Spring JPA Module:

Spring Framework JPA is a powerful abstraction layer that makes it easy to interact with relational databases in Java applications. It provides a simple and consistent API for performing CRUD operations, as well as more advanced features such as pagination, auditing, and querying.

Spring Framework JPA is built on top of the Java Persistence API (JPA), which is a standard API for accessing relational databases. This means that Spring Framework JPA is compatible with a wide range of databases, including MySQL, PostgreSQL, Oracle, and SQL Server.

What is JPA?

JPA(Jakarta Persistence API) is just a specification that facilitates object-relational mapping to manage relational data in Java applications. It provides a platform to work directly with objects instead of using SQL statements.

Jakarta Persistence API (JPA; formerly Java Persistence API) is a Jakarta EE application programming interface specification that describes the management of relational data in enterprise Java applications. The API itself, defined in the **jakarta.persistence** package (**javax.persistence for Jakarta EE 8 and below**). The Jakarta Persistence Query Language (JPQL; formerly Java Persistence Query Language)

History:

- JPA 1.0 specification was 11 May 2006 as part of Java Community.
- JPA 2.0 specification was released 10 December 2009
- JPA 2.1 specification was released 22 April 2013
- JPA 2.2 specification was released in the summer of 2017.
- JPA 3.1 specification, the latest version, was released in the spring of 2022 **as part of Jakarta EE 10**

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.

If you are looking for a powerful and easy-to-use way to interact with relational databases in Java applications, then Spring Framework JPA is a great option.

Here are some of the benefits of using Spring Framework JPA:

Simple and consistent API: Spring Framework JPA provides a simple and consistent API for performing CRUD operations, as well as more advanced features such as pagination, auditing, and querying.

Compatibility with a wide range of databases: Spring Framework JPA is compatible with a wide range of databases, including MySQL, PostgreSQL, Oracle, and SQL Server.

Ease of use: Spring Framework JPA is easy to use, even for developers who are not familiar with JPA.

Performance: Spring Framework JPA is designed for performance, and it can provide significant performance improvements over traditional JDBC code.

If you are looking for a powerful and easy-to-use way to interact with relational databases in Java applications, then Spring Framework JPA is a great option.

Here are the steps involved in using Spring JPA in our application:

- Create entity classes that represent the data that you want to store in the database.
- Annotate the entity classes with JPA annotations.
- Create a Spring Data repository interface that extends one of the Spring Data repository interfaces, such as JpaRepository or CrudRepository.
- Inject the repository interface into your application code.
- Use the repository interface to access the data in the database.

Spring JPA Module Project Example:

Requirement: CURD Operations with **Order** table

NOTE: We are integrating Hibernate ORM framework implantation of JPA in our Spring JPA module.

Project Setup:

- Create Maven Project
- Add required Dependencies for Spring JPA Modules in **pom.xml** file.

```
<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>
```

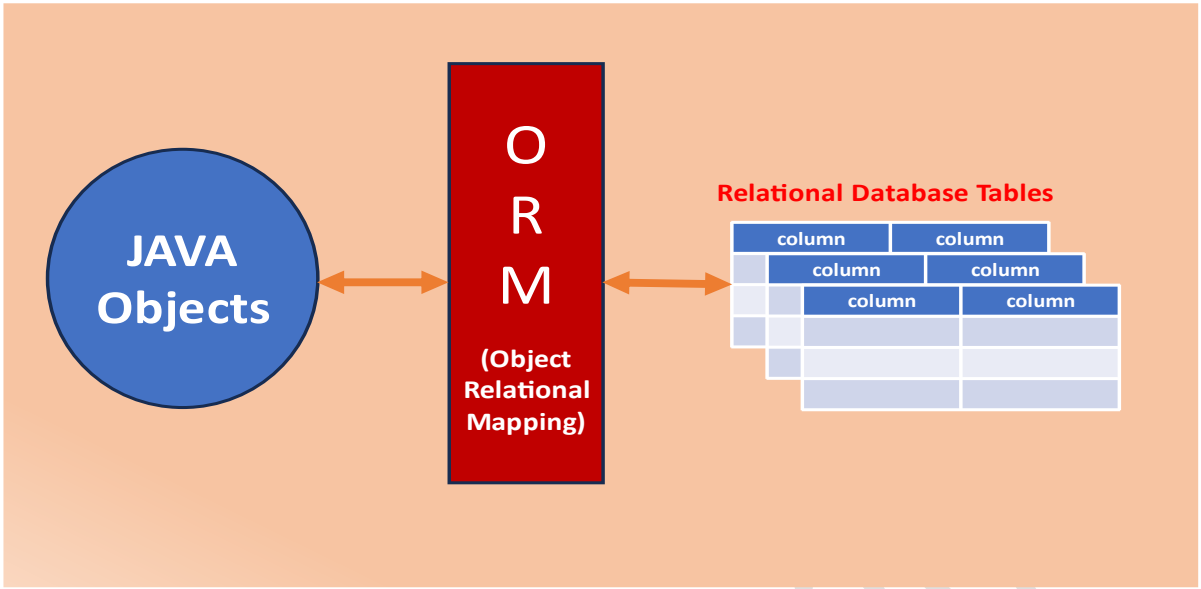
```
        <groupId>com.flipkart</groupId>
        <artifactId>flipkart</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <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>
```

- Define an Entity class for Orders table.

What is Entity Class?

An entity class is a Java class that represents a row in a database table. It is used to store data in the database and to interact with the database. Entity classes are annotated with the @Entity annotation, which tells the Java Persistence API (JPA) that the class is an entity. Entities in JPA are nothing but POJOs representing data that can be persisted 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.



Few Annotations of JPA Entity class:

@Entity: Specifies that the class is an entity. This annotation is applied to the entity class.
@Table: Specifies the primary table for the annotated entity.
@Column: Specifies the mapped column for a persistent/POJO property or field.
@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.

How to Create an 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 flipkart.entity;

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

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

```
public class FlipakartOrder {

	@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;
	}
}
```

@Table and **@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**.

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 as below in a 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:

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.

Here is an example of a simple repository:

```
public interface OrderRepository 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 **OrderRepository** 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 **OrderRepository** 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.

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.

Spring JPA Configuration:

- For using Spring Data, 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**("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:**

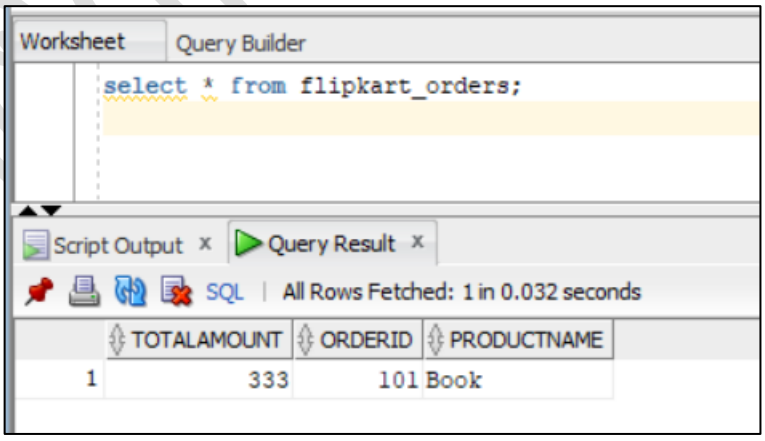
```
package flipkart.entity;

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 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:

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.

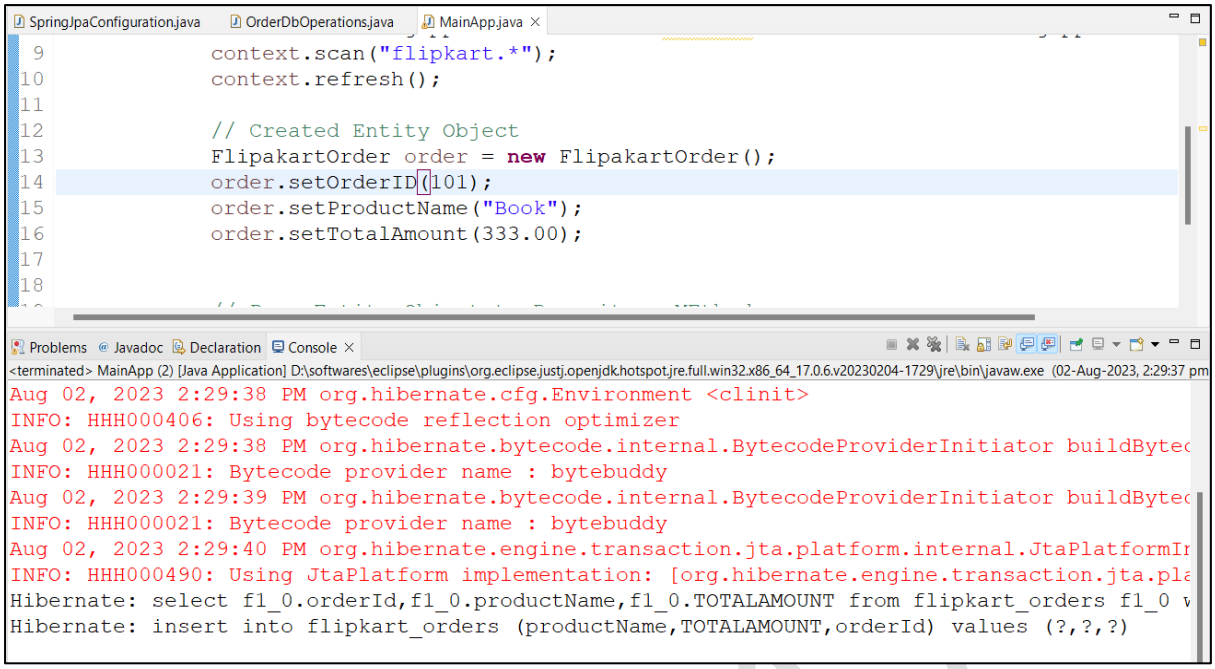
hibernate.show_sql:

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.



Creation of New Spring JPA Project:

Requirement : Patient Information

- 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 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 queries
                hibernateProperties.setProperty("hibernate.show_sql", "true");
                return hibernateProperties;
        }
}
```

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.

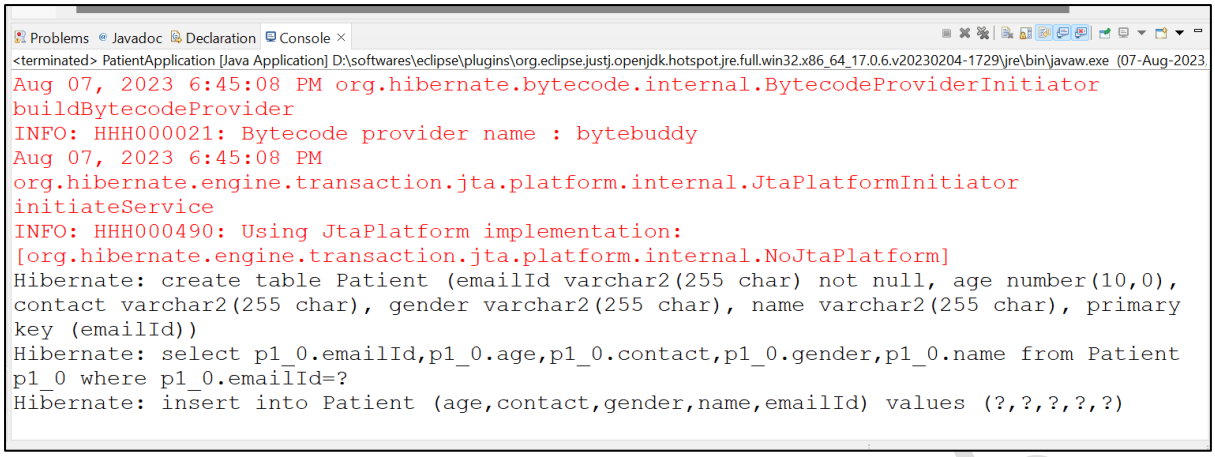
- **Add Below Method in** PatientOperations.java**:**

```
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:

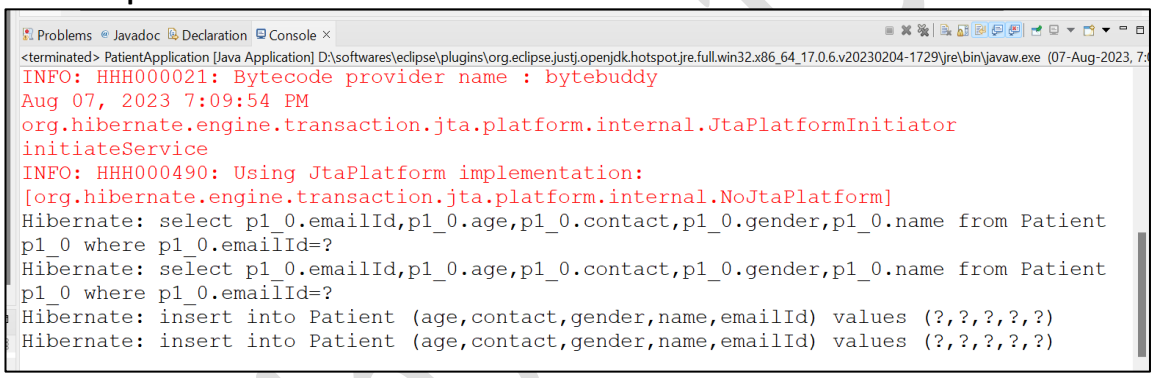
```
public void addMorePatients(List<Patient> patients) {
        repository.saveAll(patients);
}
```

Now Test it : Inside Main method class, add Logic below.

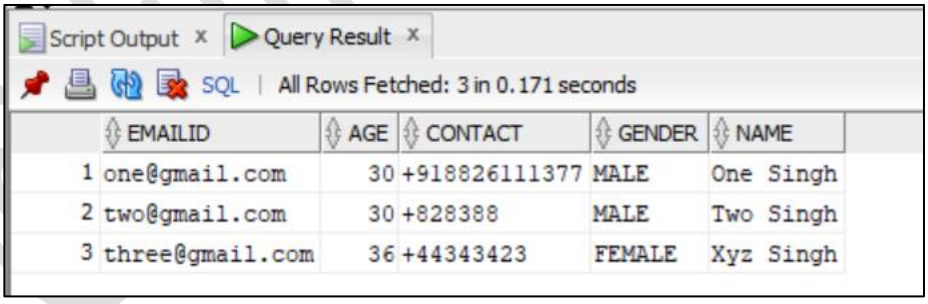
```
package com.dilip.operations;

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**

```
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

➢ **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.

PatientApplication.java × PatientOperations.java

```
2
3 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
4 import com.dilip.entity.Patient;
5
6 public class PatientApplication {
7     public static void main(String[] args) {
8         AnnotationConfigApplicationContext context = new AnnotationConfigApplicationCc
9         context.scan("com.*");
10        context.refresh();
11        PatientOperations ops = context.getBean(PatientOperations.class);
12        //Update Patient Details
13        Patient abc = new Patient();
14        abc.setEmailId("one@gmail.com");
15        abc.setName("Dilip Singh");
16        abc.setAge(30);
17        abc.setGender("MALE");
18        abc.setContact("+918826111377");
19        ops.updatePateinData(abc);
20    }
21 }
```

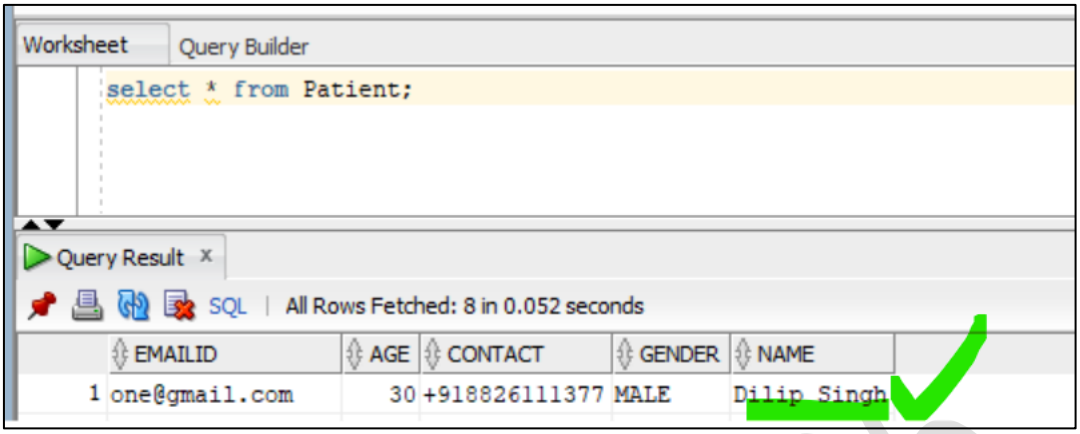
Problems Javadoc Declaration Console ×

```
<terminated> PatientApplication [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 (08-Aug-2023, 9:51
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient
p1_0 where p1_0.emailId=?
Hibernate: update Patient set age=?,contact=?,gender=?,name=? where emailId=?
```

Writable | Smart Insert | 11 : 1 : 368

ENG IN 10:00 08-08-2023

Verify In DB :



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:

```
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:**

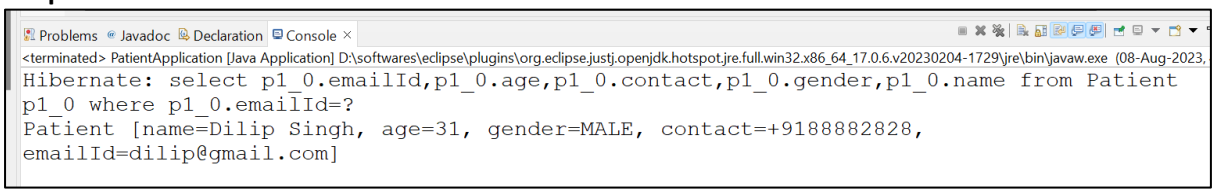
```
public Patient fetchByEmailId(String emailId) {
	return repository.findById(emailId).get();
}
```

- **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:



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.operations;

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);
                  //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 findBy 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);
	}
}
```

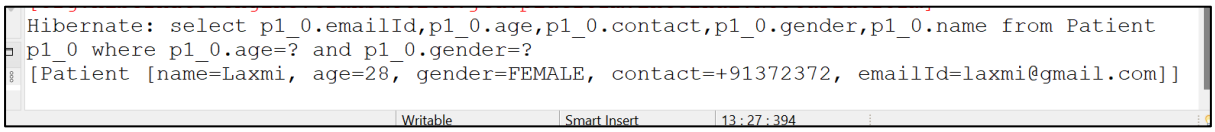
Step 3: Now Test It from Main Class.

```
package com.dilip.operations;

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);
		//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:



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);
```

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 Querys:

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

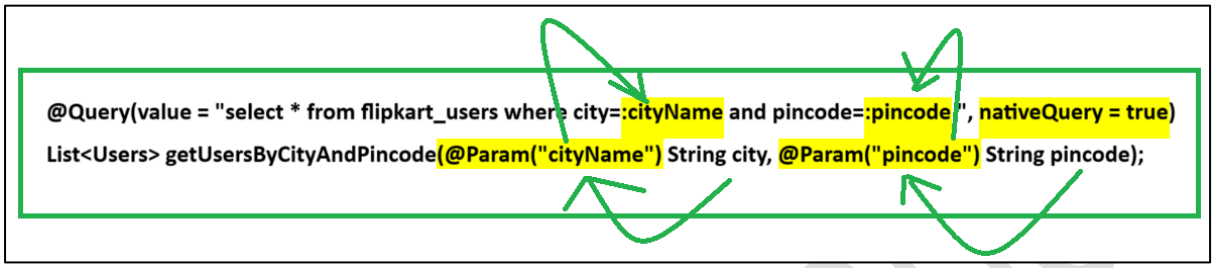
```
package com.dilip.operations;

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);

		//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: **Insert Patient Data**

Step 1: Define Method Inside Repository with Native Query:

```
package com.dilip.repository;

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.operations;

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.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);
		 //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;

	// JPQL
	// 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.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);
		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.

@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.

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 Data JPA project**
- **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() {

                // patientId : Not Passing value as part of Entity Object
                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.data;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringJpaGeneartedvalueApplication {
        public static void main(String[] args) {
            AnnotationConfigApplicationContext context =
                                                new AnnotationConfigApplicationContext();
            context.register(JPAConfiguration.class);
            context.refresh();
            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

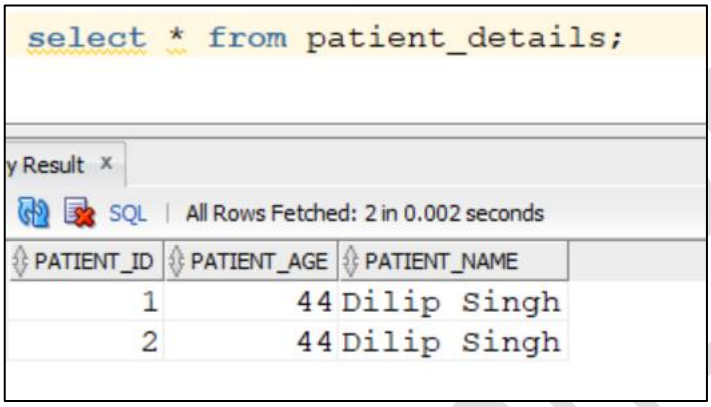
SQL | All Rows Fetched: 1 in 0.004 seconds

	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 :



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;
```

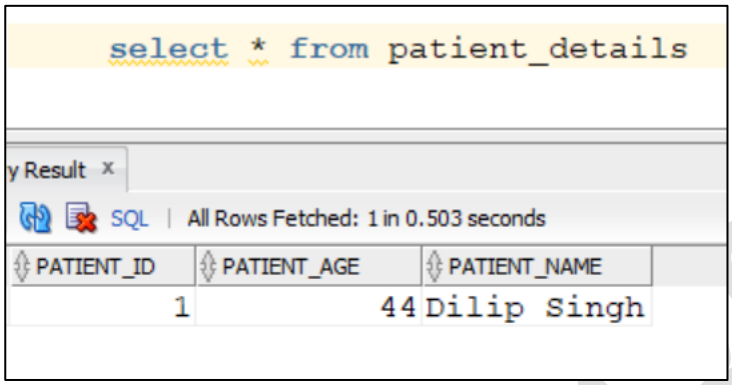
```java
@Entity
@Table(name = "patient_details")
public class Patient {

	@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.

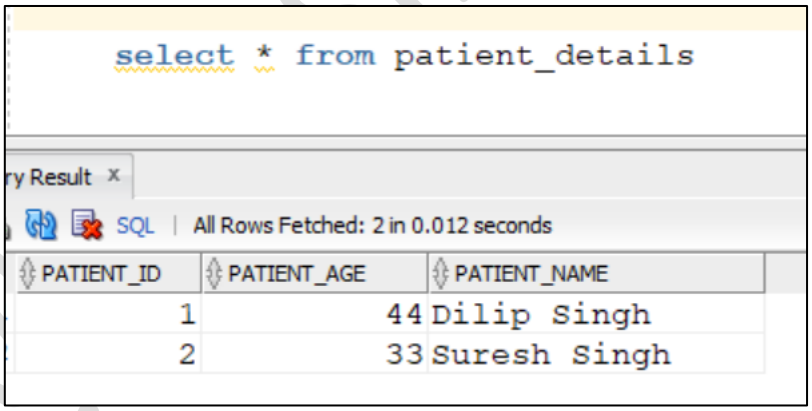
Table Data :



- 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:



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

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
	@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**.

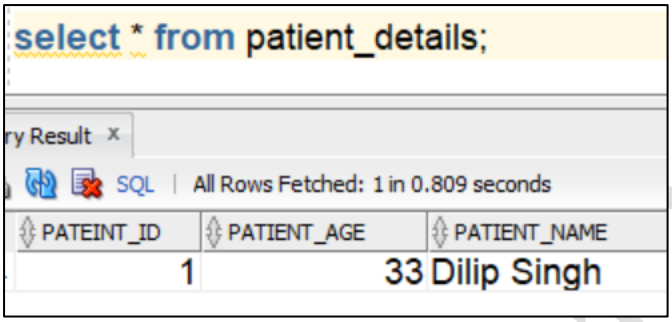
```
Hibernate: create table hibernate_sequences (next_val number(19,0), sequence_name varchar2(255 char) not null,
primary key (sequence_name))
Hibernate: insert into hibernate_sequences(sequence_name, next_val) values ('default',0)
Hibernate: create table patient_details (patient_age number(10,0), pateint_id 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.

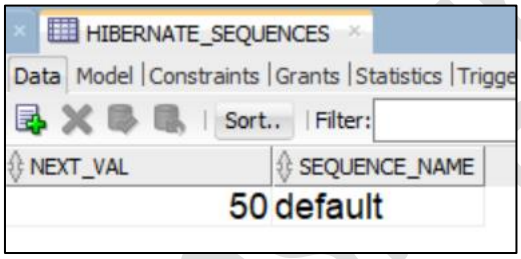
```
create table hibernate_sequences (next_val number(19,0),
sequence_name varchar2(255 char) not null, primary key
sequence_name))
```

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

Table Data:

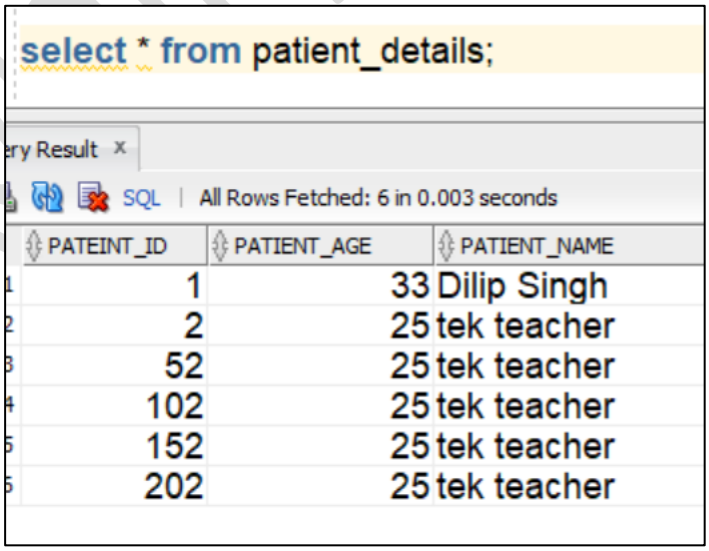


Primary Key Table: **`hibernate_sequences`**



- **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.

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
	@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 JPA Project**
- **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;

@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;

	//Setter & Getter Methods
}
```

- **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.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.dao.OrdersOperations;

public class SpringJpaSortingPaginationApplication {

	public static void main(String[] args) {
		 AnnotationConfigApplicationContext context =
					new AnnotationConfigApplicationContext();
		context.register(JPAConfiguration.class);
		context.refresh();
		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.

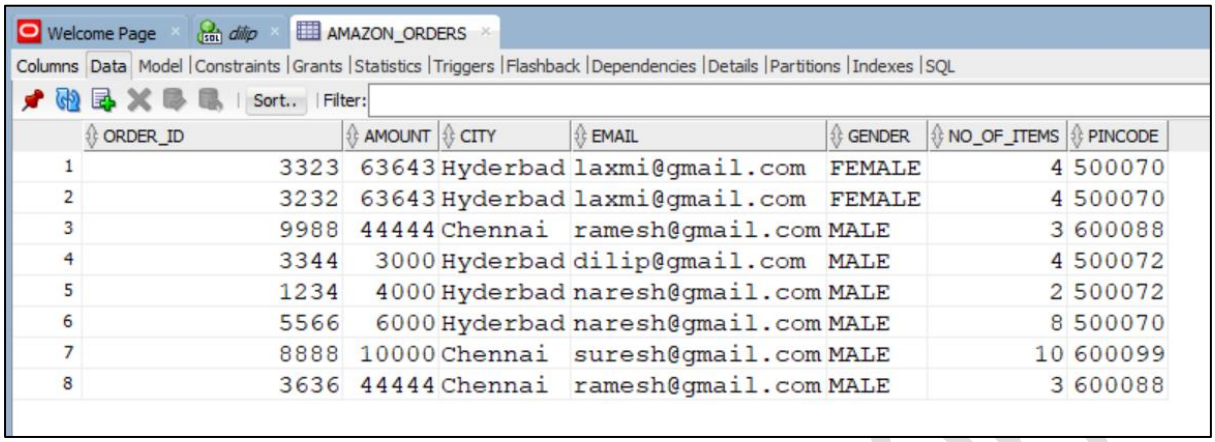
Navigation Controls: Pagination is usually accompanied by navigation controls, such as "Previous" and "Next" buttons or links. These controls allow users to move between pages easily.

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.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.dao.OrdersOperations;

public class SpringJpaSortingPaginationApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                                                new AnnotationConfigApplicationContext();
        context.register(JPAConfiguration.class);
        context.refresh();
        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
3323	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
3232	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
9988	44444	Chennai	ramesh@gmail.com	MALE	3	600088
3344	3000	Hyderbad	dilip@gmail.com	MALE	4	500072
1234	4000	Hyderbad	naresh@gmail.com	MALE	2	500072
5566	6000	Hyderbad	naresh@gmail.com	MALE	8	500070
8888	10000	Chennai	suresh@gmail.com	MALE	10	600099
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 com.dilip.dao.OrdersOperations;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringJpaSortingPaginationApplication {
        public static void main(String[] args) {
            AnnotationConfigApplicationContext context =
                                        new AnnotationConfigApplicationContext();
             context.register(JPAConfiguration.class);
             context.refresh();
             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 com.dilip.dao.OrdersOperations;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringJpaSortingPaginationApplication {

	public static void main(String[] args) {
```

```
            AnnotationConfigApplicationContext context =
                                        new AnnotationConfigApplicationContext();
            context.register(JPAConfiguration.class);
            context.refresh();
            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)
]
```