UNIT 11 CRUD APPLICATION USING SPRING BOOT AND HIBERNATE

Structure **Page No.**

11.0 INTRODUCTION

Nowadays, there are varieties of data stores such as relational databases, NoSQL databases like MongoDB, Casandraetc, Big Data solutions such as Hadoop etc. Different storage technologies have different configurations, different methods and different API's to fetch the data.

Spring Data simplifies and makes homogeneous data access technologies for relational and non-relational databases, cloud-based data services and map-reduce frameworks. Spring Data provides an abstraction that allows us to connect in the same way to relational databases and NoSQL databases. Thus, switching can be done effortlessly between data stores.

Spring Data JPA is one of the many sub-projects of **Spring Data**that focuses on simplification of the data access for relational data stores. It is not a JPA provider; instead, it wraps the JPA provider and adds its own features like a **no-code implementation** of the repository pattern. Hibernate is the default JPA provider into Spring Data JPA. Spring Data JPA is very flexible, and other providers can also be configured as JPA providers. Spring Data JPA provides a complete abstraction over the DAO layer into the project.

The actual strength of Spring Data JPA lies in the ***repository abstraction***. The ***repository abstraction*** enables us to write the business logic code on a higher abstraction level. Developers do not need to worry about DAO layer implementations;instead they just need to learn Spring Data repository interfaces.

Spring Data provides an implementation for these repository interfaces out-of-the-box.

The Hibernate mappings are one of the key features of Hibernate. Association in Hibernate tells the relationship between the objects of POJO classes, i.e. how the entities are related to each other. The established relationship can be either unidirectional or bidirectional. The supported relationships are similar to the database relationships such as **One-to-One, Many-to-Many, Many-to-One/One-to-Many**.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- describe Spring Data Repository such as CrudRepository, PagingAndSortingRepository and JpaRepository,
- elaborate Hibernate association mappings,
- describe Hibernate Performance optimization via Eager and Lazy loading,
- demonstrate Hibernate Cascading and its impact,
- create records using Spring Data JPA,
- get records using Spring Data JPA,
- update record using Spring Data JPA, and
- delete record using Spring Data JPA

11.2 SPRING DATA REPOSITORY

The previous unit gave us the overview of Spring Data JPA. This section explains the Spring Data Repository to create the persistence layer into the Spring Boot application.

The ***Spring Data repository*** 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. Thefollowing interfaces in the Spring Data repository have been explained.

- CrudRepository
- PagingAndSortingRepository
- JpaRepository

JpaRepository extends PagingAndSortingRepository interface, which extends CrudRepository. Thus, **JpaRepository contains all API of CrudRepositoty and PagingAndSortingRepository.** The persistent layer can extend any of the above interfaces based on required APIs. The implementation of persistent layers is very easy and is shown asan example.

```
@Entity
publicclass Book
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        privateintid;

        private String title;
// getter setter
}
```

Following Repository provides a simple operation – find a Book based on its title.

```
@Repository
publicinterfaceBookRepositoryextendsCrudRepository<Book, Long>
{
publicBook findByTitle(String bookTitle);
}
```

The Spring Data Repository will auto-generate the implementation based on the method name. Supported keywords inside method name can be found at - https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#jpa.query-methods.query-creation

11.2.1 CrudRepository

This interface provides all basic CRUD operation related APIs. The methods provided in CrudRepository are shown in the code.

```
publicinterfaceCrudRepository<T, ID>extends Repository<T, ID>
{
        <S extends T> S save(S entity);
        <S extends T>Iterable<S>saveAll(Iterable<S>entities);
        Optional<T>findById(ID id);
        booleanexistsById(IDid);
        Iterable<T>findAll();
        Iterable<T>findAllById(Iterable<ID>ids);
        long count();
        voiddeleteById(ID id);
        void delete(T entity);
        voiddeleteAll(Iterable<? extends T>entities);
        voiddeleteAll();
}
```

- save(…) - Saves a given entity.
- saveAll(…) - Saves all given entities.
- findById(…) - Retrieves an entity by its id.
- existsById(…) - Returns whether an entity with the given id exists.
- findAll(…) - Returns all instances of the type.
- findAllById(…) - Returns all instances of the type with the given IDs
- count(…) - Returns the number of entities available.
- deleteById(…) - Deletes the entity with the given id.
- delete(…) - Deletes a given entity.
- deleteAll(…) - Deletes the given entities.
- deleteAll() - Deletes all entities managed by the repository.

11.2.2 PagingAndSortingRepository

PagingAndSortingRepository interface extends CrudRepository interface. It provides the paging and sorting feature apart from CRUD feature. The methods in the interface are shown in the code.

```
publicinterfacePagingAndSortingRepository<T, ID>extendsCrudRepository<T, ID>
{
        Iterable<T>findAll(Sort sort);

        Page<T>findAll(Pageablepageable);
}
```

- findAll(Sort sort) - Returns all entities sorted by the given options.
- findAll(Pageablepageable) - Returns a page of entities meeting the paging restriction provided in the pageable object.

11.2.3 JpaRepository

JpaRepository extends PagingAndSortingRepository interface, which extends CrudRepository. Thus, **JpaRepository contains all API of CrudRepositoty and PagingAndSortingRepository.** The methods available in JpaRepository are shown in the code.

```
publicinterfaceJpaRepository<T, ID>extendsPagingAndSortingRepository<T, ID>,
QueryByExampleExecutor<T>
{

        @Override
        List<T>findAll();

        @Override
        List<T>findAll(Sort sort);

        @Override
        List<T>findAllById(Iterable<ID>ids);

        @Override
        <S extends T> List<S>saveAll(Iterable<S>entities);

        void flush();

        <S extends T> S saveAndFlush(S entity);

        voiddeleteInBatch(Iterable<T>entities);

        voiddeleteAllInBatch();

        T getOne(ID id);

        @Override
        <S extends T> List<S>findAll(Example<S>example);

        @Override
        <S extends T> List<S>findAll(Example<S>example, Sort sort);
}
```

☛Check Your Progress 1:

1) What is the relationship between JPA, Hibernate and Spring data JPA?

.....

.....

.....

.....

2) What is Spring Data?

.....

.....

.....

.....

3) What is the difference between CrudRepository and JpaRepository? Which canextend and when?

.....

.....

.....

.....

11.3 HIBERNATE ASSOCIATION MAPPINGS

The previous unit has explained that hibernate can identify POJO classes as persistent only when they are annotated with certain annotations. While making the POJO classes as persistent entities using JPA annotations, we may face situations where two entities can be related and must be referenced from each other, either uni-directional or bi-directional. Association mappings are one of the key features of JPA and Hibernate. That establishes the relationship between two database tables as attributes in your model and allows you to easily navigate the association in your model and JPQL or criteria queries.

When only one pair of entities contains a reference to the other, the association is **unidirectional**. If the pair of entities contains a reference to each other, then it is referred to as **bi-directional**. Entities can contain references to other entities, either directly as an embedded property or field or indirectly via a collection of some sort (arrays, sets, lists, etc.). There is no impact of unidirectional or bidirectional association on database mapping. **Foreign Key Relationships** are used to represent the associations in the underlying tables.

JPA and Hibernate have the same association as you are aware of relational databases. The followings are the association that will be described in subsequent sections.

- one-to-many/many-to-one
- one-to-one
- many-to-many

11.3.1 One-to-Many / Many-to-One

One-to-Many and Many-to-One relationships are the opposite sides of the same coin and closely related. **One-to-Many** mapping is described as one row in a table and is mapped to multiple rows in another table.

For the illustration of the One-to-Many relationship, the **Teacher** and their **Courses** will be taken. A teacher can give multiple courses but a course which is given by only one teacher is an example of one-to-many relationship. While another perspective, many courses are given by a teacher, is an example of the many-to-one relationship. Before diving into details of how to map this relationship, let's create the entities.

```
@Entity
publicclass Teacher
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        private String name;

}


@Entity
publicclass Course
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        privateintid;
        private String title;

}
```

Now, the **Teacher** class should include a list of courses, and mapping will be required to map this relationship into the database. This will be annotated with a @OnetoMany annotation.

```
@Entity
publicclass Teacher
{

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

        private String name;

        @OneToMany(cascade = CascadeType.ALL)
        private List<Course>courses;
}
```

With the above configuration, Hibernate uses an association table to map the relationship. With the above configuration, three tables named ***teacher***, ***course*** and ***teacher_courses*** will be created. But, definitely, we are not looking for such mapping. In order to avoid the third table teacher_courses, **@JoinColumn** annotation is used to specify the foreign key column.

```
@Entity
publicclass Teacher
{

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

        private String name;

        @OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "teacher_id", referencedColumnName = "id")
        private List<Course>courses;
}
```

The tables created into the database and save() operation on teacher object will result in the following into the database (as shown in Figure 11.1, 11.2 and 11.3):

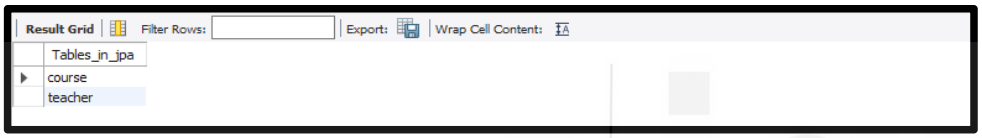

Figure11.1: Hibernate Created Tables

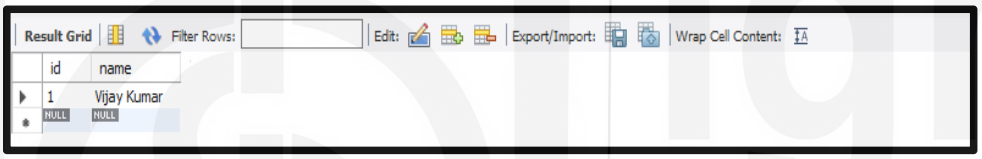

Figure11.2: Hibernate Created Records Into teacher Table

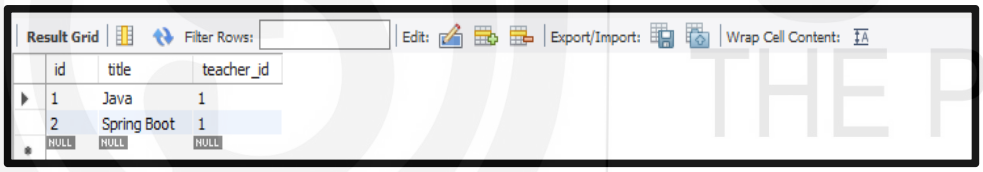

Figure11.3: Hibernate Created Records Into course Table

11.3.1.1 Owning Side and Bi-directionality

The previous example has defined the Teacher class as the owning side of the One-to-Many relationship. This is because the Teacher class defines the join column between the two tables. The **Course** is called the referencing side in the relationship.

Course class can be made as to the owning side of the relationship by mapping the Teacher field with @ManytoOne in the course class instead.

```
@Entity
publicclass Teacher
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        private String name;

}
@Entity
publicclassCourse
{
        @Id
```

```
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        privateintid;
        private String title;

        @ManyToOne(cascade = CascadeType.ALL)
        @JoinColumn(name = "teacher_id", referencedColumnName = "id")
        private Teacher teacher;

}
```

This time **@ManytoOne** annotation has been used instead of **@OnetoMany** annotation. In the above example, it can be observed that the Teacher class does not have a list of courses. Uni-directional relationship has been used here; thus, only one of the entities has reference to another entity. **@ManytoOne** will result similar to **@OnetoMany**.

Note: *It's a good practice to put the owning side of a relationship in the class/table where the foreign key will be held.*

According to best practice, the second version of code using **@ManytoOne**is better. If you look at the second version of the code, the Teacher class does not offer to access the list of courses. This can be achieved by the bidirectional relationship.

The following section explains the bi-directional one-to-many mapping between teacher and courses with a complete example. The Spring Hibernate project structure is shown in Figure 11.4 and the Source Code structure follows;

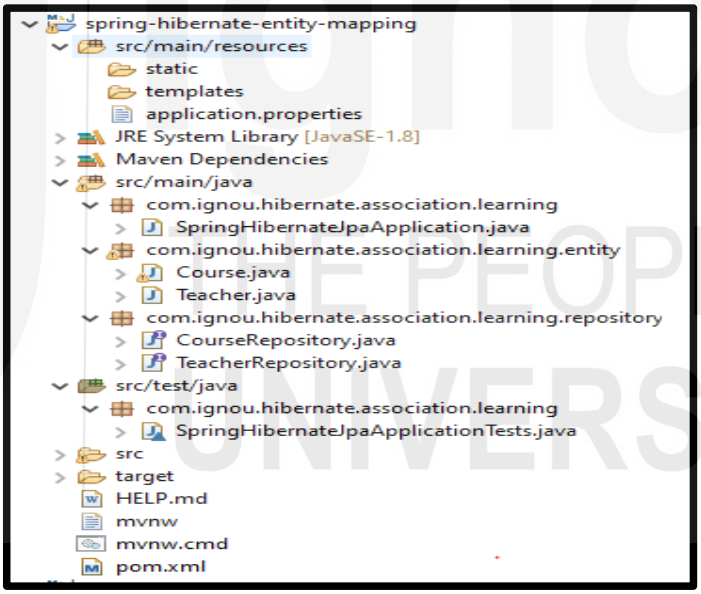

Figure11.4: Spring Hibernate Project Structure

Teacher.java

```
packagecom.ignou.hibernate.association.learning.entity;

@Entity
publicclass Teacher
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private String name;

        @OneToMany(mappedBy = "teacher", cascade= CascadeType.ALL, fetch =
```

```
FetchType.EAGER)
        private List<Course>courses;

        public Teacher() {}

        public Teacher(String name)
        {
                super();
                this.name = name;
        }

        public Long getId()
        {
                returnid;
        }

        publicvoidsetId(Long id)
        {
                this.id = id;
        }

        public String getName()
        {
                returnname;
        }

        publicvoidsetName(String name)
        {
                this.name = name;
        }

        public List<Course>getCourses()
        {
                returncourses;
        }

        publicvoidsetCourses(List<Course>courses)
        {
                this.courses = courses;
        }

        @Override
        public String toString()
        {
        return"Teacher [id=" + id + ", name=" + name + ", courses=" + courses + "]";
        }

}
```

The Teacher class has reference courses. **Teacher** and **Course** has a relationship as one-to-many, and thus, it is annotated with **@OnetoMany. @OnetoMany**annotation has an attribute **mappedBy.** Without mappedBy attribute, we would not have a two-way relationship. The ***mappedBy*** attributes indicatesthe JPA that the field is already mapped by another entity. It's mapped by the teacher field of the Course entity. Other attributes such as cascade and fetch will be explained in the next section.

Course.java

```
packagecom.ignou.hibernate.association.learning.entity;

@Entity
publicclass Course
{
```

```
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	privateintid;
	private String title;

	@ManyToOne
	@JoinColumn(name = "teacher_id", referencedColumnName = "id")
	private Teacher teacher;

	public Course()
	{

	}
	public Course(String title)
	{
		this.title = title;
	}
	publicintgetId()
	{
		returnid;
	}
	publicvoidsetId(intid)
	{
		this.id = id;
	}
	public String getTitle()
	{
		returntitle;
	}
	publicvoidsetTitle(String title)
	{
		this.title = title;
	}
	public Teacher getTeacher()
	{
		returnteacher;
	}
	publicvoidsetTeacher(Teacher teacher)
	{
		this.teacher = teacher;
	}
	@Override
	public String toString()
	{
		return"Course [id=" + id + ", title=" + title + "]";
	}
}
```

Course class is having the reference of Teacher. **@ManytoOne** annotation is used since the Course and Teacher are having many to one relationship. Course class is the owning side since @JoinColumn is defined in this class.

TeacherRepository.java

```
packagecom.ignou.hibernate.association.learning.repository;

publicinterfaceTeacherRepositoryextendsCrudRepository<Teacher, Long>
{
	// No code repository. Spring data JPA will provide the implementation.
}
```

CourseRepository.java

```
package com.ignou.hibernate.association.learning.repository;

public interface CourseRepository extends CrudRepository<Course, Long>
{
        // No code repository. Spring data JPA will provide the implementation.
}
```

SpringHibernateJpaApplication.java

```
package com.ignou.hibernate.association.learning;

@SpringBootApplication
public class SpringHibernateJpaApplication implements CommandLineRunner
{
@Autowired
private TeacherRepository teacherRepo;

public static void main(String[] args)
{
SpringApplication.run(SpringHibernateJpaApplication.class, args);
   }

@Override
public void run(String... args) throws Exception
{
    Teacher t1 = new Teacher("Vijay Kumar");

    Course c1 = new Course("Java");
    Course c2 = new Course("Spring Boot");

    List<Course> courseList = new ArrayList<Course>();
courseList.add(c1);
courseList.add(c2);

    t1.setCourses(courseList);
    c1.setTeacher(t1);
    c2.setTeacher(t1);

    t1 = teacherRepo.save(t1);
   Teacher t = teacherRepo.findById(t1.getId()).get();
System.out.println(t);

}
}
```

11.3.1.2 Eager vs Lazy loading

It is worth noting that a mapped relationship should not impact the software's memory by putting too many unnecessary entities. For example, consider that the Course is a heavy object, i.e. having too many attributes. For some business logic, all **Teacher** objects are loaded from the database. Business logic does not need to retrieve or use courses in it but **Course** objects are still being loaded alongside the **Teacher** objects.

Such loading will degrade the performance of applications. Technically, this can be solved by using the Data Transfer Object Design Pattern and retrieving Teacher information *without* the courses.

JPA has considered all such problems ahead and made **One-to-Many relationships load *lazily* by default**. Lazy loading means relationships will be loaded when it is

actually needed. The execution of **@OnetoMany** relationship with the default fetch type will issue two queries. First is for the Teacher object and the second for the Course objects when it's needed. The log of the query issued by Hibernate is as follows.

```
2021-03-06 17:52:47.095 DEBUG 8124 --- [  restartedMain] org.hibernate.SQL                : select
teacher0_.id as id1_1_0_, teacher0_.name as name2_1_0_ from teacher teacher0_ where
teacher0_.id=?
2021-03-06 17:52:47.096 TRACE 8124 --- [  restartedMain] o.h.type.descriptor.sql.BasicBinder      :
binding parameter [1] as [BIGINT] - [1]
2021-03-06 17:52:47.104 TRACE 8124 --- [  restartedMain] o.h.type.descriptor.sql.BasicExtractor   :
extracted value ([name2_1_0_] : [VARCHAR]) - [Vijay Kumar]
2021-03-06 17:52:47.111 TRACE 8124 --- [  restartedMain] org.hibernate.type.CollectionType        :
Created collection wrapper: [com.ignou.hibernate.association.learning.entity.Teacher.courses#1]
2021-03-06 17:52:47.117 DEBUG 8124 --- [  restartedMain] org.hibernate.SQL                : select
courses0_.teacher_id as teacher_3_0_0_, courses0_.id as id1_0_0_, courses0_.id as id1_0_1_,
courses0_.teacher_id as teacher_3_0_1_, courses0_.title as title2_0_1_ from course courses0_ where
courses0_.teacher_id=?
```

While the execution of **@OnetoMany** relationship with fetch type as ***Eager*** will issue only one query, Log of the query issued by Hibernate is as follows.

```
2021-03-06 18:28:11.107 DEBUG 15728 --- [  restartedMain] org.hibernate.SQL                : select
teacher0_.id as id1_1_0_, teacher0_.name as name2_1_0_, courses1_.teacher_id as teacher_3_0_1_,
courses1_.id as id1_0_1_, courses1_.id as id1_0_2_, courses1_.teacher_id as teacher_3_0_2_,
courses1_.title as title2_0_2_ from teacher teacher0_ left outer join course courses1_ on
teacher0_.id=courses1_.teacher_id where teacher0_.id=?
2021-03-06 18:28:11.107 TRACE 15728 --- [  restartedMain] o.h.type.descriptor.sql.BasicBinder      :
binding parameter [1] as [BIGINT] - [1]
```

Many-to-One **relationships are *eager* by default**.A eager relationship means all related entities are loaded at the same time. The default behavior of fetch type can be changed with fetch attribute as shown below-

```
@OneToMany(mappedBy = "teacher", cascade= CascadeType.ALL, fetch =
FetchType.EAGER)
private List<Course>courses;

@ManyToOne(fetch = FetchType.LAZY)
private Teacher teacher;
```

11.3.1.3 Cascading Operations

JPA operations affect the only entity on which it has been performed. These operations will not affect the other entities that are related to it. For example, In case of Person-Address relationship, the Address entity doesn't have any meaning of its own without a Person entity. Thus, on delete of Person entity, Address entity should also get deleted.

Cascading is about JPA actions involving one entity propagating to other entities via an association. JPA provides **javax.persistence.CascadeType** enumerated types that define the cascade operations. These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many. JPA provides the following cascade type to perform cascading operations.

Cascade Type	Description
ALL	CascadeType.ALL propagates **all operations** including hibernate specific from a parent to a child entity.
PERSIST	CascadeType.PERSIST propagates the **save()** or **persist()** operation to the related entities.
MERGE	CascadeType.MERGE propagates only the **merge()** operation to the related entities.
REFRESH	CascadeType.REFRESH propagates only the **refresh()** operation to the related entities.
REMOVE	CascadeType.REMOVE removes all related entities association when the owning entity is deleted.
DETACH	CascadeType.DETACH detaches all related entities if owning entity is detached.

Hibernate Cascade Type

Hibernate provides three additional Cascade Types along with provided by JPA. Hibernate provides *org.hibernate.annotations.CascadeType*enumerated types that define the cascade operations.

- CascadeType.REPLICATE
- CascadeType.SAVE_UPDATE
- CascadeType.Lock

Clear understanding of **One-to-Many/Many-to-One** association, as explained above, will help you to understand other associations. The following section will explain the other association as bidirectional with example.

11.3.2 One-to-One

One-to-One mapping is described as one row in a table mapped to only one row in another table.

For the illustration of One-to-One relationship, The **Student** and the **Address** have been taken.

11.3.2.1 Implementation with Foreign key

ER diagram of foreign key based one-to-one mapping is shown in Figure 11.5. The address_id column in Student is the foreign key to the address.

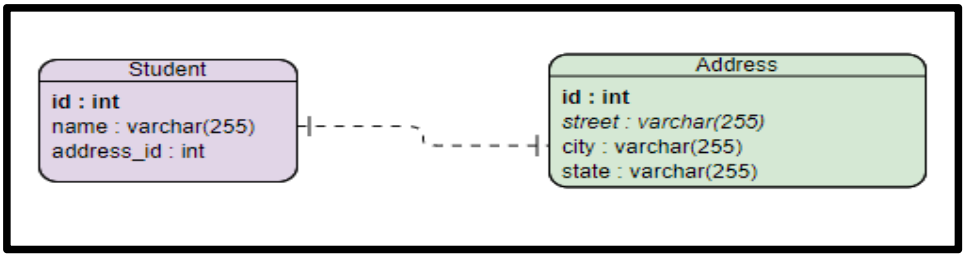

Figure 11.5: ER Diagram For One-to-One Mapping With Foreign Key

```
@Entity
publicclass Student
{

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
        private Long id;

        private String name;
        @OneToOne(cascade = CascadeType.ALL)
        @JoinColumn(name="address_id", referencedColumnName = "id")
        private Address address;

          // getter setter
}
```

The Student and the Address has a one-to-one relationship. Entity Student comprises another entity Address, and it is annotated with **@OnetoOne** annotation. Check the cascade attribute into the annotation. Student entity is the owner, and thus, it has **@JoinColumn** annotation on Address entity. @JoinColumn is to configure the name of the column in the Student table that maps to the primary key in the address table. If the name attribute is not provided in @JoinColumn, Hibernate will follow some rules to select the default name of the column.

```
@Entity
publicclass Address
{

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        privateintid;
        private String street;
        private String state;
        private String pincode;
        private String country;
        @OneToOne(mappedBy = "address")
        private Student student;

          // getter setter

}
```

The Address entity is having @OneToOne annotation on another entity Student with attribute mappedBy. The mappedBy attribute is used since the Address is non-owning. Note that on both entities @OneToOne annotations are used since its bidirectional.

11.3.2.2 Implementation with shared Primary key

In this strategy, instead of creating a new column *address_id*, **we'll mark the primary key column (*student_id*) of the *address* table as the foreign key to the *student* table**. ER diagram is shown in Figure 11.6. This strategy optimizes the storage space.

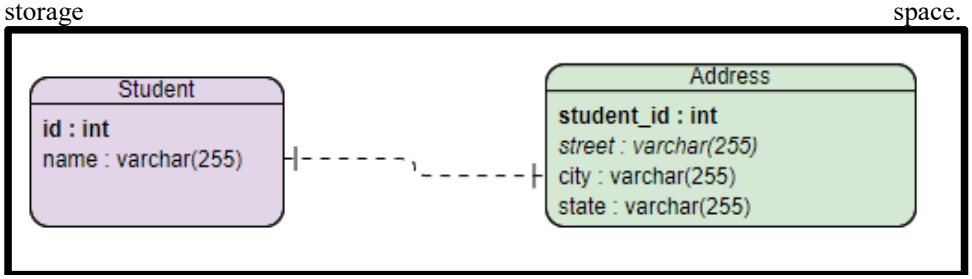

Figure 11.6: ER Diagram For One-to-One Mapping With Shared Primary Key

```
@Entity
publicclass Student
{

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

        private String name;

        @OneToOne(mappedBy = "student",  cascade = CascadeType.ALL)
        @PrimaryKeyJoinColumn
        private Address address;

          // getter setter
}
```

The **@PrimaryKeyJoinColumn**annotation indicates that the primary key of Student entity is used as the foreign key value for the associated Address entity. The mappedBy attribute is used here because the foreign key is now present in the address table.

```
@Entity

publicclass Address

{


        @Id

        @Column(name ="student_id")

        privateintid;

        private String street;

        private String state;

        private String pincode;

        private String country;


        @OneToOne

          @MapsId

          @JoinColumn(name="student_id")

        private Student student;


          // getter setter

}
```

The identifier of the Address entity is still having @Id annotation but it no longer uses @GeneratedValue annotation. Instead it references the student_id column. The **@MapsId**indicates that the primary key values will be copied from the Student entity.

11.3.3 Many-to-Many

Many-to-Many mapping is described as many rows in a table mapped to many rows in another table.

For the illustration of Many-to-Many relationship, **Book** and the **Author** have been taken. A book may be written by multiple writers and a writer will write multiple books. Thus, the Book and Author relationship is many-to-many and is shown as an ER diagram in Figure 11.7.

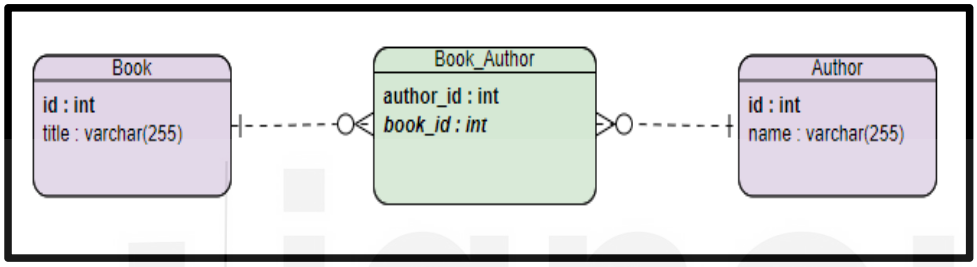

Figure11.7: ER Diagram For Many-to-Many Relationship

The JPA implementation for many-to-many mapping for Book as owning side is shown below:

```
@Entity
@Table(name = "books")
publicclass Book
{

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

//Other fields

        @ManyToMany(cascade = CascadeType.ALL)
        @JoinTable(name = "book_author" , joinColumns = @JoinColumn(name =
"book_id" ), inverseJoinColumns = @JoinColumn(name = "author_id"))
        private List<Author>authors;

//getter setter

}
```

@JoinTable and @JoinColumnaren't mandatory annotations. These are used to customize the name of the table and column name. **@ManyToMany**annotation has been used for many-to-many mapping. **@JoinTable**annotationhas a name attribute which defines the name of the joining table. It also requires the foreign keys with the *@JoinColumn* annotations**. The *joinColumn* attribute will connect to the owner side of the relationship, and the *inverseJoinColumn* to the other side.**

```
@Entity
@Table(name = "authors")
publicclass Author
```

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

        @Column(name = "author_name")
        @Size(max = 100)
        @NotNull
        private String name;


        private Integer rating;

        @ManyToMany(mappedBy = "authors", cascade = CascadeType.ALL)
        private List<Book>books;
}
```

☛Check Your Progress 2:

1) Explain One-to-Many association in Hibernate. Write an example for uni-directional one-to-many mapping.

.....

.....

.....

.....

2) Explain Eager and Lazy loading in Hibernate. How does it improve the performance of hibernate applications?

.....

.....

.....

.....

3) What are the different Cascade types supported by JPA?

.....

.....

.....

.....

11.4 CREATE RECORDS USING SPRING BOOT AND HIBERNATE

Crud Application Using Spring Boot and Hibernate

The previous unit explained Spring Data JPA and hibernate as the default provider for Spring Data JPA. This unit has elaborated on Spring Data Repository and hibernate association mapping. This section provides the complete Rest Application to create records using Spring Boot and Hibernate by using all the explained concepts. The example considers the many-to-many relationship between **Book** and **Author.** Many books can be written by an Author and many authors can write a book. The same example will be considered for CRUD operation.The required tools and software are as follows:

- Eclipse IDE
- Maven
- Spring Boot
- Hibernate
- Mysql
- Postman

Visit Spring Initializr (https://start.spring.io) in order to generate the spring project (shown in Figure 11.8). Add the required dependencies and export the project into eclipse as maven project. Modify the data source properties into application.properties file and do some hands-on.

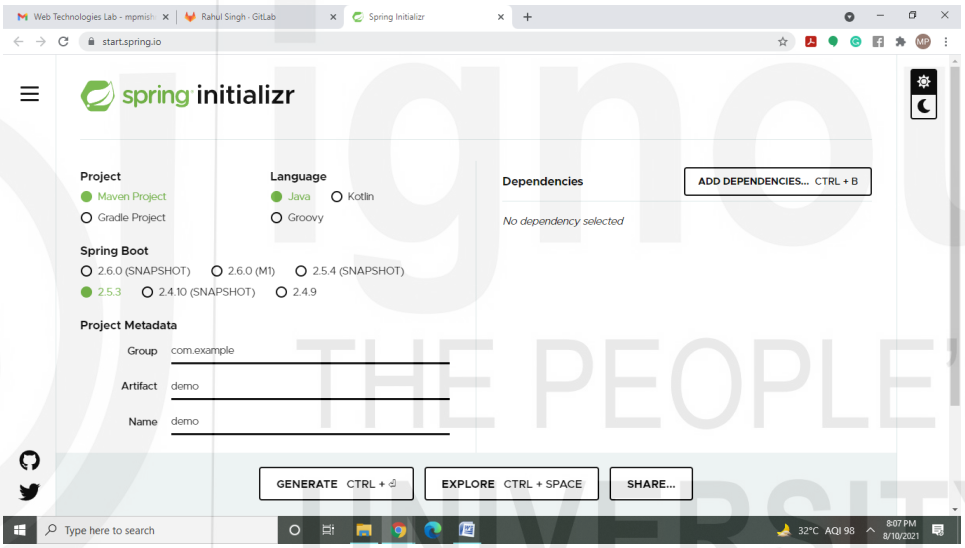

Figure11.8: SpringBoot Project Setup Using Spring Initializr

The source code structure is shown below.

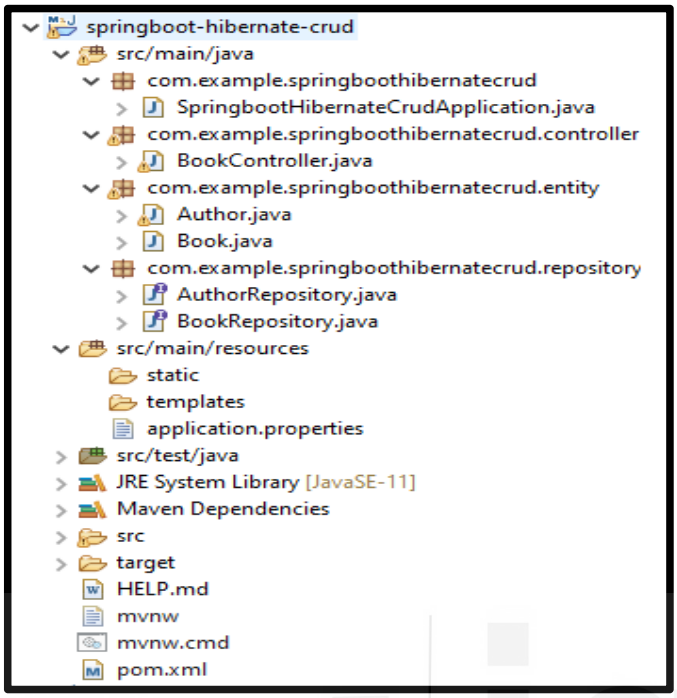

Figure 11.9: SpingBootAnd Hibernate CRUD Applicatication Folder Structure

SpringbootHibernateCrudApplication.java – This is the main class of Spring Boot application. To create the records at the time of application start, it implements CommandLineRunner interface and creates records using book repository.

```
@SpringBootApplication
@EnableJpaRepositories
publicclassSpringbootHibernateCrudApplicationimplementsCommandLineRunner
{
	@Autowired
	privateBookRepositorybookRepository;

	publicstaticvoid main(String[] args)
	{
		SpringApplication.run(SpringbootHibernateCrudApplication.class, args);
	}

	@Override
	publicvoid run(String... args) throws Exception
	{
		Book b1 = new Book("Java In Action", "ISBN-001");
		List<Book>bookList = newArrayList<Book>();
		bookList.add(b1);

		Author au1 = new Author("O. P. Sharma");
		Author au2 = new Author("K L Mishra");

		List<Author>authorList = newArrayList<Author>();
		authorList.add(au1);
		authorList.add(au2);

		b1.setAuthors(authorList);
		au1.setBooks(bookList);
		au2.setBooks(bookList);

		bookRepository.save(b1);
```

```
                Book b2 = new Book("Java 8 In Action", "ISBN-002");
                List<Book>bookList2 = newArrayList<Book>();
                bookList2.add(b2);

                Author au3 = new Author("R K Singh");
                Author au4 = new Author("A K Sharma");

                List<Author>authorList2 = newArrayList<Author>();
                authorList2.add(au3);
                authorList2.add(au4);

                b2.setAuthors(authorList2);
                au3.setBooks(bookList2);
                au4.setBooks(bookList2);

                bookRepository.save(b2);

        }
}
```

BookController.java – The ***BookController*** is responsible to handle the request from client for the URL pattern **/books/***. The controller is having a method ***createBookRecord*** which is annotated with ***@PostMapping*** which means it will accept the post request for the URL pattern **/books** with the request body having a valid Book entity json.

```
@RestController
@RequestMapping("/books")
publicclassBookController
{
        @Autowired
        privateBookRepositorybookRepository;
        @Autowired
        privateAuthorRepositoryauthorRepository;
        @PostMapping
        public Book createBookRecord(@Valid@RequestBody Book b)
        {
                returnbookRepository.save(b);
        }
}
```

In the example, there are two entities Book and Author, having a many-to-many relationship. Entities with bidirectional many-to-many associations are shown in the code.

Book.java

```
@Entity
@Table(name = "books")
publicclass Book
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
	private Long id;

	@NotNull
	@Size(max=75)
	private String title;

	@NotNull
	@Size(max = 100)
	private String isbn;

	@ManyToMany(cascade = CascadeType.ALL)
	@JoinTable(joinColumns = @JoinColumn(name = "book_id" ), inverseJoinColumns
	= @JoinColumn(name = "author_id"))
	private List<Author>authors;

	public Book() {}

	public Book(@NotNull@Size(max = 75) String title, @NotNull@Size(max = 100)
	String isbn)
	{
		this.title = title;
		this.isbn = isbn;
	}

	// getter setter

	@Override
	public String toString()
	{
	return"Book [id=" + id + ", title=" + title + ", isbn=" + isbn + ", authors=" + authors
	+ "]";
	}
}
```

Author.java

```
@Entity
@Table(name = "authors")
publicclass Author
{

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

	@Column(name = "author_name")
	@Size(max = 100)
	@NotNull
	private String name;

	private Integer rating;

	@ManyToMany(mappedBy = "authors", cascade = CascadeType.ALL)
	@JsonIgnore
	private List<Book>books;

	public Author() {}

	public Author(@Size(max = 100) @NotNull String name)
	{
```

```
                super();
                this.name = name;
        }

        // getter setter

        @Override
        public String toString()
        {
                return"Author [id=" + id + ", name=" + name + ", rating=" + rating + "]";
        }
}
```

BookRepository.java – BookRepository provides the methods to interact with databases to perform CRUD operation. BookRepository extends CrudRepository interface. To create the records, the default save() method of CrudRepository will be used.

```
@Repository
publicinterfaceBookRepositoryextendsCrudRepository<Book, Long>
{
   // No implementation is required.
}
```

application.properties

```
# DATASOURCE (DataSourceAutoConfiguration&DataSourceProperties)
spring.datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false&serverTimezone=UTC
&useLegacyDatetimeCode=false
spring.datasource.username=root
spring.datasource.password=root

# Hibernate
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

# Hibernateddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = create

logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE

spring.jpa.properties.hibernate.enable_lazy_load_no_trans=true
```

Start the rest application by executing **SpringbootHibernateCrudApplication.java.** Use the postman to send the post request for book record creation. Sample post request with postman for book record creation is shown in Figure 11.10. Check the response status as **200 OK** and response body with generated id for new record.

Figure 11.10: Create Record Using SpringBoot and Hibernate

11.5 READ RECORDS USING SPRING BOOT AND HIBERNATE

This section explains how to read the records and search the records using Spring Boot and Spring Data JPA with Hibernate. The previous section example will be used to explain the read operation. We will just add the new feature into the existing rest application. Add the new features into BookRepository and BookController as shown below in order to test the read records operation.

BookRepository.java

```
@Repository
publicinterfaceBookRepositoryextendsCrudRepository<Book, Long>
{
        public List<Book>findByTitleContaining(String pattern);     // 1

        public List<Book>findByTitleContaininsIgnoreCase(String pattern); //2

        public List<Book>findByTitleLike(String pattern);           // 3

        @Query("Select b from Book b where b.title like %:pattern%")
        public List<Book>findByTitle(String pattern);               // 4

}
```

BookRepository extends the CrudRepository interface, which provides the methods to read all records, specific record by id etc. Here we have added three new methods in order to provide search features. All three methods generate the same ***like queries,*** but there are some differences.

1. Query methods using Containing, Contains, and IsContaining generate the like query. For example, the query method findByTitleContaining(String pattern) generates the following sql query.

   SELECT * FROM books WHERE title LIKE '%<pattern>%'

2. The ***findByTitleContainsIgnoreCase(String pattern)*** is similar to ***findByTitleContaining(String pattern)*** but with case insensitive.

3. Spring Data JPA also provides a Like keyword, but it behaves differently. An example to call the **findByTitle(…)** is shown inBookRepository code in the next paragraph.
4. Sometimes we need to create queries that are too complicated for Query Methods or would result in absurdly long method names. In those cases, we can use the @Query annotation to query our database.
    - Named Parameters (**:name**) – Query parameters can be passed as named parameters into the query. Check the example in the aboveBookRepository.
    - Ordinal Parameters (**?index**) - JPQL also supports ordinal parameters, whose form is **?index**. An example of ordinal parameters is as "**Select b from Book b where b.title like %?1%"**

BookController.java

```
@RestController
@RequestMapping("/books")
publicclassBookController
{
	@Autowired
	privateBookRepositorybookRepository;

	@GetMapping
	publicIterable<Book>getBooks()
	{
		returnbookRepository.findAll();
	}
	@GetMapping("/search-by-method-containing")
	public List<Book>searchByMethod(@RequestParam(value = "q", required = true)
	String pattern)
	{
		returnbookRepository.findByTitleContaining(pattern);
	}
	@GetMapping("/search-by-method-like")
	public List<Book>searchByMethodLike(@RequestParam(value = "q", required =
	true) String pattern)
	{
		returnbookRepository.findByTitleLike("%"+pattern+"%");
	}
	@GetMapping("/search-by-query")
	public List<Book>searchByQuery(@RequestParam(value = "q", required = true)
	String pattern)
	{
		returnbookRepository.findByTitle(pattern);
	}
}
```

The BookController provides the following **GET** endpoints to get all books and search the book with title keywords.

- http://localhost:8080/books - Returns all the books with its authors.
- http://localhost:8080/search-by-method-containing?q=Java – Returns all the books with title containing Java
- http://localhost:8080/search-by-method-like?q=Java – Returns all the books with title containing Java.
- http://localhost:8080/search-by-query?q=Java – Returns all the books with title containing Java

11.6 UPDATE RECORDS USING SPRING BOOT AND HIBERNATE

To update the record/records, CrudRepository provides save(…) and saveAll(…) methods. If the entity has an identifier, JPA will perform update operation instead of save operation. Add the following into the BookController.java in order to provide the **update** feature to the rest application.

BookController.java

```
@RestController
@RequestMapping("/books")
publicclassBookController
{

        @Autowired
        privateBookRepositorybookRepository;

        @PutMapping
        public Book updateBook(@Valid@RequestBody Book b)
        {
                returnbookRepository.save(b);
        }

}
```

To test the update feature, use the postman and use the PUT method with a valid Book record into the request body. Following request, update the authors of the given book with id 3 (as shown in Figure 11.11).

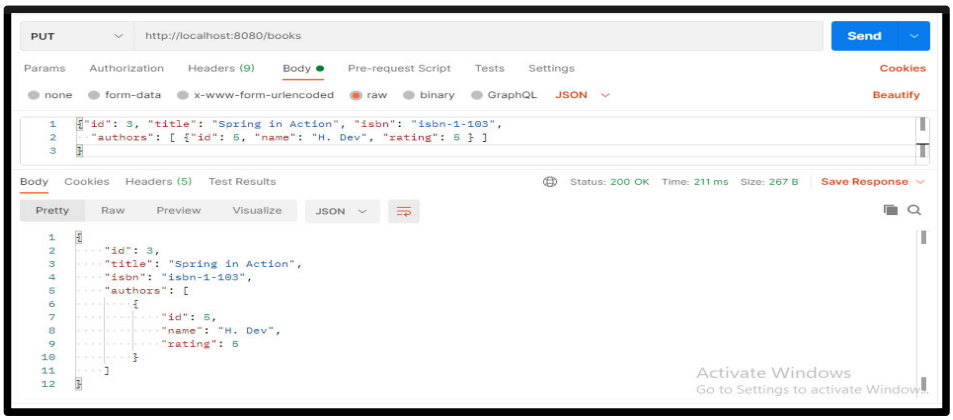

Figure 11.11: Update Record Using SpringBoot and Hibernate

11.7 DELETE RECORDS USING SPRING BOOT AND HIBERNATE

To delete a record or all records, CrudRepository provides deleteById(…), delete(…) and deleteAll(…) methods. Add the following code into the BookController.java in order to provide the **delete** feature into the rest application.

```
@RestController
@RequestMapping("/books")
publicclassBookController
{

	@Autowired
	privateBookRepositorybookRepository;

	@DeleteMapping("/{bookId}")
	@ResponseStatus(value = HttpStatus.OK)
	publicvoiddeleteBook(@PathVariable(value="bookId", required=true) Long bookId)
	{
		bookRepository.deleteById(bookId);
	}
}
```

BookController.java

The delete operation does not return any body response. It just returns the response status as 200 OK on successful deletion. Use the postman to issue the **DELETE** request, as shown into the screenshot (Figure 11.12). Thefollowing is the delete request to delete the book with id as 2. The response status is **200 OK,** and the response body is empty as expected.

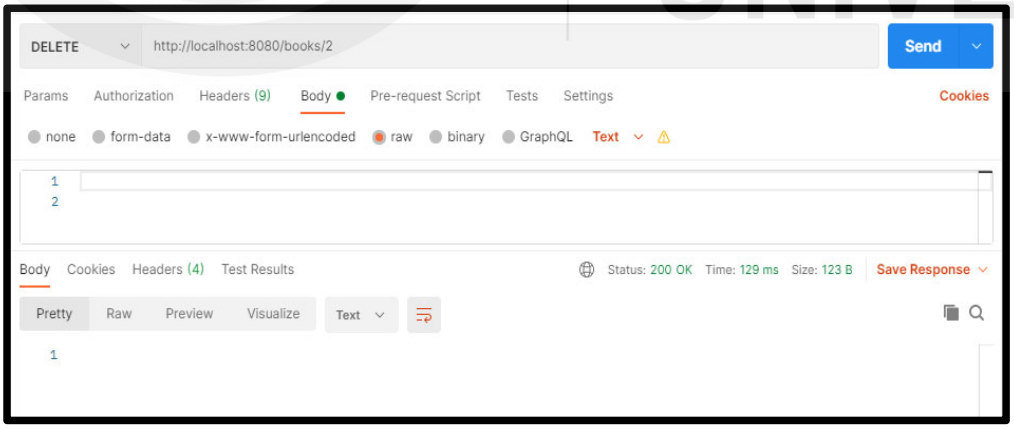

Figure 11.12: Delete Record Using SpringBoot and Hibernate

☛Check Your Progress 3

1) Explain the annotation **@OneToMany** and its attributes :

```
@OneToMany(cascade=CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;
```

2) How does Spring Data framework derive queries from method names?

3) Explain the following User repository definition.

```
@Repository
public interface UserRepository extends Repository<User, Long> {
List<User>findByEmailAddressAndLastname(String emailAddress, String lastname);
}
```

4) Explain the following repository definition.

```
@Repository
publicinterfaceBookRepositoryextendsCrudRepository<Book, Long>
{
        public List<Book>findByTitleContaining(String pattern);     // 1
        public List<Book>findByTitleContaininsIgnoreCase(String pattern); //2
        public List<Book>findByTitleLike(String pattern);           // 3
                @Query("Select b from Book b where b.title like %:pattern%")
        public List<Book>findByTitle(String pattern);               // 4

}
```

11.8 SUMMARY

This unit has explained Spring Data Repository with a complete example using other hibernate association concepts. For better understanding, start coding by using the examples explained in this unit. The important points are as following:

- The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.
- CrudRepository, PagingAndSortingRepository and JpaRepositoryhave been explained with all available methods.
- Association mappings are one of the key features of JPA and Hibernate. That establishes the relationship between two database tables as attributes in your model and allows you to navigate the association easily in your model and JPQL or criteria queries
- JPA has considered the related entity loading overhead problems ahead and made One-to-Many relationships load lazily by default.
- The fetch type for Many-to-One relationship is eager by default.
- Cascading is about JPA actions involving one entity propagating to other entities via an association. JPA provides javax.persistence.CascadeType enumerated types that define the cascade operations. JPA provides the cascade types as ALL, PERSIST, MERGE, REFRESH, REMOVE, DETACH.
- Query methods using Containing, Contains, and IsContaining generate the like query. For example, query method findByTitleContaining(String pattern) generates the sql query as SELECT * FROM books WHERE title LIKE '%<pattern>%'
- Query parameters as Named Parameters (:name) and Ordinal Parameters (?index)

11.9 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 1

1) Java Persistence API (JPA) is a specification that defines APIs for object-relational mappings and management of persistent objects. JPA is a set of interfaces that can be used to implement the persistence layer. JPA itself doesn't provide any implementation classes.
   Hibernate is an implementation of JPA.
   Spring Data JPA is one of the many sub-projects of Spring Data that focuses on simplification of the data access for relational data stores. Spring Data JPA is not a JPA provider; instead it wraps the JPA provider and adds its own

features like a no-code implementation of the repository pattern. Spring Data JPA uses Hibernate as the default JPA provider.

2) There are varieties of data stores such as relational databases, NoSQL databases like MongoDB, Casandraetc, Big Data solutions such as Hadoop etc. Different storage technologies have different configurations, different methods and different API's to fetch the data.
Spring Data aims at providing a homogeneous and consistent Spring-based programming model to fetch data along with keeping the special traits of the underlying data stores. Spring Data provides Abstractions (interfaces) that can be used irrespective of the underlying data source.
public interface EmployeeRepository extends CrudRepository<Employee, Long> {}
No implementation of EmployeeRepository is required in order to perform insert, update, delete and retrieval of EMPLOYEE entities from the database.

3) CrudRepository provides all the typical Save, Update, Delete operations for an entity while JpaRepository extends CrudRepository as well as **Repository**, **QueryByExampleExecutor** and **PagingAndSortingRepository** interfaces.
So, if only CRUD operations are required to perform, select CRUDRepository. Else, if one or more features out of Pagination, Batch processing and flush operations are required, choose JPARepository.

Check Your Progress 2

1) One-to-Many and Many-to-One relationships are the opposite sides of the same coin and closely related. **One-to-Many** mapping is described as one row in a table mapped to multiple rows in another table.
For the illustration of the One-to-Many relationship, the **Teacher** and their **Courses** will be taken. A teacher can give multiple courses, but a course is given by only one teacher is an example of one-to-many relationship. While another perspective, many courses are given by a teacher is an example of the many-to-one relationship.

```
@Entity
publicclass Teacher
{

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

        private String name;

        @OneToMany(cascade = CascadeType.ALL)
  @JoinColumn(name = "teacher_id", referencedColumnName = "id")
        private List<Course>courses;

}


@Entity
publicclass Course
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        privateintid;
        private String title;

}
```

2) The difference between lazy and eager loading is described in the table.

S.No.	Key	Lazy	Eager
1	Fetching strategy	In Lazy loading, associated data loads only when we explicitly call getter or size method.	In Eager loading, data loading happens at the time of their parent is fetched
2	Default Strategy in ORM Layers	ManyToMany and OneToMany associations used a lazy loading strategy by default.	ManyToOne and OneToOne associations used a lazy loading strategy by default.
3	Loading Configuration	It can be enabled by using the annotation parameter : fetch = FetchType.LAZY	It can be enabled by using the annotation parameter : fetch = FetchType.EAGER
4	Performance	Initial load time much smaller than Eager loading	Loading too much unnecessary data might impact performance

Performance improvement can be explained with an example. Consider the teacher and course as a one-to-many relationship. The **Course** is a heavy object i.e. having too many attributes. For some business logic, all **Teacher** objects are loaded from the database. Business logic does not need to retrieve or use courses in it, but **Course** objects are still being loaded alongside the **Teacher** objects.
Such loading will degrade the performance of applications. JPA has considered all such problems ahead and made **One-to-Many relationships load *lazily* by default**. Lazy loading means relationships will be loaded when it is actually needed.

3) One of the main aspects of JPA is that it helps to propagate Parent to Child relationships. This behavior is possible through CascadeTypes. The CascadeTypes supported by JPA are:

Cascade Type	Description
ALL	CascadeType.ALL propagates **all operations,** including hibernate specific from a parent to a child entity.
PERSIST	CascadeType.PERSIST propagates the **save()** or **persist()** operation to the related entities.
MERGE	CascadeType.MERGE propagates only the **merge()** operation to the related entities.
REFRESH	CascadeType.REFRESH propagates only the **refresh()** operation to the related entities.
REMOVE	CascadeType.REMOVE removes all related entities associations when the owning entity is deleted.
DETACH	CascadeType.DETACH detaches all related entities if the owning entity is detached.

Cascading is useful only for Parent - Child associations where the parent entity state transitions cascade to the child entity. The cascade configuration option accepts an array of CascadeTypes. The below example shows how to add the refresh and merge operations in the cascade operations for a One-to-Many relationship:

```
@OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE},
fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;
```

Check Your Progress 3

1) **One-to-Many** mapping is described as one row in a table is mapped to multiple rows in another table and Hibernate provides annotation @OnetoMany. The attribute "cascade=CascadeType.ALL" means that any change that happens to Employee Entity must cascade to all associated entities (Accounts Entity in this case) also. This means that if you save an employee into the database, then all associated accounts will also be saved into the database. If an Employee is deleted, then all accounts associated with that Employee will also be deleted. However, if we want to cascade only the save operation but not the delete operation, we need to clearly specify it using below code.

```
@OneToMany(cascade=CascadeType.PERSIST, fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;
```

Now only when the save() or persist() methods are called using an employee instance, the accounts will be persisted. If any other method is called on session, its effect will not affect/cascade to the accounts entity.

2) Spring Data framework has an in-built query builder mechanism that derives queries based on the method name. Method names are defined as 'find...By..', 'count...By...' etc. 'By' acts as the delimiter to identify the start of the criteria. 'And' and 'Or' are used to define multiple criteria. 'OrderBy' is used to identify the order by criteria. Few examples of query methods are as follows:
   - **findByFirstname(String firstName):** It derives the query to get the list based on the first name.
   - **findByFirstnameAndLastname(String firstName, String lastName):** It derives the query to get the list based on first name and last name
   - **findByFirstnameAndLastnameOrderByFirstnameAcs(String firstName, string last):** it derives the query to get the list based on the first name and sorted by the first name in ascending order.

3) UserRepository extends the Repository marker interface. Marker interface does not contain any method. Thus, UserRepository provides a single API to find the list of users based on the email address and last name. The query method:
   List<User>findByEmailAddressAndLastname(String emailAddress, String lastname) translates into the following query:
   select u from User u where u.emailAddress = ?1 and u.lastname = ?2.

4) BookRepository extends the CrudRepository interface, which provides the methods to read all records, specific record by id etc. All methods generate the same queries, but there are some differences.
   1. Query methods using Containing generates the following sql query.
      SELECT * FROM books WHERE title LIKE '%<pattern>%'

2. The ***findByTitleContainsIgnoreCase(String pattern)*** is similar to ***findByTitleContaining(String pattern)*** but with case insensitive.
3. The **findByTitleLike(String pattern)** generates **SELECT * FROM books WHERE title LIKE '<pattern>'**. JPA Like keyword behaves differently. It can be noticed in the generated query that it does not include wild char %. Thus, while calling this method, argument should have wild char included.
4. Sometimes we need to create queries that are too complicated for Query Methods or would result in absurdly long method names. In those cases, we can use the @Query annotation to query our database. The method findByTitile with @Query annotation is similar to the first method.

11.10 REFERENCES/FURTHER READING

- Craig Walls, "Spring Boot in action" Manning Publications, 2016. (https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf)
- Christian Bauer, Gavin King, and Gary Gregory, "Java Persistence with Hibernate",Manning Publications, 2015.
- Ethan Marcotte, "Responsive Web Design", Jeffrey Zeldman Publication, 2011(http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf)
- Tomcy John, "Hands-On Spring Security 5 for Reactive Applications",Packt Publishing,2018
- https://spring.io/projects/spring-data
- https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa
- https://github.com/spring-projects/spring-data-examples
- https://dzone.com/articles/magic-of-spring-data
- https://www.journaldev.com/17034/spring-data-jpa
- https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/associations.html
- https://www.baeldung.com/spring-data-rest-relationships