**SpringDataJpa**

1. **Introduction**
2. **Ce qu’on va faire pour ce cours.**

Pour ce cour on va créer ces diagrammes on utilisant springData.

Une image contenant texte, diagramme, ligne, Parallèle

Description générée automatiquement

Une image contenant texte, diagramme, ligne, Tracé

Description générée automatiquement

Une image contenant texte, capture d’écran, diagramme, nombre

Description générée automatiquement

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

1. **C’est quoi Spring Data**

Spring Data JPA est un projet de la famille Spring Framework qui simplifie le développement des applications Java/JEE en intégrant la couche d'accès aux données avec la gestion des entités JPA (Java Persistence API).

JPA est une spécification Java qui définit une API pour la gestion de la persistance des objets dans les bases de données relationnelles. Spring Data JPA simplifie l'utilisation de JPA en fournissant des fonctionnalités supplémentaires et en réduisant la quantité de code nécessaire pour interagir avec la base de données.

Les principales fonctionnalités de Spring Data JPA comprennent :

1. **Génération automatique de requêtes**: Spring Data JPA permet de générer automatiquement des requêtes SQL à partir de noms de méthodes dans les interfaces de répository. Par exemple, en définissant une méthode **findByLastName(String lastName)** dans une interface de répository, Spring Data JPA générera automatiquement une requête pour rechercher les entités par nom de famille.
2. **Pagination**: Il facilite la pagination des résultats de requête en fournissant des méthodes simples pour spécifier le numéro de page et la taille de la page.
3. **Tri des résultats**: Il permet de trier les résultats de requête en spécifiant les champs sur lesquels trier et l'ordre de tri.
4. **Spécifications dynamiques**: Spring Data JPA offre la possibilité de créer des requêtes de manière dynamique en fonction de certains critères.
5. **Auditing**: Il prend en charge la gestion automatique des champs d'audit tels que la date de création et la date de mise à jour.
6. **Requêtes nommées**: Il permet de définir des requêtes nommées dans les annotations **@NamedQuery** ou dans des fichiers XML pour une gestion plus facile des requêtes complexes.

Dans l'ensemble, Spring Data JPA simplifie le développement des applications basées sur JPA en fournissant des fonctionnalités prêtes à l'emploi et en réduisant la quantité de code boilerplate nécessaire.

1. **Getting started with spring data jpa**

**Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement**

package com.springdata.springdatajpa;  
  
import jakarta.persistence.\*;  
  
@Entity(name= "Student")

@Table(  
 name="student",  
 uniqueConstraints = {  
 @UniqueConstraint(name = "student\_email\_unique",columnNames = "email")  
 }  
)

public class Student {  
  
 @Id  
 @SequenceGenerator(  
 name = "student\_sequence",  
 sequenceName = "student\_sequence",  
 allocationSize = 1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "student\_sequence"  
 )  
 @Column(  
 name="id",  
 updatable = false,  
 nullable = false  
 )  
 private Long id;  
  
 @Column(  
 name="first\_name",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String firstName;  
  
 @Column(  
 name="last\_name",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String lastName;  
  
 @Column(  
 name="email",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String email;  
  
 @Column(  
 name = "age",  
 nullable = false  
 )  
 private Integer age;  
  
 public Student(Long id, String firstName, String lastName, String email, Integer age) {  
 this.id = id;  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.email = email;  
 this.age = age;  
 }  
  
 public Student() {  
  
 }  
  
 public Long getId() {  
 return id;  
 }  
  
 public void setId(Long id) {  
 this.id = id;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "id=" + id +  
 ", firstName='" + firstName + '\'' +  
 ", lastName='" + lastName + '\'' +  
 ", email='" + email + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

les annotations utilisées dans la classe Student :

1. **@Entity**: Cette annotation indique à JPA que la classe Student est une entité, c'est-à-dire qu'elle est mappée à une table dans la base de données.
2. **@Id**: Cette annotation marque la propriété 'id' comme étant la clé primaire de l'entité.
3. **@SequenceGenerator**: Cette annotation est utilisée pour spécifier le générateur de séquence qui sera utilisé pour générer les valeurs des clés primaires.
4. **@GeneratedValue**: Cette annotation spécifie comment la valeur de la clé primaire sera générée. Dans ce cas, la stratégie utilisée est GenerationType.SEQUENCE, ce qui signifie que la valeur sera générée en utilisant la séquence définie par @SequenceGenerator.
5. **@Column**: Cette annotation est utilisée pour mapper une propriété de l'entité à une colonne de la table de base de données. Vous pouvez spécifier des attributs tels que le nom de la colonne, s'il est unique, s'il peut être mis à jour ou s'il peut être nul, etc.

**@Table** est utilisée pour spécifier le nom de la table dans la base de données pour l'entité associée, ainsi que d'autres propriétés de la table. Voici une explication de chaque attribut utilisé dans cet exemple :

1. **name="student"**: Cet attribut spécifie le nom de la table dans la base de données. Dans cet exemple, le nom de la table est "student".
2. **uniqueConstraints**: Cet attribut est utilisé pour définir des contraintes d'unicité sur les colonnes de la table. Dans cet exemple, il y a une contrainte d'unicité sur la colonne "email".
3. **@UniqueConstraint**: Cette annotation est utilisée pour définir une contrainte d'unicité sur une ou plusieurs colonnes de la table. Elle prend plusieurs attributs, dont voici ceux utilisés dans cet exemple :
   * **name**: Cet attribut spécifie le nom de la contrainte d'unicité. Dans cet exemple, le nom de la contrainte est "student\_email\_unique".
   * **columnNames**: Cet attribut spécifie le nom de la ou des colonnes sur lesquelles la contrainte d'unicité est appliquée. Dans cet exemple, la contrainte d'unicité est appliquée à la colonne "email".

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

Ce code représente la configuration de l'application Spring Boot avec Spring Data JPA pour se connecter à une base de données PostgreSQL. Voici ce que chaque ligne signifie :

1. **spring.application.name=SpringDataJpa**: Cela définit le nom de l'application Spring Boot. Dans ce cas, le nom est "SpringDataJpa".
2. **spring.datasource.url=jdbc:postgresql://localhost:5432/postgres**: Cela spécifie l'URL JDBC de la base de données PostgreSQL à laquelle l'application se connectera. Dans cet exemple, l'application se connectera à une base de données appelée "postgres" sur le localhost (la même machine) sur le port 5432.
3. **spring.datasource.username=postgres**: Cela définit le nom d'utilisateur utilisé pour se connecter à la base de données PostgreSQL. Dans ce cas, le nom d'utilisateur est "postgres".
4. **spring.datasource.password=houssam**: Cela définit le mot de passe utilisé pour se connecter à la base de données PostgreSQL. Dans cet exemple, le mot de passe est "houssam".
5. **spring.jpa.hibernate.ddl-auto=update**: Cela spécifie le comportement de Hibernate pour la génération du schéma de base de données. "update" signifie que Hibernate mettra à jour le schéma de la base de données au démarrage de l'application pour correspondre aux entités JPA.
6. **spring.jpa.show-sql=true**: Cela active l'affichage des requêtes SQL générées par Hibernate dans les logs. Cela peut être utile pour le débogage et la compréhension de ce qui se passe dans la base de données.
7. **spring.jpa.properties.hibernate.format\_sql=true**: Cela indique à Hibernate de formater les requêtes SQL générées pour les rendre plus lisibles dans les logs. Cela n'affecte pas l'exécution des requêtes, mais facilite leur compréhension lors de la lecture des logs.
8. **Repositories**

**Une image contenant texte, capture d’écran, diagramme, affichage

Description générée automatiquement**

Les "repositories" (ou "référentiels" en français) sont une partie importante de l'architecture des applications Spring utilisant Spring Data. Ils sont responsables de la gestion de la persistance des données, c'est-à-dire de l'interaction avec la base de données.

En utilisant Spring Data, vous pouvez définir des interfaces de référentiels qui étendent des interfaces spécifiques fournies par Spring Data, telles que JpaRepository, MongoRepository, CrudRepository, etc. Ces interfaces de référentiels fournissent des méthodes prédéfinies pour effectuer des opérations courantes sur les données telles que l'insertion, la mise à jour, la suppression et la recherche.

L'interface JpaRepository en particulier est utilisée pour la persistance des données avec JPA (Java Persistence API). Elle fournit des méthodes pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur les entités JPA.

@Bean  
CommandLineRunner commandLineRunner(StudentRepository studentRepository){  
 return args -> {  
 Student houssam = new Student("houssam","baaloul","houssam.baaloul78@gmail.com",24);  
 Student abir = new Student("abir","baaloul","abir.baaloul5@gmail.com",23);  
  
 System.*out*.println("Adding houssam and abir");  
 studentRepository.saveAll(List.*of*(houssam, abir));  
  
 System.*out*.println("Number of students: ");  
 System.*out*.println(studentRepository.count());  
  
 studentRepository  
 .findById(2L)  
 .ifPresentOrElse(  
 System.*out*::println,  
 ()->System.*out*.println("Student with id 2 notFound")  
 );  
  
 studentRepository  
 .findById(3L)  
 .ifPresentOrElse(  
 System.*out*::println,  
 ()->System.*out*.println("Student with id 3 notFound")  
 );  
  
 System.*out*.println("Select all students");  
 List<Student> students = studentRepository.findAll();  
 students.forEach(System.*out*::println);  
  
 System.*out*.println("Delete houssam");  
 studentRepository.deleteById(1L);  
  
 System.*out*.println("Number of students: ");  
 System.*out*.println(studentRepository.count());  
  
 };  
}

Voici les méthodes JpaRepository correspondantes à chaque ligne de votre code :

**saveAll()** : Cette méthode enregistre tous les éléments spécifiés dans la liste dans la base de données.

**count()** : Cette méthode renvoie le nombre total d'entités dans la table associée.

**findById()** : Cette méthode recherche une entité par son identifiant. Elle renvoie un Optional qui peut être présent ou non, selon que l'entité avec cet identifiant existe ou non.

**findAll()** : Cette méthode renvoie toutes les entités de la table associée.

**deleteById()** : Cette méthode supprime une entité de la table associée en fonction de son identifiant.

**count()** : Comme déjà utilisé, cette méthode renvoie le nombre total d'entités dans la table associée, après la suppression.

Ces méthodes sont fournies par l'interface JpaRepository et sont utilisées pour interagir avec la base de données de manière pratique et efficace.

1. **Querying data**

L'interrogation des données à l'aide de Spring Data JPA consiste à récupérer des données à partir d'une base de données relationnelle en utilisant des méthodes fournies par Spring Data JPA et des requêtes dérivées.

Spring Data JPA simplifie grandement le processus d'interrogation des données en permettant aux développeurs de définir des méthodes de requête dans les interfaces de référentiels. Ces méthodes peuvent être dérivées du nom de la méthode, basées sur des conventions de nommage, ou elles peuvent utiliser des annotations pour écrire des requêtes JPQL (Java Persistence Query Language) personnalisées.

Voici quelques méthodes couramment utilisées pour interroger des données à l'aide de Spring Data JPA :

1. **findBy{PropertyName}**: Cette méthode permet de rechercher des enregistrements en fonction de la valeur d'une propriété spécifique. Par exemple, findByFirstName(String firstName) recherchera tous les enregistrements ayant un prénom correspondant à la valeur spécifiée.
2. **findBy{PropertyName}And{OtherPropertyName}**: Cette méthode permet de rechercher des enregistrements en fonction de la valeur de deux propriétés spécifiques. Par exemple, findByFirstNameAndLastName(String firstName, String lastName) recherchera tous les enregistrements ayant un prénom et un nom correspondant aux valeurs spécifiées.
3. **findBy{PropertyName}OrderBy{OtherPropertyName}Desc()**: Cette méthode permet de rechercher des enregistrements en fonction de la valeur d'une propriété spécifique, triés par ordre décroissant d'une autre propriété. Par exemple, findByAgeOrderByLastNameDesc(Integer age) recherchera tous les enregistrements ayant un âge correspondant, triés par nom de famille dans l'ordre décroissant.
4. **@Query**: Cette annotation permet de définir des requêtes JPQL personnalisées directement dans les interfaces de référentiels. Elle peut être utilisée pour écrire des requêtes plus complexes qui ne peuvent pas être exprimées à l'aide des méthodes dérivées.

En résumé, interroger des données avec Spring Data JPA implique l'utilisation de méthodes de requête dérivées ou de requêtes JPQL personnalisées pour récupérer des données à partir d'une base de données relationnelle de manière simple et efficace.

**In this link you will find how to use jpql**

**https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html**

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

**Une image contenant texte, capture d’écran, document, Police

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

**Voici un Example :**

package com.springdata.springdatajpa;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Query;  
import org.springframework.data.repository.query.Param;  
  
import java.util.List;  
import java.util.Optional;  
  
public interface StudentRepository extends JpaRepository<Student,Long> {  
 Optional<Student> findStudentByEmail(String email);  
  
 List<Student> findStudentByFirstNameEqualsAndAgeIsGreaterThanEqual(String firstName,Integer age);  
  
 @Query("SELECT s FROM Student s WHERE s.lastName=?1")  
 List<Student> chercherStudenatApartieDeSonNom(String lastName);  
  
 @Query(value = "select \* from student where first\_name=?1",nativeQuery = true)  
 List<Student> chercherStudenatApartieDeSonPrenomNativeQuery(String firstName);  
  
 @Query(value = "select \* from student where first\_name=:firstName",nativeQuery = true)  
 List<Student> chercherStudenatApartieDeSonPrenomNativeQueryUsingNamedParam(@Param("firstName") String firstName);  
  
}

Voici une explication de chaque annotation utilisée dans cette interface **StudentRepository** :

1. **@JpaRepository**:
   * Cette annotation est fournie par Spring Data et est utilisée pour marquer une interface comme un référentiel (repository) qui gère les opérations CRUD pour une entité spécifique.
   * Dans ce cas, l'interface **StudentRepository** gère les opérations CRUD pour l'entité **Student**.
   * **JpaRepository** étend l'interface **CrudRepository** et fournit des méthodes pour effectuer des opérations de base sur les entités, telles que **save()**, **findById()**, **findAll()**, etc.
2. **@Query**:
   * Cette annotation est utilisée pour définir des requêtes personnalisées pour récupérer des données de la base de données.
   * Elle peut être utilisée pour écrire des requêtes JPQL (Java Persistence Query Language) ou des requêtes SQL natives.
   * Elle peut être utilisée avec ou sans la propriété **nativeQuery** pour spécifier si la requête est JPQL ou SQL native.
   * Dans les exemples ci-dessus, **@Query** est utilisée pour définir des requêtes JPQL et SQL natives pour rechercher des étudiants en fonction de certains critères.
3. **@Param**:
   * Cette annotation est utilisée pour mapper les paramètres de la méthode à des paramètres de requête dans une requête JPQL ou SQL native.
   * Elle est utilisée avec **@Query** lorsque vous utilisez des paramètres nommés dans vos requêtes.
   * Dans l'exemple **chercherStudenatApartieDeSonPrenomNativeQueryUsingNamedParam**, **@Param("firstName")** mappe le paramètre **firstName** de la méthode au paramètre nommé **:firstName** dans la requête.

En résumé, ces annotations permettent de définir des méthodes de requête personnalisées pour interagir avec la base de données à l'aide de Spring Data JPA. Elles offrent une flexibilité et un contrôle accrus sur les requêtes de données tout en maintenant la facilité d'utilisation et la sécurité fournies par Spring Data.

**3.**

L'annotation **@Modifying** est utilisée en conjonction avec la notation **@Query** dans Spring Data JPA pour indiquer que la requête spécifiée modifie les données dans la base de données.

Voici quelques points clés à retenir sur l'annotation **@Modifying** :

1. **Objectif** : L'annotation **@Modifying** est utilisée pour signaler à Spring Data JPA que la requête annotée modifie les données dans la base de données. Cela permet à Spring Data JPA de gérer correctement la transaction pour cette requête.
2. **Utilisation** : Elle est utilisée avec l'annotation **@Query** qui définit la requête personnalisée. Lorsque vous utilisez **@Modifying**, assurez-vous que la requête effectue une opération de modification de données comme INSERT, UPDATE ou DELETE.
3. **Effets** : L'annotation **@Modifying** indique à Spring Data JPA que la requête modifie l'état de la base de données et qu'une transaction est nécessaire pour exécuter cette requête. Cela garantit que la modification des données est correctement gérée et que les transactions sont commitées ou rollbackées selon les besoins.
4. **Transactional** : L'annotation **@Modifying** est souvent utilisée en conjonction avec l'annotation **@Transactional** pour délimiter le début et la fin d'une transaction pour la requête modifiante. Cela garantit que la transaction est correctement gérée et que les modifications de données sont atomiques.

En résumé, l'annotation **@Modifying** est utilisée pour indiquer à Spring Data JPA qu'une requête personnalisée modifie les données dans la base de données, et qu'une transaction est nécessaire pour exécuter cette requête de manière sécurisée.

Voici un exemple d'utilisation de l'annotation **@Modifying** en conjonction avec **@Query** dans une interface de référentiel Spring Data JPA :

Supposons que nous voulons écrire une méthode de requête personnalisée pour mettre à jour l'âge d'un étudiant en fonction de son identifiant. Voici comment cela pourrait être fait :

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

Dans cet exemple :

* **@Modifying** indique à Spring Data JPA que la méthode modifie les données dans la base de données.
* **@Transactional** délimite le début et la fin de la transaction pour cette méthode. Cela garantit que la transaction est correctement gérée et que les modifications de données sont atomiques.
* **@Query** spécifie la requête JPQL personnalisée qui met à jour l'âge de l'étudiant en fonction de son identifiant.
* Les paramètres de méthode **studentId** et **newAge** sont mappés aux paramètres de la requête JPQL **:studentId** et **:newAge**.

Lorsque cette méthode est appelée, elle exécutera la requête personnalisée pour mettre à jour l'âge de l'étudiant dans la base de données. La transaction sera correctement gérée, assurant que les modifications de données sont commitées ou rollbackées selon les besoins.

1. **Sorting et pagination**

**Une image contenant texte, capture d’écran, ligne, Police

Description générée automatiquement**

des interfaces **JpaRepository**, **PagingAndSortingRepository** et **CrudRepository** dans Spring Data JPA :

1. **CrudRepository** :
   * CrudRepository est l'interface de base de Spring Data JPA qui fournit des méthodes CRUD (Create, Read, Update, Delete) de base pour interagir avec une source de données.
   * Elle contient des méthodes telles que **save()**, **findById()**, **findAll()**, **deleteById()**, etc., qui permettent de manipuler les entités dans la base de données.
2. **PagingAndSortingRepository** :
   * PagingAndSortingRepository étend CrudRepository et ajoute la fonctionnalité de pagination et de tri aux méthodes CRUD.
   * Il introduit des méthodes supplémentaires telles que **findAll(Pageable pageable)** pour récupérer les données paginées, et **findAll(Sort sort)** pour récupérer les données triées.
3. **JpaRepository** :
   * JpaRepository étend PagingAndSortingRepository et fournit des fonctionnalités supplémentaires spécifiques à JPA.
   * Il offre des méthodes spécifiques à JPA telles que **flush()** pour synchroniser les changements avec la base de données, **saveAndFlush()** pour sauvegarder et synchroniser, **deleteInBatch()** pour supprimer en lot, etc.
   * En plus des fonctionnalités de CrudRepository et PagingAndSortingRepository, JpaRepository est souvent utilisé dans les applications Spring Data JPA car il fournit des méthodes spécifiques à JPA qui peuvent simplifier le développement en évitant la nécessité de créer des requêtes personnalisées.

En résumé, ces interfaces dans Spring Data JPA fournissent une abstraction pour simplifier l'accès aux données et offrent différentes fonctionnalités selon les besoins de l'application, allant des opérations CRUD de base à la pagination, au tri et aux fonctionnalités spécifiques à JPA.

1. **Java faker**

Java Faker est une bibliothèque Java qui permet de générer des données factices (fake data) de manière aléatoire. Elle est souvent utilisée dans les tests unitaires, les tests d'intégration, ou même dans le développement de démonstrations.

Dans le contexte de Spring Boot, Java Faker peut être utilisée pour générer des données fictives à des fins de développement, de test ou de simulation. Par exemple, si vous avez besoin de peupler une base de données avec des données aléatoires pour tester le fonctionnement de votre application, Java Faker peut être très utile.

@Bean  
CommandLineRunner commandLineRunner(StudentRepository studentRepository){  
 return args -> {  
 Faker faker = new Faker();  
 for(int i=0; i<=100; i++){  
 String firstName = faker.name().firstName();  
 String lastName = faker.name().lastName();  
 String email= String.*format*("%s.%s@alHoussam.edu",firstName,lastName);  
 Student student = new Student(  
 firstName,  
 lastName,  
 email,  
 faker.number().numberBetween(17,55));  
 studentRepository.save(student);  
 }  
  
 };  
  
 }

**Une image contenant texte, capture d’écran, logiciel, nombre

Description générée automatiquement**

1. **Sorting**

Dans Spring Boot avec Spring Data JPA, le tri des résultats de requête est une fonctionnalité essentielle pour récupérer les données dans un ordre spécifique. Voici comment trier les résultats dans une application Spring Boot :

@Bean  
CommandLineRunner commandLineRunner(StudentRepository studentRepository){  
 return args -> {  
  
 generateRandomStudents(studentRepository);  
  
 Sort sort= Sort.*by*("firstName").ascending().and(Sort.*by*("age").descending());  
  
 studentRepository.findAll(sort)  
 .forEach(student -> System.*out*.println(student.getFirstName()));  
 };

1. **Pagination**

La pagination dans une application Spring Boot est une technique utilisée pour diviser les résultats d'une requête en pages afin de les afficher progressivement à l'utilisateur. Cela est particulièrement utile lorsque la requête renvoie un grand nombre de résultats, car afficher tous les résultats en une seule fois peut surcharger l'interface utilisateur et ralentir les performances.

Voici comment implémenter la pagination dans une application Spring Boot :

1. **Utilisation de Spring Data JPA** : Si vous utilisez Spring Data JPA pour interagir avec votre base de données, la pagination est assez simple à mettre en œuvre. Spring Data JPA fournit une interface **PagingAndSortingRepository** qui prend en charge la pagination automatique des résultats de requête.
2. **Utilisation de méthodes de pagination** : Dans votre couche de service ou de contrôleur, vous pouvez utiliser des méthodes de pagination fournies par Spring Data JPA. Par exemple, **findAll(Pageable pageable)** renvoie une page de résultats en fonction des paramètres de pagination spécifiés.
3. **Configurer Pageable** : L'objet **Pageable** est utilisé pour spécifier les paramètres de pagination tels que le numéro de page, la taille de la page et les options de tri. Vous pouvez créer un objet **PageRequest** en spécifiant le numéro de page, la taille de la page et les options de tri, puis le passer à la méthode de pagination.
4. **Affichage des résultats paginés** : Dans votre couche de vue (généralement le frontend), vous devez afficher les résultats de chaque page et fournir des liens de navigation pour permettre à l'utilisateur de passer d'une page à l'autre.

Dans votre contrôleur, vous pouvez appeler cette méthode et renvoyer la page de résultats à la vue pour affichage paginé. Ensuite, dans votre interface utilisateur, vous pouvez utiliser des boutons de pagination ou d'autres mécanismes de navigation pour permettre à l'utilisateur de parcourir les différentes pages de résultats.

voici quelques autres méthodes couramment utilisées de l'objet **Page** en français :

1. **getContent()** :
   * Cette méthode renvoie la liste des éléments de la page courante.
2. **getNumber()** :
   * Cette méthode renvoie le numéro de la page actuelle (commençant à 0).
3. **getNumberOfElements()** :
   * Cette méthode renvoie le nombre d'éléments dans la page actuelle.
4. **getSize()** :
   * Cette méthode renvoie la taille de la page.
5. **getTotalElements()** :
   * Cette méthode renvoie le nombre total d'éléments dans tous les résultats.
6. **getTotalPages()** :
   * Cette méthode renvoie le nombre total de pages.
7. **hasContent()** :
   * Cette méthode indique si la page actuelle contient des éléments.
8. **hasNext()** :
   * Cette méthode indique s'il y a une page suivante après la page actuelle.
9. **hasPrevious()** :
   * Cette méthode indique s'il y a une page précédente avant la page actuelle.
10. **isFirst()** :
    * Cette méthode indique si la page actuelle est la première page.
11. **isLast()** :
    * Cette méthode indique si la page actuelle est la dernière page.
12. **nextPageable()** :
    * Cette méthode renvoie un objet **Pageable** pour accéder à la page suivante.
13. **previousPageable()** :
    * Cette méthode renvoie un objet **Pageable** pour accéder à la page précédente.
14. **iterator()** :
    * Cette méthode renvoie un itérateur pour parcourir les éléments de la page courante.
15. **stream()** :
    * Cette méthode renvoie un flux de tous les éléments de la page courante.

Ces méthodes permettent d'interagir avec les résultats paginés et d'obtenir des informations sur la pagination en cours, ce qui est utile pour créer une interface utilisateur intuitive et fonctionnelle.

Voici un example :

@Bean  
CommandLineRunner commandLineRunner(StudentRepository studentRepository){  
 return args -> {  
  
 generateRandomStudents(studentRepository);  
  
 PageRequest pageRequest = PageRequest.*of*(  
 0  
 ,5  
 ,Sort.*by*("firstName").ascending());  
  
 Page<Student> page= studentRepository.findAll(pageRequest);  
 System.*out*.println(page);  
  
  
 };  
}

1. **Relationships**

**Une image contenant texte, diagramme, capture d’écran, ligne

Description générée automatiquement**

Les relations entre les tables dans une base de données sont des liens établis entre les données stockées dans différentes tables. Voici quelques-unes des relations les plus couramment utilisées dans les bases de données relationnelles, ainsi que leurs différences :

1. **Relation Un-à-Un (One-to-One)** :
   * Dans une relation un-à-un, chaque enregistrement dans une table est lié à un seul enregistrement dans une autre table, et vice versa. Par exemple, une personne peut avoir un seul numéro de sécurité sociale, et chaque numéro de sécurité sociale est associé à une seule personne.
2. **Relation Un-à-Plusieurs (One-to-Many)** :
   * Dans une relation un-à-plusieurs, chaque enregistrement dans une table est lié à plusieurs enregistrements dans une autre table, mais chaque enregistrement dans cette autre table est lié à un seul enregistrement dans la première table. Par exemple, un client peut passer plusieurs commandes, mais chaque commande est associée à un seul client.
3. **Relation Plusieurs-à-Un (Many-to-One)** :
   * C'est essentiellement l'inverse de la relation un-à-plusieurs. Dans une relation plusieurs-à-un, plusieurs enregistrements dans une table sont liés à un seul enregistrement dans une autre table. Par exemple, plusieurs commandes peuvent être passées par un seul client.
4. **Relation Plusieurs-à-Plusieurs (Many-to-Many)** :
   * Dans une relation plusieurs-à-plusieurs, plusieurs enregistrements dans une table peuvent être associés à plusieurs enregistrements dans une autre table. Cela nécessite une table de liaison (table de jointure) pour relier les enregistrements des deux tables. Par exemple, plusieurs étudiants peuvent être inscrits à plusieurs cours, et chaque inscription d'étudiant à un cours est une association distincte.

Les différences entre ces relations résident dans la façon dont les données sont liées entre les tables et comment elles sont interprétées. Par exemple, dans une relation un-à-un, chaque enregistrement est associé à un seul autre enregistrement, tandis que dans une relation un-à-plusieurs, un enregistrement peut être associé à plusieurs autres enregistrements, mais chaque enregistrement associé est lié à un seul enregistrement.

Ces relations sont fondamentales pour la conception de bases de données relationnelles, et le choix de la relation appropriée dépend souvent de la logique métier sous-jacente et des exigences fonctionnelles de l'application.

* 1. **One To One**

Dans une relation un-à-un, chaque enregistrement dans une table est lié à un seul enregistrement dans une autre table, et vice versa. Par exemple, une personne peut avoir un seul numéro de sécurité sociale, et chaque numéro de sécurité sociale est associé à une seule personne.

**Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement**

* + 1. **@oneToOne and @JoinColumn**

**JoinColumn** dans une relation **OneToOne** en JPA spécifie la colonne de la table propriétaire (côté propriétaire de la relation) qui est utilisée pour rejoindre la table associée (côté inverse de la relation). Cela établit un lien direct entre les deux entités, où la colonne spécifiée contient les clés étrangères nécessaires pour la relation. En résumé, **JoinColumn** définit la colonne dans la table de la première entité qui référence la clé primaire de la seconde entité.

1. **CascadeType && HibernateLifecycle**

**Une image contenant texte, diagramme, ligne, Tracé

Description générée automatiquement**

**CascadeType :**

1. **ALL** : Cette option propage toutes les opérations de persistance (PERSIST), de fusion (MERGE), de suppression (REMOVE), de rafraîchissement (REFRESH) et de détachement (DETACH) aux entités associées. Cela signifie que lorsque vous effectuez une opération sur l'entité parent, cette opération est propagée à ses entités enfants.
2. **PERSIST** : Cette option propage uniquement l'opération de persistance (INSERT) aux entités associées. Si vous persistez l'entité parent, les entités enfants seront également persistées.
3. **MERGE** : Cette option propage uniquement l'opération de fusion (UPDATE) aux entités associées. Lorsque vous fusionnez l'entité parent, les entités enfants seront également fusionnées.
4. **REMOVE** : Cette option propage l'opération de suppression aux entités associées. Lorsque vous supprimez l'entité parent, les entités enfants seront également supprimées.
5. **REFRESH** : Cette option propage l'opération de rafraîchissement aux entités associées. Lorsque vous rafraîchissez l'entité parent, les entités enfants seront également rafraîchies.
6. **DETACH** : Cette option propage l'opération de détachement aux entités associées. Lorsque vous détachez l'entité parent, les entités enfants seront également détachées.
7. **REPLICATE** : Cette option propage l'opération de réplication aux entités associées.
8. **LOCK** : Cette option propage l'opération de verrouillage aux entités associées.
9. **EVICT** : Cette option propage l'opération d'éviction aux entités associées.

**Cycle de vie Hibernate :**

1. **Transient (transitoire)** : L'entité est dans un état transitoire, elle vient d'être créée mais n'a pas encore été associée à une session Hibernate. Les changements apportés à l'entité ne sont pas suivis par Hibernate.
2. **Persistent (persistant)** : L'entité est associée à une session Hibernate et est suivie par cette session. Toute modification apportée à l'entité sera synchronisée avec la base de données lors de la prochaine validation de la transaction.
3. **Detached (détaché)** : L'entité était précédemment persistante, mais elle a été dissociée de la session Hibernate (généralement parce que la transaction a été fermée). Les modifications apportées à l'entité ne sont plus automatiquement synchronisées avec la base de données.
4. **Removed (supprimé)** : L'entité a été marquée comme supprimée et sera effectivement supprimée de la base de données lors de la prochaine validation de la transaction.
5. **FetchType**

Le FetchType (type de chargement) est une annotation utilisée en JPA (Java Persistence API) pour spécifier comment les données associées à une relation doivent être récupérées depuis la base de données. Il détermine si les données associées doivent être récupérées immédiatement avec l'entité parente ou si elles doivent être récupérées uniquement lorsque l'entité associée est explicitement accédée.

Il existe deux types de FetchType :

1. **FetchType.LAZY** : Avec cette option, les données associées ne sont récupérées de la base de données que lorsque vous accédez explicitement à la propriété correspondante. Par exemple, si vous avez une relation **@ManyToOne** ou **@OneToOne** avec FetchType.LAZY, les données de l'entité associée ne seront chargées qu'au moment où vous accédez à cette entité. Cela peut être plus efficace en termes de performances si les données associées ne sont pas toujours nécessaires.
2. **FetchType.EAGER** : Avec cette option, les données associées sont récupérées de la base de données immédiatement lorsque l'entité parente est récupérée. Cela signifie que toutes les entités associées sont chargées avec l'entité parente, même si elles ne sont pas toujours utilisées. Cela peut entraîner un surcoût en termes de performances, mais cela garantit que toutes les données associées sont disponibles sans nécessiter d'accès supplémentaire à la base de données.

Le choix entre FetchType.LAZY et FetchType.EAGER dépend souvent des besoins de votre application et des performances souhaitées. FetchType.LAZY est généralement préféré lorsque les données associées ne sont pas toujours nécessaires, tandis que FetchType.EAGER est utilisé lorsque les données associées sont toujours nécessaires avec l'entité parente.

**4. Uni vs BiDirectional on 1 to 1 RelationShips**

Dans les relations One-to-One en JPA (Java Persistence API), vous pouvez choisir entre des associations unidirectionnelles (UniDirectional) et bidirectionnelles (BiDirectional) en fonction des besoins de votre application. Voici une comparaison entre les deux :

**Unidirectionnelle (UniDirectional) :**

* **Avantages** :
  1. **Simplicité** : Les associations unidirectionnelles sont plus simples à mettre en œuvre et à comprendre.
  2. **Performance** : Elles peuvent être plus performantes car il n'y a pas de surcharge liée à la gestion de la relation dans les deux sens.
* **Inconvénients** :
  1. **Limitation de la navigation** : Vous pouvez uniquement naviguer de l'entité propriétaire vers l'entité associée, ce qui peut être contraignant selon les besoins de votre application.
  2. **Difficulté d'accès inverse** : Si vous avez besoin de naviguer de l'entité associée vers l'entité propriétaire, vous devez effectuer une requête supplémentaire.

**Bidirectionnelle (BiDirectional) :**

* **Avantages** :
  1. **Facilité de navigation** : Vous pouvez naviguer dans les deux sens entre les entités associées, ce qui offre une plus grande flexibilité dans la manipulation des données.
  2. **Accès inversé** : Vous pouvez accéder à l'entité propriétaire à partir de l'entité associée sans avoir besoin de requêtes supplémentaires.
* **Inconvénients** :
  1. **Complexité accrue** : La gestion des relations bidirectionnelles peut être plus complexe, notamment en ce qui concerne la synchronisation des deux côtés de la relation.
  2. **Surcharge de mémoire** : Les associations bidirectionnelles peuvent entraîner une surcharge de mémoire car chaque côté de la relation doit maintenir une référence vers l'autre côté.

En résumé, les associations unidirectionnelles offrent une simplicité et des performances potentiellement meilleures, tandis que les associations bidirectionnelles offrent une plus grande flexibilité dans la navigation entre les entités. Le choix dépend des besoins spécifiques de votre application en termes de facilité d'utilisation, de performance et de maintenance.

**Example :**

Certainly! Let's provide some code examples to illustrate the differences between unidirectional and bidirectional one-to-one relationships in JPA.

**Unidirectional One-to-One Relationship:**

In a unidirectional one-to-one relationship, only one entity maintains a reference to the other entity. Here's an example:

**Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement**

In this example, **Person** has a unidirectional one-to-one relationship with **Address**. Only the **Person** entity has a reference to the **Address** entity.

**Bidirectional One-to-One Relationship:**

In a bidirectional one-to-one relationship, both entities maintain references to each other. Here's how it looks:

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

In this example, **Person** and **Address** have a bidirectional one-to-one relationship. **Person** maintains a reference to **Address** using **@OneToOne(mappedBy = "person")**, while **Address** maintains a reference to **Person** using **@OneToOne** with a **person\_id** foreign key column.

With this setup, you can access the **Address** from a **Person** and vice versa.

These examples should help illustrate the differences between unidirectional and bidirectional one-to-one relationships in JPA.

**5. Orphan Removal**

L'attribut **orphanRemoval** en JPA est utilisé pour spécifier si les entités "orphelines" (entités qui ne sont plus référencées par aucune autre entité) doivent être automatiquement supprimées de la base de données lorsqu'elles ne sont plus référencées par l'entité parente.

Prenons un exemple pour illustrer son fonctionnement :

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

Dans cet exemple, nous avons une entité **Parent** qui a une relation un-à-un avec une entité **Child**. Nous avons défini **orphanRemoval = true** sur la relation avec **Child**.

Maintenant, si vous supprimez la référence à un **Child** depuis un **Parent** et que vous enregistrez le **Parent**, l'entité **Child** sera automatiquement supprimée de la base de données si elle n'est plus référencée par aucune autre entité. Voici comment vous pouvez le faire :

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

Si **orphanRemoval** était défini sur **false**, l'entité **Child** ne serait pas automatiquement supprimée de la base de données lorsqu'elle n'est plus référencée par aucun **Parent**. Vous devriez alors la supprimer explicitement de la base de données.

**6. @ForeingKey**

**@ForeignKey** est une annotation spécifique à Hibernate qui permet de personnaliser le nom de la contrainte de clé étrangère générée dans la base de données.

Voici un exemple d'utilisation de **@ForeignKey** pour spécifier le nom de la contrainte de clé étrangère :

**Une image contenant texte, Appareils électroniques, capture d’écran, multimédia

Description générée automatiquement**

Dans cet exemple, nous utilisons **@ForeignKey(name = "FK\_commande\_client")** pour spécifier que le nom de la contrainte de clé étrangère pour la relation entre **Commande** et **Client** doit être "FK\_commande\_client".

Cela permet de personnaliser le nom de la contrainte de clé étrangère dans la base de données, ce qui peut être utile dans certaines situations où vous souhaitez avoir un contrôle précis sur le nom des contraintes générées.

1. **Example**

**Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement**

**StudentCard.java**

package com.springdata.springdatajpa.StudentCard;  
  
import com.springdata.springdatajpa.Student.Student;  
import jakarta.persistence.\*;  
  
import java.util.Objects;  
  
@Entity(name= "student\_card")  
@Table(  
 name="student\_card",  
 uniqueConstraints = {  
 @UniqueConstraint(name = "student\_card\_number\_unique",columnNames = "card\_number")  
 }  
)  
public class StudentCard {  
  
 @Id  
 @SequenceGenerator(  
 name = "student\_card\_sequence",  
 sequenceName = "student\_card\_sequence",  
 allocationSize = 1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "student\_card\_sequence"  
 )  
 @Column(  
 name="id",  
 updatable = false,  
 nullable = false  
 )  
 private Long id;  
  
 @Column(  
 name="card\_number",  
 nullable = false,  
 length = 15  
 )  
 private String cardNumber;  
  
 @OneToOne(  
 cascade = {CascadeType.*PERSIST*, CascadeType.*REFRESH*},  
// orphanRemoval = true,  
 fetch = FetchType.*LAZY* )  
 @JoinColumn(  
 name = "student\_id",  
 referencedColumnName = "id",  
 foreignKey = @ForeignKey(  
 name = "student\_id\_fk"  
 )  
 )  
 private Student student;  
  
 public StudentCard(String cardNumber , Student student) {  
 this.cardNumber = cardNumber;  
 this.student=student;  
 }  
  
 public StudentCard() {  
  
 }  
  
 public String getCardNumber() {  
 return cardNumber;  
 }  
  
 public void setCardNumber(String cardNumber) {  
 this.cardNumber = cardNumber;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 StudentCard that = (StudentCard) o;  
 return Objects.*equals*(cardNumber, that.cardNumber);  
 }  
  
 @Override  
 public String toString() {  
 return "StudentCard{" +  
 "id=" + id +  
 ", cardNumber='" + cardNumber + '\'' +  
 '}';  
 }  
}

**Student.java**

package com.springdata.springdatajpa.Student;  
  
import com.springdata.springdatajpa.StudentCard.StudentCard;  
import jakarta.persistence.\*;  
  
@Entity(name= "student")  
@Table(  
 name="student",  
 uniqueConstraints = {  
 @UniqueConstraint(name = "student\_email\_unique",columnNames = "email")  
 }  
)  
public class Student {  
  
 @Id  
 @SequenceGenerator(  
 name = "student\_sequence",  
 sequenceName = "student\_sequence",  
 allocationSize = 1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "student\_sequence"  
 )  
 @Column(  
 name="id",  
 updatable = false,  
 nullable = false  
 )  
 private Long id;  
  
 @Column(  
 name="first\_name",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String firstName;  
  
 @Column(  
 name="last\_name",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String lastName;  
  
 @Column(  
 name="email",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String email;  
  
 @Column(  
 name = "age",  
 nullable = false  
 )  
 private Integer age;  
  
 @OneToOne(  
 mappedBy = "student",  
 cascade = CascadeType.*ALL* )  
 private StudentCard studentCard;  
  
 public Student(String firstName, String lastName, String email, Integer age) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.email = email;  
 this.age = age;  
 }  
  
 public Student() {  
  
 }  
  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "id=" + id +  
 ", firstName='" + firstName + '\'' +  
 ", lastName='" + lastName + '\'' +  
 ", email='" + email + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

**SpringDataJpaApplication:**

@Bean  
CommandLineRunner commandLineRunner(StudentCardRepository StudentCardRepository , StudentRepository studentRepository) {  
 return args -> {  
 Faker faker = new Faker();  
  
 String firstName = faker.name().firstName();  
 String lastName = faker.name().lastName();  
 String email = String.*format*("%s.%s@alHoussam.edu", firstName, lastName);  
 Student student = new Student(  
 firstName,  
 lastName,  
 email,  
 faker.number().numberBetween(17, 55));  
  
 StudentCard studentCard = new StudentCard("123456789123456",student);  
  
 StudentCardRepository.save(studentCard);  
  
 studentRepository.findById(1L)  
 .ifPresent(System.*out*::println);  
  
 StudentCardRepository.findById(1L)  
 .ifPresent(System.*out*::println);  
  
 StudentCardRepository.deleteById(1L);  
 };  
}

1. **One To Many**
   * 1. **Definition**

**Relation Un-à-Plusieurs (One-to-Many)** :

* + Dans une relation un-à-plusieurs, chaque enregistrement dans une table est lié à plusieurs enregistrements dans une autre table, mais chaque enregistrement dans cette autre table est lié à un seul enregistrement dans la première table. Par exemple, un client peut passer plusieurs commandes, mais chaque commande est associée à un seul client.

**Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement**

* + 1. **Many to To Bidirectional**

Dans une relation "Many-to-One" bidirectionnelle, chaque enregistrement dans la table de la relation "Many" peut être lié à un seul enregistrement dans la table de la relation "One", et chaque enregistrement dans la table de la relation "One" peut être lié à plusieurs enregistrements dans la table de la relation "Many".

En français, cela signifie qu'une relation "Many-to-One" bidirectionnelle permet à plusieurs enregistrements dans une table de référencer un seul enregistrement dans une autre table, et cet enregistrement unique peut être référencé par plusieurs enregistrements dans la première table.

Prenons un exemple concret pour illustrer cela. Supposons que nous ayons deux entités : "Équipe" et "Joueur". Une équipe peut avoir plusieurs joueurs, mais chaque joueur ne peut appartenir qu'à une seule équipe. C'est une relation "Many-to-One". De manière bidirectionnelle, cela signifie qu'à partir d'un joueur, nous pouvons accéder à l'équipe à laquelle il appartient, et à partir d'une équipe, nous pouvons accéder à la liste des joueurs qui lui sont associés.

Voici quelques avantages et inconvénients des relations bidirectionnelles dans les bases de données :

**Avantages :**

1. **Navigation aisée** : Dans une relation bidirectionnelle, il est facile de naviguer dans les deux sens entre les entités. Par exemple, à partir d'une entité "Parent", vous pouvez accéder à ses entités "Enfant" associées, et vice versa.
2. **Meilleure cohérence** : Les relations bidirectionnelles permettent souvent une meilleure cohérence des données. Par exemple, si une mise à jour est effectuée dans une entité, cette mise à jour sera automatiquement reflétée dans l'entité associée sans nécessiter de manipulation supplémentaire.
3. **Flexibilité dans la requête** : Vous pouvez effectuer des requêtes dans les deux sens pour récupérer des données associées. Cela offre une plus grande flexibilité dans la création de requêtes et peut simplifier certaines opérations.

**Inconvénients :**

1. **Complexité accrue** : Les relations bidirectionnelles peuvent rendre le modèle de données plus complexe, en particulier lorsque vous devez gérer des cas où la cohérence doit être maintenue entre les deux entités.
2. **Risque de boucle infinie** : Si la relation n'est pas correctement gérée, il existe un risque de boucle infinie lors de la navigation entre les entités. Cela peut entraîner des problèmes de performance et de stabilité.
3. **Surcharge de données** : Dans certains cas, les relations bidirectionnelles peuvent entraîner une surcharge de données, car chaque entité doit conserver une référence à l'autre entité associée.
4. **Difficulté de synchronisation** : Maintenir la synchronisation entre les entités dans une relation bidirectionnelle peut être difficile, surtout lorsque des opérations telles que l'ajout, la suppression ou la mise à jour d'entités sont effectuées.

En résumé, bien que les relations bidirectionnelles offrent une navigation facile et une meilleure cohérence des données, elles peuvent également introduire une complexité accrue et des défis de gestion des données. Il est important de peser les avantages et les inconvénients avant de décider d'utiliser des relations bidirectionnelles dans votre modèle de données.

En Spring Boot, la fonctionnalité de cascade est utilisée pour propager les opérations de persistance (comme l'insertion, la mise à jour ou la suppression) d'une entité parent à ses entités enfants. Dans le cas de relations bidirectionnelles, cela signifie que les opérations effectuées sur une entité parent peuvent être automatiquement propagées à ses entités enfants, et vice versa.

Voici comment vous pouvez utiliser la cascade dans une relation bidirectionnelle en Spring Boot :

1. **Définition de la cascade** : Dans vos entités, vous pouvez utiliser l'annotation **@OneToMany** (pour la relation One-to-Many) et spécifier la cascade appropriée. Par exemple, vous pouvez utiliser **CascadeType.ALL** pour indiquer que toutes les opérations de persistance doivent être propagées aux entités enfants.
2. **Gestion des opérations de persistance** : Lorsque vous effectuez des opérations de persistance (comme l'insertion, la mise à jour ou la suppression) sur l'entité parent, les mêmes opérations seront automatiquement propagées à ses entités enfants en fonction de la cascade définie.
3. **Attention à la boucle infinie** : Lorsque vous utilisez la cascade dans des relations bidirectionnelles, assurez-vous de gérer correctement les associations pour éviter les boucles infinies. Par exemple, si une entité parente cascade ses opérations de persistance à ses entités enfants, et que ces entités enfants sont également associées à l'entité parente, cela peut entraîner une boucle infinie si cela n'est pas correctement géré.
   * 1. **Fetch Eager and Lazy**

En OneToMany relation fetch et Lazy

En français, "fetch eager" et "fetch lazy" font référence aux stratégies de chargement de données dans une relation One-to-Many (un-à-plusieurs) entre entités.

**Fetch Eager (Chargement Immédiat)** :

* + Avec la stratégie de chargement "fetch eager", les données associées sont chargées immédiatement avec l'entité parente lorsqu'elle est récupérée de la base de données. Cela signifie que toutes les entités enfants liées à l'entité parente seront également chargées simultanément.
  + En français, "fetch eager" se traduit souvent par "chargement immédiat". Cela signifie que les données associées sont chargées immédiatement avec l'entité parente.

**Fetch Lazy (Chargement Paresseux)** :

* + Avec la stratégie de chargement "fetch lazy", les données associées ne sont chargées qu'au moment où elles sont explicitement demandées. Cela signifie que les entités enfants ne seront chargées que lorsque vous y accéderez directement.
  + En français, "fetch lazy" se traduit souvent par "chargement paresseux". Cela signifie que les données associées ne sont chargées que lorsqu'elles sont réellement nécessaires, généralement lorsque vous y accédez explicitement dans votre code.

1. **Example**

**Book.java**

package com.springdata.springdatajpa.Book;  
  
import com.springdata.springdatajpa.Student.Student;  
import jakarta.persistence.\*;  
import java.time.LocalDate;  
import java.util.Objects;  
  
@Entity(name = "book")  
@Table(name = "book")  
public class Book {  
  
 @Id  
 @SequenceGenerator(  
 name = "book\_sequence",  
 sequenceName = "book\_sequence",  
 allocationSize = 1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "book\_sequence"  
 )  
 @Column(  
 name = "id",  
 nullable = false,  
 updatable = false  
 )  
 private Long id;  
  
 @Column(  
 name = "book\_name",  
 columnDefinition = "TEXT",  
 nullable = false  
 )  
 private String bookName;  
  
 @Column(  
 name = "created\_at",  
 columnDefinition = "TIMESTAMP",  
 nullable = false  
 )  
 private LocalDate createdAt;  
  
 @ManyToOne  
 @JoinColumn(  
 name = "student\_id",  
 referencedColumnName = "id",  
 foreignKey = @ForeignKey(  
 name = "student\_id\_fk"  
 )  
 )  
 private Student student;  
  
 public Book(String bookName, LocalDate createdAt) {  
 this.bookName = bookName;  
 this.createdAt = createdAt;  
 }  
  
 public Book() {  
  
 }  
  
  
 public String getBookName() {  
 return bookName;  
 }  
  
 public void setBookName(String bookName) {  
 this.bookName = bookName;  
 }  
  
 public LocalDate getCreatedAt() {  
 return createdAt;  
 }  
  
 public void setCreatedAt(LocalDate createdAt) {  
 this.createdAt = createdAt;  
 }  
  
 public Student getStudent() {  
 return student;  
 }  
  
 public void setStudent(Student student) {  
 this.student = student;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Book book = (Book) o;  
 return Objects.*equals*(bookName, book.bookName) && Objects.*equals*(createdAt, book.createdAt);  
 }  
  
 @Override  
 public String toString() {  
 return "Book{" +  
 "id=" + id +  
 ", bookName='" + bookName + '\'' +  
 ", createdAt=" + createdAt +  
 '}';  
 }  
}

**Student.java**

package com.springdata.springdatajpa.Student;  
  
import com.springdata.springdatajpa.Book.Book;  
import com.springdata.springdatajpa.StudentCard.StudentCard;  
import jakarta.persistence.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
@Entity(name= "student")  
@Table(  
 name="student",  
 uniqueConstraints = {  
 @UniqueConstraint(name = "student\_email\_unique",columnNames = "email")  
 }  
)  
public class Student {  
  
 @Id  
 @SequenceGenerator(  
 name = "student\_sequence",  
 sequenceName = "student\_sequence",  
 allocationSize = 1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "student\_sequence"  
 )  
 @Column(  
 name="id",  
 updatable = false,  
 nullable = false  
 )  
 private Long id;  
  
 @Column(  
 name="first\_name",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String firstName;  
  
 @Column(  
 name="last\_name",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String lastName;  
  
 @Column(  
 name="email",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String email;  
  
 @Column(  
 name = "age",  
 nullable = false  
 )  
 private Integer age;  
  
 @OneToMany(  
 mappedBy = "student",  
 cascade = {CascadeType.*PERSIST*,CascadeType.*REMOVE*}  
 )  
 private List<Book> books= new ArrayList<>();  
  
 @OneToOne(  
 mappedBy = "student",  
 cascade = {CascadeType.*PERSIST*,CascadeType.*REMOVE*}  
 )  
 private StudentCard studentCard;  
  
  
  
 public Student(String firstName, String lastName, String email, Integer age) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.email = email;  
 this.age = age;  
 }  
  
 public Student() {  
  
 }  
  
 public Long getId() {  
 return id;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 public StudentCard getStudentCard() {  
 return studentCard;  
 }  
  
 public void setStudentCard(StudentCard studentCard) {  
 this.studentCard = studentCard;  
 }  
  
 public List<Book> getBooks() {  
 return books;  
 }  
  
 public void addBook(Book book){  
 if(!this.books.contains(book)){  
 this.books.add(book);  
 book.setStudent(this);  
 }  
 }  
  
 public void removeBook(Book book){  
 if(books.contains(book)){  
 books.remove(book);  
 book.setStudent(null);  
 }  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "id=" + id +  
 ", firstName='" + firstName + '\'' +  
 ", lastName='" + lastName + '\'' +  
 ", email='" + email + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

**SpringDataJpaApplication.java**

package com.springdata.springdatajpa;  
  
import com.github.javafaker.Faker;  
import com.springdata.springdatajpa.Book.Book;  
import com.springdata.springdatajpa.Student.Student;  
import com.springdata.springdatajpa.Student.StudentRepository;  
import com.springdata.springdatajpa.StudentCard.StudentCard;  
import com.springdata.springdatajpa.StudentCard.StudentCardRepository;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
import org.springframework.data.domain.Page;  
import org.springframework.data.domain.PageRequest;  
import org.springframework.data.domain.Sort;  
  
import java.time.LocalDate;  
import java.util.List;  
  
@SpringBootApplication  
public class SpringDataJpaApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringDataJpaApplication.class, args);  
 }  
  
 @Bean  
 CommandLineRunner commandLineRunner(StudentCardRepository StudentCardRepository , StudentRepository studentRepository) {  
 return args -> {  
 Faker faker = new Faker();  
  
 String firstName = faker.name().firstName();  
 String lastName = faker.name().lastName();  
 String email = String.*format*("%s.%s@alHoussam.edu", firstName, lastName);  
 Student student = new Student(  
 firstName,  
 lastName,  
 email,  
 faker.number().numberBetween(17, 55));  
  
 StudentCard studentCard = new StudentCard("123456789123456",student);  
  
 student.addBook(new Book("one piece", LocalDate.*now*().plusDays(2)));  
 student.addBook(new Book("Naruto", LocalDate.*now*().plusDays(3)));  
 student.addBook(new Book("Dragon Ball ", LocalDate.*now*()));  
  
 student.setStudentCard(studentCard);  
  
 studentRepository.save(student);  
  
 studentRepository.findById(1L)  
 .ifPresent(s->{  
 System.*out*.println("fetch book lazy...");  
 List<Book> books = student.getBooks();  
 books.forEach(book->{  
 System.*out*.println(  
 s.getFirstName() + " borrwed " + book.getBookName()  
 );  
 });  
 });  
//  
// StudentCardRepository.findById(1L)  
// .ifPresent(System.out::println);  
//  
// StudentCardRepository.deleteById(1L);  
 };  
 }  
  
 private void sortingAndPagination(StudentRepository studentRepository){  
 PageRequest pageRequest = PageRequest.*of*(  
 0  
 ,5  
 ,Sort.*by*("firstName").ascending());  
  
 Page<Student> page= studentRepository.findAll(pageRequest);  
 System.*out*.println(page);  
  
 }  
  
 private void sorting(StudentRepository studentRepository){  
 Sort sort= Sort.*by*("firstName").ascending().and(Sort.*by*("age").descending());  
  
 studentRepository.findAll(sort)  
 .forEach(student -> System.*out*.println(student.getFirstName()+" "+student.getAge()));  
 }  
  
  
 private void generateRandomStudents(StudentRepository studentRepository){  
 Faker faker = new Faker();  
 for(int i=0; i<=100; i++){  
 String firstName = faker.name().firstName();  
 String lastName = faker.name().lastName();  
 String email= String.*format*("%s.%s@alHoussam.edu",firstName,lastName);  
 Student student = new Student(  
 firstName,  
 lastName,  
 email,  
 faker.number().numberBetween(17,55));  
 studentRepository.save(student);  
 }  
 }  
  
  
}

1. **Many To Many**
   * 1. **ManyToMAny with JointTable**

**Une image contenant texte, ligne, nombre, Police

Description générée automatiquement**

La relation Many-to-Many avec **@JoinTable** en Spring Boot est une approche courante pour modéliser des relations Many-to-Many entre entités dans une base de données relationnelle à l'aide de JPA (Java Persistence API). Cette approche utilise une table de jointure explicite pour gérer la relation Many-to-Many entre les deux entités.

Voici comment vous pouvez utiliser **@JoinTable** pour définir une relation Many-to-Many entre deux entités dans Spring Boot :

1. **Définir les entités** :
   * Vous définissez deux entités Java représentant les deux côtés de la relation Many-to-Many. Vous les annotez avec **@Entity**.
   * Chaque entité a une collection (par exemple, une liste) d'entités de l'autre côté de la relation.

**Course.java**

package com.springdata.springdatajpa;  
  
import com.springdata.springdatajpa.Student.Student;  
import jakarta.persistence.\*;  
  
import java.util.ArrayList;  
import java.util.List;  
  
@Entity(name = "course")  
@Table(name = "course")  
public class Course {  
  
 @Id  
 @SequenceGenerator(  
 name = "course\_id",  
 sequenceName = "course\_id",  
 allocationSize = 1  
  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "course\_id"  
 )  
 @Column(  
 name = "id",  
 updatable = false,  
 nullable = false  
 )  
 private Long id;  
  
 @Column(  
 name = "name",  
 columnDefinition = "TEXT",  
 nullable = false  
 )  
 private String name;  
  
 @Column(  
 name = "department",  
 columnDefinition = "TEXT",  
 nullable = false  
 )  
 private String department;  
  
 @ManyToMany(  
 mappedBy = "courses"  
 )  
 private List<Student> students= new ArrayList<>();  
  
  
 public Course(String name, String department) {  
 this.name = name;  
 this.department = department;  
 }  
  
 public Course(){  
  
 }  
  
 public Long getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getDepartment() {  
 return department;  
 }  
  
 public void setDepartment(String department) {  
 this.department = department;  
 }  
  
 public List<Student> getStudents(){  
 return students;  
 }  
  
 @Override  
 public String toString() {  
 return "Course{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", department='" + department + '\'' +  
 '}';  
 }  
}

1. **Utiliser @JoinTable** :
   * Vous utilisez l'annotation **@JoinTable** sur l'une des collections pour définir la table de jointure explicite à utiliser pour gérer la relation Many-to-Many.
   * Vous spécifiez le nom de la table de jointure, ainsi que les colonnes qui font référence aux clés primaires des deux entités impliquées dans la relation.

**Student.java**

package com.springdata.springdatajpa.Student;  
  
import com.springdata.springdatajpa.Book.Book;  
import com.springdata.springdatajpa.Course;  
import com.springdata.springdatajpa.StudentCard.StudentCard;  
import jakarta.persistence.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
@Entity(name= "student")  
@Table(  
 name="student",  
 uniqueConstraints = {  
 @UniqueConstraint(name = "student\_email\_unique",columnNames = "email")  
 }  
)  
public class Student {  
  
 @Id  
 @SequenceGenerator(  
 name = "student\_sequence",  
 sequenceName = "student\_sequence",  
 allocationSize = 1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "student\_sequence"  
 )  
 @Column(  
 name="id",  
 updatable = false,  
 nullable = false  
 )  
 private Long id;  
  
 @Column(  
 name="first\_name",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String firstName;  
  
 @Column(  
 name="last\_name",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String lastName;  
  
 @Column(  
 name="email",  
 nullable = false,  
 columnDefinition = "TEXT"  
 )  
 private String email;  
  
 @Column(  
 name = "age",  
 nullable = false  
 )  
 private Integer age;  
  
 @OneToMany(  
 mappedBy = "student",  
 cascade = {CascadeType.*PERSIST*,CascadeType.*REMOVE*}  
 )  
 private List<Book> books= new ArrayList<>();  
  
 @OneToOne(  
 mappedBy = "student",  
 cascade = {CascadeType.*PERSIST*,CascadeType.*REMOVE*}  
 )  
 private StudentCard studentCard;  
  
 @ManyToMany(  
 cascade = {CascadeType.*PERSIST*,CascadeType.*REMOVE*}  
 )  
 @JoinTable(  
 name = "enrolment",  
 joinColumns = @JoinColumn(  
 name = "student\_id",  
 foreignKey = @ForeignKey(name = "enrolment\_student\_id\_fk")  
 ),  
 inverseJoinColumns = @JoinColumn(  
 name = "course\_id",  
 foreignKey = @ForeignKey(name = "enrolment\_course\_id\_fk")  
 )  
 )  
 private List<Course> courses = new ArrayList<>();  
  
  
 public Student(String firstName, String lastName, String email, Integer age) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.email = email;  
 this.age = age;  
 }  
  
 public Student() {  
  
 }  
  
 public Long getId() {  
 return id;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 public StudentCard getStudentCard() {  
 return studentCard;  
 }  
  
 public void setStudentCard(StudentCard studentCard) {  
 this.studentCard = studentCard;  
 }  
  
 public List<Book> getBooks() {  
 return books;  
 }  
  
 public void addBook(Book book){  
 if(!this.books.contains(book)){  
 this.books.add(book);  
 book.setStudent(this);  
 }  
 }  
  
 public void removeBook(Book book){  
 if(books.contains(book)){  
 books.remove(book);  
 book.setStudent(null);  
 }  
 }  
  
 public void enrolToCourse(Course course){  
 courses.add(course);  
 course.getStudents().add(this);  
 }  
  
 public void unEnrolToCourse(Course course){  
 courses.remove(course);  
 course.getStudents().remove(this);  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "id=" + id +  
 ", firstName='" + firstName + '\'' +  
 ", lastName='" + lastName + '\'' +  
 ", email='" + email + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

Dans cet exemple, nous avons défini une table de jointure nommée "inscription" pour gérer la relation Many-to-Many entre les entités **Etudiant** et **Cours**. La table de jointure a deux colonnes de clé étrangère : "etudiant\_id" faisant référence à la clé primaire de l'étudiant et "cours\_id" faisant référence à la clé primaire du cours.

L'utilisation de **@JoinTable** permet de définir explicitement la structure de la table de jointure, ce qui offre plus de contrôle sur la gestion de la relation Many-to-Many dans votre application Spring Boot.

**SpringDataJpaApplication.java**

String firstName = faker.name().firstName();  
String lastName = faker.name().lastName();  
String email = String.*format*("%s.%s@alHoussam.edu", firstName, lastName);  
Student student = new Student(  
 firstName,  
 lastName,  
 email,  
 faker.number().numberBetween(17, 55));

StudentCard studentCard = new StudentCard("123456789123456",student);  
  
student.addBook(new Book("one piece", LocalDate.*now*().plusDays(2)));  
student.addBook(new Book("Naruto", LocalDate.*now*().plusDays(3)));  
student.addBook(new Book("Dragon Ball ", LocalDate.*now*()));  
  
student.setStudentCard(studentCard);  
  
student.enrolToCourse(new Course("Computer Science","IT"));  
student.enrolToCourse(new Course("Amigoscode Spring Data Jpa","IT"));  
student.enrolToCourse(new Course("CodeEvolution","TECH"));  
  
  
studentRepository.save(student);

**Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement**