Crud App Using Spring Boot and Hibernate

Spring Data Repository

- It aims to reduce the boilerplate code required to implement the DAO layer into a project for various persistence stores. Spring Data repository has different interfaces, each having different functionality.

Hibernate Association Mapping (Source - https://www.baeldung.com/jpa-hibernate-associations)

- Associations are a fundamental concept in ORM, allowing us to define relationships between entities.

Unidirectional Associations

- Unidirectional associations are commonly used in object-oriented programming to establish relationships between entities. However, it's important to note that in a unidirectional association, only one entity holds a reference to the other.
- To define a unidirectional association in Java, we can use annotations such as
  - **@ManyToOne**
  - **@OneToMany**,
  - **@OneToOne**, and
  - **@ManyToMany**.
- By using these annotations, we can create a clear and well-defined relationship between two entities in our code.
  - **ONE-TO-MANY Relationship**
    - In a one-to-many relationship, an entity has a reference to one or many instances of another entity.
    - A common example is the relationship between a Department and its Employees. Each Department has many Employees, but each Employee belongs to one Department only.

```
@Entity
public class Department {

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

    @OneToMany
    @JoinColumn(name = "department_id")
    private List<Employee> employees;
}
```

```
@Entity
public class Employee {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```

- Here, the Department entity has a reference to a list of Employee entities. The @OneToMany annotation specifies that this is a one-to-many association. The @JoinColumn annotation specifies the foreign key column in the Employee table referencing the Department table.

- **MANY-TO-ONE Relationship**
  - In a many-to-one relationship, many instances of an entity are associated with one instance of another entity.
  - For example, let's consider Student and School. Each Student can be enrolled in one School only, but each School can have multiple Students.

```
@Entity
public class Student {

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

    private String name;

    @ManyToOne
    @JoinColumn(name = "school_id")
    private School school;
}
```

```
@Entity
public class School {

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

    private String name;
}
```

- **ONE-TO-ONE Relationship**
  - In a one-to-one relationship, an instance of an entity is associated with only one instance of another entity.
  - A common example is the relationship between an Employee and a ParkingSpot. Each Employee has a ParkingSpot, and each ParkingSpot belongs to one Employee.

```
    @Entity
    public class Employee {

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

```
        private String name;

        @OneToOne
        @JoinColumn(name = "parking_spot_id")
        private ParkingSpot parkingSpot;

    }
```

```
@Entity
public class ParkingSpot {

    @Id
    private Long id;

}
```

- **MANY-TO-MANY Relationship**
  - In a many-to-many relationship, many instances of an entity are associated with many instances of another entity.
  - Suppose we have two entities – Book and Author. Each Book can have multiple Authors, and each Author can write multiple Books. In JPA, this relationship is represented using the @ManyToMany annotation.

```
@Entity
public class Book {

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

  private String title;

  @ManyToMany
  @JoinTable(name = "book_author",
          joinColumns = @JoinColumn(name = "book_id"),
          inverseJoinColumns = @JoinColumn(name = "author_id"))
  private Set<Author> authors;

}
```

```
@Entity
public class Author {

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

  private String name;
}
```

  - Here, we can see a many-to-many unidirectional association between Book and Author entities. The @ManyToMany annotation specifies that each Book can have multiple Authors, and each Author can write

multiple Books. The @JoinTable annotation specifies the name of the join table and the foreign key columns to join the Book and Author entities.

Bidirectional Associations

- A bidirectional association is a relationship between two entities where each entity has a reference to the other.
- In order to define bidirectional associations, we use the ***mappedBy*** attribute in the @OneToMany and @ManyToMany annotations.
- However, it's important to note that only relying on unidirectional associations may not be sufficient, as bidirectional associations provide additional benefits.
  - **ONE-TO-MANY Bidirectional Association**
    - In a one-to-many bidirectional association, an entity has a reference to another entity. Additionally, the other entity has a collection of references to the first entity.
    - For instance, a Department entity has a collection of Employee entities. Meanwhile, an Employee entity has a reference to the Department entity it belongs.

```
@Entity
public class Department {

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;

}
```

```
@Entity
public class Employee {

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

}
```

    - In the Department entity, we use the @OneToMany annotation to specify the relationship between the Department entity and the Employee entity. The mappedBy attribute specifies the name of the attribute in the Employee entity that owns the relationship. In this case, the Department entity doesn't own the relationship, so we specify mappedBy = "department".
    - In the Employee entity, we use the @ManyToOne annotation to specify the relationship between the Employee entity and the Department entity. The @JoinColumn annotation specifies the name of the foreign key column in the Employee table referencing the Department table.
  - **MANY-TO-MANY Bidirectional Associations**
    - When dealing with a many-to-many bidirectional association, it's important to understand that each entity involved will have a collection of references to the other entity.
    - To illustrate this concept, let's consider the example of a Student entity that has a collection of Course entities and a Course entity that in turn has a collection of Student entities. By establishing such a bidirectional association, we enable both entities to be aware of each other and make it easier to navigate and manage their relationship.

- Here’s an example of how to create a many-to-many bidirectional association:

```
@Entity
public class Student {

    @ManyToMany(mappedBy = "students")
    private List<Course> courses;

}
```

```
@Entity
public class Course {

    @ManyToMany
    @JoinTable(name = "course_student",
        joinColumns = @JoinColumn(name = "course_id"),
        inverseJoinColumns = @JoinColumn(name = "student_id"))
    private List<Student> students;

}
```

- In the Student entity, we use the @ManyToMany annotation to specify the relationship between the Student entity and the Course entity. The mappedBy attribute specifies the attribute’s name in the Course entity that owns the relationship. In this case, the Course entity owns the relationship, so we specify mappedBy = “students”.
- In the Course entity, we use the @ManyToMany annotation to specify the relationship between the Course entity and the Student entity. The @JoinTable annotation specifies the name of the join table that stores the relationship.