appelante.

La transaction de la méthode appelante est mise en attente jusqu'à ce que la transaction de la méthode appelée soit terminée.

Propagation: SUPPORTS

Si la méthode appelante est déjà dans une transaction, la méthode appelée utilise la même transaction. Sinon, la méthode appelée est exécutée sans transaction.

Propagation: MANDATORY

La méthode appelée doit toujours être exécutée dans une transaction. Si la méthode appelante n'est pas dans une transaction, une exception est levée.

Propagation: NESTED

Si la méthode appelante est déjà dans une transaction, une transaction imbriquée est démarrée pour la méthode appelée. Sinon, une nouvelle transaction est démarrée pour la méthode appelée.

Les transactions imbriquées permettent de gérer les erreurs de manière plus fine.

Propagation: NOT_SUPPORTED

La méthode appelée est toujours exécutée sans transaction, indépendamment de l'état de la transaction de la méthode appelante.

La transaction de la méthode appelante est mise en attente jusqu'à ce que la méthode appelée soit terminée.

Propagation: NEVER

La méthode appelée ne doit jamais être exécutée dans une transaction. Si la méthode appelante est dans une transaction, une exception est levée.

Isolation

Il est possible de spécifier le niveau d'isolation de la transaction en utilisant l'attribut `isolation`.

L'isolation de la transaction permet de spécifier comment les transactions sont gérées quand plusieurs transactions entre en concurrence sur les mêmes données.

Pourquoi l'isolation est importante

Imaginons que vous avez une base de données avec des comptes bancaires.

Quand vous faites un virement bancaire, vous débitez votre compte et créditez le compte du destinataire.

Si vous lisiez le solde de votre compte avant que la transaction échoue, vous verriez que le virement a été effectué.

Isolation: Exemple

```
Le compte A a 100€
Le compte B a 10€

Debut de la transaction pour un virement de 20€ de A vers B
    Le compte A gagne virtuellement 20€ et passe à 120€
    Le compte B perd virtuellement 20 et passe à -10€
    Erreur dans la transaction: B est en dessous de 0€
    La transaction est annulée
Fin de la transaction
```

> Que ce passe t'il si une autre personne lit le solde du compte A pendant la transaction?

Isolation

Il faut donc que la transaction soit isolée des autres transactions.

Ainsi si une autre transaction lit le solde du compte A pendant la transaction, elle verra toujours 100€.

Pour ce faire elle va bloquer le compte A pour les autres transactions.

Isolation: Lock

Voici le même exemple avec un lock sur le compte A.

```
Le compte A a 100€
Le compte B a 100€

Debut de la transaction pour un virement de 20€ de B vers A
    Le compte A est bloqué pour les autres transactions
    Le compte A gagne virtuellement 20€ et passe à 120€
    Le compte B est bloqué pour les autres transactions
    Le compte B perd virtuellement 20 et passe à 80€
    Le compte A et B sont débloqués pour les autres transactions
Fin de la transaction
```

Deadlock

Mais que ce passe t'il si une autre transaction veut faire un virement de 20€ de B vers A?

```
La transaction 1 Block le compte A
La transaction 2 Block le compte B
La transaction 1 Attend le compte B
La transaction 2 Attend le compte A
```

Les types d'isolation

Il existe différents niveaux d'isolation:

- `READ_UNCOMMITTED`
- `READ_COMMITTED`
- `REPEATABLE_READ`
- `SERIALIZABLE`

Isolation: READ_UNCOMMITTED

Le niveau `READ_UNCOMMITTED` permet de lire les données non committées.

Même si une transaction n'est pas validée, les autres transactions peuvent lire les données.

Cela peut causer des problèmes de données inconsistantes.

Mais cela permet d'optimiser les performances.

Isolation: READ_COMMITTED

Le niveau `READ_COMMITTED` permet de lire les données committées.

Les autres transactions seront bloquées tant que la transaction n'est pas validée.

Cela permet d'éviter les problèmes de données inconsistantes.

Mais cela rallonge les temps de réponse.

Isolation: REPEATABLE_READ

Le niveau `REPEATABLE_READ` permet de lire les données committées et de bloquer les autres transactions.

Cela permet d'éviter les problèmes de données inconsistantes.

Mais cela rallonge les temps de réponse encore plus que `READ_COMMITTED`.

Isolation: SERIALIZABLE

Le niveau `SERIALIZABLE` permet de lire les données committées et de bloquer les autres transactions.

Chaque transaction est traitée séquentiellement ce qui offre la meilleure isolation.

Mais cela rallonge les temps de réponse encore plus que `REPEATABLE_READ`.

Comment choisir le niveau d'isolation

Il faut choisir le niveau d'isolation en fonction des besoins de l'application.

Isolation Level	Dirty Read/Write	Non-Repeatable Reads	Phantom Reads
Read un-committed Isolation level	EXISTS	EXISTS	EXISTS
Read Committed isolation level	SOLVED	EXISTS	EXISTS
Repeatable Reads Isolation level	SOLVED	SOLVED	EXISTS
Serializable Isolation level	SOLVED	SOLVED	SOLVED

Dirty Reads

Les problèmes de `Dirty Reads` surviennent quand une transaction lit une ligne modifiée par une autre transaction mais non encore validée.

On ce retrouve alors avec des données incorrectes car la transaction qui modifie les valeurs ne sera peut être pas commit.

Non-Repeatable Reads

Les problèmes de `Non-Repeatable Reads` surviennent quand une transaction démarre et lit une ligne puis qu'une autre transaction modifie cette ligne avant que la première transaction ne soit terminée.

On ce retrouve alors avec des données différentes lors de la deuxième lecture.

Phantom Reads

Les problèmes de `Phantom Reads` surviennent quand une transaction démarre et lit un ensemble de lignes puis qu'une autre transaction ajoute ou supprime une ligne de cet ensemble avant que la première transaction ne soit terminée.

On ce retrouve alors avec des données différentes lors de la deuxième lecture.

Le seul moyen de résoudre ce problème est de bloquer l'ensemble des lignes le temps de la transaction.

Timeout

Il est possible de spécifier le timeout de la transaction en utilisant l'attribut `timeout`.

Le `timeout` permet d'annuler la transaction si elle prend trop de temps.

> Pratique pour éviter les deadlocks.

Timeout

```
@Transactional(timeout = 10)
public void createOrder(long productId, int quantity) {
    Order order = new Order();
    //remplir les champs de order
    orderRepository.save(order);
}
```