Unit-3
Spring Data JPA with Boot

What is an Environment?

An Environment is a collection of software such as operating system, database and hardware tools such as RAM, expansion cards that help to build software. Every enterprise application will have multiple environments.

Example:

- Development
- Testing
- Staging
- Production

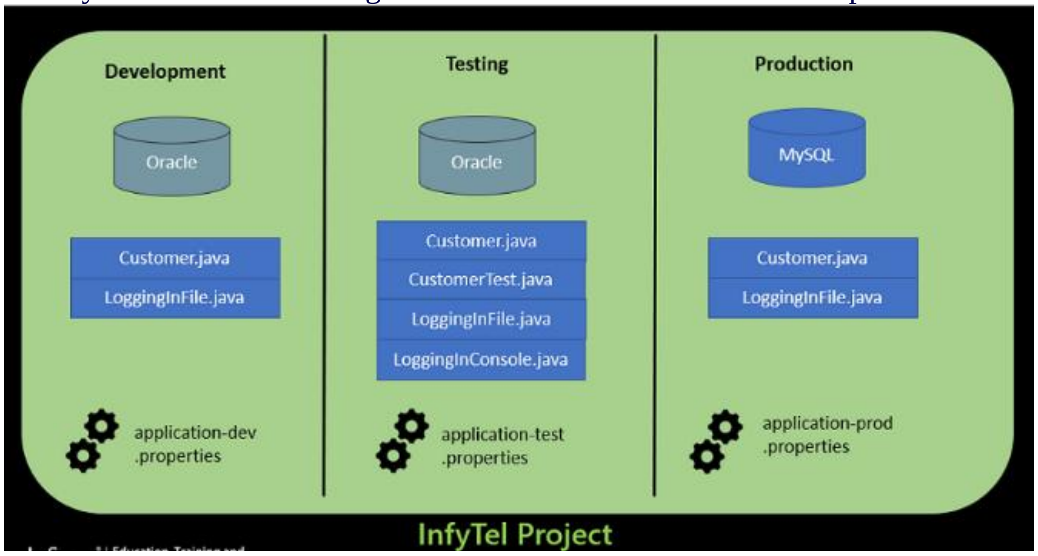
Why Spring Profiles?

Analyze the below requirements:

Requirement1: InfyTel a telecom application has to use Oracle database in development and MySQL database in production environment. So, developer has two different application.properties file according to the configuration needed in development and production.

How will spring read the appropriate application.properties files depending on the environment?

Requirement2: The project requires logging levels and logging properties (i.e., files/console output) to be set differently in testing environment and production environment.



The above two requirements can be addressed easily using **Spring Profiles.**

What is Spring Profile?

Spring Profiles helps to classify the classes and properties file for the environment. You can create multiple profiles and set one or more profiles as the active profile. Based on the active profile spring framework chooses beans and properties file to run.

Let us see how to configure different profiles using spring profiles in our projects.

Steps to be followed:

1) Identify the beans which has to be part of a particular profile [Not mandatory]
2) Create environment-based properties file
3) Set active profiles.

You will see each step in-detail.

How to declare Spring Profile?

You can identify the Spring beans which have to be part of a profile using the following ways

1)Annotation @Profile
2) Bean declaration in XML

Note: In this course, the annotation based approach is covered.

@Profile

@Profile helps spring to identify the beans that belong to a particular environment.

1. Any class which is annotated with stereotype annotations such as @Component,@Service,@Repository and @Configuration can be annotated with @Profile .
2. @Profile is applied at class level except for the classes annotated with @Configuration where @Profile is applied at the method level.

@Profile- Class level:

```
@Profile("dev")
@Component
@Aspect
public class LoggingAspect { }
```

This LoggingAspect class will run only if "dev" environment is active.

@Profile- Method level:

```
@Configuration
public class SpringConfiguration {
@Bean("customerService")
@Profile("dev")
public CustomerService customerServiceDev() {
CustomerService customerServiceDev= new CustomerService();
customerServiceDev.setName("Developement-Customer");
return customerServiceDev;
}
@Bean("customerService")
@Profile("prod")
public CustomerService customerServiceProd() {
CustomerService customerServiceProd=new CustomerService();
customerServiceProd.setName("Production-Customer");
return customerServiceProd;
}
}
```

In the above code snippet CustomerService bean is configured differently in the "dev" and "prod" environments. Depending on the currently active profile Spring fetches CustomerService accordingly.

Note: @Profile annotation in class level should not be overridden in the method level. It will cause “NoSuchBeanDefinitionException”.

@Profile value can be prefixed with !(Not) operator.

```
@Profile("!test")
@Configuration
@ComponentScan(basePackages="com.infy.service")
public class SpringConfiguration { }
```

This SpringConfiguration class will run in all environments other than test.

Note:
if you do not apply @profile annotation on a class, means that the particular bean is available in all environment(s).

Creating environment-based properties files for Spring Boot projects

Spring Boot relies on application.properties file for configuration. To write configuration based on environment, you need to create different application.properties files. The file name has a naming convention as application-<user created profile name>.properties
Example:
application-dev.properties

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1522:devDB
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver.class=oracle.jdbc.driver.OracleDriver
# logging
logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%
logging.level.org.springframework.web: ERROR
```

application-test.properties

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1522:testDB
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver.class=oracle.jdbc.driver.OracleDriver
# logging
logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n
logging.level.org.springframework.web: DEBUG
```

application-prod.properties

```
# mysql settings
spring.datasource.url=jdbc:mysql://localhost:3306/prodDB?useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver.class=com.mysql.jdbc.Driver
# logging
logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%
logging.level.org.springframework.web: INFO
```

Note: The above shows how application.properties file with different configuration is created for 3 different environments(dev,test and prod)

How to set active Profiles?

By now you are aware of how to map a bean or configure properties to a certain profile(eg :dev/prod/test), next we need to set the one which is active..
The different way to do is as follows:

1) application.properties
2) JVM System Parameter
3) Maven Profile

Note :You can make more than 1 profile as active at a time.

Set active profile - application.properties

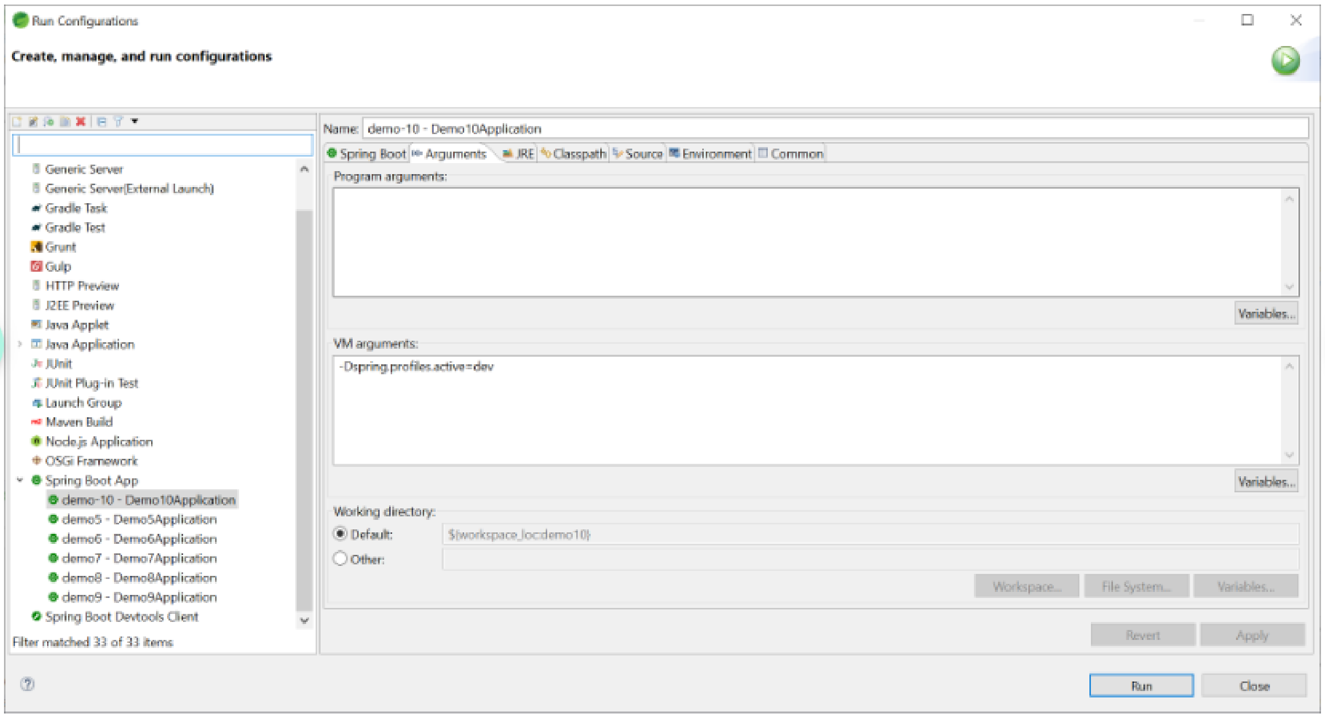
To set an active profile you need to write the below property in the main application.properties file.

```
spring.profiles.active=dev

        OR

spring.profiles.active=dev,prod
```

Set active profile - JVM System Parameter

You can set profile using JVM system arguments in two ways , either set the VM arguments in Run configuration or programmatic configuration.
To set through Run configuration give the below command in VM arguments.

```
-Dspring.profiles.active=dev
```



Run Configurations

Create, manage, and run configurations

Generic Server
Generic Server(External Launch)
Gradle Task
Gradle Test
Grunt
Gulp
HTTP Preview
J2EE Preview
Java Applet
Java Application
JUnit
JUnit Plug-in Test
Launch Group
Maven Build
Node.js Application
OSGi Framework
Spring Boot App
demo-10 - Demo10Application
demo5 - Demo5Application
demo6 - Demo6Application
demo7 - Demo7Application
demo8 - Demo8Application
demo9 - Demo9Application
Spring Boot Devtools Client
Filter matched 33 of 33 items

Name: demo-10 - Demo10Application

Spring Boot | Arguments | JRE | Classpath | Source | Environment | Common

Program arguments:

VM arguments:
-Dspring.profiles.active=dev

Working directory:
Default: ${workspace_loc:demo10}
Other:

To set through programmatic approach set system property as follows,

```
public static void main( String[] args )    {
System.setProperty(AbstractEnvironment.ACTIVE_PROFILES_PROPERTY_NAME, "prod");}
```

Set active profile - Maven

To set a profile using maven, follow the below steps.

Step 1: Add <profiles> tag to pom.xml file as follows.

```
<profiles>
<profile>
<id>dev</id>
<properties>
<spring.profiles.active>dev</spring.profiles.active>
</properties>
</profile>
<profile>
<id>test</id>
<activation>
<activeByDefault>true</activeByDefault>
</activation>
<properties>
<spring.profiles.active>test</spring.profiles.active>
</properties>
</profile>
```

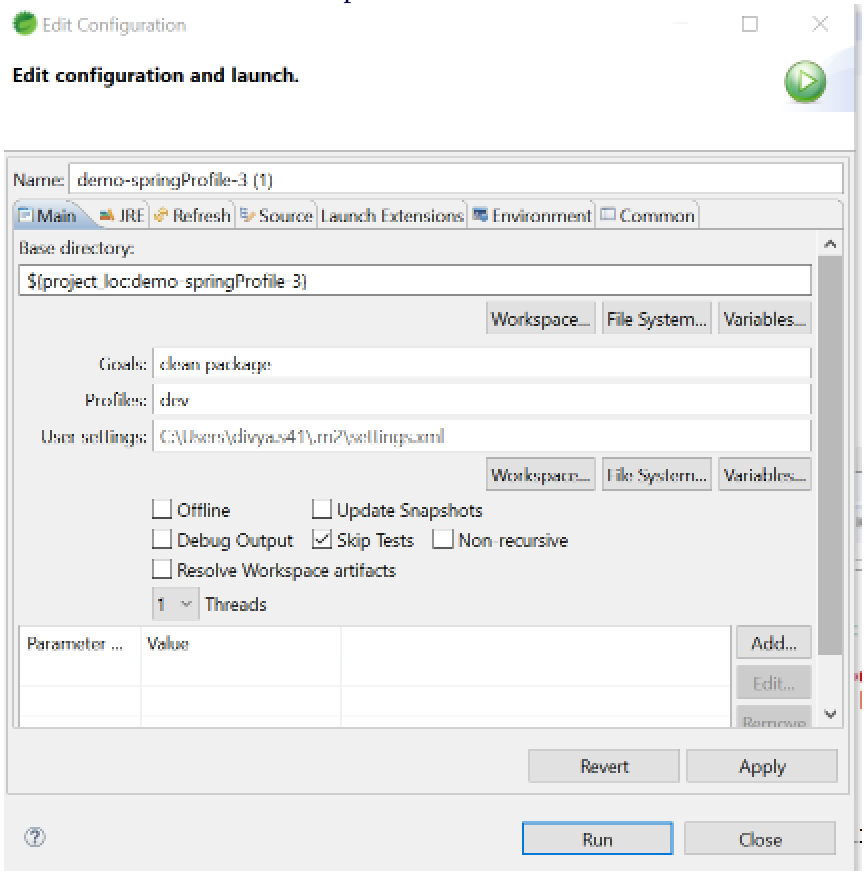
```
</profiles>
<build>
<resources>
<resource>
<directory>src/main/resources</directory>
<filtering>true</filtering>
</resource>
</resources>
</build>
```

Step 2: Add below to the application.properties file.

```
spring.profiles.active=@spring.profiles.active@
```

Step 3: Build the project as guided below,

i. Run As -> Maven build
ii. Goals as clean package
iii. Profiles as dev
iv. Check Skip Tests



Name: demo-springProfile-3 (1)

Base directory: ${project_loc:demo-springProfile-3}

Goals: clean package

Profiles: dev

User settings: C:\Users\divya.s41\.m2\settings.xml

☐ Offline ☐ Update Snapshots
☐ Debug Output ☑ Skip Tests ☐ Non-recursive
☐ Resolve Workspace artifacts
1 Threads

Parameter ...	Value

About Spring Data JPA

Spring is a popular open-source Java application development framework created by Rod Johnson. Spring supports developing any kind of Java application such as standalone applications, web applications, database-driven applications, and many more.

For implementing the Persistence Layer of an enterprise application Spring uses many approaches. A developer can choose appropriate Spring modules such as Spring JDBC, Spring ORM, or Spring Data JPA for implementing the data access layer of an enterprise application depending on the repository type. Spring simplifies transactions by allowing the developer to implement transactions in a declarative way using simple annotations and configurations. For the more fine-grained transactions, Spring also supports programmatic transactions.

Configuring a Spring application is a tedious job. Even though it provides flexibility in bean configuration with multiple ways such as XML, annotation, and Java-based configurations, there is no escape from the configuration. Configuring the Spring features may need more time for developers and may distract the developers from solving business problems.

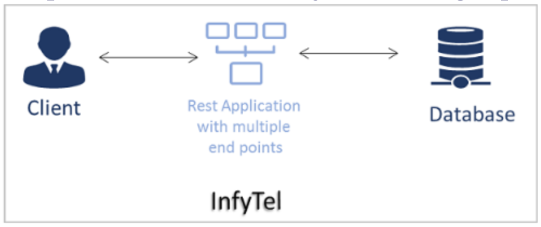
Thanks to Spring Team for releasing Spring Boot, one of the topmost innovations in all existing Spring Framework. Spring Boot provides a new prototype for developing Spring applications with nominal effort. Using Spring Boot, you will be able to develop Spring applications with extra agility and capacity to focus on solving business needs with nominal (or possibly no) thought of configuring Spring itself.

In this course, we will focus on how to implement the data access layer of an enterprise application by using Spring Data JPA with Spring Boot.

Scenario:

InfyTel Application

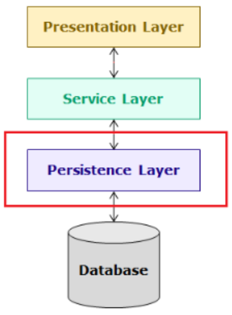
In this course, we will look at a telecom application called InfyTel. InfyTel application is a REST-based application which provides functionality for adding, updating, and deleting customer and their call plans.



Note: In InfyTel REST, the development of REST endpoints will be covered in the Spring REST course. InfyTel is built as a three-tier application that consists of

- Presentation Layer
- Service/Business Layer
- Persistence Layer

In Spring Core with Spring Boot course, we have seen how to develop the Service/Business layer for this application. In this course, we will build the Persistence layer using Spring Data JPA with Spring Boot. Once you have completed this course you can then learn to build the REST Presentation layer using the Spring REST course.

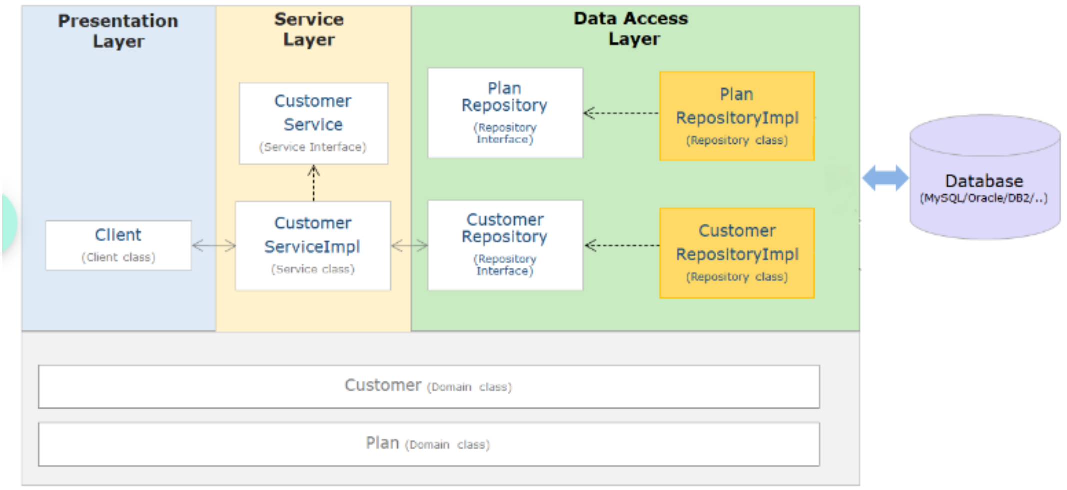


Let's look at the high-level design of the InfyTel application.

InfyTel application has to maintain its Customer and Plan details. The application should support the following features:

- Add Customer
- Edit Customer Details
- Search Customer
- View all Customer
- Remove Customer

This helps in developing an application using multi-tier architecture as shown below:


Presentation Layer
Service Layer
Data Access Layer
Customer Service (Service Interface)
Plan Repository (Repository Interface)
Plan RepositoryImpl (Repository class)
Client (Client class)
Customer ServiceImpl (Service class)
Customer Repository (Repository Interface)
Customer RepositoryImpl (Repository class)
Database (MySQL/Oracle/DB2/..)
Customer (Domain class)
Plan (Domain class)

The following table describes the files required for developing an InfyTel Application.

Required Files	**Description**
Client Application	It contains code to interact with application services
CustomerService interface CustomerServiceImpl class	An interface for client and repository Provides the implementation of CustomerService interface with appropriate logic
PlanService Interface PlanServiceImpl Class	An interface for client and repository Provides the implementation of PlanService interface with appropriate logic
CustomerRepository interface	An interface between the service layer and data layer
PlanRepository interface	An interface between the service layer and data layer
Customer Class	An entity class to represent Customer details
Plan Class	An entity class to represent Plan details
CustomerDTO Class	A DTO class that represents Customer details
PlanDTO Class	A DTO class that represents Plan details

Note: It is optional to use interfaces in the service and data access layers. However, programming with an interface is a recommended practice.

InfyTel Customer - Insert & Delete using JDBC API: Consider the InfyTel application. The Admin of the InfyTel application has to maintain its Customer and Plan details. In this course, we will see how to implement the persistence layer of the InfyTel application using Spring Data JPA.

First, let us look at how to add the below functionalities for the InfyTel Customer scenario using JDBC API.

- Add Customer - Adding customer records
- Remove Customer - Deleting customer records

Required steps

The following are the steps to implement the previous requirements of the InfyTel application.
Step 1: Define an entity class Customer.java to represent customer details.
Step 2: Add an appropriate database connector dependency in pom.xml.
Step 3: Create a data access layer with an interface CustomerDAO and a repository class CustomerDAOImpl to implement the interface.
Step 4: Create a service layer with an interface CustomerService and CustomerServiceImpl class to implement the interface
Step 5: Create a presentation layer with client class to access database operations and display details on the console
Step 6: Create a table Customer in the database.

```
create table customer(
phone_no bigint primary key,
name varchar(50),
age integer,
gender char(10),
address varchar(50),
plan_id integer
);
```

Note:
The code given in Step 6 is to create a table for the MySQL database and needs to be modified based on the database used in your application.
Let us see the complete implementation of this requirement with the help of a demo application with JDBC API on the coming page.

InfyTel Customer scenario using JDBC API

Demo 1: InfyTel Application development using JDBC API
Highlights:

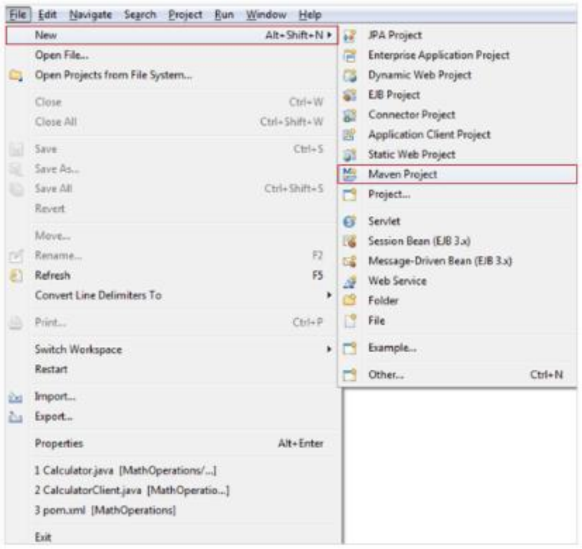
- To perform insert operation using JDBC
- To perform delete operation using JDBC

Consider the InfyTel scenario and create an application to perform the following operations using JDBC.

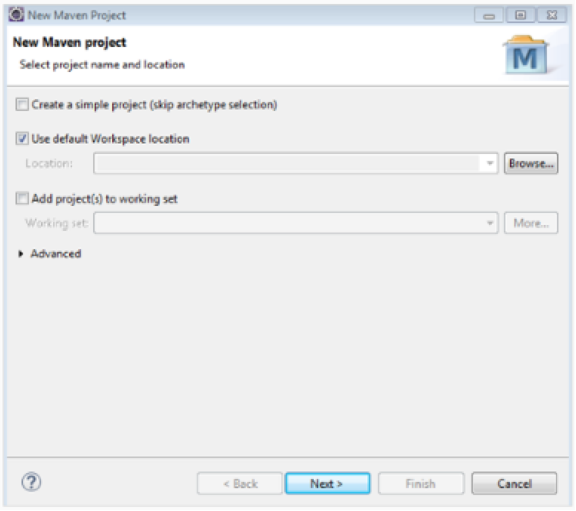
- Insert Customer details
- Delete Customer for given PhoneNo

Steps to implement the application:
Now, let us understand the steps required to create a Maven project for Spring Data JPA application using Spring Tool Suite(STS)/Eclipse IDE
Note: Screenshots of this demo have been taken from Eclipse IDE. However, the steps remain similar even for the Spring Tool Suite IDE.
Step 1: Create a Maven project in Eclipse as shown below

Go to File > new > Maven Project, you would get the below screen



Step 2: Let the current workspace be chosen as the default location for the Maven project. Click on next.


New Maven Project

New Maven project

Select project name and location

[ ] Create a simple project (skip archetype selection)

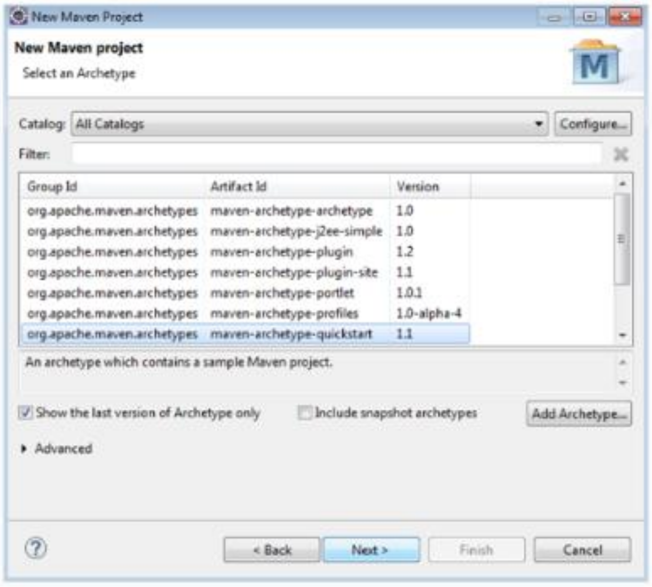
[✓] Use default Workspace location

Location:

[ ] Add project(s) to working set

Working set:

Step 3: Choose maven-archetype-quickstart


New Maven Project

New Maven project

Select an Archetype

Catalog: All Catalogs

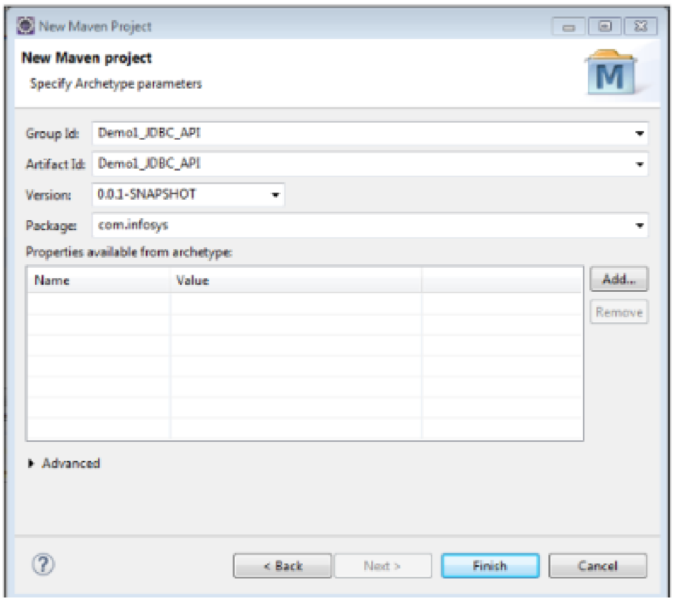
Filter:

Group Id	Artifact Id	Version
org.apache.maven.archetypes	maven-archetype-archetype	1.0
org.apache.maven.archetypes	maven-archetype-j2ee-simple	1.0
org.apache.maven.archetypes	maven-archetype-plugin	1.2
org.apache.maven.archetypes	maven-archetype-plugin-site	1.1
org.apache.maven.archetypes	maven-archetype-portlet	1.0.1
org.apache.maven.archetypes	maven-archetype-profiles	1.0-alpha-4
org.apache.maven.archetypes	maven-archetype-quickstart	1.1

An archetype which contains a sample Maven project.

[✓] Show the last version of Archetype only [ ] Include snapshot archetypes

Step 4: Provide groupId as Demo1_JDBC, artifactId as Demo1_JDBC_API and retain the default version which is 0.0.1-SNAPSHOT.
We can customize the package names as per our needs. In this demo, the package name is provided as com.infytel.
Click on the finish button to complete the project creation.

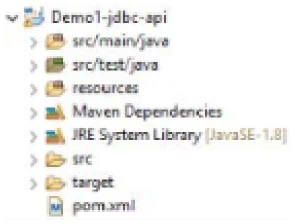


New Maven project
Specify Archetype parameters

Group Id: Demo1_JDBC_API
Artifact Id: Demo1_JDBC_API
Version: 0.0.1-SNAPSHOT
Package: com.infosys
Properties available from archetype:

Name	Value

Step 5: A new project will be created as shown below with the pom.xml file



```
Demo1-jdbc-api
  src/main/java
  src/test/java
  resources
  Maven Dependencies
  JRE System Library [JavaSE-1.8]
  src
  target
  pom.xml
```

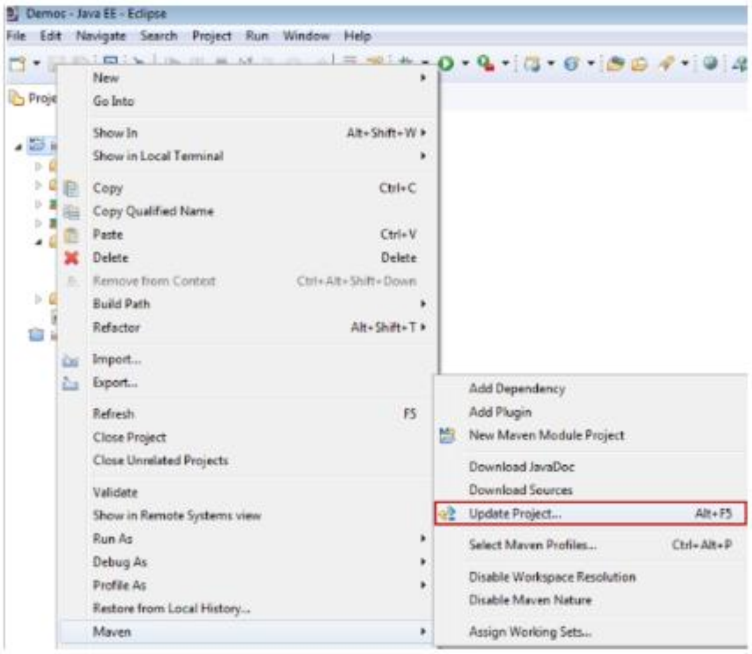
+

Step 6: Add the following dependencies in the pom.xml file.

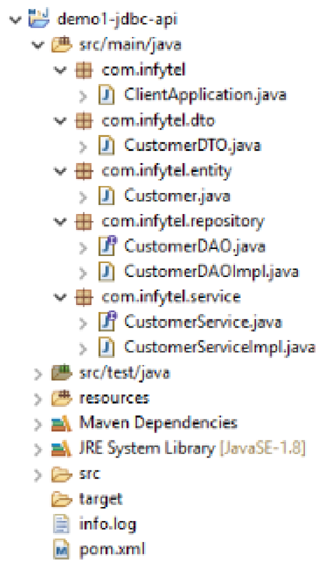
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>Demo1_JDBC_API</groupId>
<artifactId>Demo1_JDBC_API</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>Demo1_JDBC_API</name>
<url>http://maven.apache.org</url>
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>1.8</maven.compiler.target>
```

```
</properties>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.19</version>
</dependency>
</dependencies>
</project>
```

Step 7: If the dependencies are not getting downloaded automatically update the Maven project as shown below



Step 8: Add the required project files in the respective packages as shown below.


```
demo1-jdbc-api
  src/main/java
    com.infytel
      ClientApplication.java
    com.infytel.dto
      CustomerDTO.java
    com.infytel.entity
      Customer.java
    com.infytel.repository
      CustomerDAO.java
      CustomerDAOImpl.java
    com.infytel.service
      CustomerService.java
      CustomerServiceImpl.java
  src/test/java
  resources
  Maven Dependencies
  JRE System Library [JavaSE-1.8]
  src
  target
  info.log
  pom.xml
```

Step 9: Write the below code in the "Customer.java" file

```
package com.infytel.entity;
public class Customer {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String address,
Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
}
```

Step 10: Write the below code in "CustomerDTO.java":

```
package com.infytel.dto;
import com.infytel.entity.Customer;
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
public CustomerDTO() {}
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String
address, Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
```

```
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static Customer prepareCustomerEntity(CustomerDTO customerDTO)
Customer customerEntity = new Customer();
customerEntity.setPhoneNumber(customerDTO.getPhoneNumber());
customerEntity.setName(customerDTO.getName());
customerEntity.setGender(customerDTO.getGender());
customerEntity.setAge(customerDTO.getAge());
customerEntity.setAddress(customerDTO.getAddress());
customerEntity.setPlanId(customerDTO.getPlanId());
return customerEntity;
}
}
```

Step 11: Write the below code in an interface “CustomerDAO.java”:

```
package com.infytel.repository;
import com.infytel.entity.Customer;
public interface CustomerDAO {
// Method to insert a Customer record into the db
public void insert(Customer customer);
// Method to remove a Customer record from the db
public int remove(Long phoneNo);
}
```

Step 12: Write below code in “CustomerDAOImpl.java” as below:

```
package com.infytel.repository;
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Properties;
import org.apache.log4j.Logger;
import com.infytel.entity.Customer;
```

```
public class CustomerDAOImpl implements CustomerDAO {
static Logger logger = Logger.getLogger(CustomerDAOImpl.class);
Connection con = null;
PreparedStatement stmt = null;
ResultSet rs = null;
@Override
public void insert(Customer customer) {
try (FileInputStream fis = new FileInputStream("resources/application.properties");) {
Properties p = new Properties();
p.load(fis);
String dname = (String) p.get("JDBC_DRIVER");
String url = (String) p.get("JDBC_URL");
String username = (String) p.get("USER");
String password = (String) p.get("PASSWORD");
Class.forName(dname);
// Register driver
con = DriverManager.getConnection(url, username, password);
// Create connection
String query = "insert into customer values (?,?,?,?,?,?)";
// Create prepared statement
stmt = con.prepareStatement(query);
stmt.setLong(1, customer.getPhoneNumber());
stmt.setString(2, customer.getName());
stmt.setInt(3, customer.getAge());
stmt.setString(4, customer.getGender().toString());
stmt.setString(5, customer.getAddress());
stmt.setInt(6, customer.getPlanId());
// Execute query
stmt.executeUpdate();
logger.info("Record inserted");
} catch (Exception e) {
logger.error(e.getMessage(), e);
} finally {
try {
// Close the prepared statement
stmt.close();
// Close the connection
con.close();
} catch (Exception e) {
logger.error(e.getMessage(), e);
}
}
}
@Override
public int remove(Long phoneNo) {
int result = 1;
try (FileInputStream fis = new FileInputStream("resources/application.properties");) {
Properties p = new Properties();
p.load(fis);
String dname = (String) p.get("JDBC_DRIVER");
String url = (String) p.get("JDBC_URL");
String username = (String) p.get("USER");
String password = (String) p.get("PASSWORD");
Class.forName(dname);
// Create connection
con = DriverManager.getConnection(url, username, password);
String query = "DELETE FROM Customer WHERE phone_no = ?";
// Create prepared statement
stmt = con.prepareStatement(query);
stmt.setLong(1, phoneNo);
// Execute query
result = stmt.executeUpdate();
} catch (Exception e) {
logger.error(e.getMessage(), e);
} finally {
try {
// Close the prepared statement
stmt.close();
// Close the connection
con.close();
} catch (Exception e) {
logger.error(e.getMessage(), e);
}
}
return result;
}
}
```

Step 13: Write the below code in an interface “CustomerService.java”:

```
package com.infytel.service;
import com.infytel.dto.CustomerDTO;
public interface CustomerService {
// Method to access the repository layer method to insert Customer record
public void insert(CustomerDTO customer);
// Method to access the repository layer method to delete Customer record
public int remove(Long phoneNo);
}
```

Step 14: Write the below code in a class "CustomerServiceImpl.java":

```
package com.infytel.service;
import com.infytel.dto.CustomerDTO;
import com.infytel.repository.CustomerDAO;
import com.infytel.repository.CustomerDAOImpl;
public class CustomerServiceImpl implements CustomerService {
CustomerDAO customerDAO = new CustomerDAOImpl();
@Override
public void insert(CustomerDTO customer) {
customerDAO.insert(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public int remove(Long phoneNo) {
return customerDAO.remove(phoneNo);
}
}
```

Step 15: Right-click on the Project -> New -> Source Folder -> Name the folder as resources -> Create a new file in th resources folder -> Name it as application.properties -> Paste the below configuration in application.properties.

```
JDBC_DRIVER=com.mysql.cj.jdbc.Driver
JDBC_URL=jdbc:mysql://localhost:3306/ormdb
USER=root
PASSWORD=root
```

Step 16: Under the same resources create a new file, Name it is log4j.properties, and paste the below configuration in log4j.properties.

```
# Root logger
log4j.rootLogger=INFO, file,console
# Direct log messages to a log file
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=info.log
log4j.appender.file.MaxFileSize=10MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%t] %-5p %c %x - %m%n
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.consoleAppender.layout.ConversionPattern=[%t] %-5p %c %x - %m%n
```

Step 17: Write the below code in a class "ClientApplication.java":

```
package com.infytel;
```

```
import java.util.Scanner;
import org.apache.log4j.Logger;
import com.infytel.dto.CustomerDTO;
import com.infytel.service.CustomerService;
import com.infytel.service.CustomerServiceImpl;
public class ClientApplication {
static Logger logger = Logger.getLogger(ClientApplication.class);
public static void main(String[] args) {
CustomerService customerService = new CustomerServiceImpl();
CustomerDTO customer = new CustomerDTO(7022713722L, "Lucy", 27, 'F', "INDIA", 3);
customerService.insert(customer);
logger.info("Records are successfully added..");
System.out.println("Enter the phone Number of the Customer which has to be deleted.");
Scanner scanner = new Scanner(System.in);
Long phoneNo = scanner.nextLong();
int result = customerService.remove(phoneNo);
if (result == 1) {
logger.info("Success : Record deleted successfully ");
} else {
logger.info("Error : No record found for the given phone number ");
}
scanner.close();
}
}
```

Expected Output on the console:

```
Records are successfully added..
Enter the phone Number of the Customer which has to be deleted.
7022713722
Success : Record deleted successfully
```

Note: Verify updated changes in the database table

Drawbacks of JDBC API

InfyTel Application's data access layer has been implemented using JDBC API. But there are some limitations of using JDBC API.

As seen in the demo when JDBC API is used for database operation, the developer needs to provide a lot of data access code before and after the execution of a query. This increases the code size and it becomes difficult to maintain and make changes.

All the exceptions thrown in JDBC have checked exceptions that require try-catch blocks in the code. This also makes the code difficult to read and maintain.

If the database connection is not closed in an application, it will lead to resource leakage.

Let us understand the limitations of JDBC API

- A developer needs to open and close the connection.
- A developer has to create, prepare, and execute the statement and also maintain the resultset.
- A developer needs to specify the SQL statement(s), prepare, and execute the statement.
- A developer has to set up a loop for iterating through the result (if any).
- A developer has to take care of exceptions and handle transactions.

The above limitations can be solved with the technologies Spring ORM.

So Let's discuss more on ORM and also let's see some classes and interfaces of **Spring ORM** which will help to overcome the above limitations. Later we'll look at some features of Spring ORM which unnecessarily increase our repository implementations and will see how **Spring Data JPA** will help us to completely remove the repository class implementations by using Spring ORM specifications internally.

Need for ORM

To understand Spring Data JPA we need to know about ORM and Spring ORM

As seen in the previous demo of the InfyTel application where the persistence layer has been implemented using JDBC API, there are several limitations and challenges. Let us now look at them.

JDBC, I/O, Serialization do not solve the problem of data persistence effectively. For a medium to be effective, it needs to take care of the fundamental difference in the way Object-Oriented Programs(OOP) and RDBMS deals with the data.

- In Programming languages like Java, the related information or the data will be persisted in the form of hierarchical and interrelated objects.
- In the relational database, the data is persisted as table format or relations.

The greatest challenge in integrating the concepts of RDBMS and OOP is a mapping of the Java objects to databases. When object and relational paradigms work with each other, a lot of technical and conceptual difficulties arise, as mapping of an object to a table may not be possible in all the contexts. Storing and retrieving Java objects using a Relational database exposes a paradigm mismatch called "Object-Relational Impedance Mismatch". These differences are because of perception, style, and patterns involved in both the paradigms that lead to the following paradigm mismatches:

- **Granularity**: Mismatch between the number of classes in the object model and the number of tables in the relational model.
- **Inheritance or Subtype**: Inheritance is an object-oriented paradigm that is not available in RDBMS.
- **Associations**: In object-oriented programming, the association is represented using reference variables, whereas, in the relational model foreign keys are used for associating two tables.
- **Identity**: In Java, object equality is determined by the "==" operator or "equals()" method, whereas in RDBMS, uses the primary key to uniquely identify the records.
- **Data Navigation**: In Java, the dot(.) operator is used to travel through the object network, whereas, in RDBMS join operation is used to move between related records.

What is ORM?

Object Relational Mapping (ORM) is a technique or design pattern, which maps object models with the relational model. It has the following features:

- It resolves the object-relational impedance mismatch by mapping
  - Java classes to tables in the database
  - Instance variables to columns
  - Objects to rows in the table
- It helps the developer to get rid of SQL queries. They can concentrate on the business logic and work with the object model which leads to faster development of the application.
- It is database independent. All database vendors provide support for ORM. Hence, the application becomes portable without worrying about the underlying database.

Benefits of Object Relational Mapping(ORM)

- ORM provides a programmatic approach to implement database operations.
- ORM maps Java objects to the relational database tables in an easier way based on simple configuration.
- Supports simple query approaches like HQL(Hibernate Query language) and JPQL (Java Persistence Query Language)
- Supports object-oriented concepts such as inheritance, mapping, etc.

To use ORM in Java applications Java Persistence API (JPA) specification is used. There are several implementations of JPA available in the market, such as Hibernate, OpenJPA, DataNucleus, EclipseLink, etc. EclipseLink is the reference implementation for JPA.

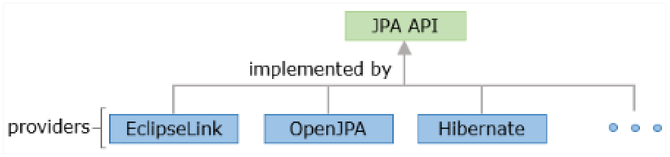
Note: Hibernate is one of the most popular implementations and in this course, our Spring Data JPA applications internally use this implementation.

ORM Providers

The Java Persistence API (JPA) is a Java EE specification that defines how data persistence-related tasks are handled using object-relational mapping (ORM) frameworks in Java applications. It provides the following features:

- Defines an API for mapping the object model with the relational model
- Defines an API for performing CRUD operations
- Standardizes ORM features and functionalities in Java.
- Provides an object query language called Java Persistence Query Language (JPQL) for querying the database.
- Provides Criteria API to fetch data over an object graph.

There are multiple providers available in the market which provides an implementation of JPA specification such as EclipseLink, OpenJPA, Hibernate, etc. as shown below:



Hibernate is the most widely used framework among these.
Once you have understood about ORM and its providers let's look at Spring ORM in brief.

Spring ORM:

Spring Framework is the most popular open-source Java application framework which supports building all types of Java applications like web applications, database-driven applications, batch applications, and many more. Spring framework's features such as Dependency Injection and Aspect-Oriented Programming help in developing a simple, easily testable, reusable, and maintainable application.

Spring is organized in a modular fashion. Developers can pick and choose the modules as per their needs. Spring ORM is a module under the Spring umbrella that covers many persistence technologies, namely JPA, JDO, Hibernate, and iBatis, etc. For each technology, the configuration basically consists of injecting a DataSource bean into SessionFactory or EntityManagerFactory and helps in performing CRUD operations.

Repository implementation using Spring ORM:

Spring ORM provides integration classes to integrate the application with ORM solutions such as JPA, Hibernate, MyBatis to perform database operations and transaction management smoothly. Hibernate specific exceptions need not be declared or caught as @Repository annotation. Spring enables the exception of clean translation.

The repository implementation class requires appropriate persistence resource bean from Spring Framework as below:

- JPA based repository uses EntityManagerFactory bean
- Hibernate based repository uses SessionFactory bean

JPA based repositories are the most convenient way of developing our repository layer of Spring ORM applications as JPA is a specification and Hibernate is an implementation. So with little modification in the future, we can migrate from one ORM implementation to another.

Developing the persistence layer using Spring ORM JPA possess the following advantages:

- Provides a programmatic approach to implement database operations.
- Maps Java objects to the relational database tables with help of entity classes based on simple configuration.
- Supports HQL(Hibernate Query language) and JPQL (Java Persistence Query Language)
- Supports object-oriented concepts such as inheritance, mapping, etc.

So let's see additional configurations required for developing the repository layer of a Spring ORM JPA application with Spring Boot.

Spring ORM JPA Configurations

Let's take the same InfyTel Customer management scenario and develop the repository layer to perform the CRUD operations. In the Spring Boot application, a DAO(Data Access Object)/ repository class can be defined using **@Repository** annotation.

@Repositoy:

It indicates the repository class(POJO class) in the persistence layer. Repository class is defined using **@Repository** annotation at the class level. This annotation is a specialization of **@Component** annotation for the persistence layer.

Benefits of the @Repository:

- Enables the Spring Framework auto scanning support to create a repository bean without the explicit bean definition in the configuration file.
- Causes Spring exception translation(translates checked exceptions into unchecked exceptions).

EntityManagerFactory:
EntityManagerFactory will be used by the Spring Boot application to obtain an EntityManager instance. An EntityManagerFactory is constructed for a specific database, and managing resources efficiently provides an efficient way to construct multiple EntityManagers instances for that database to perform CRUD operations. Spring Boot generally uses LocalContainerEntityManagerFactoryBean for full JPA capabilities.

The main dependency of LocalContainerEntityManagerFactoryBean is:

- Datasource

Datasource:
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.

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.

Sample application.properties:

```
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/ormjpa
spring.datasource.username=root
spring.datasource.password=root
```

EntityManager
Once the EntityManagerFactory is properly configured, it provides an instance of EntityManager interface whose methods can be used to interact with the database for CRUD operations with the help of entity objects. Some important methods of this interface are as follows:

- void persist(Object entity)
- find(Class entityClass, Object primaryKey)
- void remove(Object entity)
- void detach(Object entity)
- createQuery(JPQL)

ORM application generally uses Entity classes to support programmatic database operations so let's see what is an Entity class and its features.

Entity Class
JPA needs an Entity class to perform all the CRUD operations because Entities can represent fine-grained persistent objects and they are not remotely accessible components. An entity can aggregate objects together and effectively persist data and related objects using the transactional, security, and concurrency services of a JPA persistence provider. Let's discuss more on Entity classes.

Entity Class:
An Entity class is a class in Java that is mapped to a database table in a relational database. Entity classes can be created using @Entity annotation from javax.persistence package.

Let's map our Customer class used in the InfyTel application to a database table Customer and each of the attributes of the class phoneNumber, name, age, gender, address, and planId maps to the columns of the Customer table. The Customer class is persisted in the database it is also known as an **Entity class**. Each Java object which is created corresponds to a row in the given database table.
Eg: Customer customer= new Customer(9009009009L, "Jack", 27, 'M', "BBSR", 1);

Let's create a sample entity class for our InfyTel application i.e. "Customer.java" :

```
package com.infytel.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
@Column(name = "plan_id")
private Integer planId;
//constructors
//getters and setters
}
```

Few of the common annotations used for defining an Entity Class are explained below:

ANNOTATION	**DEFINITION**
@Entity	Declares an Entity class
@Id	Declares one of the attributes of the entity class as the primary key in the table in the database
@Column	Declares mapping of the particular column in the database to an attribute
@Table	Declares mapping of a certain entity class to a table name in a database
@Temporal	Specified for persistent table fields or attributes of the type java.util.Date and java.util.Calendar. It solves one of the major issues of converting the date and time values from java object to compatible database type and retrieving back to the application
@Transient	It will not be persisted into the database

Note:

- All the annotations explained in the above table are available in javax.persistence package.

- In Hibernate 5.0 you don't need any @Temporal annotations or converter to persist the classes of the Date and Time API. You can use them in the same way as any other supported attribute types.

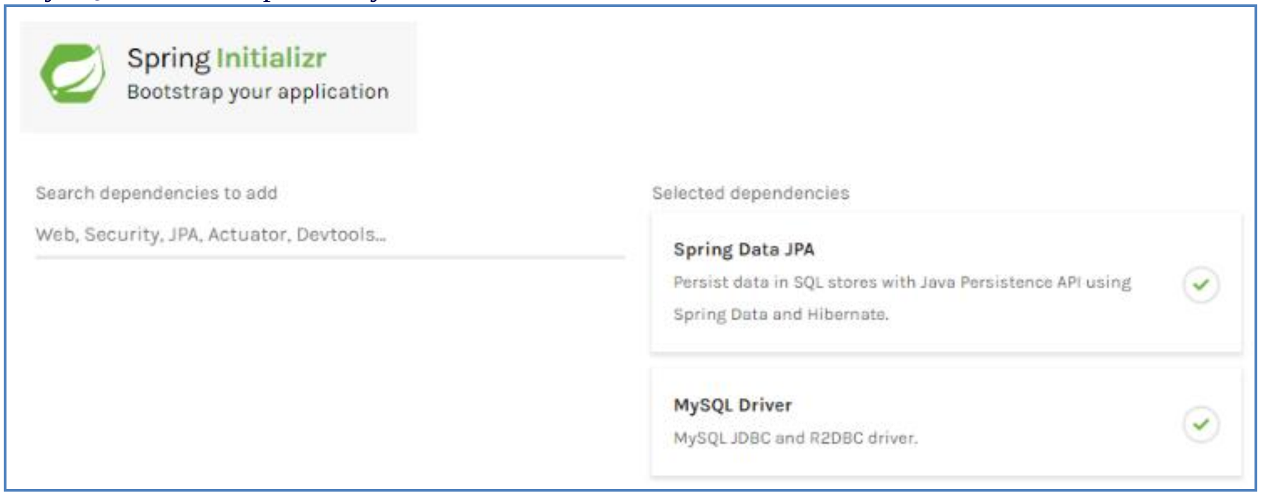
So as we can see a lot of coding effort has to be done in the repository class implementation to perform our CRUD operations. So let's see a complete demo using Spring ORM JPA to perform CRUD operation, understand it's drawbacks and why we need Spring Data JPA.

Demo- InfyTel Customer using Spring ORM

CRUD Operation in Spring ORM JPA with Spring Boot
Demo 2: Application Development Using Spring ORM JPA
Highlights:
- To understand key concepts and configuration of Spring ORM JPA
- To perform CRUD operations using Spring ORM - JPA

Consider the InfyTel scenario and perform the following operations using ORM-JPA.
- Insert Customer details
- Delete Customer for given phone number
- Update Customer address details for a given phone number
- Display all Customer details

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


Spring Initializr
Bootstrap your application
Search dependencies to add
Web, Security, JPA, Actuator, Devtools...
Selected dependencies

Spring Data JPA
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver
MySQL JDBC and R2DBC driver.

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

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>
```

Note that the version number of the jars is not defined here as it can be determined by spring-boot-starter-parent. Once you have properly configured your maven project our pom.xml will look as below:

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.infytel</groupId>
<artifactId>demo-spring-orm-jpa-crud</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-spring-orm-jpa-crud</name>
<description>Spring Boot project to demonstrate update and fetch operation using Spring ORM
JPA</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
```

```
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Step 1: Create an entity class “Customer.java” as shown below:

```
package com.infytel.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import com.infytel.dto.CustomerDTO;
@Entity
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
@Column(name = "plan_id")
private Integer planId;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String address,
Integer planId) {
```

```
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
```

```
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static CustomerDTO prepareDTO(Customer customerEntity)
{
CustomerDTO custDTO = new CustomerDTO();
custDTO.setPhoneNumber(customerEntity.getPhoneNumber());
custDTO.setName(customerEntity.getName());
custDTO.setGender(customerEntity.getGender());
custDTO.setAge(customerEntity.getAge());
custDTO.setAddress(customerEntity.getAddress());
custDTO.setPlanId(customerEntity.getPlanId());
return custDTO;
}
}
```

Step 2: Create a DTO class “CustomerDTO.java” as shown below:

```
package com.infytel.dto;
import com.infytel.entity.Customer;
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
public CustomerDTO() {}
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String
address, Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
```

```
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static Customer prepareCustomerEntity(CustomerDTO customerDTO)
{
Customer customerEntity = new Customer();
```

```
customerEntity.setPhoneNumber(customerDTO.getPhoneNumber());
customerEntity.setName(customerDTO.getName());
customerEntity.setGender(customerDTO.getGender());
customerEntity.setAge(customerDTO.getAge());
customerEntity.setAddress(customerDTO.getAddress());
customerEntity.setPlanId(customerDTO.getPlanId());
return customerEntity;
}
}
```

Step 3: Create an interface “CustomerService.java” as shown below:

```
package com.infytel.service;
import java.util.List;
import com.infytel.dto.CustomerDTO;
public interface CustomerService {
// Method to access the repository layer method to insert Customer record
public void insert(CustomerDTO customer);
// Method to access the repository layer method to delete Customer record
public int remove(Long phoneNo);
// Method to get all the Customer record from the db
public List<CustomerDTO> getAll();
// Method to update a Customer record from the db
public void update(Long phoneNo, String address);
}
```

Step 4: Create a service class “CustomerServiceImpl.java” as shown below:

```
package com.infytel.service;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infytel.dto.CustomerDTO;
import com.infytel.entity.Customer;
import com.infytel.repository.CustomerDAO;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
@Autowired
CustomerDAO customerDAO;
@Override
public void insert(CustomerDTO customer) {
customerDAO.insert(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public int remove(Long phoneNo) {
return  customerDAO.remove(phoneNo);
}
@Override
public List<CustomerDTO> getAll() {
List<CustomerDTO> custList = new ArrayList<>();
List <Customer> custEntityList = customerDAO.getAll();
for (Customer customerEntity : custEntityList) {
CustomerDTO custDTO = Customer.prepareDTO(customerEntity);
custList.add(custDTO);
}
return custList;
}
@Override
public void update(Long phoneNo, String address) {
customerDAO.update(phoneNo, address);
}
}
```

Step 5: Create an interface “CustomerDAO.java” as shown below:

```
package com.infytel.repository;
import com.infytel.entity.Customer;
public interface CustomerDAO {
// Method to insert a Customer record into the db
public void insert(Customer customer);
// Method to remove a Customer record from the db
public int remove(Long phoneNo);
// Method to get all the Customer record from the db
public List<Customer> getAll();
// Method to update a Customer record from the db
public void update(Long phoneNo, String address);
}
```

Step 6: Create a repository class “CustomerDAOImpl.java” as shown below:

```
package com.infytel.repository;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import javax.persistence.PersistenceUnit;
import org.springframework.stereotype.Repository;
import com.infytel.entity.Customer;
@Repository("customerRepository")
public class CustomerDAOImpl implements CustomerDAO {
```

```
private EntityManagerFactory entityManagerFactory;
@PersistenceUnit
public void setEntityManagerFactory (EntityManagerFactory entityManagerFactory) {
this.entityManagerFactory = entityManagerFactory;
}
@Override
public void insert(Customer customer) {
EntityManager entityManager = this.entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
entityManager.persist(customer);
entityManager.getTransaction().commit();
}
@Override
public int remove(Long phoneNo) {
EntityManager entityManager = this.entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
int result = 0;
try {
Customer emp = entityManager.find(Customer.class, phoneNo);
entityManager.remove(emp);
result = 1;
entityManager.getTransaction().commit();
} catch (Exception exp) {
entityManager.getTransaction().rollback();
}
entityManager.close();
return result;
}
@Override
@SuppressWarnings("unchecked")
public List<Customer> getAll() {
EntityManager entityManager = this.entityManagerFactory.createEntityManager();
Query query = entityManager.createQuery("Select c from Customer c");
return (List<Customer>)query.getResultList();
}
@Override
public void update(Long phoneNo, String address) {
EntityManager entityManager = this.entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
Customer cust =  entityManager.find(Customer.class, phoneNo);
cust.setAddress(address);
entityManager.getTransaction().commit();
}
}
```

Step 7: Update "application.properties" as shown below:

```
#SQL
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/ormdb
spring.datasource.username=root
spring.datasource.password=root
#Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
hibernate.show_sql: true
```

Step 8: Create a class “Client.java” as shown below:

```
package com.infytel;
import java.util.Scanner;
import org.apache.log4j.Logger;
import java.util.ArrayList;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.support.AbstractApplicationContext;
import com.infytel.dto.CustomerDTO;
import com.infytel.service.CustomerService;
@SpringBootApplication
public class Client implements CommandLineRunner {
static Logger logger = Logger.getLogger(Client.class);
@Autowired
AbstractApplicationContext context;
public static void main(String[] args) {
SpringApplication.run(Client.class, args);
}
@Override
public void run(String... args) throws Exception {
CustomerService customerService = (CustomerService)context.getBean("customerService");
CustomerDTO customer1= new CustomerDTO(9009009009L, "Debashis", 27, 'M', "BBSR", 1);
CustomerDTO customer2= new CustomerDTO(9009009010L, "Robert", 27, 'M', "PUNE", 2);
CustomerDTO customer3= new CustomerDTO(9009009011L, "Lucy", 27, 'F', "MUMBAI", 3);
customerService.insert(customer1);
customerService.insert(customer2);
customerService.insert(customer3);
logger.info("Records are successfully added..");
System.out.println("Enter the phone Number of the Customer which has to be deleted.");
Scanner scanner = new Scanner(System.in);
Long phoneNo = scanner.nextLong();
// Invoking Service layer method to remove Customer details from
// Customer table
int result = customerService.remove(phoneNo);
if (result == 1) {
logger.info("Success : Record deleted successfully ");
} else {
logger.info("Error : No record found for the given phone number ");
}
logger.info("Viewing All Customer Details");
ArrayList<CustomerDTO> cList = (ArrayList<CustomerDTO>)customerService.getAll();
for (CustomerDTO customer : cList) {
logger.info(customer);
}
logger.info("Display completed");
logger.info("");
logger.info("Let's update a Customer with new Address details");
System.out.println("Enter the phone number of the Customer to be updated:");
Scanner sc = new Scanner(System.in);
Long phoneNo1 =  sc.nextLong();
System.out.println("Enter new Address");
String newAddress =  sc.next();
customerService.update(phoneNo1, newAddress);
logger.info("Customer's address updated successfully!");
scanner.close();
context.close();
}
}
```

Run the Client.java as "Spring Boot App".

Expected output on the Console:

```
Records are successfully added..
Enter the phone Number of the Customer which has to be deleted.

9009009009
Success : Record deleted successfully

Viewing All Customer Details
Customer [phoneNumber=9009009010, name=Robert, age=27, gender=M, address=PUNE, planId=2]
Customer [phoneNumber=9009009011, name=Lucy, age=27, gender=F, address=MUMBAI, planId=3]
Display completed

Let's update a Customer with new Address details
Enter the phone number of the Customer to be updated:
9009009011

Enter new Address
BBSR
Customer's address updated successfully!
```

Spring ORM JPA provides a lot of advantages in the development of the repository layer of an enterprise application still it comes with some limitations. Let's discuss those limitations and how Spring Data JPA helps in overcoming those limitations in the upcoming discussion.

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.

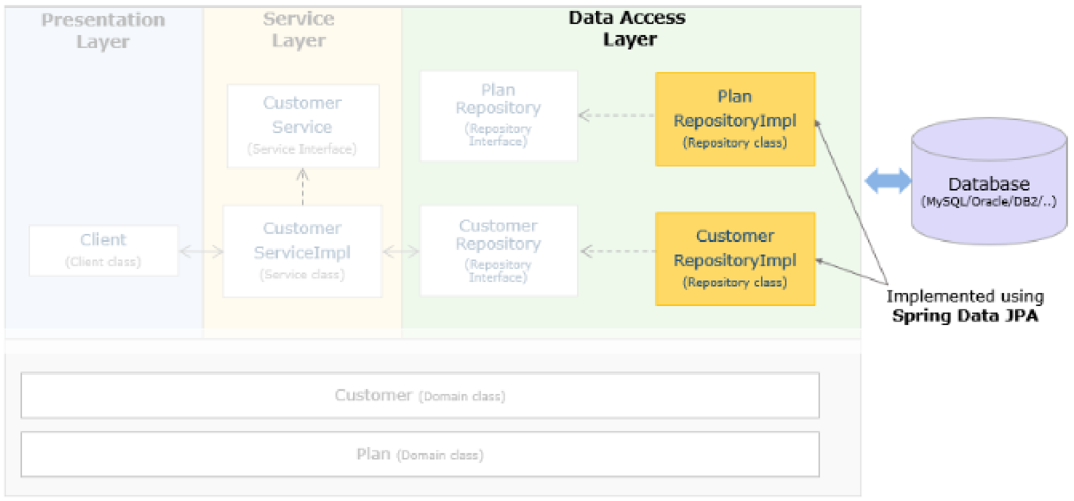
Let us look at the limitations while using these approaches:

- 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.

Higlevel Diagram of InfyTel Customer using Spring Data JPA

InfyTel system's data access layer implementation using Spring Data JPA is as shown below:


Presentation Layer
Service Layer
Data Access Layer
Customer Service (Service Interface)
Plan Repository (Repository Interface)
Plan RepositoryImpl (Repository class)
Database (MySQL/Oracle/DB2/..)
Client (Client class)
Customer ServiceImpl (Service class)
Customer Repository (Repository Interface)
Customer RepositoryImpl (Repository class)
Implemented using Spring Data JPA
Customer (Domain class)
Plan (Domain class)

Now let's understand What's Spring Data JPA.

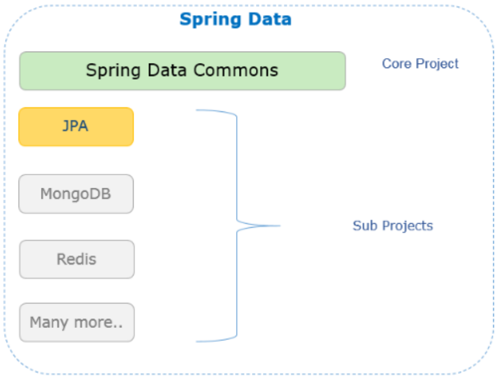
What is Spring Data JPA?

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.

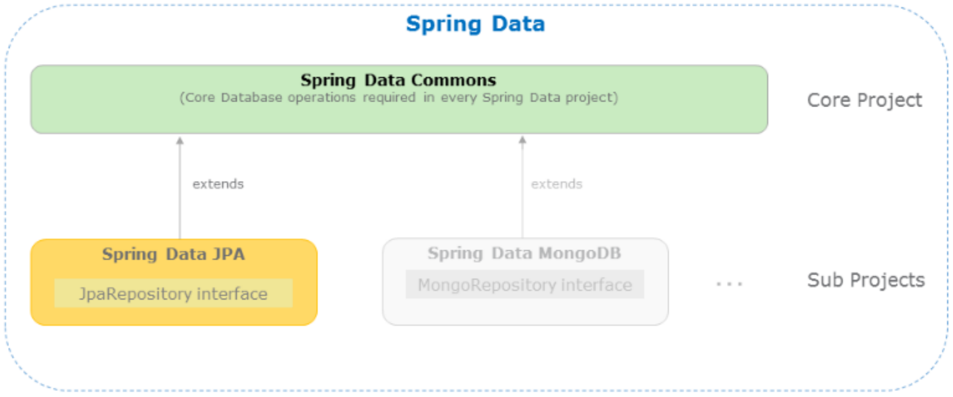


Spring Data JPA is one of the Spring Data subproject to support the implementation of JPA-based repositories.

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.

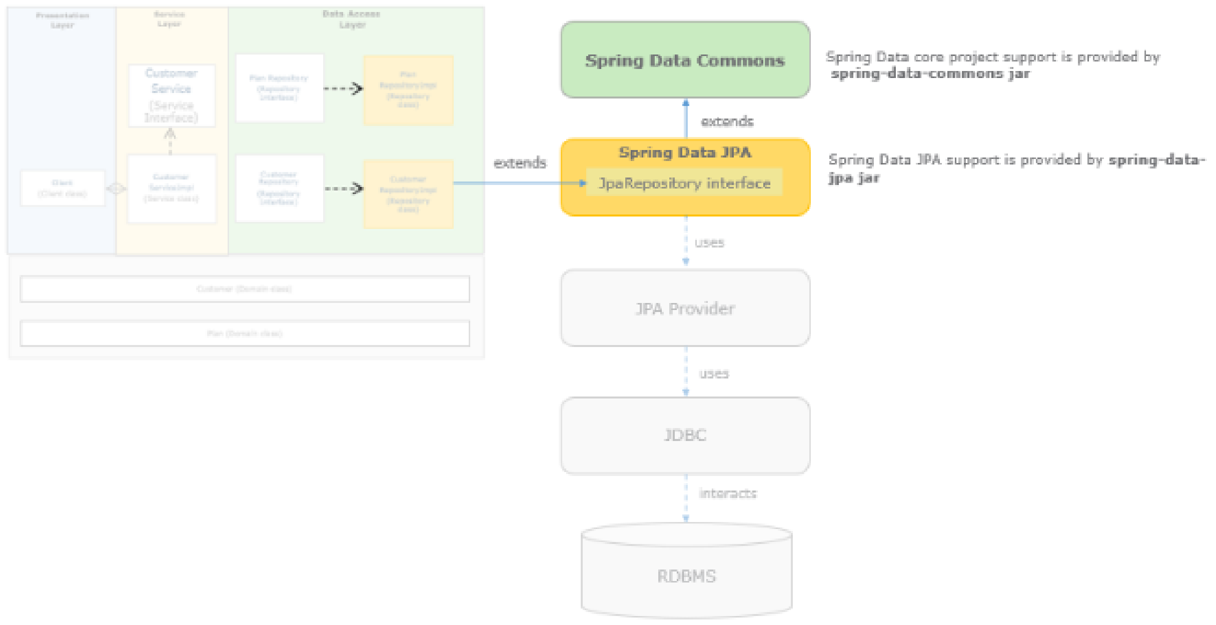


Let us understand more about how to implement the application data access layer using Spring Data JPA.

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.


Presentation Layer
Service Layer
Data Access Layer
Customer Service (Service Interface)
Client (Client class)
CustomerServiceImpl (Service class)
Plan Repository (Repository interface)
Plan RepositoryImpl (Repository class)
Customer Repository (Repository interface)
Customer RepositoryImpl (Repository class)
Customer (Domain class)
Plan (Domain class)
extends
Spring Data Commons
Spring Data core project support is provided by spring-data-commons jar
extends
Spring Data JPA
JpaRepository interface
Spring Data JPA support is provided by spring-data-jpa jar
uses
JPA Provider
uses
JDBC
interacts
RDBMS

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

Spring Data JPA with Spring Boot

Introduction to 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

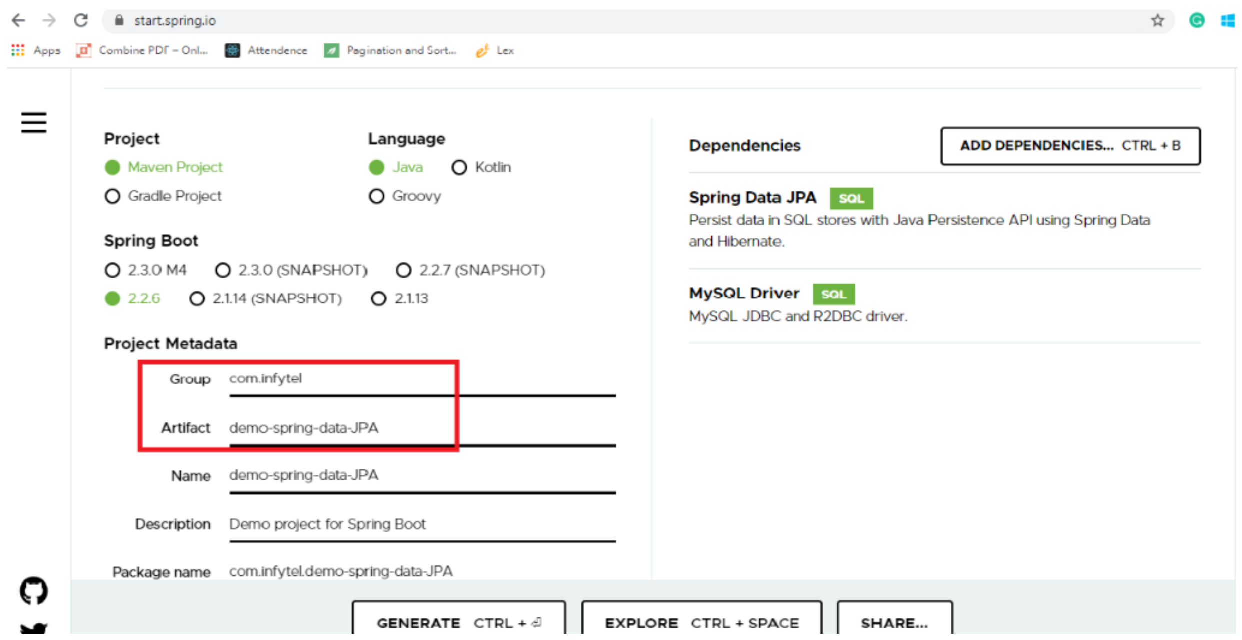
There are multiple approaches to create a Spring Boot application. You can use any of the following approaches to create the application:
- Using Spring Initializr
- Using the Spring Tool Suite (STS)
- Using Spring Boot CLI

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

Spring Initializr:
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)



Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.3.0 M4 ○ 2.3.0 (SNAPSHOT) ○ 2.2.7 (SNAPSHOT)
● 2.2.6 ○ 2.1.14 (SNAPSHOT) ○ 2.1.13

Project Metadata

Group	com.infytel
Artifact	demo-spring-data-JPA
Name	demo-spring-data-JPA
Description	Demo project for Spring Boot
Package name	com.infytel.demo-spring-data-JPA

Dependencies

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

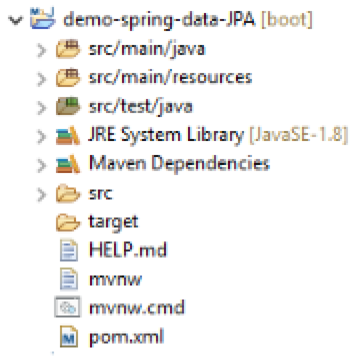
MySQL Driver SQL
MySQL JDBC and R2DBC driver.

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.

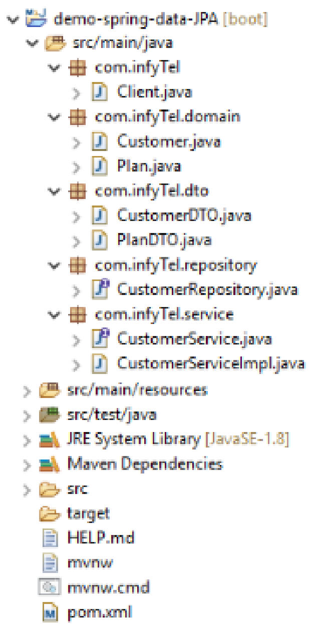
Step 4: Unzip the zip file and extract to a folder.

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. After finishing, our Spring Boot project should look like as follows:



```
demo-spring-data-JPA [boot]
  src/main/java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Add the respective files in the project as below:



```
demo-spring-data-JPA [boot]
  src/main/java
    com.infyTel
      Client.java
    com.infyTel.domain
      Customer.java
      Plan.java
    com.infyTel.dto
      CustomerDTO.java
      PlanDTO.java
    com.infyTel.repository
      CustomerRepository.java
    com.infyTel.service
      CustomerService.java
      CustomerServiceImpl.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

We have successfully created a Spring Boot Maven-based project with Spring Data JPA dependencies.

Spring Data JPA- 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.6.RELEASE</version>
```

```
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.infytel</groupId>
<artifactId>demo-spring-data-JPA</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-spring-data-JPA</name>
<description>Demo project for Spring Data JPA with Spring Boot</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Spring Boot Starter Parent and Spring Boot starters

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:

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.6.RELEASE</version>
<relativePath/>
</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.

```
<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.

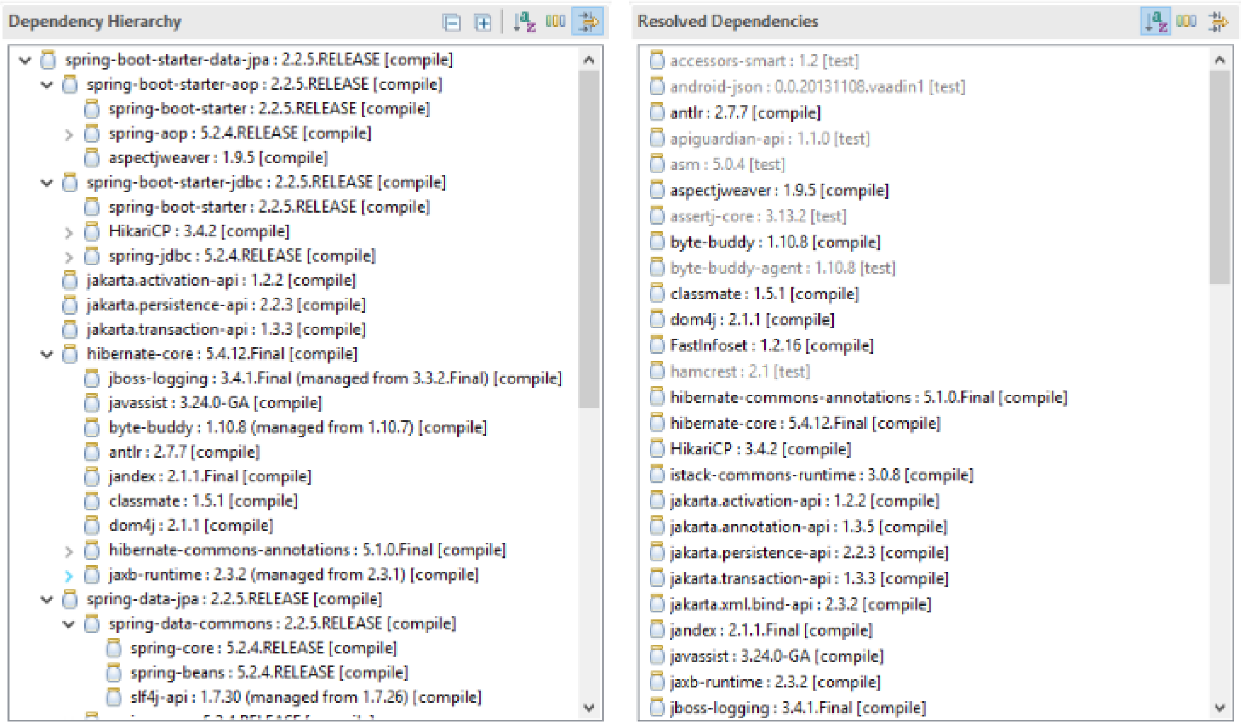
Spring Boot comes with many starters. Some popular starters are as follows:

- 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.

```
1.  <dependency>
2.  <groupId>org.springframework.boot</groupId>
3.  <artifactId>spring-boot-starter-data-jpa</artifactId>
4.  </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-boot-starter-data-jpa : 2.2.5.RELEASE [compile]
  spring-boot-starter-aop : 2.2.5.RELEASE [compile]
    spring-boot-starter : 2.2.5.RELEASE [compile]
    spring-aop : 5.2.4.RELEASE [compile]
    aspectjweaver : 1.9.5 [compile]
  spring-boot-starter-jdbc : 2.2.5.RELEASE [compile]
    spring-boot-starter : 2.2.5.RELEASE [compile]
    HikariCP : 3.4.2 [compile]
    spring-jdbc : 5.2.4.RELEASE [compile]
  jakarta.activation-api : 1.2.2 [compile]
  jakarta.persistence-api : 2.2.3 [compile]
  jakarta.transaction-api : 1.3.3 [compile]
  hibernate-core : 5.4.12.Final [compile]
    jboss-logging : 3.4.1.Final (managed from 3.3.2.Final) [compile]
    javassist : 3.24.0-GA [compile]
    byte-buddy : 1.10.8 (managed from 1.10.7) [compile]
    antlr : 2.7.7 [compile]
    jandex : 2.1.1.Final [compile]
    classmate : 1.5.1 [compile]
    dom4j : 2.1.1 [compile]
    hibernate-commons-annotations : 5.1.0.Final [compile]
    jaxb-runtime : 2.3.2 (managed from 2.3.1) [compile]
  spring-data-jpa : 2.2.5.RELEASE [compile]
    spring-data-commons : 2.2.5.RELEASE [compile]
      spring-core : 5.2.4.RELEASE [compile]
      spring-beans : 5.2.4.RELEASE [compile]
      slf4j-api : 1.7.30 (managed from 1.7.26) [compile]
```

```
accessors-smart : 1.2 [test]
android-json : 0.0.20131108.vaadin1 [test]
antlr : 2.7.7 [compile]
apiguardian-api : 1.1.0 [test]
asm : 5.0.4 [test]
aspectjweaver : 1.9.5 [compile]
assertj-core : 3.13.2 [test]
byte-buddy : 1.10.8 [compile]
byte-buddy-agent : 1.10.8 [test]
classmate : 1.5.1 [compile]
dom4j : 2.1.1 [compile]
FastInfoset : 1.2.16 [compile]
hamcrest : 2.1 [test]
hibernate-commons-annotations : 5.1.0.Final [compile]
hibernate-core : 5.4.12.Final [compile]
HikariCP : 3.4.2 [compile]
istack-commons-runtime : 3.0.8 [compile]
jakarta.activation-api : 1.2.2 [compile]
jakarta.annotation-api : 1.3.5 [compile]
jakarta.persistence-api : 2.2.3 [compile]
jakarta.transaction-api : 1.3.3 [compile]
jakarta.xml.bind-api : 2.3.2 [compile]
jandex : 2.1.1.Final [compile]
javassist : 3.24.0-GA [compile]
jaxb-runtime : 2.3.2 [compile]
jboss-logging : 3.4.1.Final [compile]
```

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.

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

Database dependency:

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

```
1.  <dependency>
2.  <groupId>mysql</groupId>
3.  <artifactId>mysql-connector-java</artifactId>
4.  <scope>runtime</scope>
5.  </dependency>
```

Note: Incase if any other DB is used an appropriate dependency can be used.

Configuring database and ClientApplication

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.Driver
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);
}
}
```

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:

```
@SpringBootApplication
public class ClientApplication implements CommandLineRunner {
public static void main(String[] args) {
SpringApplication.run(DemoSpringBootApplication.class, args);
}
@Override
public void run(String... args) throws Exception {
System.out.println("Welcome to CommandLineRunner");
}
}
```

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 Configuration:

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:

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

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.

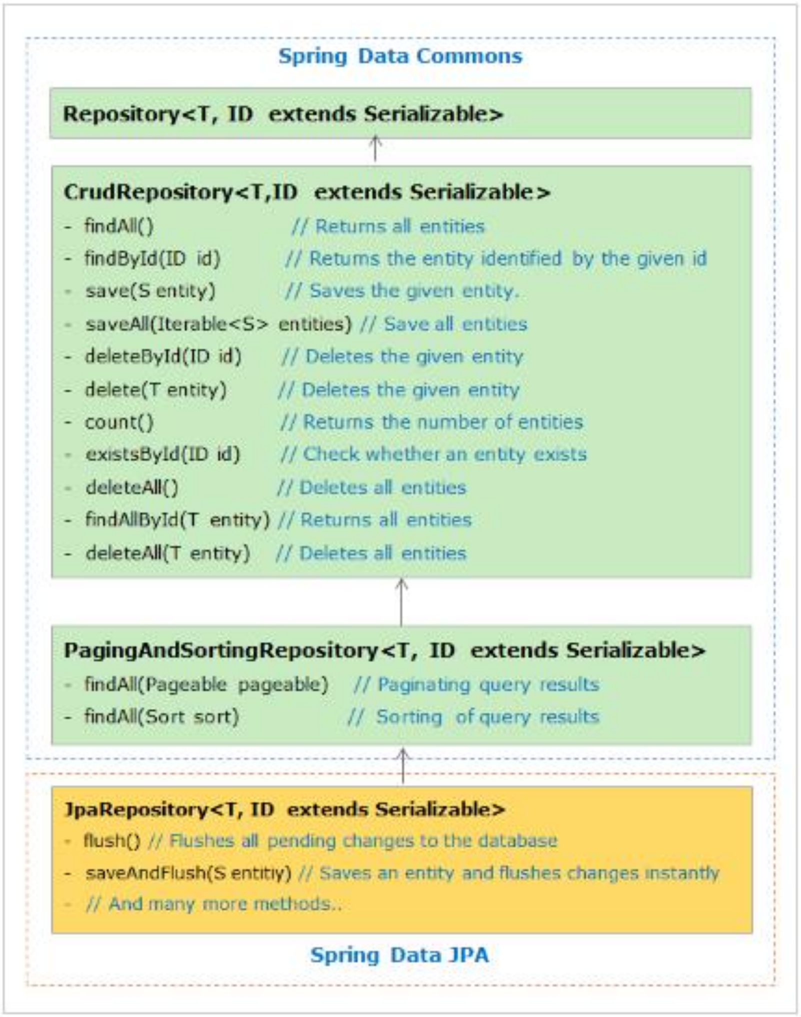
@Repository annotation can be used at the user-defined interface as Spring provides repository implementation of this interface. However, it is optional to mention the annotation explicitly because Spring auto-detects this interface as a Repository.

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

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.

Spring Data JPA Repository hierarchy

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


Spring Data Commons
Repository<T, ID extends Serializable>
CrudRepository<T,ID extends Serializable>
- findAll() // Returns all entities
- findById(ID id) // Returns the entity identified by the given id
- save(S entity) // Saves the given entity.
- saveAll(Iterable<S> entities) // Save all entities
- deleteById(ID id) // Deletes the given entity
- delete(T entity) // Deletes the given entity
- count() // Returns the number of entities
- existsById(ID id) // Check whether an entity exists
- deleteAll() // Deletes all entities
- findAllById(T entity) // Returns all entities
- deleteAll(T entity) // Deletes all entities
PagingAndSortingRepository<T, ID extends Serializable>
- findAll(Pageable pageable) // Paginating query results
- findAll(Sort sort) // Sorting of query results
JpaRepository<T, ID extends Serializable>
- flush() // Flushes all pending changes to the database
- saveAndFlush(S entitiy) // Saves an entity and flushes changes instantly
- // And many more methods..
Spring Data JPA

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.

Additional configurations - DataSource

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.

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

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

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 Operation 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:

```
package com.infyTel.service;
import com.infyTel.dto.CustomerDTO;
public interface CustomerService {
public void insertCustomer(CustomerDTO Customer) ;
public void removeCustomer(Long phoneNo);
public CustomerDTO getCustomer(Long phoneNo);
public String updateCustomer(Long phoneNo,Integer newPlanId);
}
```

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

CustomerServiceImpl.java:

```
package com.infyTel.service;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
@Autowired
private CustomerRepository repository;
@Override
public void insertCustomer(CustomerDTO customer) {
repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public void removeCustomer(Long phoneNo) {
repository.deleteById(phoneNo);
}
@Override
public CustomerDTO getCustomer(Long phoneNo) {
Optional<Customer> optionalCustomer = repository.findById(phoneNo);
Customer customerEntity = optionalCustomer.get();// Converting Optional<Customer> to Customer
CustomerDTO customerDTO = Customer.prepareCustomerDTO(customerEntity);
return customerDTO;
}
@Override
public String updateCustomer(Long phoneNo, Integer newPlanId) {
Optional<Customer> optionalCustomer = repository.findById(phoneNo);
Customer customerEntity = optionalCustomer.get();
customerEntity.setPlanId(newPlanId);
repository.save(customerEntity);
return "The plan for the customer with phone number :" + phoneNo + " has been updated
successfully.";
}
}
```

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.

Let us see the complete demo to implement CRUD operation for a better understanding.

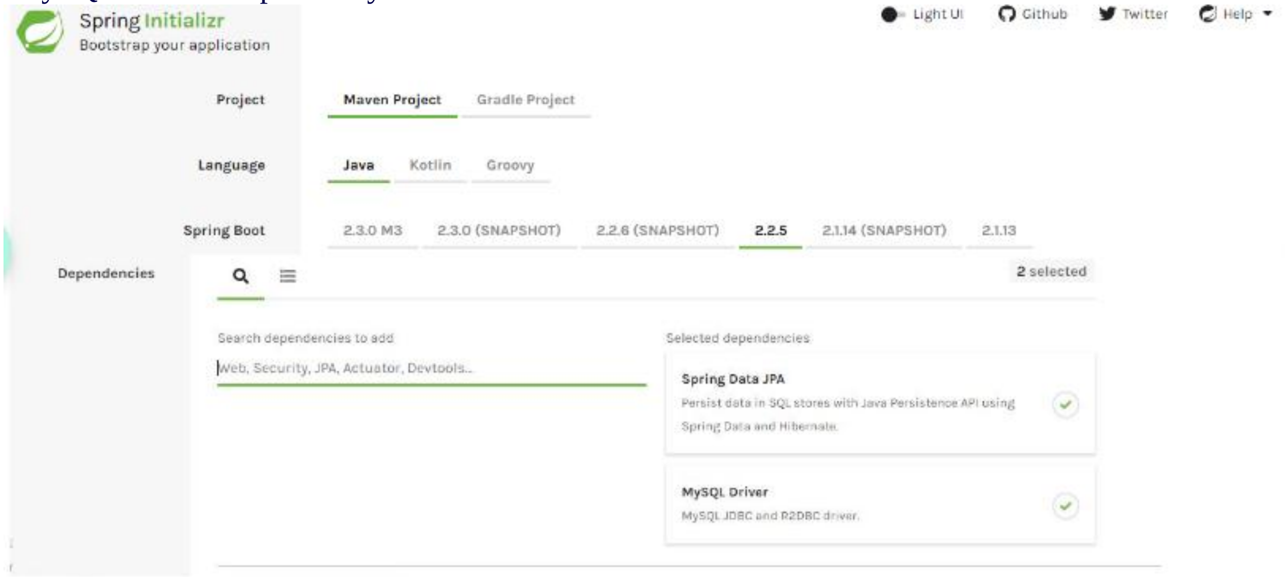
Demo- CRUD Operation using Spring Data JPA
Demo 3: InfyTel Application development using Spring Data JPA
Highlights:

- 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.



Project: Maven Project
Language: Java
Spring Boot: 2.2.5
Dependencies: 2 selected
Search dependencies to add
Web, Security, JPA, Actuator, Devtools...
Selected dependencies
Spring Data JPA
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
MySQL Driver
MySQL JDBC and R2DBC driver.

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

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>
```

Note that the version number of the jars is not defined here as it can be determined by spring-boot-starter-parent. Once you have properly configured your maven project our pom.xml will look as below:

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.infyTel</groupId>
<artifactId>demo-jpa-insertdelete</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-jpa-insertdelete</name>
<description>Spring Boot project to demonstrate insert and delete operation using Spring Data JPA</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Step 1: Create an Entity class "Customer.java" as shown below:

```
package com.infyTel.domain;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
@Column(name = "plan_id")
private Integer planId;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String address, Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
```

```
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static CustomerDTO prepareCustomerDTO(Customer customer)
{
CustomerDTO customerDTO = new CustomerDTO();
customerDTO.setPhoneNumber(customer.getPhoneNumber());
customerDTO.setName(customer.getName());
customerDTO.setGender(customer.getGender());
customerDTO.setAge(customer.getAge());
customerDTO.setAddress(customer.getAddress());
customerDTO.setPlanId(customer.getPlanId());
return customerDTO;
}
}
```

Step 2: Create a DTO class "CustomerDTO.java" as shown below:

```
package com.infyTel.dto;
import com.infyTel.domain.Customer;
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
public CustomerDTO() {}
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String
address, Integer planId) {
super();
```

```
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
```

```
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ", gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static Customer prepareCustomerEntity(CustomerDTO customerDTO)
{
Customer customerEntity = new Customer();
customerEntity.setPhoneNumber(customerDTO.getPhoneNumber());
customerEntity.setName(customerDTO.getName());
customerEntity.setGender(customerDTO.getGender());
customerEntity.setAge(customerDTO.getAge());
customerEntity.setAddress(customerDTO.getAddress());
customerEntity.setPlanId(customerDTO.getPlanId());
return customerEntity;
}
}
```

Step 3: Create an interface "CustomerService.java" as shown below:

```
package com.infyTel.service;
import com.infyTel.dto.CustomerDTO;
public interface CustomerService {
public void insertCustomer(CustomerDTO Customer) ;
public void removeCustomer(Long phoneNo);
public CustomerDTO getCustomer(Long phoneNo);
public String updateCustomer(Long phoneNo,Integer newPlanId);
}
```

Step 4: Create a service class "CustomerServiceImpl.java" as shown below:

```
package com.infyTel.service;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
@Autowired
private CustomerRepository repository;
@Override
public void insertCustomer(CustomerDTO customer) {
```

```
repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public void removeCustomer(Long phoneNo) {
repository.deleteById(phoneNo);
}
@Override
public CustomerDTO getCustomer(Long phoneNo) {
Optional<Customer> optionalCustomer = repository.findById(phoneNo);
Customer customerEntity = optionalCustomer.get();// Converting Optional<Customer> to Customer
CustomerDTO customerDTO = Customer.prepareCustomerDTO(customerEntity);
return customerDTO;
}
@Override
public String updateCustomer(Long phoneNo, Integer newPlanId) {
Optional<Customer> optionalCustomer = repository.findById(phoneNo);
Customer customerEntity = optionalCustomer.get();
customerEntity.setPlanId(newPlanId);
repository.save(customerEntity);
return "The plan for the customer with phone Number :" + phoneNo + " has been updated
successfully.";
}
}
```

Step 5: Create an interface “CustomerRepository.java” as shown below:

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

Step 6: Update "application.properties" as shown below:

```
spring.datasource.url = jdbc:mysql://localhost:3306/sample
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.generate-ddl=true
```

Step 7: Create a class “Client.java” as shown below:

```
package com.infyTel;
import java.util.Scanner;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.service.CustomerService;
@SpringBootApplication
public class Client implements CommandLineRunner{
static Logger logger = Logger.getLogger(Client.class);
@Autowired
ApplicationContext context;
@Autowired
CustomerService service;
public static void main(String[] args) {
SpringApplication.run(Client.class, args);
}
@Override
public void run(String... args) throws Exception {
CustomerDTO customer1= new CustomerDTO(7022713754L, "Adam", 27, 'M', "Chicago", 1);
CustomerDTO customer2= new CustomerDTO(7022713744L, "Susan", 27, 'F', "Alberta", 2);
CustomerDTO customer3= new CustomerDTO(7022713722L, "Lucy", 27, 'F', "MUMBAI", 3);
//invoke service layer method to insert Customer
service.insertCustomer(customer1);
service.insertCustomer(customer2);
service.insertCustomer(customer3);
logger.info("Records are successfully added..");
System.out.println("Enter the phone Number of the Customer which has to be deleted.");
Scanner scanner = new Scanner(System.in);
Long phoneNo = scanner.nextLong();
// Invoking Service layer method to remove Customer details from
// Customer table
service.removeCustomer(phoneNo);
logger.info("Record Deleted");
logger.info("Let's print the details of a Customer");
System.out.println("Enter the phone Number of the Customer whose details have to be printed.");
Long phoneNo1 = scanner.nextLong();
CustomerDTO customerDTO = service.getCustomer(phoneNo1);
logger.info("Customer Details:");
logger.info("Phone Number : "+customerDTO.getPhoneNumber());
logger.info("Name         : "+customerDTO.getName());
logger.info("Age         : "+customerDTO.getAge());
logger.info("Gender       : "+customerDTO.getAge());
logger.info("Address      : "+customerDTO.getAddress());
logger.info("Plan ID      : "+customerDTO.getPlanId());
logger.info("Let's update the current plan of a Customer");
System.out.println("Enter the phone Number of the Customer whose current plan has to be
    updated.");
Long phoneNo2 = scanner.nextLong();
System.out.println("Enter the new plan id for the Customer");
Integer newPlanId = scanner.nextInt();
String msg = service.updateCustomer(phoneNo2, newPlanId);
logger.info(msg);
scanner.close();
}
}
```

Run the Client.java as "Spring Boot App".

Expected output on the Console:

```
Records are successfully added..
Enter the phone Number of the Customer which has to be deleted.
7022713722
Record Deleted
Let's print the details of a Customer
Enter the phone Number of the Customer whose details have to be printed.
7022713744
Customer Details:
Phone Number : 7022713744
Name         : Susan
Age        : 27
Gender       : 27
Address      : Alberta
Plan ID      : 2
Let's update the current plan of a Customer
Enter the phone Number of the Customer whose current plan has to be updated.
7022713744
Enter the new plan id for the Customer
5
The plan for the customer with phone Number :7022713744 has been updated successfully.
```

Note:

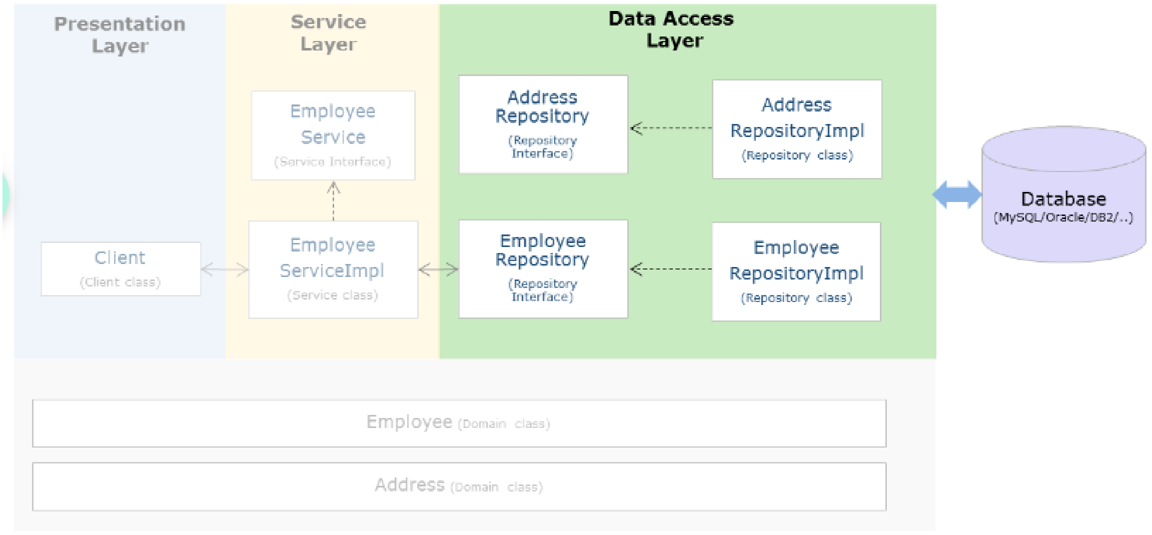
- The above demo has been given as two separate demos as 'demo-jpa-insertdelete' and 'demo-jpa-update' in the course demos.
- Verify the database after each operation.

Exercise: CRUD Operations using Spring Data JPA:

Employee Management Application: The HR team of an organization needs an application to maintain its employee details. The application should support the following features:

- Add Employee
- Edit Employee details
- Search employee
- View all employees
- Remove Employee

The way to develop the application using multi-tier architecture is shown below:


Presentation Layer
Service Layer
Data Access Layer
Employee Service (Service Interface)
Address Repository (Repository Interface)
Address RepositoryImpl (Repository class)
Database (MySQL/Oracle/DB2/..)
Client (Client class)
Employee ServiceImpl (Service class)
Employee Repository (Repository Interface)
Employee RepositoryImpl (Repository class)
Employee (Domain class)
Address (Domain class)

In this exercise, develop the data access layer of this application using Spring Data JPA with Spring Boot then modify the data access layer using different approaches in Spring Data JPA with Spring Boot. Also, know the benefits of these approaches.

Employee Management Application Files

The following table describes the files required for developing an Employee Management Application.

Required Files	**Description**
Client Application	It contains code to interact with application services
EmployeeService interface EmployeeServiceImpl class	It is an interface for client and repository Provides the implementation of EmployeeService interface with appropriate logic
AddressService interface AddressServiceImpl class	It is an interface between the service layer and data layer Provides the implementation of EmployeeRepository interface with appropriate logic
EmployeeRepository interface	It is an interface between the service layer and data layer
AddressRepository interface	It is an interface between the service layer and data layer
Employee class	It is an entity class to represent Employee details
Address class	It is an entity class to represent Address details

Note:

- It is optional to use interfaces in the service and data access layers. However, programming with an interface is a recommended practice.
- Sample Entity classes are given in the exercises.

Employee Management Application - Read operation Using Spring Data JPA
Exercise 1 :
Problem Statement:
Develop the data access layer of the Employee Management Application to perform the database operations given below using Spring Data JPA.
Add the operations given below:

- Retrieve an employee record based on employee id and display employee details on console.
- Retrieve and display the details of all the employees on the console.

Verify database table content after each operation, cross-check the data retrieved with the existing database, and ensure that they are updated.

Employee Management Application - Data Access Layer Using Spring ORM - JPA
Exercise 2 :
Problem Statement:
Develop the data access layer of the Employee Management Application to perform below-given database operations using Spring ORM integration with JPA:

- Add Employee
- Search Employee
- View All Employee
- Edit Employee
- Remove Employee

Refer to the application architecture discussed in the Employee Management scenario.
Create the following tables in the database:

```
CREATE TABLE Address(
address_id  INT(11) NOT NULL Primary Key,
city VARCHAR(255) NOT NULL,
pincode VARCHAR(255) NULL
);
CREATE TABLE Employee(
emp_id INT(11) NOT NULL Primary Key,
emp_name VARCHAR(255) NOT NULL,
department VARCHAR(255) NOT NULL,
base_Location VARCHAR(255) NOT NULL,
address INT(11) null,
CONSTRAINT Emp_FK Foreign key (Address) REFERENCES Address(address_id)
);
```

Entity classes as shown below:
Configure these Entity classes with proper annotations and import statements as per the table structure given above.
Employee.java

```
package com.infosys.domain;
//import statements
public class Employee {
```

```
private int empId;
private String empName;
private String department;
private String baseLocation;
private Address address;
// constructors
// getter and setter methods
}
```

Address.java

```
package com.infosys.domain;
//import statements
public class Address {
private int addressId;
private String city;
private String pincode;
// constructors
// getter and setter methods
}
```

Implement the complete application and verify the database table content after each operation.

Pagination and Sorting

Spring Data JPA – Pagination

Let us now understand paging and sorting support from Spring Data.
PagingAndSortingRepository interface of Spring Data Commons provides methods to support paging and sorting functionalities.

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

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.

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

Note: findAll(Pageable page) and findAll(Sort sort) methods are internally provided by Spring.

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.

```
//First argument '0" indicates first page and second argument 4 represents number of records.
Pageable pageable = PageRequest.of(0, 4);
```

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

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

Spring Data JPA – Sorting

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

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

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.

Demo: Pagination and Sorting in Spring Data JPA

Demo 4: Application Development Using Spring Data JPA

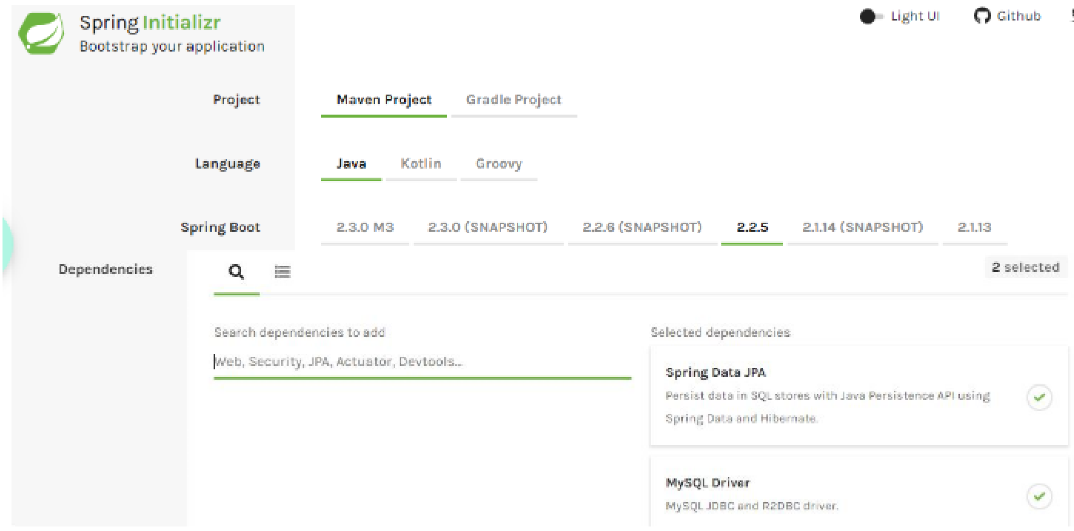
Highlights:

- To understand Paging and Sorting in Spring Data JPA

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

- Display 3 Customer per page
- Sort customers in descending order of their name

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



Project: [✓ Maven Project] Gradle Project

Language: [✓ Java] Kotlin Groovy

Spring Boot: 2.3.0 M3 2.3.0 (SNAPSHOT) 2.2.6 (SNAPSHOT) [✓ 2.2.5] 2.1.14 (SNAPSHOT) 2.1.13

Dependencies — 2 selected

Search dependencies to add: Web, Security, JPA, Actuator, Devtools...

Selected dependencies

Spring Data JPA — Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate. ✓

MySQL Driver — MySQL JDBC and R2DBC driver. ✓

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

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>
```

Note: The version number of the jars is not defined here as it can be determined by spring-boot-starter-parent. Once you have properly configured your maven project our pom.xml will look as below:

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
```

```
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.infyTel</groupId>
<artifactId>demo-jpa-pagination-sorting</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-jpa-pagination-sorting</name>
<description>Spring Boot project to demonstrate pagination and sorting operation using Spring Data
JPA</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Step 1: Create an Entity class "Customer.java" as shown below:

```
package com.infyTel.domain;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
@Column(name = "plan_id")
private Integer planId;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String address,
Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
```

```
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
}
```

Step 2: Create a DTO class "CustomerDTO.java" as shown below:

```
package com.infyTel.dto;
import com.infyTel.domain.Customer;
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
```

```
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String address, Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static Customer prepareCustomerEntity(CustomerDTO customerDTO)
{
Customer customerEntity = new Customer();
customerEntity.setPhoneNumber(customerDTO.getPhoneNumber());
customerEntity.setName(customerDTO.getName());
customerEntity.setGender(customerDTO.getGender());
customerEntity.setAge(customerDTO.getAge());
customerEntity.setAddress(customerDTO.getAddress());
customerEntity.setPlanId(customerDTO.getPlanId());
return customerEntity;
}
}
```

Step 3: Create an interface "CustomerService.java" as shown below:

```
package com.infyTel.service;
import java.util.List;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
public interface CustomerService {
public void insertCustomer(CustomerDTO Customer) ;
Page<Customer> findAll(Pageable page);
List<Customer> findAll(Sort sort);
}
```

Step 4: Create a service class "CustomerServiceImpl.java" as shown below:

```
package com.infyTel.service;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerRepository  repository;
public void insertCustomer(CustomerDTO customer) {
repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public Page<Customer> findAll(Pageable page) {
return repository.findAll(page);
}
@Override
public List<Customer> findAll(Sort sort) {
return repository.findAll(sort);
}
}
```

Step 5: Create a repository class “CustomerRepository.java” as shown below:

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

Step 6: Update "application.properties" as shown below:

```
spring.datasource.url = jdbc:mysql://localhost:3306/sample
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.generate-ddl=true
```

Step 7: Create a class “Client.java” as shown below:

```
package com.infyTel;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.repository.CustomerRepository;
import com.infyTel.service.CustomerService;
@SpringBootApplication
public class Client implements CommandLineRunner{
static Logger logger = Logger.getLogger(Client.class);
@Autowired
CustomerService service;
@Autowired
CustomerRepository repository;
public static void main(String[] args) {
SpringApplication.run(Client.class, args);
}
@Override
public void run(String... args) throws Exception {
// TODO Auto-generated method stub
CustomerDTO customer1= new CustomerDTO(7022713754L, "Adam", 27, 'M', "Chicago", 1);
CustomerDTO customer2= new CustomerDTO(7022713744L, "Susan", 27, 'F', "Alberta", 2);
CustomerDTO customer3= new CustomerDTO(7022713745L, "Andrew", 27, 'M', "New York", 2);
CustomerDTO customer4= new CustomerDTO(7022713746L, "Diana", 25, 'F', "Alberta", 1);
CustomerDTO customer5= new CustomerDTO(7022713747L, "Grace", 27, 'F', "Chicago", 1);
service.insertCustomer(customer1);
service.insertCustomer(customer2);
service.insertCustomer(customer3);
service.insertCustomer(customer4);
service.insertCustomer(customer5);
int k=(int) (repository.count()/3);
for(int i=0;i<=k;i++){
Pageable pageable = PageRequest.of(i,3);
logger.info("Records are: ");
Iterable<Customer> customer8 = service.findAll(pageable);
for(Customer alist3 : customer8){
logger.info(alist3);
}
}
logger.info("Sorted records..");
Iterable<Customer> customer8 = service.findAll(Sort.by(Sort.Direction.DESC,"name"));
for(Customer alist3 : customer8){
logger.info(alist3);
}
}
}
```

Run the Client.java as "Spring Boot App".

Expected output on the Console:

```
Records are:
Customer [phoneNumber=7022713744, name=Susan, age=27, gender=F, address=Alberta, planId=2]
Customer [phoneNumber=7022713745, name=Andrew, age=27, gender=M, address=New York, planId=2]
Customer [phoneNumber=7022713746, name=Diana, age=25, gender=F, address=Alberta, planId=1]


Records are:
Customer [phoneNumber=7022713747, name=Grace, age=27, gender=F, address=Chicago, planId=1]
Customer [phoneNumber=7022713754, name=Adam, age=27, gender=M, address=Chicago, planId=1]


Sorted records..
Customer [phoneNumber=7022713744, name=Susan, age=27, gender=F, address=Alberta, planId=2]
Customer [phoneNumber=7022713747, name=Grace, age=27, gender=F, address=Chicago, planId=1]
Customer [phoneNumber=7022713746, name=Diana, age=25, gender=F, address=Alberta, planId=1]
Customer [phoneNumber=7022713745, name=Andrew, age=27, gender=M, address=New York, planId=2]
Customer [phoneNumber=7022713754, name=Adam, age=27, gender=M, address=Chicago, planId=1]
```

Note:
The **count()** method resides in the **CrudRepository<T, ID extends Serializable>** which returns the number of records in the repository. In other words the count() method will return the number of rows present in the database table.
In our **Client.java**, while doing the pagination when we call **repository.count()**, it will return the **no of rows present in the Customer table**. As in this demo it is required to display **3 rows at a single time** so the number pages can be obtained by dividing the number of total rows by 3 which has been assigned to a variable 'k' of type int.

So the outer for loop will iterate the number of pages i.e. 'k' times, which we have got in the previous step and the inner for loop will iterate the number of desired records per page. Let's understand how the pagination is working in this demo.So when we are writting:

```
int k=(int) (repository.count()/3);
for(int i=0;i<=k;i++){
Pageable pageable = PageRequest.of(i,3);
Iterable<Customer> customer8 = service.findAll(pageable);
for(Customer alist3 : customer8){
logger.info(alist3);
}
}
```

The above code basically means:

```
/*Assume there are 5 records in the Customer table*/
int noOfPages=(int) (repository.count()/3); /* Here noOfPages = 1 as here we are doing a int division*/
for(int pageNumber=0;i<=noOfPages;pageNumber++){ /*So pageNumber will go from 0 - 1, So basically there will be 2 pages (page-0 and page-1)*/
Pageable pageable = PageRequest.of(pageNumber in the outer for loop, 3 records per page);/*For each page maximum 3 records should be displayed*/
Iterable<Customer> customer8 = service.findAll(pageable);/*The same pageable object is passed as a parameter to findAll(Pageable pageable) which resides in PagingAndSortingRepository<T, ID extends Serializable>*/
for(Customer alist3 : customer8){ /* This is the outer for loop which with iterate over the customer8 which is a type of Iterable<Customer> and will print 3 records per page. So in our example as the number of records is 5, so it will print first 3 records in the first page and next 2 records in the second page.*/
logger.info(alist3);
}
}
```

Spring Data JPA - 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.

Spring Data JPA - Query Creation Based on Method Name

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;
private Character gender;
private String address;
private Integer planId;
//getters and setters
}
```

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.

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

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.

Let us understand this concept in detail, through a demo.

Common findBy methods

Consider the Customer and Address classes given below:

Customer.java

```
@Entity
public class Customer{
@Id
int customerId;
boolean active;
int creditPoints;
String firstName;
String lastName;
String contactNumber;
String email;
@OneToOne( cascade = CascadeType.ALL)
@JoinColumn
Address address;
-----------
}
```

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
List<Customer> findByLastNameOrderByFirstNameAsc(String lastname);
// Query records based on specified list of cities
//select c from Customer c where c.address.city in ?1
List<Customer> findByAddress_CityIn(Collection<String> cities);
// Query records based if customer is active
//select c from Customer c where c.active = true
List<Customer> findByActiveTrue();
// Query records based on creditPoints >= specified value
//select c from Customer c where c.creditPoints >=?1
List<Customer> findByCreditPointsGreaterThanEqual(int points) ;
// Query records based on creditpoints between specified values
//select c from Customer c where c.creditPoints between ?1 and ?2
List<Customer> findByCreditPointsBetween(int point1, int point2)
}
```

Note:
List<Customer> findByFirstNameAndAddress_City(String fname, String city)
It can also be written as:

List<Customer> findByFirstNameAndAddressCity(String fname, String city);

In case there is a property by name “addressCity” in Customer class then to resolve this ambiguity we can add _ inside findBy method for manual traversal.

Spring Data JPA - Query Precedence

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

Note: If a developer wants to change the query precedence, then he can provide with extra configuration. This is not been discussed in this course.

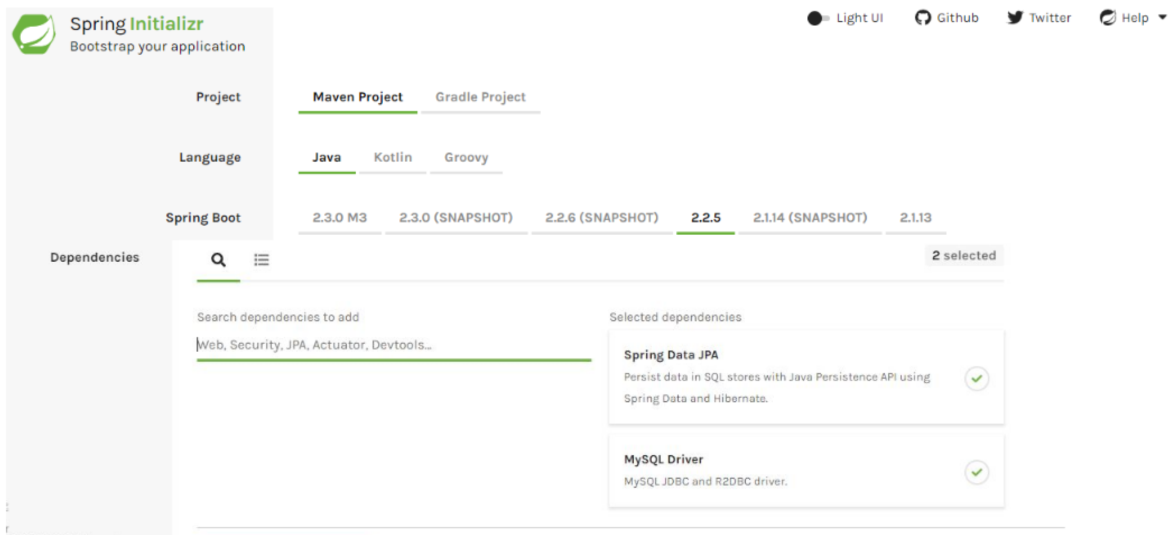
Demo: Query creation based on the method name – findBy
Demo 5: Application Development Using Spring Data JPA
Highlights:

- To understand query creation based on the method name

Consider the InfyTel scenario and perform the following operations using Spring Data JPA.

- Retrieve customer records based on the address and display the retrieved customer details on the console.

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


Spring Initializr
Bootstrap your application
Light UI
Github
Twitter
Help
Search dependencies to add
Web, Security, JPA, Actuator, Devtools...

Project: Maven Project | Gradle Project

Language: Java | Kotlin | Groovy

Spring Boot: 2.3.0 M3 | 2.3.0 (SNAPSHOT) | 2.2.6 (SNAPSHOT) | 2.2.5 | 2.1.14 (SNAPSHOT) | 2.1.13

Dependencies: 2 selected

Selected dependencies

Spring Data JPA
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver
MySQL JDBC and R2DBC driver.

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

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>
```

Note: The version number of the jars is not defined here as it can be determined by spring-boot-starter-parent. After configuring Maven project, the pom.xml is as shown below:

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.infyTel</groupId>
<artifactId>demo-jpa-findby</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-jpa-findby</name>
<description>Spring Boot project for using findBy clause in querying using Spring Data
JPA</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Step 1: Create an Entity class "Customer.java" as shown below:

```
package com.infyTel.domain;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
@Column(name = "plan_id")
private Integer planId;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String address,
Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
}
```

Step 2: Create a DTO class "CustomerDTO.java" as shown below:

```
package com.infyTel.dto;
import com.infyTel.domain.Customer;
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String
address, Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
```

```
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static Customer prepareCustomerEntity(CustomerDTO customerDTO)
{
Customer customerEntity = new Customer();
customerEntity.setPhoneNumber(customerDTO.getPhoneNumber());
customerEntity.setName(customerDTO.getName());
customerEntity.setGender(customerDTO.getGender());
customerEntity.setAge(customerDTO.getAge());
customerEntity.setAddress(customerDTO.getAddress());
customerEntity.setPlanId(customerDTO.getPlanId());
return customerEntity;
}
}
```

Step 3: Create an interface "CustomerService.java" as shown below:

```
package com.infyTel.service;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
public interface CustomerService {
public void insertCustomer(CustomerDTO Customer) ;
public Iterable<Customer> getCustomer(String address);
}
```

Step 4: Create a service class "CustomerServiceImpl.java" as shown below:

```
package com.infyTel.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerRepository  repository;
public void insertCustomer(CustomerDTO customer) {
repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public Iterable<Customer> getCustomer(String address) {
return repository.findByAddress(address);
}
}
```

Step 5: Create an interface “CustomerRepository.java” as shown below:

```
package com.infyTel.repository;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import com.infyTel.domain.Customer;
public interface CustomerRepository extends JpaRepository<Customer, Long>{
List<Customer> findByAddress(String address);
}
```

Step 6: Update "application.properties" as shown below:

```
spring.datasource.url = jdbc:mysql://localhost:3306/sample
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.generate-ddl=true
```

Step 7: Create a class “Client.java” as shown below:

```
package com.infyTel;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.service.CustomerService;
@SpringBootApplication
public class Client implements CommandLineRunner {
static Logger logger = Logger.getLogger(Client.class);
@Autowired
ApplicationContext context;
@Autowired
CustomerService service;
public static void main(String[] args) {
SpringApplication.run(Client.class, args);
}
@Override
public void run(String... args) throws Exception {
// TODO Auto-generated method stub
CustomerDTO customer1= new CustomerDTO(7022713754L, "Adam", 27, 'M', "Chicago", 1);
CustomerDTO customer2= new CustomerDTO(7022713744L, "Susan", 27, 'F', "Alberta", 2);
CustomerDTO customer3= new CustomerDTO(7022713745L, "Andrew", 27, 'M', "Chicago", 2);
//invoke service layer method to insert Customer
service.insertCustomer(customer1);
service.insertCustomer(customer2);
service.insertCustomer(customer3);
Iterable<Customer> cus = service.getCustomer("Chicago");
for (Customer alist : cus) {
logger.info(alist);
}
}
}
```

Run the Client.java as "Spring Boot App".

Expected output on the Console:

```
Customer [phoneNumber=7022713745, name=Andrew, age=27, gender=M, address=Chicago, planId=2]
Customer [phoneNumber=7022713754, name=Adam, age=27, gender=M, address=Chicago, planId=1]
```

Exercise: Employee Management Application - Data Access Layer Using Spring Data JPA
Exercise 4:
Problem Statement:
Considering the Employee Management scenario discussed earlier, add the below properties to the existing Employee class:

```
private double employeeSalary;
private String employeeBandLevel;
private String employeeContactNumber;
```

Implement the following requirements to the application:

- Retrieve the employees based on address
- Update the employeeSalary details for the given employeeBandLevel as follows:

employeeBandLevel	employeeSalary
"A"	15%
"B"	10%
"C"	5%

- Add Query creation by method name to retrieve Employee details for the given employeeBandLevel.

Verify database table content and cross-check the data retrieved with the existing database and ensure that they are updated.

Named Queries and Query

Spring Data JPA - Query Creation Using JPA NamedQueries

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.

```
@Entity
```

```
//name starts with the entity class name followed by the method name separated by a dot.
@NamedQuery(name = "Customer.findByAddress", query = "select c from Customer c where
c.address = ?1")
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
--------
}
```

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

```
public interface CustomerRepository extends JpaRepository<Customer, Long>{
Customer findByAddress(String address);
}
```

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.

Spring Data JPA - Query Creation Using @Query Annotation
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**.

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

@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.

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

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:

```
Query query = em.createQuery("Select c.name from CustomerEntity c");
```

- Updating the plan of a customer:

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

- Deleting a customer:

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

Let us understand @Query and @NamedQuery through a demo.

Demo - @NamedQuery and @Query in Spring Data JPA

Demo 6: Application Development Using Spring ORM integration with JPA

Highlights:

- To understand usage of @Namedquery and @Query in Spring Data JPA

Consider the InfyTel scenario and perform the following operations using Spring Data JPA.

- Retrieve customer records based on the address and display the retrieved customer details on the console using @NamedQuery.
- Retrieve customer records based on the address and display the retrieved customer details on the console using @Query.

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

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

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>
```

Note: The version number of the jars is not defined here as it can be determined by spring-boot-starter-parent. Once you have properly configured your maven project our pom.xml will look as below:

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.infyTel</groupId>
<artifactId>demo-jpa-findby</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-jpa-findby</name>
<description>Spring Boot project for using namedquery and query technique using Spring Data JPA</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
```

```
</exclusions>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Step 1: Create an Entity class "Customer.java" as shown below:

```
package com.infyTel.domain;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
//@NamedQuery(name = "Customer.findByAddress", query = "select c from Customer c where
c.address = ?1")
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
@Column(name = "plan_id")
private Integer planId;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String address,
Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
```

```
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
}
```

Step 2: Create a DTO class "CustomerDTO.java" as shown below:

```
package com.infyTel.dto;
import com.infyTel.domain.Customer;
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String
address, Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
   gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static Customer prepareCustomerEntity(CustomerDTO customerDTO)
{
Customer customerEntity = new Customer();
customerEntity.setPhoneNumber(customerDTO.getPhoneNumber());
customerEntity.setName(customerDTO.getName());
customerEntity.setGender(customerDTO.getGender());
customerEntity.setAge(customerDTO.getAge());
customerEntity.setAddress(customerDTO.getAddress());
customerEntity.setPlanId(customerDTO.getPlanId());
return customerEntity;
}
}
```

Step 3: Create an interface "CustomerService.java" as shown below:

```
package com.infytel.service;
import com.infytel.domain.Customer;
import com.infytel.dto.CustomerDTO;
public interface CustomerService {
public void insertCustomer(CustomerDTO customer) ;
public Customer getCustomer(String address);
}
```

Step 4: Create a service class "CustomerServiceImpl.java" as shown below:

```
package com.infytel.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infytel.domain.Customer;
import com.infytel.dto.CustomerDTO;
import com.infytel.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerRepository  repository;
public void insertCustomer(CustomerDTO customer) {
repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public Customer getCustomer(String address) {
return repository.findByAddress(address);
}
}
```

Step 5: Create an interface “CustomerRepository.java” as shown below:

```
package com.infyTel.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import com.infyTel.domain.Customer;
public interface CustomerRepository extends JpaRepository<Customer, Long>{
@Query("select cus from Customer cus where cus.address = ?1")
Customer findByAddress(String address);
}
```

Step 6: Update "application.properties" as shown below:

```
spring.datasource.url = jdbc:mysql://localhost:3306/sample
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.generate-ddl=true
```

Step 7: Create a class “Client.java” as shown below:

```
package com.infyTel;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.service.CustomerService;
@SpringBootApplication
public class Client implements CommandLineRunner {
static Logger logger = Logger.getLogger(Client.class);
@Autowired
ApplicationContext context;
@Autowired
CustomerService service;
public static void main(String[] args) {
SpringApplication.run(Client.class, args);
}
@Override
public void run(String... args) throws Exception {
// TODO Auto-generated method stub
CustomerDTO customer1= new CustomerDTO(7022713754L, "Adam", 27, 'M', "Chicago", 1);
CustomerDTO customer2= new CustomerDTO(7022713744L, "Susan", 27, 'F', "Alberta", 2);
CustomerDTO customer3= new CustomerDTO(7022713745L, "Andrew", 27, 'M', "New York", 2);
CustomerDTO customer4= new CustomerDTO(7022713746L, "Diana", 25, 'F', "Toronto", 1);
CustomerDTO customer5= new CustomerDTO(7022713747L, "Grace", 27, 'F', "Calgary", 1);
service.insertCustomer(customer1);
service.insertCustomer(customer2);
service.insertCustomer(customer3);
service.insertCustomer(customer4);
service.insertCustomer(customer5);
Customer cus=service.getCustomer("Toronto");
if(cus!=null)
logger.info("Customer found: "+cus);
else
logger.info("not found");
}
}
```

Run the Client.java as "Spring Boot App".

Expected output on the Console:

```
Customer found: Customer [phoneNumber=7022713746, name=Diana, age=25, gender=F, address=Toronto, planId=1]
```

Tryout: Comment @Query from EmployeeRepository and uncomment @NamedQuery in Employee class, execute the code and observe the output.

Exercise: Employee Management Application - Namedqueries and query technique Using Spring ORM - JPA

Exercise 5:
Problem Statement:
Develop the data access layer of the Employee Management Application to perform the database operations given below using Spring Data JPA:
Add the operations given below:

- Retrieve employees based on the employee email ID and display employee details on console based on querying.
- Retrieve an employee record based on the email address and display the employee details on the console based on namedquery.
- Retrieve Employee details for the given employeeBandLevel and employeeSalary using @Query annotation of Spring Data JPA.
- Retrieve Employee details for the given emailAddress and employeeContactNumber using @Query annotation with native SQL.

Verify database table content after each operation, cross-check the data retrieved with the existing database and ensure that they are the same.

@Param annotation

Problem Statement:
What is the possibility, if only named parameter is to be used instead of position-based parameter binding in the queries?
Spring supports named parameter using @Param annotation, this annotation is used to provide the concrete name to method parameter and bind the name in the query.

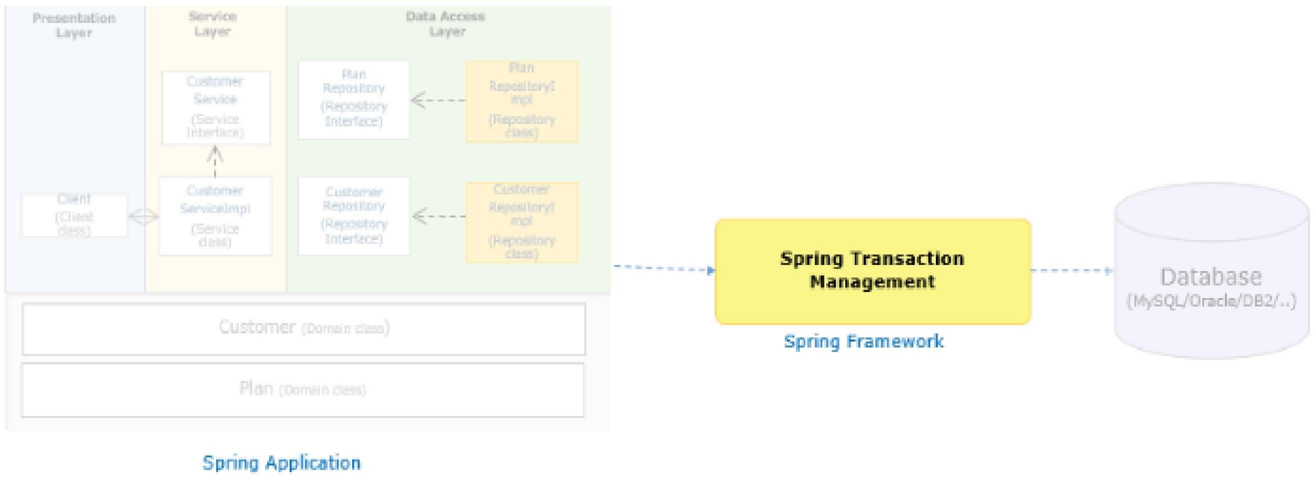
Example: @Param annotation is used to provide names to position parameters.

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
@Query("select e from Employee e where e.firstName = :firstName or e.lastName = :lastName")
List<Employee>  findByLastNameOrFirstName(@Param("lastName") String lastname,
@Param("firstName") String firstname);
}
```

It is advisable to to give parameter names for better readability and documentation purpose.

Spring Transaction

InfyTel application transaction requirement is implemented using Spring Transaction Management.


Presentation Layer
Service Layer
Data Access Layer
Customer Service (Service Interface)
Plan Repository (Repository Interface)
Plan RepositoryImpl (Repository class)
Client (Client class)
Customer ServiceImpl (Service class)
Customer Repository (Repository Interface)
Customer RepositoryImpl (Repository class)
Customer (Domain class)
Plan (Domain class)
Spring Application
Spring Transaction Management
Spring Framework
Database (MySQL/Oracle/DB2/..)

Why Spring 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.

Spring Declarative Transaction:

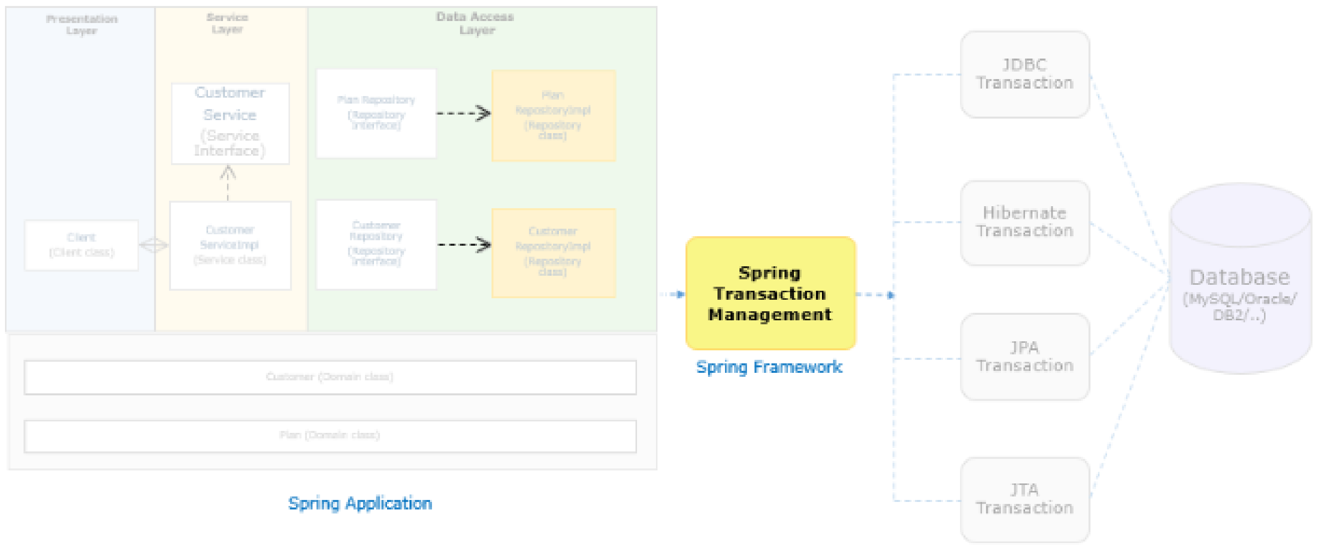
What is Spring Transaction?

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.


Presentation Layer
Service Layer
Data Access Layer
Customer Service (Service Interface)
Plan Repository (Repository Interface)
Plan RepositoryImpl (Repository class)
Client (Client class)
Customer ServiceImpl (Service class)
Customer Repository (Repository Interface)
Customer RepositoryImpl (Repository class)
Customer (Domain class)
Plan (Domain class)
Spring Application
Spring Transaction Management
Spring Framework
JDBC Transaction
Hibernate Transaction
JPA Transaction
JTA Transaction
Database (MySQL/Oracle/ DB2/..)

Spring 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 • 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) • 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.

Application Development Using Spring Declarative Transaction

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.

Declarative Transaction configurations

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) {
------------------
}
}
```

Spring Declarative Transaction Management - Transaction Managers

The key interface to implement the Spring transaction is org.springframework.transaction.PlatformTransactionManager Interface.

There are different implementations available for PlatformTransactionManager to support different data access technologies such as JDBC, JTA, Hibernate, JPA and so on.

The PlatformTransactionManager implementations are responsible for managing transactions. Hence, they are transaction managers.

The following table describes some of the commonly used transaction managers:

TransactionManager	**Data Access Technique**
DataSourceTransactionManager	JDBC
HibernateTransactionManager	Hibernate
JpaTransactionManager	JPA
JtaTransactionManager	Distributed transaction

Transaction managers can be defined as any other beans in the Spring configuration only if we are not using Spring Boot.

Since the InfyTel application is developed using Spring Boot and Spring Data JPA, So no bean needed to be defined explicitly.

Spring Transaction in Spring Data JPA

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.

Demo - Spring Transaction Management in Spring Data JPA

Demo 7: Application Development with Spring Transaction using Spring Data JPA

Highlights:

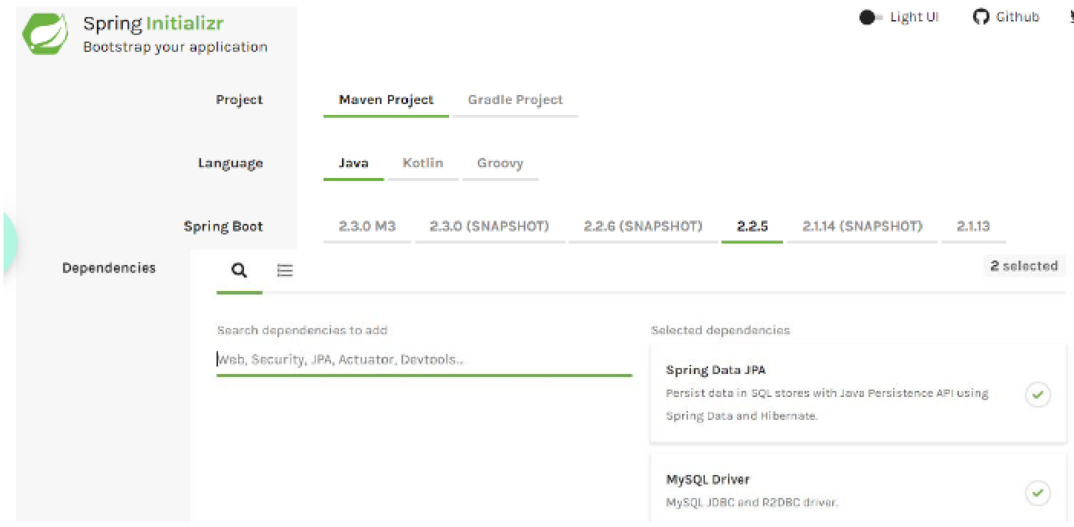
- To understand how to apply Spring transaction in Spring Data JPA application
- Understand required configuration details

Demo steps:

Let us now implement the transaction scenario of InfyTel application to perform the operation given below using Spring Data JPA with Spring Transactions:

- Insert Customer details.
- Update the Customer’s current plan and Plan details.

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



Project: Maven Project | Gradle Project

Language: Java | Kotlin | Groovy

Spring Boot: 2.3.0 M3 | 2.3.0 (SNAPSHOT) | 2.2.6 (SNAPSHOT) | 2.2.5 | 2.1.14 (SNAPSHOT) | 2.1.13

Dependencies — 2 selected

Search dependencies to add: Web, Security, JPA, Actuator, Devtools..

Selected dependencies

Spring Data JPA — Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate. ✓

MySQL Driver — MySQL JDBC and R2DBC driver. ✓

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>
```

Note: The version number of the jars is not defined here as it can be determined by spring-boot-starter-parent. Once you have properly configured your maven project our pom.xml will look as below:

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath /> <!-- lookup parent from repository -->
</parent>
```

```
<groupId>com.infytel</groupId>
<artifactId>demo-jpa-transaction</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-spring-data-jpa-transaction</name>
<description>Spring Boot project to demonstrate Transaction
using Spring Data JPA
</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
<exclusions>
<exclusion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-logging</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
</dependencies>
```

```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Steps to implement the application:

Step 1: Create an entity class "Customer.java" as shown below:

```
package com.infytel.entity;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import com.infytel.dto.CustomerDTO;
import com.infytel.dto.PlanDTO;
@Entity
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private String currentPlan;
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name ="plan_name")
private Plan plan;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String
address,String currentPlan, Plan plan) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.currentPlan=currentPlan;
this.plan = plan;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Plan getPlan() {
return plan;
}
public void setPlan(Plan plan) {
this.plan = plan;
}
public String getCurrentPlan() {
return currentPlan;
```

```
}
public void setCurrentPlan(String currentPlan) {
this.currentPlan = currentPlan;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", currentPlan=" + currentPlan + ", plan=" + plan + "]";
}
public static CustomerDTO prepareDTO(Customer custEntity)
{
Plan plan = new Plan(custEntity.getPlan().getPlanId(), custEntity.getPlan().getPlanName(),
custEntity.getPlan().getLocalRate(), custEntity.getPlan().getNationalRate());
PlanDTO planDTO = Plan.preparePlaDTO(plan);
return new CustomerDTO(custEntity.getPhoneNumber(), custEntity.getName(), custEntity.getAge(),
custEntity.getGender(), custEntity.getAddress(), custEntity.getCurrentPlan(), planDTO);
}
}
```

Step 2: Create an entity class "Plan.java" as shown below:

```
package com.infytel.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import com.infytel.dto.PlanDTO;
@Entity
public class Plan {
@Id
@Column(name = "plan_id")
private Integer planId;
@Column(name = "plan_name")
private String planName;
@Column(name = "local_rate")
private Integer localRate;
@Column(name = "national_rate")
private Integer nationalRate;
public Plan() {
super();
}
public Plan(Integer planId, String planName, Integer localRate, Integer nationalRate) {
super();
this.planId = planId;
this.planName = planName;
this.localRate = localRate;
this.nationalRate = nationalRate;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
public String getPlanName() {
return planName;
}
public void setPlanName(String planName) {
this.planName = planName;
}
public Integer getLocalRate() {
return localRate;
}
public void setLocalRate(Integer localRate) {
this.localRate = localRate;
}
public Integer getNationalRate() {
return nationalRate;
}
public void setNationalRate(Integer nationalRate) {
this.nationalRate = nationalRate;
}
@Override
public String toString() {
return "Plan [plainId=" + planId + ", planName=" + planName + ", localRate=" + localRate + ",
nationalRate="+ nationalRate + "]";
}
public static PlanDTO preparePlaDTO(Plan planEntity)
{
return new PlanDTO(planEntity.getPlanId(), planEntity.getPlanName(), planEntity.getLocalRate(),
planEntity.getNationalRate());
}
}
```

Step 3: Create a DTO class “CustomerDTO.java” as shown below:

```
package com.infytel.dto;
import com.infytel.entity.Customer;
import com.infytel.entity.Plan;
```

```
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private String currentPlan;
private PlanDTO plan;
public CustomerDTO() {}
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String
address,String currentPlan, PlanDTO plan) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.currentPlan=currentPlan;
this.plan = plan;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
```

```
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public PlanDTO getPlan() {
return plan;
}
public void setPlan(PlanDTO plan) {
this.plan = plan;
}
public String getCurrentPlan() {
return currentPlan;
}
public void setCurrentPlan(String currentPlan) {
this.currentPlan = currentPlan;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", currentPlan=" + currentPlan + ", plan=" + plan + "]";
}
public static Customer prepareEntity(CustomerDTO custDTO)
{
Plan plan = new Plan(custDTO.getPlan().getPlanId(), custDTO.getPlan().getPlanName(),
custDTO.getPlan().getLocalRate(), custDTO.getPlan().getNationalRate());
return new Customer(custDTO.getPhoneNumber(),custDTO.getName(), custDTO.getAge(),
custDTO.getGender(), custDTO.getAddress(), custDTO.getCurrentPlan(), plan);
}
}
```

Step 4: Create a DTO class “PlanDTO.java” as shown below:

```
package com.infytel.dto;
import com.infytel.entity.Plan;
public class PlanDTO {
private Integer planId;
private String planName;
private Integer localRate;
private Integer nationalRate;
public PlanDTO() {
```

```
super();
}
public PlanDTO(Integer planId, String planName, Integer localRate, Integer nationalRate) {
super();
this.planId = planId;
this.planName = planName;
this.localRate = localRate;
this.nationalRate = nationalRate;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
public String getPlanName() {
return planName;
}
public void setPlanName(String planName) {
this.planName = planName;
}
public Integer getLocalRate() {
return localRate;
}
public void setLocalRate(Integer localRate) {
this.localRate = localRate;
}
public Integer getNationalRate() {
return nationalRate;
}
public void setNationalRate(Integer nationalRate) {
this.nationalRate = nationalRate;
}
@Override
public String toString() {
return "Plan [plainId=" + planId + ", planName=" + planName + ", localRate=" + localRate + ",
nationalRate=" + nationalRate + "]";
}
public static Plan preparePlanEntity(PlanDTO planDTO)
{
return new Plan(planDTO.getPlanId(), planDTO.getPlanName(), planDTO.getLocalRate(),
planDTO.getNationalRate());
```

```
}
}
```

Step 5: Create an interface “CustomerRepository.java” as shown below:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
// Method to update a Customer record into the db
@Transactional
@Modifying
@Query(value = "update  Customer set plan_name = ? " + " where phone_No = ?", nativeQuery =
true)
public void customerUpdate(Plan plan, long phoneNumber);
}
```

Step 6: Create an interface “PlanRepository.java” as shown below:

```
package com.infytel.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import com.infytel.entity.Plan;
public interface PlanRepository extends JpaRepository<Plan, Integer>{
//Method to update Plan details in DB
@Transactional
@Modifying
@Query(value = "update  Plan set local_rate = ?, national_rate=? " + " where plan_id = ?",
nativeQuery = true)
public void updatePlan(Integer localRate, Integer nationalRate, Integer planId);
}
```

Step 7: Create an interface “CustomerService.java” as shown below:

```
package com.infytel.service;
import com.infytel.dto.CustomerDTO;
public interface CustomerService {
// Method to access the repository layer method to insert Customer record
public void insert(CustomerDTO customerDTO);
// Method to update a Customer record into the db
public void updateCustomer(CustomerDTO customerDTO);
}
```

Step 8: Create a service class “CustomerServiceImpl.java” as shown below:

```
package com.infytel.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.infytel.dto.CustomerDTO;
import com.infytel.dto.PlanDTO;
import com.infytel.repository.CustomerRepository;
import com.infytel.repository.PlanRepository;
@Transactional
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
@Autowired
CustomerRepository customerRepository;
@Autowired
PlanRepository planRepository;
@Override
public void insert(CustomerDTO customerDTO) {
customerRepository.saveAndFlush(CustomerDTO.prepareEntity(customerDTO));
}
@Override
//Update on Customer and Plan table are executed in transaction scope
public void updateCustomer(CustomerDTO customerDTO) {
//Method to update the current plan in Customer table
customerRepository.customerUpdate(PlanDTO.preparePlanEntity(customerDTO.getPlan()),customer
DTO.getPhoneNumber());
//Method to update the new plan details in Plan table
planRepository.updatePlan(CustomerDTO.prepareEntity(customerDTO).getPlan().getLocalRate(),Cu
stomerDTO.prepareEntity(customerDTO).getPlan().getNationalRate(),CustomerDTO.prepareEntity(c
ustomerDTO).getPlan().getPlanId());
}
}
```

Step 9: Update "application.properties" as shown below:

```
#SQL
spring.datasource.url = jdbc:mysql://localhost:3306/sample
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.generate-ddl=true
#Logging
logging.level.root=INFO
#logging.pattern.console = %d{yyyy-MM-dd HH:mm:ss,SSS}
logging.pattern.file = %5p [%t] %c [%M] - %m%n
```

Step 10: Create a User Interface class "Client.java" as shown below:

```
package com.infytel;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.dao.DataAccessException;
import com.infytel.dto.CustomerDTO;
import com.infytel.dto.PlanDTO;
import com.infytel.service.CustomerService;
@SpringBootApplication
public class Client implements CommandLineRunner {
@Autowired
AbstractApplicationContext context;
static Logger logger = Logger.getLogger(Client.class);
public static void main(String[] args) {
SpringApplication.run(Client.class, args);
}
@Override
public void run(String... args) throws Exception {
CustomerService customerService = (CustomerService)context.getBean("customerService");
PlanDTO plan1 = new PlanDTO(1,"INFY_60",60,60);
CustomerDTO customer1= new CustomerDTO(8009009010L, "Mary", 27, 'F', "Colorado","INFY",
plan1);
customerService.insert(customer1);
logger.info("Records are successfully added..");
Long phoneNo=8009009009L;
String newCurrentPlan= "INFY000000";
PlanDTO plan2=new PlanDTO(1,"INFY_100",100,120);
CustomerDTO customer2= new CustomerDTO();
customer2.setPhoneNumber(phoneNo);
customer2.setAddress("Colorado");
customer2.setAge(27);
customer2.setGender('F');
customer2.setName("Mary");
customer2.setCurrentPlan(newCurrentPlan);
customer2.setPlan(plan2);
try {
// Method to update customer plane name and Plan table
customerService.updateCustomer(customer2);
logger.info("Success : Both Customer and Plan details updated successfully!");
} catch (DataAccessException exp) {
logger.error("ERROR : "+exp.getMessage());
logger.error(exp.getMessage(),exp);
}
finally
{
context.close();
}
}
}
```

Run the Client.java as "Spring Boot App".

Expected Output on the Console:

Case 1: If both the tables exist in the database:

Success : Both Customer and Plan detail updated successfully!

Note : You can verify the updates in both the tables in the database.

Case 2: If the Plan table doesn't exist in the database:

```
org.hibernate.exception.SQLGrammarException: could not extract ResultSet

caused by: java.sql.SQLSyntaxErrorException: Table 'sample.plan' doesn't exist

ERROR : could not extract ResultSet; SQL [n/a]; nested exception is
org.hibernate.exception.SQLGrammarException: could not extract ResultSet
```

Note : You can verify the updates in the Customer table has not reflected.

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.

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

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.

More on @Modifying

@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:

```
@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:

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

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:

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

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

```
public class CustomerServiceImpl implements CustomerService {
Customer customerTom = customerRepository.findById(1); // Stored in the First Level Cache
customerRepository.deleteInActiveCustomers(); // We think that customerTom is deleted now
System.out.println(customerRepository.findById(1).isPresent()) // Will return TRUE
System.out.println(customerRepository.count()) // Will return 1
// TOM still exist in this transaction persistence context
// TOM's object was not cleared upon @Modifying query execution
// TOM's object will still be fetched from First Level Cache
/* clearAutomatically attribute takes care of doing the clear part on the objects being modified for current transaction persistence context*/
}
```

Demo- Update operation Using Spring Data JPA

Demo 8: Application Development Using Spring Data JPA

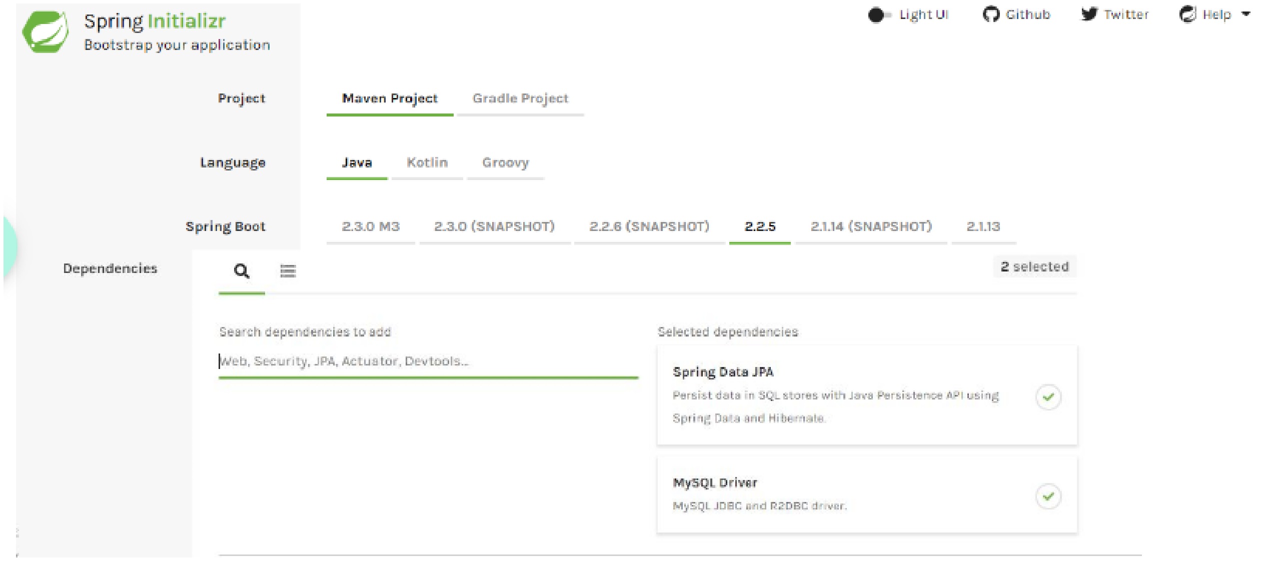
Highlights:

- To understand how to update Customer details in Spring Data JPA.

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

- Update the customer record

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



Project: Maven Project | Gradle Project
Language: Java | Kotlin | Groovy
Spring Boot: 2.3.0 M3 | 2.3.0 (SNAPSHOT) | 2.2.6 (SNAPSHOT) | 2.2.5 | 2.1.14 (SNAPSHOT) | 2.1.13
Dependencies: 2 selected
Search dependencies to add: Web, Security, JPA, Actuator, Devtools...
Selected dependencies
Spring Data JPA: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
MySQL Driver: MySQL JDBC and R2DBC driver.

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

Spring Data JPA Dependency:

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

```xml
</dependency>
```

MySQL Driver dependency:

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

Note: The version number of the jars is not defined here as it can be determined by spring-boot-starter-parent. Once you have properly configured your maven project our pom.xml will look as below:

pom.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.infyTel</groupId>
<artifactId>demo-jpa-update</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-jpa-update</name>
<description>Spring Boot project for update operation using Spring Data JPA</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
```

```
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Step 1: Create an Entity class "Customer.java" as shown below:

```
package com.infyTel.domain;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Transient;
@Entity
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
@Transient
```

```
@Column(name = "plan_id")
private Integer planId;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String address,
Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
}
```

Step 2: Create a DTO class "CustomerDTO.java" as shown below:

```
package com.infyTel.dto;
import com.infyTel.domain.Customer;
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String
address, Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
```

```
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static Customer prepareCustomerEntity(CustomerDTO customerDTO)
{
Customer customerEntity = new Customer();
customerEntity.setPhoneNumber(customerDTO.getPhoneNumber());
customerEntity.setName(customerDTO.getName());
customerEntity.setGender(customerDTO.getGender());
customerEntity.setAge(customerDTO.getAge());
customerEntity.setAddress(customerDTO.getAddress());
customerEntity.setPlanId(customerDTO.getPlanId());
return customerEntity;
}
}
```

Step 3: Create an interface "CustomerService.java" as shown below:

```
package com.infyTel.service;
public interface CustomerService {
public void insertCustomer(CustomerDTO Customer) ;
public void insert();
public void addressUpdate(String address, Long phoneNumber);
}
```

Step 4: Create a service class "CustomerServiceImpl.java" as shown below:

```
package com.infyTel.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infyTel.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerRepository  repository;
public void insertCustomer(CustomerDTO customer) {
repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public void insert() {
repository.insert();
}
@Override
public void addressUpdate(String address, Long phoneNumber) {
repository.addressUpdate(address, phoneNumber);
}
}
```

Step 5: Create a repository class “CustomerRepository.java” as shown below:

```
package com.infyTel.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.transaction.annotation.Transactional;
import com.infyTel.domain.Customer;
public interface CustomerRepository extends JpaRepository<Customer, Long>{
@Transactional
/* clearAutomatically = true is to clear entity manager automatically, drop all pending changes
that have not been flushed to database yet. This attribute should be used carefully otherwise will lose
any pending changes*/
@Modifying(clearAutomatically=true, flushAutomatically=true)
@Query(value="INSERT into Customer(phone_no,name,age,gender,address) values (7022713753,
Alex', 27, 'M', 'Chicago')", nativeQuery = true)
void insert();
@Transactional
@Modifying(clearAutomatically=true, flushAutomatically=true)
@Query(value = "update  Customer set address = ? " + " where phone_no = ?", nativeQuery = true)
void addressUpdate(String address,long phoneNumber);
}
```

Step 6: Update "application.properties" as shown below:

```
spring.datasource.url = jdbc:mysql://localhost:3306/sample
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.generate-ddl=true
```

Step 7: Create a class “Client.java” as shown below:

```
package com.infyTel;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.infyTel.repository.CustomerRepository;
@SpringBootApplication
public class Client implements CommandLineRunner {
static Logger logger = Logger.getLogger(Client.class);
@Autowired
ApplicationContext context;
@Autowired
CustomerService service;
public static void main(String[] args) {
SpringApplication.run(Client.class, args);
}
@Override
public void run(String... args) throws Exception {
service.insert();
service.addressUpdate("Toronto",7022713754L);
logger.info("address updated");
}
25. }
```

Run the Client.java as "Spring Boot App".

Expected output on the Console:
address updated
Note: Confirm the changes in the database

More About Spring Declarative Transaction - @Transactional Attributes

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.

More About Spring Declarative Transaction - @Transactional Read-Only Attribute

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

More About Spring Declarative Transaction - @Transactional Read-Only Attribute

Read-only status

The readOnly attribute specifies that the transaction will only read data from a database.

A read-only transaction can be used when your code reads but does not modify data.

The advantage is that the database may apply certain optimization to the transaction when it is declared to be read-only. Eg: Some cases where Hibernate is used.

By default, @Transactional is read/write

Example: In the below-given code, the getCustomer() method executes in transaction scope with read-only.

```
@Transactional
public class CustomerServiceImpl {
-----------
// use of @Transactional at method level overrides class level annotation
@Transactional(readOnly = true)
public Customer getCustomer(int custId) {
------------------
}
public void updateCustomerDetails(Customer customer) {
------------------
}
}
```

More About Spring Declarative Transaction - @Transactional timeout Attribute

In the transaction scenario of the InfyTel application, updating the plan and customer method should not take an indefinite time to complete the transaction.
The transaction timeout has to be set within which the transaction must be completed for better application performance.
Spring supports the timeout attribute for transactions. A transaction timeout can be configured that determines the maximum time (in seconds) within which a transaction has to commit, failing which the transaction will be rolled back.

Example 1: Transaction is rolled back if updatePlanmethod is not completed within 40 seconds.

```
public class PlanServiceImpl implements PlanService {
-----------
//timeout attribute value set to 40sec
@Transactional(timeout = 40)
public void updatePlan(PlanDTO newPlanDTO) throws RuntimeException{
```

```
------------------
}
}
```

Example 2: Overriding timeout set at class level

```
//timeout specified at class level is applicable to all methods
@Transactional(timeout = 60)
public class PlanServiceImpl implements PlanService {
-----------
//overrides timeout attribute value to 40sec at method level
@Transactional(timeout = 40)
public void updatePlan(PlanDTO newPlanDTO) throws RuntimeException{
------------------
}
}
```

@Transactional default rollback behavior

Spring automatically rolls back the transaction for run time exception and proceeds to commit the transactions for checked exceptions.
Overriding default rollback behavior: Use the rollbackFor and noRollbackFor attributes of @Transactional annotation to override the default rollback behavior.

Example 1:

```
public class CustomerServiceImpl {
-----------
//Transaction will not be rollback automatically, in case of any run time exception.
@Transactional(noRollbackFor = RuntimeException.class)
public Customer getCustomer(Customer customer) {
------------------
}
}
```

Example 2:

```
public class CustomerServiceImpl {
-----------
// Transaction will rollback for the user defined exception of type MyException.
@Transactional(rollbackFor = MyException.class)
public void updateCustomerDetails(Customer customer) {
------------------
}
}
```

More About Spring Declarative Transaction - @Transactional Default Settings

The default @Transactional settings are as follows:

@Transactional Attribute	**Default values**
Propagation	PROPOGATION_REQUIRED
Isolation level	ISOLATION_DEFAULT
Transaction Type	Read/Write
Timeout	Based on the underlying database settings
Rollback condition	Rollback on runtime exception and no rollback on checked exception

Custom Repository Implementation

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.

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
```

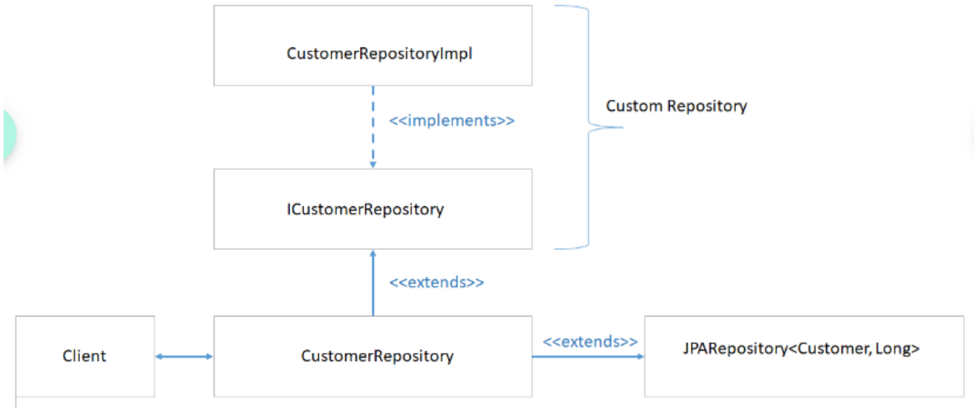
```
public List<Customer> searchCustomer(String name, String address, Character gender, Integer age) {
EntityManager em = emf.createEntityManager();
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> query = builder.createQuery(Customer.class);
Root<Customer> root = query.from(Customer.class);
Predicate cName = builder.equal(root.get("name"), name);
Predicate cAddress = builder.equal(root.get("address"), address);
Predicate exp1 = builder.and(cName, cAddress);
Predicate cGender = builder.equal(root.get("gender"), gender);
Predicate cAge = builder.equal(root.get("age"), age);
Predicate exp2 = builder.or(cGender,cAge);
query.where(builder.or(exp1, exp2));
return em.createQuery(query.select(root)).getResultList();
}
```

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

```
public interface CustomerRepository extends JpaRepository<Customer, Long>,ICustomerRepository{
}
```

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.

Data access layer of an application has the following files with dependencies is as shown below:

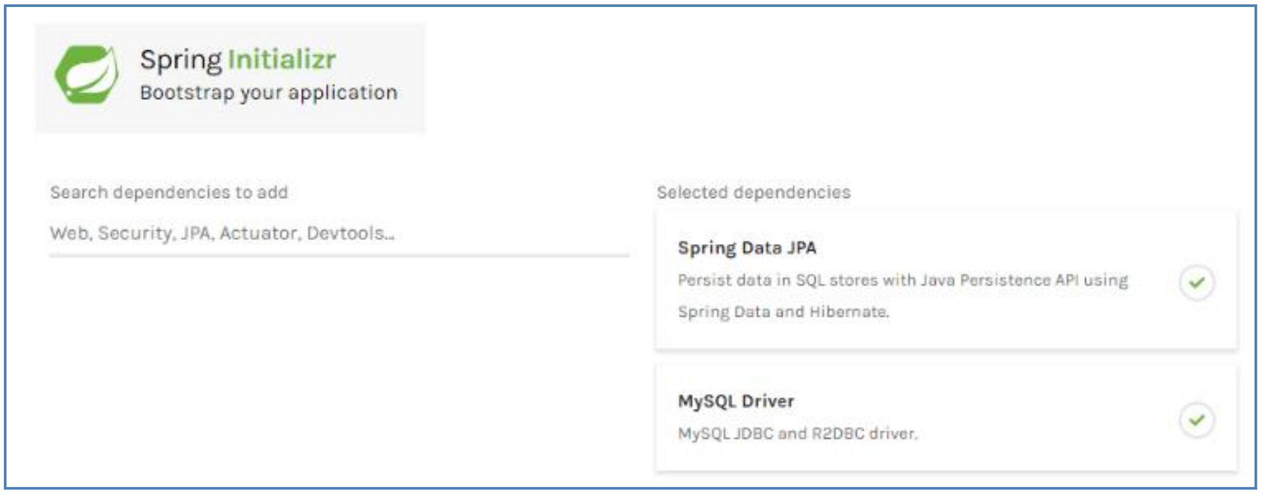


Demo - Custom Repository Implementation using Spring Data JPA
Demo 8: Application Development Using Spring Data JPA
Highlights:

- To understand how custom repositories are supported in Spring Data JPA.
- To know how to implement custom repositories.

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

- Retrieve customer records based on name and address or gender or age


Spring Initializr
Bootstrap your application
Search dependencies to add
Web, Security, JPA, Actuator, Devtools...
Selected dependencies

Spring Data JPA
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver
MySQL JDBC and R2DBC driver.

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

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>
```

Note: The version number of the jars is not defined here as it can be determined by spring-boot-starter-parent. After configuring Maven project, the pom.xml will be as shown below:

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
```

```
<groupId>com.infyTel</groupId>
<artifactId>demo-jpa-custom-repository</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-jpa-custom-repository</name>
<description>Spring Boot project to implement custom repository using Spring Data
JPA</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

```
</plugins>
</build>
</project>
```

Step 1: Create an Entity class "Customer.java" as shown below:

```
package com.infyTel.domain;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
@Column(name = "plan_id")
private Integer planId;
public Customer() {}
public Customer(Long phoneNumber, String name, Integer age, Character gender, String address,
    Integer planId) {
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
}
```

Step 2: Create a DTO class "CustomerDTO.java" as shown below:

```
package com.infyTel.dto;
import com.infyTel.domain.Customer;
public class CustomerDTO {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
public CustomerDTO(Long phoneNumber, String name, Integer age, Character gender, String
address, Integer planId) {
```

```
super();
this.phoneNumber = phoneNumber;
this.name = name;
this.age = age;
this.gender = gender;
this.address = address;
this.planId = planId;
}
public Long getPhoneNumber() {
return phoneNumber;
}
public void setPhoneNumber(Long phoneNumber) {
this.phoneNumber = phoneNumber;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public Character getGender() {
return gender;
}
public void setGender(Character gender) {
this.gender = gender;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}
public Integer getPlanId() {
return planId;
}
public void setPlanId(Integer planId) {
```

```
this.planId = planId;
}
@Override
public String toString() {
return "Customer [phoneNumber=" + phoneNumber + ", name=" + name + ", age=" + age + ",
gender=" + gender + ", address=" + address + ", planId=" + planId + "]";
}
public static Customer prepareCustomerEntity(CustomerDTO customerDTO)
{
Customer customerEntity = new Customer();
customerEntity.setPhoneNumber(customerDTO.getPhoneNumber());
customerEntity.setName(customerDTO.getName());
customerEntity.setGender(customerDTO.getGender());
customerEntity.setAge(customerDTO.getAge());
customerEntity.setAddress(customerDTO.getAddress());
customerEntity.setPlanId(customerDTO.getPlanId());
return customerEntity;
}
}
```

Step 3: Create an interface "CustomerService.java" as shown below:

```
package com.infyTel.service;
import java.util.List;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
public interface CustomerService {
public void insertCustomer(CustomerDTO Customer) ;
public List<Customer> searchCustomer(String name, String address, Character gender, Integer age) ;
}
```

Step 4: Create a service class "CustomerServiceImpl.java" as shown below:

```
package com.infyTel.service;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerRepository  repository;
public void insertCustomer(CustomerDTO customer) {
```

```
repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public List<Customer> searchCustomer(String name, String address, Character gender, Integer age) {
return repository.searchCustomer(name, address, gender, age);
}
}
```

Step 5: Create custom interface “ICustomerRepository.java” as shown below:

```
package com.infyTel.repository;
import java.util.List;
import com.infyTel.domain.Customer;
public interface ICustomerRepository {
public List<Customer> searchCustomer(String name, String addr, Character gender, Integer age);
}
```

Step 6: Create an implementation class "CustomerRepositoryImpl.java" of custom interface (ICustomerRepository) as shown below:

```
package com.infyTel.repository;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Predicate;
import javax.persistence.criteria.Root;
import org.springframework.beans.factory.annotation.Autowired;
import com.infyTel.domain.Customer;
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();
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> query = builder.createQuery(Customer.class);
Root<Customer> root = query.from(Customer.class);
Predicate cName = builder.equal(root.get("name"), name);
Predicate cAddress = builder.equal(root.get("address"), address);
Predicate exp1 = builder.and(cName, cAddress);
Predicate cGender = builder.equal(root.get("gender"), gender);
Predicate cAge = builder.equal(root.get("age"), age);
Predicate exp2 = builder.or(cGender,cAge);
//Predicate dept = builder.equal(root.get("dept"), department);
query.where(builder.or(exp1, exp2));
return em.createQuery(query.select(root)).getResultList();
}
}
```

Step 7: Create a standard repository interface "CustomRepository.java" to support Spring Data JPA methods along with custom methods as below:

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

Step 8: Update "application.properties" as shown below:

```
spring.datasource.url = jdbc:mysql://localhost:3306/sample
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.generate-ddl=true
```

Step 9: Create a class “Client.java” as shown below:

```
package com.infytel;
import java.util.List;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.infytel.domain.Customer;
import com.infytel.dto.CustomerDTO;
import com.infytel.service.CustomerService;
@SpringBootApplication
public class Client implements CommandLineRunner {
static Logger logger = Logger.getLogger(Client.class);
@Autowired
ApplicationContext context;
@Autowired
CustomerService service;
public static void main(String[] args) {
SpringApplication.run(Client.class, args);
}
@Override
public void run(String... args) throws Exception {
CustomerDTO customer1 = new CustomerDTO(7022713754L, "Adam", 27, 'M', "Chicago", 1);
CustomerDTO customer2 = new CustomerDTO(7022713744L, "Susan", 25, 'F', "Alberta", 2);
CustomerDTO customer3 = new CustomerDTO(7022713745L, "Andrew", 27, 'M', "Chicago", 2);
// invoke service layer method to insert Customer
service.insertCustomer(customer1);
service.insertCustomer(customer2);
service.insertCustomer(customer3);
List<Customer> cus1 = service.searchCustomer(null, null, 'F', null);
logger.info(cus1);
List<Customer> cus2 = service.searchCustomer("Adam", null, 'M', null);
logger.info(cus2);
List<Customer> cus3 = service.searchCustomer(null, null,null , 27);
logger.info(cus3);
List<Customer> cus4 = service.searchCustomer("Susan","Alberta",null, null);
logger.info(cus4);
}
}
```

Run the Client.java as "Spring Boot App".

Expected output on the Console:

```
[Customer [phoneNumber=7022713744, name=Susan, age=25, gender=F, address=Alberta, planId=2]]
[Customer [phoneNumber=7022713745, name=Andrew, age=27, gender=M, address=Chicago, planId=2],
Customer [phoneNumber=7022713754, name=Adam, age=27, gender=M, address=Chicago, planId=1]]
[Customer [phoneNumber=7022713745, name=Andrew, age=27, gender=M, address=Chicago, planId=2],
Customer [phoneNumber=7022713754, name=Adam, age=27, gender=M, address=Chicago, planId=1]]
[Customer [phoneNumber=7022713744, name=Susan, age=25, gender=F, address=Alberta, planId=2]]
```

Note:
In this demo, criteria API is used for filtering.

Spring Boot Best Practices

Some of the best practices that need to be followed as part of the Quality and Security for Spring Boot applications. These practices, when applied during designing and developing a Spring Boot application, yields better performance.

Best Practices:

1. Use Spring Initializr for creating Spring Boot projects
2. Use the correct project Structure while creating Spring Boot projects
3. Choose Java-based configuration over XML based configuration
4. Use Setter injection for optional dependencies and Constructor injection for mandatory dependencies

5. Use @Service for Business Layer classes
6. Follow the Spring bean naming conventions while creating beans

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

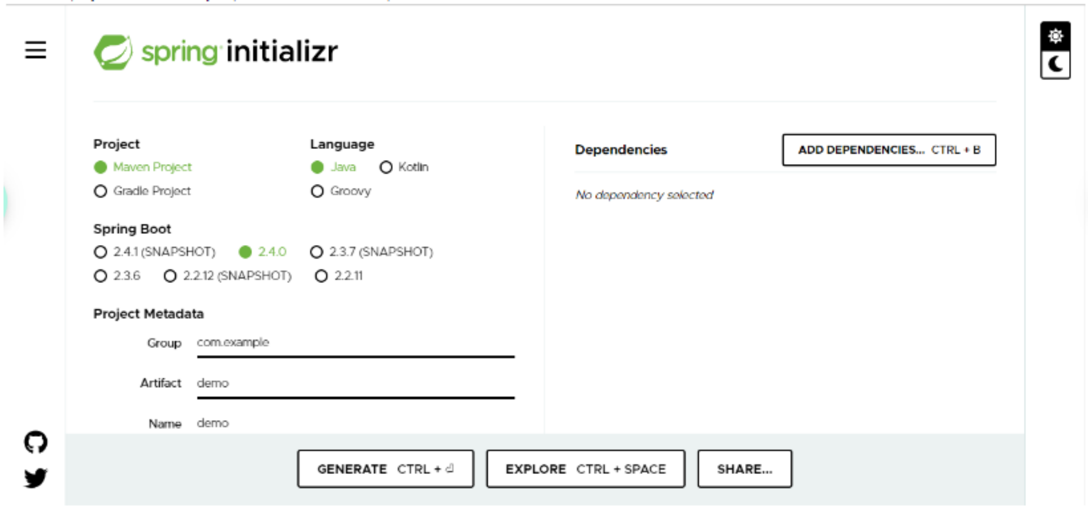
Note: For the demos already covered in the course, we would be applying these best practices.

Use Spring Initializr for starting new Spring Boot projects

There are three different ways to generate a Spring Boot project. They are:

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

But the recommended and simplest way to create a Spring Boot application is the Spring Boot Initializr as it has good UI to download a production-ready project. And the same project can be directly imported into the STS/Eclipse.



Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.4.1 (SNAPSHOT) ● 2.4.0 ○ 2.3.7 (SNAPSHOT)
○ 2.3.6 ○ 2.2.12 (SNAPSHOT) ○ 2.2.11

Project Metadata
Group com.example
Artifact demo
Name demo

Dependencies
No dependency selected

Note: The above screen keeps changing depending on updates from Pivotal and changes in the Spring Boot version.

Standard Project Structure for Spring Boot Projects

To ease of use the Developers can follow two ways of creating the package structure in a Spring Boot application, which will also help them to maintain the application in the future.

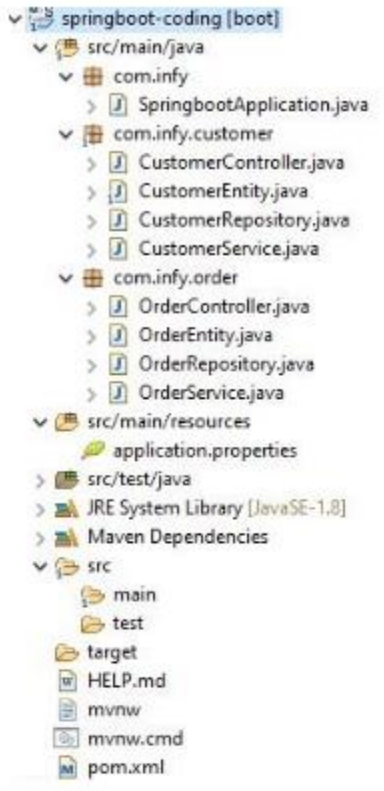
Before delving deeper let us discuss if a Developer is not providing any package structure. i.e. "default" package for all the classes and its disadvantages.

“default” Package:

When the Developer is not providing the package declaration at the class level, then the class will be in the "default" package. This approach should be avoided because it can cause problems for the Boot applications that frequently use the annotations like @EntityScan, @ComponentScan, or @SpringBootApplication and the Spring Boot application has to mandatorily scan all the classes in the "default" package instead of the required classes which will decrease the performance of the application. So it is recommended to create the appropriate classes inside appropriate packages with proper Java's package naming conventions. For example, com.infosys.utilities, com.infosys.client, com.infosys.service, etc.

So the developer can follow either one of the two recommended approaches.

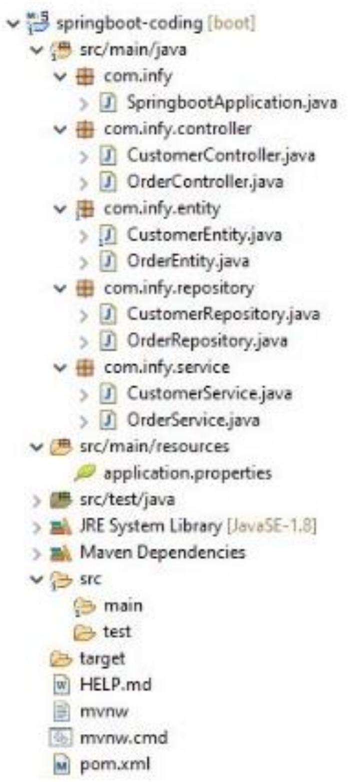
First approach: The first approach shows a layout which generally recommended by the Spring Boot team. In this approach, all the related classes for the Customer have grouped in the **"com.infy.customer"** package and all the related classes for the Order have grouped in the **"com.infy.order"** package



```
springboot-coding [boot]
  src/main/java
    com.infy
      SpringbootApplication.java
    com.infy.customer
      CustomerController.java
      CustomerEntity.java
      CustomerRepository.java
      CustomerService.java
    com.infy.order
      OrderController.java
      OrderEntity.java
      OrderRepository.java
      OrderService.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
    main
    test
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Second approach: The above-shown approach works fine but the developers can follow the second approach as that's better readable and maintainable.

In this approach, all the service classes related to Customer and Order are grouped in the **"com.infy.service"** package. Similar to that we can see we grouped the **dao**, **controller**, and **model** classes.



Note: In our course, we are following the second approach.

Using Java-based configuration - @Configuration

Spring Boot gives us the flexibility to configure the project either using the .properties file or the .java file or the .xml file. So for a large multitier application if configuration related code(apart from Spring Boot auto-configuration) it's a good practice to write the configurations in a Java-based configuration file and divide them into multiple files as per the developer's convenience.

Writing the configurations in a Java-based configuration file serves compile-time checking and dividing them into multiple files serves more readability and maintainability.

So it's advisable to keep DAO related configurations in one file (DAOConfig.java). Web configurations in another file (WebConfig.java) which finally can be imported in the bootstrap class or main configuration file.

Let's analyze more on the above said best practice.

The developer needs to follow certain steps to put all the **@Configuration** files into a single **@SpringBootApplication** file OR **@Configuration** file(centralized configuration file). So, for putting all the configuration files into a single file the developer needs to use **@Import** annotation.

Let's discuss the steps.

Step 1: Create multiple configuration files using the Java-based configuration.

```
@Configuration
public class DAOConfig {
//Database related configurations
}



@Configuration
public class WebConfig {
//Web related configurations
}
```

Step 2: Multiple configuration files can be imported to a single centralized configuration file or to the bootstrap class of the Spring Boot application.

```
@Configuration
@Import({ DAOConfig.class, WebConfig.class })
public class ApplicationConfig extends ConfigurationSupport {
//This is a centralized class containing all the Configurations
/*All the @Bean methods from DAOConfig and WebConfig will be referred to in this configuration
class along with the @Bean methods in ApplicationConfig class*/
}
```

OR

```
@SpringBootApplication
@Import({ DAOConfig.class, WebConfig.class })
public class AppConfig implements CommandLineRunner {
//Bootstrap class
/*@Bean methods from DAOConfig and WebConfig will be referred here along with the Spring
Boot's auto configurations*/
}
```

Note: In our course as the demos are very small demos so we have used the .properties file to configure our Spring Boot projects.

Constructor injection or Setter injection for dependencies

There are three ways to achieve dependency injection in Spring. They are:

1. Constructor injection
2. Setter injection
3. Field injection

The Constructor injection and Setter injection can be mixed by a Spring Boot Developer during the application development but it is a very good practice if he/she can use Constructor injection while injecting the mandatory dependencies and Setter injection while injecting the optional dependencies.
Many Spring developers prefer to use Constructor injection over Setter injection as Constructor injection makes the bean class object immutable.

Using @Service Annotation Class for Business Layer

In a survey, it's observed that few developers directly call the Spring repository classes in the Controller classes which should be avoided. It's recommended to use a service class annotated with @Service annotation to write the business logic.

For example:

```
@Service
public class OrderServiceImpl implements OrderService{
//Assuming a valid interface exists as - OrderService
//All the business logics for OrderService are written here
//Repository method invocations
}
```

Spring Bean Naming Conventions

It is the default feature of a Spring Boot project to auto-configure based on the starter jars as well as to create the beans for all the classes which are annotated with @Controller/@Service/@Repository/@Component by lowering the first character of the class name. So, for example, if the class name is "CustomerService" then the ID for the same bean created in the Spring IoC container will be "customerService".

But when a developer wants to configure some additional beans by creating their own configuration file then, they need to follow the standard Java naming conventions when naming the beans. The standard bean name starts with lowercase and should be a camel-case format.

For example:

```
@Configuration
public class AppConfig{
@Bean("dataSource") //Bean id is dataSource
public DataSource dataSource(){
//Additional DataSource bean configurations
return new DataSource();
}
@Bean("xmlParser")//Bean id is xmlParser
public XMLParser xmlParser(){
//Additional XMLParser bean configurations
return new XMLParser();
}
}
```

Note: Don't use single characters for bean names like @Bean("a") or @Bean("x") or @Bean("A") etc.

Spring Data JPA Best Practices

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.
Note: For the demos already covered in the course, we would be applying these best-practices.

Extended interface usage

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.

```
@Repository
public interface OrderRepository extends JpaRepository<Order, Integer> {
// HQL
@Query("SELECT "
+ "  DISTINCT o "
+ "FROM "
+ "  Order o "
+ "INNER JOIN "
+ "  o.sender.friends f "
+ "WHERE "
+ "  (p.sender = ?1 OR f = ?1) "
+ "AND p.dateCreated < ?2")
Page<Post> findAllBySenderOrRecieverAndDateCreated(Person person, Date dateCreated
, Pageable page);
}
```

Note: It's a good practice to use HQL as it provides simplicity and more maintainable code.

Don't fetch more data than you need

Pagination can be directly used within the database. You can specify and fetch the required no. of rows.

```
public List<Customer> getAllCustomers() {
EntityManager em = emf.createEntityManager();
Query query = em.createQuery("From Customer");
int pageNumber = 1;
int pageSize = 10;
query.setFirstResult((pageNumber-1) * pageSize);
query.setMaxResults(pageSize);
List <Customer> custList = query.getResultList();
return custList;
}
```

- ***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.

```
public List<String> getAllCustomersName() {
EntityManager em = emf.createEntityManager();
Query query = em.createQuery("select c.name From Customer c");
List <String> custList = query.getResultList();
return custList;
}
```

@NamedQuery vs @Query

Although creating query creation using a method name is a suitable option but in certain scenarios, it's difficult to derive a method name for the required query and the method definition looks so nasty. In this case, below approached can be followed:

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

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

Let us now understand JPA named queries:

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

```
@Entity
//Use entity class name followed by the user-defined method name separated with a dot(.)
@NamedQuery(name = "FullTimeEmployee.findByEmail", query = "select e from
FullTimeEmployee e where e.email = ?1")
public class FullTimeEmployee {
@Id
private Integer employeeNumber;
private String firstName;
private String lastName;
private String department;
private String email;
-------
}
```

Note: As mentioned in the NamedQueries topic in the course for executing this NamedQuery one needs to specify an interface with method declaration.

For a small number of queries, the NamedQueries approach is valid and works fine.

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

@Query has a high priority over @NamedQuery.

@Query could be used to write more flexible queries to fetch data.

Declaration of the query at the method level with the help of @Query :

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
//Query string is in JPQL
@Query("select emp from FullTimeEmployee emp where emp.emailAddress = ?1")
FullTimeEmployee  findByEmailAddress(String emailAddress);
}
```

@Query annotation for JPQL and native SQL queries can be used.

By default, it supports JPQL. One has to set the native query attribute to true to support native SQL.

The disadvantage of writing queries in native SQL is that, they become vendor-specific database and hence portability becomes a challenge

JPQL in Custom Repository

The code given below is used to search employees in a custom repository:

```
public List<Employee> searchEmployee(String fName, String lName, String number, String
email,String department) {
EntityManager em = emf.createEntityManager();
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Employee> query = builder.createQuery(Employee.class);
Root<Employee> root = query.from(Employee.class);
Predicate firstName = builder.equal(root.get("firstName"), fName);
Predicate lastName = builder.equal(root.get("lastName"), lName);
Predicate exp1 = builder.and(firstName, lastName);
Predicate phoneNumber = builder.equal(root.get("phoneNumber"), number);
Predicate emailId = builder.equal(root.get("email"), email);
Predicate exp2 = builder.or(phoneNumber,emailId);
Predicate dept = builder.equal(root.get("dept"), department);
query.where(builder.or(exp1, exp2,dept));
return em.createQuery(query.select(root)).getResultList();
}
}
```

The above code can be rewritten in JPQL in a simpler way, below is the revised code:

```
@Override
public List<String> searchCustomer(String name, String address, Character gender, Integer age) {
EntityManager em = emf.createEntityManager();
Query query=em.createQuery("select c from Customer c where c.name=:name1 and
c.address=:address or c.gender=:gender or c.age=:age");
query.setParameter("name1", name);
query.setParameter("address", address);
query.setParameter("gender", gender);
query.setParameter("age", age);
return query.getResultList();
}
```

Spring Transaction Best Practices

Some of the best practices that need to be followed as part of the Quality for Spring Transactions. When these practices are applied during applying Spring Transaction in Spring applications, yield more manageable code.

Best Practices:

1. Let Spring Framework do the transaction management
2. Use @Transactional annotation at Service layer
3. Know the defaults of the @Transactional annotation
4. Take care of the method call while using the @Transactional

Let us understand the recommendations and their implications in details.

Note: For the demos already covered in the course, we would be applying these best practices.

Use Spring to manage the transaction

Mainly Spring Framework supports two ways for managing the transactions:

1. **Programmatic Transaction Management:** In this approach, developers are responsible for managing the transaction with the help of a lot of coding. This approach is very flexible but very difficult to maintain.
2. **Declarative Transaction Management:** In this approach, developers separate the transaction management code from the business code where he/she needs to use either annotations / XML-based configurations for managing the transactions.

Spring recommends using Declarative Transaction Management as:

- additional responsibility to the Spring Framework to manage the Transactions

- transaction-related code is separate from the business code and can be developed using a loosely coupled way
- transactions to be applied appropriately at the runtime just like AOP

Keep the @Transactional definition at the Service layer
Spring people recommend using @Transactional at the class level of the Service layer enforcing the databases used in the project to initiate transaction management.

Let's discuss why.

If there are several repositories/DAO that a service layer method works with, and which needs to be part of a single transaction, then it's advised to apply @Transactional at the service layer and at the same time on the DAO / repository layer we can apply @Transactional(propagation = Propagation.MANDATORY) which enforces that no new transaction has to be started and has to work with an already started transaction(e.g. the one that is applied by service layer).

Know what are the defaults of the @Transactional annotation

Before using the Spring Declarative Transaction management, a developer should know the defaults for a @Transactional annotation.

If the attributes are not set, then Spring Transaction uses the default attributes for @Transactional annotation which may lead to some issues in the future.

The defaults for @Transactional annotations are as follows:

@Transactional Attribute attributes	Default values	Explanation
Propagation	PROPOGATION_REQUIRED	Use an existing transaction otherwise create a new one
Isolation level	ISOLATION_DEFAULT	Defined by the underlying DB default which normally results to Isolation.READ_COMMITED
Transaction Type	Read/Write	readOnly flag is switched off by default
Timeout	Based on the underlying database settings	Defined by the underlying DB default
Rollback condition	Rollback on runtime exception and no rollback on checked exception	By default, rollback occurs only if a RuntimeException is thrown unless this parameter is setup.

Take care of the method call while using the @Transactional
If the employee's base location and address details are to be updated, such calls should be executed in the transaction scope.
In Spring, calling the same class method (e.g. this.updateEmployeenew()) with a conflicting @Transactional requirement causes runtime exceptions. Because Spring only identifies the caller and makes no provisions to invoke the callee.

In such cases, a method shouldn't call another method inside the same class having a conflicting @Transaction configuration as below:

```
@Override
// Update on Employee and Address table are executed in transaction scope
public void updateEmployee(EmployeeDTO employeeDTO) {
updateEmployeenew(employeeDTO);  // Noncompliant
}
@Override
@Transactional
// Update on Employee and Address table are executed in transaction scope
public void updateEmployeenew(EmployeeDTO employeeDTO) {
// Method to update the baseLocation for Employee
employeeRepository.employeeUpdate(employeeDTO.getBaseLocation());
// Method to update the new address details in Address table
addressRepository.updateAddress(employeeDTO.getAddress().getAddressId(),
employeeDTO.getAddress().getCity(),
employeeDTO.getAddress().getPincode());
}
```

The above code causes an exception.

ERROR : Executing an update/delete query; nested exception is javax.persistence.TransactionRequiredException: Executing an update/delete query

Rewrite the code of updateEmployee() to avoid the exception.

```
@Override
@Transactional
// Update on Employee and Address table are executed in transaction scope
public void updateEmployee(EmployeeDTO employeeDTO) {
// Method to update the baseLocation for Employee
employeeRepository.employeeUpdate(employeeDTO.getBaseLocation());
// Method to update the new address details in Address table
addressRepository.updateAddress(employeeDTO.getAddress().getAddressId(),
employeeDTO.getAddress().getCity(),
employeeDTO.getAddress().getPincode());
}
```