Spring Boot JDBC:

Spring Boot JDBC:

Spring Boot JDBC is a part of the Spring Boot framework that simplifies the use of JDBC (Java Database Connectivity) for database operations in your Java applications. Spring JDBC is a framework that provides an abstraction layer on top of JDBC. This makes it easier to write JDBC code and reduces the amount of boilerplate code that needs to be written. Spring JDBC also provides a number of features that make it easier to manage database connections, handle transactions, and execute queries. Spring Boot builds on top of the Spring Framework and provides additional features and simplifications to make it easier to work with databases.

Here are some key aspects of Spring Boot JDBC:

Data Source Configuration: Spring Boot simplifies the configuration of data sources for your application. It can automatically configure a data source for you based on the properties you specify in the application configuration files **application.properties** or **application.yml**

Template Classes: Spring Boot includes a set of template classes, such as **JdbcTemplate**, that simplify database operations. These templates provide higher-level abstractions for executing SQL queries, managing connections, and handling exceptions.

Exception Handling: Spring Boot JDBC helps manage database-related exceptions. It translates database-specific exceptions into more meaningful, standardized Spring exceptions, making error handling easier and more consistent.

Connection Pooling: Connection pooling is a technique for efficiently managing database connections. Spring Boot can configure a connection pool for your data source, helping improve application performance by reusing existing database connections.

Transaction Management: Spring Boot simplifies transaction management in JDBC applications. It allows you to use declarative transaction annotations or programmatic transaction management with ease.

Here are some concepts of Spring JDBC:

DataSource: DataSource is a JDBC object that represents a connection to a database. Spring provides a number of data source implementations, such as **DriverManagerDataSource**.

JdbcTemplate: The **org.springframework.jdbc.core.JdbcTemplate** is a central class in Spring JDBC that simplifies database operations. It encapsulates the common JDBC operations like executing queries, updates, and stored procedures. It handles resource management, exception handling, and result set processing.

RowMapper: A RowMapper is an interface used to map rows from a database result set to Java objects. It defines a method to convert a row into an object of a specific class.

Transaction management: Spring provides built-in transaction management capabilities through declarative or programmatic approaches. You can easily define transaction boundaries and have fine-grained control over transaction behaviour with Spring JDBC.

Overall, Spring Boot JDBC is a powerful framework that can make it easier to write JDBC code. However, it is important to be aware of the limitations of Spring JDBC before using it. Using Spring JDBC, you can perform typical CRUD (Create, Read, Update, Delete) operations on databases without dealing with the boilerplate code typically required in JDBC programming.

Here are some of the basic steps involved in using Spring Boot JDBC:

- Create a JdbcTemplate.
- Execute a JDBC query.

Let's see few methods of spring JdbcTemplate class.

- **int update(String query)** is used to insert, update and delete records.
- **void execute(String query)** is used to execute DDL query.
- **List query(String query, RowMapper rm)** is used to fetch records using RowMapper.

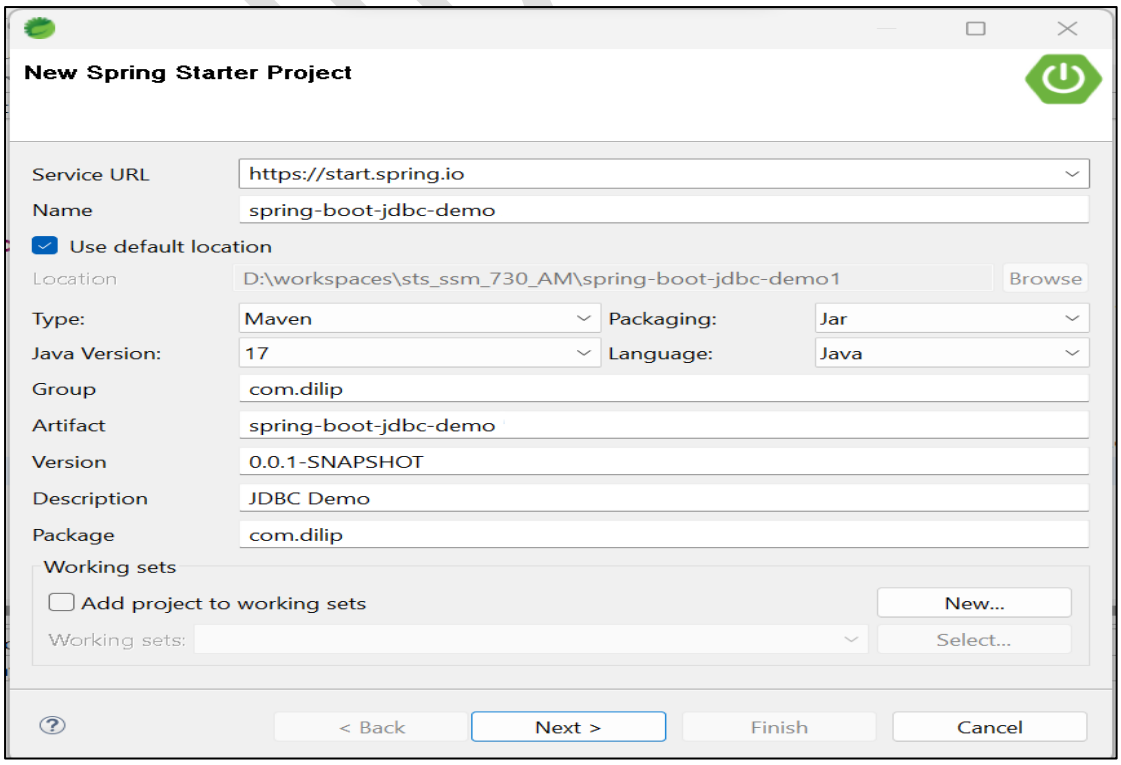
Similarly Spring / Spring Boot JDBC module provided many predefined classes and methods to perform all database operations whatever we can do with JDBC API.

NOTE: Please be ready with Database table before writing JDBC logic.

Steps for Spring Boot JDBC Project:

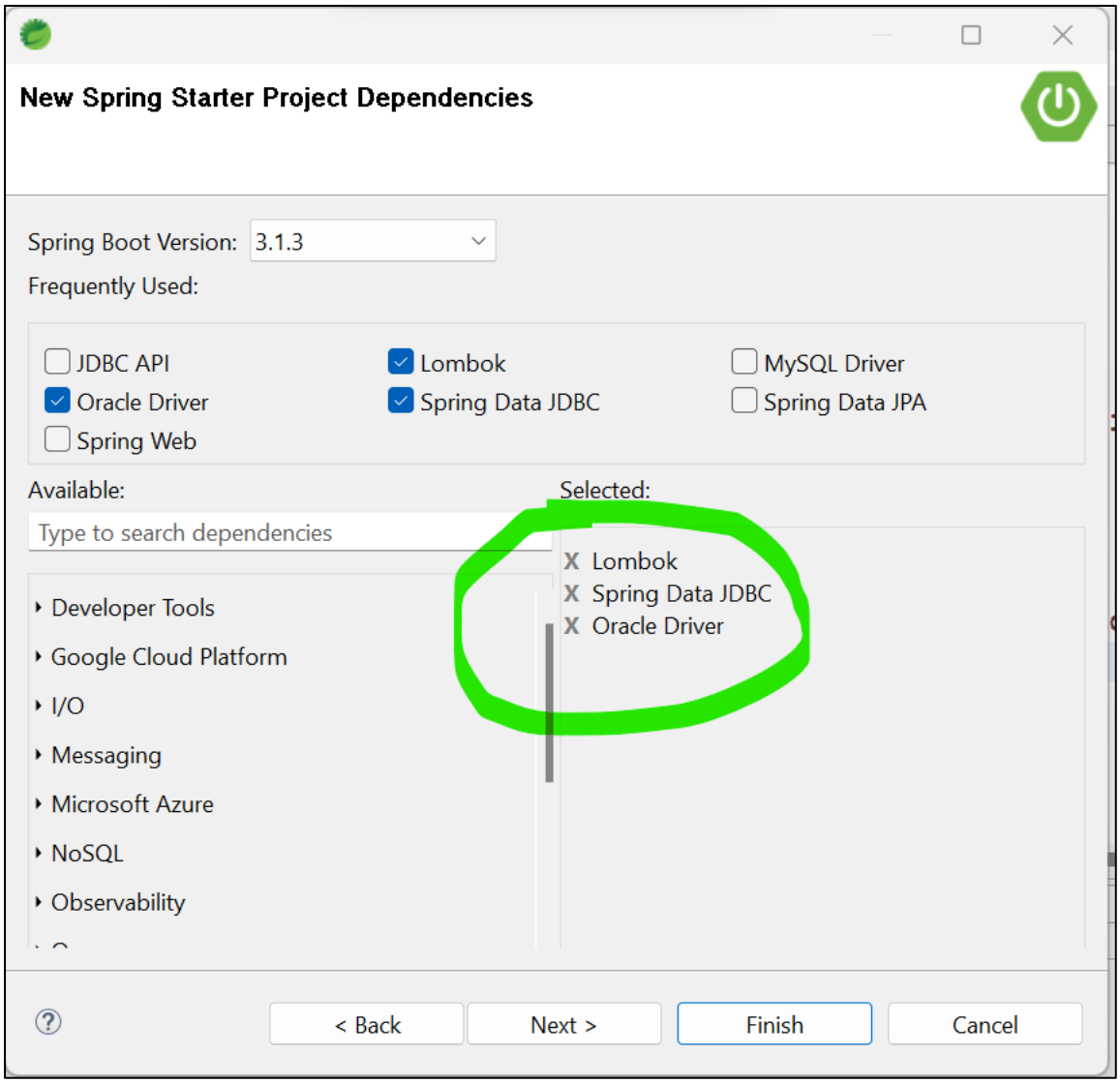
Note: I am **using Oracle Database** for all examples. If you are using another database other than Oracle, We just need to replace **URL**, **User Name** and **Password** of other database.

➢ Open STS and Create Project with Spring Boot Data JDBC dependency.



➢ Select **Spring Data JDBC** and **Oracle Driver** Dependencies -> click on Finish

Oracle Driver: When developing an application that needs to interact with Oracle Database, you typically need to include the appropriate Oracle driver as a dependency in your project. The driver provides the necessary classes and methods for establishing connections, executing SQL queries, and processing database results. How we added **ojdbc.jar** file in JDBC programming projects. Same **ojdbc.jar** file here also added by Spring Boot into project level.



Now We have to start Programming.

Requirement: **Add Product Details.**

Create Table in Database:

CREATE TABLE PRODUCT(ID NUMBER(10), NAME VARCHAR2(50), PRICE NUMBER(10));

- We have to add database details inside **application.properties** with help of pre-defined properties provided by Spring Boot i.e. **DataSource Configuration**. Here, DataSource configuration is controlled by external configuration properties in **spring.datasource.***

Note: Spring Boot reduce the JDBC driver class for most databases from the URL. If you need to specify a specific class, you can use the **spring.datasource.driver-class-name** property.

```
#Oracle Database Details

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
```

- Create a Component class for doing Database operations. Inside class, we have to autowire **JdbcTemplate** to utilize pre-defined functionalities.

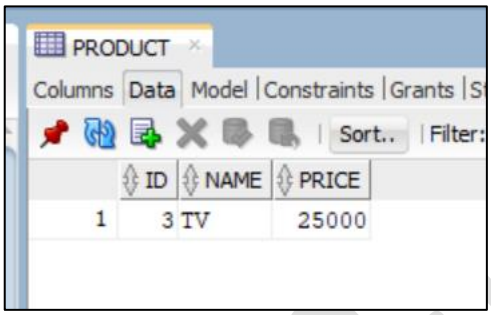
```
package com.dilip;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class DataBaseOperations {

	@Autowired
	JdbcTemplate jdbcTemplate;

	public void addProduct() {
		jdbcTemplate.update("insert into product values(3,'TV', 25000)");
	}
}
```

- Now Call above method.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringBootJdbcDemoApplication {
```

```
        public static void main(String[] args) {
        ApplicationContext context =
                        SpringApplication.run(SpringBootJdbcDemoApplication.class, args);

        DataBaseOperations dbOperations = context.getBean(DataBaseOperations.class);
        dbOperations.addProduct();
        }
}
```

Verify in Database: Data is inserted or not.



This is how Spring boot JDBC module simplified Database operations will very less amount of code.

Let's perform other database operations.

Requirement: Load all Product Details from Database as List of Product Objects.

- Here we have to create a POJO class of Product with properties.

```
package com.dilip;

import lombok.Data;

@Data
public class Product {
        int id;
        String name;
        int price;
}
```

- Create another method inside Database Operations class.

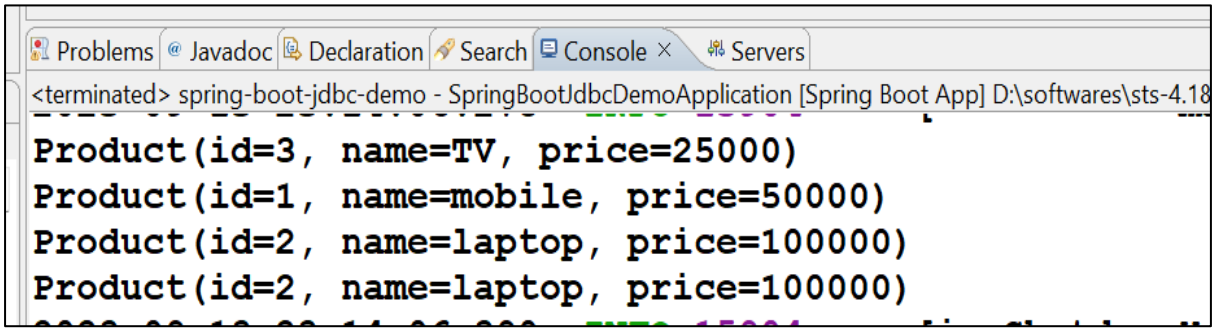
```
package com.dilip;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;
```

```
@Component
public class DataBaseOperations {

	@Autowired
	JdbcTemplate jdbcTemplate;

	public void addProduct() {
		jdbcTemplate.update("insert into product values(2,'laptop', 100000)");
	}

   public void loadAllProducts() {

	String query = "select * from product";
	List<Product> allProducts =
       jdbcTemplate.query(query, new BeanPropertyRowMapper<Product>(Product.class));

	for (Product p : allProducts) {
		System.out.println(p);
	}
    }
}
```

➢ Execute above logic.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringBootJdbcDemoApplication {

	public static void main(String[] args) {

	ApplicationContext context =
				SpringApplication.run(SpringBootJdbcDemoApplication.class, args);

	  DataBaseOperations dbOperations = context.getBean(DataBaseOperations.class);
	  dbOperations.addProduct();
	  dbOperations.loadAllProducts();

	}
}
```

- Now verify in Console Output. All Records are loaded and converted as Lis of Product Objects.



Adding few more Requirements.

```
package com.dilip;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class DataBaseOperations {

	@Autowired
	JdbcTemplate jdbcTemplate;

	public void addProduct() {
		jdbcTemplate.update("insert into product values(2,'laptop', 100000)");
	}

	public void loadAllProducts() {
		String query = "select * from product";
	List<Product> allProducts =
  jdbcTemplate.query(query, new BeanPropertyRowMapper<Product>(Product.class));

		for (Product p : allProducts) {
			System.out.println(p);
		}
	}

	// Select all product Ids as List Object of Ids
	public void getAllProductIds() {
		List<Integer> allIds =
			jdbcTemplate.queryForList("select id from product", Integer.class);
		System.out.println(allIds);
```

```
	}

	//delete product by id
	public void deleteProduct(int id) {
		String query = "delete from product where id=" + id;
		System.out.println(query);
		int count = jdbcTemplate.update(query);
		System.out.println("Nof of Records Deleted :" + count);
	}
}
```

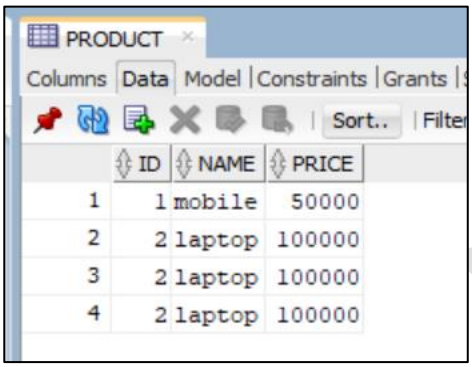
- Test above methods and logic now and verify in database.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringBootJdbcDemoApplication {

	public static void main(String[] args) {
		ApplicationContext context =
			SpringApplication.run(SpringBootJdbcDemoApplication.class, args);
		DataBaseOperations dbOperations =
				context.getBean(DataBaseOperations.class);
		dbOperations.addProduct();
		dbOperations.loadAllProducts();
		dbOperations.getAllProductIds();
		dbOperations.deleteProduct(3);
	}
}
```

- Verify Console Output and inside Database record delated or not.

```
Problems  Javadoc  Declaration  Search  Console  Servers
<terminated> spring-boot-jdbc-demo - SpringBootJdbcDemoApplication [Spring Boot App] D:\softwares\sts-4.18.0.R
Product(id=3, name=TV, price=25000)
Product(id=1, name=mobile, price=50000)
Product(id=2, name=laptop, price=100000)
Product(id=2, name=laptop, price=100000)
Product(id=2, name=laptop, price=100000)
[3, 1, 2, 2, 2]
delete from product where id=3
Nof of Records Deleted :1
2023-09-13 23:33:27.051  INFO 2628 --- [ionShutdownHook]
```

Inside Database: Product Id with 3 is deleted.



Positional Parameters in Query :

Requirement : Update Product Details with Product Id.

Here, I am using positional parameters "**?**" as part of database Query to pass real values to queries.

In this scenario, Spring JDBC provided an overloaded method update(String sql, @Nullable Object... args). In this method, we have to pass positional parameter values in an order.

```
package com.dilip;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class DataBaseOperations {

	@Autowired
	JdbcTemplate jdbcTemplate;

	public void updateProductData(int price, int id) {
		String query = "update product set price=? where id=?";
		jdbcTemplate.update(query, price, id);
	}
}
```

Testing:

```
package com.dilip;

import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringBootJdbcDemoApplication {
public static void main(String[] args) {
        ApplicationContext context
                = SpringApplication.run(SpringBootJdbcDemoApplication.class, args);
        DataBaseOperations dbOperations = context.getBean(DataBaseOperations.class);
        dbOperations.updateProductData(60000, 3);
        }
}
```

- Verify inside database Table, Table Data is updated or not.

Now Understand If we want to do same in Spring:

Steps for Spring JDBC Project:

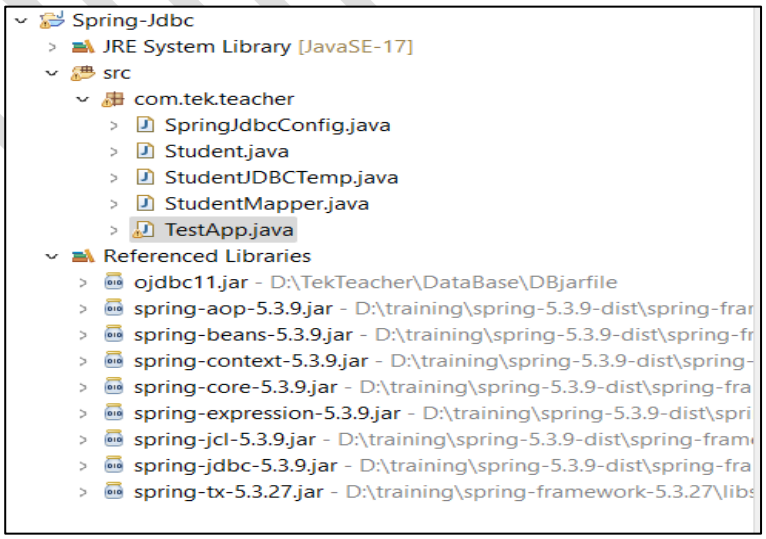
Create a Project with Spring JDBC module Functionalities: Perform below Operations on Student Table.

- **Insert Data**
- **Update Data**
- **Select Data**
- **Delete Data**

Table Creation : **create table student(sid number(10), name varchar2(50), age number(3));**

Please add Specific jar files which are required for JDBC module, as followed.

Project Structure : for jar files reference



- **Please Create Configuration class for Configuring JdbcTemplate Bean Object with DataSource Properties. So Let's start with some simple configuration of the data source.**

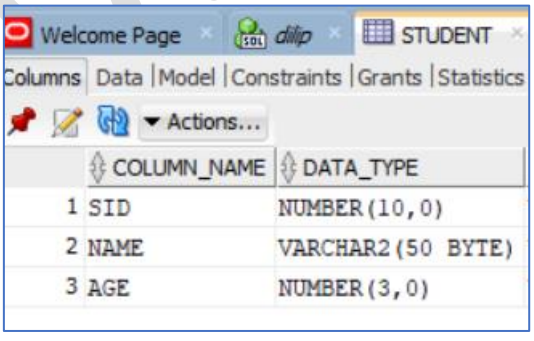
- **We are using Oracle database:**
- **The DriverManagerDataSource is used to contain the information about the database such as driver class name, connection URL, username and password.**

SpringJdbcConfig.java

```
package com.tek.teacher;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
@ComponentScan("com.tek.teacher")
public class SpringJdbcConfig {

	@Bean
	public JdbcTemplate getJdbcTemplate() {
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");
		dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
		dataSource.setUsername("c##dilip");
		dataSource.setPassword("dilip");
		return  new JdbcTemplate(dataSource);
	}
}
```

- **Create a POJO class of Student which should be aligned to database table columns and data types.**

  Table : Student



➢ **Student.java POJO class:**

```
package com.tek.teacher;

public class Student {
```

```
	// Class data members
	private Integer age;
	private String name;
	private Integer sid;

	// Setters and Getters
	public void setAge(Integer age) {
		this.age = age;
	}
	public Integer getAge() {
		return age;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getName() {
		return name;
	}
	public Integer getSid() {
		return sid;
	}
	public void setSid(Integer sid) {
		this.sid = sid;
	}
	@Override
	public String toString() {
		return "Student [age=" + age + ", name=" + name + ", sid=" + sid + "]";
	}
}
```

Mapping Query Results to Java Object:

Another very useful feature is the ability to map query results to Java objects by implementing the **RowMapper** interface i.e. when we execute select query's, we will get ResultSet Object with many records of database table. So if we want to convert every row as a single Object then this row mapper will be used. For every row returned by the query, Spring uses the row mapper to populate the java bean object.

A **RowMapper** is an interface in Spring JDBC that is used to map a row from a **ResultSet** to an object. The **RowMapper** interface has a single method, **mapRow(),** which takes a **ResultSet** and a row number as input and returns an object.

The **mapRow()** method is called for each row in the **ResultSet**. The RowMapper implementation is responsible for extracting the data from the **ResultSet** and creating the corresponding object. The object can be any type of object, but it is typically a POJO (Plain Old Java Object).

StudentMapper.java

```
package com.tek.teacher;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student>{

	@Override
	public Student mapRow(ResultSet rs, int arg1) throws SQLException {

			Student student = new Student();
			student.setSid(rs.getInt("sid"));
			student.setName(rs.getString("name"));
			student.setAge(rs.getInt("age"));

		return student;
	}
}
```

- **Write a class to perform all DB operation i.e. execution of Database Queries based on our requirement. As part of this class we will use Spring JdbcTemplate Object, and methods to execute database queries.**

StudentJDBCTemp.java

```
package com.tek.teacher;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class StudentJDBCTemp  {

@Autowired
private JdbcTemplate jdbcTemplateObject;

public List<Student> listStudents() {
	// Custom SQL query
	String query = "select * from student";
	List<Student> students = jdbcTemplateObject.query(query, new StudentMapper());
	return students;
```

```
}

        //@Override
        public int addStudent(Student student) {

         String query =  "insert into student
               values("+student.getSid()+",'"+student.getName()+"',"+student.getAge()+")";

                  System.out.println(query);
                  return jdbcTemplateObject.update(query);
        }
}
```

NOTE: Instead of implementing mapper from **RowMapper** Interface, we can use class **BeanPropertyRowMapper** to convert Result Set as List of Objects. Same being used in previous Spring Boot example.

- **Write class for testing all our functionalities.**

TestApp.java

```
package com.tek.teacher;

import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class TestApp {

        public static void main(String[] args) {
                 AnnotationConfigApplicationContext context =
                                                  new AnnotationConfigApplicationContext();
                context.scan("com.tek.*");
                context.refresh();

                StudentJDBCTemp template = context.getBean(StudentJDBCTemp.class);

                //Insertion of Data
                Student s = new Student();
                s.setAge(30);
                s.setName("tek");
                s.setSid(2);

                int count = template.addStudent(s);
                System.out.println(count);

                // Load all Students
                List<Student> students = template.listStudents();
```

```
                students.stream().forEach(System.out::println);
                //students.stream().map(st -> st.getSid()).forEach(System.out::println);
        }
}
```

So, Finally main difference between Spring and Spring Boot JDBC module is we have to write Configuration class for getting **JdbcTemplate** Object. This is automated in Spring Boot JDBC module. Except this, rest of all logic is as usual common in both.

Spring Data JPA

JPA and Spring Data JPA:

The Java Persistence API is a standard technology that lets you “map” objects to relational databases. **Spring Data JPA**, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies. Too much boilerplate code has to be written to execute simple queries. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

- Spring Data JPA is not a JPA provider. It is a library/framework that adds an extra layer of abstraction on top of our JPA provider (like Hibernate).
- Spring Data JPA uses Hibernate as a default JPA provider.

The **spring-boot-starter-data-jpa** POM provides a quick way to get started. It provides the following key dependencies.

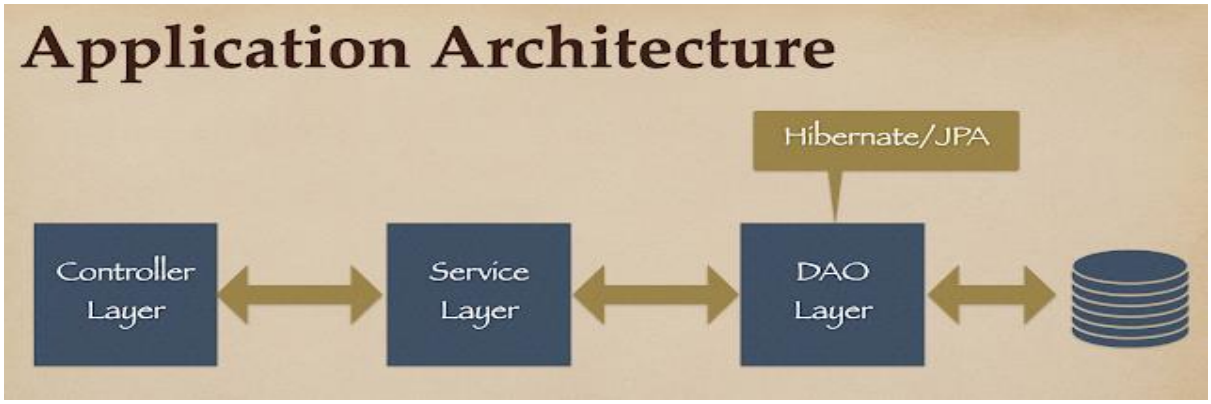
  - Hibernate: One of the most popular JPA implementations.
  - Spring Data JPA: Helps you to implement JPA-based repositories.
  - Spring ORM: Core ORM support from the Spring Framework.

Java/Jakarta Persistence API (JPA) :

The **Java/Jakarta Persistence API (JPA)** is a specification of Java. It is used to persist data between Java object and relational database. **JPA acts as a bridge between object-oriented domain models and relational database systems.** As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence. JPA represents how to define POJO (Plain Old Java Object) as an entity and manage it with relations using some meta configurations. They are defined either by annotations or by XML files.

Features:

- **Idiomatic persistence** : It enables you to write the persistence classes using object oriented classes.
- **High Performance** : It has many fetching techniques and hopeful locking techniques.
- **Reliable** : It is highly stable and eminent. Used by many industrial programmers.



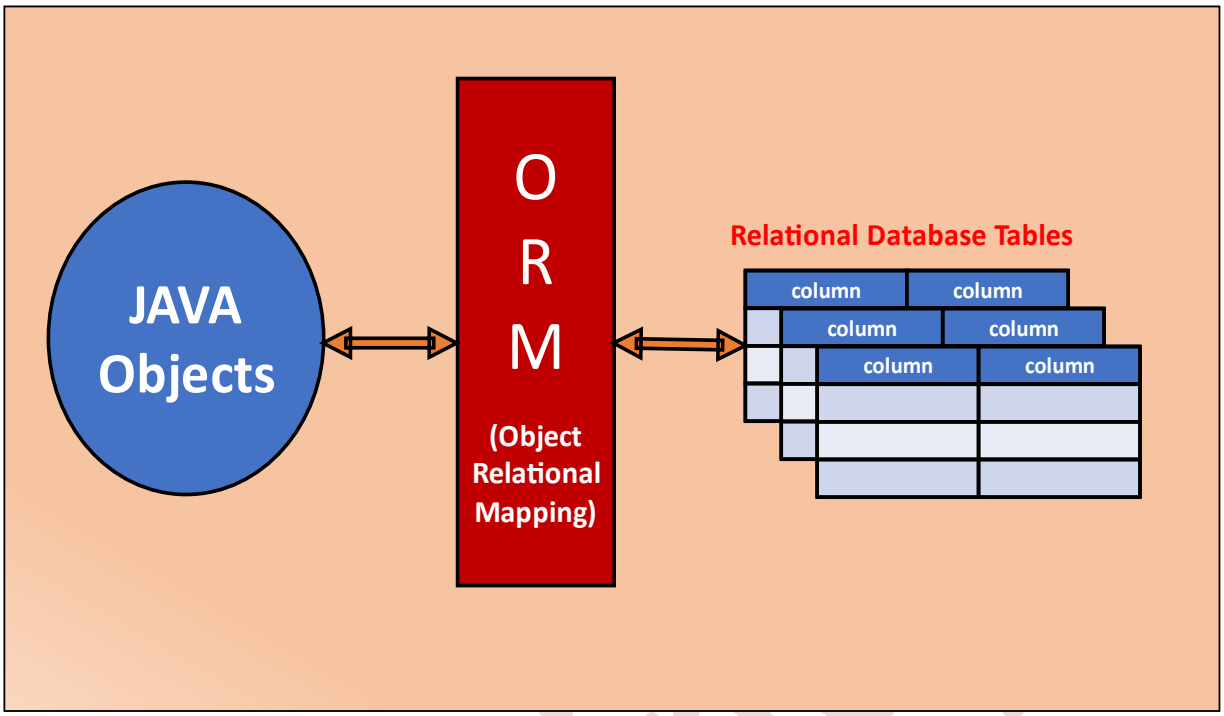
ORM(Object-Relational Mapping))

ORM(Object-Relational Mapping) is the method of querying and manipulating data from a database using an object-oriented paradigm/programming language. By using this method, we are able to interact with a relational database without having to use SQL. Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column. It is capable to handle various database operations easily such as inserting, updating, deleting etc.
We are going to implement Entity classes to map with Database Tables.

What is Entity Class?

Entities in JPA are nothing but POJOs representing data that can be persisted in the database. a class of type Entity indicates a class that, at an abstract level, is correlated with a table in the database. An entity represents a table stored in a database. Every instance of an entity represents a row in the table.

We will define POJOs with JPA annotations aligned to DB tables. We will see all annotations with an example.



Annotations Used In Entity Class:

- **@Table**, **@Id**, **@Column** Annotations are used in **@Entity** class to represent database table details, **name** is an attribute.
- Inside **@Table**, **name** value should be **Database table name**.
- Inside **@Column**, **name** value should be **table column name**.

@Entity Annotation: In Java Persistence API (JPA), the **@Entity** annotation is used to declare a class as an entity class. An entity class represents an object that can be stored in a database table. In JPA, entity classes are used to map Java objects to database tables, allowing you to perform CRUD (Create, Read, Update, Delete) operations on those objects in a relational database.

Here's how we use the **@Entity** annotation in JPA.

```
@Entity
public class Product {
    //Properties
}
```

Entity classes in JPA represent the structure of your database tables and serve as the foundation for database operations using JPA. You can create, retrieve, update, and delete instances of these entity classes, and the changes will be reflected in the corresponding database tables, making it a powerful tool for working with relational databases in Java applications.

@Table Annotation: In Java Persistence API (JPA), the **@Table** annotation is used to specify the details of the database table that corresponds to an **@Entity** class. When we create an

entity class, we want to map it to a specific table in the database. The **@Table** annotation allows you to define various attributes related to the database table.

Here's how we use the **@Table** annotation in JPA.

```
@Entity
@Table(name = "products")
public class Product {
    //properties
}
```

@Table(name="products") , Specifies that this entity is associated with a table named "products" in the database. You can provide the `**name**` attribute to specify the name of the table. If you don't provide the `**name**` attribute, JPA will use the default table name, which is often derived from the name of the entity class (in this case, "**Product**").

We can also use other attributes of the `**@Table**` annotation to specify additional information about the table, such as the schema, unique constraints, indexes, and more, depending on your database and application requirements.

@Id Annotation: In Java Persistence API (JPA), `**@Id**` is an annotation used to declare a field or property as the primary key of an entity class. JPA is a Java specification for object-relational mapping (ORM), which allows you to map Java objects to database tables. The **@Id** annotation is an essential part of defining the structure of your entity classes when working with JPA. Here's how you use **@Id** in JPA.

Field-Level Annotation: We can place **@Id** annotation directly on a field in our entity class.
Property-Level Annotation: We can also place the **@Id** annotation **on a getter method** if you prefer property access instead of field access.

```
@Entity
@Table(name = "products")
public class Product {
   private Long id;

   @Id
   public Long getId() {
      return id;
   }
}
```

The **@Id** annotation marks the specified field or property as the primary key for the associated entity. This means that the value of this field uniquely identifies each row in the corresponding database table. Additionally, you may need to specify how the primary key is generated, such as using database-generated values or providing your own. JPA provides various strategies for

generating primary keys, and you can use annotations like **@GeneratedValue** in conjunction with **@Id** to define the strategy for generating primary key values.

@Id, The field or property to which the Id annotation is applied should be one of the following types: any Java primitive type; any primitive wrapper type; String; java.util.Date; java.sql.Date; java.math.BigDecimal; java.math.BigInteger. The mapped column for the primary key of the entity is assumed to be the primary key of the primary table.

@Column Annotation: In Java Persistence API (JPA), the **@Column** annotation is used to specify the details of a database column that corresponds to a field or property of an entity class. When you create an entity class, you often want to map its fields or properties to specific columns in the associated database table. The **@Column** annotation allows you to define various attributes related to the database column.

Here's how you use the **@Column** annotation in JPA:

```
@Entity
@Table(name = "products")
public class Product {

        @Id
        @Column
        private Long id;

        @Column(name = "product_name", length = 100, nullable = false)
        private String name;

        @Column(name = "product_price", precision = 10, scale = 2)
        private BigDecimal price;

        // Constructors, getters, setters, and other methods...
}
```

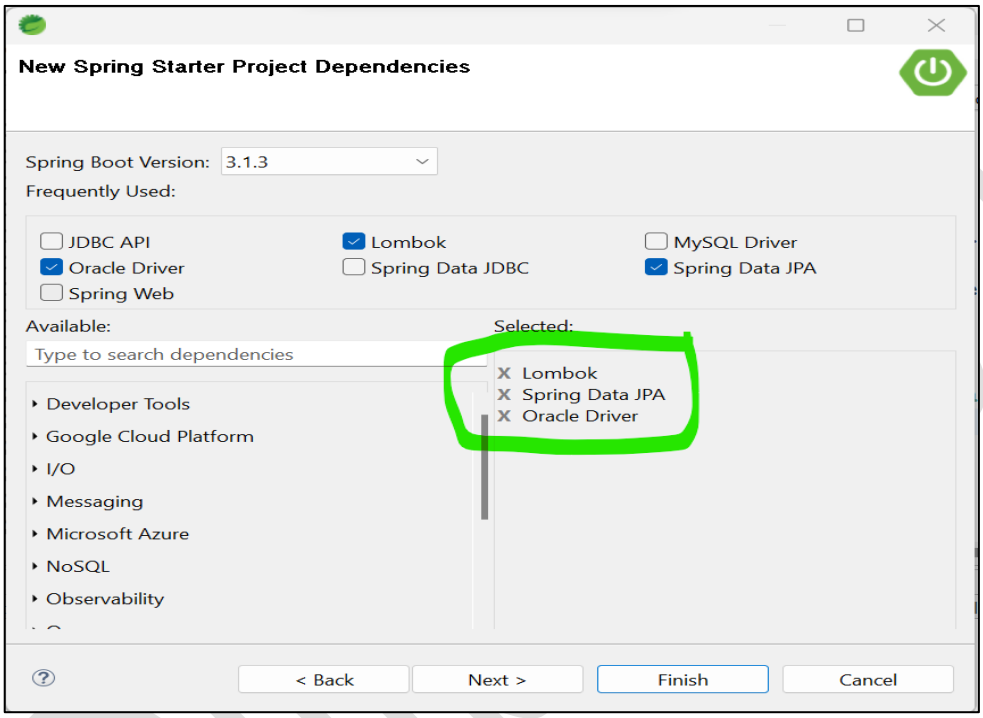
@Column(name = "product_name", length = 100, nullable = false): The **@Column** annotation is used to specify the mapping of entity class fields to table columns. In this example, it indicates that the "**name"** field should be mapped to a column named "**product_name**" in the database table. Additional attributes like **length** and **nullable** specify constraints on the column:

- **name**: Specifies the name of the database column. If you don't provide the `name` attribute, JPA will use the field or property name as the default column name.
- **length**: Specifies the maximum length of the column's value.
- **nullable**: Indicates whether the column can contain null values. Setting it to **false** means the column is mandatory (cannot be null).

@Column(name = "product_price", precision = 10, scale = 2): Similarly, this annotation specifies the mapping for the `**price**` field, including the column name and precision/scale for numeric values.

The **@Column** annotation provides a way to customize the mapping between your entity class fields or properties and database columns. You can use it to specify various attributes like column name, data type, length, and more, depending on your database and application requirements.

- **Create Spring Boot application with JPA dependency and respective Database Driver.**



- Here we are working with Database, So please Add Database Details like URL, username and password inside **application.properties** file.

application.properties:

```
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
```

- **Create Entity Class:**

First we should have Database table details with us, Based on Table Details we are creating a POJO class i.e. configuring Table and column details along with POJO class Properties. I have a table in my database as following. Then I will create POJO class by creating Properties aligned to DB table datatypes and column names with help of JPA annotations.

Table Name : flipkart_orders

Table : FLIPKART_ORDERS		JAVA Entity Class : FlipakartOrder	
ORDERID	NUMBER(10)	orderID	long
PRODUCTNAME	VARCHAR2(50)	productName	String
TOTALAMOUNT	NUMBER(10,2)	totalAmount	float

Entity Class: FlipakartOrder.java

```
package com.flipkart.dao;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;

@Entity
@Table(name = "FLIPKART_ORDERS")
public class FlipakartOrder {

	@Id
	@Column(name = "ORDERID")
	private long orderID;

	@Column(name = "PRODUCTNAME")
	private String productName;

	@Column(name = "TOTALAMOUNT")
	private float totalAmount;

	public long getOrderID() {
		return orderID;
	}
	public void setOrderID(long orderID) {
		this.orderID = orderID;
	}
	public String getProductName() {
		return productName;
	}
	public void setProductName(String productName) {
		this.productName = productName;
	}
	public float getTotalAmount() {
		return totalAmount;
	}
	public void setTotalAmount(float totalAmount) {
		this.totalAmount = totalAmount;
```

```
        }
}
```

Note: When Database **Table column name** and **Entity class property** name are equal, it's not mandatory to use **@Column** annotation i.e. It's an Optional in such case. If both are different then we should use **@Column** annotation along with value.

For Example : Assume, we written property and column as below in an Entity class.

```
@Column(name="pincode")
private int pincode;
```

In this case we can define only property name i.e. internally JPA considers **pincode** is aligned with **pincode** column in table

```
private int pincode;
```

Spring JPA Repositories:

Spring Data JPA repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. In Spring Data JPA, a repository is an abstraction that provides an interface to interact with a database using Java Persistence API (JPA). Spring Data JPA repositories offer a set of common methods for performing CRUD (Create, Read, Update, Delete) operations on database entities without requiring you to write boilerplate code. These repositories also allow you to define custom queries using method names, saving you from writing complex SQL queries manually.

Spring JPA Provided 2 Types of Repositories

- **JpaRepository**
- **CrudRepository**

Repositories work in Spring JPA by extending the **JpaRepository/CrudRepository** interface. These interfaces provides a number of default methods for performing CRUD operations, such as **save, findById, and delete etc..** we can also extend the JpaRepository interface to add your own custom methods.

When we create a repository, Spring Data JPA will automatically create an implementation for it. This implementation will use the JPA provider that you have configured in your Spring application.

Create Spring JPA Repository:

```
package com.flipkart.dao;
```

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface FlipakartOrderRepository extends JpaRepository< FlipakartOrder, Long> {

}
```

This repository is for a **FlipakartOrder** entity. The **Long** parameter in the extends **JpaRepository** statement specifies the type of the entity's identifier representing primary key column in Table.

The **FlipakartOrderRepository** interface provides a number of default methods for performing **CRUD operations** on **FlipakartOrder** entities. For example, the **save()** method can be used to save a new **FlipakartOrder** entity, and the **findById()** method can be used to find a **FlipakartOrder** entity by its identifier.

We can also extend the **FlipakartOrderRepository** interface to add your own custom methods. For example, you could add a method to find all **FlipakartOrder** entities that based on a Product name.

Benefits of using Spring JPA Repositories:

- **Reduced boilerplate code:** Repositories provide a number of default methods for performing CRUD operations, so you don't have to write as much code and SQL Queries.
- **Enhanced flexibility:** Repositories allow you to add your own custom methods, so you can tailor your data access code to your specific requirements.

Note: If We are using JPA in your Spring application, highly recommended using Spring JPA Repositories. They will make your code simpler, more consistent, and more flexible.

Here's how repositories work in Spring Data JPA:

- **Define an Entity Class**: An entity class is a Java class that represents a database table. It is annotated with **@Entity** and contains fields that map to table columns.
- **Create a Repository Interface**: Create an interface that extends the **JpaRepository** interface provided by Spring Data JPA. This interface will be used to perform database operations on the associated entity. You can also extend other repository interfaces such as **PagingAndSortingRepository**, **CrudRepository**, etc., based on your needs.
- **Method Naming Conventions**: Spring Data JPA automatically generates queries based on the method names defined in the repository interface. For example, a method named findByFirstName will generate a query to retrieve records based on the first name.
- **Custom Queries**: You can define custom query methods by using specific keywords in the method name, such as find...By..., read...By..., query...By..., or get...By.... Additionally, you can use **@Query** annotations to write **JPQL** (**Java Persistence Query Language**) or native SQL queries.

- **Dependency Injection**: Inject the repository interface into your service or controller classes using Spring's dependency injection.
- **Use Repository Methods**: You can now use the methods defined in the repository interface to perform database operations.

Spring Data JPA handles the underlying database interactions, such as generating SQL queries, executing them, and mapping the results back to Java objects.

Now create a Component class For Performing Database operations as per Requirements
Requirement: Add One Order Details to our database table "FLIPKART_ORDERS"

The **save()** method of Spring JPA Repository, can be used to both insert a new entity or update an existing one if an ID is provided.

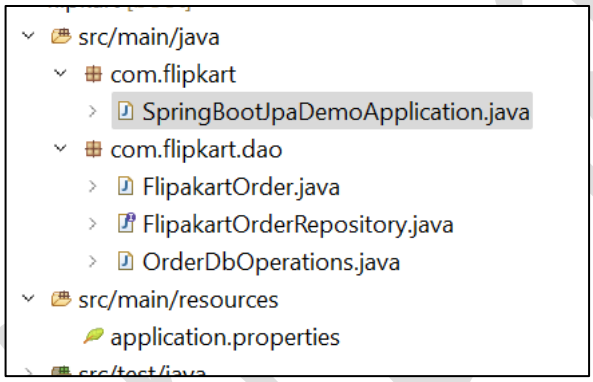
```
package com.flipkart.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

// To Execute/Perform DB operations
@Component
public class OrderDbOperations {

	@Autowired
	FlipakartOrderRepository flipakartOrderRepository;

	public void addOrderDetaisl(FlipakartOrder order) {
		flipakartOrderRepository.save(order);
	}
}
```

- Now Inside Spring Boot Application Main method class, get the **OrderDbOperations** instance and call methods.

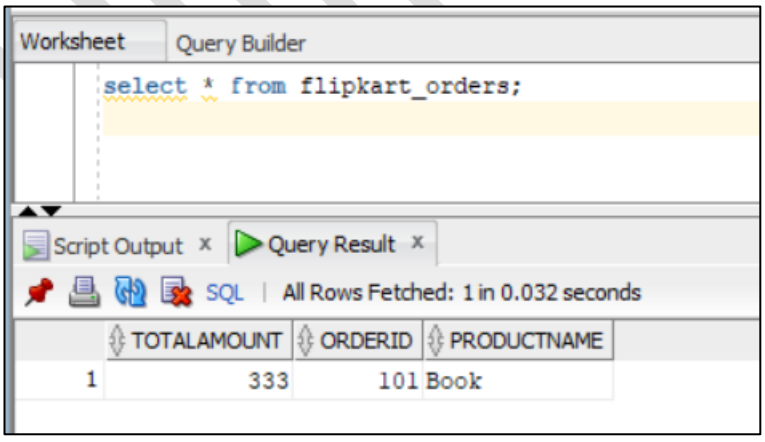
```
package com.flipkart;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.flipkart.dao.FlipakartOrder;
import com.flipkart.dao.OrderDbOperations;

@SpringBootApplication
public class SpringBootJpaDemoApplication {
	public static void main(String[] args) {
	ApplicationContext context
			= SpringApplication.run(SpringBootJpaDemoApplication.class, args);
```

```
                // Created Entity Object
                FlipakartOrder order = new FlipakartOrder();
                order.setOrderID(9988);
                order.setProductName("Book");
                order.setTotalAmount(333.00f);

                // Pass Entity Object to Repository Method
            OrderDbOperations dbOperation
                                        = context.getBean(OrderDbOperations.class);
                dbOperation.addOrderDetaisl(order);
        }
}
```

Project Structure:



Testing: Now Execute The Programme. If No errors/exceptions verify Inside Database table, Data inserted or not.



Based on Repository method **save()** of flipakartOrderRepository.save(order), JPA internally generates implementation code as well as SQL queries and then those Queries will be executed on database.

NOTE: In our example, we are nowhere written any SQL query to do Database operation.

Similarly, we have many predefined methods of Spring repository to do CRUD operations.

We learned how to configure the persistence layer of a Spring application that uses Spring Data JPA and Hibernate. Let's create few more examples to do CRUD operations on Database tables.

Internally, Spring JPA/Hibernate Generates SQL query based on repository methods which we are using in our logic i.e. Spring JPA Internally generates implementation for our Repository Interface like **FlipakartOrderRepository** and injects instance of that implementation inside Repository.

spring.jpa.show-sql: The **spring.jpa.show-sql** property is a spring JPA configuration property that controls whether or not Hibernate will log the SQL statements that it generates. The possible values for this property are:

true: Hibernate will log all SQL statements to the console.
false: Hibernate will not log any SQL statements.

The default value for this property is **false**. This means that Hibernate will not log any SQL statements by default. If you want to see the SQL statements that Hibernate is generating, you will need to set this property to **true**.

Logging SQL statements can be useful for debugging purposes. If you are having problems with your application, you can enable logging and see what SQL statements Hibernate is generating. This can help you to identify the source of the problem.

Add below Property In : application.properties

```
spring.jpa.show-sql=true
```

Requirement: Add List of Orders at time to the table.

- **saveAll():** This method will be used for persisting all objects into table.

Add Logic in : OrderDbOperations.java

```
package com.flipkart.dao;

import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

// To Execute/Perform DB operations
@Component
public class OrderDbOperations {
```

```
	@Autowired
	FlipakartOrderRepository flipakartOrderRepository;

	//Adding List Of Orders ata time
	public void addListOfOrders() {

		List<FlipakartOrder> orders = new ArrayList<>();

		FlipakartOrder order1 = new FlipakartOrder();
		order1.setOrderID(123);
		order1.setProductName("Keyboard");
		order1.setTotalAmount(500.00f);

		FlipakartOrder order2 = new FlipakartOrder();
		order2.setOrderID(124);
		order2.setProductName("Mouse");
		order2.setTotalAmount(300.00f);

		FlipakartOrder order3 = new FlipakartOrder();
		order3.setOrderID(125);
		order3.setProductName("Monitor");
		order3.setTotalAmount(10000.00f);

		orders.add(order1);
		orders.add(order2);
		orders.add(order3);

		flipakartOrderRepository.saveAll(orders);
	}
}
```

- **Execute above logic.**

```
package com.flipkart;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.flipkart.dao.OrderDbOperations;

@SpringBootApplication
public class SpringBootJpaDemoApplication {
	public static void main(String[] args) {
	ApplicationContext context =
				SpringApplication.run(SpringBootJpaDemoApplication.class, args);

	OrderDbOperations dbOperation =  context.getBean(OrderDbOperations.class);
```

```
            dbOperation.addListOfOrders();
    }
}
```

➢ Now Three records will be inserted in database.

FLIPKART_ORDERS

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | De

	TOTALAM...	ORDERID	PRODUCTNAME
1	333	9988	Book
2	500	123	Keyboard
3	300	124	Mouse
4	10000	125	Monitor

Now in application console logs, we can see SQL Queries are executed internally by JPA.

```
<terminated> flipkart - SpringBootJpaDemoApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bi
c.flipkart.SpringBootJpaDemoApplication   : Started SpringBootJpaDemoApplication in 3.124
seconds (JVM running for 3.983)
Hibernate: select flipakarto0_.orderid as orderid1_0_0_, flipakarto0_.productname as
productname2_0_0_, flipakarto0_.totalamount as totalamount3_0_0_ from flipkart_orders
flipakarto0_ where flipakarto0_.orderid=?
Hibernate: select flipakarto0_.orderid as orderid1_0_0_, flipakarto0_.productname as
productname2_0_0_, flipakarto0_.totalamount as totalamount3_0_0_ from flipkart_orders
flipakarto0_ where flipakarto0_.orderid=?
Hibernate: select flipakarto0_.orderid as orderid1_0_0_, flipakarto0_.productname as
productname2_0_0_, flipakarto0_.totalamount as totalamount3_0_0_ from flipkart_orders
flipakarto0_ where flipakarto0_.orderid=?
Hibernate: insert into flipkart_orders (productname, totalamount, orderid) values (?, ?, ?)
Hibernate: insert into flipkart_orders (productname, totalamount, orderid) values (?, ?, ?)
Hibernate: insert into flipkart_orders (productname, totalamount, orderid) values (?, ?, ?)
2023-09-16 22:58:04.579  INFO 3200 --- [ionShutdownHook]
j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persistence
```

Requirement: Load all Order Details from database as a List of Orders Object.

findAll(): This method will load all records of a table and converts all records as Entity Objects and all Objects added to ArrayList i.e. finally returns List object of FlipakartOrder entity objects.

```
public void loadAllOrders() {
      List<FlipakartOrder>  allOrders = flipakartOrderRepository.findAll();
            for(FlipakartOrder order : allOrders) {
                        System.out.println(order.getOrderID());
                        System.out.println(order.getProductName());
                        System.out.println(order.getTotalAmount());
            }
}
```

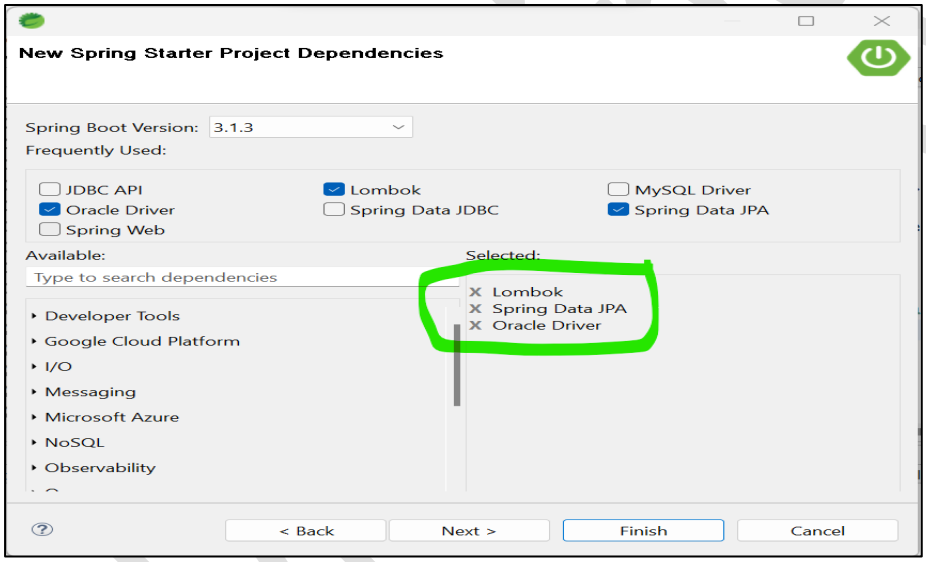
Please execute above logic by calling **loadAllOrders().**

➢ **New Requirement : Let's Have** Patient Information as follows.

- Name
- Age
- Gender
- Contact Number
- Email Id

1. Add Single Patient Details
2. Add More Than One Patient Details
3. Update Patient Details
4. Select Single Patient Details
5. Select More Patient Details
6. Delete Patient Details

1. **Create Spring Boot Project with JPA and Oracle Driver**



2. Now Add Database Details in **application.properties**

```
#database details
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip

#to print SQL Queries
spring.jpa.show-sql=true

#DDL property
spring.jpa.hibernate.ddl-auto=update
```

spring.jpa.hibernate.ddl-auto: The **spring.jpa.hibernate.ddl-auto** property is used to configure the automatic schema/tables generation and management behaviour of Hibernate. This property allows you to control how Hibernate handles the database schema/tables based on your entity classes i.e. When we created Entity Classes

Here are the possible values for the **spring.jpa.hibernate.ddl-auto** property:

1. **none**: No action is performed. The schema will not be generated.
2. **validate:** The database schema will be validated using the entity mappings. This means that Hibernate will check to see if the database schema matches the entity mappings. If there are any differences, Hibernate will throw an exception.
3. **update:** The database schema will be updated by comparing the existing database schema with the entity mappings. This means that Hibernate will create or modify tables in the database as needed to match the entity mappings.
4. **create:** The database schema will be created. This means that Hibernate will create all of the tables needed for the entity mappings.
5. **create-drop:** The database schema will be created and then dropped when the **SessionFactory** is closed. This means that Hibernate will create all of the tables needed for the entity mappings, and then drop them when the **SessionFactory** is closed.

3. Create Entity Class

NOTE : Configured **spring.jpa.hibernate.ddl-auto** value as **update**. So Table Creation will be done by JPA internally if tables are not available.

```
package com.dilip.dao;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import lombok.Data;

@Data
@Entity
@Table
public class Patient {
	@Id
	@Column
	private String emailId;

	@Column
	private String name;

	@Column
	private int age;
```

```
	@Column
	private String gender;

	@Column
	private String contact;
}
```

4. **Create A JPA Repository Now :** PatientRepository.java

```
package com.dilip.dao;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Component;

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {

}
```

5. **Create a class for DB operations :** PatientOperations.java

```
package com.dilip.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;

}
```

Spring JPA Repositories Provided many predefined abstract methods for all DB CURD operations. We should recognize those as per our DB operation.

Requirement : **Add Single Patient Details**

Here, we are adding Patient details means at Database level this is insert Query Operation.

save() : Used for insertion of Details. We should pass Entity Object.

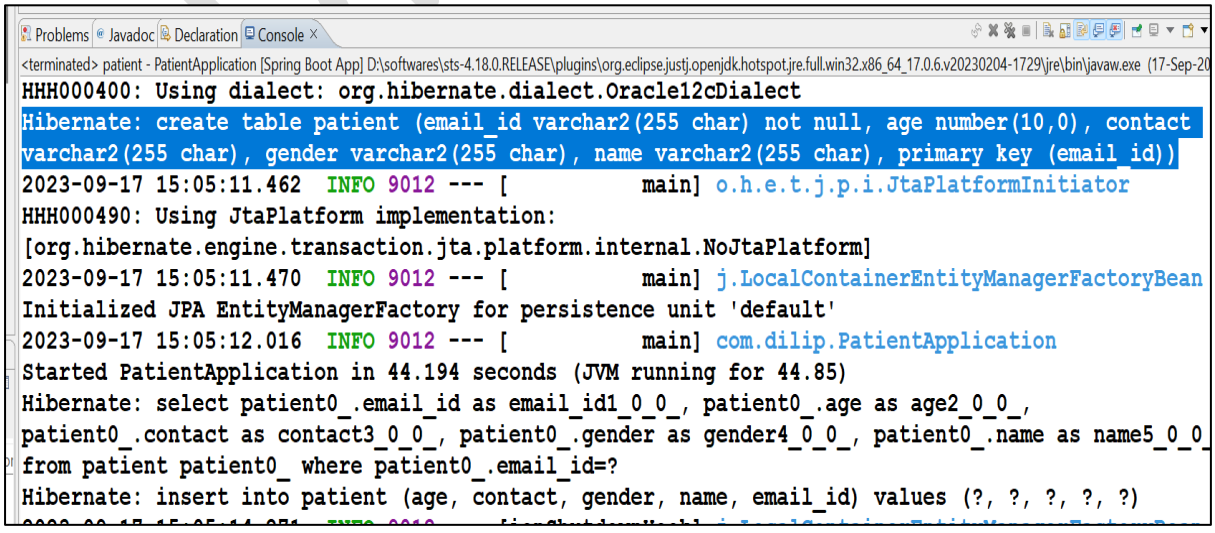
Add Below Method in PatientOperations.java:

```
public void addPatient(Patient p) {
	repository.save(p);
}
```

Now Test it : From Main Class : **PatientApplication.java**

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {

	public static void main(String[] args) {
		ApplicationContext context
				= SpringApplication.run(PatientApplication.class, args);
		PatientOperations ops = context.getBean(PatientOperations.class);

			// Add Single Patient
			Patient p = new Patient();
			p.setEmailId("one@gmail.com");
			p.setName("One Singh");
			p.setContact("+918826111377");
			p.setAge(30);
			p.setGender("MALE");

		ops.addPatient(p);
	}
}
```

Now Execute It. Table also created by Spring Boot JPA module and One Record is inserted.



Requirement : Update Patient Details

In Spring Data JPA, the **save()** method is commonly used for both **insert** and **update** operations. When you call the **save()** method on a repository, Spring Data JPA checks whether the entity you're trying to save already exists in the database. If it does, it updates the existing entity; otherwise, it inserts a new entity.

So that is the reason we are seeing a select query execution before inserting data in previous example. After select query execution with primary key column JPA checks row count and if it is 1, then JPA will convert entity as insert operation. If count is 0 , then Spring JPA will convert entity as update operation specific to Primary key.

Using the **save()** method for updates is a common and convenient approach, especially when we want to leverage Spring Data JPA's automatic change tracking and transaction management.

Requirement: Please update name as Dilip Singh for email id: one@gmail.com

```
select * from Patient;
```

Query Result

SQL | All Rows Fetched: 6 in 0.002 seconds

	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	30	+918826111377	MALE	One Singh
2	two@gmail.com	30	+828388	MALE	Two Singh
3	three@gmail.com	36	+44343423	FEMALE	Xyz Singh
4	naresh@gmail.com	28	+91372372	MALE	Naresh
5	dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
6	suresh@gmail.com	28	+91372372	MALE	Suresh

Add Below Method in PatientOperations.java:

```
public void updatePateinData(Patient p) {
		repository.save(p);
}
```

Now Test it from Main class: In below if we observe, first select query executed by JPA as per our entity Object, JPA found data so JPA decided for update Query execution. We have to send updated data as part of Entity Object.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {

	public static void main(String[] args) {
		ApplicationContext context
				= SpringApplication.run(PatientApplication.class, args);
		PatientOperations ops = context.getBean(PatientOperations.class);
		// Update Existing Patient
		Patient p = new Patient();
		p.setEmailId("one@gmail.com");
		p.setName("Dilip Singh");
		p.setContact("+918826111377");
		p.setAge(30);
		p.setGender("MALE");
		ops.addPatient(p);
	}
}
```

Verify In DB :

	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	30	+918826111377	MALE	Dilip Singh

Requirement: Delete Patient Details : **Deleting Patient Details based on Email ID.**

Spring JPA provided a predefined method **deleteById()** for primary key columns delete operations.

deleteById(): The **deleteById()** method in Spring Data JPA is used to remove an entity from the database based on its primary key (ID). It's provided by the **JpaRepository** interface and allows you to delete a single entity by its unique identifier.

Here's how you can use the **deleteById()** method in a Spring Data JPA repository:

Add Below Method in PatientOperations.java:

```
package com.dilip.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;

	public void addPatient(Patient p) {
		repository.save(p);
	}
	public void deletePatient(String email) {
		repository.deleteById(email);
	}
}
```

Testing from Main Class:

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {

	public static void main(String[] args) {
		ApplicationContext context = SpringApplication.run(PatientApplication.class, args);
		PatientOperations ops = context.getBean(PatientOperations.class);
		// Delete Existing Patient by email Id
		ops.deletePatient("two@gmail.com");
	}
}
```

Before Execution/Deletion:

	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	30	+918826111377	MALE	Dilip Singh
2	two@gmail.com	30	+828388	MALE	Two Singh
3	three@gmail.com	36	+44343423	FEMALE	Xyz Singh
4	naresh@gmail.com	28	+91372372	MALE	Naresh
5	dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
6	suresh@gmail.com	28	+91372372	MALE	Suresh
7	laxmi@gmail.com	28	+91372372	MALE	Suresh
8	vijay@gmail.com	28	+91372372	MALE	Suresh

After Execution/Deletion:

	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	30	+918826111377	MALE	Dilip Singh
2	three@gmail.com	36	+44343423	FEMALE	Xyz Singh
3	naresh@gmail.com	28	+91372372	MALE	Naresh
4	dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
5	suresh@gmail.com	28	+91372372	MALE	Suresh
6	laxmi@gmail.com	28	+91372372	MALE	Suresh
7	vijay@gmail.com	28	+91372372	MALE	Suresh

Requirement: Get Patient Details Based on Email Id.

Here **Email Id** is Primary key Column in table. Finding Details based on Primary key column name Spring JPA provided a method **findById().**

findById(): The **findById()** method is used **to retrieve** an entity by its **primary key or ID** from a relational database. Here's how you can use the **findById()** method in JPA.

Add Below Method in PatientOperations.java:

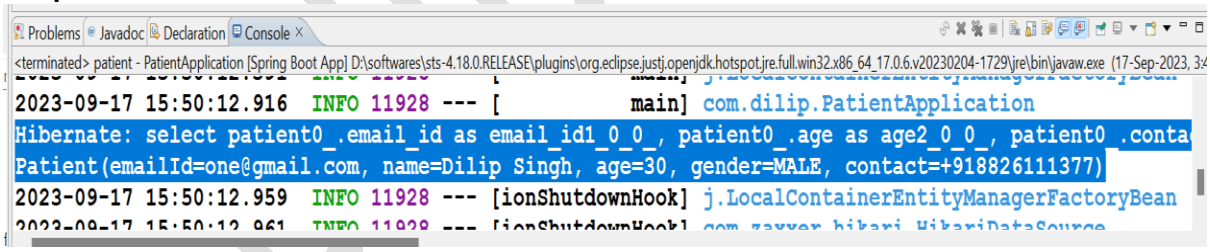
```
package com.dilip.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;

	public void addPatient(Patient p) {
		repository.save(p);
	}
	public void deletePatient(String email) {
		repository.deleteById(email);
	}
```

```
	public Patient fetchByEmailId(String emailId) {
		return repository.findById(emailId).get();
	}
}
```

Testing from Main Class:

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {

public static void main(String[] args) {
	ApplicationContext context = SpringApplication.run(PatientApplication.class, args);
	PatientOperations ops = context.getBean(PatientOperations.class);
	// Fetch Patient Details By Email ID
	Patient patient = ops.fetchByEmailId("one@gmail.com");
	System.out.println(patient);
}
}
```

Output in Console:



Similarly Spring JPA provided many useful and predefined methods inside JPA repositories to perform CRUD Operations.

For example :

findAll() : for retrieve all Records From Table
deleteAll(): for deleting all Records From Table
etc..

For Non Primary Key columns of Table or Entity, Spring JPA provided Custom Query Repository Methods. Let's Explore.

Spring Data JPA Custom Query Repository Methods:

Spring Data JPA allows you to define custom repository methods by simply declaring method signature with **entity class property Name** which is aligned with Database column. The method name must start with **findBy**, **getBy**, **queryBy**, **countBy**, or **readBy**. The **findBy** is mostly used by the developer.

For Example: Below query methods are valid and gives same result like Patient name matching data from Database.

```
public List<Patient> findByName(String name);
public List<Patient> getByName(String name);
public List<Patient> queryByName(String name);
public List<Patient> countByName(String name);
public List<Patient> readByName(String name);
```

- **Patient:** Name of Entity class.
- **Name:** Property name of Entity.

Rule: After **findBy**, The first character of Entity class field name should Upper case letter. Although if we write the first character of the field in lower case then it will work but we should use **camelCase** for the method names. Equal Number of Method Parameters should be defined in abstract method.

Requirement: Get Details of Patients by Age i.e. Single Column.
Result we will get More than One record i.e. List of Entity Objects. So return type is List<Patient>

Step 1: Create **a Custom method inside Repository**

```
package com.dilip.repository;

import java.util.List;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {
        List<Patient> findByAge(int age);
}
```

Step 2: Now call Above Method inside Db operations to pass Age value.

```
package com.dilip.operations;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;

	// Non- primary Key column
	public List<Patient> fetchByAge(int age) {
		return repository.findByAge(age);
	}
}
```

Step 3: Now Test It from Main Class.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {
public static void main(String[] args) {

	ApplicationContext context = SpringApplication.run(PatientApplication.class, args);
	PatientOperations ops = context.getBean(PatientOperations.class);
	//Fetch Patient Details By Age
	List<Patient> patients = ops.fetchByAge(31);
	System.out.println(patients);
}
}
```

Output: In Below, Query generated by JPA and Executed. Got Two Entity Objects In Side List .

```
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient
p1_0 where p1_0.age=?
[Patient [name=Dilip Singh, age=31, gender=MALE, contact=+918826111377,
emailId=one@gmail.com], Patient [name=Dilip Singh, age=31, gender=MALE, contact=
+9188882828, emailId=dilip@gmail.com]]
```

Similar to **Age**, we can fetch data with other columns as well by defining custom Query methods.

Fetching Data with Multiple Columns:

Rule: We can write the query method using multiple fields using predefined keywords(eg. **And**, **Or** etc) but these keywords are case sensitive. **We must use "And" instead of "and".**

Requirement: Fetch Data with Age and Gender Columns.

- Age is 28
- Gender is Female

Step 1: Create a Custom method inside Repository.

Method should have 2 parameters **age** and **gender** in this case because we are getting data with 2 properties.

```
package com.dilip.repository;

import java.util.List;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {
        List<Patient> findByAgeAndGender(int age, String gender);
}
```

Step 2: Now call Above Method inside Db operations to pass Age and gender values.

```
package com.dilip.operations;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {


        @Autowired
        PatientRepository repository;

        // based on Age + Gender
        public List<Patient> getPateintsWithAgeAndGender(int age, String gender) {
                return repository.findByAgeAndGender(age, gender);
```

```
	}
}
```

Step 3: Now Test It from Main Class.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import java.util.List;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {
	public static void main(String[] args) {
		ApplicationContext context
				= SpringApplication.run(PatientApplication.class, args);
		PatientOperations ops = context.getBean(PatientOperations.class);

		//Fetch Patient Details By Age and Gender
		List<Patient> patients = ops.getPateintsWithAgeAndGender(28,"FEMALE");
		System.out.println(patients);
	}
}
```

Table Data:

	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	31	+918826111377	MALE	Dilip Singh
2	three@gmail.com	36	+44343423	FEMALE	Xyz Singh
3	naresh@gmail.com	28	+91372372	MALE	Naresh
4	vijay45@gmail.com	28	+91372372	MALE	Suresh
5	dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
6	suresh@gmail.com	28	+91372372	MALE	Suresh
7	laxmi@gmail.com	28	+91372372	FEMALE	Laxmi
8	vijay@gmail.com	28	+91372372	MALE	Suresh

Expected Output:

```
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient
p1_0 where p1_0.age=? and p1_0.gender=?
[Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com]]
```

We can write the query method if we want to restrict the number of records by directly providing the number as the digit in method name. We need to add the First or the Top keywords before the **By** and after **find**.

```
public List<Student> findFirst3ByName(String name);
public List<Student> findTop3ByName(String name);
```

Both query methods will return only first 3 records.

Similar to these examples and operations we can perform multiple Database operations however we will do in SQL operations.

List of keywords used to write custom repository methods:

And, Or, Is, Equals, Between, LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, After, Before, IsNull, IsNotNull, NotNull, Like, NotLike, StartingWith, EndingWith, Containing, OrderBy, Not, In, NotIn, True, False, IgnoreCase.

Some of the examples for Method Names formations:

```
public List<Student> findFirst3ByName(String name);
public List<Student> findByNameIs(String name);
public List<Student> findByNameEquals(String name);
public List<Student> findByRollNumber(String rollNumber);
public List<Student> findByUniversity(String university);
public List<Student> findByNameAndRollNumber(String name, String rollNumber);
public List<Student> findByRollNumberIn(List<String> rollNumbers);
public List<Student> findByRollNumberNotIn(List<String> rollNumbers);
public List<Student> findByRollNumberBetween(String start, String end);
public List<Student> findByNameNot(String name);
public List<Student> findByNameContainingIgnoreCase(String name);
public List<Student> findByNameLike(String name);
public List<Student> findByRollNumberGreaterThan(String rollnumber);
public List<Student> findByRollNumberLessThan(String rollnumber);
```

@GeneratedValue Annotation:

In **Java Persistence API (JPA)**, the **@GeneratedValue** annotation is used to specify how primary key values for database entities should be generated. This annotation is typically used in conjunction with the **@Id** annotation, which marks a field or property as the primary key of

an entity class. The **@GeneratedValue** annotation provides options for automatically generating primary key values when inserting records into a database table.

When you annotate a field with **@GeneratedValue**, you're telling Spring Boot to automatically generate unique values for that field.

Here are some of the key attributes of the **@GeneratedValue** annotation:

strategy:

This attribute specifies the generation strategy for primary key values. This is used to specify how to auto-generate the field values. There are five possible values for the strategy element on the **GeneratedValue** annotation: **IDENTITY**, **AUTO**, **TABLE**, **SEQUENCE** and **UUID**. These five values are available in the enum, **GeneratorType**.

1. **GenerationType.AUTO**: This is the default strategy. The JPA provider selects the most appropriate strategy based on the database and its capabilities. Assign the field a generated value, leaving the details to the JPA vendor. Tells JPA to pick the strategy that is preferred by the used database platform.

   The preferred strategies are IDENTITY for MySQL, SQLite and MsSQL and SEQUENCE for Oracle and PostgreSQL. This strategy provides full portability.

2. **GenerationType.IDENTITY**: The primary key value is generated by the database system itself (e.g., auto-increment in MySQL or identity columns in SQL Server, SERIAL in PostgreSQL).

3. **GenerationType.SEQUENCE**: The primary key value is generated using a database sequence. Tells JPA to use a database sequence for ID generation. This strategy does currently not provide full portability. Sequences are supported by Oracle and PostgreSql. When this value is used then **generator** filed is mandatory to specify the generator.

4. **GenerationType.TABLE**: The primary key value is generated using a database table to maintain unique key values. Tells JPA to use a separate table for ID generation. This strategy provides full portability. When this value is used then generator filed is mandatory to specify the generator.

5. **GenerationType.UUID:** Jakarta EE 10 now adds the GenerationType for a **UUID**, so that we can use Universally Unique Identifiers (UUIDs) as the primary key values. Using the GenerationType.UUID strategy, This is the easiest way to generate **UUID** values. Simply annotate the primary key field with the **@GeneratedValue** annotation and set the strategy attribute to **GenerationType.UUID**. The persistence provider will automatically generate a UUID value for the primary key column.

NOTE: Here We are working with Oracle Database. Sometimes Different Databases will exhibit different functionalities w.r.to different Generated Strategies.

Examples For All Strategies:

GenerationType.AUTO:

In JPA, the **GenerationType.AUTO** strategy is the default strategy for generating primary key values. It instructs the persistence provider to choose the most appropriate strategy for generating primary key values based on the underlying database and configuration. This typically maps to either **GenerationType.IDENTITY** or **GenerationType.SEQUENCE**, depending on database capabilities.

When to Use GenerationType.AUTO?

The **GenerationType.AUTO** strategy is a convenient choice for most applications because it eliminates the need to explicitly specify a generation strategy of primary key values. It is particularly useful when you are using a database that supports both **GenerationType.IDENTITY** and **GenerationType.SEQUENCE,** as the persistence provider will automatically select the most efficient strategy for your database.

However, there are some cases where we may want to explicitly specify a generation strategy. For example, if you need to ensure that primary key values are generated sequentially, you should use the **GenerationType.SEQUENCE** strategy. Or, if you need to use a custom generator, you should specify the name of the generator using the generator attribute of the **@GeneratedValue** annotation.

Benefits of GenerationType.AUTO

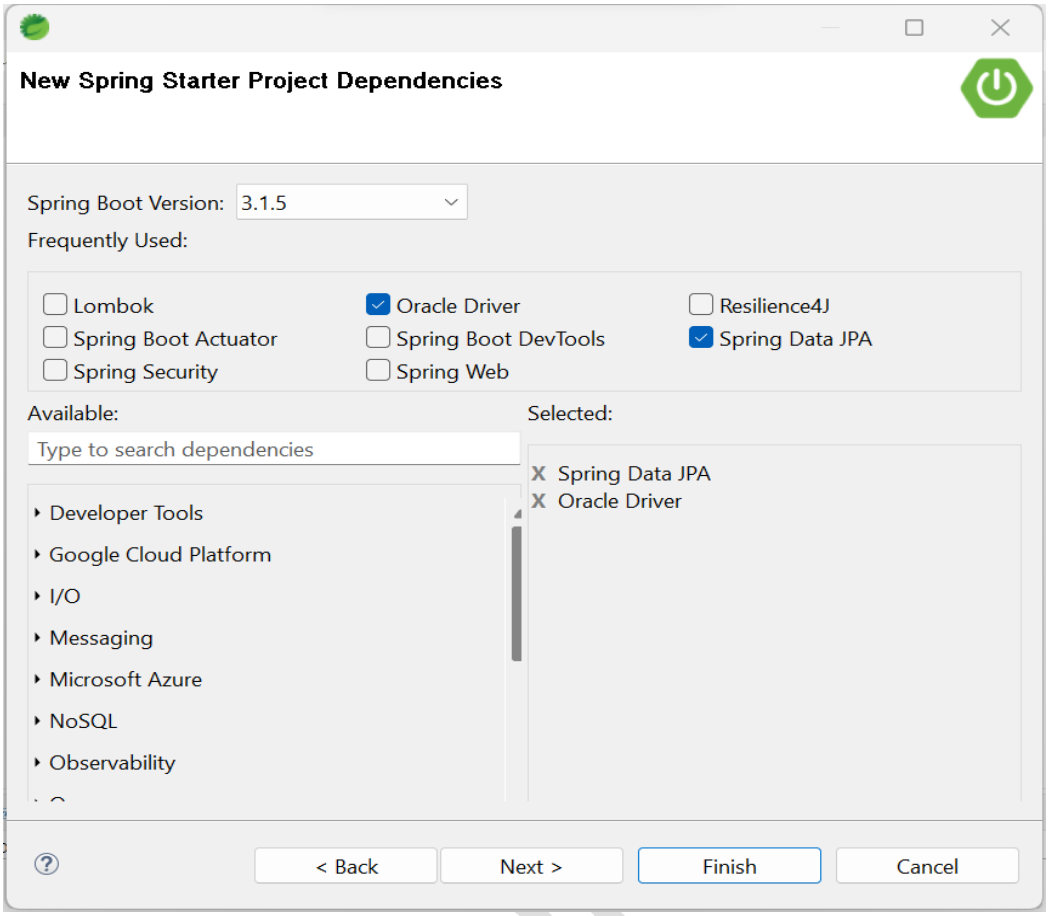
- **Convenience**: It eliminates the need to explicitly specify a generation strategy.
- **Automatic selection**: It selects the most appropriate strategy for the underlying database.
- **Compatibility**: It is compatible with a wide range of databases.

Limitations of GenerationType.AUTO

- **Lack of control**: It may not be the most efficient strategy for all databases.
- **Potential for performance issues:** If the persistence provider selects the wrong strategy, it could lead to performance issues.

Overall, the **GenerationType.AUTO** strategy is a good default choice for generating primary key values in JPA applications. However, you should be aware of its limitations and consider explicitly specifying a generation strategy if you have specific requirements.

- **Create Spring Boot Data JPA project**



➢ **Now Add Database and JPA properties in application.properties file:**

```
#database details
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip

#to print SQL Queries
spring.jpa.show-sql=true

#DDL property
spring.jpa.hibernate.ddl-auto=update
```

➢ **Now Create An Entity class with @GeneratedValue column : `Patient.java`**

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
```

```
@Entity
@Table(name = "patient_details")
public class Patient {

	@Id
	@Column(name = "patient_id")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private long pateintId;

	@Column(name = "patient_name")
	private String name;

	@Column(name = "patient_age")
	private int age;

	public long getPateintId() {
		return pateintId;
	}
	public void setPateintId(long pateintId) {
		this.pateintId = pateintId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
}
```

- **Now Try to Persist Data with above entity class. Create a Repository.**

```
package com.tek.teacher.data;

import org.springframework.data.jpa.repository.JpaRepository;

public interface PatientRepository extends JpaRepository<Patient, Long> {

}
```

- **Now Create Entity Object and try to execute. Here we are not passing pateintId value to Entity Object.**

```
package com.tek.teacher.data;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;

	public void addPatient() {
		Patient patient = new Patient();
		patient.setAge(44);
		patient.setName("naresh Singh");
		repository.save(patient);
	}
}
```

- **Call/Execute above method for persisting data.**

```
package com.tek.teacher;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.tek.teacher.data.PatientOperations;

@SpringBootApplication
public class SpringBootJpaGeneartedvalueApplication {

	 public static void main(String[] args) {
	ApplicationContext context =
	     SpringApplication.run(SpringBootJpaGeneartedvalueApplication.class, args);

	    PatientOperations ops = context.getBean(PatientOperations.class);
	    ops.addPatient();
	 }
}
```

Result : Please Observe in Console Logs, How Spring JPA created values of Primary Key Column of Patient table.

```
Hibernate: create table patient_details (patient_id number(19,0) not null, patient_age number(10,0),
patient_name varchar2(255 char), primary key (patient_id))
Hibernate: create sequence patient_details_seq start with 1 increment by 50
2023-11-09T18:58:22.530+05:30  INFO 20572 --- [           main] j.LocalContainerEntityManagerFactoryBean :
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-09T18:58:22.843+05:30  INFO 20572 --- [           main] t.SpringBootJpaGeneartedvalueApplication :
Started SpringBootJpaGeneartedvalueApplication in 98.978 seconds (process running for 99.614)
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)
2023-11-09T18:58:23.112+05:30  INFO 20572 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean :
```

i.e. Spring JPA created by a new **sequence** for the column **PATIENT_ID** values**.**

Verify Data Inside Table now.

```
select * from patient_details;
```

uery Result ×

SQL | All Rows Fetched: 1 in 0.004 seconds

	PATIENT_ID	PATIENT_AGE	PATIENT_NAME
1	1	44	Dilip Singh

- Now Whenever we are persisting data in **patient_details**, **patient_id** column values will be inserted by executing sequence automatically.
- Execute Logic one more time.

```
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)
2023-11-09T19:24:08.977+05:30  INFO 4812 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean :
```

Table Result :

```
select * from patient_details;
```

y Result ×

SQL | All Rows Fetched: 2 in 0.002 seconds

PATIENT_ID	PATIENT_AGE	PATIENT_NAME
1	44	Dilip Singh
2	44	Dilip Singh

GenerationType.IDENTITY:

This strategy will help us to generate the primary key value by the database itself using the auto-increment or identity of column option. It relies on the database's native support for generating unique values.

➢ **Entity class with IDENTITY Type:**

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

	@Id
	@Column(name = "patient_id")
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private long pateintId;

	@Column(name = "patient_name")
	private String name;

	@Column(name = "patient_age")
	private int age;

	public long getPateintId() {
		return pateintId;
	}
	public void setPateintId(long pateintId) {
		this.pateintId = pateintId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
```

```
        public void setAge(int age) {
                this.age = age;
        }
}
```

- **Now Execute Logic again to Persist Data in table.**
- **If we observe console logs JPA created table as follows**

```
Hibernate: create table patient_details (patient_age number(10,0), patient_id number(19,0) generated as
identity, patient_name varchar2(255 char), primary key (patient_id))
2023-11-09T19:51:16.768+05:30  INFO 15408 --- [           main] j.LocalContainerEntityManagerFactoryBean :
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-09T19:51:17.240+05:30  INFO 15408 --- [           main] t.SpringBootJpaGeneartedvalueApplication :
Started SpringBootJpaGeneartedvalueApplication in 12.662 seconds (process running for 13.861)
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,default)
2023-11-09T19:51:19.387+05:30  INFO 15408 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean :
```

- **For Column patient_id of table patient_details, JPA created IDENTITY column.**
- Oracle introduced a way that allows you to define an identity column for a table, which is similar to the AUTO_INCREMENT column in MySQL or IDENTITY column in SQL Server.
- i.e. If we connected to MySQL and used **GenerationType.IDENTITY** in JPA, then JPA will create AUTO_INCREMENT column to generate Primary Key Values. Similarly If it is SQL Server , then JPA will create IDENTITY column for same scenario.
- The identity column is very useful for the surrogate primary key column. When you insert a new row into the identity column, Oracle auto-generates and insert a sequential value into the column.

Table Data :

```
select * from patient_details;
```

ery Result ×

SQL | All Rows Fetched: 1 in 0.853 seconds

	PATIENT_AGE	PATIENT_ID	PATIENT_NAME
1	44	1	Dilip Singh

Execute Again :

```
2023-11-09T19:59:52.976+05:30  INFO 20280 --- [           main] t.SpringBootJpaGeneartedvalueApplic
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,default)
2023-11-09T19:59:53.424+05:30  INFO 20280 --- [ionShutdownHook] j.LocalContainerEntityManagerFactor
```

Table Result:

```
select * from patient_details;
```

y Result ×

SQL | All Rows Fetched: 2 in 0.002 seconds

PATIENT_AGE	PATIENT_ID	PATIENT_NAME
44	1	Dilip Singh
44	2	Dilip Singh

GenerationType.SEQUENCE:

GenerationType.SEQUENCE is used to specify that a database sequence should be used for generating the primary key value of the entity.

➢ **Entity class with IDENTITY Type:**

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

	@Id
	@Column(name = "patient_id")
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private long pateintId;

	@Column(name = "patient_name")
	private String name;

	@Column(name = "patient_age")
	private int age;

	public long getPateintId() {
		return pateintId;
```

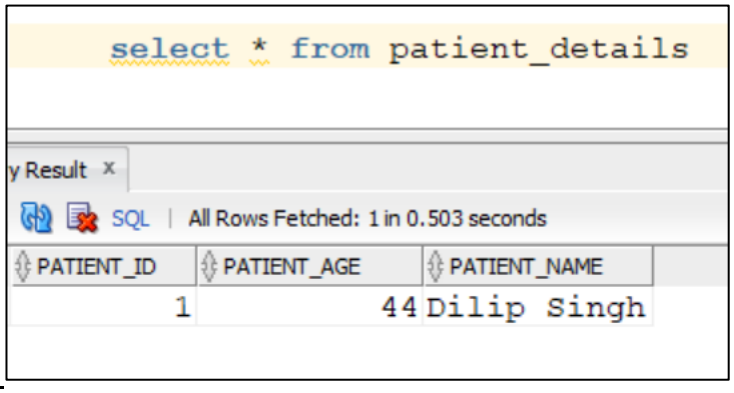
```
	}
	public void setPateintId(long pateintId) {
		this.pateintId = pateintId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
}
```

- **Now Execute Logic to Persist Data in table.**
- **If we observe console logs JPA created table as follows**

```
Hibernate: create sequence patient_details_seq start with 1 increment by 50
Hibernate: create table patient_details (patient_age number(10,0), patient_id number(19,0) not null, patient_name
varchar2(255 char), primary key (patient_id))
2023-11-10T18:26:17.446+05:30  INFO 14180 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized
JPA EntityManagerFactory for persistence unit 'default'
2023-11-10T18:26:17.733+05:30  INFO 14180 --- [           main] t.SpringBootJpaGeneartedvalueApplication : Started
SpringBootJpaGeneartedvalueApplication in 93.094 seconds (process running for 94.269)
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)
```

- JPA created a Sequence to generate unique values. By executing this sequence, values are inserted into Patient table for primary key column.
- Now when we are persisting data inside Patient table by Entity Object, always same sequence will be used for next value.

Table Data :



- Execute Logic for saving data again inside Patient table.
- Primary Key column value is generated from sequence and same supplied to Entity Level.

```
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)
```

Table Data:

```
select * from patient_details
```

ry Result

SQL | All Rows Fetched: 2 in 0.012 seconds

PATIENT_ID	PATIENT_AGE	PATIENT_NAME
1	44	Dilip Singh
2	33	Suresh Singh

This is how JPA will create a sequence when we defined @GeneratedValue(strategy = GenerationType.*IDENTITY*) with **@Id** column of Entity class.

Question: In case, if we want to generate a custom sequence for entity primary key column instead of default sequence created by JPA, do we have any solution?

Yes, JPA provided generator with annotation **@SequenceGenerator,** for creating a custom Sequence should be created by JPA instead of default one like before example.

generator:

This is used to specify the name of the named generator. Named generators are defined using **SequenceGenerator**, **TableGenerator**. When **GenerationType.SEQUENCE** and **GenerationType.TABLE** are used as a strategy then we must specify the generators. Value for this generator field should be the name of **SequenceGenerator**, **TableGenerator.**

@SequenceGenerator Annotation:

Most databases allow you to create native sequences. These are database structures that generate sequential values. The **@SequenceGenerator** annotation is used to represent a named database sequence. This annotation can be kept on class level, member level. **@SequenceGenerator** annotation has the following properties:

Attributes:

1. **name**: The generator name. This property is mandatory.
2. **sequenceName**: The name of the database sequence. If you do not specify the database sequence, your vendor will choose an appropriate default.
3. **initialValue**: The initial sequence value.
4. **allocationSize**: The number of values to allocate in memory for each trip to the database. Allocating values in memory allows the JPA runtime to avoid accessing the database for every sequence request. This number also specifies the amount that the sequence value is incremented each time the sequence is accessed. Defaults to 50.
5. **schema**: The sequence's schema. If you do not name a schema, JPA uses the default schema for the database connection.

Example with Custom Sequence Generator:

- **Create Entity Class With @SequenceGenerator Annotation.**

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

	@Id
	@Column(name = "patient_id")
	@SequenceGenerator(name = "pat_id_seq", sequenceName = "patient_id_seq",
                                        initialValue = 1000, allocationSize = 1)
	@GeneratedValue(strategy = GenerationType.SEQUENCE,
                                                    generator = "pat_id_seq")
	private long pateintId;

	@Column(name = "patient_name")
	private String name;

	@Column(name = "patient_age")
	private int age;

	public long getPateintId() {
		return pateintId;
```

```
	}
	public void setPateintId(long pateintId) {
		this.pateintId = pateintId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
}
```

Data Persisting in Table:

```
package com.tek.teacher.data;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;

	public void addPatient() {
		Patient patient = new Patient();
		patient.setAge(33);
		patient.setName("Dilip Singh");
		repository.save(patient);
	}
}
```

- Now JPA created a custom sequence with details provided as part of annotation @SequenceGenerator inside Entity class Id column.
- **Same Sequence will be executed every time for new Primary key values of column.**

```
Hibernate: create sequence patient_id_seq start with 1000 increment by 1
Hibernate: create table patient_details (patient_age number(10,0), patient_id number(19,0) not null,
patient_name varchar2(255 char), primary key (patient_id))
2023-11-12T10:54:28.955+05:30  INFO 16972 --- [           main] j.LocalContainerEntityManagerFactoryBean
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-12T10:54:29.230+05:30  INFO 16972 --- [           main] t.SpringBootJpaGeneartedvalueApplication
Started SpringBootJpaGeneartedvalueApplication in 3.544 seconds (process running for 4.345)
Hibernate: select patient_id_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)
```

- From Above Console Logs, Sequence Created on database with starting value 1000 and increment always by 1 for next value.

Table Result :

```
select * from patient_details
```

Result

SQL | All Rows Fetched: 1 in 0.104 seconds

PATIENT_ID	PATIENT_AGE	PATIENT_NAME
1000	33	Dilip Singh

- Execute Logic again for Persisting Data. In Below we can see patient ID column value started with 1000 and every time incremented with 1 value continuously.

```
select * from patient_details
```

Result

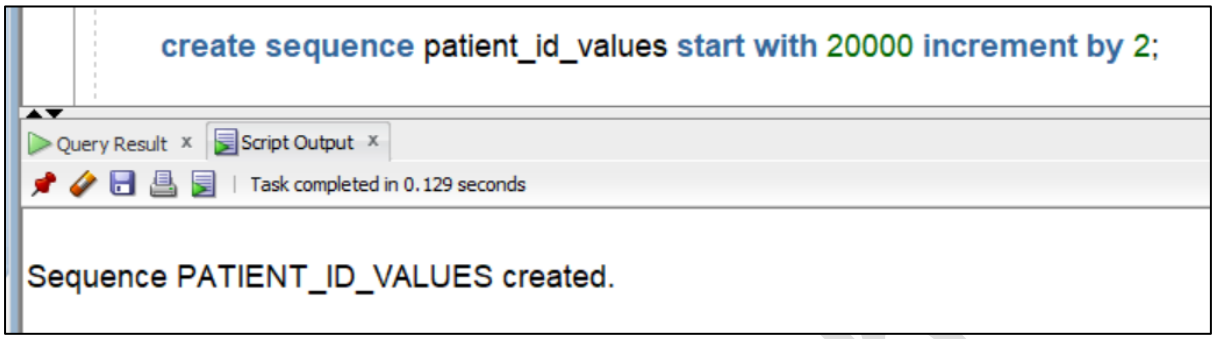
SQL | All Rows Fetched: 4 in 0.004 seconds

PATIENT_ID	PATIENT_AGE	PATIENT_NAME
1000	33	Dilip Singh
1001	33	Dilip Singh
1002	33	Dilip Singh
1003	33	Dilip Singh

This is how we can generate a sequence by providing details with annotation inside Entity class.

How to Use a Sequence already created/available on database inside Entity class:

- Created a new Sequence inside database directly as shown in below.



- Now use above created Sequence with JPA entity class to generate values automatically.

<TBD>

GenerationType.TABLE:

When we use **GenerationType.TABLE**, the persistence provider uses a separate database table to manage the primary key values. A table is created in the database specifically for tracking and generating primary key values for each entity.

This strategy is less common than some others (like **GenerationType.IDENTITY** or **GenerationType.SEQUENCE**) but can be useful in certain scenarios, especially when dealing with databases that don't support identity columns or sequences.

Example With GenerationType.TABLE:

- Create a Entity class and it's ID property should be aligned with **GenerationType.TABLE**

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

	@Id
```

```
	@Column
	@GeneratedValue(strategy = GenerationType.TABLE)
	private long pateintId;

	@Column(name = "patient_name")
	private String name;

	@Column(name = "patient_age")
	private int age;

	public long getPateintId() {
		return pateintId;
	}
	public void setPateintId(long pateintId) {
		this.pateintId = pateintId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
}
```

- In this example, the **@GeneratedValue** annotation is used with the **GenerationType.TABLE** strategy to indicate that the **id** field of **Entity** should have its values generated using a separate table.

- Now Try to insert data inside table **patient_details** from JPA Repository.

```
package com.tek.teacher.data;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;
```

```
        public void addPatient() {
                Patient patient = new Patient();
                patient.setAge(33);
                patient.setName("Dilip Singh");
                repository.save(patient);
        }
}
```

- Now execute Logic and try to monitor in application console logs, how JPA working with GenerationType.TABLE strategy of **GeneratedValue**.

```
Hibernate: create table hibernate_sequences (next_val number(19,0), sequence_name varchar2(255 char) not null,
primary key (sequence_name))
Hibernate: insert into hibernate_sequences(sequence_name, next_val) values ('default',0)
Hibernate: create table patient_details (patient_age number(10,0), pateint_id number(19,0) not null,
patient_name varchar2(255 char), primary key (pateint_id))
2023-11-15T17:11:56.341+05:30  INFO 22708 --- [           main] j.LocalContainerEntityManagerFactoryBean :
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-15T17:11:56.603+05:30  INFO 22708 --- [           main] t.SpringBootJpaGeneartedvalueApplication :
Started SpringBootJpaGeneartedvalueApplication in 4.255 seconds (process running for 4.855)
Hibernate: select tbl.next_val from hibernate_sequences tbl where tbl.sequence_name=? for update
Hibernate: update hibernate_sequences set next_val=?  where next_val=? and sequence_name=?
Hibernate: insert into patient_details (patient_age,patient_name,pateint_id) values (?,?,?)
```

- Now, JPA created a separate database table to manage primary key values of Entity Table as follows.

```
create table hibernate_sequences (next_val number(19,0),
sequence_name varchar2(255 char) not null, primary key
sequence_name))
```

- This Table will be used for generating next Primary key values of our Table **patient_details**

Table Data:

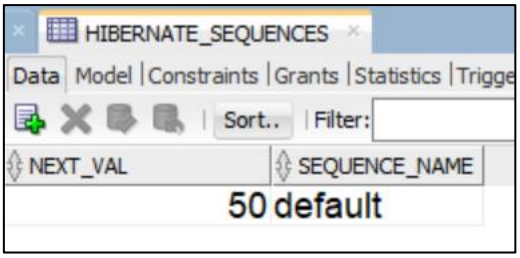
```
select * from patient_details;
```

ry Result

SQL | All Rows Fetched: 1 in 0.809 seconds

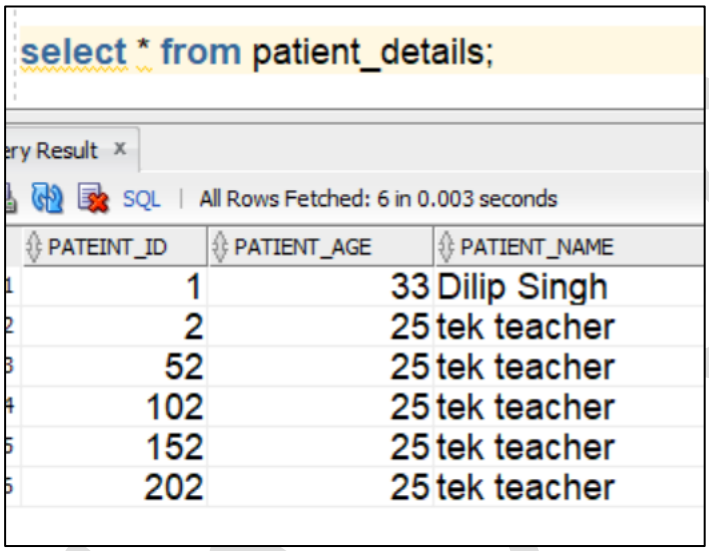
PATEINT_ID	PATIENT_AGE	PATIENT_NAME
1	33	Dilip Singh

Primary Key Table: **`hibernate_sequences`**



Execute Same Logic Again and Again With New Patients Data:

Table Data : Primary key Values are Generated by default with help of table.



Note:

Keep in mind that the choice of the generation strategy depends on the database you are using and its capabilities. Some databases support identity columns (**GenerationType.IDENTITY**), sequences (**GenerationType.SEQUENCE**), or a combination of strategies. The **GenerationType.TABLE** strategy is generally used when other strategies are not suitable for the underlying database.

Question: Can we Generate custom table for strategy of GenerationType.TABLE instead of default one created by JPA?

Answer: Yeah definitely, We can create custom table for managing Primary Key values of Table with help of generator annotation **@TableGenerator.**

JPA @TableGenerator Annotation:

@TableGenerator annotation refers to a database table which is used to store increasing sequence values for one or more entities. This annotation can be kept on class level, member level. **@TableGenerator** has the following properties:

Attributes:

1. **name**: The generator name. This property is mandatory.
2. **table**: The name of the generator table. If left unspecified, database vendor will choose a default table.
3. **schema**: The named table's schema.
4. **pkColumnName**: The name of the primary key column in the generator table. If unspecified, your implementation will choose a default.
5. **valueColumnName**: The name of the column that holds the sequence value. If unspecified, your implementation will choose a default.
6. **pkColumnValue**: The primary key column value of the row in the generator table holding this sequence value. You can use the same generator table for multiple logical sequences by supplying different pkColumnValue s. If you do not specify a value, the implementation will supply a default.
7. **initialValue**: The value of the generator's first issued number.
8. **allocationSize**: The number of values to allocate in memory for each trip to the database. Allocating values in memory allows the JPA runtime to avoid accessing the database for every sequence request.

Example: Create Entity Class: Patient.java

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import jakarta.persistence.TableGenerator;

@Entity
@Table(name = "patient_details")
public class Patient {

	@Id
	@Column
	@TableGenerator(name = "pat_id_generator", table = "pat_id_values",
			pkColumnName= "pat_id",  pkColumnValue = "pat_id_nxt_value",
			    initialValue = 1000, allocationSize = 1)
	@GeneratedValue(strategy = GenerationType.TABLE,
							generator = "pat_id_generator")
	private long pateintId;

	@Column(name = "patient_name")
	private String name;

	@Column(name = "patient_age")
	private int age;
```

```
	public long getPateintId() {
		return pateintId;
	}
	public void setPateintId(long pateintId) {
		this.pateintId = pateintId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
}
```

- In this example, the **@GeneratedValue** annotation is used with the **GenerationType.TABLE** strategy along with custom Table **generator** value.

- Now Try to insert data inside table **patient_details** from JPA Repository.

```
package com.tek.teacher.data;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;

	public void addPatient() {

		Patient patient = new Patient();
		patient.setAge(33);
		patient.setName("Dilip Singh");
		repository.save(patient);
	}
}
```

- Now execute Logic and try to monitor in application console logs, how JPA created our own primary key values table instead of default table data and format.
- JPA created table with custom values provided as part of @TableGenerator annotation and attributes values as shown in console logs.

```
Hibernate: create table pat_id_values (next_val number(19,0), pat_id varchar2(255 char) not null, primary key
(pat_id))
Hibernate: insert into pat_id_values(pat_id, next_val) values ('pat_id_nxt_value',1000)
Hibernate: create table patient_details (patient_age number(10,0), pateint_id number(19,0) not null,
patient_name varchar2(255 char), primary key (pateint_id))
2023-11-16T14:42:36.964+05:30  INFO 12304 --- [           main] j.LocalContainerEntityManagerFactoryBean :
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-16T14:42:37.274+05:30  INFO 12304 --- [           main] t.SpringBootJpaGeneartedvalueApplication :
Started SpringBootJpaGeneartedvalueApplication in 4.898 seconds (process running for 5.542)
Hibernate: select tbl.next_val from pat_id_values tbl where tbl.pat_id=? for update
Hibernate: update pat_id_values set next_val=?  where next_val=? and pat_id=?
Hibernate: insert into patient_details (patient_age,patient_name,pateint_id) values (?,?,?)
```

Tables Result: After multiple Executions of different patients records.

```
select * from patient_details;
```

Query Result

SQL | All Rows Fetched: 5 in 0.003 seconds

	PATEINT_ID	PATIENT_AGE	PATIENT_NAME
1	1001	25	tek teacher
2	1002	25	tek teacher
3	1003	25	tek teacher
4	1004	25	tek teacher
5	1005	25	tek teacher

PAT_ID_VALUES

Data | Model | Constraints | Grants | Statistics | Triggers

Sort.. | Filter:

NEXT_VAL	PAT_ID
1005	pat_id_nxt_value

Question: What is Difference between **GeneratedValue** and **SequenceGenerator / TableGenerator?**

- **GeneratedValue** is used only to get the generated values of column. The two arguments **strategy** and **generator** are used to define how the value is created or gained. We can define to use the database sequence or value from table which is used to store increasing sequence values. But to specify database sequence or table generators name, we specify the named generators to **generator** argument.

- **SequenseGenerator/TableGenerator** is used to define named generators, to map a user defined sequence generator with your JPA session. This is used to give a name to database sequence or database value of table or any kind of generators. This name can be now referred by the **generator** argument of **GeneratedValue**.

As discussed above, we can define Primary key values generators with the help of JPA annotation **@GeneratedValue** and respective Generators.

GenerationType.UUID:

In Spring Data JPA, **UUID**s can be used as the primary key type for entities. Indicates that the persistence provider must assign primary keys for the entity by generating Universally Unique Identifiers. These are non-numerical values like alphanumeric type.

What is UUID?

A **UUID**, or Universally Unique Identifier, is a 128-bit identifier standardized by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE). It is also known as a GUID (Globally Unique Identifier). A UUID is typically expressed as a string of 32 hexadecimal digits, displayed in five groups separated by hyphens, in the form 8-4-4-4-12 for a total of 36 characters (32 alphanumeric characters and 4 hyphens).

For example: **a3335f0a-82ef-47ae-a7e1-1d5c5c3bc4e4**

UUIDs are widely used in various computing systems and scenarios where unique identification is crucial. They are commonly used in databases, distributed systems, and scenarios where it's important to generate unique identifiers without centralized coordination.

In the context of databases and Spring JPA, using UUIDs as primary keys for entities is a way to generate unique identifiers that can be more suitable for distributed systems compared to traditional auto-incremented numeric keys.

Note: we have a pre-defined class in JAVA, **java.util.UUID** for dealing with UUID values. We can consider as String value as well.

Create Entity Class with UUID Generator Strategy:

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
```

```
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

	@Id
	@Column
	@GeneratedValue(strategy = GenerationType.UUID)
	private String pateintId;

	@Column(name = "patient_name")
	private String name;

	@Column(name = "patient_age")
	private int age;

	public String getPateintId() {
		return pateintId;
	}
	public void setPateintId(String pateintId) {
		this.pateintId = pateintId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
}
```

- Now execute Logic and try to monitor in application console logs, how JPA working with GenerationType.**UUID** strategy of **GeneratedValue**.

- **Execute Same Logic Again and Again With New Patients Data:**

Table Data: Primary key **UUID** type Values are Generated and persisted in table.

```
SELECT * FROM PATIENT_DETAILS
```

ts 1 ×

CT * FROM PATIENT_DETAILS | Enter a SQL expression to filter results (use Ctrl+Space)

PATEINTID	PATIENT_AGE	PATIENT_NAME
a3335f0a-82ef-47ae-a7e1-1d5c5c3bc4e4	44	naresh Singh
003e48fd-c1cd-40a1-9aae-1b828efc1397	33	Dilip Singh

Sorting and Pagination in JPA:

Sorting: Sorting is a fundamental operation in data processing that involves arranging a collection of items or data elements in a specific order. The primary purpose of sorting is to make it easier to search for, retrieve, and work with data. Sorting can be done in ascending (from smallest to largest) or descending (from largest to smallest) order, depending on the requirements.

Table and Data:

Welcome Page | dilip | AMAZON_ORDERS

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL

Sort.. | Filter:

	ORDER_ID	AMOUNT	CITY	EMAIL	GENDER	NO_OF_ITEMS	PINCODE
1	3323	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
2	3232	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
3	9988	44444	Chennai	ramesh@gmail.com	MALE	3	600088
4	3344	3000	Hyderbad	dilip@gmail.com	MALE	4	500072
5	1234	4000	Hyderbad	naresh@gmail.com	MALE	2	500072
6	5566	6000	Hyderbad	naresh@gmail.com	MALE	8	500070
7	8888	10000	Chennai	suresh@gmail.com	MALE	10	600099
8	3636	44444	Chennai	ramesh@gmail.com	MALE	3	600088

Requirement: Get Details by Email Id with Sorting

- **Create Spring Boot JPA Project with Lombok Library.**
- **Add Database Properties inside application.properties file**

```
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip

# to print SQL queries executed by JPA
spring.jpa.show-sql=true
```

- **Create Entity Class as Per Database Table.**

```
package com.dilip.dao;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@Table(name = "amazon_orders")
public class AmazonOrders {

	@Id
	@Column(name="order_id")
	private int orderId;

	@Column(name ="no_of_items")
	private int noOfItems;

	@Column(name = "amount")
	private double amount;

	@Column(name="email" )
	private String email;

	@Column(name="pincode")
	private int pincode;

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

```
        @Column(name="gender")
        private String gender;
}
```

- **Now Create A Repository.**

```
package com.dilip.dao;

import org.springframework.data.jpa.repository.JpaRepository;

public interface AmazonOrderRepository extends JpaRepository<AmazonOrders, Integer> {

}
```

- **Now Create a Component Class for Database Operations and Add a Method for Sorting Data**

To achieve this requirement, Spring Boot JPA provided few methods in side **JpaRepository**. Inside **JpaRepository,** JPA provided a method **findAll(…)** with different Arguments.

For Sorting Data : **findAll(Sort sort)**

Sort: In Spring Data JPA, you can use the **Sort** class to specify sorting criteria for your query results. The **Sort** class allows you to define sorting orders for one or more attributes of our entity class. we can use it when working with repository methods to sort query results.

Here's how you can use the **Sort** class in Spring Data JPA:

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

        @Autowired
        AmazonOrderRepository repository;

        // getting order Details in ascending order
```

```
	public void loadDataByemailIdWithSorting() {
		List<AmazonOrders> allOrders = repository.findAll(Sort.by("email"));
		System.out.println(allOrders);
	}
}
```

Note: we have to pass Entity class Property name as part of **by(..)** method, which is related to database table column.

- **Now Execute above Logic**

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.OrdersOperations;

@SpringBootApplication
public class SpringBootJpaSortingPaginationApplication {

	public static void main(String[] args) {

	ApplicationContext context
	   = SpringApplication.run(SpringBootJpaSortingPaginationApplication.class, args);

	 OrdersOperations ops = context.getBean(OrdersOperations.class);
	 ops.loadDataByemailIdWithSorting();
	}
}
```

Output: Table Records are Sorted by email ID and got List of Entity Objects

```
[
AmazonOrders(orderId=3344, noOfItems=4, amount=3000.0, email=dilip@gmail.com,
pincode=500072, city=Hyderbad, gender=MALE),

AmazonOrders(orderId=3232, noOfItems=4, amount=63643.0, email=laxmi@gmail.com,
pincode=500070, city=Hyderbad, gender=FEMALE),

AmazonOrders(orderId=3323, noOfItems=4, amount=63643.0, email=laxmi@gmail.com,
pincode=500070, city=Hyderbad, gender=FEMALE),

AmazonOrders(orderId=1234, noOfItems=2, amount=4000.0, email=naresh@gmail.com,
pincode=500072, city=Hyderbad, gender=MALE),
```

```
AmazonOrders(orderId=5566, noOfItems=8, amount=6000.0, email=naresh@gmail.com,
pincode=500070, city=Hyderbad, gender=MALE),

AmazonOrders(orderId=9988, noOfItems=3, amount=44444.0, email=ramesh@gmail.com,
pincode=600088, city=Chennai, gender=MALE),

AmazonOrders(orderId=3636, noOfItems=3, amount=44444.0, email=ramesh@gmail.com,
pincode=600088, city=Chennai, gender=MALE),

AmazonOrders(orderId=8888, noOfItems=10, amount=10000.0, email=suresh@gmail.com,
pincode=600099, city=Chennai, gender=MALE)

]
```

Requirement: Get Data by sorting with property **noOfItems** of **Descending Order**.

In Spring Data JPA, you can specify the direction (**ascending** or **descending**) for sorting when using the **Sort** class. The **Sort** class allows you to create sorting orders for one or more attributes of our entity class. To specify the direction, you can use the **Direction enum**.

Here's how you can use the **Direction** enum in Spring Data JPA:

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.data.domain.Sort.Direction;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

	@Autowired
	AmazonOrderRepository repository;

	// getting order Details in ascending order of email
	public void loadDataByemailIdWithSorting() {
		List<AmazonOrders> allOrders = repository.findAll(Sort.by("email"));
		System.out.println(allOrders);
	}

	// Get Data by sorting with property noOfItems of Descending Order
	public void loadDataByNoOfItemsWithDescOrder() {
	List<AmazonOrders> allOrders =
				repository.findAll(Sort.by(Direction.DESC, "noOfItems"));
	System.out.println(allOrders);
	}
}
```

Output: We got Entity Objects, by following noOfItems property in Descending Order.

```
[
AmazonOrders(orderId=8888, noOfItems=10, amount=10000.0, email=suresh@gmail.com, pincode=600099, city=Chennai, gender=MALE),
AmazonOrders(orderId=5566, noOfItems=8, amount=6000.0, email=naresh@gmail.com, pincode=500070, city=Hyderbad, gender=MALE),
AmazonOrders(orderId=3232, noOfItems=4, amount=63643.0, email=laxmi@gmail.com, pincode=500070, city=Hyderbad, gender=FEMALE),
AmazonOrders(orderId=3323, noOfItems=4, amount=63643.0, email=laxmi@gmail.com, pincode=500070, city=Hyderbad, gender=FEMALE),
AmazonOrders(orderId=3344, noOfItems=4, amount=3000.0, email=dilip@gmail.com, pincode=500072, city=Hyderbad, gender=MALE),
AmazonOrders(orderId=3636, noOfItems=3, amount=44444.0, email=ramesh@gmail.com, pincode=600088, city=Chennai, gender=MALE),
AmazonOrders(orderId=9988, noOfItems=3, amount=44444.0, email=ramesh@gmail.com, pincode=600088, city=Chennai, gender=MALE),
AmazonOrders(orderId=1234, noOfItems=2, amount=4000.0, email=naresh@gmail.com, pincode=500072, city=Hyderbad, gender=MALE)
]
```

Similarly we can get table Data with Sorting Order based on any table column by using Spring Boot JPA. We can sort data with multiple columns as well.

Example: `repository.findAll(Sort.by("email","noOfItems"));`

Pagination:

Pagination is a technique used in software applications to divide a large set of data or content into smaller, manageable segments called "**pages**." Each page typically contains a fixed number of items, such as records from a database, search results, or content items in a user interface. Pagination allows users to navigate through the data or content one page at a time, making it easier to browse, consume, and interact with large datasets or content collections.

Key features and concepts related to pagination include:

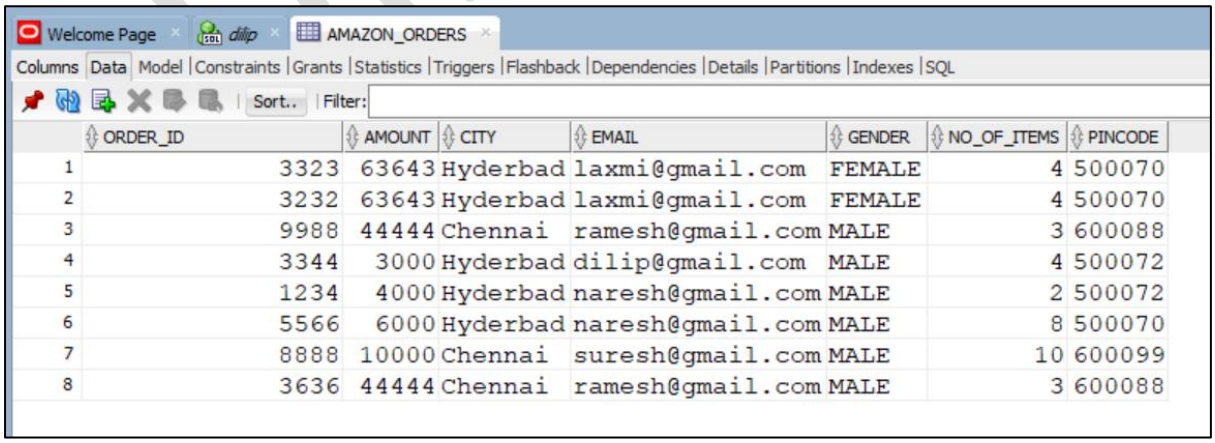
Page Size: The number of items or records displayed on each page is referred to as the "page size" or "items per page." Common page sizes might be 10, 20, 50, or 100 items per page. The choice of page size depends on usability considerations and the nature of the data.

Page Number: Pagination is typically associated with a page number, starting from 1 and incrementing as users navigate through the data. Users can move forward or backward to view different segments of the dataset.
Navigation Controls: Pagination is usually accompanied by navigation controls, such as "Previous" and "Next" buttons or links. These controls allow users to move between pages easily.
Total Number of Pages: The total number of pages in the dataset is determined by dividing the total number of items by the page size. For example, if there are 100 items and the page size is 10, there will be 10 pages.

- Assume a scenario, where we have 200 Records. Each page should get 25 Records, then page number and records are divided as shown below.

Total Records	**200**
Page 1	1 - 25
Page 2	26 - 50
Page 3	51 - 75
Page 4	76 - 100
Page 5	101 - 125
Page 6	126 - 150
Page 7	151 - 175
Page 8	176 - 200

Requirement: Get first set of Records by default with some size from below Table data.



	ORDER_ID	AMOUNT	CITY	EMAIL	GENDER	NO_OF_ITEMS	PINCODE
1	3323	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
2	3232	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
3	9988	44444	Chennai	ramesh@gmail.com	MALE	3	600088
4	3344	3000	Hyderbad	dilip@gmail.com	MALE	4	500072
5	1234	4000	Hyderbad	naresh@gmail.com	MALE	2	500072
6	5566	6000	Hyderbad	naresh@gmail.com	MALE	8	500070
7	8888	10000	Chennai	suresh@gmail.com	MALE	10	600099
8	3636	44444	Chennai	ramesh@gmail.com	MALE	3	600088

In Spring Data JPA, **Pageable** is an interface that allows you to paginate query results easily. It provides a way to specify the page number, the number of items per page (page size), and

optional sorting criteria for your query results. This is particularly useful when you need to retrieve a large set of data from a database and want to split it into smaller pages.

Here's how you can use **Pageable** in Spring JPA:

Pageable.*ofSize*(int size)) : size is, number of records to be loaded.

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

    public void getFirstPageData() {
        List<AmazonOrders> orders = repository.findAll(Pageable.ofSize(2)).getContent();
        System.out.println(orders);
    }
}
```

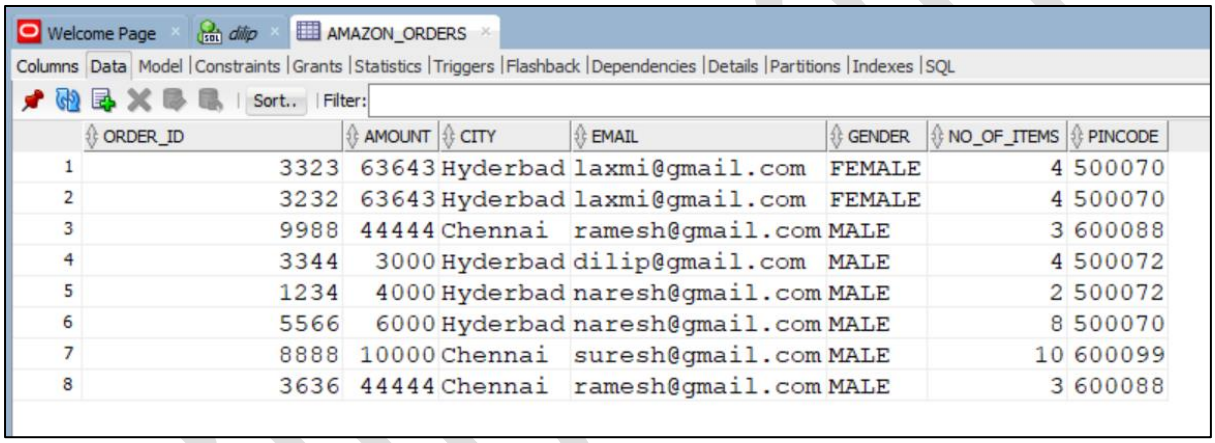
- **Now execute above logic**

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.OrdersOperations;

@SpringBootApplication
public class SpringBootJpaTablesAutoCreationApplication {
    public static void main(String[] args) {
        ApplicationContext context
            = SpringApplication.run(SpringBootJpaTablesAutoCreationApplication.class, args);
        OrdersOperations ops = context.getBean(OrdersOperations.class);
        ops.getFirstPageData();
    }
}
```

Output: We got first 2 records of table.

```
[
AmazonOrders(orderId=3323, noOfItems=4, amount=63643.0, email=laxmi@gmail.com, pincode=500070, city=Hyderbad, gender=FEMALE),

AmazonOrders(orderId=3232, noOfItems=4, amount=63643.0, email=laxmi@gmail.com, pincode=500070, city=Hyderbad, gender=FEMALE),
]
```

- **Requirement:** Get **2nd page** of Records with some size of records i.e. **3 Records**.



	ORDER_ID	AMOUNT	CITY	EMAIL	GENDER	NO_OF_ITEMS	PINCODE
1	3323	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
2	3232	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
3	9988	44444	Chennai	ramesh@gmail.com	MALE	3	600088
4	3344	3000	Hyderbad	dilip@gmail.com	MALE	4	500072
5	1234	4000	Hyderbad	naresh@gmail.com	MALE	2	500072
6	5566	6000	Hyderbad	naresh@gmail.com	MALE	8	500070
7	8888	10000	Chennai	suresh@gmail.com	MALE	10	600099
8	3636	44444	Chennai	ramesh@gmail.com	MALE	3	600088

Here we will use **PageRequest** class which provides pre-defined methods, where we can provide page Numbers and number of records.

Method: PageRequest.of(int page, int size);

Note: In JPA**,** Page Index always Starts with **0** i.e. Page number 2 representing 1 index.

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {
```

```
        @Autowired
        AmazonOrderRepository repository;

        public void getRecordsByPageIdAndNoOfRecords(int pageId, int noOfReorcds) {

                Pageable pageable = PageRequest.of(pageId, noOfReorcds);

                List<AmazonOrders> allOrders = repository.findAll(pageable).getContent();
                System.out.println(allOrders);

        }
}
```

- **Now execute above logic**

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

import com.dilip.dao.OrdersOperations;

@SpringBootApplication
public class SpringBootJpaTablesAutoCreationApplication {
  public static void main(String[] args) {
        ApplicationContext context
            = SpringApplication.run(SpringBootJpaTablesAutoCreationApplication.class, args);
        OrdersOperations ops = context.getBean(OrdersOperations.class);
        ops.getRecordsByPageIdAndNoOfRecords(1,3);
  }
}
```

Output: From our Table data, we got **4-6 Records** which is representing **2nd Page** of Data.

```
[
AmazonOrders(orderId=3344, noOfItems=4, amount=3000.0, email=dilip@gmail.com,
pincode=500072, city=Hyderbad, gender=MALE),

AmazonOrders(orderId=1234, noOfItems=2, amount=4000.0, email=naresh@gmail.com,
pincode=500072, city=Hyderbad, gender=MALE),

AmazonOrders(orderId=5566, noOfItems=8, amount=6000.0, email=naresh@gmail.com,
pincode=500070, city=Hyderbad, gender=MALE)
]
```

Requirement: Pagination with Sorting:

Get **2nd page** of Records with some size of records i.e. **3 Records** along with Sorting by **noOfItems** column in Descending Order.

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.data.domain.Sort.Direction;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

        @Autowired
        AmazonOrderRepository repository;

        public void getDataByPaginationAndSorting(int pageId, int noOFReorcds) {

           List<AmazonOrders> allOrders =
                    repository.findAll(PageRequest.of(pageId, noOFReorcds,
                                        Sort.by(Direction.DESC, "noOfItems"))).getContent();

                    System.out.println(allOrders);
        }
}
```

➢ **Execute Above Logic**

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.OrdersOperations;

@SpringBootApplication
public class SpringBootJpaTablesAutoCreationApplication {
   public static void main(String[] args) {
     ApplicationContext context
          = SpringApplication.run(SpringBootJpaTablesAutoCreationApplication.class, args);
     OrdersOperations ops = context.getBean(OrdersOperations.class);
     ops.getDataByPaginationAndSorting(1,3);
  }
```

```
}
```

Output: We got Entity Objects with Sorting by noOfItems column, and we got 2nd page set of records.

```
[
AmazonOrders(orderId=3344, noOfItems=4, amount=3000.0, email=dilip@gmail.com, pincode=500072, city=Hyderbad, gender=MALE),
AmazonOrders(orderId=3232, noOfItems=4, amount=63643.0, email=laxmi@gmail.com, pincode=500070, city=Hyderbad, gender=FEMALE),
AmazonOrders(orderId=9988, noOfItems=3, amount=44444.0, email=ramesh@gmail.com, pincode=600088, city=Chennai, gender=MALE)
]
```

Native Queries & JPQL Queries with Spring JPA:

Native Query is Custom SQL query. In order to define SQL Query to execute for a Spring Data repository method, we have to annotate the method with the **@Query** annotation. This annotation value attribute contains the SQL or JPQL to execute in Database. We will define **@Query** above the method inside the repository.

Spring Data JPA allows you to execute native SQL queries by using the **@Query** annotation with the **nativeQuery** attribute set to **true**. For example, the following method uses the **@Query** annotation to execute a native SQL query that selects all customers from the database.

```
@Query(value = "SELECT * FROM customer", nativeQuery = true)
public List<Customer> findAllCustomers();
```

The **@Query** annotation allows you to specify the SQL query that will be executed. The **nativeQuery** attribute tells Spring Data JPA to execute the query as a native SQL query, rather than considering it to JPQL.

JPQL Query:

The JPQL (**Java Persistence Query Language**) is an object-oriented query language which is used to perform database operations on persistent entities. Instead of database table, **JPQL** uses entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks. JPQL is developed based on SQL syntax, but it won't affect the database directly. JPQL can retrieve information or data using SELECT clause, can do bulk updates using UPDATE clause and DELETE clause.

By default, the query definition uses JPQL in Spring JPA. Let's look at a simple repository method that returns Users entities based on city value from the database:

// JPQL Query in Repository Layer

```
@Query(value = "Select u from Users u")
List<Users> getUsers ();
```

JPQL can perform:

- It is a platform-independent query language.
- It can be used with any type of database such as MySQL, Oracle.
- join operations
- update and delete data in a bulk.
- It can perform aggregate function with sorting and grouping clauses.
- Single and multiple value result types.

Native SQL Query's:

We can use **@Query** to define our Native Database SQL query. All we have to do is set the value of the **nativeQuery** attribute to **true** and define the native SQL query in the **value** attribute of the annotation.

Example, Below Repository Method representing Native SQL Query to get all records.

```
@Query(value = "select * from flipkart_users", nativeQuery = true)
List<Users> getUsers();
```

For passing values to Positional parameters of SQL Query from method parameters, JPA provides 2 possible ways.

1. Indexed Query Parameters
2. Named Query Parameters

By using Indexed Query Parameters:

If SQL query contains positional parameters and we have to pass values to those, we should use Indexed Params i.e. index count of parameters. For indexed parameters, Spring JPA Data will pass method parameter values to the query in the same order they appear in the method declaration.

Example: Get All Records Of Table

```
@Query(value = "select * from flipkart_users ", nativeQuery = true)
List<Users>  getUsersByCity();
```

Example: Get All Records Of Table where city is matching

Now below method declaration in repository will return List of Entity Objects with city parameter.

```
@Query(value = "select * from flipkart_users where city= ?1 ", nativeQuery = true)
List<Users> getUsersByCity(String city);
```

Example with more indexed parameters: users from either **city** or **pincode** matches.
Example: Get All Records Of Table where city or pincode is matching

```
@Query(value = "select * from flipkart_users where city=?1 or pincode=?2 ", nativeQuery = true)
List<Users> getUsersByCityOrPincode(String cityName, String pincode);
```

Examples:

Requirement:

1. **Get All Patient Details**
2. **Get All Patient with Email Id**
3. **Get All Patients with Age and Gender**

Step 1: Define Methods Inside Repository with Native Queries:

```
package com.dilip.repository;

import java.util.List;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import com.dilip.entity.Patient;


@Component
public interface PatientRepository extends CrudRepository<Patient, String> {

	//Get All Patients
	@Query(value = "select * from patient", nativeQuery = true)
	public List<Patient> getAllPatients();

	//Get All Patient with EmailId
	@Query(value = "select * from patient  where emailid=?1", nativeQuery = true)
	public Patient getDetailsByEmail(String email);

	//Get All Patients with Age and Gender
```

```
    @Query(value = "select * from patient  where age=?1 and gender=?2", nativeQuery = true)
    public List<Patient>  getPatientDetailsByAgeAndGender(int age, String gender);
}
```

➢ Step 2: Call Above Methods from DB Operations Class

```
package com.dilip.operations;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {

        @Autowired
        PatientRepository repository;

        //Select all patients
        public List<Patient> getPatientDetails() {
                return repository.getAllPatients();
        }
        //by email Id
        public Patient getPatientDetailsbyEmailId(String email) {
                return repository.getDetailsByEmail(email);
        }
        //age and gender
        public List<Patient> getPatientDetailsbyAgeAndGender(int age, String gender) {
                return repository.getPatientDetailsByAgeAndGender(age, gender);
        }
}
```

➢ Step 3: Testing From Main Method class

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;
import java.util.List;
```

```
@SpringBootApplication
public class PatientApplication {
	public static void main(String[] args) {
		ApplicationContext context =
					SpringApplication.run(PatientApplication.class, args);

		PatientOperations ops = context.getBean(PatientOperations.class);

		//All Patients
		List<Patient> allPatients = ops.getPatientDetails();
		System.out.println(allPatients);

		//By email Id
		System.out.println("******** with email Id **************");
		Patient patient = ops.getPatientDetailsbyEmailId("laxmi@gmail.com");
		System.out.println(patient);

		//By Age and Gender
	System.out.println("******** PAteints with Age and gender**************");
		List<Patient> patients = ops.getPatientDetailsbyAgeAndGender(31,"MALE");
		System.out.println(patients);
	}
}
```

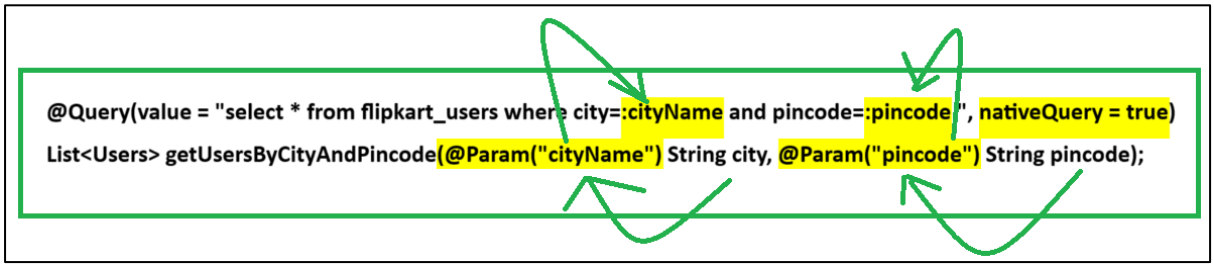
Output:

```
Hibernate: select * from patient
[Patient [name=Naresh, age=28, gender=MALE, contact=+91372372, emailId=naresh@gmail.com], Patient [name=Rakhi, age=44,
gender=MALE, contact=+91372372, emailId=Rakhi@gmail.com], Patient [name=Dilip Singh, age=31, gender=MALE, contact=
+9188882828, emailId=dilip@gmail.com], Patient [name=Suresh, age=28, gender=MALE, contact=+91372372,
emailId=suresh@gmail.com], Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com],
Patient [name=Anusha, age=31, gender=FEMALE, contact=+9188882828, emailId=anusha@gmail.com]]
******** with email Id **************
Hibernate: select * from patient  where emailid=?
Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com]
******** PAteints with Age and gender**************
Hibernate: select * from patient  where age=? and gender=?
[Patient [name=Dilip Singh, age=31, gender=MALE, contact=+9188882828, emailId=dilip@gmail.com]]
```

By using Named Query Parameters:

We can also pass values of method parameters to the query using named parameters i.e. we are providing We define these using the **@Param** annotation inside our repository method declaration. Each parameter annotated with **@Param** must have a value string matching the

corresponding **JPQL or SQL** query parameter name. A query with named parameters is easier to read and is less error-prone in case the query needs to be refactored.



NOTE: In JPQL also, we can use index and named Query parameters.

Requirement:

1. **Insert Patient Data**

Step 1: Define Method Inside Repository with Native Query:

```
package com.dilip.dao;

import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import com.dilip.entity.Patient;


@Component
public interface PatientRepository extends CrudRepository<Patient, String> {

//adding Patient Details
@Transactional
@Modifying
@Query(value = "INSERT INTO patient VALUES(:emailId,:age,:contact,:gender,:name)",
nativeQuery = true)
public void addPAtient( @Param("name") String name,
                        @Param("emailId") String email,
                        @Param("age") int age,
                        @Param("contact") String mobile,
                        @Param("gender")  String gender );
}
```

Step 2: Call Above Method from DB Operations Class

```
package com.dilip.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {

	@Autowired
	PatientRepository repository;

	//add Pateint
public void addPAtient(String name,  String email, int age, String mobile, String gender) {
	repository.addPAtient(name, email, age, mobile, gender);
}
}
```

Step 3: Test it From Main Method class.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {
	public static void main(String[] args) {
		ApplicationContext context =
				SpringApplication.run(PatientApplication.class, args);
		PatientOperations ops = context.getBean(PatientOperations.class);
		//add Pateint
		System.out.println("*** Adding Patient ************");
		ops.addPAtient("Rakhi", "Rakhi@gmail.com", 44, "+91372372", "MALE");
	}
}
```

Output:

```
*** Adding Pateint ************
Hibernate: INSERT INTO patient VALUES(?,?,?,?,?)
```

In above We executed DML Query, So it means some Modification will happen finally in Database Tables data. In Spring JPA, for **DML Queries like insert, update and delete** provided mandatory annotations **@Transactional** and **@Modifying**. We should declare these annotations while executing DML Queries.

@Transactional:

Package : **org.springframework.transaction.annotation.Transactional**

In Spring Framework, the **@Transactional** annotation is used to indicate that a method, or all methods within a class, should be executed within a transaction context. Transactions are used to ensure data integrity and consistency in applications that involve database operations. Specifically, when used with Spring Data JPA, the **@Transactional** annotation plays a significant role in managing transactions around JPA (Java Persistence API) operations.

Describes a transaction attribute on an individual method or on a class. When this annotation is declared at the class level, it applies as a default to all methods of the declaring class and its subclasses. If no custom rollback rules are configured in this annotation, the transaction will roll back on **RuntimeException** and **Error** but not on checked exceptions.

@Modifying:

The **@Modifying** annotation in Spring JPA is used to indicate that a method is a modifying query, which means that it will update, delete, or insert data in the database. This annotation is used in conjunction with the **@Query** annotation to specify the query that the method will execute. The **@Modifying** annotation is a powerful tool that can be used to update, delete, and insert data in the database. It is often used in conjunction with the **@Transactional** annotation to ensure that the data is updated or deleted in a safe and consistent manner.

Here are some of the benefits of using the **@Modifying** annotation:

- It makes it easy to update, delete, and insert data in the database.
- It can be used in conjunction with the **@Transactional** annotation to ensure that the data is updated or deleted in a safe and consistent manner.
- It can be used to optimize performance by batching updates and deletes.

If you are developing an application that needs to update, delete, or insert data in the database, I highly recommend using the **@Modifying** annotation. It is a powerful tool that can help you to improve the performance and reliability of your application.

JPQL Queries Execution:

Examples for executing JPQL Query's. Here We will not use **nativeQuery** attribute means by default **false** value. Then Spring JPA considers **@Query** Value as JPQL Query.

Requirement:

- **Fetch All Patients**
- **Fetch All Patients Names**
- **Fetch All Male Patients Names**

Step1: Define Repository Methods along with JPQL Queries.

```
package com.dilip.repository;

import java.util.List;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {

	//JPQL  Queries
	//Fetch All Patients
	@Query(value="Select p from Patient p")
	public List<Patient> getAllPAtients();

	//Fetch All Patients Names
	@Query(value="Select p.name from Patient p")
	public List<String> getAllPatientsNames();

	 //Fetch All Male Patients Names
	@Query(value="Select p from Patient p where gender=?1")
	public List<Patient> getPatientsByGender(String gender);
}
```

Step 2: Call Above Methods From DB Operations class.

```
package com.dilip.operations;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {
```

```
	@Autowired
	PatientRepository repository;

	// All Patients
	public List<Patient> getAllpatients() {
		return repository.getAllPAtients();
	}
	// All Patients Names
	public List<String> getAllpatientsNames() {
		return repository.getAllPatientsNames();
	}
	// All Patients Names
	public List<Patient> getAllpatientsByGender(String gender) {
		return repository.getPatientsByGender(gender);
	}
}
```

Step 3: Test it From Main Method class.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {
	public static void main(String[] args) {
		ApplicationContext context =
				SpringApplication.run(PatientApplication.class, args);
		PatientOperations ops = context.getBean(PatientOperations.class);
		System.out.println("====> All Patients Details ");
		System.out.println(ops.getAllpatients());
		System.out.println("====> All Patients Names ");
		System.out.println(ops.getAllpatientsNames());
		System.out.println("====> All MALE Patients Details ");
		System.out.println(ops.getAllpatientsByGender("MALE"));
	}
}
```

Output:

```
<terminated> PatientApplication (1) [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe  (10-Aug-2023, 9:43:35 am – 9:44
====> All Patients Details
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0
[Patient [name=Naresh, age=28, gender=MALE, contact=+91372372, emailId=naresh@gmail.com], Patient [name=Rakhi, age=44,
gender=MALE, contact=+91372372, emailId=Rakhi@gmail.com], Patient [name=Dilip Singh, age=31, gender=MALE, contact=
+9188882828, emailId=dilip@gmail.com], Patient [name=Suresh, age=28, gender=MALE, contact=+91372372,
emailId=suresh@gmail.com], Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com],
Patient [name=Anusha, age=31, gender=FEMALE, contact=+9188882828, emailId=anusha@gmail.com]]
====> All Patients Names
Hibernate: select p1_0.name from Patient p1_0
[Naresh, Rakhi, Dilip Singh, Suresh, Laxmi, Anusha]
====> All MALE Patients Details
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.gender=?
[Patient [name=Naresh, age=28, gender=MALE, contact=+91372372, emailId=naresh@gmail.com], Patient [name=Rakhi, age=44,
gender=MALE, contact=+91372372, emailId=Rakhi@gmail.com], Patient [name=Dilip Singh, age=31, gender=MALE, contact=
+9188882828, emailId=dilip@gmail.com], Patient [name=Suresh, age=28, gender=MALE, contact=+91372372,
emailId=suresh@gmail.com]]
```

Internally JPA Translates JPQL queries to Actual Database SQL Queries and finally those queries will be executed. We can see those queries in Console Messages.

JPQL Query Guidelines

JPQL queries follow a set of rules that define how they are parsed and executed. These rules are defined in the JPA specification. Here are some of the key rules of JPQL:

- The SELECT clause: The SELECT clause specifies the entities that will be returned by the query.
- The FROM clause: The FROM clause specifies the entities that the query will be executed against.
- The WHERE clause: The WHERE clause specifies the conditions that the entities must meet in order to be included in the results of the query.
- The GROUP BY clause: The GROUP BY clause specifies the columns that the results of the query will be grouped by.
- The HAVING clause: The HAVING clause specifies the conditions that the groups must meet in order to be included in the results of the query.
- The ORDER BY clause: The ORDER BY clause specifies the order in which the results of the query will be returned.

Here are some additional things to keep in mind when writing JPQL queries:

- JPQL queries are case-insensitive. This means that you can use the names of entities and columns in either upper or lower case.
- JPQL queries can use parameters. Parameters are variables that can be used in the query to represent values that are not known at the time the query is written.

Relationship Mapping with JPA:

In Java Persistence API (JPA), relationship mappings refer to the way in which entities are connected or associated with each other in a relational database. JPA provides annotations that allow you to define these relationships in your Java code, and these annotations influence how the corresponding tables and foreign key constraints are created in the underlying database.

Here are JPA relationship mappings:

1. One-to-One (1:1) Relationship:

- An entity A is associated with only one entity B, and vice versa.

Example: An Employee has one Address.

2. One-to-Many (1:N) Relationship:

- An entity A is associated with many instances of entity B, but each instance of entity B is associated with only one instance of entity A.

Example: An Employee has more than one Address.

3. Many-to-One (N:1) Relationship:

- The reverse of a One-to-Many relationship. Many instances of entity A can be associated with one instance of entity B.

Example: Many Employees belong to one Role.

4. Many-to-Many (N:N) Relationship:

- Many instances of entity A are associated with many instances of entity B, and vice versa.

Example: A Employee can enrol in many Roles, and a Role can have many Employees.

5. Unidirectional and Bidirectional Relationships:

- In a unidirectional relationship, one entity knows about the other, but the other entity is not aware of the relationship. In a bidirectional relationship, both entities are aware of the relationship.

Example: A Post has many Comments (unidirectional) vs. A Comment belongs to a P ost, and a Post has many Comments (bidirectional).

To map these relationships, JPA uses annotations such as **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany**, and additional annotations like **@JoinColumn** and **@JoinTable** to define the details of the database schema.

Cascading and Cascade Types:

What Is Cascading?

Entity relationships often depends on the existence of another entity, **for example** the **Employee–Address relationship**. Without the **Employee**, the **Address** entity doesn't have any meaning of its own. When we delete the **Employee** entity, our **Address** entity should also get deleted.

When we perform some action on the target entity, the same action will be applied to the associated entity. Cascading is the way to achieve this. o enable this behaviour, we had use "**CascadeType**" attribute with mappings. To establish a dependency between related entities, JPA provides **jakarta.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 Cascade Types:

JPA allows us to propagate the state transition from a parent entity to the associated child entity. For this purpose, JPA defines various cascade types under **CascadeType** Enum.

- **PERSIST**
- **MERGE**
- **REMOVE**
- **REFRESH**
- **DETACH**
- **ALL**

Type	Description
PERSIST	if the parent entity is persisted then all its related entity will also be persisted.
MERGE	if the parent entity is merged then all its related entity will also be merged.
DETACH	if the parent entity is detached then all its related entity will also be detached.
REFRESH	if the parent entity is refreshed then all its related entity will also be refreshed.
REMOVE	if the parent entity is removed then all its related entity will also be removed.
ALL	In this case, all the above cascade operations can be applied to the entities related to parent entity. The value is equivalent to cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}

CascadeType.PERSIST:

CascadeType.PERSIST is a cascading type in JPA that specifies that the create (or persist) operation should be cascaded from the parent entity to the child entities.

When **CascadeType.PERSIST** is used, any new child entities associated with a parent entity will be automatically persisted when the parent entity is persisted. However, updates or deletions made to the parent entity will not be cascaded to the child entities.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.PERSIST**, any new Order entities associated with a Customer entity will be persisted when the Customer entity is persisted. However, if you update or delete a Customer entity, any associated Order entities will not be automatically updated or deleted.

CascadeType.MERGE:

CascadeType.MERGE is a cascading type in JPA that specifies that the update (or merge) operation should be cascaded from the parent entity to the child entities.

When **CascadeType.MERGE** is used, any changes made to a detached parent entity (i.e., an entity that is not currently managed by the persistence context) will be automatically merged with its associated child entities when the parent entity is merged back into the persistence context. However, new child entities that are not already associated with the parent entity will not be automatically persisted.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.MERGE**, any changes made to a detached Customer entity (such as changes made in a different session or transaction) will be automatically merged with its associated Order entities when the Customer entity is merged back into the persistence context.

CascadeType.REMOVE:

CascadeType.REMOVE is a cascading type in JPA that specifies that the delete operation should be cascaded from the parent entity to the child entities.

When **CascadeType.REMOVE** is used, any child entities associated with a parent entity will be automatically deleted when the parent entity is deleted. However, updates or modifications made to the parent entity will not be cascaded to the child entities.

For example, consider a scenario where we have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.REMOVE**, any Order entities associated with a Customer entity will be automatically deleted when the Customer entity is deleted.

CascadeType.REFRESH:

CascadeType.REFRESH is a cascading type in JPA that specifies that the refresh operation should be cascaded from the parent entity to the child entities.

When **CascadeType.REFRESH** is used, any child entities associated with a parent entity will be automatically refreshed when the parent entity is refreshed. This means that the latest state of the child entities will be loaded from the database and any changes made to the child entities will be discarded.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.REFRESH**, any associated Order entities will be automatically refreshed when the Customer entity is refreshed.

CascadeType.DETACH:

CascadeType.DETACH is a cascading type in JPA that specifies that the detach operation should be cascaded from the parent entity to the child entities.

When **CascadeType.DETACH** is used, any child entities associated with a parent entity will be automatically detached when the parent entity is detached. This means that the child entities will become detached from the persistence context and their state will no longer be managed.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using CascadeType.DETACH, any associated Order entities will be automatically detached when the Customer entity is detached.

CascadeType.ALL:

CascadeType.ALL is a cascading type in JPA that specifies that all state transitions (create, update, delete, and refresh) should be cascaded from the parent entity to the child entities.

When **CascadeType.ALL** is used, and any operation performed on the parent entity will be automatically propagated to all child entities. This means that if you persist, update, or delete a parent entity, all child entities associated with it will also be persisted, updated, or deleted accordingly.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.ALL**, any operation performed on the Customer entity (such as persist, merge, remove, or refresh) will also be propagated to all associated Order entities.

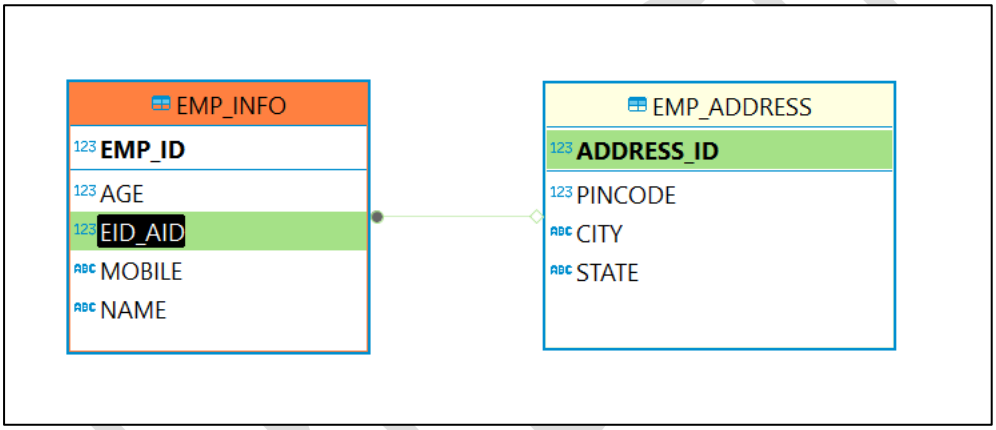
One-to-One (1:1) Relationship:

A one-to-one relationship is a relationship where a record in one table is associated with exactly one record in another table i.e. a record in one entity (table) is associated with exactly one record in another entity (table).

@OneToOne annotation is used to define a one-to-one relationship between two entities. This means that one instance of an entity is associated with exactly one instance of another entity, and vice versa. In database terms, this often translates to a shared primary key or a unique constraint on one of the tables. One way, we can implement a one-to-one relationship in a database is to add a new column and make it as foreign key.

Here's a example of how to use **@OneToOne** in JPA:

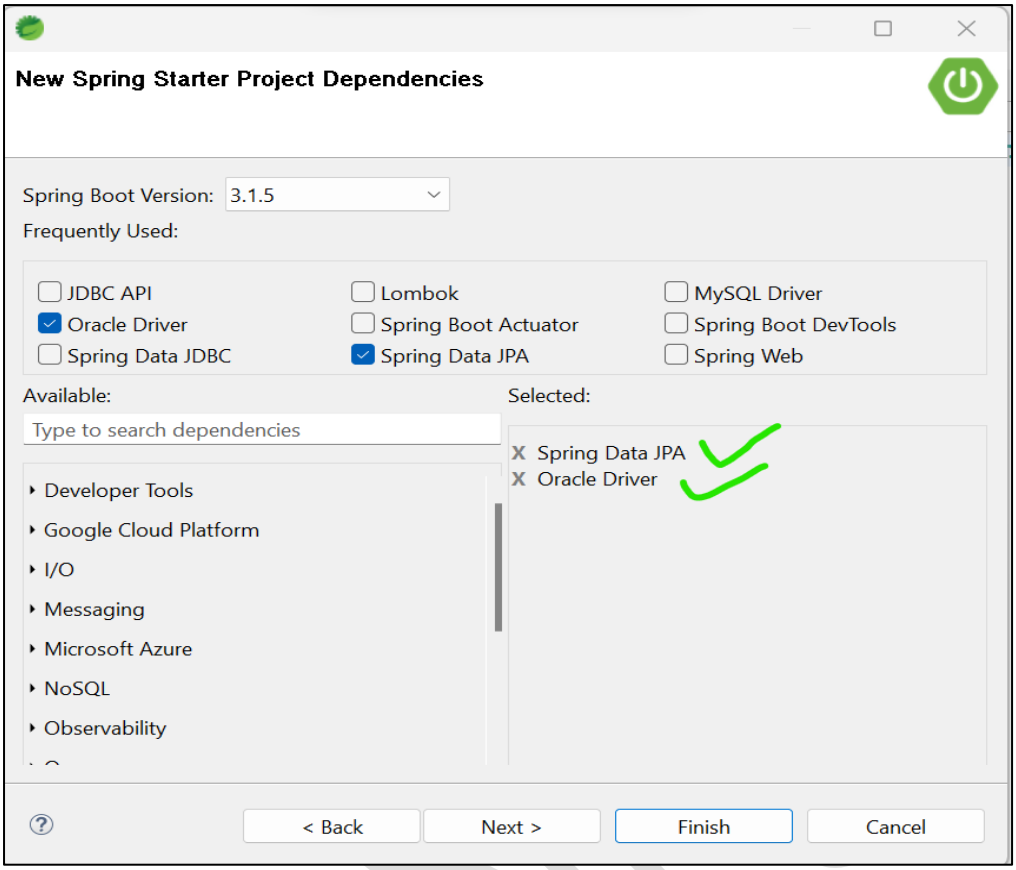
Requirement: Add Employee and Address Details with One to One Relationship as shown in below Entity Relationship Diagram.



- From Above ER diagram, **EID_AID** column of **EMP_INFO** table representing a Foreign Key relationship with another table **EMP_ADDRESS** primary key column **ADDRESS_ID.**

Implementation:

- Now Create a Spring Boot JPA Project.



- After Project Creation, Please add database details inside **application.properties** file

```
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

- Now Create Entity Classes as details shown ER diagram.

Entity Class : Address.java

```
package com.dilip.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_address")
```

```
public class Address {

	@Id
	@Column
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int addressId;

	@Column
	private String city;

	@Column
	private int pincode;

	@Column
	private String state;

	public int getAddressId() {
		return addressId;
	}
	public void setAddressId(int addressId) {
		this.addressId = addressId;
	}
	public String getCity() {
		return city;
	}
	public void setCity(String city) {
		this.city = city;
	}
	public int getPincode() {
		return pincode;
	}
	public void setPincode(int pincode) {
		this.pincode = pincode;
	}
	public String getState() {
		return state;
	}
	public void setState(String state) {
		this.state = state;
	}
}
```

Entity Class : Employye.java

```
package com.dilip.entity;

import jakarta.persistence.CascadeType;
```

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_info")
public class Employye {

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

	@Column
	private String name;

	@Column
	private int age;

	@Column
	private String mobile;

	@OneToOne(cascade = CascadeType.ALL)
	@JoinColumn(name = "eid_aid")
	Address address;

	public Long getEmpId() {
		return empId;
	}
	public void setEmpId(Long empId) {
		this.empId = empId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
```

```
		this.age = age;
	}
	public String getMobile() {
		return mobile;
	}
	public void setMobile(String mobile) {
		this.mobile = mobile;
	}
	public Address getAddress() {
		return address;
	}
	public void setAddress(Address address) {
		this.address = address;
	}
}
```

@JoinColumn:

In JPA, the **@JoinColumn** annotation is used to define the column that will be used to join two entities in an association. It is typically used in one-to-many, many-to-one, and many-to-many relationships.

The **@JoinColumn** annotation has several attributes that can be used to customize the join column mapping. Some of the most common attributes are:

- **name:** Specifies the name of the join column in the owning entity's table.
- **referencedColumnName:** Specifies the name of the column in the referenced entity's table.
- **insertable:** Whether the join column should be included in INSERT statements.
- **updatable:** Whether the join column should be included in UPDATE statements.
- **nullable:** Whether the join column can be null.
- **unique:** Whether the join column is a unique key.

From above example, the **@JoinColumn** annotation is used to define a foreign key column named **eid_aid** in the **emp_info** entity's table. This column will reference the primary key column (addressId) in the **emp_address** entity's table.

➢ Create JPA Repository's for Employye Entity Classes.

Repository Interface : EmployyeRepository.java

```
package com.dilip.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
```

```
import com.dilip.entity.Employye;

@Repository
public interface EmployyeRepository extends JpaRepository<Employye, Long>{

}
```

- Define a Service Class to perform Database Operations with Entity.

EmployyeService.java

```
package com.dilip.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.dilip.entity.Address;
import com.dilip.entity.Employye;
import com.dilip.repository.EmployyeRepository;

@Component
public class EmployyeService {

	@Autowired
	EmployyeRepository repository;

	public String addNewEmployye() {

		//Create Address Entity Object
		Address address = new Address();
		address.setCity("Hyderabad");
		address.setState("Telangana");
		address.setPincode(500072);

		//Create Employye Entity Object
		Employye employye = new Employye();
		employye.setName("Dilip Singh");
		employye.setAge(30);
		employye.setMobile("+91-8826111377");

		// Setting Address Entity Object inside Employye
		employye.setAddress(address);

		Employye resultEntity = repository.save(employye);
		return "Employye Data Submitted Successfully.
				Please Find Your Employye Id "+ resultEntity.getEmpId();
	}
}
```

➢ Execute above Service class methods to do Database Operations.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.service.EmployyeService;

@SpringBootApplication
public class SpringBootJpaMappingsApplication {

   public static void main(String[] args) {
         ApplicationContext context =
                  SpringApplication.run(SpringBootJpaMappingsApplication.class, args);

         EmployyeService service = context.getBean(EmployyeService.class);
         String result = service.addNewEmployye();
         System.out.println(result);
   }
}
```

Results:

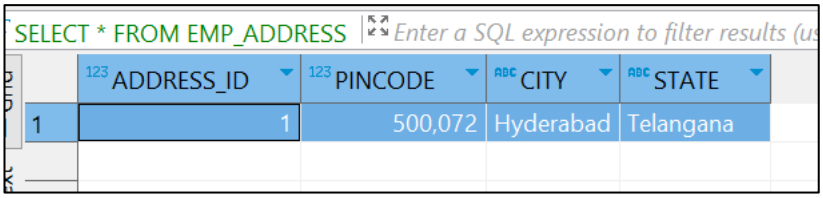
If we observe application logs, JPA created both Tables with Foreign Key Relationship.

```
Hibernate: create table emp_address (address_id number(10,0) generated as identity, pincode
number(10,0), city varchar2(255 char), state varchar2(255 char), primary key (address_id))
Hibernate: create table emp_info (age number(10,0), eid_aid number(10,0) unique, emp_id
number(19,0) generated as identity, mobile varchar2(255 char), name varchar2(255 char),
primary key (emp_id))
Hibernate: alter table emp_info add constraint FK6kyk4grokudd1we20ha2pnmkn foreign key
(eid_aid) references emp_address
2023-11-20T19:38:39.243+05:30  INFO 20036 --- [           main]
j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for
persistence unit 'default'
2023-11-20T19:38:39.601+05:30  INFO 20036 --- [           main]
c.d.SpringBootJpaMappingsApplication     : Started SpringBootJpaMappingsApplication in 4.05
seconds (process running for 4.689)
Hibernate: insert into emp_address (city,pincode,state,address_id) values (?,?,?,default)
Hibernate: insert into emp_info (eid_aid,age,mobile,name,emp_id) values (?,?,?,?,default)
Employye Data Submitted Successfully. Please Find Your Employye Id 1
```

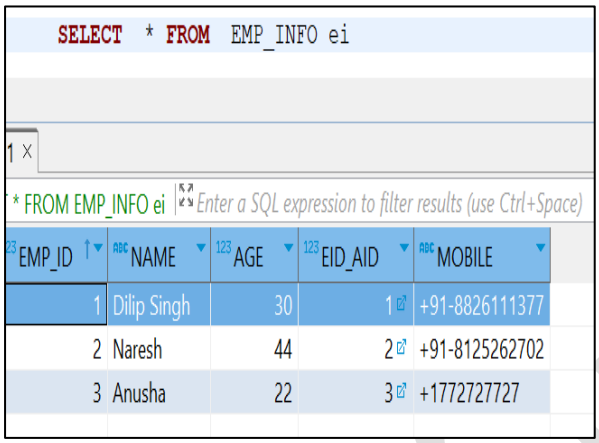
Data in Tables:

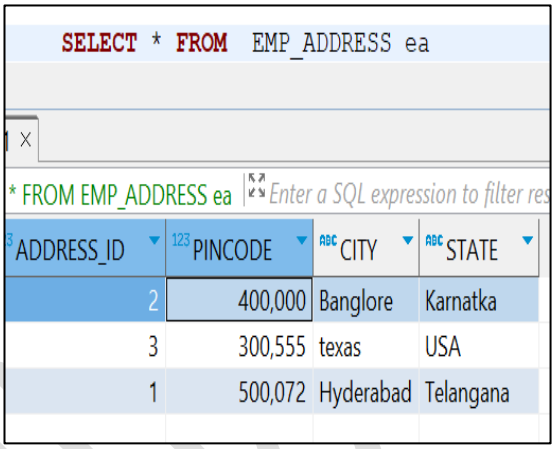
SELECT * FROM EMP_INFO — Enter a SQL expression to filter results (use Ctrl+Space)

	EMP_ID	NAME	MOBILE	AGE	EID_AID
1	1	Dilip Singh	+91-8826111377	30	1

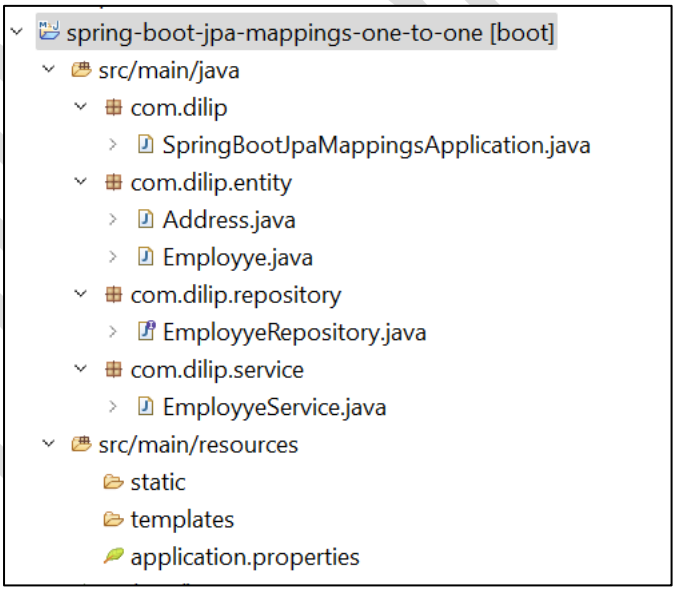


- Similarly Try to Insert Few More Records.





Project Directory Structure:



Delete Owner Entity Records:

We are enabled Cascade Propagation level as ALL i.e. @OneToOne(cascade = CascadeType.ALL), in Entity Relationship between Employee and Address Tables. So in such case, Whatever DB operations we are performing at Owner Entity **Employee** side, same action should be performed on target entity Address level.

Delete One Employee Details From Employee Table:

- Add a method in our Service class : **EmployyeService.java**

```
package com.dilip.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.dilip.repository.EmployyeRepository;

@Component
public class EmployyeService {

	@Autowired
	EmployyeRepository repository;

	//Passing only Employee  ID
	public void deleteEmployye(Long empId) {
		repository.deleteById(empId);
	}
}
```

- Execute Above Method From Main Method:

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.service.EmployyeService;

@SpringBootApplication
public class SpringBootJpaMappingsApplication {

  public static void main(String[] args) {
	ApplicationContext context =
		SpringApplication.run(SpringBootJpaMappingsApplication.class, args);
		EmployyeService service = context.getBean(EmployyeService.class);
		service.deleteEmployye(Long.valueOf(2));
  }
}
```

Results: From followed Console Logs, if we observe Employee details deleted and as well as associated Address Details also deleted from both tables. So Delete Operation cascaded from source to target table.

```
Hibernate: delete from emp_info where emp_id=?
Hibernate: delete from emp_address where address_id=?
```

Data in Tables : Employee **2 details deleted from both table.**

```
SELECT * FROM EMP_INFO ei
```

EMP_ID	NAME	MOBILE	AGE	EID_AID
1	Dilip Singh	+91-8826111377	30	1
3	Anusha	+1772727727	22	3

```
SELECT * FROM EMP_ADDRESS
```

ADDRESS_ID	PINCODE	CITY	STATE
1	500,072	Hyderabad	Telangana
3	300,555	texas	USA

This is how we can enable One to One relationship between tables by using Spring JPA.

One-to-Many (1:N) Relationship:

One-to-Many relationship represents a situation where one entity is associated with multiple instances of another entity. This is a common scenario in database design, where one record in a table is related to multiple records in another table. In JPA, we can model One-to-Many relationships using annotations.

One-to-many relationships are very common in databases and are often used to model real-world relationships. For example, a Employee can have many Addresses, but each Address only have one Employee. Or, a customer can have many orders, but each order can only belong to one customer. **One-to-many** relationships are often implemented in databases using foreign keys. A foreign key is a column in a table that references the primary key of another table.

Requirement: Define One to Many Relationship between Employee and Addresses.

- Define Entity Classes with One to Many Relationships.

Entity class : Employee.java

```
package com.dilip.entity;

import java.util.List;
import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
```

```
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_info")
public class Employee {

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

	@Column
	private String name;

	@Column
	private int age;

	@Column
	private String mobile;

	@OneToMany(cascade = CascadeType.ALL)
	@JoinColumn(name = "eid_aid")
	List<Address> address;

	public Long getEmpId() {
		return empId;
	}
	public void setEmpId(Long empId) {
		this.empId = empId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public String getMobile() {
		return mobile;
```

```
	}
	public void setMobile(String mobile) {
		this.mobile = mobile;
	}
	public List<Address> getAddress() {
		return address;
	}
	public void setAddress(List<Address> address) {
		this.address = address;
	}
}
```

- **@OneToMany** is used to define a One-to-Many relationship. The **cascade** attribute is used to specify operations that should be cascaded to the target of the association (e.g., if you delete an **Employee**, delete all associated **Address** entities).

Entity class : Address.java

```
package com.dilip.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_address")
public class Address {

	@Id
	@Column
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int addressId;

	@Column
	private String city;
	@Column
	private int pincode;
	@Column
	private String state;

	public int getAddressId() {
		return addressId;
```

```
	}
	public void setAddressId(int addressId) {
		this.addressId = addressId;
	}
	public String getCity() {
		return city;
	}
	public void setCity(String city) {
		this.city = city;
	}
	public int getPincode() {
		return pincode;
	}
	public void setPincode(int pincode) {
		this.pincode = pincode;
	}
	public String getState() {
		return state;
	}
	public void setState(String state) {
		this.state = state;
	}
}
```

- **Now Add More Addresses with One Employee Object.**

```
package com.dilip.service;

import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.dilip.entity.Address;
import com.dilip.entity.Employee;
import com.dilip.repository.EmployyeRepository;

@Service
public class EmployyeService {

	@Autowired
	EmployyeRepository repository;

	public String addNewEmployye() {

		List<Address> empAddresses = new ArrayList<>();
		Address home = new Address();
		home.setCity("HYDERBAD");
```

```
                home.setPincode(500072);
                home.setState("TELANGANA");
                empAddresses.add(home);

                Address office = new Address();
                office.setCity("BANGLORE");
                office.setPincode(400072);
                office.setState("KARNATAKA");
                empAddresses.add(office);

                Employee employye = new Employee();
                employye.setName("Dilip Singh");
                employye.setMobile("+918826111377");
                employye.setAge(30);
                employye.setAddress(empAddresses);

                employye = repository.save(employye);

                return "Employye Data Submitted Successfully. Please Find Employye
                                                        Id :" + employye.getEmpId();
        }

        public void deleteEmployye(String empId) {
                repository.deleteById(Long.valueOf(empId));
        }
}
```

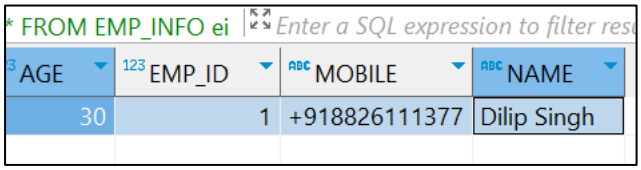
- In Above, we are adding 2 Address instance with one Employee Entity Instance.
- Execute the logic of **addNewEmployye()**

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.service.EmployyeService;

@SpringBootApplication
public class SbootJpaOneToManyMapping {

        public static void main(String[] args) {
        ApplicationContext context =
                        SpringApplication.run(SbootJpaOneToManyMapping.class, args);

        EmployyeService employyeService = context.getBean(EmployyeService.class);
```

```
        employyeService.addNewEmployye();

        }
}
```

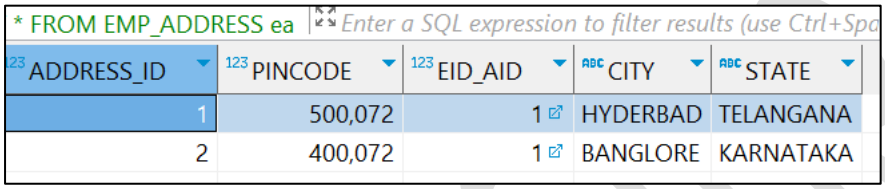
- If we observe Application console logs, both Employee and Address tables are created with Foreign Key relationship.

- One Record inside Employee and Two Records inside Address Tables are inserted.



* FROM EMP_INFO ei | Enter a SQL expression to filter res

AGE	EMP_ID	MOBILE	NAME
30	1	+918826111377	Dilip Singh



* FROM EMP_ADDRESS ea | Enter a SQL expression to filter results (use Ctrl+Spc

ADDRESS_ID	PINCODE	EID_AID	CITY	STATE
1	500,072	1	HYDERBAD	TELANGANA
2	400,072	1	BANGLORE	KARNATAKA

Many-to-One (N:1) Relationship:

The **@ManyToOne** mapping is used to represent a many-to-one relationship between entities in JPA Hibernate. It is used when multiple instances of one entity are associated with a single instance of another entity. Let's explain this mapping with a clear explanation and a code example.

Consider two entities: **Employee** and **Department**. Each employee belongs to a single department, but a department can have multiple employees. This scenario represents a many-to-one relationship, where **multiple employees (many)** can be associated with a **single department (one).**

Here's how you can implement the **@ManyToOne** mapping:

In the **Employee** entity, we have defined a department field with the **@ManyToOne** annotation. This indicates that **multiple employees can be associated with a single department**. The **@JoinColumn** annotation is used to specify the foreign key column in the **employees** table that references the **departments** table. In this case, the foreign key column is **dept_id.**

To understand the usage, let's consider an example:

Creating Employee Entity Class: Employee.java

```
package com.dilip.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_info")
public class Employee {

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

	@Column
	private String name;

	@Column
	private int age;

	@Column
	private String mobile;

	@ManyToOne
	@JoinColumn(name = "dept_id")
	Department department;

	public Long getEmpId() {
		return empId;
	}
	public void setEmpId(Long empId) {
		this.empId = empId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
```

```
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public String getMobile() {
		return mobile;
	}
	public void setMobile(String mobile) {
		this.mobile = mobile;
	}
	public Department getDepartment() {
		return department;
	}
	public void setDepartment(Department department) {
		this.department = department;
	}
}
```

Creating Department Entity Class: Department.java

```
package com.dilip.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "departments")
public class Department {

	@Id
	private Long id;

	private String name;

	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
```

```
	public void setName(String name) {
		this.name = name;
	}
}
```

- Now Create JPA Repositories for both Employee and Department Entity's .

```
package com.dilip.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.dilip.entity.Employee;

@Repository
public interface EmployyeRepository extends JpaRepository<Employee, Long> {

}
```

```
package com.dilip.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.dilip.entity.Department;

@Repository
public interface DepartmentRepository extends JpaRepository<Department, Long> {

}
```

- Now Create Data like More Employees of same Department.

```
package com.dilip.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.dilip.entity.Department;
import com.dilip.entity.Employee;
import com.dilip.repository.DepartmentRepository;
import com.dilip.repository.EmployyeRepository;

@Service
public class EmployyeService {

	@Autowired
	EmployyeRepository employyeRepository;
```

```
	@Autowired
	DepartmentRepository departmentRepository;

	public void addNewEmployye() {

		Department department = new Department();
		department.setId(1L);
		department.setName("HR");

		Employee employee1 = new Employee();
		employee1.setMobile("+918826111377");
		employee1.setName("Dilip Singh");
		employee1.setAge(30);
		employee1.setDepartment(department);

		Employee employee2 = new Employee();
		employee2.setName("Naresh");
		employee2.setMobile("+918125262702");
		employee2.setAge(31);
		employee2.setDepartment(department);

		departmentRepository.save(department);
		employyeRepository.save(employee1);
		employyeRepository.save(employee2);

	}
}
```

In this example, we create a **Department** object representing the HR department. Then, we create two **Employee** objects and associate them with the HR department using the **setDepartment()** method. When you persist these entities, it will automatically handle the foreign key relationship between the **employees** and **departments** tables. The **dept_id** column in the employees table will store the appropriate department ID.

This way, you can establish a many-to-one relationship between entities using the @ManyToOne mapping in JPA Hibernate.

- Now Execute Above Logic.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.service.EmployyeService;
```

```
@SpringBootApplication
public class SpringJpaApplication {
	public static void main(String[] args) {
		ApplicationContext context =
			SpringApplication.run(SpringJpaApplication.class, args);
		EmployyeService employyeService =
				context.getBean(EmployyeService.class);
		employyeService.addNewEmployye();
	}
}
```

- Now Observe Console logs and see how tables created and data getting inserted.

```
Hibernate: create table departments (id number(19,0) not null, name varchar2(255 char), primary key (id))
Hibernate: create table emp_info (age number(10,0), dept_id number(19,0), emp_id number(19,0) generated as identity,
mobile varchar2(255 char), name varchar2(255 char), primary key (emp_id))
Hibernate: alter table emp_info add constraint FKci9jim8lqk0nt9dkaisvik4mv foreign key (dept_id) references departments
2023-12-29T18:54:50.860+05:30  INFO 19896 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized
JPA EntityManagerFactory for persistence unit 'default'
2023-12-29T18:54:51.187+05:30  INFO 19896 --- [           main] com.dilip.SpringJpaNotesApplication      : Started
SpringJpaNotesApplication in 4.103 seconds (process running for 4.714)
Hibernate: select d1_0.id,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into departments (name,id) values (?,?)
Hibernate: select null,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into emp_info (age,dept_id,mobile,name,emp_id) values (?,?,?,?,default)
Hibernate: select null,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into emp_info (age,dept_id,mobile,name,emp_id) values (?,?,?,?,default)
```

- Now Try to insert data of employees with invalid department id i.e. department id is not existed in Department table. Then we will get an exception and data will not be inserted in employee table i.e. when Department Id is existed then only employee data will be inserted because of foreign key relationship.

Many-to-Many (N:N) Relationship:

A many-to-many relationship in the context of databases refers to a relationship between two entities where each record in one entity can be related to multiple records in another entity, and vice versa. This type of relationship is common in relational database design and is typically implemented using an intermediary table, often called a junction or linking table. Here's a simple example to illustrate a many-to-many relationship:

Let's consider two entities: "**Employee**" and "**Role**"

Employee
- **Id (Primary Key)**
- **name**
- **email**

Role:

Id (Primary Key)
name

In a many-to-many relationship, a **Employee** can have multiple **Roles**, and a **Role** can have multiple **Employees**. To represent this relationship, you would introduce a third table, often referred to as a junction table or linking table.

Junction Table: employee_roles

employee_id (Foreign Key referencing Students)
role_id (Foreign Key referencing Courses)

Each record in the **employee_roles** table represents a connection between a **Employee** and a **Role**. The combination of Employee ID and Role ID in the Enrolment table creates a unique identifier for each mapping, preventing duplicate entries for the same **Employee-Role** pair.

This setup allows for flexibility and efficiency in querying the database. You can easily find all Roles of an Employee is enrolled or all Employs enrolled in a particular Role by querying the : **employee_roles** table.

In summary, many-to-many relationships are handled by introducing a linking table to manage the associations between entities. This linking table resolves the complexity of directly connecting entities with a many-to-many relationship in a relational database.

- Define **Employee** and **Role** Entity classes as per above solution.

Employee Entity: Employee.java

```
package com.tek.teacher.employye;

import java.util.List;
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.Table;

@Entity
@Table(name = "employees")
public class Employee {

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

```
	Long id;

	String name;
	String email;

	@ManyToMany(cascade = CascadeType.ALL)
	@JoinTable(name = "employee_roles",
			joinColumns = @JoinColumn(name = "employee_id"),
			inverseJoinColumns = @JoinColumn(name = "role_id"))
	List<Role> roles;

	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public List<Role> getRoles() {
		return roles;
	}
	public void setRoles(List<Role> roles) {
		this.roles = roles;
	}
}
```

Role Entity: Role.java

```
package com.tek.teacher.employye;

import java.util.List;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToMany;
```

```
import jakarta.persistence.Table;

@Entity
@Table(name = "roles")
public class Role {

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

	String name;

	@ManyToMany
	List<Employee> employees;

	public List<Employee> getEmployees() {
		return employees;
	}
	public void setEmployees(List<Employee> employees) {
		this.employees = employees;
	}
	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
}
```

- **Create JPA Repository :** EmployyeRepository.java

```
package com.tek.teacher.employye;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface EmployyeRepository extends JpaRepository<Employee, Long>{

}
```

- Now Create a class and Persist Data inside tables.

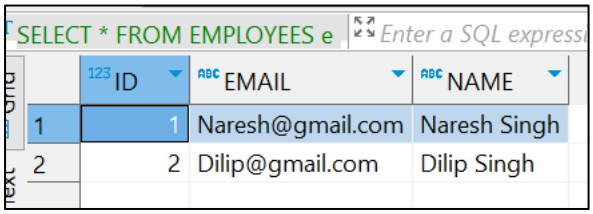
```
package com.tek.teacher.employye;

import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class EmployyeOperations {

	@Autowired
	EmployyeRepository repository;

	public void addEmployye() {
		Employee e = new Employee();
		e.setEmail("Naresh@gmail.com");
		e.setName("Naresh Singh");

		Role r1 = new Role();
		r1.setName("DEVLOPER");
		Role r2 = new Role();
		r2.setName("USER");

		List<Role> roles = new ArrayList<Role>();
		roles.add(r1);
		roles.add(r2);
		e.setRoles(roles);
		repository.save(e);
	}
}
```

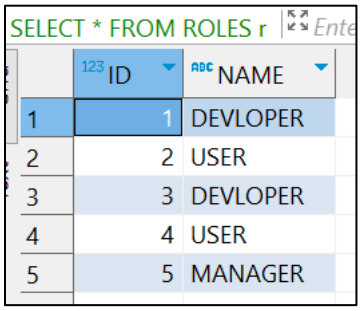
- Now Execute Above Logic

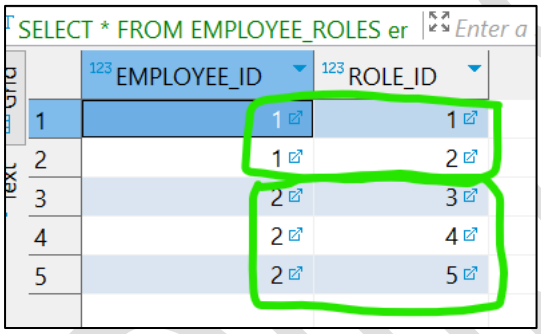
```
Hibernate: create table employee_roles (employee_id number(19,0) not null, role_id number(19,0) not null)
Hibernate: create table employees (id number(19,0) generated as identity, email varchar2(255 char), name
varchar2(255 char), primary key (id))
Hibernate: create table roles (id number(19,0) generated as identity, name varchar2(255 char), primary
key (id))
Hibernate: alter table employee_roles add constraint FK398vvu81xw154mvy3g2eytscn foreign key (role_id)
references roles
Hibernate: alter table employee_roles add constraint FK3uwwaxeiucvfixgd45etkjgmg foreign key
(employee_id) references employees
2024-01-02T18:52:17.969+05:30  INFO 17312 --- [           main] j.LocalContainerEntityManagerFactoryBean
: Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-01-02T18:52:18.378+05:30  INFO 17312 --- [           main] t.SpringBootJpaGeneartedvalueApplication
: Started SpringBootJpaGeneartedvalueApplication in 4.375 seconds (process running for 4.819)
Hibernate: insert into employees (email,name,id) values (?,?,default)
Hibernate: insert into roles (name,id) values (?,default)
Hibernate: insert into roles (name,id) values (?,default)
Hibernate: insert into employee_roles (employee_id,role_id) values (?,?)
Hibernate: insert into employee_roles (employee_id,role_id) values (?,?)
```

- From the above Console Logs, if we observe a joining Table is created along with Employee and Roles. Data is inserted in total 3 tables and **employee_roles** table maintaining the relation between **Employees** and **Roles** of **Many to Many** Association.

Database Data of 3 Tables:







Spring Framework JPA Project Configuration:

- For using Spring Data JPA, first of all we have to configure DataSource bean. Then we need to configure **LocalContainerEntityManagerFactoryBean** bean.
- The next step is to configure bean for transaction management. In our example it's **JpaTransactionManager**.
- **@EnableTransactionManagement**: This annotation allows users to use transaction management in application.
- **@EnableJpaRepositories**("com.flipkart.*")**:** indicates where the repositories classes are present.

Configuring the DataSource Bean:

- Configure the database connection. We need to set the name of the the JDBC url, the username of database user, and the password of the database user.

Configuring the Entity Manager Factory Bean:

We can configure the entity manager factory bean by following steps:

- Create a new **LocalContainerEntityManagerFactoryBean** object. We need to create this object because it creates the JPA **EntityManagerFactory**.
- Configure the Created DataSource in Previous Step.
- Configure the Hibernate specific implementation of the **HibernateJpaVendorAdapter** . It will initialize our configuration with the default settings that are compatible with Hibernate.
- Configure the packages that are scanned for entity classes.
- Configure the JPA/Hibernate properties that are used to provide additional configuration to the used JPA provider.

Configuring the Transaction Manager Bean:

Because we are using JPA, we have to create a transaction manager bean that integrates the JPA provider with the Spring transaction mechanism. We can do this by using the **JpaTransactionManager** class as the transaction manager of our application.

We can configure the transaction manager bean by following steps:

- Create a new JpaTransactionManager object.
- Configure the entity manager factory whose transactions are managed by the created JpaTransactionManager object.

```
package flipkart.entity;

import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
```

```
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;

@Configuration
@EnableJpaRepositories("flipkart.*")
public class SpringJpaConfiguration {

	//DB Details
	@Bean
	public DataSource getDataSource() {
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
		dataSource.setUsername("c##dilip");
		dataSource.setPassword("dilip");
		return dataSource;
	}

	@Bean("entityManagerFactory")
	LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {
		LocalContainerEntityManagerFactoryBean factory
				= new LocalContainerEntityManagerFactoryBean();

		// 1. Setting Datasource Object // DB details
		factory.setDataSource(getDataSource());
		// 2. Provide package information of entity classes
		factory.setPackagesToScan("flipkart.*");
		// 3. Providing Hibernate Properties to EM
		factory.setJpaProperties(hibernateProperties());
		// 4. Passing Predefined Hiberante Adaptor Object EM
		HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
		factory.setJpaVendorAdapter(adapter);

		return factory;
	}

	@Bean("transactionManager")
	public PlatformTransactionManager createTransactionManager() {
	JpaTransactionManager transactionManager =  new JpaTransactionManager();
	transactionManager.setEntityManagerFactory(createEntityManagerFactory()
	                                                    .getObject());
		return transactionManager;
	}

	// these are all from hibernate FW , Predefined properties : Keys
	Properties hibernateProperties() {

		Properties hibernateProperties = new Properties();
```

```
                hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "create");

                //This is for printing internally genearted SQL Quries
                hibernateProperties.setProperty("hibernate.show_sql", "true");
                return hibernateProperties;
        }
}
```

Now create another Component class For Performing DB operations as per our Requirement:

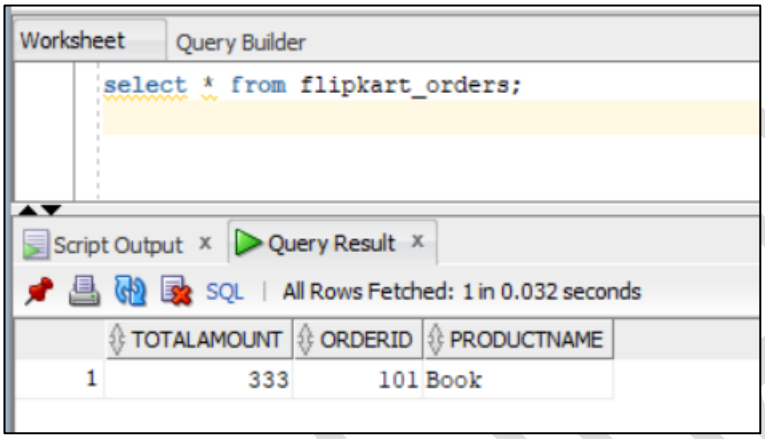
```
package flipkart.entity;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

// To Execute/Perform DB operations
@Component
public class OrderDbOperations {

        @Autowired
        FlipakartOrderRepository flipakartOrderRepository;

        public void addOrderDetaisl(FlipakartOrder order) {
                flipakartOrderRepository.save(order);
        }
}
```

Now Create a Main method class for creating Spring Application Context for loading all Configurations and Component classes.

```
package flipkart.entity;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

        public static void main(String[] args) {

                        AnnotationConfigApplicationContext context =
                                        new AnnotationConfigApplicationContext();
                        context.scan("flipkart.*");
                        context.refresh();

                        // Created Entity Object
                        FlipakartOrder order = new FlipakartOrder();
                        order.setOrderID(9988);
                        order.setProductName("Book");
```

```
                order.setTotalAmount(333.00);

                // Pass Entity Object to Repository MEthod
        OrderDbOperations dbOperation =   context.getBean(OrderDbOperations.class);
                dbOperation.addOrderDetaisl(order);
        }
}
```

Execute The Programme Now. Verify In Database Now.



In Eclipse Console Logs, Printed internally generated SQL Quries to perform insert operation.

NOTE: In our example, we are nowhere written any SQL query to do Database operation.

Internally, Based on Repository method **save()** of flipakartOrderRepository.save(order), JPA internally generates implementation code, SQL queries and will be executed internally.

Similarly, we have many predefined methods of Spring repository to do CRUD operations.

We learned to configure the persistence layer of a Spring application that uses Spring Data JPA and Hibernate. Let's create few more examples to do CRUD operations on Db table. From Spring JPA Configuration class, we have used two properties related to Hibernate FW**.**

hibernate.hbm2ddl.auto: The `**hibernate.hbm2ddl.auto**` property is used to configure the automatic schema/tables generation and management behaviour of Hibernate. This property allows you to control how Hibernate handles the database schema based on your entity classes.

Here are the possible values for the **hibernate.hbm2ddl.auto** property:

none: No action is performed. The schema will not be generated.

validate: The database schema will be validated using the entity mappings. This means that Hibernate will check to see if the database schema matches the entity mappings. If there are any differences, Hibernate will throw an exception.

update: The database schema will be updated by comparing the existing database schema with the entity mappings. This means that Hibernate will create or modify tables in the database as needed to match the entity mappings.

create: The database schema will be created. This means that Hibernate will create all of the tables needed for the entity mappings.

create-drop: The database schema will be created and then dropped when the SessionFactory is closed. This means that Hibernate will create all of the tables needed for the entity mappings, and then drop them when the SessionFactory is closed.

Note: In Spring Boot, we are using property **`spring.jpa.hibernate.ddl-auto`** with same values for same functionalities as we discussed.

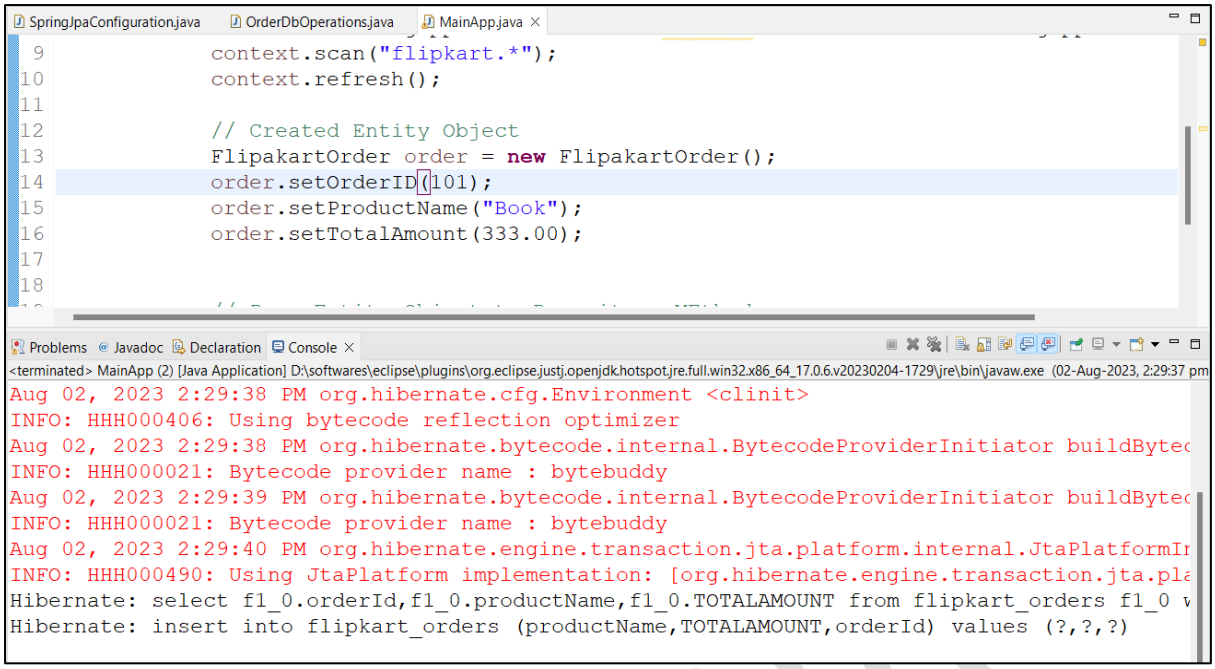
hibernate.show_sql: The **hibernate.show_sql** property is a Hibernate configuration property that controls whether or not Hibernate will log the SQL statements that it generates. The possible values for this property are:

true: Hibernate will log all SQL statements to the console.
false: Hibernate will not log any SQL statements.

The default value for this property is **false**. This means that Hibernate will not log any SQL statements by default. If you want to see the SQL statements that Hibernate is generating, you will need to set this property to **true**.

Logging SQL statements can be useful for debugging purposes. If you are having problems with your application, you can enable logging and see what SQL statements Hibernate is generating. This can help you to identify the source of the problem.

Note: In Spring Boot, we are using property **`spring.jpa.show-sql`** with same values for same functionalities as we discussed.



Creation of New Spring JPA Project:

Requirement : Patient Information

- Name
- Age
- Gender
- Contact Number
- Email Id

Requirements:

- Add Single Patient Details
- Add More Than One Patient Details
- Update Patient Details
- Select Single Patient Details
- Select More Patient Details
- Delete Patient Details

1. Create Simple Maven Project and Add Required Dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.dilip</groupId>
        <artifactId>spring-jpa-two</artifactId>
        <version>0.0.1-SNAPSHOT</version>
```

```
        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <version>6.0.11</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>6.0.11</version>
                </dependency>
                <dependency>
                        <groupId>com.oracle.database.jdbc</groupId>
                        <artifactId>ojdbc8</artifactId>
                        <version>21.9.0.0</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-orm</artifactId>
                        <version>6.0.11</version>
                </dependency>
                <dependency>
                        <groupId>org.hibernate</groupId>
                        <artifactId>hibernate-core</artifactId>
                        <version>6.2.6.Final</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.data</groupId>
                        <artifactId>spring-data-jpa</artifactId>
                        <version>3.1.2</version>
                </dependency>
        </dependencies>
</project>
```

2. Now Create Spring JPA Configuration

```
package com.dilip;

import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;

@Configuration
@EnableJpaRepositories("com.*")
```

```
public class SpringJpaConfiguration {

	//DB Details
	@Bean
	public DataSource getDataSource() {
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
		dataSource.setUsername("c##dilip");
		dataSource.setPassword("dilip");
		return dataSource;
	}

	@Bean("entityManagerFactory")
	LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {
LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();

		// 1. Setting Datasource Object // DB details
		factory.setDataSource(getDataSource());

		// 2. Provide package information of entity classes
		factory.setPackagesToScan("com.*");

		// 3. Providing Hibernate Properties to EM
		factory.setJpaProperties(hibernateProperties());

		// 4. Passing Predefined Hiberante Adaptor Object EM
		HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
		factory.setJpaVendorAdapter(adapter);

		return factory;
	}

	//Spring JPA: configuring data based on your project req.
	@Bean("transactionManager")
	public PlatformTransactionManager createTransactionManager() {
	JpaTransactionManager transactionManager = new JpaTransactionManager();
	transactionManager.setEntityManagerFactory(createEntityManagerFactory().getObject());
		return transactionManager;
	}

	// these are all from hibernate FW , Predefined properties : Keys
	Properties hibernateProperties() {
		Properties hibernateProperties = new Properties();
		hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");
		//This is for printing internally genearted SQL Quries
		hibernateProperties.setProperty("hibernate.show_sql", "true");
		return hibernateProperties;
	}

}
```

Note: Until this Step/Configuration in Spring Framework, we have written manually of JPA configuration. From here onwards, Rest of functionalities implementations are as it is like how we implemented in Spring Boot.

Means, The above 2 Steps are automated/ auto configured in Spring Boot internally. We just need to provide database details and JPA properties inside **application.properties.**

3. Create Entity Class

NOTE : Configured **hibernate.hbm2ddl.auto** value as **update**. So Table Creation will be done by JPA internally.

```
package com.dilip.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table
public class Patient {

	@Id
	@Column
	private String emailId;

	@Column
	private String name;

	@Column
	private int age;

	@Column
	private String gender;

	@Column
	private String contact;

	public Patient() {
		super();
		// TODO Auto-generated constructor stub
	}

	public Patient(String name, int age, String gender, String contact, String emailId) {
		super();
		this.name = name;
		this.age = age;
```

```
		this.gender = gender;
		this.contact = contact;
		this.emailId = emailId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public String getGender() {
		return gender;
	}
	public void setGender(String gender) {
		this.gender = gender;
	}
	public String getContact() {
		return contact;
	}
	public void setContact(String contact) {
		this.contact = contact;
	}
	public String getEmailId() {
		return emailId;
	}
	public void setEmailId(String emailId) {
		this.emailId = emailId;
	}

	@Override
	public String toString() {
		return "Patient [name=" + name + ", age=" + age + ", gender=" + gender + ",
contact=" + contact + ", emailId="
				+ emailId + "]";
	}
}
```

4. Create A Repository Now

```
@Component
public interface PatientRepository extends CrudRepository<Patient, String> {

}
```

5. Create a class for DB operations

```
@Component
public class PatientOperations {
	@Autowired
	PatientRepository repository;
}
```

Spring JPA Repositories Provided many predefined abstract methods for all DB CURD operations. We should recognize those as per our DB operation.

Requirement : Add Single Patient Details
Here, we are adding Patient details means at Database level this is insert Query Operation.

save() : Used for insertion of Details. We should pass Entity Object.

Add Below Method in PatientOperations.java:

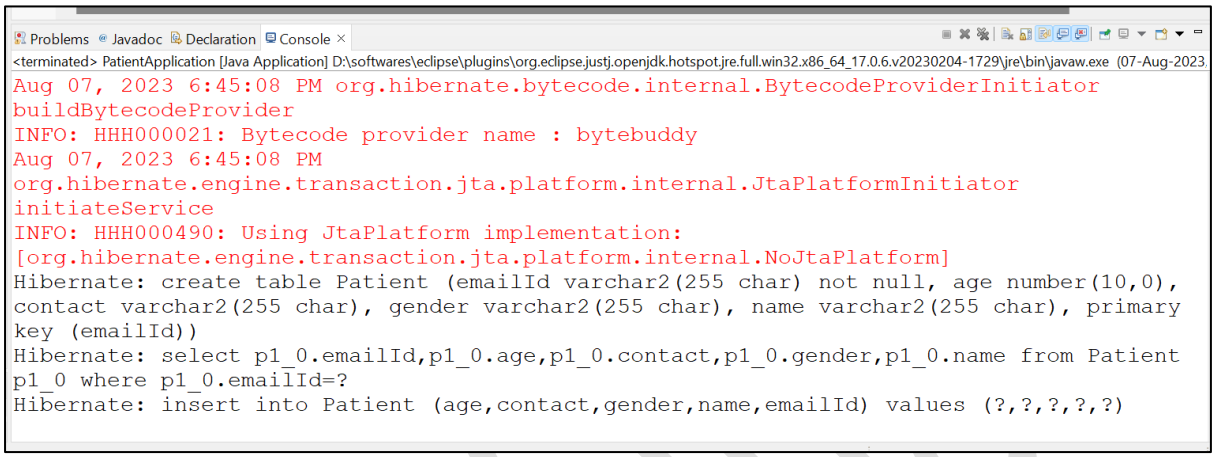
```
public void addPatient(Patient p) {
		repository.save(p);
}
```

Now Test it : Create Main method class

```
package com.dilip.operations;

import java.util.ArrayList;
import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.dilip.entity.Patient;

public class PatientApplication {

	public static void main(String[] args) {

		AnnotationConfigApplicationContext context =
					new AnnotationConfigApplicationContext();
		context.scan("com.*");
		context.refresh();

		PatientOperations ops = context.getBean(PatientOperations.class);

		// Add Single Patient
		Patient p = new Patient();
		p.setEmailId("one@gmail.com");
		p.setName("One Singh");
		p.setContact("+918826111377");
```

```
		p.setAge(30);
		p.setGender("MALE");

		ops.addPatient(p);
}
}
```

Now Execute It. Table also created by JPA module and One Record is inserted.



Requirement : **Add multiple Patient Details at a time**

Here, we are adding Multiple Patient details means at Database level this is also insert Query Operation.

saveAll() : This is for adding List of Objects at a time. We should pass List Object of Patient Type.

Add Below Method in PatientOperations.java:

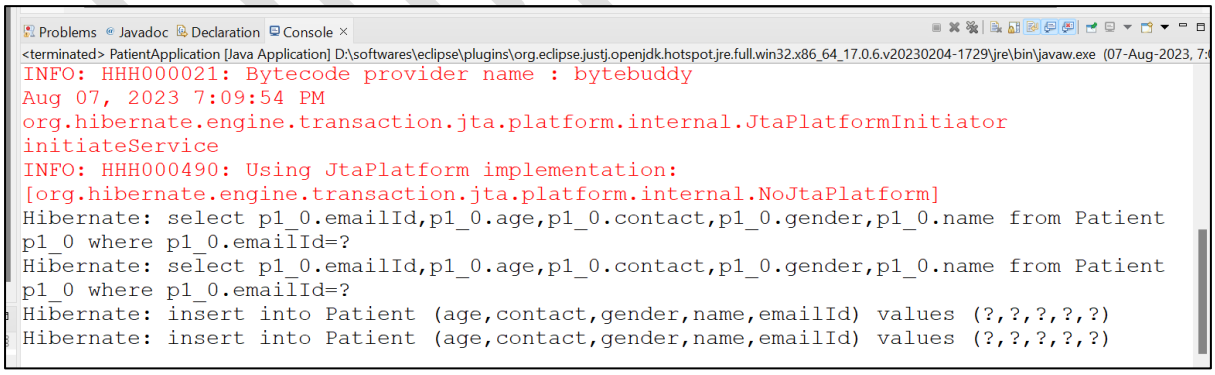
```
public void addMorePatients(List<Patient> patients) {
		repository.saveAll(patients);
}
```

Now Test it : Inside Main method class, add Logic below.

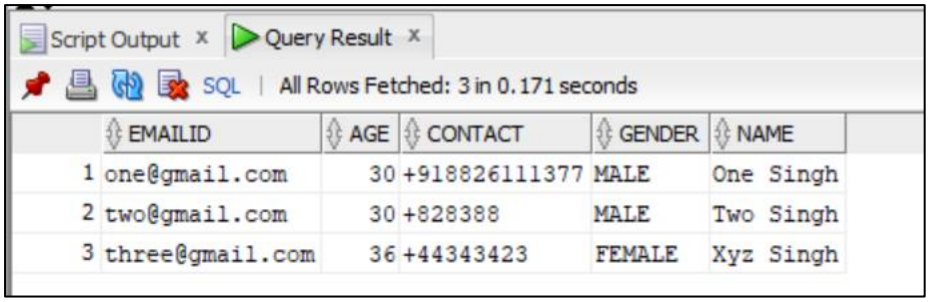
```
package com.dilip.operations;

import java.util.ArrayList;
import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {

	public static void main(String[] args) {
```

```
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
                context.scan("com.*");
                context.refresh();

                PatientOperations ops = context.getBean(PatientOperations.class);

                // Adding More Patients
                Patient p1 = new Patient();
                p1.setEmailId("two@gmail.com");
                p1.setName("Two Singh");
                p1.setContact("+828388");
                p1.setAge(30);
                p1.setGender("MALE");

                Patient p2 = new Patient();
                p2.setEmailId("three@gmail.com");
                p2.setName("Xyz Singh");
                p2.setContact("+44343423");
                p2.setAge(36);
                p2.setGender("FEMALE");

                List<Patient> allPatients = new ArrayList<>();
                allPatients.add(p1);
                allPatients.add(p2);
                ops.addMorePatients(allPatients);
        }
}
```

Execution Output:



Verify In DB Table:



	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	30	+918826111377	MALE	One Singh
2	two@gmail.com	30	+828388	MALE	Two Singh
3	three@gmail.com	36	+44343423	FEMALE	Xyz Singh

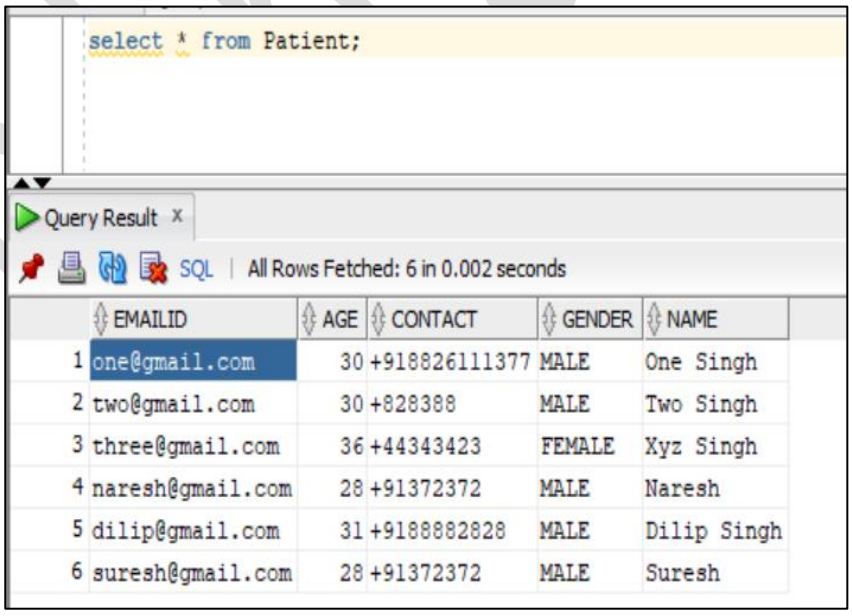
Requirement : **Update Patient Details**

In Spring Data JPA, the **save()** method is commonly used for both **insert** and **update** operations. When you call the **save()** method on a repository, Spring Data JPA checks whether the entity you're trying to save already exists in the database. If it does, it updates the existing entity; otherwise, it inserts a new entity.

So that is the reason we are seeing a select query execution before inserting data in previous example. After select query execution with primary key column JPA checks row count and if it is 1, then JPA will convert entity as insert operation. If count is 0 , then Spring JPA will convert entity as update operation specific to Primary key.

Using the **save()** method for updates is a common and convenient approach, especially when we want to leverage Spring Data JPA's automatic change tracking and transaction management.

Requirement: Please update name as Dilip Singh for email id: one@gmail.com

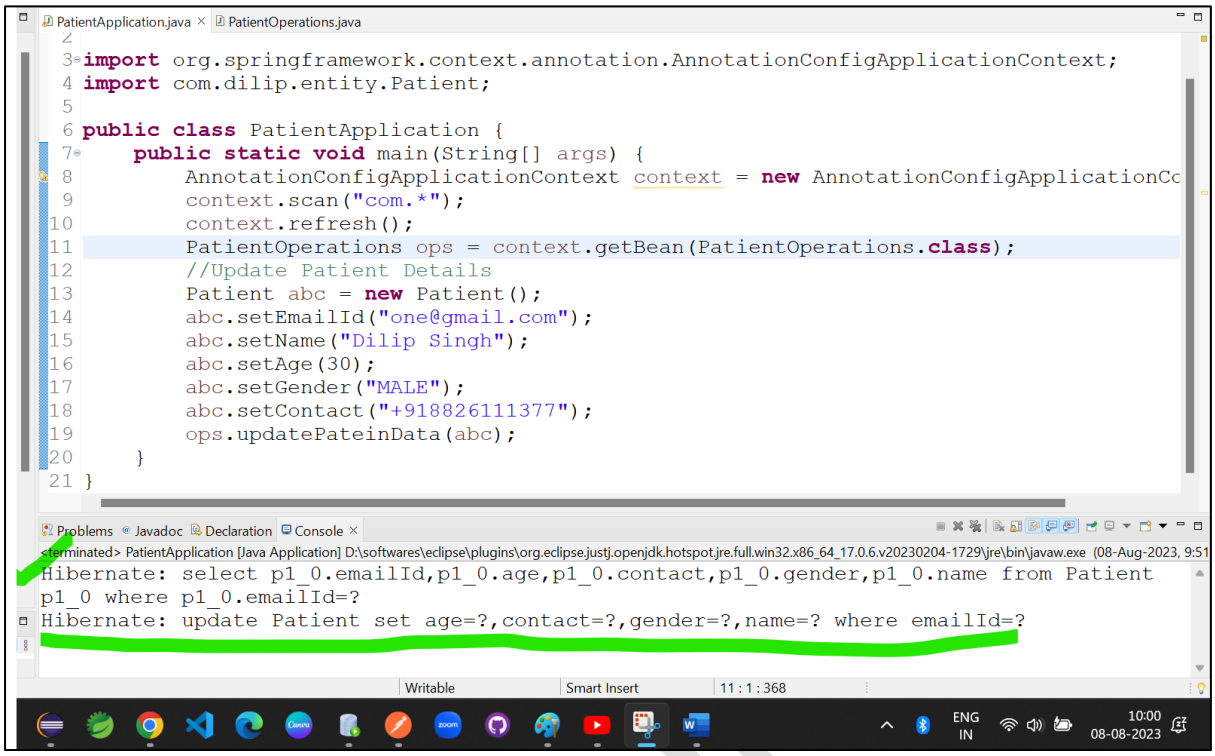


	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	30	+918826111377	MALE	One Singh
2	two@gmail.com	30	+828388	MALE	Two Singh
3	three@gmail.com	36	+44343423	FEMALE	Xyz Singh
4	naresh@gmail.com	28	+91372372	MALE	Naresh
5	dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
6	suresh@gmail.com	28	+91372372	MALE	Suresh

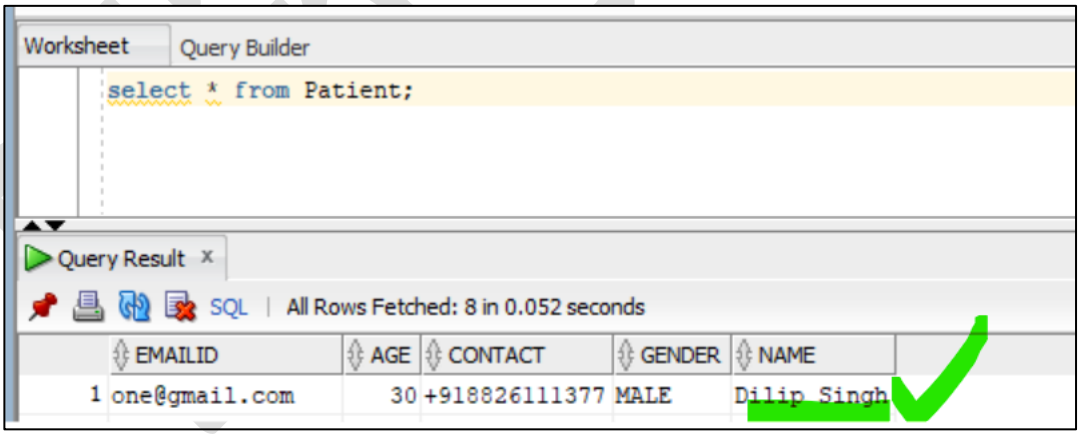
Add Below Method in PatientOperations.java:

```
public void updatePateinData(Patient p) {
		repository.save(p);
}
```

Now Test it from Main class: In below if we observe, first select query executed by JPA as per our entity Object, JPA found data so JPA decided for update Query execution.



Verify In DB :



Requirement: Deleting Patient Details based on Email ID.

Spring JPA provided a predefined method **deleteById()** for primary key columns delete operations.

deleteById():

The **deleteById()** method in Spring Data JPA is used to remove an entity from the database based on its primary key (ID). It's provided by the **JpaRepository** interface and allows you to delete a single entity by its unique identifier.

Here's how you can use the **deleteById()** method in a Spring Data JPA repository:

Add Below Method in PatientOperations.java:

```
public void deletePatient(String email) {
	repository.deleteById(email);
}
```

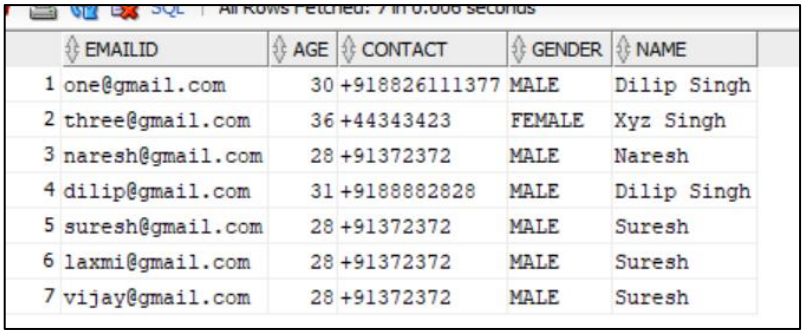
Testing from Main Class:

```
package com.dilip.operations;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class PatientApplication {
	public static void main(String[] args) {

		 AnnotationConfigApplicationContext context =
							new AnnotationConfigApplicationContext();
		context.scan("com.*");
		context.refresh();

		PatientOperations ops = context.getBean(PatientOperations.class);
		//Delete Patient Details
		ops.deletePateintWithEmailID("two@gmail.com");
	}
}
```

Before Execution/Deletion:

	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	30	+918826111377	MALE	Dilip Singh
2	two@gmail.com	30	+828388	MALE	Two Singh
3	three@gmail.com	36	+44343423	FEMALE	Xyz Singh
4	naresh@gmail.com	28	+91372372	MALE	Naresh
5	dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
6	suresh@gmail.com	28	+91372372	MALE	Suresh
7	laxmi@gmail.com	28	+91372372	MALE	Suresh
8	vijay@gmail.com	28	+91372372	MALE	Suresh

After Execution/Deletion:



	EMAILID	AGE	CONTACT	GENDER	NAME
1	one@gmail.com	30	+918826111377	MALE	Dilip Singh
2	three@gmail.com	36	+44343423	FEMALE	Xyz Singh
3	naresh@gmail.com	28	+91372372	MALE	Naresh
4	dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
5	suresh@gmail.com	28	+91372372	MALE	Suresh
6	laxmi@gmail.com	28	+91372372	MALE	Suresh
7	vijay@gmail.com	28	+91372372	MALE	Suresh

Requirement: **Get Patient Details Based on Email Id.**

Here Email Id is Primary key Column. Finding Details based on Primary key column name Spring JPA provide a method **findById().**

Add Below Method in PatientOperations.java:

```
public Patient fetchByEmailId(String emailId) {
	return repository.findById(emailId).get();
}
```

Testing from Main Class:

```
package com.dilip.operations;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {

	public static void main(String[] args) {
		AnnotationConfigApplicationContext context =
							new AnnotationConfigApplicationContext();
		context.scan("com.*");
		context.refresh();

		PatientOperations ops = context.getBean(PatientOperations.class);
		//Fetch Patient Details By Email ID
		Patient patient = ops.fetchByEmailId("dilip@gmail.com");
		System.out.println(patient);
	}
}
```

Output:

- Remaining all functionalities are similar like we discussed in Spring BOOT.