## ABOUT SPRING DATA JPA:

## WHY SPRING DATA JPA

We saw how to develop the data access layer of an application using the Spring ORM JPA (with Spring Boot) module of the Spring framework.

**Let us look at the limitations while using these approaches:**

- The Programmer has to write the code to perform common database operations(CRUD operations) in repository class implementation.
- A developer can define the required database access methods in the repository implementation class in his/her own way, which leads to inconsistency in the data access layer implementation.

So in this course, we will see how **Spring Data JPA** helps us to overcome these limitations.

Now let's look at the high-level design of our InfyTel application when it will use the Spring Data JPA module of Spring Framework.

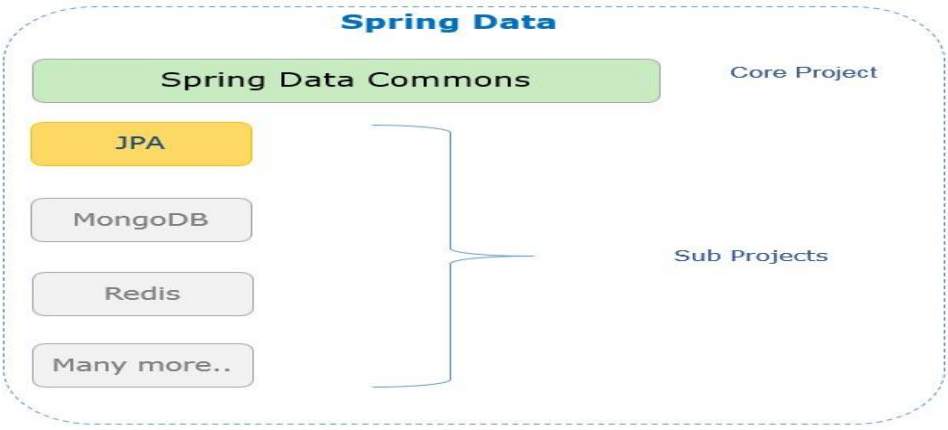
### WHAT IS SPRING DATA JPA?

Spring Data is a high-level project from Spring whose objective is to unify and ease access to different types of data access technologies including both SQL and NoSQL data stores. It has many sub-projects.

Spring Data simplifies the data access layer by removing the repository(DAO) implementations entirely from your application. Now, the only artifact that needs to be explicitly defined is the interface. Let us look at this with an example later in this course.

Spring Data supports many environments as shown below:

- Core Project supports concepts applicable to all Spring Data projects.
- Sub Projects support technology-specific details.



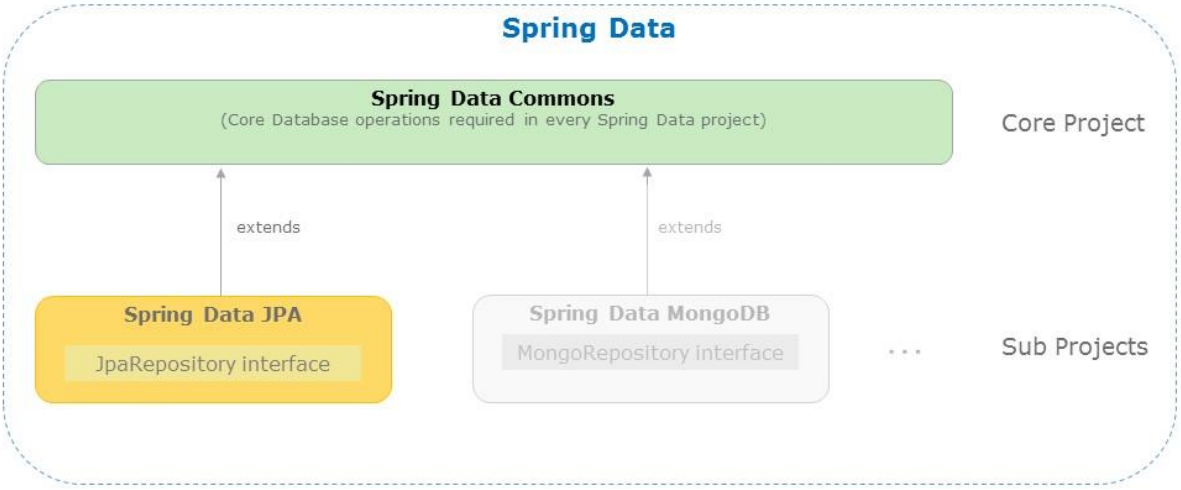
Spring Data JPA is one of the Spring Data subproject to support the implementation of JPA-based repositories.

## SPRING DATA JPA-INTERFACES

Spring Data JPA helps to implement the persistence layer by reducing the effort that is actually needed.

As a part of the core project, **Spring Data Commons** provides basic interfaces to support the following commonly used database operations:

- Performing CRUD (create, read, update, delete) operations
- Sorting of data
- Pagination of data
- Spring Data provides persistent technology-specific abstractions as interfaces through its sub-projects.
- JpaRepository interface to support JPA.
- MongoRepository interface to support MongoDB and many more.




Let us understand more about how to implement the application data access layer using Spring Data JPA.

## SPRING DATA JPA-REQUIRED DEPENDENCIES

Spring Data abstracts the data access technology-specific details from your application. Now, the application has to extend only the relevant interface of Spring Data to perform required database operations.

For example, if you would like to implement your application's data access layer using JPA repository, then your application has to define an interface that extends the JpaRepository interface.

Let us understand how Spring Data JPA applications can be created with Spring Boot

# INTRO TO SPRINGBOOT:

Let us understand Spring Boot

We have learned that Spring is a lightweight framework for developing enterprise applications. But using Spring for application development is challenging for developer because of the following reason which reduces productivity and increases the development time:

**1. Configuration**

Developing a Spring application requires a lot of configuration. This configuration also needs to be overridden for different environments like production, development, testing, etc. For example, the database used by the testing team may be different from the one used by the development team. So we have to spend a lot of time writing configuration instead of writing application logic for solving business problems.

**2. Project Dependency Management**

When you develop a Spring application you have to search for all compatible dependencies for the Spring version that you are using and then manually configure them. If the wrong version of dependencies is selected, then it will be an uphill task to solve this problem. Also for every new feature added to the application, the appropriate dependency needs to be identified and added. All this reduces productivity.

So to handle all these kinds of challenges Spring Boot came in the market.

**What is Spring Boot?**

Spring Boot is a framework built on the top Spring framework that helps developers build Spring-based applications quickly and easily. The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.

But how does it work? It works because of the following reasons,

**1. Spring Boot is an opinionated framework**

Spring Boot forms opinions. It means that Spring Boot has some sensible defaults which you can use to quickly build your application. For example, Spring Boot uses embedded Tomcat as the default web container.

**2. Spring Boot is customizable**

Though Spring Boot has its defaults, you can easily customize it at any time during your development based on your needs. For example, if you prefer log4j for logging over Spring Boot built-in logging support then you can easily make dependency change in your pom.xml file to replace the default logger with log4j dependencies.

The main Spring Boot features are as follows:

1. Starter Dependencies
2. Automatic Configuration
3. Spring Boot Actuator
4. Easy-to-use Embedded Servlet Container Support

## CREATING A SPRING BOOT APPLICATION

There are multiple approaches to create a Spring Boot application. You can use any of the following approaches to create the application:

- Using Spring Initializr
- Using the Spring Tool Suite (STS)
- Using Spring Boot CLI

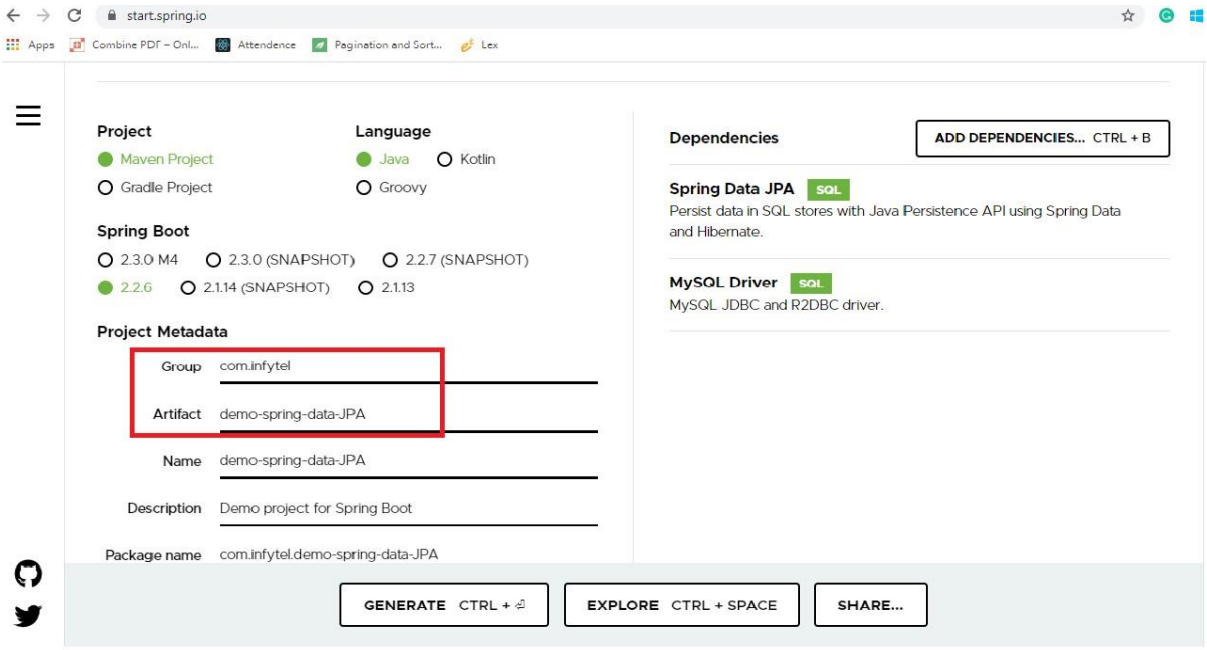
In this course, you will learn how to use Spring Initializr for creating Spring Data JPA applications.

## SPRING INITIALIZER

It is an online tool provided by Spring for generating Spring Boot applications which is accessible at http://start.spring.io. You can use it for creating a Spring Boot project using the following steps:

Step 1: Launch Spring Initializr to create your Spring Data Boot application and do the below.

Group and Artifact under Project Metadata stand for package name and project name respectively. Give them any valid value according to your project, retain other default values (Project, Language & Spring Boot version). Add dependencies required to run your Spring Boot application. In this particular case, you need to add two dependencies namely Spring Data JPA & MySQL Driver( InfyTel application uses MySQL DB)



**Note: This screen keeps changing depending on updates from Pivotal and change in the Spring Boot version.**

Step 2: Select Project as Maven, Language as Java, and Spring Boot as 2.2.6 and the necessary dependencies for SpringORM as shown in the image, then enter the project details as follows:

Choose com.infytel as Group

Choose demo-spring-data-JPA as Artifact

Click on More options and choose com.infytel as package Name

Step 3: Click on Generate Project. This would download a zip file to the local machine.

Step 4: Unzip the zip file and extract to a folder.

Step 5: In Eclipse/ STS, Click File → Import → Existing Maven Project. Navigate or type in the path of the folder where you extracted the zip file to the next screen.

**PROJECT COMPONENTS**

The generated project contains the following files:

1. **pom.xml:** This file contains information about the project and configuration details used by Maven to build the project.
2. **Spring Boot Starter Parent and Spring Boot starters**: Defines key versions of dependencies and combine all the related dependencies under a single dependency.
3. **application.properties**: This file contains application-wide properties. Spring reads the properties defined in this file to configure a Spring Boot application. A developer can define a server's default port, server's context path, database URLs, etc, in this file.
4. **ClientApplication**: Main application with @SpringBootApplication and code to interact with the end user.

## DEPENDENCIES

**1. pom.xml :**

This file contains information about the project and configuration/dependency details used by Maven to build the Spring Data JPA project.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   https://maven.apache.org/xsd/maven-4.0.0.xsd">
4.   <modelVersion>4.0.0</modelVersion>
5.   <parent>
```

```
6.          <groupId>org.springframework.boot</groupId>
7.          <artifactId>spring-boot-starter-
  parent</artifactId>
8.          <version>2.2.6.RELEASE</version>
9.          <relativePath/> <!-- lookup parent from
  repository -->
10.       </parent>
11.     <groupId>com.infytel</groupId>
12.       <artifactId>demo-spring-data-JPA</artifactId>
13.     <version>0.0.1-SNAPSHOT</version>
14.
15.     <name>demo-spring-data-JPA</name>
16.
17.     <description>Demo project for Spring Data JPA with
  Spring Boot</description>
18.       <properties>
19.             <java.version>1.8</java.version>
20.       </properties>
21.       <dependencies>
22.             <dependency>
23.
    <groupId>org.springframework.boot</groupId>
24.                   <artifactId>spring-boot-starter-data-
  jpa</artifactId>
25.             </dependency>
26.             <dependency>
27.                   <groupId>mysql</groupId>
28.                   <artifactId>mysql-connector-
  java</artifactId>
29.                   <scope>runtime</scope>
30.             </dependency>
31.             <dependency>
32.
    <groupId>org.springframework.boot</groupId>
33.                   <artifactId>spring-boot-starter-
  test</artifactId>
34.                   <scope>test</scope>
35.                   <exclusions>
36.                         <exclusion>
37.
    <groupId>org.junit.vintage</groupId>
38.                                     <artifactId>junit-
  vintage-engine</artifactId>
39.                         </exclusion>
40.                   </exclusions>
41.             </dependency>
42.       </dependencies>
43.       <build>
44.             <plugins>
45.                   <plugin>
```

```
46.
    <groupId>org.springframework.boot</groupId>
47.                             <artifactId>spring-boot-maven-
   plugin</artifactId>
48.                     </plugin>
49.                 </plugins>
50.         </build>

51. </project>
```

Let us see more about the content of pom.xml

**2. Spring Boot Starter Parent:**

The Spring Boot Starter Parent defines key versions of dependencies and default plugins for quickly building Spring Boot applications. It is present in pom.xml file of application as a parent as follows:

```
1. <parent>
2.         <groupId>org.springframework.boot</groupId>
3.         <artifactId>spring-boot-starter-
   parent</artifactId>
4.         <version>2.2.6.RELEASE</version>
5.         <relativePath/>

6. </parent>
```

It allows you to manage the following things for multiple child projects and modules:

- Configuration – The Java version and other properties.
- Dependencies – The version of dependencies.

Default Plugins Configuration – This includes default configuration for Maven plugins such as maven-failsafe-plugin, maven-jar-plugin, maven-surefire-plugin, maven-war-plugin.

**Spring Boot starters:**

Now let us discuss other starters of Spring Boot.

Spring Boot starters are pre-configured dependency descriptors with the most commonly used libraries that you can add in your application. So you don't need to search for compatible libraries and configure them manually. Spring Boot will ensure that the necessary libraries are added to the build. To use these starters, you have to add them to the pom.xml file. For example, to use spring-boot-starter following dependency needs to be added in pom.xml.

```
1. <dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>

</dependency>
```

The starters combine all the related dependencies under a single dependency. It is a one-stop-shop for all required Spring related technologies without browsing through the net and downloading them individually.

Spring Boot comes with many starters. Some popular starters are as follows:

- spring-boot-starter - This is the core starter that includes support for auto-configuration, logging, and YAML.
- spring-boot-starter-aop - This starter is used for aspect-oriented programming with Spring AOP and AspectJ.
- spring-boot-starter-data-jdbc - This starter is used for Spring Data JDBC.
- spring-boot-starter-data-jpa - This starter is used for Spring Data JPA with Hibernate.
- spring-boot-starter-web - This starter is used for building a web application using Spring MVC and Spring REST. It also provides Tomcat as the default embedded container.
- spring-boot-starter-test - This starter provides support for testing Spring Boot applications using libraries such as JUnit, Hamcrest, and Mockito.

**One of the starters of Spring Boot is spring-boot-starter-data-jpa.** It informs Spring Boot that it is a Spring Data JPA application.

```
<dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-
jpa</artifactId>

</dependency>
```

Note that the version number of spring-boot-starter is not defined here as it can be determined by spring-boot-starter-parent. By opening the Dependency Hierarchy tab of your pom.xml, you can see that this single starter POM combines several dependencies in it.

**Spring Data JPA dependency:**

The following is the Spring Data JPA dependency - spring-boot-starter-data-jpa added in pom.xml. This dependency will add all the necessary libraries for Spring ORM to the project automatically.

```
<dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-
jpa</artifactId>
</dependency>

```

**Database dependency:**

The MySQL dependency mysql-connector-java added in pom.xml.

```
<dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-
java</artifactId>
                <scope>runtime</scope>
</dependency>
```

**3.application.properties:** This is for adding the database details to the project (added the necessary details for MySQL DB).

```
spring.datasource.driverClassName=com.mysql.cj.jdbc.Drive
r
spring.datasource.url=jdbc:mysql://localhost:3306/sample
spring.datasource.username=root

spring.datasource.password=root
```

**4.ClientApplication:**

The class Client is annotated with @SpringBootApplication annotation which is used to bootstrap Spring Boot application.

```
@SpringBootApplication
public class ClientApplication{
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class,
args);
    }
```

```
6. }
```

The @SpringBootApplication annotation indicates that it is a configuration class and also triggers auto-configuration and component scanning. It is a combination of following annotations with their default attributes.

- @EnableAutoConfiguration – This annotation enables auto-configuration for Spring boot application which automatically configures our application based on the dependencies that a developer has already added.
- @ComponentScan – This enables the Spring bean dependency injection feature by using @Autowired annotation. All application components which are annotated with @Component, @Service, @Repository or @Controller are automatically registered as Spring Beans. These beans can be injected by using @Autowired annotation.
- @Configuration – This enables Java based configurations for Spring boot application.

The class that is annotated with @SpringBootApplication will be considered as the main class, is also a bootstrap class. It kicks starts the application by invoking the SpringApplication.run() method. The developer needs to pass the .class file name of the main class to the run() method.

Note: A developer can write any piece of code to interact with an end user by implementing the CommandLineRunner interface and overriding the run() method. Let's discuss more on this.

Executing the Spring Boot application:

To execute the Spring Boot application run the application as a standalone Java class which contains the main method.

**Spring Boot Runners:**

So far you have learned how to create and start Spring Boot application. Now suppose you want to perform some action immediately after the application has started then for this Spring Boot provides the following two interfaces:

- CommandLineRunner
- ApplicationRunner

CommandLineRunner is the Spring Boot interface with a run() method. Spring Boot automatically calls this method of all beans implementing this interface after the application context has been loaded. To use this interface, you can modify the ClientApplication.java class as follows:

```
1. @SpringBootApplication
2. public class ClientApplication implements
   CommandLineRunner {
3.  public static void main(String[] args) {
```

```
4.
    SpringApplication.run(DemoSpringBootApplication.class,
   args);
5.   }
6.   @Override
7.   public void run(String... args) throws Exception {
8.         System.out.println("Welcome to
   CommandLineRunner");
9.   }

10. }
```

We have seen how Spring Boot helps in developing Spring Data JPA applications.

Let us now discuss how to implement the InfyTel application CRUD operation using Spring Data JPA with Spring Boot in the coming modules.

## SPRING DATA JPA CONFIG:

### DEFINING REPOSITORY INTERFACE

We have seen how to create a Spring Boot project with the necessary dependencies for Spring Data JPA.

Let's now look at how to implement the required interfaces for the persistence layer of the InfyTel Customer management application.

We need to define a repository interface for InfyTel Customer as below when we need to insert/delete a Customer data:

In the data access layer of the application, we need only the interface extending Spring's JpaRepository<T, K> interface as shown below:

```
1. public interface CustomerRepository extends
   JpaRepository<Customer, Long> {

2.   }
```

Here, the CustomerRepository interface extends a Spring provided interface JpaRepository but the below details needs to be provided to Spring through JpaRepository<Customer, Long> interface:

1. Entity class name to which you need the database operations (In this example, entity class is Customer).
2. The Datatype of the primary key of your entity class (Customer class primary key type is a long).

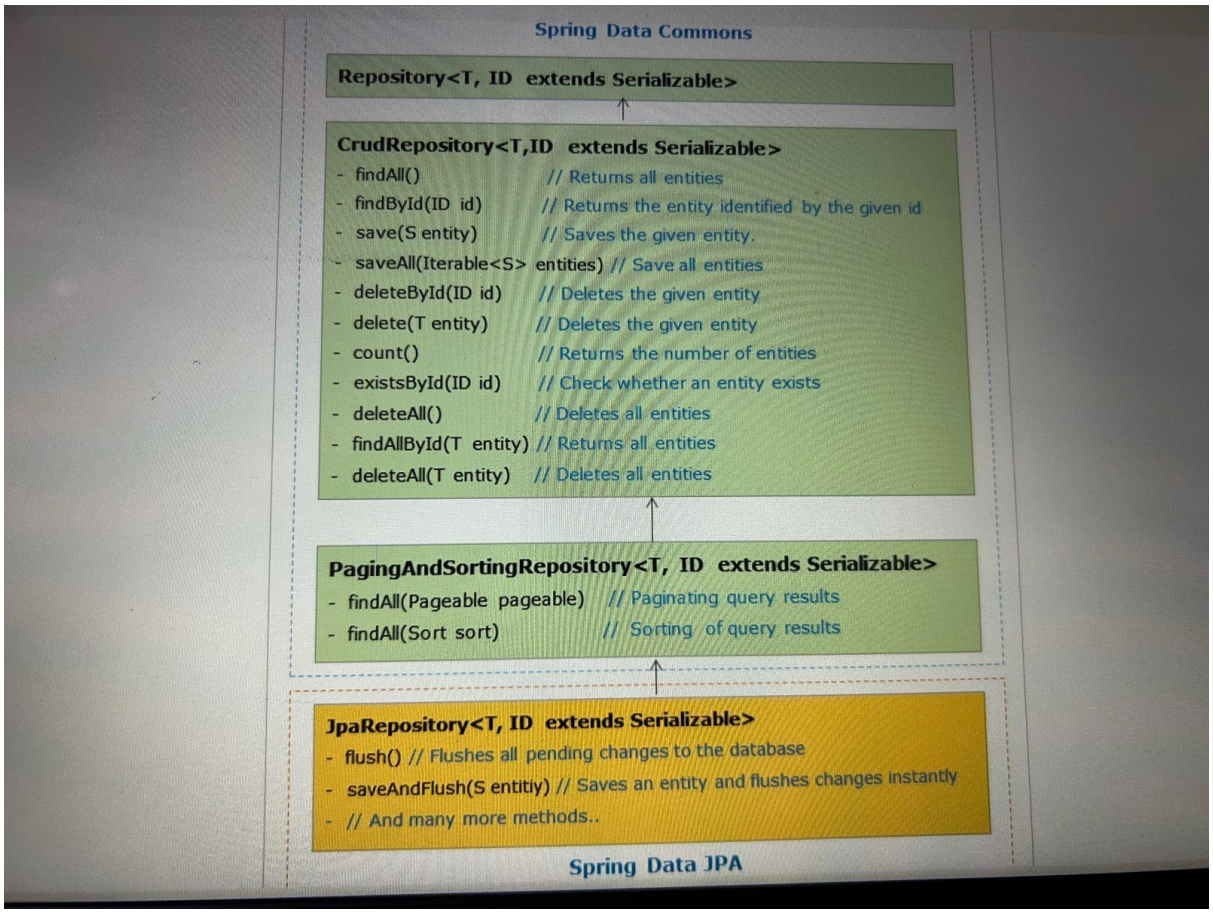
Spring scans the interface that extends the JpaRepository interface, auto-generates common CRUD methods, paging, and sorting methods at run time through a proxy object.

@Repository annotation can be used at the user-defined interface as Spring provides repository implementation of this interface. However, it is optional to mention the annotation explicitly because Spring auto-detects this interface as a Repository.

```
1. @Repository
2. public interface CustomerRepository extends
   JpaRepository<Customer, Long> {
3.

4. }
```

**Note:** It is recommended to give interface name as entity class name concatenated with Repository. Hence in the example, it is defined as an interface CustomerRepository for the entity class Customer.

The below diagram shows the methods available to the application from Spring:




If the application interface is extending the JpaRepository interface, then Spring provides auto implementation of all the above-listed methods and makes them available for the application.

An appropriate method can be used in the application depending on the required database operation.

In the demo discussed **saveAndFlush()** and **delete()** methods of Spring are used to perform insertion and removal operations using repository bean.

As all of us know Spring obtains the connection to the database through **DataSource** bean which is the mandatory configuration. It allows Spring container to hide database-specific details from the application code. But as we are developing our Spring Data JPA application using Spring Boot so there is no need to explicitly definine the DataSource bean in a configuration file.

Spring's DriverManagerDataSource class is used to define the Datasource bean. Spring Boot will automatically create the Datasource bean by reading the database details like driver details, connection URL, username, and password present in the application.properties file.

So we need to define the below configurations in the application.properties.

```
1. spring.datasource.driverClassName=com.mysql.jdbc.Driver
2. spring.datasource.url=jdbc:mysql://localhost:3306/sample
3. spring.datasource.username=root

4. spring.datasource.password=root
```

Let's see how our service layer and repository layer should be implemented for implementing CRUD operations using Spring Data JPA with Spring Boot.

**CRUD OPERATIONS WITH SPRING DATA JPA-REPOSITORY LAYER**

As seen in the **Spring Data JPA Repository hierarchy** when we are extending the **JpaRepository** interface then Spring provides the auto implementation of all the methods from *JpaRepository*, *PagingAndSortingRepository*, *CrudRepository* and will make them available to the application. An appropriate method can be used in the **service layer** of the application depending on the required database operation.

So our CustomerRepository interface will look as below:

```
1. package com.infyTel.repository;
2. import
   org.springframework.data.jpa.repository.JpaRepository;
3. import com.infyTel.domain.Customer;
4. public interface CustomerRepository extends
   JpaRepository<Customer, Long>{

5. }
```

Spring Data JPA provides a default implementation for each method defined by one of its repository interfaces. That means that you no longer need to implement basic read or write operations by implementing the above interface. So for implementing our CRUD operations we are going to use 4 methods from their respective interfaces, which are listed below:

- saveAndFlush(S entity) of JpaRepository which saves an entity and flushes changes instantly.
- deleteById(ID id) of CrudRepository which deletes the entity based on the given id/primary key.
- findById(ID id) of CrudRepository which returns a given entity based on the given id/primary key.
- save(S entity) of CrudRepository which saves the given entity in the database.

Now let's look at how the service layer is implemented and the service implementation class uses these 4 methods depending on the required database operation.

## CRUD OPERATIONS WITH SPRING DATA JPA- SERVICE LAYER

The service interface will act as a bridge between our Client class and Service implementation class. So to perform CRUD operation we can define all the required methods in the CustomerService interface as below:

**CustomerService.java:**

```
1. package com.infyTel.service;
2. import com.infyTel.dto.CustomerDTO;
3. public interface CustomerService {
4.  public void insertCustomer(CustomerDTO Customer) ;
5.  public void removeCustomer(Long phoneNo);
6.  public CustomerDTO getCustomer(Long phoneNo);
7.  public String updateCustomer(Long phoneNo,Integer
   newPlanId);

8. }
```

The service implementation class invokes repository methods through repository bean to perform the required CRUD operation as shown below:

CustomerServiceImpl.java:

```
1. package com.infyTel.service;
2. import java.util.Optional;
3. import
   org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.stereotype.Service;
5. import com.infyTel.domain.Customer;
```

```
6. import com.infyTel.dto.CustomerDTO;
7. import com.infyTel.repository.CustomerRepository;
8. @Service("customerService")
9. public class CustomerServiceImpl implements
   CustomerService {
10.         @Autowired
11.         private CustomerRepository repository;
12.         @Override
13.         public void insertCustomer(CustomerDTO customer)
   {
14.
     repository.saveAndFlush(CustomerDTO.prepareCustomerEntit
   y(customer));
15.         }
16.         @Override
17.         public void removeCustomer(Long phoneNo) {
18.               repository.deleteById(phoneNo);
19.         }
20.         @Override
21.         public CustomerDTO getCustomer(Long phoneNo) {
22.               Optional<Customer> optionalCustomer =
   repository.findById(phoneNo);
23.               Customer customerEntity =
   optionalCustomer.get();// Converting Optional<Customer>
   to Customer
24.               CustomerDTO customerDTO =
   Customer.prepareCustomerDTO(customerEntity);
25.               return customerDTO;
26.         }
27.         @Override
28.         public String updateCustomer(Long phoneNo,
   Integer newPlanId) {
29.               Optional<Customer> optionalCustomer =
   repository.findById(phoneNo);
30.               Customer customerEntity =
   optionalCustomer.get();
31.               customerEntity.setPlanId(newPlanId);
32.               repository.save(customerEntity);
33.               return "The plan for the customer with
   phone number :" + phoneNo + " has been updated
   successfully.";
34.         }

35. }
```

In service class implementation, we are autowiring CustomerRepository instance to make use of required repository methods in the service layer.

Even though we are defining CustomerRepository as an interface, we are able to get an instance of CustomerRepository because Spring internally provides a proxy object for this interface with auto-generated methods. Hence we could autowire bean of CustomerRepository.

Demo 3: InfyTel Application development using  Spring Data JPA

Highlights:

- To understand key concepts and configuration of Spring Data JPA
- To perform CRUD operations using Spring Data JPA

Consider the InfyTel scenario, create an application to perform the following operations using Spring Data JPA:

- Insert customer details
- Delete customer for given phone number
- Find customer for a given phone number
- Update customer for a given phone number

While generating the Spring Boot Maven project from Spring initializer, Select "Spring Data JPA" and "MySQL Driver" dependency.

PAGINATION AND SORTING

**PagingAndSortingRepository** interface of Spring Data Commons provides methods to support paging and sorting functionalities.

```
1. public interface PagingAndSortingRepository<T, ID extends
   Serializable> extends CrudRepository<T, ID> {
2.              Iterable<T> findAll(Sort sort);
3.              Page<T> findAll(Pageable pageable);

4. }
```

A Page object provides the data for the requested page as well as additional information like total result count, page index, and so on.

Pageable instance dynamically adds paging details to statically defined query. Let us see more details in the example.

**Pagination**

The steps to paginate the query results are:

Step 1: JpaRepository is a sub-interface of the PagingAndSortingRepository interface. The Application standard interface has to extend JpaRepository to get paging and sorting methods along with common CRUD methods.

```
1. public interface CustomerRepository extends
   JpaRepository<Customer, Long> {
2.
```

}

Step 2: In client code, create a org.springframework.data.domain.Pageable object by instantiating org.springframework.data.domain.PageRequest describing the details of the requested page is shown below:

To ask for the required page by specifying page size, a new PageRequest instance must be created.

```
1. //First argument '0" indicates first page and second
   argument 4 represents number of records.

2. Pageable pageable = PageRequest.of(0, 4);
```

Step 3: In the client code, pass the Pageable object to the repository method as a method parameter.

```
1. Page<Customer> customers =
   customerRepository.findAll(pageable);
```

Sorting is to order query results based on the property values of the entity class.

For example, to sort employee records based on the employee's first name field can be done using the below steps:

**Step 1:** Standard interface has to extend JpaRepository to use sorting method provided by Spring

```
1. public interface CustomerRepository extends
   JpaRepository<Customer, Long> {
2.

3. }
```

**Step 2:** In the client code, create org.springframework.data.domain.Sort instance describing the sorting order based on the entity property either as ascending or descending and pass the instance of Sort to the repository method.

```
customerRepository.findAll(Sort.by(Sort.Direction.ASC,"name"));
```

In the Sort() method used above,

- The first parameter specifies the order of sorting i.e. is ascending order.
- The second parameter specifies the field value for sorting.

QUERY APPROACHES:

So far, you have learned how Spring provides common database operations.

Spring Data by default provides the following query methods through its interface:

- findAll //Returns all entities
- findById(ID id) //Returns the entity depending upon given id

What if one wants to query customer records based on the customer's address?

How does Spring support this scenario?

Spring supports these kinds of scenarios using the following approaches:

- Query creation based on the method name.
- Query creation using @NamedQuery: JPA named queries through a naming convention.
- Query creation using @Query: annotate the query method with @Query.

Query method names are derived by combining the property name of an entity with supported keywords such as "findBy", "getBy", "readBy" or "queryBy".

```
//method name where in <Op> is optional, it can be Gt,Lt,Ne,Between,Like etc..
findBy <DataMember><Op>
```

Example: Consider the Customer class as shown below:

```
public class Customer {

  private Long phoneNumber;
  private String name;
  private Integer age;
```

```
6.  private Character gender;
7.  private String address;
8.  private Integer planId;
9. //getters and setters

10. }
```

To query a record based on the address using query creation by the method name, the following method has to be added to the CustomerRepository interface.

```
1. interface CustomerRepository extends
   JpaRepository<Customer, Long>{
2.          Customer findByAddress(String address); //
   method declared

3. }
```

The programmer has to provide only the method declaration in the interface. Spring takes care of auto-generating the query, the mechanism strips the prefix findBy from the method and considers the remaining part of the method name and arguments to construct a query.

Consider the Customer and Address classes given below:

**Customer.java**

```
1. @Entity
2. public class Customer{
3.  @Id
4.     int customerId;
5.     boolean active;
6.     int creditPoints;
7.     String firstName;
8.     String lastName;
9.     String contactNumber;
10.     String email;
11.     @OneToOne( cascade = CascadeType.ALL)
12.     @JoinColumn
13.     Address address;
14.     -----------

15. }
```

Address.java

```
@Entity
public class Address {
    @Id
    private int addressId;
  private String city;
  private String pincode;
    ----------------

}
```

The CustomerRepository interface with some common findBy methods is shown below:

```
    public interface CustomerRepository extends
JpaRepository<Customer, Integer> {

        // Query record based on email
        // Equivalent JPQL: select c from Customer c
where c.email=?
        Customer findByEmail(String email);

        // Query records based on LastName is like the
provided last name
        // select c from Customer c where c.lastName
LIKE CONCAT('%',?,'%')
        List<Customer> findByLastNameLike(String
lastname);

        // Query records based on email or contact
number
        // select c from Customer c where c.email=? or
c.contactNumber=?
        List<Customer>
findByEmailOrContactNumber(String email, String number);

        // Query records based on FirstName and city
details. Following query creates the property traversal
for city as Customer.address.city
        // select c from Customer c  where
c.firstName=? and c.address.city=?
         List<Customer>
findByFirstNameAndAddress_City(String fname, String
city);

        // Query records based on last name and order
by ascending based on first name
        // select c from Customer c where c.lastName=?
order by c.firstName
```

```
21.         List<Customer>
findByLastNameOrderByFirstNameAsc(String lastname);
22.
23.         // Query records based on specified list of
cities
24.         //select c from Customer c where
c.address.city in ?1
25.         List<Customer>
findByAddress_CityIn(Collection<String> cities);
26.
27.         // Query records based if customer is active
28.         //select c from Customer c where c.active =
true
29.         List<Customer> findByActiveTrue();
30.
31.         // Query records based on creditPoints >=
specified value
32.         //select c from Customer c where
c.creditPoints >=?1
33.         List<Customer>
findByCreditPointsGreaterThanEqual(int points) ;
34.
35.         // Query records based on creditpoints between
specified values
36.         //select c from Customer c where
c.creditPoints between ?1 and ?2
37.         List<Customer> findByCreditPointsBetween(int
point1, int point2)
38.     }
```

So far, we learned the following Query creation approaches in Spring Data JPA.

- Query creation based on the method name.
- Query creation using @NamedQuery: JPA named queries through a naming convention.
- Query creation using @Query: annotate your query method with @Query.

If a query is provided using more than one approach in an application. What is the default precedence given by the Spring?

Following is the order of default precedence:

1. @Query always takes high precedence over other options
2. @NameQuery
3. findBy methods

**NAMED QUERIES:**

By now you know, how to support a query through a method name.

Though a query created from the method name suits very well but, in certain situations, it is difficult to derive a method name for the required query.

The following options can be used in these scenarios:

1. Using JPA NamedQueries: JPA named queries through a naming convention
2. Using @Query: Use @Query annotation to your query method

Let us now understand JPA NamedQueries:

Define annotation-based configuration for a NamedQuery at entity class with @NamedQuery annotation specifying query name with the actual query.

```
1. @Entity
2. //name starts with the entity class name followed by the
   method name separated by a dot.
3. @NamedQuery(name = "Customer.findByAddress", query =
   "select c from Customer c where c.address = ?1")
4. public class Customer {
5.
6.   @Id
7.   @Column(name = "phone_no")
8.   private Long phoneNumber;
9.   private String name;
10.        private Integer age;
11.        private Character gender;
12.        private String address;
13. --------

14. }
```

Now for executing this named query one needs to specify an interface as given below with method declaration.

```
1. public interface CustomerRepository extends
   JpaRepository<Customer, Long>{
2.   Customer findByAddress(String address);
3.

4. }
```

Spring Data will map a call to findByAddress() method to a NamedQuery whose name starts with entity class followed by a dot with the method name. Hence, in the above code, Spring will use the NamedQuery with the name Customer.findByAddress() method instead of creating it.

NamedQuery approach has advantage as maintenance costs are less because the queries are provided through the class. However, the drawback is that for every new query declaration domain class needs to be recompiled.

The NamedQueries approach is valid for the small number of queries.

@Query annotation can be used to specify query details at repository interface methods instead of specifying at entity class. This will also reduce the entity class from persistence related information.

Queries annotated to the query method has high priority than queries defined using @NamedQuery.

@Query is used to write flexible queries to fetch data.

**Example:** Declare query at the method level using **@Query**.

```
1. public interface CustomerRepository extends
   JpaRepository<Customer, Long>{
2. //Query string is in JPQL
3.  @Query("select cus from Customer cus where cus.address =
   ?1")
4.  Customer findByAddress(String address);
5.

6. }
```

@Query annotation supports both JPQL (Java Persistence Query Language) and native SQL queries.

By default, it supports JPQL. The nativeQuery attribute must be set to true to support native SQL.

**Example:** Declare query at the method level using @Query with nativeQuery set to true.

```
1. public interface CustomerRepository extends
   JpaRepository<Customer, Long>{
2.  //Query string is in SQL
3.  @Query("select cus from Customer cus where cus.address =
   ?1", nativeQuery = true)
4.  Customer findByAddress(String address);
5.

6. }
```

The disadvantage of writing queries in native SQL is that they become vendor-specific database and hence portability becomes a challenge. Thus, both @NamedQuery and @Query supports JPQL.

Now, what is JPQL?

**JPQL:**

JPQL is an object-oriented query language that is used to perform database operations on persistent entities. Instead of a database table, JPQL uses the 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.

**Features:**

The features of JPQL are that it can:

- perform join operations
- update and delete data in a bulk
- perform an aggregate function with sorting and grouping clauses
- provide both single and multiple value result types

**Creating Queries using JPQL:**

JPQL provides two methods that can be used to perform database operations. They are: -

1. **Query createQuery(String name)** - The createQuery() method of EntityManager interface is used to create an instance of the Hibernate Query interface for executing JPQL statement. This method creates dynamic queries that can be defined within business logic.

Some of the examples of JPQL using createQuery method:(assuming the entity class name as - 'CustomerEntity' which is mapped to a relational table 'Customer')

- Fetching all the Customer names:

```
1. Query query = em.createQuery("Select c.name from

   CustomerEntity c");
```

- Updating the plan of a customer:

```
1. Query query = em.createQuery( "update CustomerEntity SET
   planId=5 where phoneNumber=7022713766");

2. query.executeUpdate();
```

- Deleting a customer:

```
1. Query query = em.createQuery( "delete from CustomerEntity
   where phoneNumber=7022713766");

2. query.executeUpdate();
```

2. **Query createNamedQuery(String name)** - The createNamedQuery() method of EntityManager interface is used to create an instance of the Hibernate Query interface for executing named queries. This method is used to create static queries that can be defined in the entity class.

Let's see a simple example of JPQL using createNamedQuery method to fetch all the details of customers in InfyTel application:(Assume the entity class name is - 'CustomerEntity' which is mapped to a relational table 'Customer')

```
@NamedQuery(name = "getAll" , query = "Select c from
CustomerEntity s")
```

**WHY SPRINGG TRANSACTION:**

For the InfyTel application, consider a scenario where the admin of the InfyTel wants to deactivate a Plan. Once a Plan is deactivated, the Customer should be assigned with immediate next active Plan.

So, once a Plan is deactivated by the admin of the InfyTel, the Customer table should be updated with the immediate next active plan as well as the plan name in the Customer table should be updated accordingly.

This task requires the application to perform an update on two database tables Customer and Plan. What should happen if one of the table updates fails, should it continue with another table update?

The answer is **NO**. The application should not continue with other updates in order to maintain its data integrity and consistency.

The application should not do any changes to both the tables in case of failure in any one of the operations. Update of Customer details should be successful only if the update on both Plan and Customer table is successful.

How do we achieve this kind of requirement in enterprise applications?

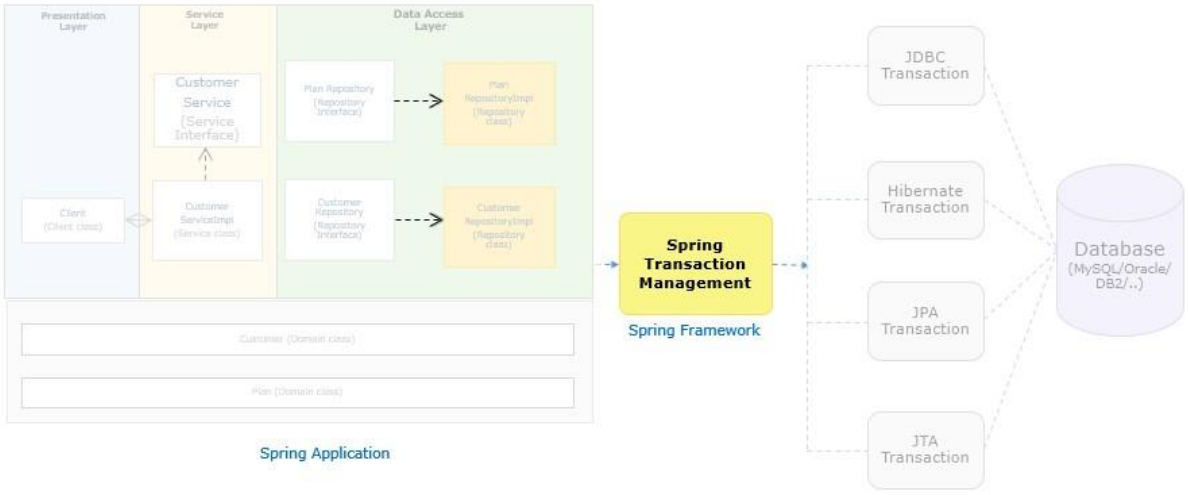
This can be achieved by executing related database operations in a transaction scope.

When we implement transaction using JDBC API, it has few limitations like:

- Transaction related code is mixed with application code, hence it is difficult to maintain the code.
- Requires a lot of code modification to migrate between local and global transactions in an application.

Let us look at how the Spring transaction helps to overcome these limitations.

SPRING DECLARATIVE TRANSACTION:




The Spring framework provides a common transaction API irrespective of underlying transaction technologies such as JDBC, Hibernate, JPA, and JTA.

You can switch from one technology to another by modifying the application's Spring configuration. Hence you need not modify your business logic anytime.

Spring provides an abstract layer for transaction management by hiding the complexity of underlying transaction APIs from the application.

- Spring supports both local and global transactions using a consistent approach with minimal modifications through configuration.
- Spring supports both declarative and programmatic transaction approaches.

Different ways to achieve Spring transaction:

| Type | Definition | Implementation |
|---|---|---|
| Declarative Transaction | Spring applies the required transaction to the application code in a declarative way using a simple configuration. Spring internally uses Spring AOP to provide transaction logic to the application code. | • Using pure XML configuration<br>• Using @Transactional annotation approach |

| | | |
|---|---|---|
| Programmatic Transaction | The Required transaction is achieved by adding transaction-related code in the application code. This approach is used for more fine level control of transaction requirements. This mixes the transaction functionality with the business logic and hence it is recommended to use only based on the need. | • Using the TransactionTemplate (adopts the same approach as JdbcTemplate)<br>• Using a PlatformTransactionManager implementation. |

Which is the preferred approach to implement Spring transactions?

Declarative transaction management is the most commonly used approach by Spring Framework users.

This option keeps the application code separate from the transaction serves as the required transaction is provided through the only configuration.

Let us proceed to understand, how Spring declarative transactions can be implemented using the annotation-based approach in the Spring Data JPA application in detail.

Note:

- In this course, we will be covering only Spring declarative transactions using the annotation-based approach.
- Spring Declarative Transaction is treated as an aspect. Spring will apply the required transaction at run time using Spring AOP.

Considering the InfyTel scenario update the Customer's current plan and Plan details. The InfyTel application uses Spring ORM for data access layer implementation. Let us now apply the Spring transaction to this application.

This requirement provides an update on the following tables:

1. Customer table: To update the current plan.
2. Plan table: To update the plan details.

The important steps to implement Spring declarative transaction in Spring ORM application is :

- Add @Transactional annotation on methods that are required to be executed in the transaction scope as all other things like dependencies management, etc are already taken care of by spring-boot-starter-data-jpa jar
- Spring Data JPA Dependency:

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-
jpa</artifactId>
```

```
</dependency>
```

MySQL Driver dependency:

```
<dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-
java</artifactId>
                <scope>runtime</scope>

</dependency>
```

We can implement Spring declarative transactions using the annotation-based approach.

Let us understand more on @Transactional annotation.

@Transactional annotation offers ease-of-use to define the transaction requirement at method, interface, or class level.

Through declarative transaction management, update (Customer customer) method can be executed in transaction scope using Spring.

```
public class CustomerServiceImpl {
  ------------------------
  @Transactional
  public void update(Customer customer) {
        //Method to update the current plan in Customer
table
        customerDAO.update(customer);
        //Method to update the new plan details in Plan
table
        planDAO.updatePlan(customer.getPlan());
  }

}
```

This annotation will be identified automatically by Spring Boot if we have already included the spring-boot-starter-data-jpa jar.

Should the developer use @Transactional annotation before every method?

No, @Transactional annotation can be placed at the class level which makes all its methods execute in a transaction scope. For e.g. insertCustomer() and updateCustomerDetails() methods executes in transaction scope.

```
@Transactional    // This annotation makes all the
methods of this class to execute in transaction scope
public class CustomerServiceImpl {
    -----------
   public void insertCustomer(Customer customer) {
          ------------------
   }

   public void updateCustomerDetails(Customer customer) {
          ------------------
     }
}
```

As the transaction is an important aspect of enterprise applications, let us understand how Spring Data JPA supports transactions.

Spring data CRUD methods on repository instances are by default transactional.

For read operations, readOnly flag is set to true and all other operations are configured by applying @Transactional with default transaction configuration.

Suppose there is a requirement to change the transaction configuration for a method declared in the CustomerRepository interface. Simply re-declare the method as shown below:

```
public interface CustomerRepository extends
JpaRepository<Customer,Long>{
@Override
@Transactional(readOnly = true)
public List<Customer> findAll();
// Further query method declarations

}
```

Here, findAll() is annotated with @Transactional by setting the readOnly flag to true, this will override the default behavior provided by Spring Data. We'll discuss more attributes of @Transacational annotation in the upcoming topics.

In a scenario with multiple transactions, the following attributes can be used with @Transactional to determine the behavior of a transaction.

@Transactional annotation supports the following attributes:

**Transaction isolation**: The degree of isolation of this transaction with other transactions.

**Transaction propagation**: Defines the scope of the transaction.

**Read-only status**: A read-only transaction will not modify any data. Read-only transactions can be useful for optimization in some cases.

**Transaction timeout**: How long a transaction can run before timing out.

Let us understand these attributes in detail.

**Transaction Isolation**

The issues that occur during concurrent data access are as follows:

| Issue | Description |
|---|---|
| Dirty read(uncommitted dependency) | A dirty read occurs when the transaction is allowed to read data from a row that has been modified by another running transaction that is not yet committed |
| Non-Repeatable Reads | A non-Repeatable read is the one in which data read happens twice inside the same transaction and cannot be guaranteed to contain the same value. |
| Phantom Reads | A Phantom read occurs when two identical queries are executed and the collection of rows returned by the second query is different from the first |

The various isolation levels supported to handle concurrency issues are as follows:

| Isolation Level | Description |
|---|---|
| DEFAULT | The Application uses the default isolation level of the database |
| READ_UNCOMMITED | Read changes that are not committed |
| READ_COMMITED | Allows concurrent committed reads |
| REPEATABLE_READ | Allows multiple reads by the same transaction. Prevents dirty/non-repeatable read from other transaction |
| SERIALIZABLE | Allows maximum serializability |

**UPDATE OPERATION IN SPRING DATA JPA:**

Now that you know, how to use @Query annotation for query operations.

Can @Query annotation be used for performing modification operations?

Yes, it can execute modifying queries such as update, delete or insert operations using @Query annotation along with @Transactional and @Modifying annotation at query method.

**Example:** Interface with a method to update the name of a customer based on the customer's address.

```
1. public interface CustomerRepository extends
   JpaRepository<Customer, Long> {
2.      @Transactional
3.      @Modifying(clearAutomatically = true)
4.      @Query("update Customer c set c.name = ?1 where
   c.address = ?2")
5.      void update(String name, String address);
6. }

7.
```

Let us now understand in detail, why there is a need for the following:

**@Modifying:** This annotation will trigger @Query annotation to be used for an update operation instead of a query operation.

**@Modifying(clearAutomatically = true)**: After executing modifying query, EntityManager might contain unrequired entities. It is a good practice to clear the EntityManager cache automatically by setting @Modifying annotation's clearAutomatically attribute to true.

**@Transactional:** Spring Data provided CRUD methods on repository instances that support transactional by default with read operation and by setting readOnly flag to true. Here, @Query is used for an update operation, and hence we need to override default readOnly behavior to read/write by explicitly annotating a method with @Transactional.

**@Modifying:** This annotation triggers the query annotated to a particular method as an updating query instead of a selecting query. As the EntityManager might contain outdated entities after the execution of the modifying query, we should clear it. This effectively drops all non-flushed changes still pending in the EntityManager. If we don't wish the EnetiyManager to be cleared automatically we can set @Modifying annotation's clearAutomatically attribute to false.

Fortunately, starting from Spring Boot 2.0.4.RELEASE, Spring Data added **flushAutomatically** flag to auto flush any managed entities on the persistence context before executing the modifying query.

Thus, the safest way to use **Modifying** is:

```
1. @Modifying(clearAutomatically=true,

   flushAutomatically=true)
```

Now let's take a small scenario where we don't use these two attributes with **@Modifying** annotation:

Assume a Customer table with two columns name and active, exists in the database with only one row as Tom, true.

**Repository:**

```
1. public interface CustomerRepository extends
   JPARepository<Customer, Intger> {
2.    @Modifying
3.    @Query("delete Customer c where c.active=0")
4.    public void deleteInActiveCustomers();
5.

6. }
```

In the above repository, @Modifying annotation is used without clearAutomatically and flushAutomatically attributes. So let's visualize what happens when the Service call happens.

Visualization -1: If flushAutomatically attribute is not used in the Repository:

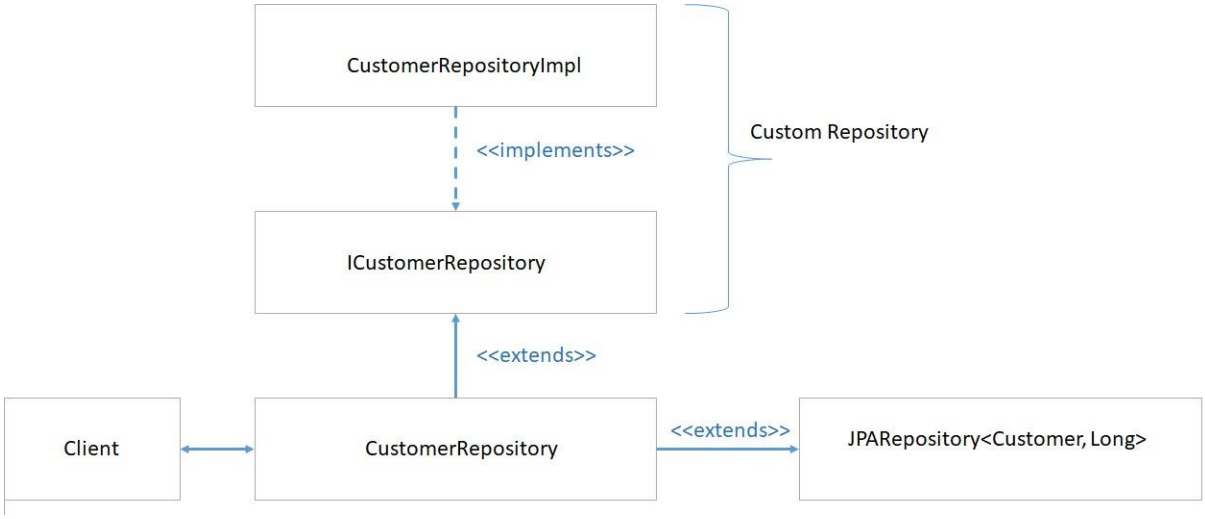
```
1. public class CustomerServiceImpl implements
   CustomerService {
2.         Customer customerTom =
   customerRepository.findById(1); // Stored in the First
   Level Cache
3.         customerTom.setActive(false);
4.         customerRepository.save(customerTom);
5.
6.         customerRepository.deleteInActiveUsers();// By
   all means it won't delete the customerTom
7.
8.         /*customerTom still exist since customerTom with
   'active' attribute being set to false was not flushed
   into the database when @Modifying kicks in*/

9.     }
```

Visualization -2: If **clearAutomatically** attribute is not used in the Repository:

```
1. public class CustomerServiceImpl implements
   CustomerService {
2.        Customer customerTom =
   customerRepository.findById(1); // Stored in the First
   Level Cache
```

```
3.        customerRepository.deleteInActiveCustomers(); //
  We think that customerTom is deleted now
4.
  System.out.println(customerRepository.findById(1).isPrese
  nt()) // Will return TRUE
5.        System.out.println(customerRepository.count()) //
  Will return 1
6.
7.        // TOM still exist in this transaction persistence
  context
8.        // TOM's object was not cleared upon @Modifying
  query execution
9.        // TOM's object will still be fetched from First
  Level Cache
10.         /* clearAutomatically attribute takes care of
  doing the clear part on the objects being modified for
  current transaction persistence context*/
11. }
```

**CUSTOMER REPO IMPLEMENT:**




So far, we have seen different approaches to create queries.

Sometimes, customization of a few complex methods is required. Spring easily support this, you can use custom repository code and integrate it with Spring Data abstraction.

Let us look at how to define the custom repository in detail.

**Example:** Let us consider a customer search scenario wherein customer details needs to be fetched based on name and address or gender or age.

Entity class Customer is shown below:

```
@Entity
public class Customer {

  @Id
  private Long phoneNumber;
  private String name;
  private Integer age;
  private Character gender;
  private String address;
        private Integer planId;
      -------

 }
```

The steps to implement the custom repository with a method to Retrieve customer records based on search criteria are as follows:

Step 1: Define an interface and an implementation for the custom functionality.

```
public interface ICustomerRepository {

  public List<Customer> searchCustomer(String name, String
addr, Character gender, Integer age);
}


```

Step 2: Implement this interface using repository class as shown below:

```
public class CustomerRepositoryImpl implements
ICustomerRepository{

  private EntityManagerFactory emf;

  @Autowired
  public void setEntityManagerFactory(EntityManagerFactory
emf) {
        this.emf = emf;
  }

        @Override
        public List<Customer> searchCustomer(String name,
String address, Character gender, Integer age) {
              EntityManager em =
emf.createEntityManager();
```

```
13.             CriteriaBuilder builder =
    em.getCriteriaBuilder();
14.
15.             CriteriaQuery<Customer> query =
    builder.createQuery(Customer.class);
16.             Root<Customer> root =
    query.from(Customer.class);
17.
18.             Predicate cName =
    builder.equal(root.get("name"), name);
19.             Predicate cAddress =
    builder.equal(root.get("address"), address);
20.
21.             Predicate exp1 = builder.and(cName,
    cAddress);
22.
23.             Predicate cGender =
    builder.equal(root.get("gender"), gender);
24.             Predicate cAge =
    builder.equal(root.get("age"), age);
25.
26.             Predicate exp2 = builder.or(cGender,cAge);
27.
28.             query.where(builder.or(exp1, exp2));
29.
30.             return
    em.createQuery(query.select(root)).getResultList();
31.         }

32.
```

Step 3: Define repository interface extending the custom repository interface as shown below:

```
1. public interface CustomerRepository extends
   JpaRepository<Customer, Long>,ICustomerRepository{
2.
3. }

4.
```

Now, standard repository interface CustomerRepository extends both JpaRepository and the custom interface(ICustomerRepository). Hence, all the Spring data provided default methods, as well as the custom defined method(searchCustomer), will be accessible to the clients.

**BEAT PRACTICES- SPRING DATA JPA:**

Let us discuss the best practices which need to be followed as part of the Quality for Spring Data JPA applications. Once these best practices are applied they can help in improving the performance in JPA implementations.

**Best Practices:**

- Extended interface usage
- Don't fetch more data than you need
- @NamedQuery vs @Query
- JPQL in Custom Repository

Let us understand the reason behind these recommendations and their implications.

A Spring Data JPA developer can choose the most common way i.e. to extend the appropriate interface from the spring data jpa module. After inheriting the appropriate interface, it immediately provides the basic CRUD operations.

A developer can choose specific method names for parsing queries, which is one of the special features of Spring Data JPA.

But when the method names become more complex it's convenient to use HQL and native SQL.

Let's take a small example to demonstrate the above best practice.

```
1. @Repository
2. public interface OrderRepository extends
   JpaRepository<Order, Integer> {
3.
4.      // HQL
5.      @Query("SELECT "
6.     + "  DISTINCT o "
7.     + "FROM "
8.     + "  Order o "
9.     + "INNER JOIN "
10.      + "  o.sender.friends f "
11.      + "WHERE "
12.      + "  (p.sender = ?1 OR f = ?1) "
13.      + "AND p.dateCreated < ?2")
14.      Page<Post>
   findAllBySenderOrRecieverAndDateCreated(Person person
15.                                                , Date
   dateCreated
16.                                                ,
   Pageable page);
17.
18.
```

```
19. }
```

- Pagination can be directly used within the database. You can specify and fetch the required no. of rows.
- ***setFirstResult(int)***: Offset index is set to start the Pagination
- ***setMaxResults(int)***: Maximum number of entities can be set which should be included in the page

Column Select:

- Fetch the required columns for the select query. If only Customer's name is required, then get only that.