

## CHAPTER 1

### INTRODUCTION SPRING DATA JPA

#### 1.1 INFYTEL CUSTOMER- INSERT AND DELETE USING JDBC API

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_nobigint primary key,  
    namevarchar(50),  
    age integer,  
    gender char(10),  
    addressvarchar(50),  
    plan_id integer  
);
```

##### 1.1.1 DEMO:INFYTEL CUSTOMER 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

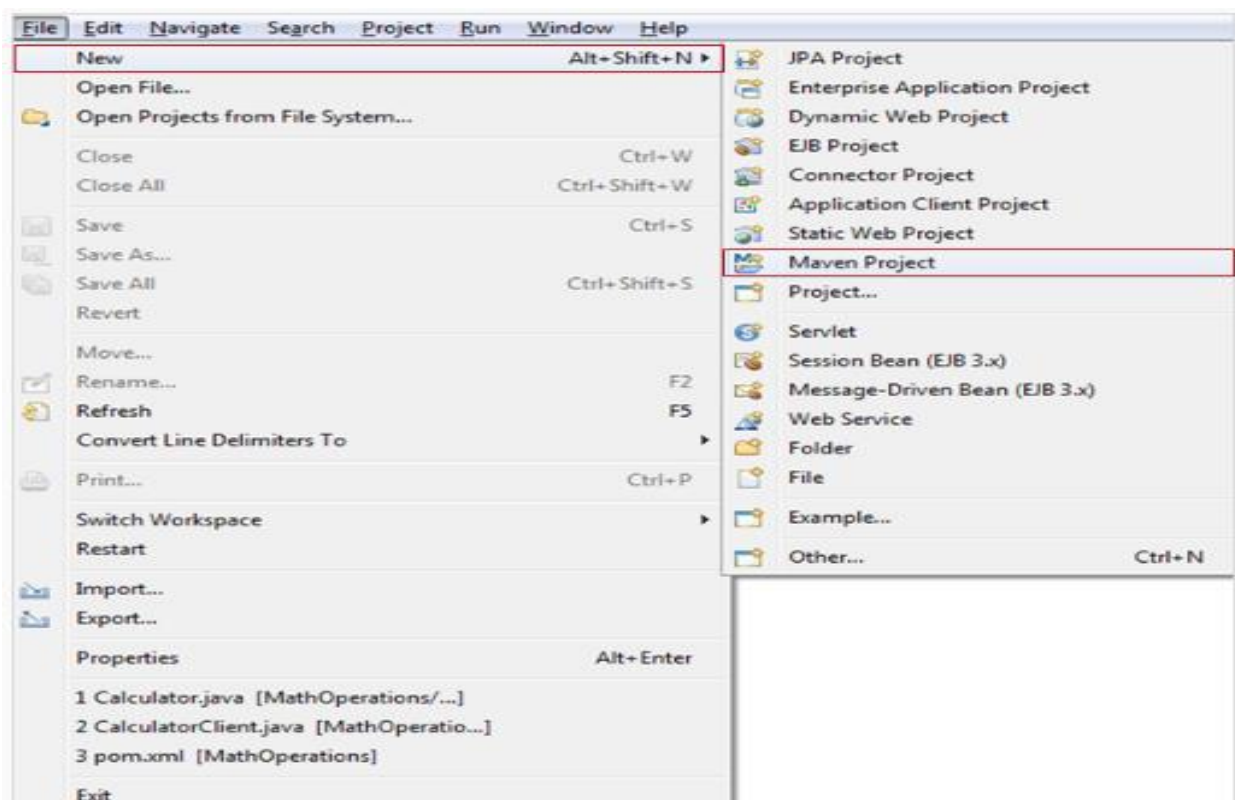
Steps to implement the application:

Now, let us understand the steps required to create a Maven project for Spring Data JPA application using Spring Tool Suite(STS)/Eclipse IDE

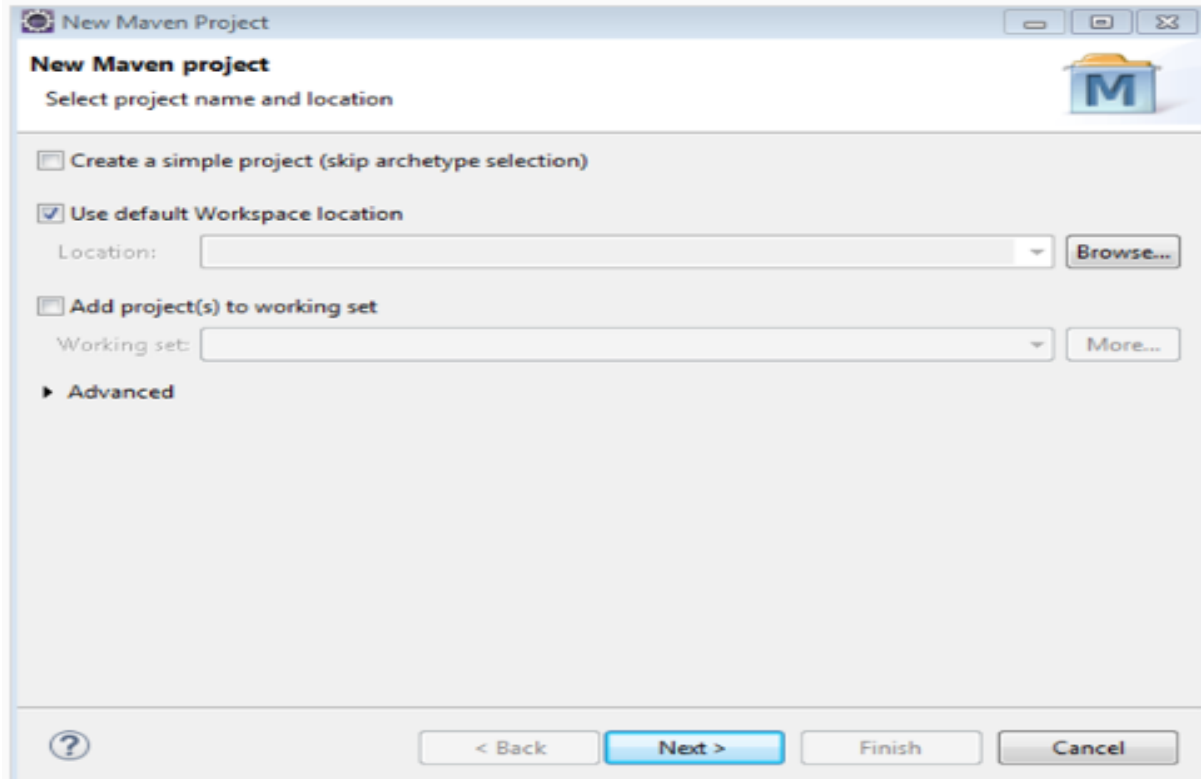
Note: Screenshots of this demo have been taken from Eclipse IDE. However, the steps remain similar even for the Spring Tool Suite IDE.

Step 1: Create a Maven project in Eclipse as shown below

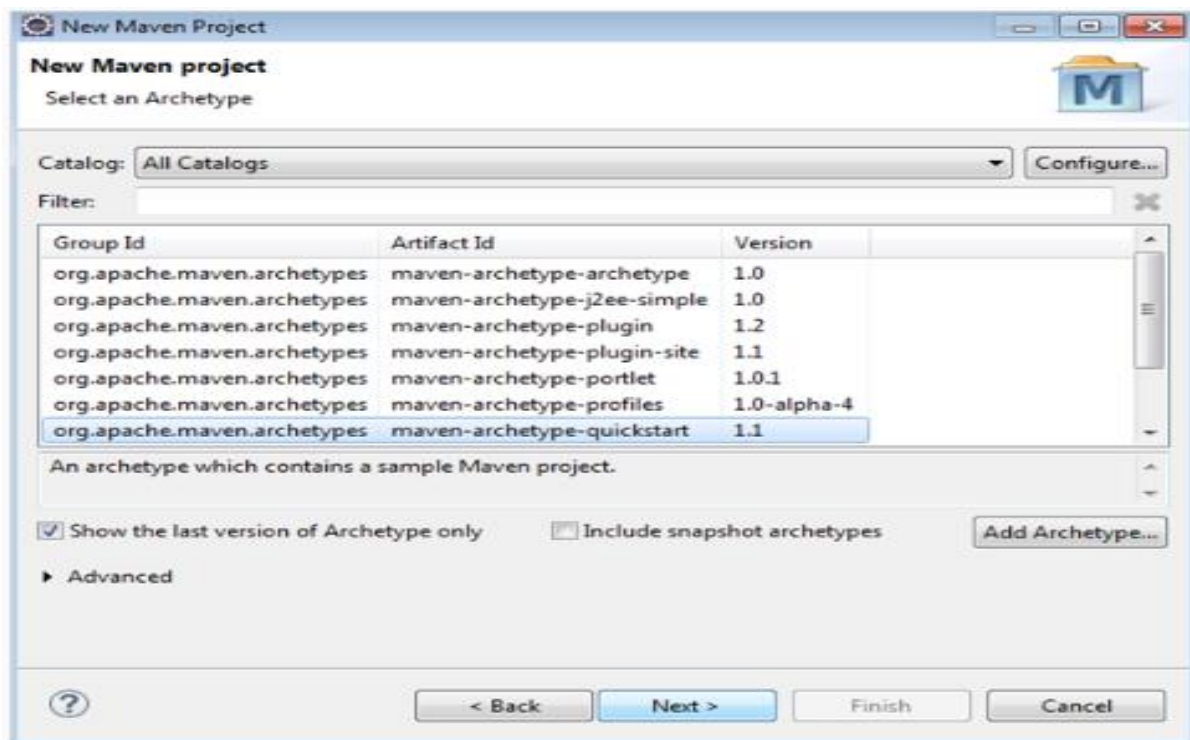
Go to File > new > Maven Project, you would get the below screen



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



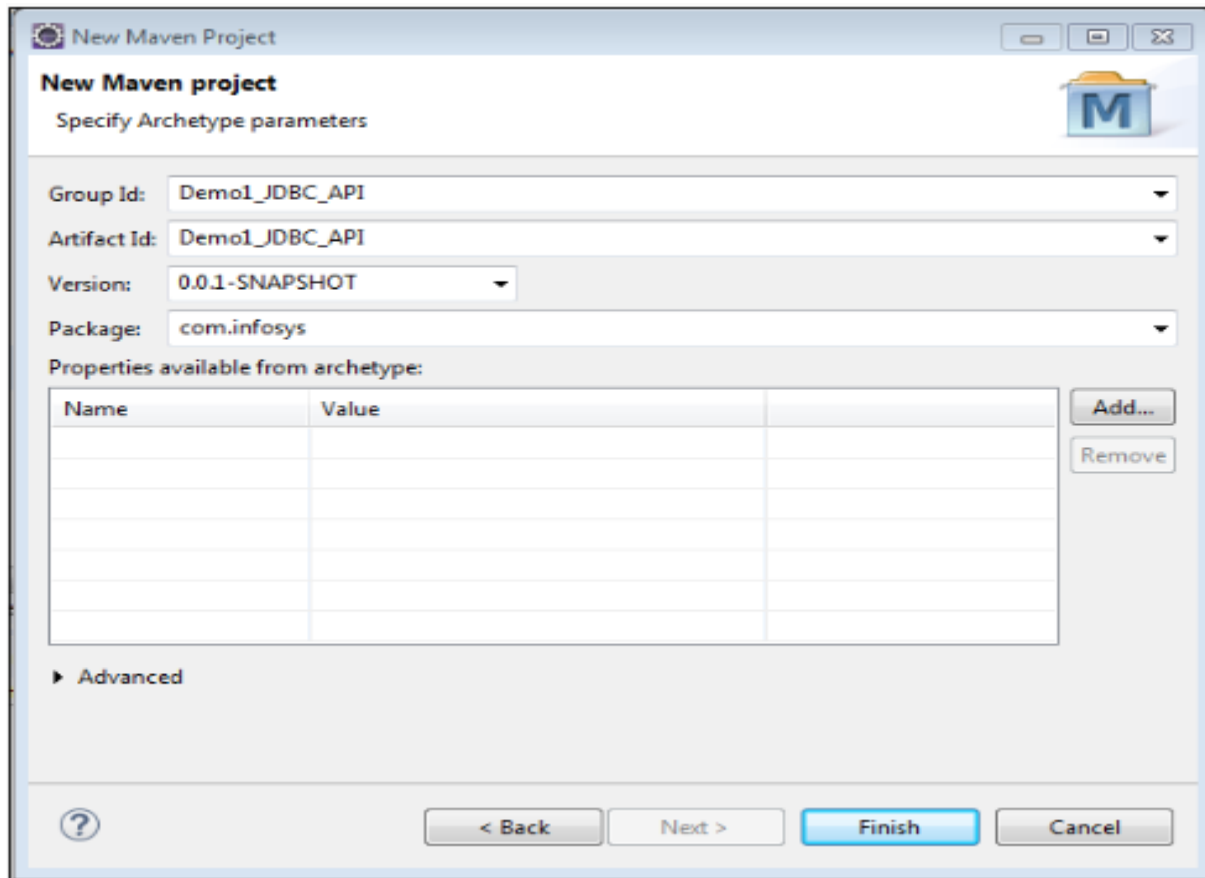
Step 3: Choose maven-archetype-quickstart



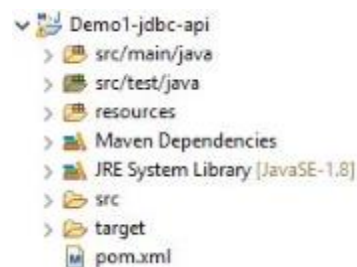
Step 4: Provide groupId as Demo1\_JDBC, artifactId as Demo1\_JDBC\_API and retain the default version which is 0.0.1-SNAPSHOT.

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.



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



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>

```

### 1.1.2 LIMITATIONS OF JDBC API

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

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.

### 1.1.3 ORM AND SPRING ORM

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.

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. 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

#### 1.1.4 DEMO:INFYTEL CUSTOMER USING SPRING ORM

Demo 2: Application Development Using Spring ORM JPA

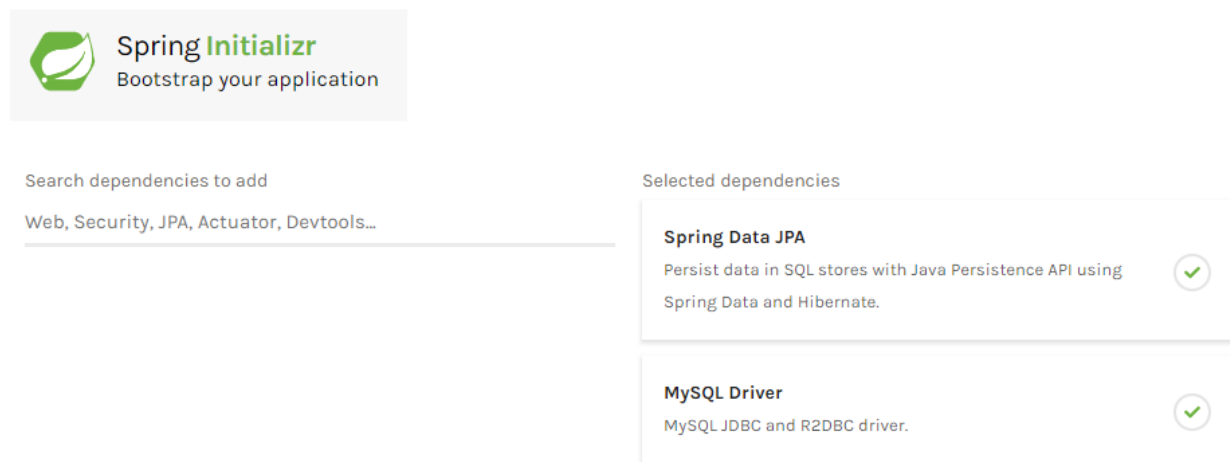
Highlights:

- 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 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>

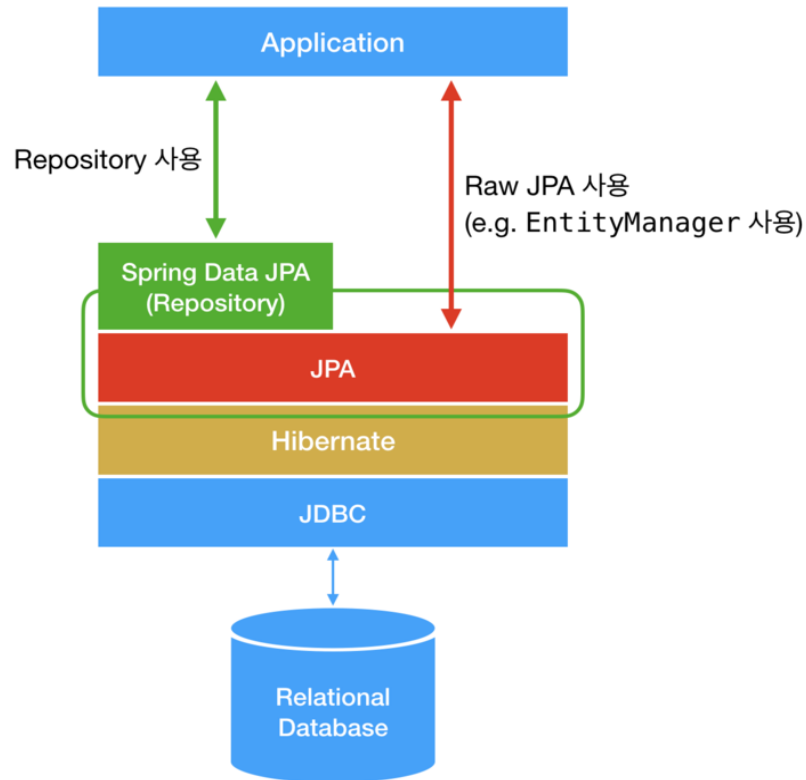
<scope>runtime</scope>

</dependency>



### 1.1.5 WHY SPRING DATA JPA?

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



**Fig: 1.1 Spring data JPA**

### 1.1.6 WHAT IS SPRING DATA JPA

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.

### 1.1.7 SPRING DATA JPA WITH SPRING BOOT

#### **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

@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.

## 1.2 SPRING DATA JPA CONFIGURATION

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> {  
  
}
```

## **1.2.1 DEMO:CRUD OPERATION USING SPRING DATA JPA**

### **CRUD Operations with Spring Data JPA**

#### **Setting Up the Project:**

Ensure you have a Spring Boot project set up with the necessary dependencies:

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

#### **Defining the Entity:**

```
Java  
@Entity  
public class Customer {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String firstName;  
    private String lastName;  
  
    private String email;
```

```
// ... getters and  
setters  
}
```

### Creating the Repository:

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

### Implementing the Service:

```
Java  
@Service  
public class CustomerService {  
    @Autowired  
    private CustomerRepository customerRepository;  
  
    public Customer saveCustomer(Customer customer) {  
        return customerRepository.save(customer);  
    }  
  
    public List<Customer> getAllCustomers() {  
        return customerRepository.findAll();  
    }  
  
    public Customer getCustomerById(Long id)  
    {  
        return customerRepository.findById(id).orElse(null);  
    }  
  
    public void deleteCustomer(Long id) {  
        customerRepository.deleteById(id);  
    }  
}
```

### Explanation:

- **JpaRepository:** Provides basic CRUD operations for JPA.
- **entities() method:** Inserts a new entity or updates an existing one based on the id field.
- **findAll() method:** Retrieves all entities.
- **findById() method:** Retrieves an entity by its id.
- **deleteById() method:** Deletes an entity by its id.

## 1.3 PAGINATION AND SORTING

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);

}
```

## 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> {

}
```

### 1.3.1 DEMO:PAGINATION AND SORTING OPERATION IN SPRING DATA JPA

Demo 4: Application Development Using Spring Data JPA

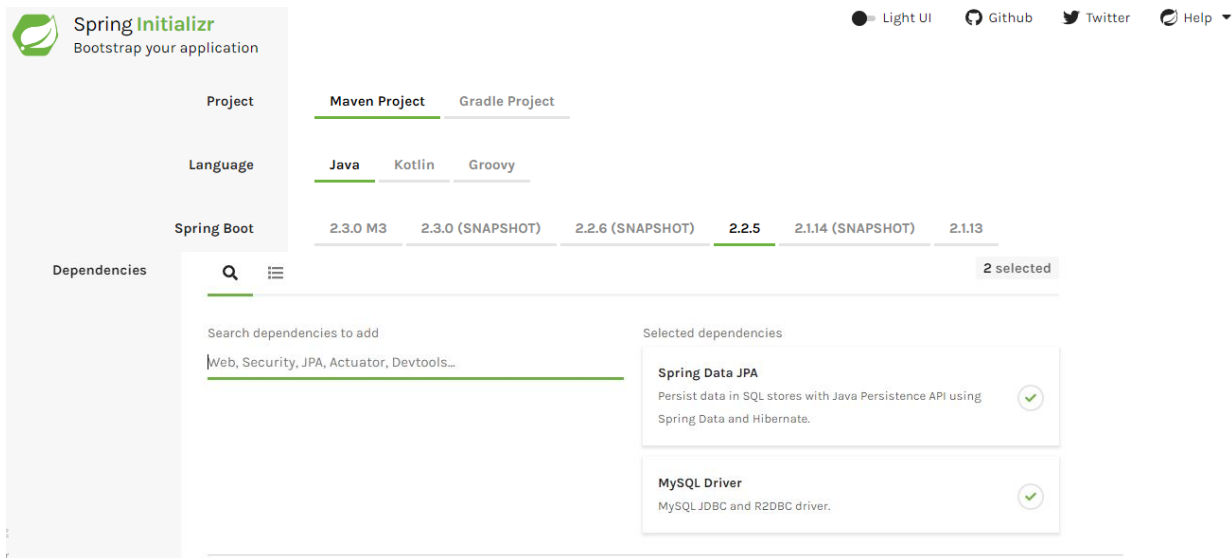
Highlights:

- 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.



## 1.4 QUERY APPROACHES

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. `@NamedQuery`
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.

### 1.4.1 DEMO: QUERY CREATION BASED ON THE METHOD NAME IN SPRING DATA JPA

Demo 5: Application Development Using Spring Data JPA

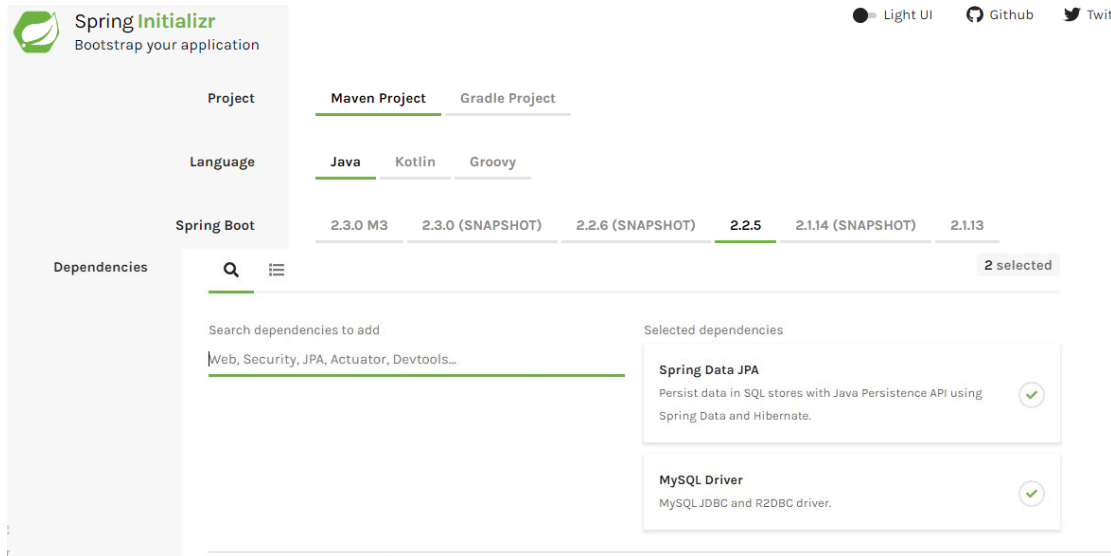
#### Highlights:

- 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 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>

### 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]

## 1.5 WHY SPRING TRANSACTION

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

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

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

### 1.5.1 SPRING DECLARATION TRANSACTION

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();
```

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 @Transactional annotation in the upcoming topics.

### 1.5.2 DEMO: SPRING TRANSACTION USING 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.

Step 1: Create an entity class "Customer.java"

Step 2: Create an entity class "Plan.java"

Step 3: Create a DTO class "CustomerDTO.java"

Step 4: Create a DTO class "PlanDTO.java"

Step 5: Create an interface "CustomerRepository.java"

Step 6: Create an interface "PlanRepository.java"

Step 7: Create an interface "CustomerService.java"

Step 8: Create a service class "CustomerServiceImpl.java"

Step 9: Update "application.properties"

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

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

### 1.5.3 UPDATE OPERATION IN SPRING DATA JPA

**@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 EntityManager to be cleared automatically we can set @Modifying annotation's clearAutomatically attribute to false.

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();

}
```

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

```
public class CustomerServiceImpl implements CustomerService {

    Customer customerTom = customerRepository.findById(1);

    customerRepository.deleteInactiveCustomers();

    System.out.println(customerRepository.findById(1).isPresent())

    System.out.println(customerRepository.count())
}
```

### **1.5.4 DEMO:UPDATE OPERATION USING SPRING DATA JPA**

Demo 8: Application Development Using Spring Data JPA

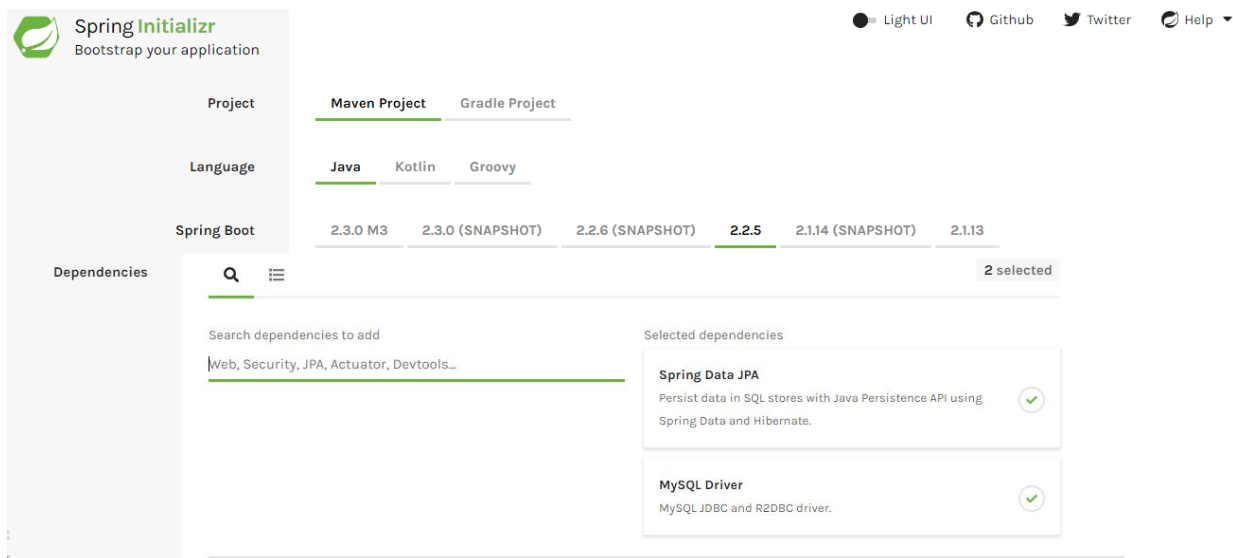
Highlights:

- 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.



## 1.5.5 MORE ABOUT SPRING DECLTIVE TRANSACTION

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

`@Transactional` annotation supports the following attributes:

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

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

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

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

Let us understand these attributes in detail.

### Transaction Isolation

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

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

	the first
--	-----------

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

Isolation Level	Description
DEFAULT	The Application uses the default isolation level of the database
READ_UNCOMMITTED	Read changes that are not committed
READ_COMMITTED	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

### 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
Phantom Reads	A Phantom read occurs when two identical queries are executed and the collected.

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_UNCOMMITTED	Read changes that are not committed
READ_COMMITTED	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

## 1.6 CUSTOMER REPOSITORY IMPLEMENTATION

### Customer Repository Implementation: A Comprehensive Guide:

A Customer Repository is a crucial component of many applications, providing a way to interact with and manage customer data. The specific implementation will vary based on the technology stack and project requirements, but the core concepts remain consistent.

### Core Concepts:

1. **Data Access Layer:** This layer is responsible for interacting with the underlying data storage mechanism (like a database).
2. **Repository Interface:** Defines the methods for CRUD (Create, Read, Update, Delete) operations on customer data.
3. **Repository Implementation:** Implements the methods defined in the repository interface, providing concrete implementations for data access.

## 1.6.1 DEMO: CUSTOMER REPOSITORY IMPLEMENTATION USING SPRING DATA JPA

### Creating a Custom Repository Implementation with Spring Data JPA:

#### Understanding the Scenario:

Let's assume we have a Customer entity with fields like id, firstName, lastName, and email. We want to create a custom repository to fetch customers based on a specific criteria, such as customers with a certain email domain.

#### 1. Define the Entity:

```
Java
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;

    private String email;
    // ... getters and
    setters
}
```

#### 2. Create the Custom Repository Interface:

```
Java

public interface CustomerRepository extends JpaRepository<Customer, Long> {
    List<Customer> findByEmailDomain(String domain);
}
```

#### 3. Implementation:

Spring Data JPA will automatically generate the implementation for the `findByEmailDomain` method based on the method name. It uses a naming convention to derive the query.

#### 4. Usage in a Service Layer:

```
Java
@Service
public class CustomerService {
    @Autowired
    private CustomerRepository customerRepository;

    public List<Customer> getCustomersByDomain(String
domain) {
return customerRepository.findByEmailDomain(domain);
    }
}
```

### 1.7 BEST PRACTICES-SPRING BOOT

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();
```

```
}
```

```
}
```

### **1.7.1 BEST PRACTICES-SPRING DATA JPA**

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

```
public List<Employee>search Employee(String fName, String lName, String number, String email,String department) {
```

```
EntityManagerem = 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));
```

```
returnem.createQuery(query.select(root)).getResultList();
```

```
}
```

```
}
```

### 1.7.2 BEST PRACTICES-SPRING TRANSACTION

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 caller.

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

```
@Override
```

```
public void updateEmployee(EmployeeDTO employeeDTO) {
```

```
    updateEmployeeNew(employeeDTO);    }
```

```
@Override
```

```
@Transactional
```

```
    public void updateEmployeeNew(EmployeeDTO employeeDTO) {
```

```
        employeeRepository.employeeUpdate(employeeDTO.getBaseLocation());
```

```
        addressRepository.updateAddress(employeeDTO.getAddress().getAddressId(),  
employeeDTO.getAddress().getCity(),
```

```
employeeDTO.getAddress().getPincode()); }
```

#### Spring Rest:

Spring REST provides a robust and efficient way to build RESTful web services. It leverages the power of Spring Framework's core features, such as dependency injection, aspect-oriented programming, and declarative transaction management, to simplify the development process.

**Client-Server Architecture:** Clear separation between client and server.

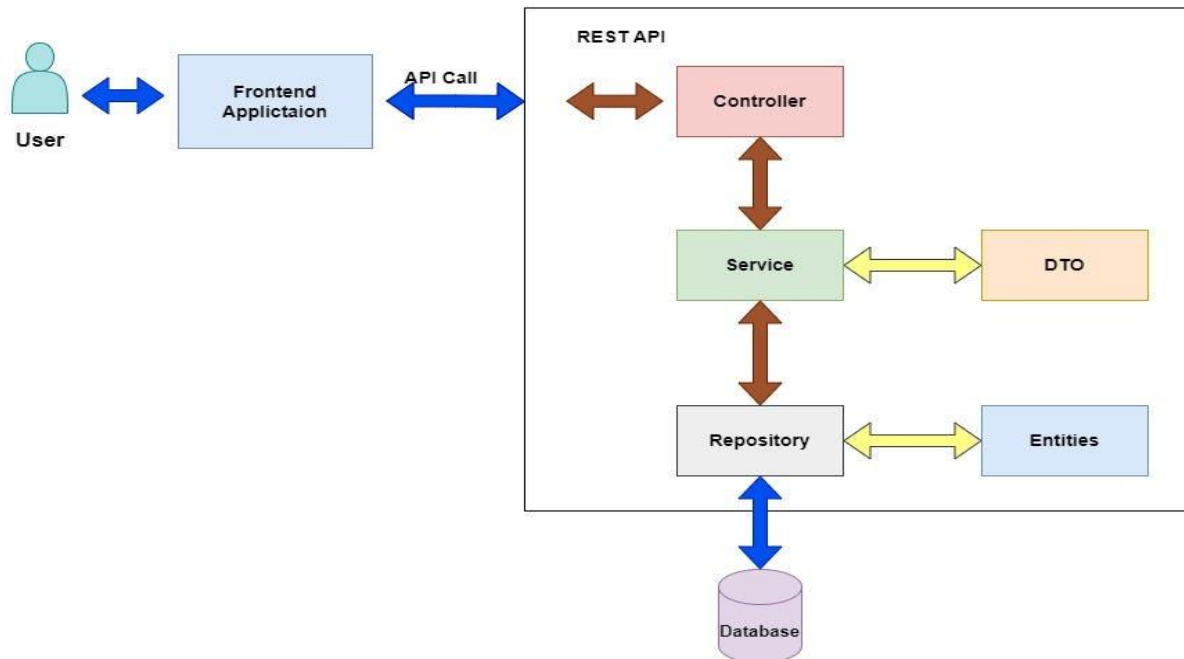
**Statelessness:** Each request from a client must contain all the information necessary to understand and process the request.

**Cacheability:** Responses can be cached to improve performance.



**Uniform Interface:** A consistent interface using standard HTTP methods (GET, POST, PUT, DELETE, etc.).

**Layered System:** The architecture is organized in layers, such as presentation, business logic, and data access.

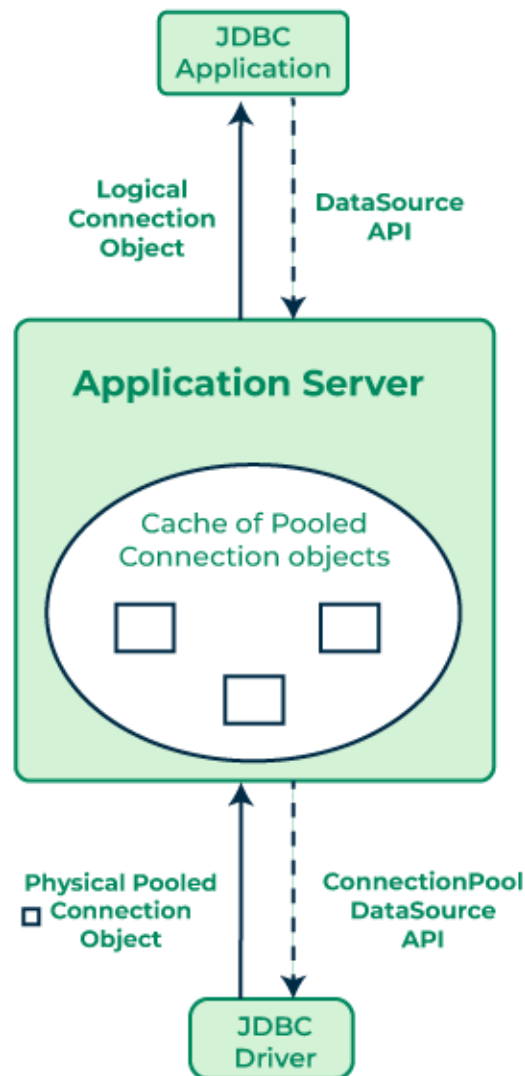


**Fig:1.2 Spring Rest**

### **Spring JDBC:**

Spring JDBC offers a more convenient and efficient way to interact with relational databases compared to traditional JDBC programming. It provides a higher-level abstraction, reducing boilerplate code and simplifying error handling. Simplifies common database operations like executing queries, updating data, and batch operations. The architecture is

organized in layers, such as presentation, business logic, and data access.



**Fig:1.3 Spring JDBC**

## **1.8SPRING DATAJPA ENTERTAINMENT DOMAIN CAPSTONES-EXPRESS MOVIES**

### **Problem Statement:**

Express Movies is an application to maintain the details of released and upcoming movies. This application is helpful for knowing details about movies and their respective directors. The application requires different CRUD operations to be performed using Spring Data JPA to meet the given functionalities.

The following functionalities have to be implemented in the application:

- Add movie
- Search movie and director details

- View movies
- Update movie and director details
- Remove movie

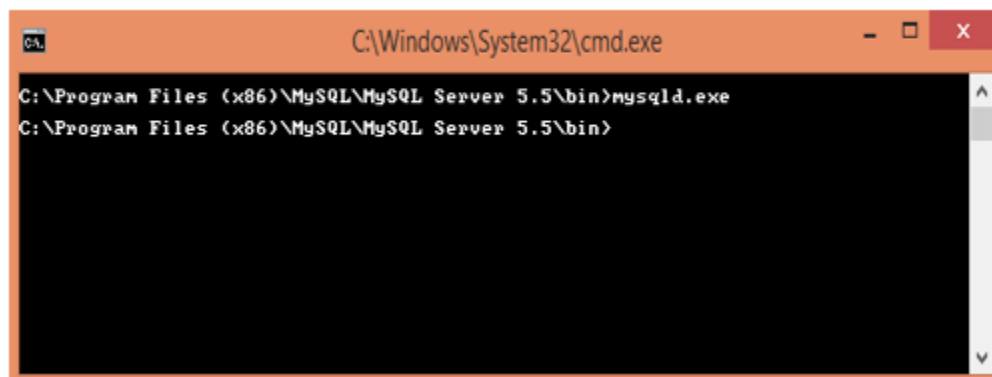
## Software Requirement

1. Spring Tool Suite(STS) / Eclipse IDE
2. MySQL

## Project Setup

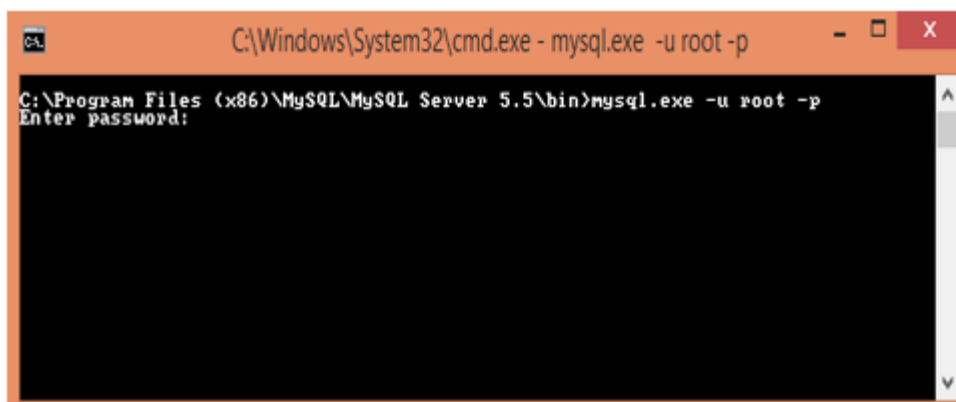
### MySQL Database setup:

- Open the command prompt and navigate to the folder which has the MySQL server bin folder.
- Enter the command `mysqld.exe` to start the MySQL Server



```
C:\Windows\System32\cmd.exe
C:\Program Files (x86)\MySQL\MySQL Server 5.5\bin>mysqld.exe
C:\Program Files (x86)\MySQL\MySQL Server 5.5\bin>
```

Once MySQL server is started, type the command `mysql.exe -u root -p` and key in the MySQL password to establish the connection with the database.

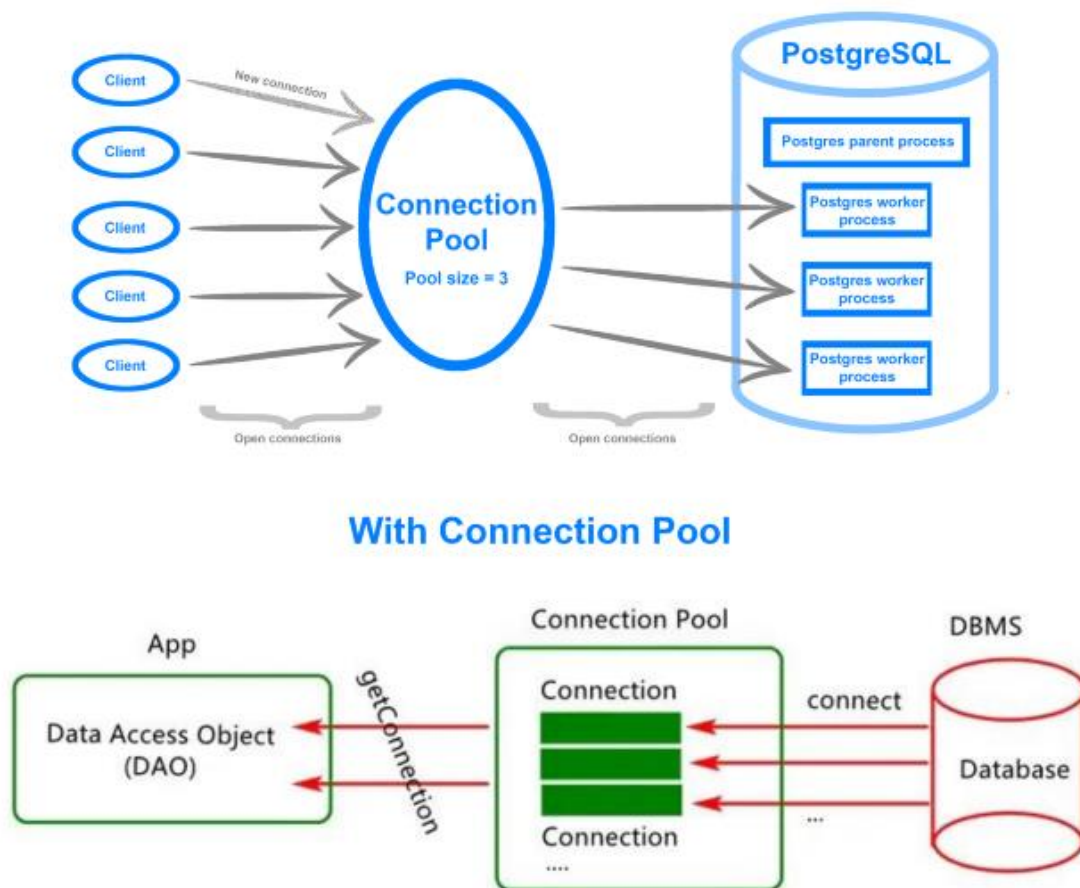


```
C:\Windows\System32\cmd.exe - mysql.exe -u root -p
C:\Program Files (x86)\MySQL\MySQL Server 5.5\bin>mysql.exe -u root -p
Enter password:
```

Your database is now ready to use. Make sure your MySQL server instance is open while running your application.

## 1.8.1 INTRODUCTION TO SPRING DATA MONGODB

If you would like to implement your application's data access layer using MongoDB repository, then your application has to define an interface that extends MongoRepository interface.



## 1.8.2 SPRING DATA MONGODB

**Step 1:** We need to add the following Spring Data dependencies:

spring-boot-starter-data-mongodb

**Step 2:** Define domain class with annotations to provide document name and primary key details

```
@Document(collection = "Customer")
```

```
public class Customer {
```

```
    @Id
```

```
    private Long phoneNumber;
```

```
private String name;

private Integer age;

private Character gender;

private String address;

}
```

**Step 3:** In the data access layer of the application, now we need only the interface extending Spring's `MongoRepository<T,K>` interface as shown below:

```
public interface CustomerRepository extends MongoRepository<Customer, Long> {

}
```

We need to provide the below details to Spring through `MongoRepository<Customer, Long>` interface:

- Domain class name to which you need the database operations (In this example, domain class is Customer).
- The Datatype of the primary key of your domain class (the primary key type of Customer class is an Long).

Spring scans the interface extending `MongoRepository` interface and auto-generates common CRUD methods, paging & sorting methods at run time through a proxy object.

Provide host and port details of MongoDB to connect in **application.properties** file.

### **1.8.3 DEMO:INSERT AND DELETE OPERATION USING SPRING DATA MONGODB**

Demo 9: InfyTel Application Development Using Spring Data MongoDB

Highlights:

- To understand data access layer implementation using Spring Data MongoDB
- To understand the required configuration to support Spring Data MongoDB

Problem Description:

For the InfyTel scenario, create a Spring application to maintain Customer details such as phone number, age, name, address, gender.

The Application should support the following operations using Spring Data MongoDB:

#### **Inserting and Deleting Documents with Spring Data MongoDB**

## 1. Setting Up the Project:

Ensure you have a Spring Boot project set up with the necessary dependencies:

XML

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

## 2. Defining the Document:

```
Java
@Document
public class Customer {
    @Id
    private String id;
    private String firstName;
    private String lastName; private String email;

    // ... getters and setters
}
```

## 3. Creating the Repository:

```
Java
public interface CustomerRepository extends MongoRepository<Customer, String> {
}
```

## 4. Implementing the Service:

```
Java
@Service
public class CustomerService {
    @Autowired
    private CustomerRepository customerRepository;

    public void insertCustomer(Customer customer) {
        customerRepository.save(customer);
    }

    public void deleteCustomer(String id) {
        customerRepository.deleteById(id);
    }
}
```

## 5. Inserting a Document:

```
Java
Customer customer = new Customer();
customer.setFirstName("John");
```

```
customer.setLastName("Doe");  
customer.setEmail("johndoe@example.com");
```

```
customerService.insertCustomer(customer);
```

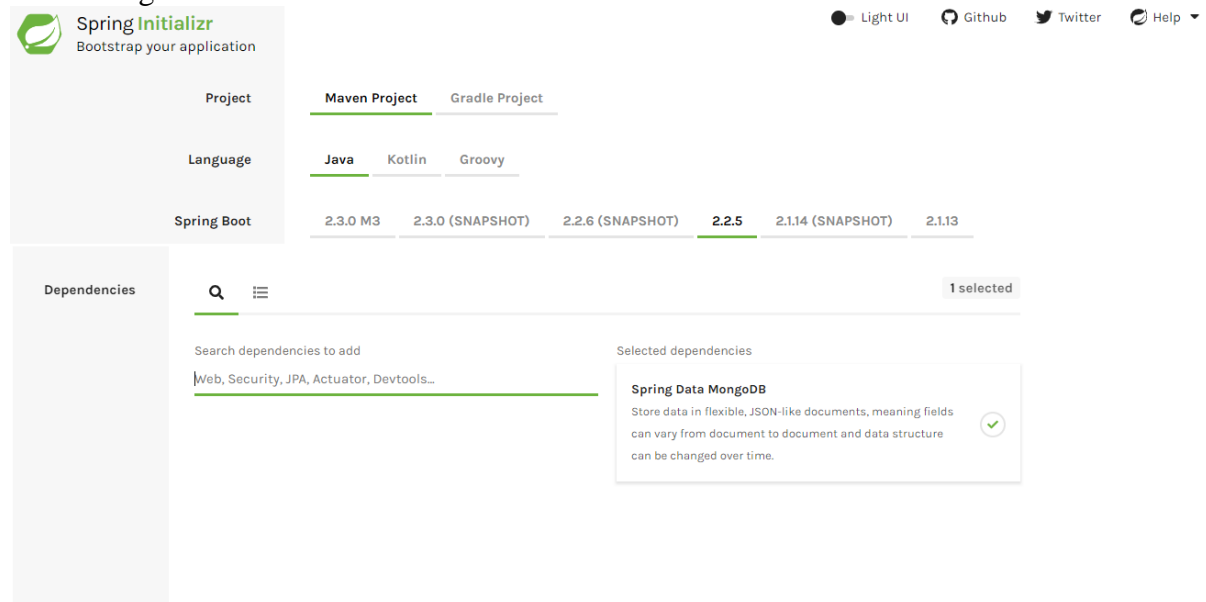
## 6. Deleting a Document:

Java

```
customerService.deleteCustomer(customer.getId());
```

## Steps to implement the application

While generating the Spring Boot Maven project from Spring initializer, Select "Spring Data MongoDB".



**Step 1:** Create a class “**Customer.java**” and “**CustomerDTO.java**”

**Step 2:** Create a repository interface “**CustomerRepository.java**”

**Step 3:** Create a Service layer with “**CustomerService.java**” interface and service class “**CustomerServiceImpl.java**”

**Step 4:** Update **application.properties** file in resources

Run Client.java as Spring Boot App

## OUTPUT:

Added Customers successfully

Enter the Customer Phone Number to be deleted  
785671000

Customer removed successfully.

## CHAPTER 2

### INTRODUCTION TO SPRING FRAMEWORK

#### 2.1 WHY SPRING?

Spring is a popular open-source Java application development framework created by Rod Johnson. Spring is a Java-based framework for constructing web and enterprise applications. Spring supports developing any kind of Java application such as standalone applications, web applications, Data base driven applications.

Spring is widely considered to be a Secure ,low cost and flexible framework that improves coding efficiency and reduce overall application development time through efficient use of system resources .Spring removes tedious configuration work so that developers can focus on writing business logic.

Spring promotes loose coupling between components, making applications more modular, testable, and maintainable. AOP allows developers to modularize cross-cutting concerns like logging, security, and transaction management, resulting in cleaner and more organized code. Spring simplifies database interactions with support for various technologies like JDBC, Hibernate, and JPA.

It provides a consistent abstraction layer for data access. Spring has a large and active community, providing extensive documentation, tutorials, and forums. Spring applications are designed to be scalable and performance. Spring's modular design and DI make it easy to write unit and integration tests.

#### 2.1.1 WHY IS SPRING FRAMEWORK?

Spring is widely considered to be a secure, low-cost and flexible framework that improves coding efficiency and reduces overall application development time through efficient use of system resources. Spring removes tedious configuration work so that developers can focus on writing business logic.

Some Key Reasons why Spring Framework is used:

- ☐ **Dependency Injection (DI):**

Spring promotes loose coupling between components by managing dependencies through DI. This makes code more modular, testable, and maintainable.

- ☐ **Aspect-Oriented Programming (AOP):**



AOP allows developers to add cross-cutting concerns (like logging, security, and transaction management) to different parts of the application without modifying the core business logic. This leads to cleaner and more organized code.

□ **Data Access:**

Spring simplifies database interactions with support for various technologies like JDBC, Hibernate, and JPA. It provides a consistent abstraction layer for data access, making it easier to switch between different data sources.

□ **Transaction Management:**

Spring offers declarative transaction management, allowing developers to define transaction boundaries using annotations or XML configuration.

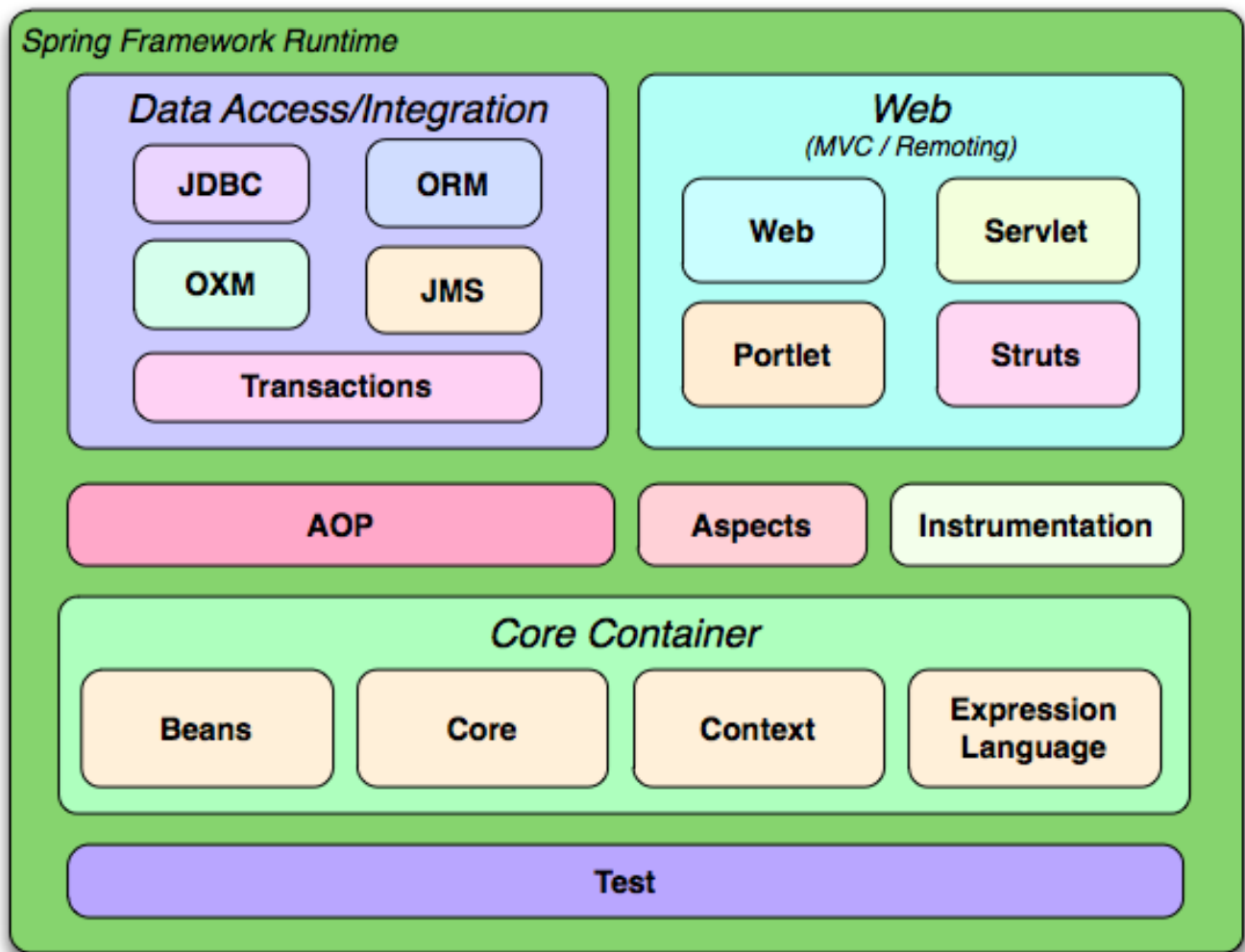
This simplifies the management of database transactions and ensures data consistency. Spring MVC provides a flexible and powerful framework for building web applications. It supports various view technologies (JSP, Thymeleaf, Freemarker) and simplifies the development of RESTful web services.

## 2.1.2 SPRING FRAMEWORK – MODULES

Spring Framework Web module provides basic support for web application development. The Web module has a web application context that is built on the application context of the core container. Web module provides complete Model-View-Controller (MVC) implementation to develop a presentation tier of the application and also supports a simpler way to implement RESTful web services.

Spring framework is divided into modules which makes it really easy to pick and choose in parts to use in any application:

- **Core:** Provides core features like DI (Dependency Injection), Internationalisation, Validation, and AOP (Aspect Oriented Programming)
- **Data Access:** Supports data access through JTA (Java Transaction API), JPA (Java Persistence API), and JDBC (Java Database Connectivity)
- **Web:** Supports both Servlet API and of recently Reactive API and additionally supports WebSockets, STOMP, and WebClient
- **Integration:** Supports integration to Enterprise Java through JMS (Java Message Service), JMX (Java Management Extension), and RMI (Remote Method Invocation)
- **Testing:** Wide support for unit and integration testing through Mock Objects, Test Fixtures, Context Management, and Caching.



**Fig : 2.1 Core container of spring framework modules**

### **2.1.3 SPRING IOC**

it is the developer's responsibility to create the dependent application object using the new operator in an application. Hence any change in the application dependency requires code change and this results in more complexity as the application grows bigger. Inversion of Control (IoC) helps in creating a more loosely coupled application. IoC represents the inversion of the responsibility of the application object's creation, initialization, and destruction from the application to the third party such as the framework. Now the third party takes care of application object management and dependencies thereby making an application easy to maintain, test, and reuse. We need not create objects in dependency injection instead describe how objects should be created through configuration. DI is a software design pattern that provides better software design to facilitate loose coupling, reuse, and ease of testing.

#### **Benefits of Dependency Injection (DI):**

- Helps to create loosely coupled application architecture facilitating re-usability and easy testing.
- Separation of responsibility by keeping code and configuration separately. Hence dependencies can be easily modified using configuration without changing the code.
- Allows to replace actual objects with mock objects for testing, this improves testability by writing simple JUnit tests that use mock objects.

#### 2.1.4 CONFIGURING IOC CONTAINER USING JAVA BASED CONFIGURATION

The Java-based configuration metadata is provided in the Java class using the following annotations:

##### **@Configuration:**

The Java configuration class is marked with this annotation. This annotation identifies this as a configuration class, and it's expected to contain details on beans that are to be created in the Spring application context.

##### **@Bean:**

This annotation is used to declare a bean. The methods of configuration class that creates an instance of the desired bean are annotated with this annotation. These methods are called by the Spring containers during bootstrap and the values returned by these methods are treated as Spring beans. By default, only one bean instance is created for a bean definition by the Spring Container, and that instance is used by the container for the whole application lifetime.

**For example:** The Spring Configuration class can be configured in a Java class using the above annotations as follows:

@ configuration

Public class spring configuration{

@Bean

Public customerserviceImpl customerservice(){

Return new customerserviceImpl()

}

}

#### 2.1.5 DEMO: SPRING IOC

- To understand Java Code Based configuration in Spring

- To understand the application container instantiation for Java configuration

### **Customer service.java**

```
Package com.infy.service;

Public interface customerservice{

    Public string createcustomer();

}
```

### **CustomerServiceImpl.java**

```
Package com.infy.service;

Public class Customerserviceimpl implements Customerservice {

    Public string createcustomer(){

        Return "customer is successfully created";

    }

}
```

### **SpringConfiguration.java**

```
Package com.infy.util;

Import org.springframework.context.annotation.bean;

Import org.springframework.context.annotation.configuration;

Import com.infy.service.customerserviceImpl;

@Configuration

Public class springconfiguration{

    @bean(name ="customerService")

    Public customerserviceImpl customerserviceImpl(){

        Return new customerserviceImpl();

    }

}
```

```
}
```

### **Client.java**

```
Package com.infy
```

```
Import
```

```
Org.springframework.context.Annotation.AnnotationconfigApplicationContext;
```

```
Import org.springframework.context.support.abstractionapplicationcontext;
```

```
Import com.infy.service.customerserviceImpl;
```

```
Import com.infy.util.springconfiguration;
```

```
Public class client{
```

```
    Public static void main(string[] args){
```

```
        customerserviceImpl service=null;
```

```
        Abstractapplicationcontext context=new
```

```
Annotationconfigapplicationcontext(springconfiguration.class);
```

```
        Service=(customerserviceImpl)context.getBean("customerservice");
```

```
        System.out.println(service.createcustomer());
```

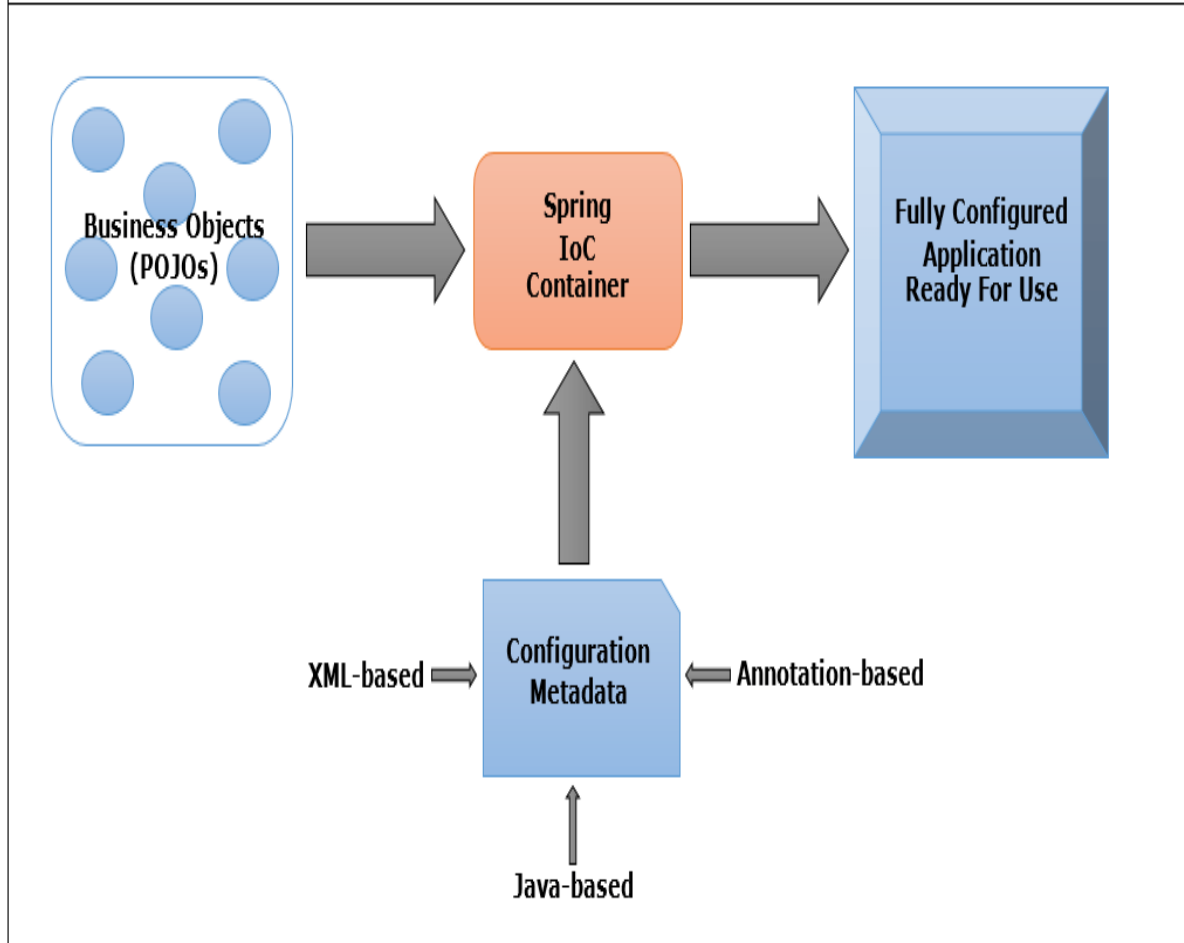
```
        Context.close();
```

```
    }
```

```
}
```

### **OUTPUT:**

```
Customer is successfully created
```



**Fig: 2.2 Spring IOC container**

## 2.2 INTRODUCTION TO DEPENDENCY INJECTION

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. It's a very useful technique for testing, since it allows dependencies to be mocked or stubbed out. DI enforces this principle by injecting abstractions (interfaces) instead of concrete implementations. In essence, dependency injection promotes a design philosophy where objects rely on well-defined interfaces rather than specific implementations. This leads to a more modular, flexible, and maintainable codebase.

There are three types of dependency injection:

### **Constructor injection:**

Constructor injection is a dependency injection technique where dependencies are injected into an object through its constructor. This approach promotes the creation of immutable objects and ensures that all required dependencies are provided at object creation time. Constructor injection is a powerful technique for managing dependencies in object-oriented programming.

### **Method injection:**

Method injection is a less common dependency injection technique compared to constructor injection and property injection. It involves injecting dependencies into a method's parameters at runtime. Dependencies are defined in a configuration file or using annotations. The method to be injected is annotated with a specific annotation. When the method is called, the dependency injection framework injects the required dependencies as arguments to the method.

### **Property injection:**

Property injection is a dependency injection technique where dependencies are injected into an object's properties. This is typically done using setters or fields. Property injection is a versatile technique that can be used to inject dependencies into objects in various ways.

## **2.2.1 CONSTRUCTOR INJECTION:**

Customer Service class has a count property, let us now modify this class to initialize count property during bean instantiation using the constructor injection approach.

How do inject object dependencies through configuration using constructor injection?

```
package com.infy.util;
```

```
@Configuration
```

```
public class SpringConfiguration {
```

```
    @Bean// customerRepository bean definition
```

```
    public CustomerRepository customerRepository() {
```

```
        return new CustomerRepository();
```

```
    }
```

```
    @Bean
```

```

        public CustomerServiceImpl customerService() {

            return new CustomerServiceImpl(customerRepository(),20);

        }
    }

```

### **2.2.2 SETTER INJECTION**

How do we inject object dependencies using setter injection?

Consider the Customer ServiceImpl class of InfyTel Customer application.

```
package com.infy.util;
```

```
@Configuration
```

```
public class SpringConfiguration {
```

```
    @Bean// customerRepository bean definition
```

```
    public CustomerRepository customerRepository() {
```

```
        return new CustomerRepository();
```

```
    }
```

```
    @Bean // CustomerServic bean definition with bean dependencies through constructor
    injection
```

```
    public CustomerServiceImpl customerService() {
```

```
        return new CustomerServiceImpl(customerRepository(),20);
```

```
    }
```

```
}
```

### **2.3 WHAT IS AUTOSCANNING**

Auto scanning refers to a feature in various devices and software that automatically detects and captures images or documents without manual intervention. It allows the device to automatically detect the type of document placed on the scanner bed (e.g., photo, text document) and adjust the scanning settings accordingly.



**@Component:** It is a general purpose annotation to mark a class as a Spring-managed bean.

**@Component**

```
public class Customer Logging{  
  
}
```

**@Service :**It is used to define a service layer Spring bean. It is a specialization of the @Component annotation for the service layer.

**@Service**

```
public class CustomerSeviceImpl implements CustomerService {  
  
}
```

### **2.3.1 DEMO:AUTOSCANNING**

#### **SpringConfiguration .java**

```
package com.infy.util;  
  
import org.springframework.context.annotation.ComponentScan;  
  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
  
@ComponentScan(basePackages="com.infy")  
  
public class SpringConfiguration {  
  
}
```

#### **CustomerService.java**

```
package com.infy.service;  
  
public interface CustomerService {  
  
    public String fetchCustomer(int count);  
  
    public String createCustomer();  
  
}
```

### **CustomerServiceImpl.java**

```
package com.infy.service;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service("customerService")

public class CustomerServiceImpl implements CustomerService {

    @Value("10")

    private int count;

    public String fetchCustomer(int count) {

        return " The no of customers fetched are : " + count;

    }

    public String createCustomer() {

        return "Customer is successfully created";

    }

}
```

### **Output:**

Customer is successfully created

## **2.4 SPRING BOOT**

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

### **2.4.1 CREATING A SPRING BOOT APPLICATION**

**InfyTelmessage.properties**

In Spring @PropertySource annotation is used to read from properties file using Spring's Environment interface. The location of the properties file is mentioned in the Spring configuration file using @PropertySource annotation.

So InfyTelmessage.properties which are present in classpath can be loaded using @PropertySource as follows:

```
import org.springframework.context.annotation.PropertySource;

@SpringBootApplication

@PropertySource("classpath:InfyTelmessage.properties")

public class DemoSpringBootApplication {

    public static void main(String[] args) throws Exception {

    }

}
```

#### **2.4.2 UNDERSTANDING @SPRINGBOOT APPLICATION ANNOTATION**

SpringApplication scans the configuration class package and all its sub-packages. So if our SpringBootApplication class is in "com.eta" package, then it won't scan com.infy.service or com.infy.repository package. We can fix this situation using the SpringApplication scanBasePackages property.

```
package com.eta;

@SpringBootApplication(scanBasePackages={"com.infy.service","com.infy.repository"})

public class DemoSpringBootApplication {

    public static void main(String[] args) {

        SpringApplication.run(DemoSpringBootApplication.class, args);

    }

}
```

#### **2.4.3 DEMO ON SPRING BOOT**

### **Demo5Application .java**

```
package com.infy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.support.AbstractApplicationContext;
import com.infy.service.CustomerServiceImpl;

@SpringBootApplication

public class Demo5Application{

    public static void main(String[] args) {

        CustomerServiceImpl service = null;

        AbstractApplicationContext context = (AbstractApplicationContext)
SpringApplication.run(Demo5Application.class, args);

        service = (CustomerServiceImpl) context.getBean("customerService");

        System.out.println(service.fetchCustomer());

        context.close();

    }

}
```

### **CustomerService.java**

```
package com.infy.service;

public interface CustomerService {

    public String fetchCustomer();

    public String createCustomer();

}
```

### **CustomerServiceImpl.java**

```

package com.infy.service;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service("customerService")

public class CustomerServiceImpl implements CustomerService {

    @Value("10")

    private int count;

    public String fetchCustomer() {

        return " The no of customers fetched are : " + count;

    }

    public String createCustomer() {

        return "Customer is successfully created";

    }

}

```

### **OUTPUT:**

```

:: Spring Boot :: (v2.1.13.RELEASE)

2020-04-07 17:21:31.908 INFO 119940 --- [ main]
com.infy.Demo5Application : Starting Demo5Application on
2020-04-07 17:21:31.912 INFO 119940 --- [      main]
com.infy.Demo5Application :no active profile set, falling back to default profiles:
2020-04-07 17:21:34.211 INFO 119940 --- [      main]
com.infy.Demo5Application:StartedDemo5Application in 3.998 seconds

The no of customers fetched are :10

```

## 2.5 WHAT IS AUTOWIRING?

In Spring @Value annotation is used to insert values into variables and method arguments. Using @Value we can either read spring environment variables or system variables.

**We can assign a default value to a class property with @Value annotation:**

```
public class CustomerDTO {  
  
    @Value("1234567891")  
  
    long phoneNo;  
  
    @Value("Jack")  
  
    String name;  
  
    @Value("Jack@xyz.com")  
  
    String email;  
  
    @Value("ANZ")  
  
    String address;  
  
}
```

### 2.5.1 DEMO ON @ AUTOWIRING SPRING BOOT

#### **Demo6Application .java**

```
package com.infy;  
  
import java.util.List;  
  
import org.springframework.boot.SpringApplication;  
  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
import org.springframework.context.support.AbstractApplicationContext;  
  
import com.infy.dto.CustomerDTO;  
  
import com.infy.service.CustomerServiceImpl;  
  
@SpringBootApplication  
  
public class Demo6Application {
```

```

    public static void main(String[] args) {

        CustomerServiceImpl service = null;

        AbstractApplicationContext context = (AbstractApplicationContext)
SpringApplication.run(Demo6Application.class,

            args);

        service = (CustomerServiceImpl) context.getBean("customerService");

        List<CustomerDTO> listcust = service.fetchCustomer();

        System.out.println("PhoneNumer" + " " + "Name" + " " + "Email" + " " +
"Address");

        for (CustomerDTO customerDTO2 : listcust) {

            System.out.format("%5d%10s%20s%10s",
customerDTO2.getPhoneNo(),customerDTO2.getName(),customerDTO2.getEmail(),
customerDTO2.getAddress());

            System.out.println();

        }

    }
}

```

### **CustomerService.java**

```

package com.infy.service;

import java.util.List;

import com.infy.dto.CustomerDTO;

public interface CustomerService {

    public String createCustomer(CustomerDTO customerDTO);

    public List<CustomerDTO> fetchCustomer();

}

```

### **CustomerServiceImpl.java**

```

package com.infy.service;

import java.util.List;

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

import org.springframework.stereotype.Service;

import com.infy.dto.CustomerDTO;

import com.infy.repository.CustomerRepository;

@Service("customerService")

public class CustomerServiceImpl implements CustomerService{

    @Autowired

    private CustomerRepository customerRepository;

    public String createCustomer(CustomerDTO customerDTO) {

        customerRepository.createCustomer(customerDTO);

        return "Customer with " + customerDTO.getPhoneNo() + " added successfully";

    }

    public List<CustomerDTO> fetchCustomer() {

        return customerRepository.fetchCustomer();

    }

}

```

### **CustomerDTO.java**

```

package com.infy.dto;

public class CustomerDTO {

    long phoneNo;

```



```
String name;

String email;

String address;

public long getPhoneNo() {

    return phoneNo;

}

public void setPhoneNo(long phoneNo) {

    this.phoneNo = phoneNo;

}

public String getName() {

    return name;

}

public void setName(String name) {

    this.name = name;

}

public String getEmail() {

    return email;

}

public void setEmail(String email) {

    this.email = email;

}

public String getAddress() {

    return address;

}
```

```

    public void setAddress(String address) {

        this.address = address;

    }

    public CustomerDTO(long phoneNo, String name, String email, String address) {

        this.phoneNo = phoneNo;

        this.name = name;

        this.email = email;

        this.address = address;

    }

    public CustomerDTO() {

    }

}

```

## OUTPUT:

:: Spring Boot ::(v2.1.13.RELEASE)

2020-04-07 12:11:56.259 INFO 78268 --- [main]

com.infy.Demo6Application :

2020-04-07 12:11:56.263 INFO 78268 --- [ main]

com.infy.Demo6Application :

2020-04-07 12:11:57.137 INFO 78268 --- [ main]

com.infy.Demo6Application :

PhoneNumer Name Email Address

9951212222 Jack Jack@infy.com Chennai

## 2.6 LOGGER

## Log into file

By default Spring Boot logs the message on the console. To log into a file, you have to include either logging.file or logging.path property in your application.properties file.

## Custom log pattern

Include logging.pattern.\* property in application.properties file to write the log message in your own format

Logging property	Sample value	Description
logging.pattern.console	%d{yyyy-MM-dd HH:mm:ss,SSS}	Specifies the log pattern to use on the console
logging.pattern.file	%5p [%t] %c [%M] - %m%n	Specifies the log pattern to use in a file

### 2.6.1 DEMO :LOGGER

#### CustomerService.java

```
package com.infy.service;

import com.infy.dto.CustomerDTO;

public interface CustomerService {

    public String createCustomer(CustomerDTO dto);

    public String fetchCustomer();

    public void deleteCustomer(long phoneNumber) throws Exception;

}
```

#### CustomerServiceImpl.java

```
package com.infy.service;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.stereotype.Service;
```

```

import com.infy.dto.CustomerDTO;

import com.infy.repository.CustomerRepository;

@Service("customerService")

public class CustomerServiceImpl implements CustomerService {

    private static Logger logger = LoggerFactory.getLogger(CustomerServiceImpl.class);

    @Autowired

    private CustomerRepository customerRepository;


    @Override

    public String createCustomer(CustomerDTO dto) {

        return customerRepository.createCustomer(dto);

    }

    @Override

    public String fetchCustomer() {

        return customerRepository.fetchCustomer();

    }

    @Override

    public void deleteCustomer(long phoneNumber) {

        try {

            customerRepository.deleteCustomer(phoneNumber);

        } catch (Exception e) {

            logger.info("In log Exception ");

            logger.error(e.getMessage(),e);

        }

    }

}

```

```

    }

}

Demo8Application.java

package com.infy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.support.AbstractApplicationContext;
import com.infy.service.CustomerServiceImpl;

@SpringBootApplication

public class Demo8Application {

    public static void main(String[] args) {

        CustomerServiceImpl service = null;

        AbstractApplicationContext context = (AbstractApplicationContext)
SpringApplication.run(Demo8Application.class,

                        args);

        service = (CustomerServiceImpl) context.getBean("customerService");

        service.deleteCustomer(1151212222l);

        // service.deleteCustomer(9951212222l);

    }

}

```

## **OUTPUT:**

:: Spring Boot :: (v2.1.13.RELEASE)

2020-04-07 14:50:12.615 INFO 99756 --- [main] com.infy.Demo8Application: Starting Demo8Application

2020-04-07 14:50:12.621INFO 99756 --- [main] com.infy.Demo8Application : No active profile set,

2020-04-07 14:50:14.183 INFO 99756 --- [main] com.infy.Demo8Application:Started Demo8Application in

2020-04-07 14:50:14.187 INFO 99756 --- [ main] com.infy.service.CustomerServiceImpl : In log Exception

2020-04-07 14:50:14.192 ERROR 99756 --- [main] com.infy.service.CustomerServiceImpl: Customer does not exist

java.lang.Exception: Customer does not exist

atcom.infy.repository.CustomerRepository.deleteCustomer(CustomerRepository.java:49)  
~[classes/:na]

atcom.infy.service.CustomerServiceImpl.deleteCustomer(CustomerServiceImpl.java:38)  
~[classes/:na]

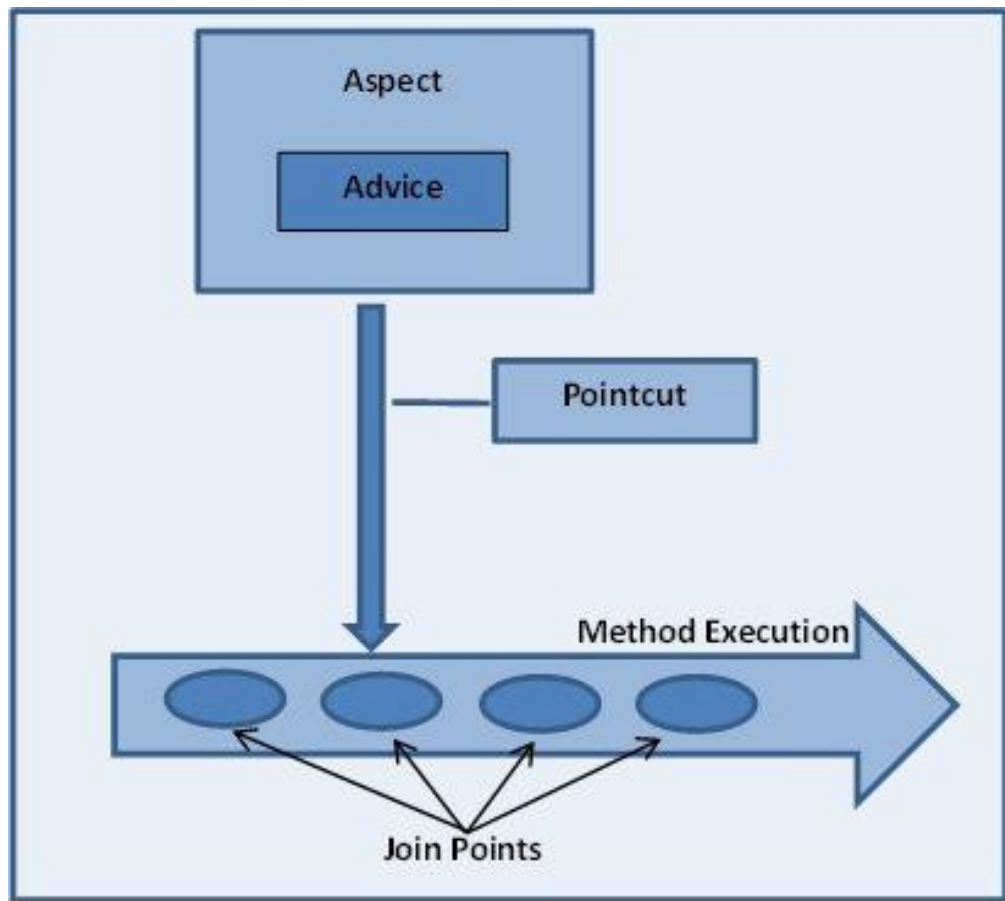
atcom.infy.Demo8Application.main(Demo8Application.java:19) [classes/:na]

## 2.7 INTRODUCTION TO SPRING AOP

AOP (Aspect Oriented Programming) is used for applying common behaviors like transactions, security, logging, etc. to the application.

- Aspect is a class that implements cross-cutting concerns. To declare a class as an Aspect it should be annotated with @Aspect annotation. It should be applied to the class which is annotated with @Component annotation or with derivatives of it.
- Join point is the specific point in the application such as method execution, exception handling, changing object variable values, etc. In Spring AOP a join point is always the execution of a method.
- Advice is a method of the aspect class that provides the implementation for the cross-cutting concern. It gets executed at the selected join point(s).

Type Of Execution	Execution Point
Before	Before advice is executed before the Join point execution.
After	After advice will be executed after the execution of Join point whether it returns with or without exception. Similar to finally block in exception handling.
After Returning	After Returning advice is executed after a Join point executes and returns successfully without exceptions



**Fig:2.3 Spring AOP**

### 2.7.1 IMPLEMENTING AOP ADVICES

This advice gets executed around the joinpoint i.e. before and after the execution of the target method. It is declared using `@Around` annotation.

```

@Around("execution(*
com.infy.service.CustomerServiceImpl.fetchCustomer(..)")

public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {

    System.out.println("Before proceeding part of the Around advice.");

    Object cust = joinPoint.proceed();

    System.out.println("After proceeding part of the Around advice.");

    return cust;
}

```

## 2.7.2DEMO: AOP

### CustomerService.java

```
package com.infy.service;

import java.util.List;

import com.infy.dto.CustomerDTO;

public interface CustomerService {

    public String createCustomer(CustomerDTO customerDTO);

    public List<CustomerDTO> fetchCustomer();

    public String updateCustomer(long phoneNumber, CustomerDTO customerDTO);

    public String deleteCustomer(long phoneNumber);

}
```

### CustomerServiceImpl.java

```
package com.infy.service;

import java.util.List;

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

import org.springframework.stereotype.Service;

import com.infy.dto.CustomerDTO;

import com.infy.repository.CustomerRepository;

@Service("customerService")

public class CustomerServiceImpl implements CustomerService {

    @Autowired

    private CustomerRepository customerRepository;

    public String createCustomer(CustomerDTO customerDTO) {

        customerRepository.createCustomer(customerDTO);

    }

}
```



```

        return "Customer with " + customerDTO.getPhoneNo() + " added successfully";
    }

    public List<CustomerDTO> fetchCustomer() {
        return customerRepository.fetchCustomer();
    }

    public String updateCustomer(long phoneNumber, CustomerDTO customerDTO) {
        return customerRepository.updateCustomer(phoneNumber, customerDTO);
    }

    public String deleteCustomer(long phoneNumber) {
        return customerRepository.deleteCustomer(phoneNumber);
    }
}

```

## 2.8 SPRING PROFILES

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.

### **@Profile- Class level:**

```
@Profile("dev")
```

```
@Component
```

```
@Aspect
```

```
public class LoggingAspect { }
```

### **@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;

}

}

```

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

```

@Profile("!test")

@Configuration

@ComponentScan(basePackages="com.infy.service")

public class SpringConfiguration { }

```

## **2.8.1 SPRING5 BASICS TRANSPORT DOMAIN CAPSTONE:INFYGO**

### **Problem Statement:**

**Background:** This problem statement provides the high level design of the project that has to be implemented as part of the hands-on assessment in order to complete the course Spring Basics.

InfyGo is an airline booking application that provides services to its customers to search for flight details. InfyGo wants a lightweight, loosely coupled application to be implemented using Spring.

Let us start with basic implementation using Spring core concepts for the following functionalities

- Add Flight
- Search Flight

As part of the Spring Basics course, let us develop the business tier of this application.

**Domain class Flight with below details:**

```
public class Flight
```

```
{
```

```
    private String flightId;
```

```
    private String airlines;
```

```
    private String source;
```

```
    private String destination;
```

```
    private Double fare;
```

```
    private LocalDate journeyDate;
```

```
    private Integer seatCount;
```

```
}
```

Implement the required repository using the collection

- FlightRepository interface
- FlightRepositoryImpl class

Implement the Service layer of the application

- FlightService interface
- FlightServiceImpl class

Implement client code to access bean of FlightServiceImpl class and perform below tasks in a menu based approach.

Add Flight: Add flight details by auto generating FlightId starting from 1001 and accepting other flight details from the user.

Search Flight: User can Search flights based on source, destination, and journey date.

- Before displaying search details to the user, if the provided journey date is during the festival season(Dec to Jan) then increase the flight fare by 20% for all search results.

## **CHAPTER 3**

### **WEB SERVICES – AN INTRODUCTION**

#### **3.1 WHY WEB SERVICES?**

Web services are reusable software components which are available on the network for consumption .By other applications, irrespective of their platform and technology. Web Services can either be SOAP or REST based. And, REST has become more popular in recent days. REST stands for Representational State Transfer.

Following are the advantages of making the functionalities of an application as Restful services.

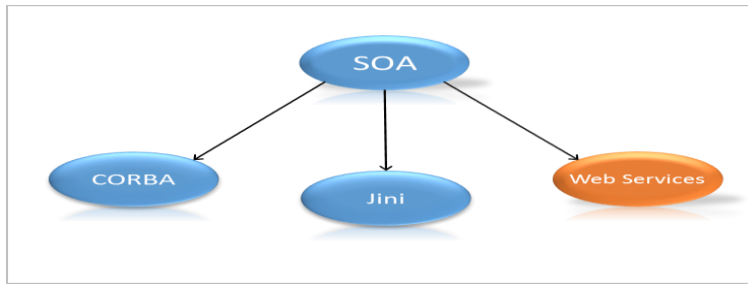
- Interoperability
- Resource sharing
- Reusability
- Independent client-server architecture
- Stateless

##### **3.1.1 SOA – SERVICE ORIENTED ARCHITECTURE**

SOA expands to Service Oriented Architecture, is a software model designed for achieving communication among distributed application components, irrespective of the differences in terms of technology, platform, etc.The application component that requests for a service is the consumer and the one that renders the service is the producer.

Following is the pictorial representation that speaks about the different ways of realizing and implementing SOA

.



**Fig: 3.1 SOA**

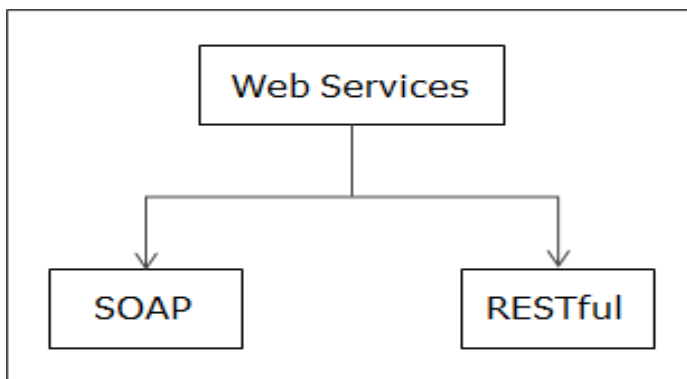
### **3.1.2WHAT ARE WEB SERVICES**

"A web service is a software system designed to support interoperable machine-to-machine interaction over a network". Web services are client-server applications that communicate (mostly) over Hyper Text Transfer Protocol (HTTP) Web services are a standard means of achieving interoperability between software applications running on diverse platforms and frameworks.

Following are some of the common benefits of Web Services.

- Integrates with other systems easily irrespective of the difference in technology or platform - Interoperability
- Creates reusable components - Reusability
- Saves cost, effort and time - Ease of use

### **3.2 TYPES OF WEB SERVICES**



**Fig: 3.2 web services**

#### **3.2.1SOAP –BASEDWEB SERVICES**

**SOAP (Simple Object Access Protocol)** defines a very strongly typed messaging framework that relies heavily on XML and schemas. And, SOAP itself is a protocol (transported

over HTTP, can also be carried over other transport layer protocol like SMTP/FTP, etc.) for developing SOAP-based APIs.

### 3.2.2 A SAMPLE SOAP BASED WEB SERVICES

Here, we will

Understand the basic concepts of SOAP

Analyze the SOAP Requests and Responses

Let us look at a sample SOAP-based Web Service which when invoked by sending the name of the user will respond back with a string **"Hello <name>, Welcome to the world of Web Services"**.

This sample code is written in Java using **JAX-WS** API (API that helps to develop SOAP-based Web Services in Java)

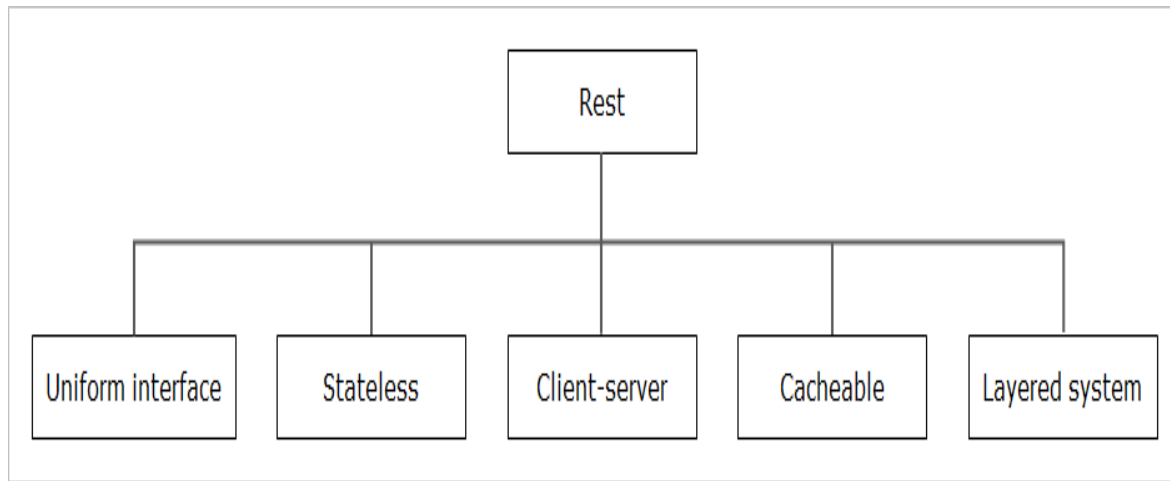
```
1. packagecom.infy;
2. importjavax.jws.WebService;
3. importjavax.jws.WebMethod;
4. importjavax.jws.WebParam;
5. //Below annotation is used to denote that this class is going to be exposed as a
6. //Soap based Web Service and also names your Web Service while generation the WSDL
7.
8. @WebService(serviceName = "GreetingService")
9. publicclassGreetingService {
10.
11. /**
12.  * This is a sample web service operation called "hello"
13.  */
14. @WebMethod(operationName = "hello")
15. publicStringmyhello(@WebParam(name = "name") Stringuname) {
16. return"Hello " + uname + ",Welcome to the world of Web Services !";
17.     }
18. }
19.
```

### Cacheable

Restful Web Services use the HTTP protocol as a carrier for the data. And, they can make use of the metadata that is available in the HTTP headers to enable caching. For example, Cache-Control headers are used to optimize caching to enhance performance

### Rest principles

- Uniform interface
- Stateless
- Client server



**Fig: 3.3 Rest**

### 3.3 HOW TO CREATE RESTFUL SERVICES

**Restful**Web services can be developed and consumed in Java using **JAX-RS API** or **Spring REST**JAX-RS expands to Java API for Restful Web services.

Restful services, or Representational State Transfer services, have become a cornerstone of modern web application development. They offer a simple, stateless approach to building scalable and maintainable APIs .JAX-RS is a set of specifications to extend support for building and consuming Restful Web services in Java

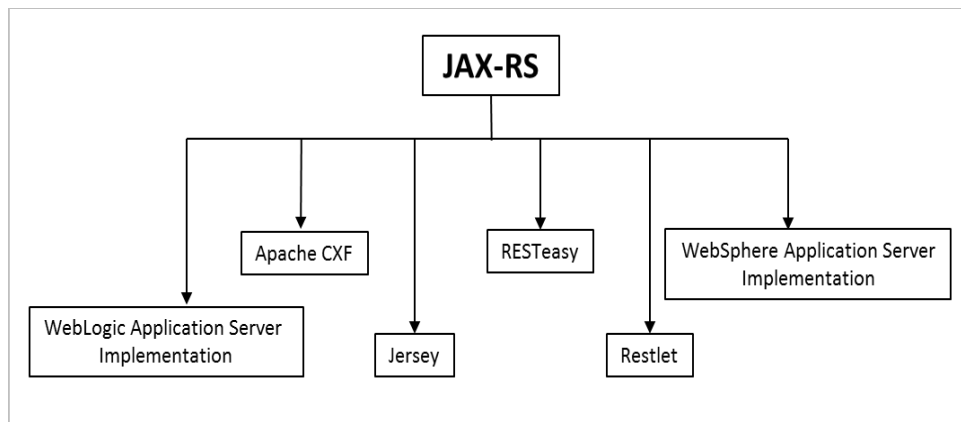
JAX-RS is simply a specification and we need actual implementations to write web services. There are many vendors in the market who adhere to the standards of JAX-RS such as, Jersey and Rest Easy.

Restful services, or Representational State Transfer services, have become a cornerstone of modern web application development. They offer a simple, stateless approach to building scalable and maintainable APIs.

#### CORE COMPONENTS OF RESTFUL SERVICES

- Resource
- Representation
- HTTP Method

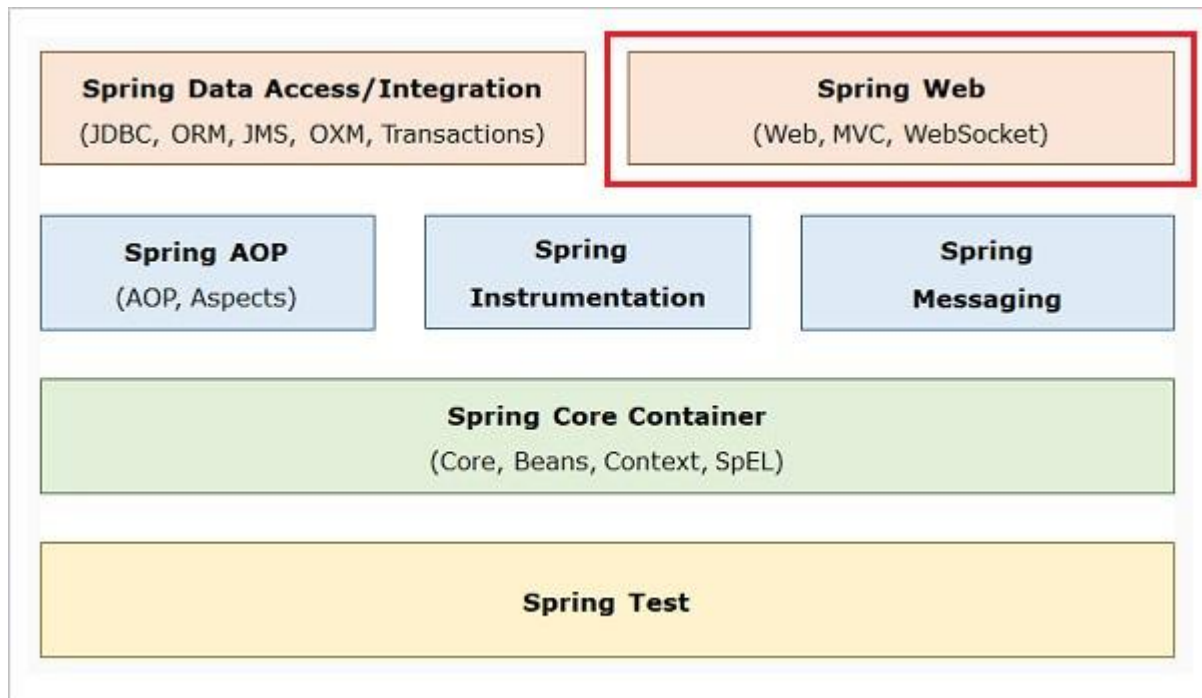
Below are some of the JAX-RS implementations.



**Fig: 3.4 Jax-Rs**

### 3.3.1 SPRING REST – AN INTRODUCTION

Spring's web module carries the support for REST as well.



**Fig:3.5 spring rest**

### Spring MVC

Spring Web MVC is the source of Spring REST as well. Spring MVC framework is based on MVC and the front controller design patterns.

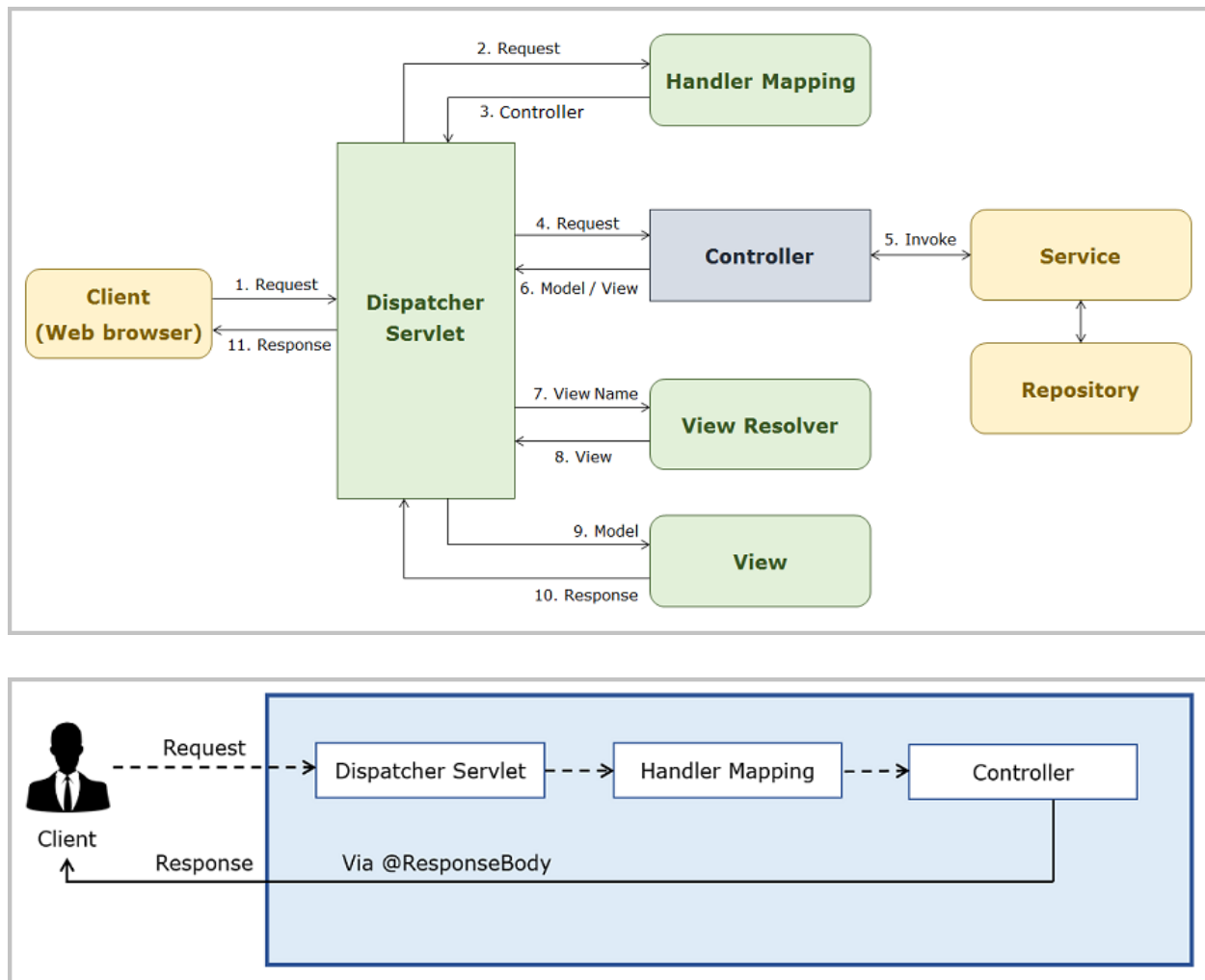
Spring Web MVC is the source of Spring REST as well. Spring MVC framework is based on MVC and the front controller design patterns. The core element of Spring MVC is the Dispatcher



servlet and, that is the Front Controller which provides an entry point for handling all the requests and forwards the same to the appropriate components.

## Spring MVC Architecture

The following figure shows the working internals of Spring MVC.



**Fig:3.6 spring MVC architecture**

- In Spring 4.0, the **@RestController** annotation was introduced.
- This annotation is a combination of **@Controller** and **@ResponseBody**.
- This annotation when used on a REST controller class bounds all the values returned by controller methods to the response body.
- Developing Spring REST as a Boot project still simplifies the developers' job as a lot of things are auto-configured.
- In our course, we will learn , how to develop a **Spring REST** application using **Spring Boot**.

## DIFFERENCE BETWEEN SPRING MVC AND SPRING REST

Spring MVC	Spring REST
In the purpose of spring MVC is Building web applications	In the purpose of spring MVC Building Restful web services
Its Focus on Full-stack web applications	Its Focus on API development
The core concepts are MVC pattern, views, templates	The core concepts are HTTP methods, resource-based URLs, statelessness
Annotations : @Controller, @Request Mapping, @ModelAttribute	Annotations : @Rest Controller, @Request Mapping, @Path Variable, @Request Body

### 3.4 CASE STUDY – INFYTEL

Spring Boot helps to create a REST application with less effort and time. This is because Spring Boot manages dependencies in an efficient way. Also, it abstracts the configurations that help the developers focus only on the business.

Additionally, Spring Boot provides embedded defaults (example - deploys a web application in the embedded Tomcat server) and opinions (example - suggests Hibernate implementation of JPA for ORM).

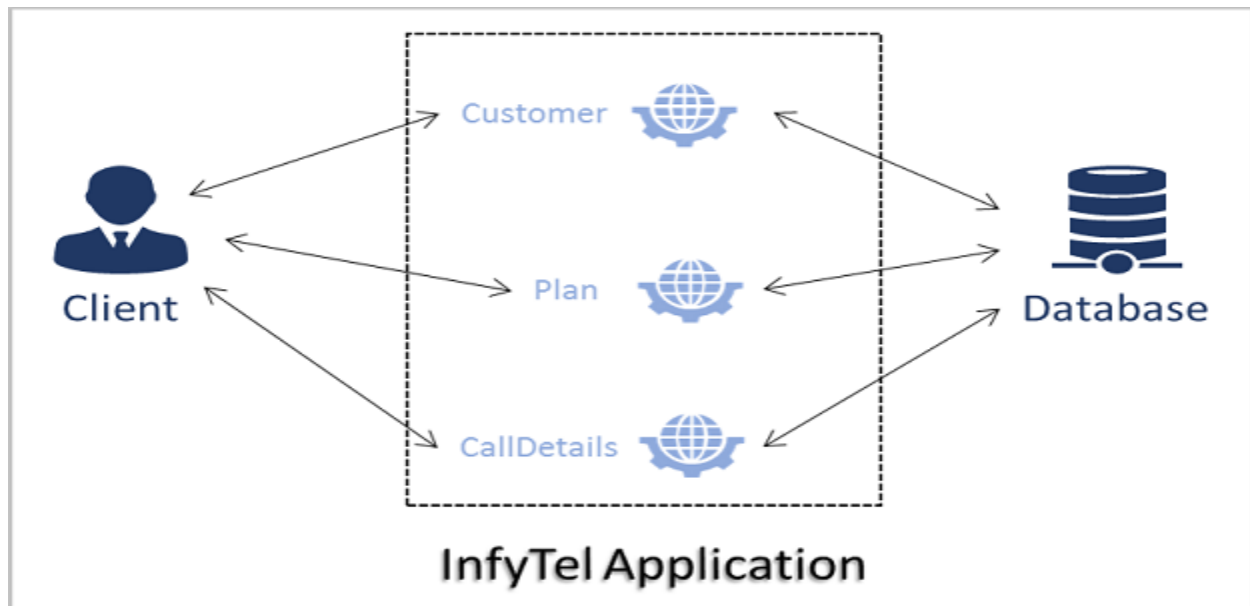
In this course, we will look at a telecom application called Infytel. As the course proceeds, we will build this app incrementally to make it a collection of REST endpoints. And, we will use Spring Boot to develop this application.

The Infytel application will have the following REST resources.

The Infytel application will have the following REST resources.

Controller	Functionality
Customer Controller	Add, update, delete and fetch customers
Plan Controller	Fetch plan details
Call Details Controller	Fetch call details

And, these resources contact the same database.



**Fig: 3.7 Infytel**

**About the course** section has the complete set of downloadable code and script for the Infytel case study.

Once downloaded, proceed with the following steps for execution.

**Step 1:** Extract the project and import the same as Maven project in STS.

**Step 2:** Make sure the database is up and ready with tables.

**Step 3:** Execute the project as Spring Boot application.

**Step 4:** Pay attention to the console to gain information on port number and the context path.

**Step 5:** Gain request mapping details of the Restful service methods. That is, have a look at the request and method mapping annotations of the controllers to understand the

- URI format
- Template parameters, if any
- Requestmethod (GET, POST, etc.,)
- MIME type of Java objects to be passed

Finally, trigger requests to the endpoints using a tool like Post Man.

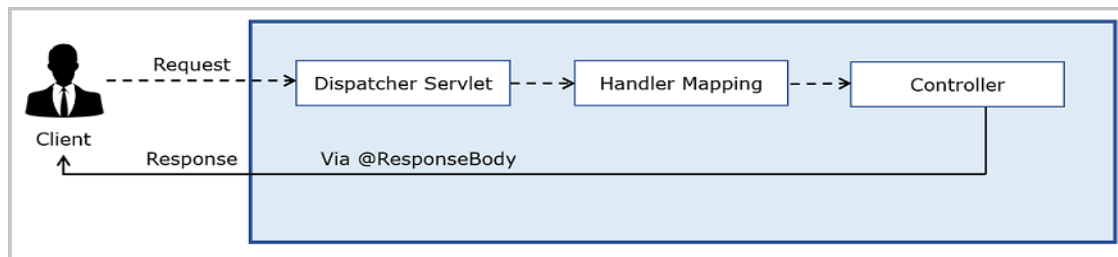
### 3.4.1 CREATING A SPRING REST CONTROLLER

Spring Web MVC module is the source of Spring REST as well.

#### **Working Internals of Spring REST:**

Spring REST requests are delegated to the DispatcherServlet that identifies the specific controller with the help of handler mapper. Then, the identified controller processes the request and renders the response. This response, in turn, reaches the dispatcher servlet and finally gets rendered to the client.

Here, ViewResolver has no role to play.



**What are the steps involved in exposing business functionality as a Restful web service?**

**Step 1:** Create a REST Resource

**Step 2:** Add the service methods that are mapped against the standard HTTP methods

**Step 3:** Configure and deploy the REST application

Since we are going to develop REST applications using Spring Boot, lot of configurations needed in Step-3 can be avoided as Spring Boot takes care of the same.

Let us look at each step in detail :

Any class that needs to be exposed as a Restful resource has to be annotated with `@RestController`

### **@REST CONTROLLER**

- This annotation is used to create REST controllers.
- It is applied on a class in order to mark it as a request handler/REST resource.
- This annotation is a combination of `@Controller` and `@ResponseBody` annotations.
- `@ResponseBody` is responsible for the automatic conversion of the response to a JSON string literal. If `@RestController` is in place, there is no need to use `@ResponseBody` annotation to denote that the Service method simply returns data, not a view.
- `@RestController` is an annotation that takes care of instantiating the bean and marking the same as REST controller.

It belongs to the package, `org.springframework.web.bind.annotation.RestController`

### **@Request Mapping**

This annotation is used for mapping web requests onto methods that are available in the resource classes. It is capable of getting applied at both class and method levels. At method level, we use this annotation mostly to specify the HTTP method.

### **PROGRAM**

```
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
@RestController
```

```

@RequestMapping("/customers")
Public class CustomerController
{
@RequestMapping(method=RequestMethod.POST)
    Public string createCustomer()
    {
        //Functionality goes here
    }
}

```

REST resources have handler methods with appropriate HTTP method mappings to handle the incoming HTTP requests. And, this method mapping usually happens with annotations.

URI	HTTP Method	CustomerController method	Method description	Annotation to be applied at the method level	New Annotation that can be applied instead of @RequestMapping at the method level
/customers	GET	Fetch Customer()	Will fetch all the customers of Infytel App and return the same.	@RequestMapping(method = RequestMethod.GET)	@GetMapping
/customers	POST	Create Customer()	Will create a new customer	@RequestMapping(method = RequestMethod.POST)	@PostMapping
/customers	DELETE	Delete Customer()	Will delete an existing customer	@RequestMapping(method = RequestMethod.DELETE)	@DeleteMapping
/customers	UPDATE	Update Customer()	Will update the details of an existing customer	@RequestMapping(method = RequestMethod.PUT)	@PutMapping

Since we are creating a Spring Boot application with spring-boot-starter-web dependency, we need not pay much focus on the configurations.

- spring-boot-starter-web dependency in the pom.xml will provide the dependencies that are required to build a Spring MVC application including the support for REST. Also, it will ensure that our application is deployed on an embedded Tomcat server and we can replace this option with the one that we prefer based on the requirement.

Application class is simply annotated with @SpringBootApplication

- **@Spring Boot Application** is a convenient annotation that adds the following:
- **@Configuration** marks the class as a source where bean definitions can be found for the application context.

- **@Enable Auto Configuration** tells Spring Boot to start adding beans based on the classpath and property settings.
- **@Component Scan** tells Spring to look for other components, configurations, and, services in the packages being specified.

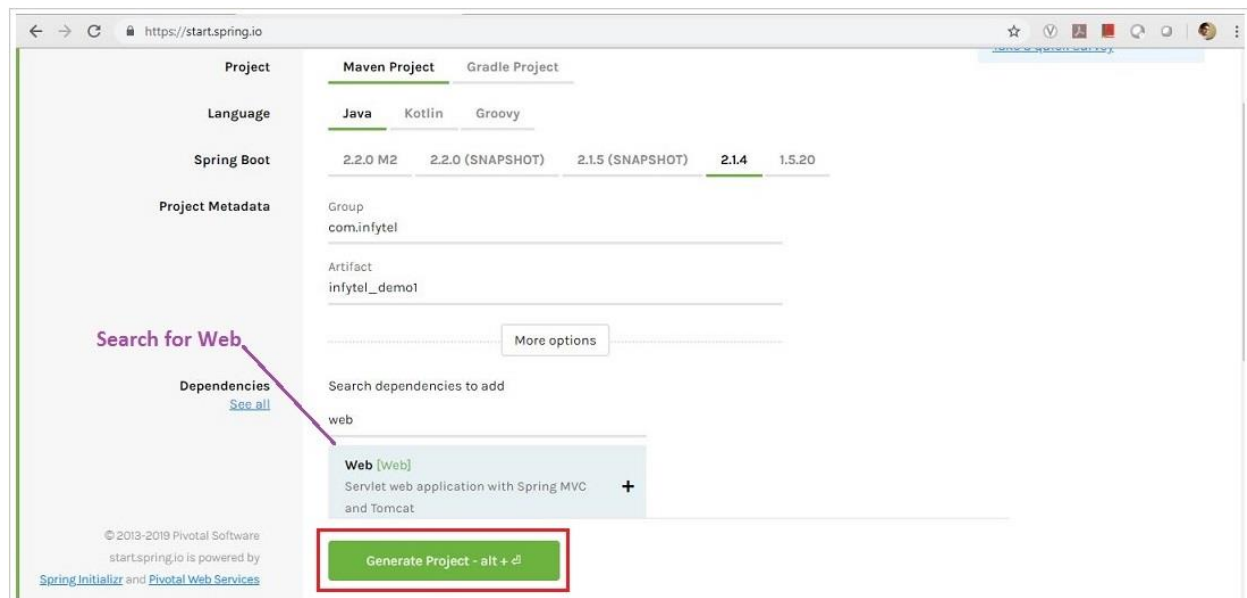
### 3.4.2 DEMO - REST USING SPRING BOOT

#### Objectives:

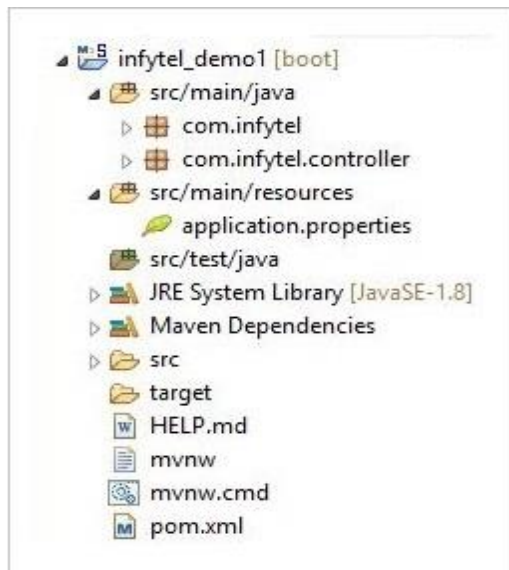
To create a Spring REST application using Spring Boot.

#### Steps:

**Step 1:** Create a Maven project using Spring initializer with maven web dependencies and import the same in STS.



**Step 2:** Modify the imported project according to the following project structure:



**Step 3:** Look at the class, **InfytelDemo1Application** in com.infytel package, this will be auto-created when the project is generated. We will modify this only when we need to include custom configurations

**Step 4:** Create a class CustomerController in com.infytel.controller package

**Step 5:** Add the following content to application.properties file

**Step 6:** Make sure that the project's pom.xml looks similar to the one that is shown below.

**Step 7:** Deploy the service on the server by executing the class containing the main method.

So, we have successfully created and deployed RESTful web service. Now let us see how the same can be tested using Postman client.

### 3.5 @REQUESTBODY AND RESPONSEENTITY

**@RequestBody** is the annotation that helps map our HTTP request body to a Java DTO. And, this annotation has to be applied on the local parameter (Java DTO) of the request method.

Whenever Spring encounters @RequestBody, it takes the help of the registered `HttpMessageConverters` which will help convert the HTTP request body to Java Object depending on the MIME type of the Request body.

Example: In the below code snippet, the incoming HTTP request body is deserialized to CustomerDTO. If the MIME type of the incoming data is not mentioned, it will be considered as JSON by default and Spring will use the JSON message converter to deserialize the incoming data.

```
@PostMapping
Public string createcustomer(@RequestBodycustomerDTOcustomerDTO)
{
```

```
//logic goes here
}
```

How to send a Java Object in the response?

### Scenario-1

Java objects can be returned by the handler method just like how the normal Java methods can return an object.

Example: In the below example, the `fetchCustomer()` returns a list of Customer Objects. This list will be converted by Spring's message converter to JSON data.

```
@GetMapping
Public List<CustomerDTO>fetchcustomer()
{
    //business logic goes here
    Return customerService.fetchCustomer();
}
```

### Scenario-2

We can specify the MIME type, to which the data to be serialized, using the **produces** attribute of HTTP method matching annotations

```
@GetMapping(produces="application/json")
Public List<customerDTO>fetchCustomer()
{
    //This method will return the customer of Infytel
    returncustomerService.fetchcustomer();
}
```

While sending a response, we may like to set the HTTP status code and headers .To help achieving this, we can use `ResponseEntity` class.

**`ResponseEntity<T>`** Will help us add a `HttpStatus` status code and headers to our response.

**Example:** In the below code snippet, `createCustomer()` method is returning a String value and setting the status code as 200.

**Below is the list of constructors available to create Response Entity**

Constructor	DESCRIPTION
<code>ResponseEntity(HttpStatus status)</code>	Creates a <code>ResponseEntity</code> with only status code and no



	body
ResponseEntity(MultiValueMap<String,String> headers, HttpStatus status)	Creates a ResponseEntity object with headers and statuscode but, no body
ResponseEntity(T body, HttpStatus status)	Creates a ResponseEntity with a body of type T and HTTP status
ResponseEntity(T body, MultiValueMap<String,String> headers, HttpStatus status)	Creates a ResponseEntity with a body of type T, header and HTTP status

### 3.5.1 DEMO HANDLING IN JAVA OBJECTS

#### Objectives:

To create a Spring REST application using Spring Boot, where the handler methods of a REST controller, consumes and produces Java objects. We will learn,

- The usage of **produces** and **consumes** attributes of HTTP method handling annotations.
- The usage of **ResponseEntity**
- 

**Scenario:** An online telecom app called **Infytel** is exposing its customer management profile as a Restful service. The Customer resource titled CustomerController allows us to create, fetch, delete and update customer details. This demo works with the following HTTP operations.

#### Steps:

**Step 1:** Create a Maven project using Spring Initializer with web dependency and import the same in STS.

**Step 2:** Modify the imported project according to the following project structure:



**Step 3:** Look at the class `InfytelDemo2Application` in `com.infytel` package that gets created automatically when the project is generated. We will modify this only when we need to go with custom configuration.

**Step 4:** Create the class **CustomerController** under `com.infytel.controller` package

**Step 5:** Create the DTO classes **Customer DTO**, **FriendFamilyDTO** and **PlanDTO** under `com.infytel.dto` package

**Step 6:** Create the class **CustomerRepository** under `com.infytel.repository` package

**Step 7:** Create the class **CustomerService** under `com.infytel.service` package

**Step 8:** Add the following content to **application.properties** file

**Step 9:** Make sure that the project's `pom.xml` looks similar to the one that is shown below.

**Step 10:** Deploy the application by executing the class that contains the main method.

So, we have successfully created and deployed the REST endpoints. Now, let us see how to test the same using Postman client.

### 3.5.2 PARAMETER INJECTION

It is not always that the client prefers sending the data as request body. The client can even choose to send the data as part of the request URI. For example, if the client feels that the data is not sensitive and doesn't need a separate channel to get transferred. So, how to receive this kind of data that appears in the URI.

Before we look into the ways to extract the URI data, we will have a look at the formal categorization of the data that appear in the request URI.

- Query Parameter
- Path Variables

- Matrix Variables

### Query Parameter:

- Query parameters or request parameters usually travel with the URI and are delimited by question mark.
- The query parameters in turn are delimited by ampersand from one another.
- The annotation `@RequestParam` helps map query/request parameters to the method arguments.

### Path Variables :

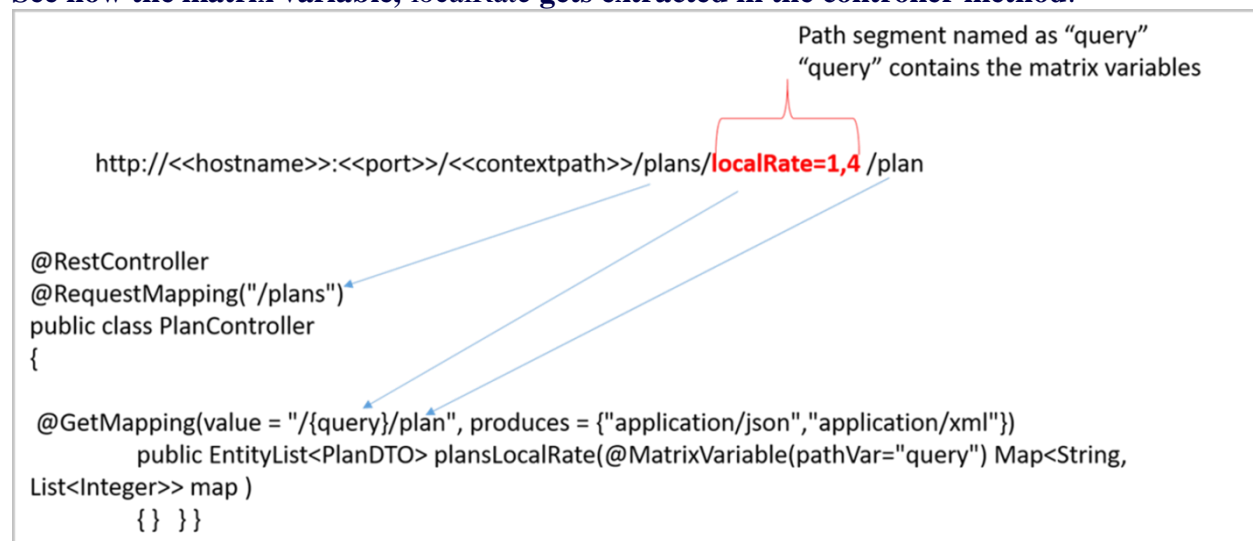
- Path variables are usually available at the end of the request URIs delimited by slash (/).
- `@PathVariable` annotation is applied on the argument of the controller method whose value needs to be extracted out of the request URI.
- A request URI can have any number of path variables.

### Matrix variables:

- Matrix variables are a block/segment of values that travel along with the URI. For example, `/localRate=1,2,3/`
- These variables may appear in the middle of the path unlike query parameters which appear only towards the end of the URI.
- Matrix variables follow **name=value** format and use semicolon to get delimited from one other matrix variable.
- A matrix variable can carry any number of values, delimited by commas.
- `@MatrixVariable` is used to extract the matrix variables.

**URI:**`http://<<hostname>>:<<port>>/<<contextpath>>/plans/localRate=1,4 /plan`

**See how the matrix variable, localRate gets extracted in the controller method.**



## Code:

```
@RestController
@RequestMapping("/plans")
public class PlanController
{
    //{query} here is a place holder for the matrix variables that travel in the URI,
    //it is not mandatory that the client URI should hold a string literal called query
    @GetMapping(value =("/{query}/plan", produces = {"application/json","application/xml"})
    publicEntityList<PlanDTO>plansLocalRate(
        @MatrixVariable(pathVar="query") Map<String, List<Integer>> map ) {
        //code goes here
    }
}
```

**@MatrixVariable(pathVar="query") Map<String, List<Integer>> map** :The code snippet mentions that all the matrix variables that appear in the path segment of name **query** should be stored in a Map instance called **map**. Here, the map's key is nothing but the name of the matrix variable and that is nothing but **localRate**. And, the **value** of the map is a collection of **localRates (1,4)** of type Integer.

### 3.5.3 DEMO-USAGE OF @PATH VARIABLE

#### Testing the REST endpoints using Postman

**Step 1:** Launch Postman.

**Step 2:** Test GET request

From the dropdown, select GET and enter **http://localhost:8080/infytel-3/customers** into the URL field to test the service. Click on Send. The following response will be generated

**Step 3:** Test PUT request

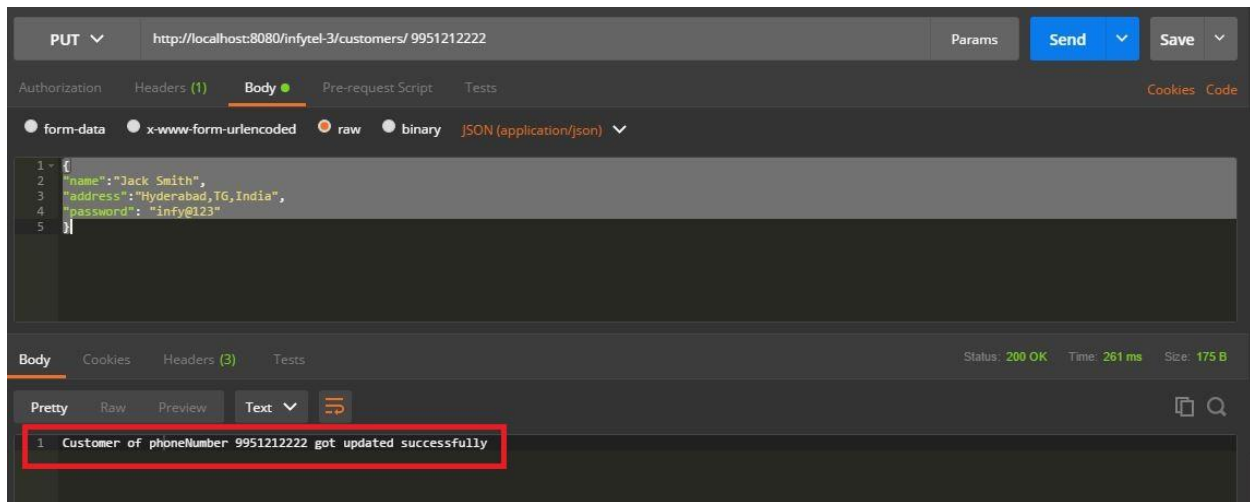
From the drop-down, select PUT and enter **http://localhost:8080/infytel-3/customers/9951212222** into the URL field to test the service. And, click on the body to enter the following JSON data. The below data is used to update the details of an existing customer (customer of phone number, 9951212222)

The JSON data has to match the DTO object to which it is converted.

```
{
  "name":"Jack Smith",
  "address":"Hyderabad,TG,India",
  "password": "infy@123"
}
```

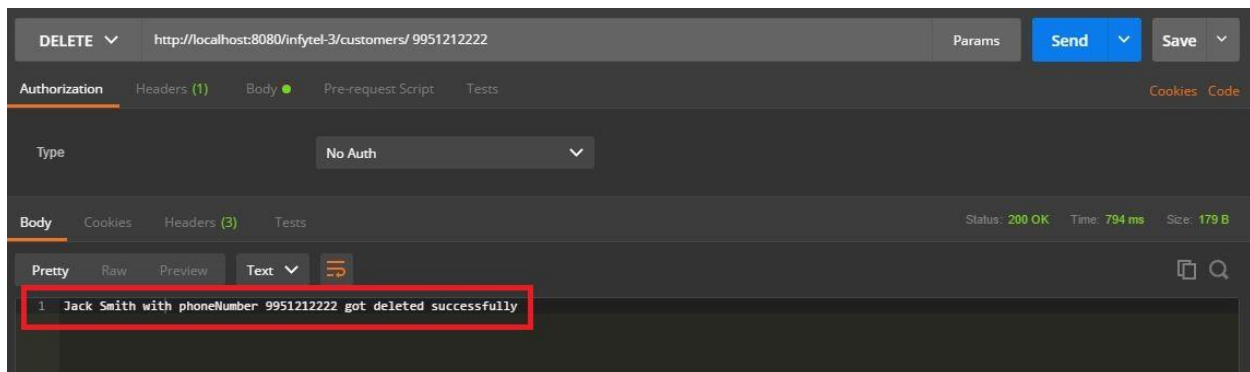
**Note:** Observe the CustomerDTO's variable names. It is important that the variable name in CustomerDTO and JSON key has to match exactly, for the correct mapping to happen.

Click on Send. Following will be the response.



#### Step 4: Test DELETE request

From the drop down, select DELETE and enter `http://localhost:8080/infytel-3/customers/ 9951212222` into the URL field to test the service. Click on Send. It deletes the customer with the given phone number and generates the following response



Now, from the drop down, select GET and enter the URL `http://localhost:8080/infytel-3/customers` into the URL field to test the service. Click on Send. Following will be the response



Notice that the details of the customer of phone number, 9951212222 is not appearing while fetching the data after the delete operation

### 3.6 EXCEPTION HANDLING

It is important to map the exceptions to objects that can provide some specific information which allows the API clients to know, what has happened exactly. So, instead of returning a String, we can send an object that holds the error message and error code to the client back.

Let us look at an example:

The object which is going to hold a custom error message is **ErrorMessage** with two variables, errorcode and message.

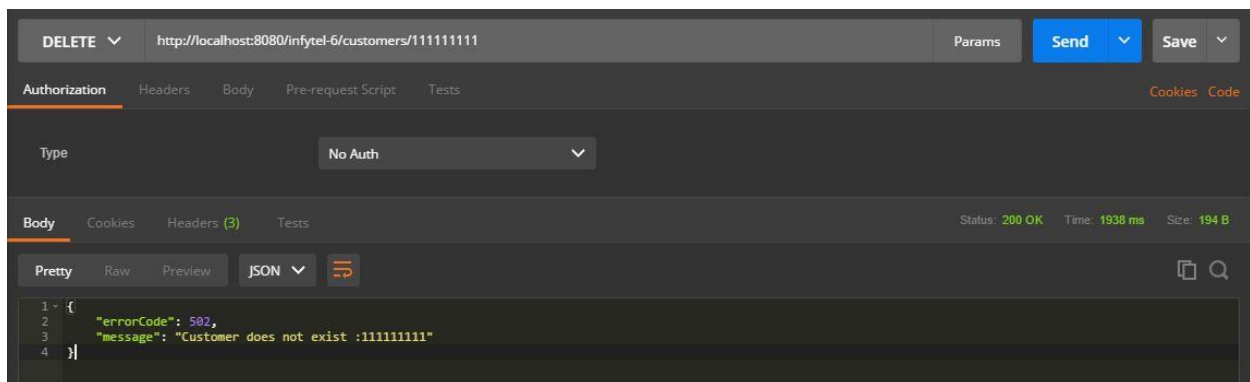
```
public class ErrorMessage {  
  
    private int errorCode;  
  
    private String message;  
  
    //getters and setters go here  
  
}
```

#### 3.6.1 DEMO EXCEPTION HANDLING

Testing Web Service using Postman

**Step 1:** Launch Postman.

**Step 2:** Test this URL - **http://localhost:8080/infytel-6/customers/111111111** using HTTP DELETE that will delete the customer based on the phone number (invalid phoneNumber that does not exist in Infytel) being passed.



#### 3.6.2 EXERCISE –EXCEPTION HANDLING

CookPick online grocery application has a requirement where the admin should be able to delete a product based on its product code.

Write a RESTful endpoint that would serve this purpose. Also, handle the code in such a way that it renders a customized error message when the product is not found.

RESTful URL	HTTP Action	Business Operation
/product/{productCode}	DELETE	deleteProduct()

This method,

- Takes product code
- Renders a response entity with the appropriate status code and message

Hint: Status code-200 and message-“product deleted” if everything is ok

Status code-404 and message-“product is not found” otherwise

Note:

1. You can implement this requirement in the project that you used for previous exercise.
2. You can simply hardcode sample product details in a collection and delete the product details from the collection in your logic. It is optional to use a database to delete the product details.

\*Time given for this exercise indicates the time required to create the REST endpoint and is excluding the time required to write the logic for deleting the product details from database.

### 3.6.3 DATA VALIDATION

So, we know how to apply validation on the incoming objects and URI parameters.

Now, its time to handle the validation failures, if any. We have attempted to handle the validation exceptions with respect to the incoming DTOs in the earlier sample code. But, that was not in a centralized way. Only, the controller was coded to do so. This way of coding will lead to code duplication and the code will not be manageable as well.

So, the code that should be reached out during validation failures should also be made centralized as how we did for exceptions. We can enhance the **ExceptionHandlerAdvice** class .

Have a look at the code snippet below to understand the concept better.

```
@ExceptionHandler(MethodArgumentNotValidException.class)
```

```
publicResponseEntity<ErrorMessage>handleValidationExceptions(
```

```

MethodArgumentNotValidException ex) {

    ErrorMessage error = new ErrorMessage();

    error.setErrorCode(HttpStatus.BAD_REQUEST.value());

    error.setMessage(ex.getBindingResult().getAllErrors().stream()

        .map(ObjectError::getDefaultMessage)

        .collect(Collectors.joining(", ")));

    return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);

}

ExceptionHandler(ConstraintViolationException.class)

publicResponseEntity<ErrorMessage>handleConstraintValidationExceptions(

    ConstraintViolationException ex) {

    ErrorMessage error = new ErrorMessage();

    error.setErrorCode(HttpStatus.BAD_REQUEST.value());

    error.setMessage(ex.getConstraintViolations().stream()

        .map(ConstraintViolation::getMessage)

        .collect(Collectors.joining(", ")));

    return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);

}

```

### 3.7 CREATING A REST CLIENT

There are situations where a Spring REST endpoint might be in need of contacting other RESTful resources. Situations like this can be handled with the help of REST client that is available in the Spring framework. And, the name of this REST client is RestTemplate.

RestTemplate has a wonderful support for standard HTTP methods.

The methods of the RestTemplate need to be provided with the absolute URI where the service can be found, input data, if any and, the response type that is expected out of the service.



## Calling a RESTful service of HTTP request method Get:

The following code snippet shows how to consumes a RESTful service exposed by url :<http://localhost:8080/infytel/customers> using HTTP GET.

```
RestTemplaterestTemplate = new RestTemplate();

String url="http://localhost:8080/infytel/customers";

List<CustomerDTO>  customers  =  (List<CustomerDTO>)  restTemplate.getForObject(url,
List.class);

for(CustomerDTOcustomer:customers)

{

    System.out.println("Customer Name: "+customer.getName());

    System.out.println("Customer Phone: "+customer.getPhoneNo());

    System.out.println("Email Id: "+customer.getEmail());

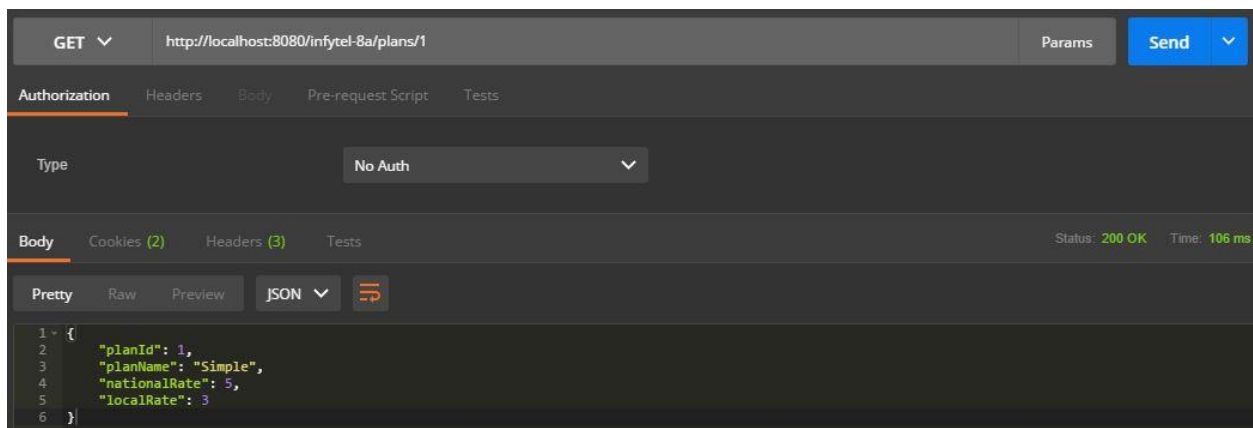
}
```

### 3.7.1 DEMO-CREATING A REST END POINT

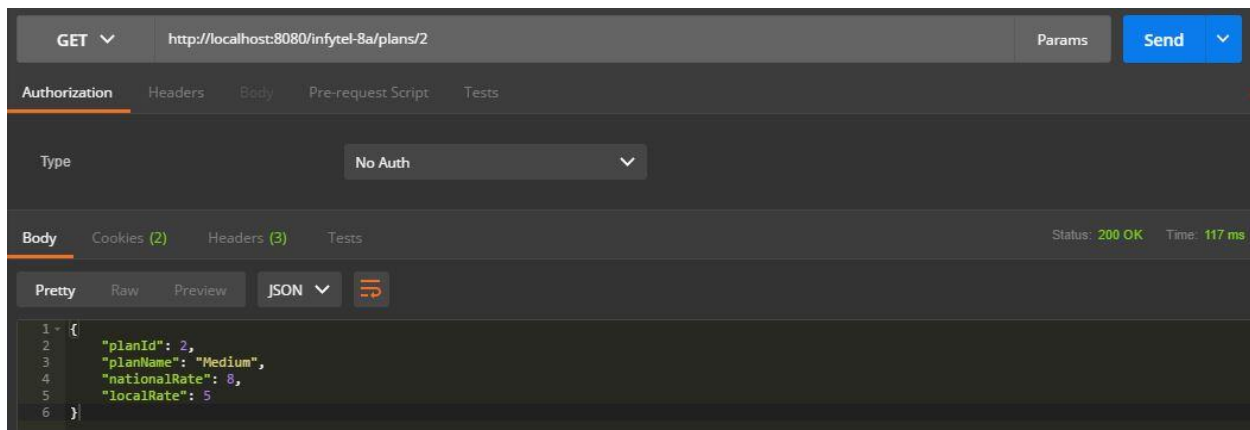
Testing Web Service using Postman

**Step 1:** Launch Postman.

**Step 2:** Test this URL - <http://localhost:8080/infytel-8a/plans/1> using HTTP GET to get the details of the existing planId, 1.



**Step 3:** Testing this URL - **http://localhost:8080/infytel-8a/plans/2** using HTTP GET to get the details of the existing planId, 2.



### 3.7.2 VERSIONING A SPRING REST END POINT

Request parameter versioning can be achieved by providing the request parameter in the URI that holds the version details.

GET `http://localhost:8080/infytel/plans/{planId}?version=`

eg:`http://localhost:8080/infytel/plans/1?version=1`

GET `http://localhost:8080/infytel/plans/{planId}?version=`

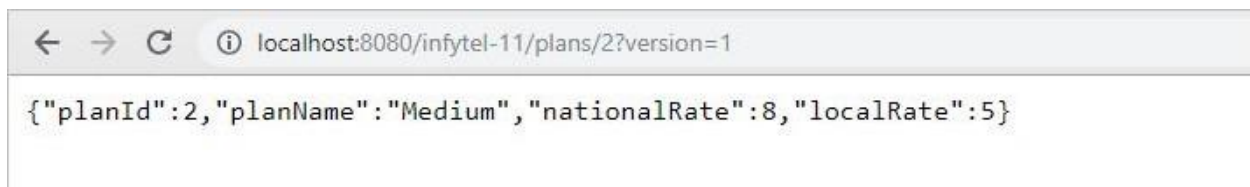
eg:`http://localhost:8080/infytel/plans/1?version=2`

### 3.7.3 DEMO-VERSIONING A SPRING REST END POINT

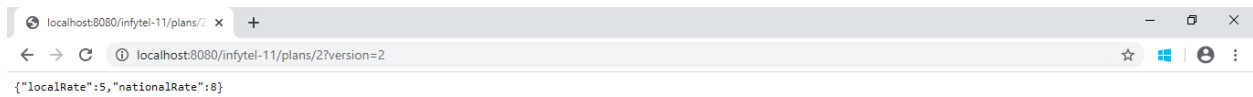
#### Testing Web Service

**Step 1:** Open any standard web browser.

**Step 2:** Test this URL - **http://localhost:8080/infytel-11/plans/2?version=1**. Following will be the response that contains the plan details with all the fields.

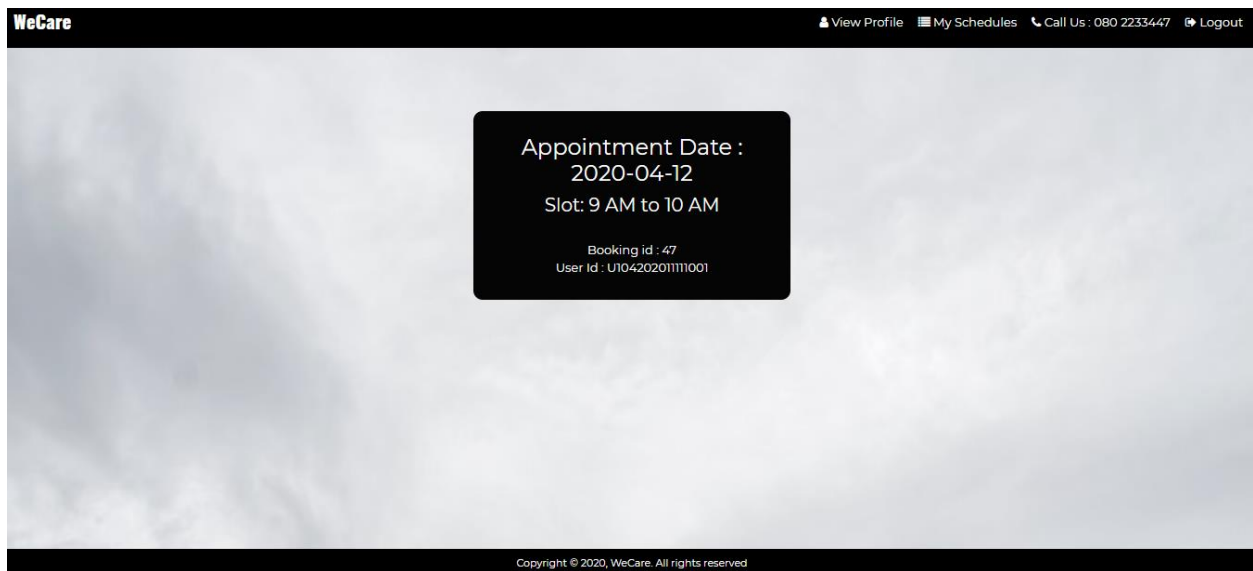


**Step 3:** Test this URL - **http://localhost:8080/infytel-11/plans/2?version=2**. Only the local and national rates of plan details will be rendered as output.



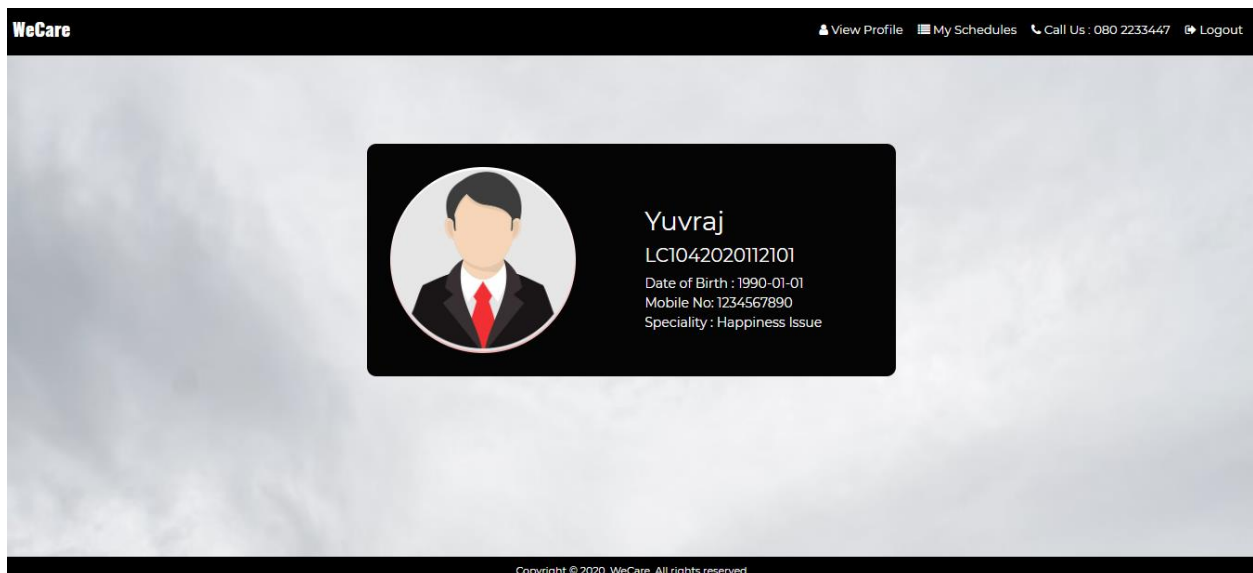
### 3.8 SPRING REST HEALTH CARE DOMAIN CAPSTONE –WECARE

#### Homepage and Schedule Screen (if there is any upcoming schedule)



The upcoming schedule will be displayed with a **GET** request at the backend.

#### View Profile Screen



The Life Coach can see the profile details with a **GET** request at the backend.

**CoachRestController**, **CoachService** and **BookService** classes are used here.

### 3.8.1 SPRING RESTTRANSPORT & LOGISTIC DOMAIN

#### Problem Statement:

As part of this course, you will be developing an application called **InfyRail** using Spring Boot.

InfyRail is an online train inquiry application which is used to check for the details of trains in a particular route. The current version of InfyRail is developed using Spring MVC. Now, the requirement is to create the Spring REST version of the same.

- InfyRail has two types of users, “Admin” and “User”.
- “Admin” can create a route, add and update train details to the existing routes, update fare details of trains and delete trains from the routes.
- “User” can search trains by route id and view trains of a particular source and destination.

Let us focus on the software and project setup, before having a look at the requirements in detail.

#### Software Requirement:

1. Spring Tool Suite(STS): STS is an Eclipse-based IDE for developing Spring-based applications.
2. JDK (min version required 8): JDK (Java Development Kit) provides tools for developing, debugging and monitoring, as well as the Java runtime environment (JRE) for Java applications
3. Maven 3.x: The most popular software project management and comprehensive tool to simplify the build process of any Java-based project
4. MySQL database: Raise AHD request and contact local CCD to install as it needs admin privilege. You can also choose any relational database of your choice.
5. Postman: It is a popular tool to send different HTTP requests to RESTful web services.

#### Project Setup:

##### MySQL Database setup

- Open the command prompt and navigate to the folder which has MySQL server bin folder.
- Enter the command `mysqld.exe` to start the MySQL Server

**Note:** You can ignore create table command if your application is configured to auto-create tables using Hibernate/JPA properties.

Find below functionalities that need to be exposed as REST endpoints.

**Requirement 1:** Create a route. Route id should get auto-generated and it should be a 3 digit number.

Also, validate the incoming Route DTO. Validation details are as follows,

- source and destination should not be empty and must contain only alphabets
- trainList should not be empty

URL	HTTP Action	Business Operation
/routes	POST	This request should create a route in the database and return the route id generated

**Requirement 2:** View route details by route id. Also, validate whether the incoming path variable "route id" is a number that comprises of 3 digits.

URL	HTTP Action	Business Operation
/routes/{routeId}	GET	This request should take the routeID as a path parameter and give the details of that particular route

**Requirement 3:** View train details by source and destination. Also, validate whether the incoming request parameters contain only alphabets.

URL	HTTP Action	Business Operation
/routes/trains?source=&destination=	GET	This request should fetch a list of trains available for a particular source and destination

**Requirement 4:** Update route details.

URL	HTTP Action	Business Operation
/routes/{routeId}	PUT	This request should update the source and destination based on the routeId  Note: Take source and destination as matrix parameters

**Requirement 5:** Delete a train from the list of trains in a particular route.

URL	HTTP Action	Business Operation
/routes/{routeId}/{trainId}	DELETE	This request should delete a train from the list of trains available in a particular route.

**Requirement 6:** Update train details in the existing route.

URL	HTTP Action	Business Operation
/routes/{routeId}	PUT	This request should update train details in an existing route.  Note: Take train details in JSON format

**Requirement 7:** Create train details.

URL	HTTP Action	Business Operation
/trains	POST	This request should add the details of a train to the database and return the train id

**Requirement 8:** Update fare.

URL	HTTP Action	Business Operation
/trains/{trainId}	PUT	This request should update the fare of a particular train and return the success message

**Project implementation should also include the following:**

- Use ValidationMessages.properties file to keep the messages related to validation failures and exceptions.
- Validation failures and exceptions should be handled in a centralized way using controller advice class.
- Set server port number of your choice and application context path as **/InfyRail** for InfyRail boot project.

\*\* \*\* \*