

Spring Boot & Micro Services

By –

Dilip Singh

 dilipitacademy@gmail.com

Follow Here for Updates



@DilipItAcademy

+91 8125262702

© [2025] Dilip IT Academy. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher, Dilip IT Academy.

Legal Disclaimer: The information provided in this book is for educational and informational purposes only. While the publisher has made every effort to ensure the accuracy and completeness of the information contained herein, we assume no responsibility for errors, omissions, or contrary interpretation of the subject matter. The reader assumes full responsibility for the use of the information contained in this book.

Any unauthorized use, sharing, reproduction, or distribution of this book, in whole or in part, is strictly prohibited. Legal action, including but not limited to injunctive relief, damages, and attorney's fees, will be taken against individuals or entities found in violation of these terms.

Contents

Introduction:

What is Framework
Framework V/S Programming Language
Prerequisites of Spring & Spring Boot
Spring & Spring Boot Overview
Spring & Spring Boot Release Versions History
Spring & Spring Boot Modules
Differences between Spring & Spring Boot
Approaches to create Spring Boot Application
Spring and Spring Boot Architecture
Importance of Spring & Spring Boot Modules in Real Time Projects

Core Module with Spring

Introduction and Importance of Core Module
Inversion of Control (IOC) principle
Dependency Injection
Spring Container
Spring Beans
Bean Class
Beans Creation
Beans Configuration with XML
Bean Factory
Bean Factory V/S Application Context
Bean Life Cycle
Bean Scopes

- Singleton
- Prototype
- Request
- Session
- Application
- Websocket

Beans Wiring
Types of Dependency Injection

- Constructor Injection
- Setter Injection
- Field Injection

Beans Configuration with Java Annotations
Configuration Classes
@Bean annotation
@Configuration Annotations
Base Package Naming convention
Component class
Component Scanning
@Component annotation

- @ComponentScan Annotation**
- Auto Wiring of Beans**
- @Autowired annotation**
- @Qualifierannotation**
- @Primary annotation**

Implementation of Core Module with Spring Boot

- Creating Spring Boot Application**
- Spring Initializer (start.spring.io)**
- Spring Starter Wizard in STS IDE**
- Spring Boot Approach with Maven**
- Introduction to Spring Boot Starters**
- Spring Boot Parent Starter**
- Spring-boot-starter**
- Spring-boot-starter-web**
- Spring-boot-starter-data-jpa**
- Spring-boot-devtools**
- Spring-boot-actuator**
- @SpringBootApplication annotation**
- SpringApplication.run(..) method**
- Spring Boot Application Boot strapping**
- AutoConfiguration in Spring Boot**
- Runners in Spring Boot**
 - a) Application Runner
 - b) CommandLine Runner

JDBC/ORM/JPA Module

- Spring JDBC Module**
 - JdbcTemplate**
 - DataSource**
 - RowMapper**
- SpringBoot JDBC Module Implementation**
- JPA Module with Spring and SpringBoot**
 - What is JPA**
 - Spring Data JPA Introduction**
 - What is ORM**
 - ORM Basics**
 - What is Persistence Layer**
 - Hibernate Integration with JPA**
 - What is Entity Class**
 - JPA Annotations**
 - Repository Interfaces**

- CurdRepository introduction**
- Database CRUD Operations**
- Internal Flow of Database Query Creation**

CurdRepository methods for DB Operations
Derived Query Methods in JPA
Native Queries Execution in JPA
JpaRepository introduction
JpaRepository methods for DB Operations
CurdRepository V/S JpaRepository
What is a Transaction in Database
Transaction Management
Pagination Using Data JPA methods
Sorting Using Data JPA Methods
Async Data JPA
Asynchronous Calls with JPA

MVC Module

Spring Web MVC Introduction
Spring Web MVC Advantages
MVC Architecture
Creation of Spring MVC Application
Understanding Spring Web MVC flow
What is Front Controller & Front Controller Design Pattern
What is Dispatcher Servlet
Handler Mappers / Mappings
Controller, Service, Repository Layers
Stereo Type Annotations
 @Controller
 @Service
 @Repository
Creation of SpringBootWeb MVC Application
Difference between Spring & SpringBoot MVC Application
MVC Module Annotations
 @RestController
 @RequestBody
 @RequestMapping
 @ResponseBody
 @PathVariable
 @RequestParam
 @GetMapping
 @PostMapping
 @PutMapping
 @DeleteMapping
MVC CRUD example
REST API/Services
 SOAPvsREST
 RESTful Services Introduction
 REST principles
 JSON Introduction
 XML vs JSON
 JACKSON API

- Converting Java object to JSON
- Converting JSON object to Java
- HTTP Protocol & Methods
- HTTP Status Codes
- HTTP Headers
- POSTMAN
- Swagger In Spring Boot
- REST Client Introduction
- RestTemplate
- Exception Handling in MVC Modules
- Validations on request Body properties
- View Resolvers
- Form Based application development
- Sending Request from UI to Controller
- Sending Response from Controller to UI

Spring Boot Security:

- Importance of Security in Application
- What is Authentication
- What is Authorization
- Basic Authentication
- Custom Security
- Database Security
- Security With Encryption
- JWT Introduction
- JWT Authentication
- Oauth Introduction

Miscellaneous

- DevTools In SpringBoot
- Profiles in Spring Boot
- Actuator in Spring Boot

Development TOOLS

- Maven
- Logging
- POSTMAN
- LomBok

Micro Services :

- Monolith Architecture Introduction
- Monolith Architecture case study
- Monolith Application Deployment Process
- Monolith Architecture Drawbacks
- Micro services Introduction
- Micro Services Advantages
- Micro Services Dis-Advantages
- Micro Services case study

Identifying Micro services boundaries
Micro services Architecture
Micro services Development
API Gateway
Service Registry
Service Discovery
Load Balancer
Interservice communication
 RestTemplate
 Feign Clients
Config Server
Circuit Breaker

@DilipItAcademy

Spring Boot Core Module

Before starting with Spring framework, we should understand more about **Programming Language vs Framework**. The difference between a programming language and a framework is obviously need for a programmer. I will try to list the few important things that students should know about programming languages and frameworks.

What is a programming language?

Shortly, it is a set of keywords and rules of their usage that allows a programmer to tell a computer what to do. From a technical point of view, there are many ways to classify languages - compiled and interpreted, functional and object-oriented, low-level and high-level, etc..

do we have only one language in our project?

Probably not. Majority of applications includes at least two elements:

- **The server part.** This is where all the "heavy" calculations take place, background API interactions, Database write/read operations, etc.
Languages Used : Java, .net, python etc..
- **The client part.** For example, the interface of your website, mobile applications, desktop apps, etc.
Languages Used : HTML, Java Script, Angular, React etc.

Obviously, there can be much more than two languages in the project, especially considering such things as SQL used for database operations.

What is a Framework?

When choosing a technology stack for our project, we will surely come across such as framework. A framework is a set of ready-made elements, rules, and components that simplify the process and increase the development speed. Below are some popular frameworks as an example:

- JAVA : Spring, SpringBoot, Struts, Hibernate, Quarkus etc..
- PHP Frameworks: Laravel, Symfony, Codeigniter, Slim, Lumen
- JavaScript Frameworks: ReactJs, VueJs, AngularJs, NodeJs
- Python Frameworks: Django, TurboGears, Dash

What kind of tasks does a framework solve?

Frameworks can be general-purpose or designed to solve a particular type of problems. In the case of web frameworks, they often contain out-of-the-box components for handling:

- Routing URLs
- Security
- Database Interaction,
- caching
- Exception handling, etc.

Do I need a framework?

- **It will save time.** Using premade components will allow you to avoid reinventing the logics again and writing from scratch those parts of the application which already exist in the framework itself.
- **It will save you from making mistakes.** Good frameworks are usually well written. Not always perfect, but on average much better than the code your team will deliver from scratch, especially when you're on a short timeline and tight budget.
- **Opens up access to the infrastructure.** There are many existing extensions for popular frameworks, as well as convenient performance testing tools, CI/CD, ready-to-use boilerplates for creating various types of applications.

Conclusion:

While a programming language is a foundation, a framework is an add-on, a set of components and additional functionality which simplifies the creation of applications. In My opinion - using a modern framework is in **95%** of cases a good idea, and it's **always** a great idea to create an application with a framework rather than raw language.

Spring Introduction

The Spring Framework is a popular Java-based application framework used for building enterprise-level applications. It was developed by Rod Johnson in 2003 and has since become one of the most widely used frameworks in the Java ecosystem. The term "Spring" means different things in different contexts.



The framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications, with support for features such as dependency injection, aspect-oriented programming, data access, and transaction management. Spring handles the infrastructure so you can focus on your application. A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

One of the key features of the Spring Framework is its ability to promote loose coupling between components, making it easier to develop modular, maintainable, and scalable applications. The framework also provides a wide range of extensions and modules that can be used to integrate with other technologies and frameworks, such as Hibernate, Struts, and JPA.

Overall, the Spring Framework is widely regarded as a powerful and flexible framework for building enterprise-level applications in Java.

The Spring Framework provides a variety of features, including:

- **Dependency Injection:** Spring provides a powerful dependency injection mechanism that helps developers write code that is more modular, flexible, and testable.
- **Inversion of Control:** Spring also provides inversion of control (IoC) capabilities that help decouple the application components and make it easier to manage and maintain them.
- **AOP:** Spring's aspect-oriented programming (AOP) framework helps developers modularize cross-cutting concerns, such as security and transaction management.
- **Spring MVC:** Spring MVC is a popular web framework that provides a model-view-controller (MVC) architecture for building web applications.
- **Integration:** Spring provides integration with a variety of other popular Java technologies, such as Hibernate, JPA, JMS.

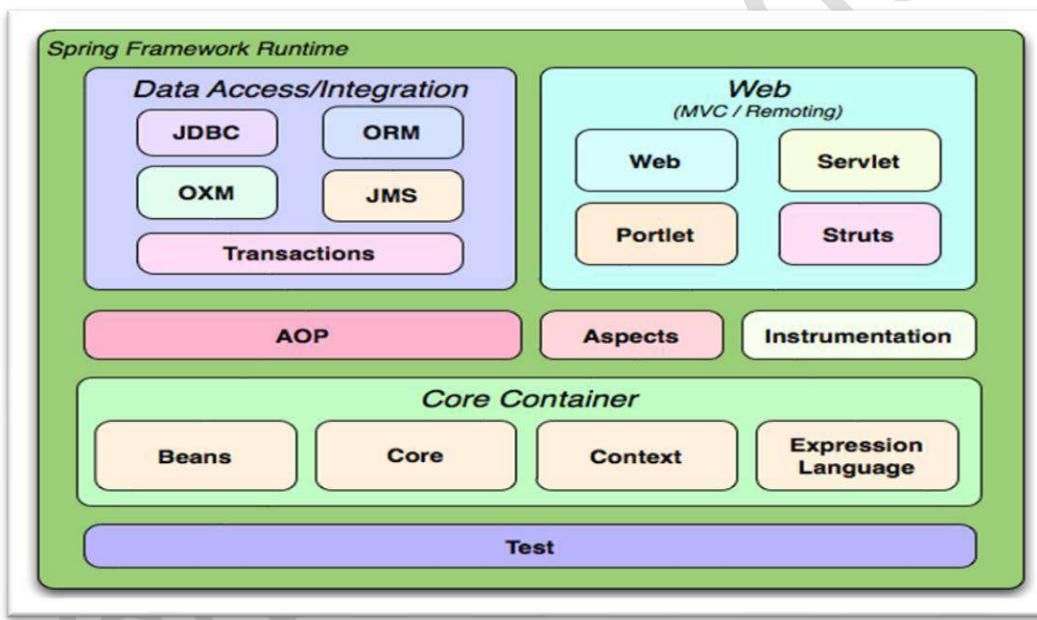
Overall, Spring Framework has become one of the most popular Java frameworks due to its ease of use, modularity, and extensive features. It is widely used in enterprise applications, web applications, and other types of Java-based projects.

Spring continues to innovate and to evolve. Beyond the Spring Framework, there are other projects, such as Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, among others.

Spring Framework architecture is an arranged layered architecture that consists of different modules. All the modules have their own functionalities that are utilized to build an application.

The Spring Framework includes several modules that provide a range of services:

- **Spring Core Container:** this is the base module of Spring and provides spring containers (BeanFactory and ApplicationContext).
- **Aspect-oriented programming:** enables implementing cross-cutting concerns.
- **Data access:** working with relational database management systems on the Java platform using Java Database Connectivity (JDBC) and object-relational mapping tools and with NoSQL databases
- **Authentication and authorization:** configurable security processes that support a range of standards, protocols, tools and practices via the Spring Security sub-project.
- **Model–View–Controller:** an HTTP- and servlet-based framework providing hooks for web applications and RESTful (representational state transfer) Web services.
- **Testing:** support classes for writing unit tests and integration tests



Spring Release Version History:

Version	Date	Notes
0.9	2003	
1.0	March 24, 2004	First production release.
2.0	2006	
3.0	2009	
4.0	2013	
5.0	2017	
6.0	November 16, 2022	Current/Latest Version

Advantages of Spring Framework:

The Spring Framework is a popular open-source application framework for developing Java applications. It provides a number of advantages that make it a popular choice among developers. Here are some of the key advantages of the Spring Framework:

1. **Lightweight:** Spring is a lightweight framework, which means it does not require a heavy runtime environment to run. This makes it faster and more efficient than other frameworks.
2. **Inversion of Control (IOC):** The Spring Framework uses IOC to manage dependencies between different components in an application. This makes it easier to manage and maintain complex applications.
3. **Dependency Injection (DI):** The Spring Framework also supports DI, which allows you to inject dependencies into your code at runtime. This makes it easier to write testable and modular code.
4. **Modular:** Spring is a modular framework, which means you can use only the components that you need. This makes it easier to develop and maintain applications.
5. **Loose Coupling:** The Spring applications are loosely coupled because of dependency injection.
6. **Integration:** The Spring Framework provides seamless integration with other frameworks and technologies such as Hibernate, Struts, and JPA.
7. **Aspect-Oriented Programming (AOP):** The Spring Framework supports AOP, which allows you to separate cross-cutting concerns from your business logic. This makes it easier to develop and maintain complex applications.
8. **Security:** The Spring Framework provides robust security features such as authentication, authorization, and secure communication.
9. **Transaction Management:** The Spring Framework provides robust transaction management capabilities, which make it easier to manage transactions across different components in an application.
10. **Community Support:** The Spring Framework has a large and active community, which provides support and contributes to its development. This makes it easier to find help and resources when you need them.

Overall, the Spring Framework provides a number of advantages that make it a popular choice among developers. Its lightweight, modular, and flexible nature, along with its robust features for managing dependencies, transactions, security, and integration, make it a powerful tool for developing enterprise-level Java applications.

Why do we use Spring in Java?

- Works on POJOs (Plain Old Java Object) which makes your application lightweight.
- Provides predefined templates for JDBC, Hibernate, JPA etc., thus reducing your effort of writing too much code.
- Because of dependency injection feature, your code becomes loosely coupled.
- Using Spring Framework, the development of Java Enterprise Edition (JEE) applications became faster.
- It also provides strong abstraction to Java Enterprise Edition (JEE) specifications.
- It provides declarative support for transactions, validation, caching and formatting.

What is the difference between Java and Spring?

The below table represents the differences between Java and Spring:

Java	Spring
Java is one of the prominent programming languages in the market.	Spring is a Java-based open-source application framework.
Java provides a full-highlighted Enterprise Application Framework stack called Java EE for web application development	Spring Framework comes with various modules like Spring MVC, Spring Boot, Spring Security which provides various ready to use features for web application development.
Java EE is built upon a 3-D Architectural Framework which are Logical Tiers, Client Tiers and Presentation Tiers.	Spring is based on a layered architecture that consists of various modules that are built on top of its core container.

Since its origin till date, Spring has spread its popularity across various domains.

Spring Core Module

Core Container:

Spring Core Module has the following three concepts:

1. **Spring Core:** This module is the core of the Spring Framework. It provides an implementation for features like IOC (Inversion of Control) and Dependency Injection with a singleton design pattern.
2. **Spring Bean:** This module provides an implementation for the factory design pattern through BeanFactory.
3. **Spring Context:** This module is built on the solid base provided by the Core and the Beans modules and is a medium to access any object defined and configured.

Spring Bean:

Beans are java objects that are configured at run-time by Spring IoC Container. In Spring, the objects of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by Spring container.

Dependency Injection in Spring:

Dependency Injection is the concept of an object to supply dependencies of another object. Dependency Injection is one such technique which aims to help the developer code easily by providing dependencies of another object. Dependency injection is a pattern we can

use to implement IoC, where the control being inverted is setting an object's dependencies. Connecting objects with other objects, or “**injecting**” objects into other objects, is **done by an container** rather than by the objects themselves.

When we hear the term dependency, what comes on to our mind? Obviously, something relying on something else for support, right? Well, that's the same, in the case of programming also.

Dependency in programming is an approach where a class uses specific functionalities of another class. So, for example, If you consider two classes A and B, and say that class A using functionalities of class B, then its implied that class A has a dependency of class B i.e. A depends on B. Now, if we are coding in Java then you must know that, you have to create an instance/Object of class B before the functionalities are being used by class A.

Dependency Injection in Spring can be done through constructors, setters or fields. Here's how we would create an object dependency in traditional programming:

Employee.java

```
public class Employee {
    private String ename;
    private Address addr;

    public Employee() {
        this.addr = new Address();
    }
    // setter & getter methods
}
```

Address.java

```
public class Address {
    private String cityName;
    // setter & getter methods
}
```

In the example above, we need to instantiate an implementation of the Address within the *Employee* class itself.

By using DI, we can rewrite the example without specifying the implementation of the *Address* that we want:

```
public class Employee {
    private String ename;
    private Address addr;

    public Employee(Address addr) {
        this.addr = addr;
    }
}
```

```
}
```

In the next sections, we'll look at how we can provide the implementation of *Address* through metadata. Both IoC and DI are simple concepts, but they have deep implications in the way we structure our systems, so they're well worth understanding fully.

Spring Container / IOC Container:

An IoC container is a common characteristic of frameworks that implement IoC principle in software engineering.

Inversion of Control:

Inversion of Control is a principle in software engineering, which transfers the control of objects of a program to a container or framework. We most often use it in the context of object-oriented programming.

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets information's from the XML file or Using annotations and works accordingly.

The main tasks performed by IoC container are:

- to instantiate the application java classes
- to configure data with the objects
- to assemble the dependencies between the objects internally

As I have mentioned above Inversion of Control is a principle based on which, Dependency Injection is made. Also, as the name suggests, Inversion of Control is basically used to invert different kinds of additional responsibilities of a class rather than the main responsibility.

If I have to explain you in simpler terms, then consider an example, wherein you have the ability to cook. According to the IoC principle, you can invert the control, so instead of you cooking food, you can just directly order from outside, wherein you receive food at your doorstep. Thus, the process of food delivered to you at your doorstep is called the Inversion of Control.

You do not have to cook yourself, instead, you can order the food and let a delivery executive, deliver the food for you. In this way, you do not have to take care of the additional responsibilities and just focus on the main work.

Spring IOC is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, objects dependencies are injected by other assembler objects.

Spring provides two types of Container Implementations namely as follows:

1. BeanFactory Container
2. ApplicationContext Container

Spring IoC container is the program that injects dependencies into an object and make it ready for our use. Spring IoC container classes are part of **org.springframework.beans** and **org.springframework.context** packages from spring framework. Spring IoC container provides us different ways to decouple the object dependencies. **BeanFactory** is the root interface of Spring IoC container. **ApplicationContext** is the child interface of BeanFactory interface. These Interfaces are having many implementation classes in same packages to create IOC container in execution time.

Spring Framework provides a number of useful **ApplicationContext** implementation classes that we can use to get the spring context and then the Spring Bean. Some of the useful **ApplicationContext** implementations that we use are.

- **AnnotationConfigApplicationContext**: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.
- **ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.
- **FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
- **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

Spring is actually a container and behaves as a factory of Beans.

Spring – BeanFactory:

This is the simplest container providing the basic support for DI and defined by the **org.springframework.beans.factory.BeanFactory** interface. BeanFactory interface is the simplest container providing an advanced configuration mechanism to instantiate, configure and manage the life cycle of beans. BeanFactory represents a basic IoC container which is a parent interface of **ApplicationContext**. BeanFactory uses Beans and their dependencies metadata i.e. what we configured in XML file to create and configure them at run-time. BeanFactory loads the bean definitions and dependency amongst the beans based on a configuration file(XML) or the beans can be directly returned when required using Java Configuration.

Spring ApplicationContext:

The **org.springframework.context.ApplicationContext** interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. Several implementations of the ApplicationContext interface are supplied with Spring. In standalone applications, it is common to create an instance of **ClassPathXmlApplicationContext** or **FileSystemXmlApplicationContext**. While XML has been the traditional format for defining configuration of spring bean classes. We can instruct the

container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

The following diagram shows a high-level view of how Spring Container works. Your application bean classes are combined with configuration metadata so that, after the **ApplicationContext** is created and initialized, you have a fully configured and executable system or application.

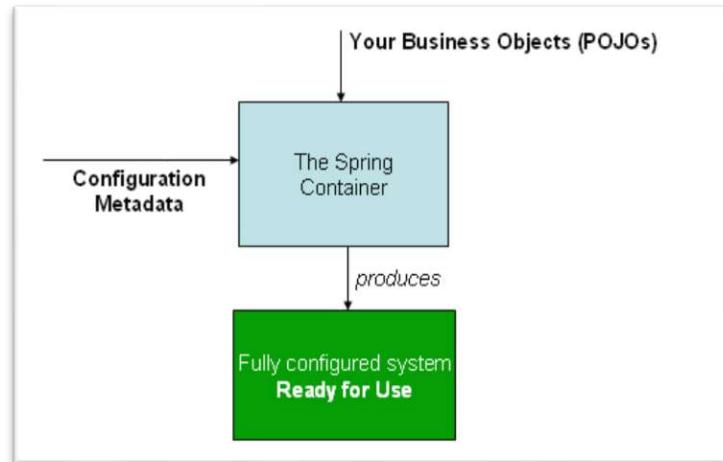


Figure: The Spring IoC container

Configuration Metadata:

As diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application. Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container. These days, many developers choose Java-based configuration for their Spring applications.

Instantiating a Container:

The location path or paths supplied to an **ApplicationContext** constructor are resource Strings that let the container load configuration metadata from a variety of external resources, such as the local file system, the Java CLASSPATH, and so on. The Spring provides **ApplicationContext** interface: **ClassPathXmlApplicationContext** and **FileSystemXmlApplicationContext** for standalone applications, and **WebApplicationContext** for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations. Here's one way to manually instantiate a container:

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

Difference Between BeanFactory Vs ApplicationContext:

BeanFactory	ApplicationContext
It is a fundamental container that provides the basic functionality for managing beans.	It is an advanced container that extends the BeanFactory that provides all basic functionality and adds some advanced features.
It is suitable to build standalone applications.	It is suitable to build Web applications, integration with AOP modules, ORM and distributed applications.
It supports only Singleton and Prototype bean scopes.	It supports all types of bean scopes such as Singleton, Prototype, Request, Session etc.
It does not support Annotation based configuration.	It supports Annotation based configuration in Bean Autowiring.
This interface does not provide messaging (i18n or internationalization) functionality.	ApplicationContext interface extends MessageSource interface, thus it provides messaging (i18n or internationalization) functionality.
BeanFactory will create a bean object when the getBean() method is called thus making it Lazy initialization.	ApplicationContext loads all the beans and creates objects at the time of startup only thus making it Eager initialization.

NOTE: Usually, if we are working on Spring MVC application and our application is configured to use Spring Framework, Spring IoC container gets initialized when the application started or deployed and when a bean is requested, the dependencies are injected automatically. However, for a standalone application, you need to initialize the container somewhere in the application and then use it to get the spring beans.

Create First Spring Core module Application:

NOTE: We can Create Spring Core Module Project in 2 ways.

1. Manually Downloading Spring JAR files and Copying/Configuring Build Path
2. By Using Maven Project Setup, Configuring Spring JAR files in Maven. This is preferred in Real Time practice.

In Maven Project, JAR files are always configured with **pom.xml** file i.e. we should not download manually JAR files in any Project. In First Approach we are downloading JAR files manually from Internet into our computer and then setting class Path to those jar file, this is not recommended in Real time projects.

Creation of Project with Downloaded Spring Jar Files :

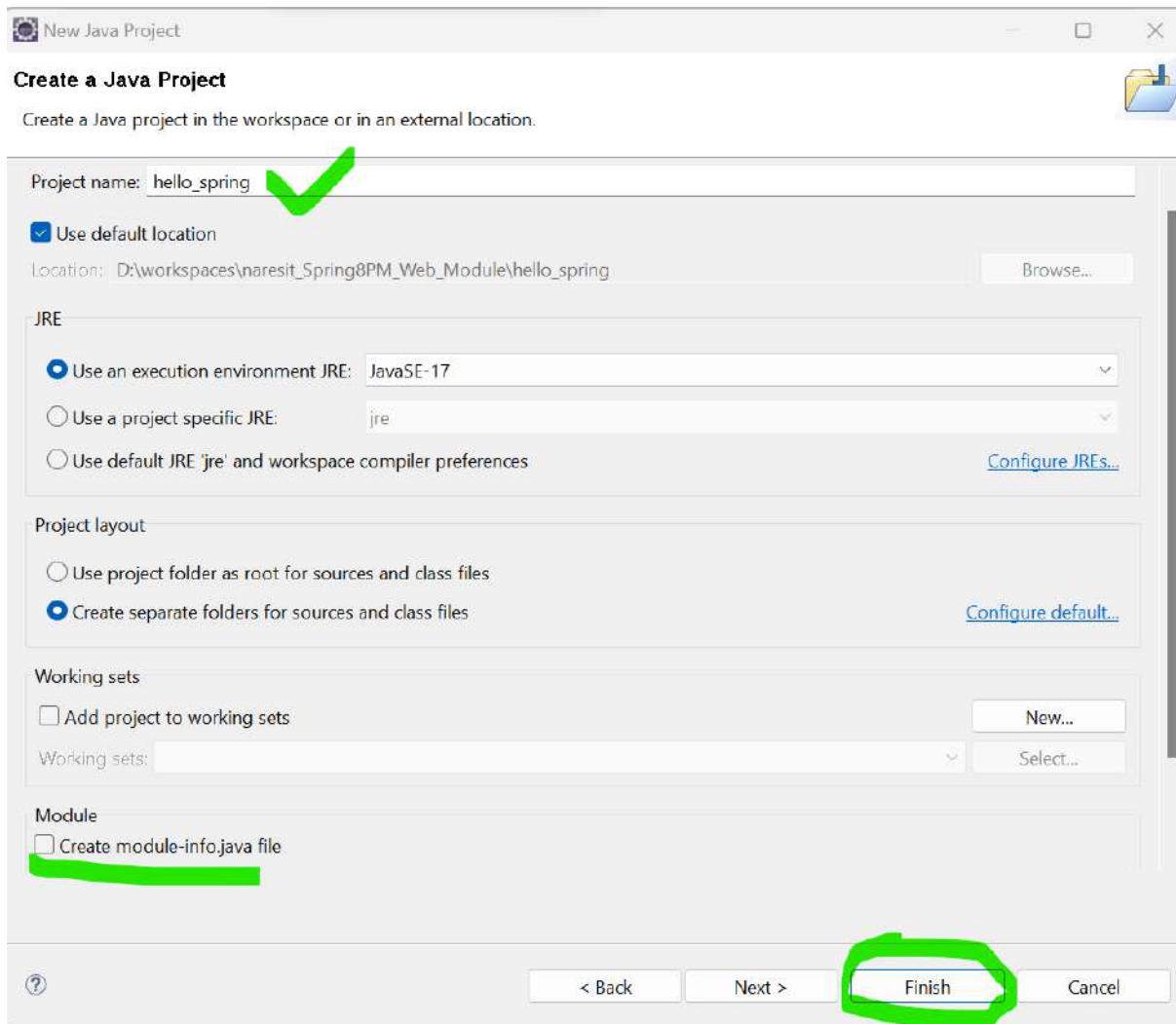
1. Open Eclipse

File -> new -> Project -> Java Project

Enter Project Name

Un-Select Create Module-Info

Click Finish.



2. Now Download Spring Framework Libraries/jar files from Online/Internet.

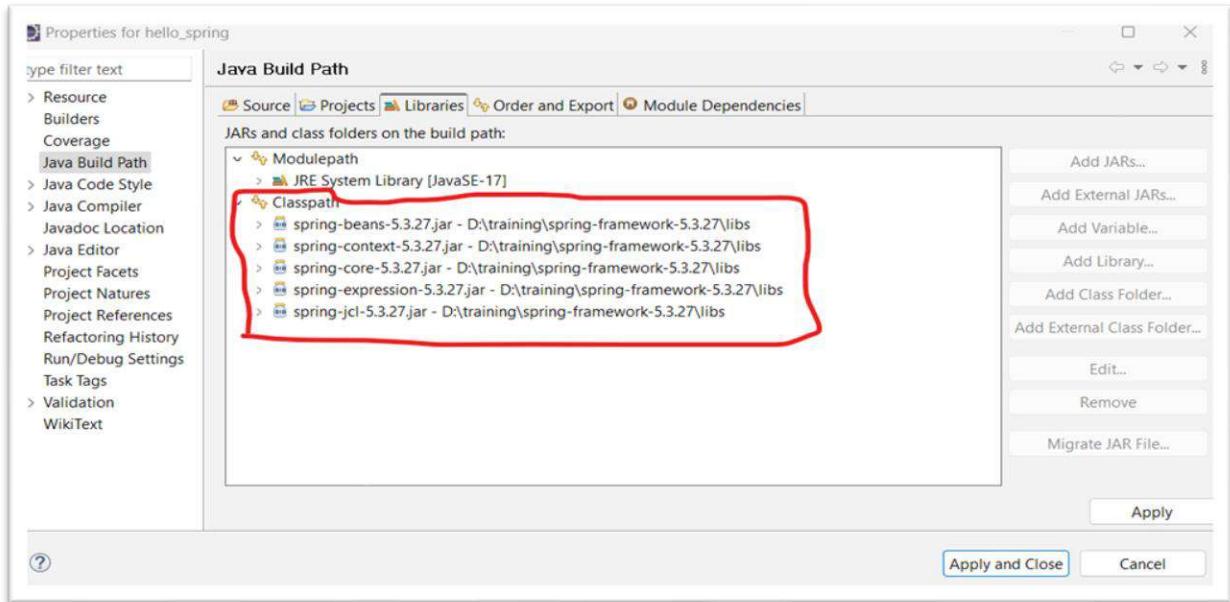
I have uploaded copy of all Spring JAR files uploaded in Google Drive. You can download from below link directly.

https://drive.google.com/file/d/1FnbtP3yqjTN5arlEGeoUHCrlJcdcBgM7/view?usp=drive_link

After Download completes, Please extract .zip file.

3. Now Please set build path to java project with Spring core jar files from lib folder in downloaded in step 2, which are shown in image.

Right Click on Project -> Build Path -> Configure Build Path -> Libraries -> ClassPath

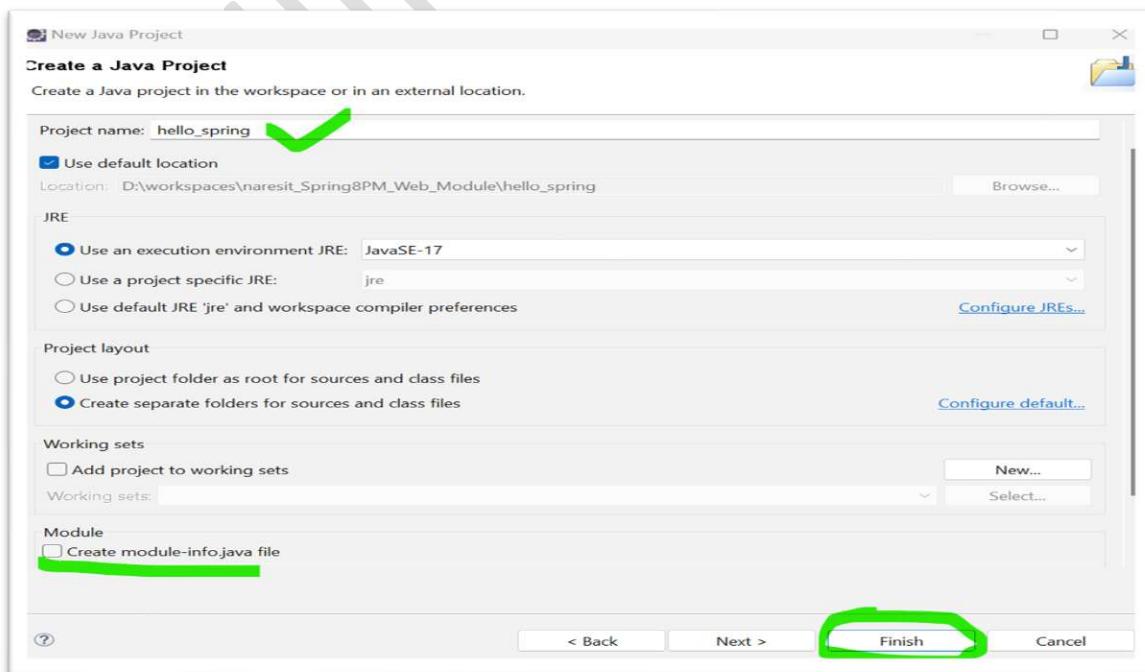


With This Our Java Project is Supporting Spring Core Module Functionalities. We can Continue with Spring Core Module Functionalities.

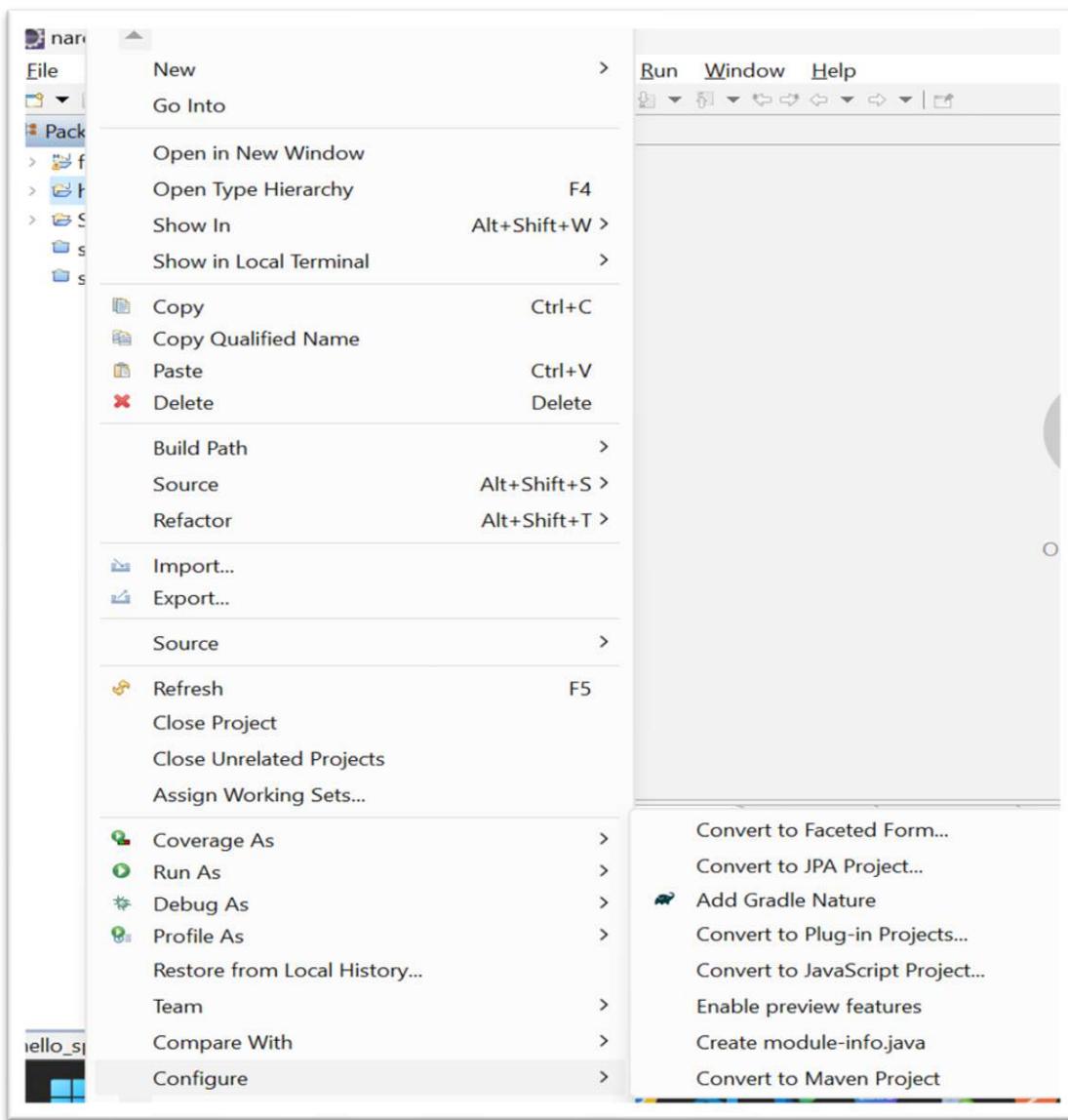
Creating Spring Core Project with Maven Configuration:

1. Create Java Project.

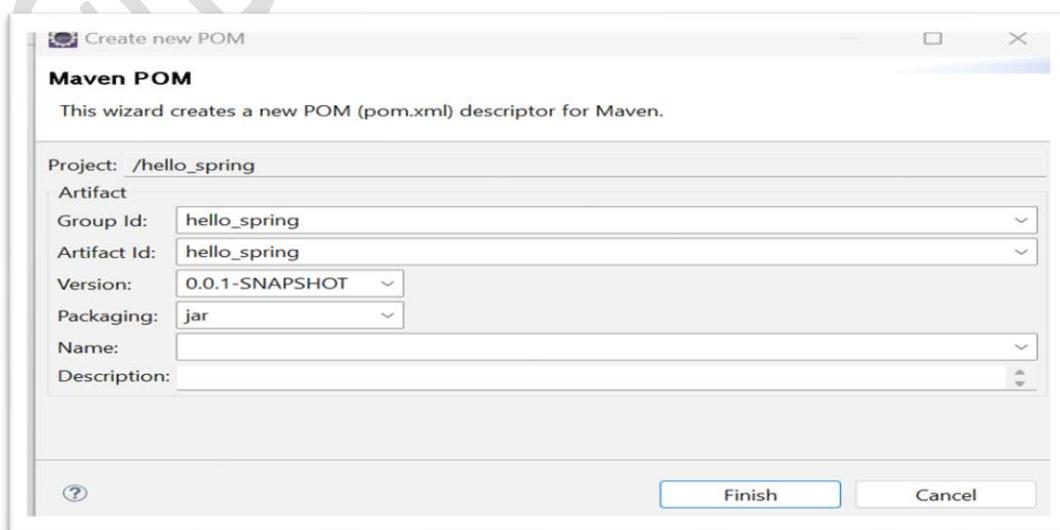
Open Eclipse -> File -> new -> Project -> Java Project -> Enter Project Name -> Un-Select Create Module-Info -> Click Finish.



2. Now Right Click On Project and Select Configure -> Convert to Maven Project.



Immediately It will show below details and click on Finish.



3. Now With Above Step, Java Project Supporting Maven functionalities. Created a default pom.xml as well. Project Structure shown as below.



Now Open pom.xml file, add Spring Core JAR Dependencies to project and save it.

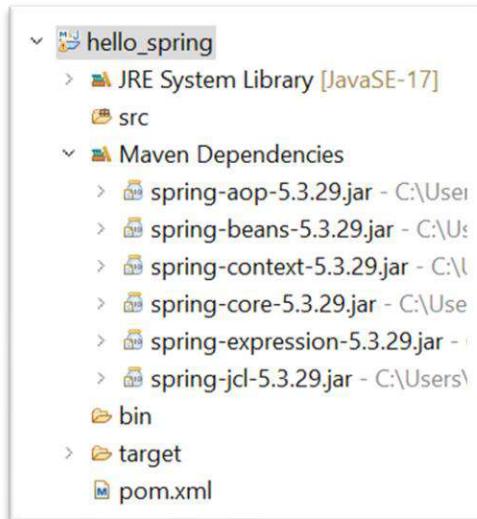
```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>hello_spring</groupId>
  <artifactId>hello_spring</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>5.3.29</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.3.29</version>
    </dependency>
  </dependencies>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <release>17</release>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

After adding Dependencies, Maven downloads all Spring Core Jar files with internal dependencies of jars at the same time configures those as part of Project automatically. As a Developer we no need to configure of jars in this approach. Now we can See Downloaded JAR files under Maven Dependencies Section as shown in below.



With This Our Java Project is Supporting Spring Core Module Functionalities. We can Continue with Spring Core Module Functionalities.

NOTE: Below Steps are now common across our Spring Core Project created by either Manual Jar files or Maven Configuration.

4. Now Create a java POJO Class in src inside package.

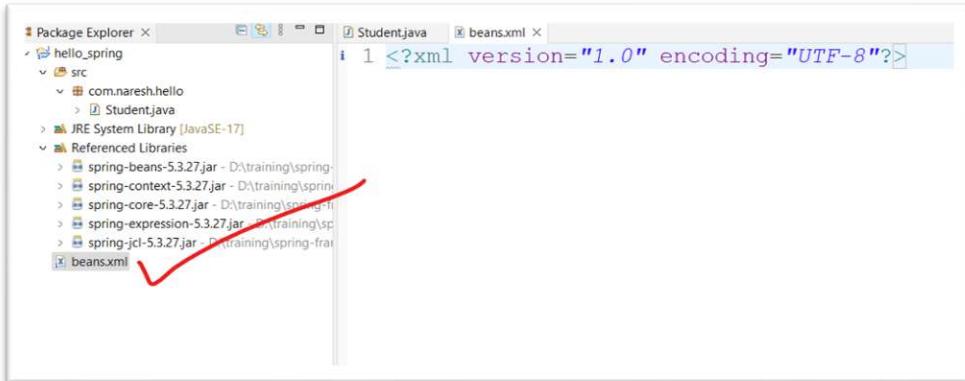
```
package com.naresh.hello;

public class Student {
    private String studnetName;

    public String getStudnetName() {
        return studnetName;
    }
    public void setStudnetName(String studnetName) {
        this.studnetName = studnetName;
    }
}
```

5. Now create a xml file with any name in side our project root folder:

Ex: **beans.xml**



6. Now Inside **beans.xml**, and paste below XML Shema content to configure all our bean classes.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Configure Our Bean classes Here -->
</beans>

```

We can get above content from below link as well.

<https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/xsd-configuration.html>

7. Now configure our POJO class **Student** in side **beans.xml** file.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="stu" class="com.naresh.hello.Student"> </bean>
</beans>

```

From above Configuration, Points to be Noted:

- Every class will be configured with **<bean>** tag, we can call it as Bean class.
- The **id** attribute is a string that identifies the individual bean name in Spring IOC Container i.e. similar to Object Name or Reference.
- The **class** attribute is fully qualified class name our class i.e. class name with package name.

8. Now create a main method class for testing.

Here we are getting the object of Student class from the Spring IOC container using the **getBean()** method of **BeanFactory**. Let's see the code

```
package com.naresh.hello;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.FileSystemXmlApplicationContext;

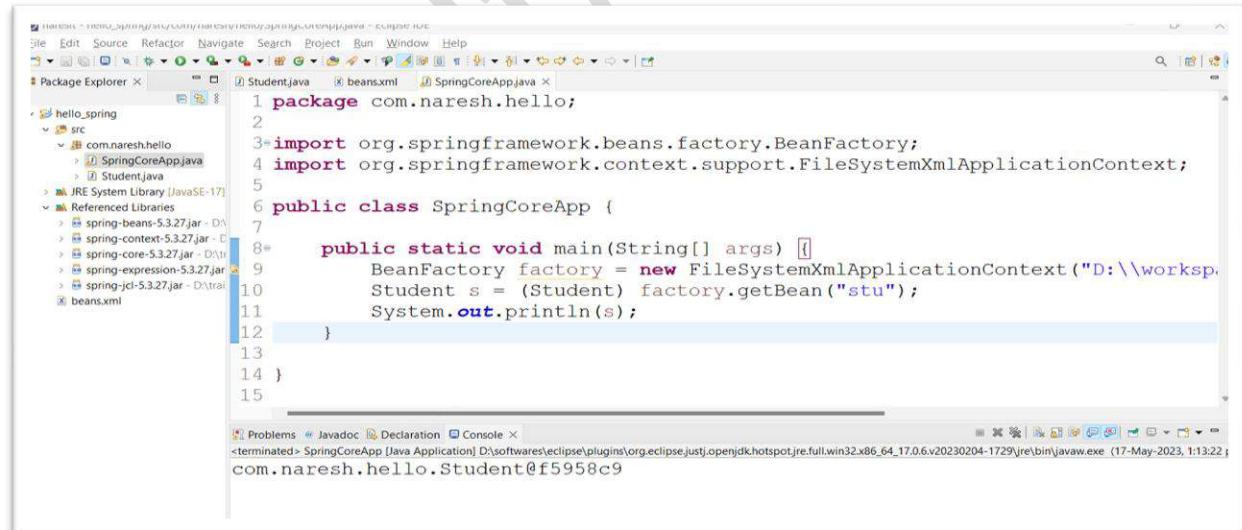
public class SpringCoreApp {
    public static void main(String[] args) {

        // BeanFactory Object is called as IOC Container.
        BeanFactory factory = new

        FileSystemXmlApplicationContext("D:\\\\workspaces\\\\naresit\\\\hello_spring\\\\beans.xml");

        Student s = (Student) factory.getBean("stu");
        System.out.println(s);
    }
}
```

9. Now Execute Your Program : Run as Java Application.



In above example Student Object Created by Spring IOC container and we got it by using **getBean()** method. If you observe, we are not written code for Student Object Creation i.e. using new operator.

- We can create multiple Bean Objects for same Bean class with multiple bean configurations in xml file.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

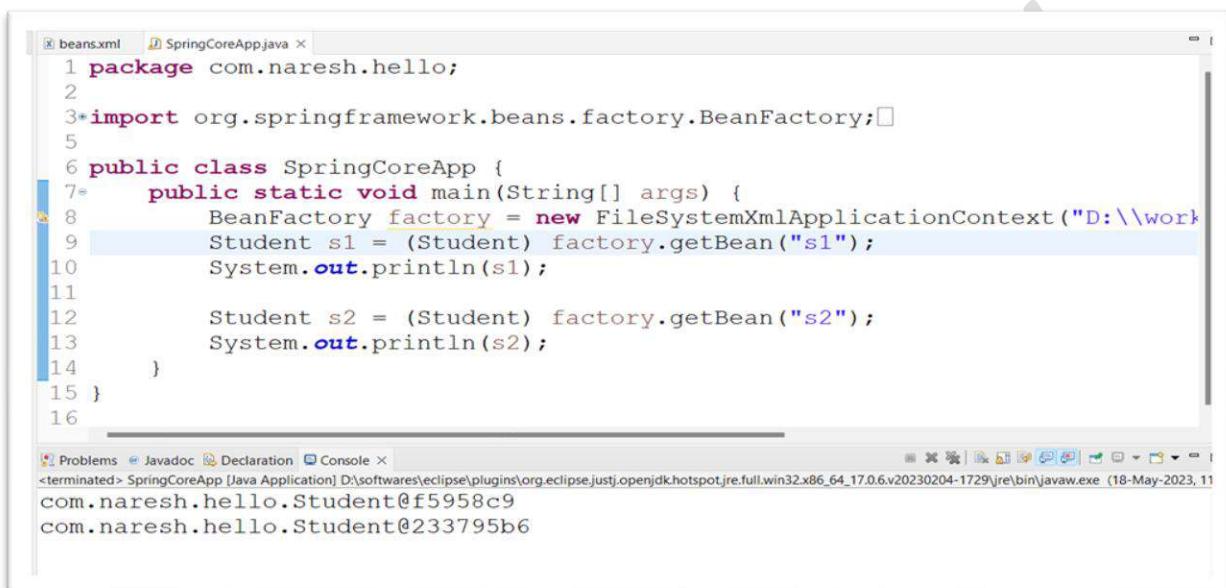
```

xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="s1" class="com.naresh.hello.Student"> </bean>
<bean id="s2" class="com.naresh.hello.Student"> </bean>
</beans>

```

➤ Now In Main Application class, Get Second Object.



The screenshot shows the Eclipse IDE interface. On the left, there are two tabs: 'beans.xml' and 'SpringCoreApp.java'. The code in 'SpringCoreApp.java' is as follows:

```

1 package com.naresh.hello;
2
3 import org.springframework.beans.factory.BeanFactory;
4
5 public class SpringCoreApp {
6     public static void main(String[] args) {
7         BeanFactory factory = new FileSystemXmlApplicationContext("D:\\work\\beans.xml");
8         Student s1 = (Student) factory.getBean("s1");
9         System.out.println(s1);
10
11         Student s2 = (Student) factory.getBean("s2");
12         System.out.println(s2);
13     }
14 }
15
16

```

The 'Console' tab at the bottom shows the execution output:

```

<terminated> SpringCoreApp [Java Application] D:\\softwares\\eclipse\\plugins\\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\\jre\\bin\\javaw.exe (18-May-2023, 11:54:23)
com.naresh.hello.Student@f5958c9
com.naresh.hello.Student@233795b6

```

So we can provide multiple configurations and create multiple Bean Objects for a class.

Bean Overview:

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML <bean/> definitions). Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier. However, if it requires more than one, the extra ones can be considered aliases. In XML-based configuration metadata, you use the id attribute, the name attribute, or both to specify bean identifiers. The id attribute lets you specify exactly one id.

Bean Naming Conventions:

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter and are camel-cased from there. Examples of such names include **accountManager**, **accountService**, **userDao**, **loginController**.

Instantiating Beans:

A bean definition is essentially a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the **class** attribute of the **<bean/>** element. This **class** attribute (which, internally, is a Class property on a BeanDefinition instance) is usually mandatory.

Types of Dependency Injection:

Dependency Injection (DI) is a design pattern that allows us to decouple the dependencies of a class from the class itself. This makes the class more loosely coupled and easier to test. In Spring, DI can be achieved through constructors, setters, or fields.

1. **Setter Injection**
2. **Constructor Injection**
3. **Filed Injection**

There are many benefits to using dependency injection in Spring. Some of the benefits include:

- **Loose coupling:** Dependency injection makes the classes in our application loosely coupled. This means that the classes are not tightly coupled to the specific implementations of their dependencies. This makes the classes more reusable and easier to test.
- **Increased testability:** Dependency injection makes the classes in our application more testable. This is because we can inject mock implementations of dependencies into the classes during testing. This allows us to test the classes in isolation, without having to worry about the dependencies.
- **Increased flexibility:** Dependency injection makes our applications more flexible. This is because we can change the implementations of dependencies without having to change the classes that depend on them. This makes it easier to change the underlying technologies in our applications.

Dependency injection is a powerful design pattern that can be used to improve the design and testability of our Spring applications. By using dependency injection, we can make our applications more loosely coupled, increase their testability, and improve their flexibility.

Setter Injection:

Setter injection is another way to inject dependencies in Spring. In this approach, we specify the dependencies in the class setter methods. The Spring container will then create an instance of the class and then call the setter methods to inject the dependencies.

The **<property>** sub element of **<bean>** is used for setter injection. Here we are going to inject

- primitive and String-based values
- Dependent object (contained object)
- Collection values etc.

Now Let's take example for setter injection.

1. Create a class.

```
package com.naresh.first.core;

public class Student {

    private String studentName;
    private String studentId;
    private String clgName;
    public String getClgName() {
        return clgName;
    }
    public void setClgName(String clgName) {
        this.clgName = clgName;
    }
    public String getStudentName() {
        return studentName;
    }
    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }
    public String getStudentId() {
        return studentId;
    }
    public void setStudentId(String studentId) {
        this.studentId = studentId;
    }
    public void printStudentDeatils() {
        System.out.println("This is Student class");
    }
    public double getAvgOfMArks() {
        return 456 / 6;
    }
}
```

2. Configure bean in beans xml file :

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="s1" class="com.naresh.first.core.Student">
        <property name="clgName" value="ABC College " />
        <property name="studentName" value="Dilip Singh " />
        <property name="studentId" value="100" />
    </bean>
</beans>

```

From above configuration, `<property>` tag referring to setter injection i.e. injecting value to a variable or property of Bean Student class.

`<property>` tag contains some attributes.

name: Name of the Property i.e. variable name of Bean class

value: Real/Actual value of Variable for injecting/storing

3. Now get the bean object from Spring Container and print properties values.

```

package com.naresh.first.core;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class SpringApp {
    public static void main(String[] args) {

        BeanFactory factory = new
        FileSystemXmlApplicationContext("D:\\\\workspaces\\\\nareshit\\\\spring_first\\\\beans.xml");
        // Requesting Spring Container for Student Object
        Student s1 = (Student) factory.getBean("s1");
        System.out.println(s1.getStudentId());
        System.out.println(s1.getStudentName());
        System.out.println(s1.getClgName());
        s1.printStudentDeatils();
        System.out.println(s1.getAvgOfMArks());
    }
}

```

Output:

```
100
Dilip Singh
ABC College
This is Student class
76.0
```

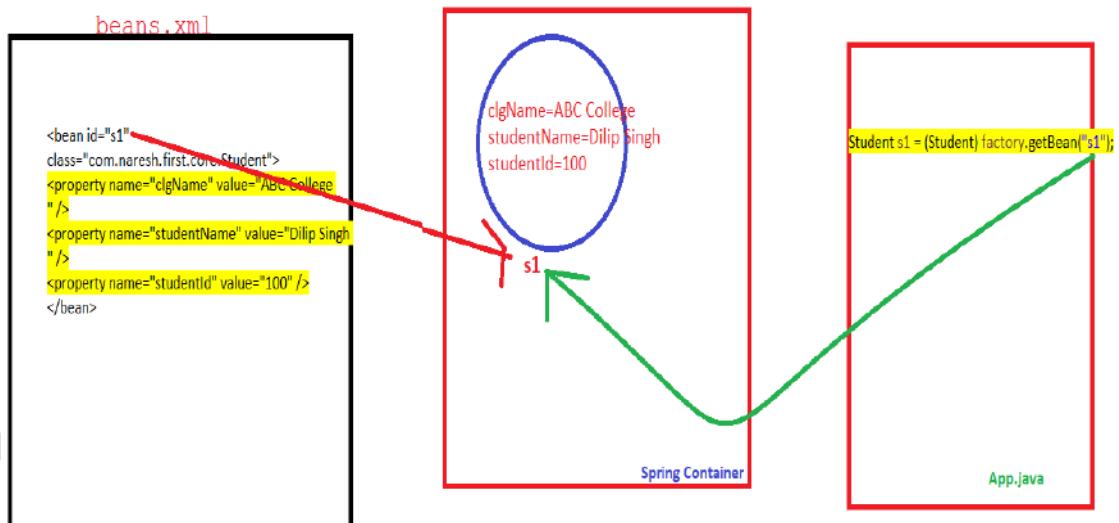
Internal Workflow/Execution of Above Program.

1. From the Below Line execution, Spring will create Spring IOC container and Loads our beans xml file in JVM memory and Creates Bean Objects inside Spring Container.

```
BeanFactory factory = new FileSystemXmlApplicationContext("D:\\workspaces\\naresit\\spring_first\\beans.xml");
```

2. Now from below line, we are getting bean object of Student class configured with bean id : s1

```
Student s1 = (Student) factory.getBean("s1");
```



3. Now we can use s1 object and call our method as usual.

Injecting primitive and String Data properties:

Now we are injecting/configuring primitive and String data type properties into Spring Bean Object.

- Define a class, with different primitive datatypes and String properties.

```
package com.naresh.hello;
```

```

public class Student {

    private String stuName;
    private int studId;
    private double avgOfMarks;
    private short passedOutYear;
    private boolean isSelected;

    public String getStuName() {
        return stuName;
    }

    public void setStuName(String stuName) {
        this.stuName = stuName;
    }

    public int getStudId() {
        return studId;
    }

    public void setStudId(int studId) {
        this.studId = studId;
    }

    public double getAvgOfMarks() {
        return avgOfMarks;
    }

    public void setAvgOfMarks(double avgOfMarks) {
        this.avgOfMarks = avgOfMarks;
    }

    public short getPassedOutYear() {
        return passedOutYear;
    }

    public void setPassedOutYear(short passedOutYear) {
        this.passedOutYear = passedOutYear;
    }

    public boolean isSelected() {
        return isSelected;
    }

    public void setSelected(boolean isSelected) {
        this.isSelected = isSelected;
    }
}

```

- Now configure above properties in spring beans xml file.

```

<bean id="studentOne" class="com.naresh.hello.Student">
    <property name="stuName" value="Dilip"></property>
    <property name="studId" value="101"></property>
    <property name="avgOfMarks" value="99.88"></property>

```

```

<property name="passedOutYear" value="2022"></property>
<property name="isSelected" value="true"></property>
</bean>

```

- For primitive and String data type properties of bean class, we can use both **name** and **value** attributes.
- Now let's test values injected or not from above bean configuration.

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

```

```

public class SpringCoreApp {
    public static void main(String[] args) {
        ApplicationContext context = new
        FileSystemXmlApplicationContext("D:\\workspaces\\naresit\\spring_notes\\beans.xml");
        Student s1 = (Student) context.getBean("studentOne"); // get it from
        container
        System.out.println(s1.getStudId());
        System.out.println(s1.getStuName());
        System.out.println(s1.getPassedOutYear());
        System.out.println(s1.getAvgOfMarks());
        System.out.println(s1.isSelected());
    }
}

```

Output:

```

101
Dilip
2022
99.88
True

```

Injecting Collection Data Types properties:

Now we are injecting/configuring Collection Data Types like List, Set and Map properties into Spring Bean Object.

- For **List** data type property, Spring Provided **<list>** tag, sub tag of **<property>**.

```

<list>
    <value> ... </value>
    <value>... </value>
    <value> .. </value>
    .....
</list>

```

- For **Set** data type property, Spring Provided **<list>** tag, sub tag of **<property>**.

```

<set>

```

```

<value> ... </value>
<value>... </value>
<value> .. </value>
.....
</set>

```

- For **Map** data type property, Spring Provided **<list>** tag, sub tag of **<property>**.

```

<map>
<entry key="..." value="..." />
<entry key="..." value="..." />
<entry key="..." value="..." />
.....
</map>

```

- Created Bean class with List, Set and Map properties.

```

package com.naresh.hello;

import java.util.List;
import java.util.Map;
import java.util.Set;

public class Student {

    private String stuName;
    private int studId;
    private double avgOfMarks;
    private short passedOutYear;
    private boolean isSelected;
    private List<String> emails;
    private Set<String> mobileNumbers;
    private Map<String, String> subMarks;

    public List<String> getEmails() {
        return emails;
    }

    public void setEmails(List<String> emails) {
        this.emails = emails;
    }

    public Set<String> getMobileNumbers() {
        return mobileNumbers;
    }
}

```

```
public void setMobileNumbers(Set<String> mobileNumbers) {  
    this.mobileNumbers = mobileNumbers;  
}  
  
public Map<String, String> getSubMarks() {  
    return subMarks;  
}  
  
public void setSubMarks(Map<String, String> subMarks) {  
    this.subMarks = subMarks;  
}  
  
public String getStuName() {  
    return stuName;  
}  
  
public void setStuName(String stuName) {  
    this.stuName = stuName;  
}  
  
public int getStudId() {  
    return studId;  
}  
  
public void setStudId(int studId) {  
    this.studId = studId;  
}  
  
public double getAvgOfMarks() {  
    return avgOfMarks;  
}  
  
public void setAvgOfMarks(double avgOfMarks) {  
    this.avgOfMarks = avgOfMarks;  
}  
  
public short getPassedOutYear() {  
    return passedOutYear;  
}  
  
public void setPassedOutYear(short passedOutYear) {  
    this.passedOutYear = passedOutYear;  
}  
  
public boolean isSelected() {  
    return isSelected;
```

```

    }

    public void setSelected(boolean isSelected) {
        this.isSelected = isSelected;
    }
}

```

Now configure in beans xml file.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="studentOne" class="com.naresh.hello.Student">
        <property name="stuName" value="Dilip"></property>
        <property name="studId" value="101"></property>
        <property name="avgOfMarks" value="99.88"></property>
        <property name="passedOutYear" value="2022"></property>
        <property name="isSelected" value="true"></property>
        <property name="emails">
            <list>
                <value>dilip@gmail.com</value>
                <value>laxmi@gmail.com</value>
                <value>dilip@gmail.com</value>
            </list>
        </property>
        <property name="mobileNumbers">
            <set>
                <value>8826111377</value>
                <value>8826111377</value>
                <value>+1234567890</value>
            </set>
        </property>
        <property name="subMarks">
            <map>
                <entry key="maths" value="88" />
                <entry key="science" value="66" />
                <entry key="english" value="44" />
            </map>
        </property>
    </bean>
</beans>

```

- Now let's test values injected or not from above bean configuration.

```
package com.naresh.hello;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

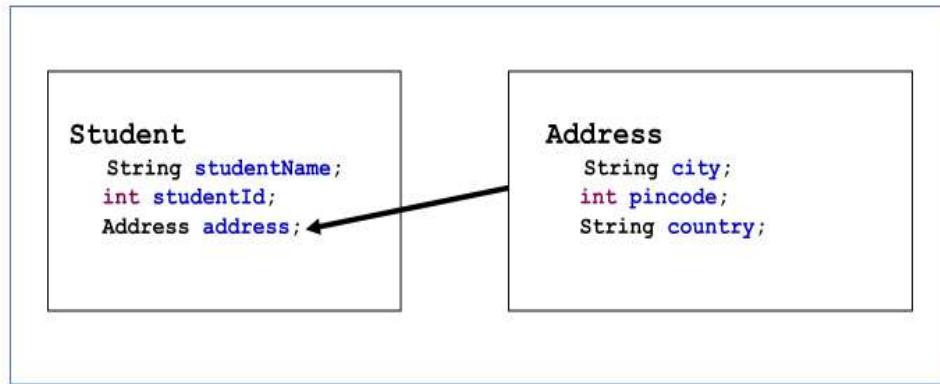
public class SpringCoreApp {
    public static void main(String[] args) {
        ApplicationContext context = new
        FileSystemXmlApplicationContext("D:\\workspaces\\naresit\\spring_notes\\beans.xml");
        Student s1 = (Student) context.getBean("studentOne"); // get it from
        container
        System.out.println(s1.getStudId());
        System.out.println(s1.getStuName());
        System.out.println(s1.getPassedOutYear());
        System.out.println(s1.getAvgOfMarks());
        System.out.println(s1.isSelected());
        System.out.println(s1.getEmails()); // List Values
        System.out.println(s1.getMobileNumbers()); // Set Values
        System.out.println(s1.getSubMarks()); //Map values
    }
}
```

Output:

```
101
Dilip
2022
99.88
true
[dilip@gmail.com, laxmi@gmail.com, dilip@gmail.com]
[8826111377, +1234567890]
{maths=88, science=66, english=44}
```

Injecting Dependency's of Other Bean Objects:

Now we are injecting/configuring other Bean Objects into another Spring Bean Object.



- Now Create a class : **Address.java**

```

package com.naresh.training.spring.core;

public class Address {

    private String city;
    private int pincode;
    private String country;

    public Address() {
        System.out.println("Address instance/constructed ");
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public int getPincode() {
        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
}
  
```

- Now Create another class with Address Type Property: **Student.java**

```
package com.naresh.training.spring.core;

public class Student {

    private String studentName;
    private int studentId;
    private Address address;

    public Student() {
        System.out.println("Student Constructor executed.");
    }
    public String getStudentName() {
        return studentName;
    }
    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }
    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
}
```

- Now Configure Address and Student Bean classes. Here, Address Bean Object is dependency of Student object i.e. Address should be referred inside Student. To achieve this collaboration, Spring provided **ref** element/attribute.

ref attribute / <ref> element:

The **ref** element is the element inside a **<property>** or **<constructor-arg>** element. Here, you set the value of the specified property of a bean to be a referenced to another bean (a collaborator) managed by the container. Sometimes we can use **ref** attribute as part of **<property>** or **<constructor-arg>**. We will provide bean Id for **ref** element which should be

injected into target Object. Please refer below, how to inject Bean Objects via **ref** element or attribute.

- Configure Bean classes now inside **beans.xml** file.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
  <bean
    class="com.naresh.training.spring.core.Address"
    id="universityAddress"
    >
    <property name="city" value="Bangalore"></property>
    <property name="country" value="India"></property>
    <property name="pincode" value="400066"></property>
  </bean>
  <!-- Student Bean Objects -->
  <bean id="student1" class="com.naresh.training.spring.core.Student">
    <property name="studentName" value="Dilip Singh"></property>
    <property name="studentId" value="100"></property>
    <property name="address" ref="universityAddress"></property>
  </bean>
  <bean id="student2" class="com.naresh.training.spring.core.Student">
    <property name="studentName" value="Naresh"></property>
    <property name="studentId" value="101"></property>
    <property name="address">
      <ref bean="universityAddress"/>
    </property>
  </bean>
</beans>

```

- Now let's test values and references injected or not from above bean configuration.

```

package com.naresh.training.spring.core;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class SpringSetterInjectionDemo {

  public static void main(String[] args) {

    BeanFactory factory = new FileSystemXmlApplicationContext(
      "D:\\Spring\\spring-
    injection\\beans.xml");

    System.out.println("***** Student1 Data *****");
  }
}

```

```

Student stu1 = (Student) factory.getBean("student1");
System.out.println(stu1.getStudentId());
System.out.println(stu1.getStudentName());
Address stu1Address = stu1.getAddress();
System.out.println(stu1Address.getCity());
System.out.println(stu1Address.getCountry());
System.out.println(stu1Address.getPincode());

System.out.println("***** Student2 Data *****");
Student stu2 = (Student) factory.getBean("student2");
System.out.println(stu2.getStudentId());
System.out.println(stu2.getStudentName());
System.out.println(stu2.getAddress().getCity());
System.out.println(stu2.getAddress().getCountry());
System.out.println(stu2.getAddress().getPincode());
}

}

```

Output:

```

Address instance/constructed
Student Contructor executed.
Student Contructor executed.
***** Student1 Data *****
100
Dilip Singh
Bangalore
India
400066
***** Student2 Data *****
101
Naresh
Bangalore
India
400066

```

From above output, same **universityAddress** bean Object is injected by Spring Container internally inside both **student1** and **student2** Bean Objects.

Constructor Injection:

Constructor injection is a form of dependency injection where dependencies are provided to a class through its constructor. It is a way to ensure that all required dependencies are supplied when creating an object. In constructor injection, the class that requires dependencies has one or more parameters in its constructor that represent the

dependencies. When an instance of the class is created, the dependencies are passed as arguments to the constructor. Constructor injection is often considered a best practice in Spring because it helps ensure that the dependencies required for an object to function are provided at the time of its creation. This can lead to more maintainable and testable code.

In XML configuration, Spring provided a child tag `<constructor-arg>` of `<bean>` to achieve or configure Constructor Injection

Example: Defining Bean Class With Primitive and String Data type.

```
package com.naresh.spring.di.ci;

public class Product {

    private int productId;
    private String productName;
    private double price;

    // All Params Constructor
    public Product(int productId, String productName, double price) {
        super();
        System.out.println("Product All Param's Constructor Executed");
        this.productId = productId;
        this.productName = productName;
        this.price = price;
    }
    public int getProductId() {
        return productId;
    }
    public String getProductName() {
        return productName;
    }
    public double getPrice() {
        return price;
    }
    public void setProductId(int productId) {
        System.out.println("setProductId is called");
        this.productId = productId;
    }
    public void setProductName(String productName) {
        System.out.println("setProductName is called");
        this.productName = productName;
    }
    public void setPrice(double price) {
        System.out.println("setPrice is called");
        this.price = price;
    }
}
```

}

- Now Configure Bean Data With Constructor Injection.

```
<bean id="iphone" class="com.naresh.spring.di.ci.Product">
    <constructor-arg value="1000"></constructor-arg>
    <constructor-arg value="Apple Iphone 15 "></constructor-arg>
    <constructor-arg value="150000.00"></constructor-arg>
</bean>
```

- From the above Bean Configuration, Spring Container Internally passes values to constructor of our class while creating Bean Object.

Testing:

```
package com.naresh.spring.di.ci;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class SpringConstructorInjectionDemo {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
                "D:\\spring-
beans.xml");
        Product product= (Product) context.getBean("iphone");
        System.out.println(product.getProductId());
        System.out.println(product.getProductName());
        System.out.println(product.getPrice());
    }
}
```

Output:

```
Product All Param's Constructor Executed
1000
Apple Iphone 15
150000.0
```

- Finally, values are injected into properties of bean Object by executing constructor.

NOTE: When we are defining `<constructor-arg>` and values in Beans XML Configuration, we should follow same order w.r.to Constructor Parameters.

```
<bean id="iphone" class="com.naresh.spring.di.ci.Product">
    <constructor-arg value="1000"></constructor-arg>
    <constructor-arg value="Apple Iphone 15 "></constructor-arg>
    <constructor-arg value="150000.00"></constructor-arg>
</bean>
```

```
public Product(int productId, String productName, double price) {
    // Body
}
```

- If we are not following same order in vice versa, then we will get below Exception while Creating Bean Object.

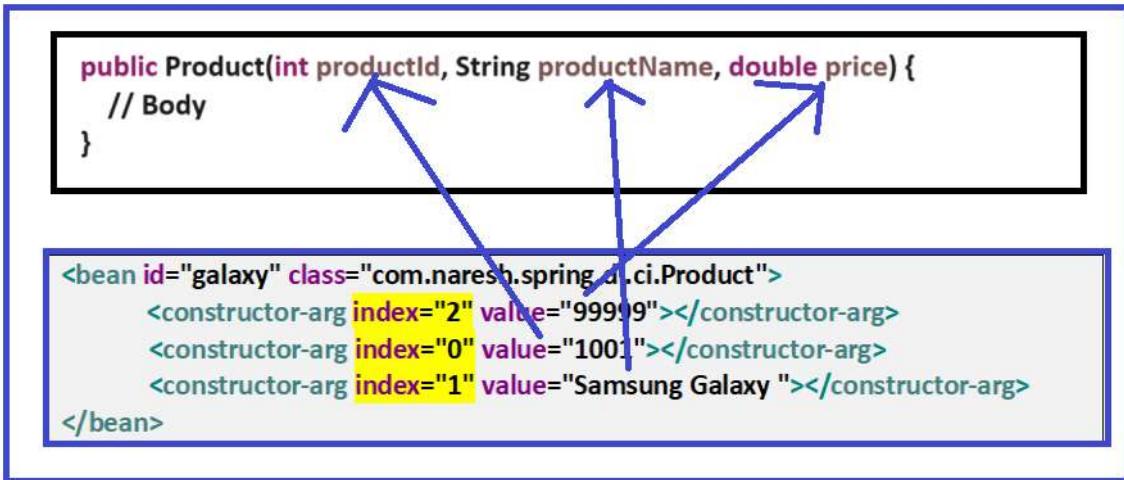
Exception in thread "main"

org.springframework.beans.factory.UnsatisfiedDependencyException:

Question: In any case, if we don't want to follow same order in beans configuration, do we have any alternative solution?

Answer: Yes, Spring provided an attribute **index** as part of **<constructor-arg>** tag i.e. we should provide index value for every property w.r.to Constructor Parameters Order. Here, Index starts from **0** always.

```
<bean id="galaxy" class="com.naresh.spring.di.ci.Product">
    <constructor-arg index="2" value="99999"></constructor-arg>
    <constructor-arg index="0" value="1001"></constructor-arg>
    <constructor-arg index="1" value="Samsung Galaxy "></constructor-arg>
</bean>
```



Question: Do we need to configure all values for all constructor parameters in Spring Bean Configuration?

Answer: Yes, We should configure every constructor parameter value inside bean configuration in Spring i.e. From above example, Constructor Defined with 3 parameters in Product class, so we should configure 3 values of <constructor-arg>.

If we are not configured same number of values respectively Spring will create an Exception while creating Bean Object for that Bean Configuration.

Exception in thread "main"

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'laptop' defined in file

- we can create many constructors in a Spring Bean class, and we should configure values respectively for every bean Object with Constructor Injection.

Constructor Injection: Example with Collection Data Type and Another Object Reference.

- Define AccountDetails Bean class.

```

package com.naresh.hello;

import java.util.Set;

public class AccountDetails {

    private String name;
    private double balance;
    private Set<String> mobiles;
    private Address customerAddress;
}

```

```

public AccountDetails(String name, double balance, Set<String> mobiles,
                      Address customerAddress) {
    super();
    this.name = name;
    this.balance = balance;
    this.mobiles = mobiles;
    this.customerAddress = customerAddress;
}
public AccountDetails() {

}
public Address getCustomerAddress() {
    return customerAddress;
}
public void setCustomerAddress(Address customerAddress) {
    this.customerAddress = customerAddress;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public double getBalance() {
    return balance;
}
public void setBalance(double balance) {
    this.balance = balance;
}
public Set<String> getMobiles() {
    return mobiles;
}
public void setMobiles(Set<String> mobiles) {
    this.mobiles = mobiles;
}
}

```

- Define Dependency Class Address.

```

package com.naresh.hello;

public class Address {

    private int flatNo;
    private String houseName;
    private long mobile;
}

```

```

public int getFlatNo() {
    return flatNo;
}
public void setFlatNo(int flatNo) {
    this.flatNo = flatNo;
}
public String getHouseName() {
    return houseName;
}
public void setHouseName(String houseName) {
    this.houseName = houseName;
}
public long getMobile() {
    return mobile;
}
public void setMobile(long mobile) {
    this.mobile = mobile;
}
}

```

- Define Bean Configuration

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="addr" class="com.naresh.hello.Address">
        <property name="flatNo" value="333"/>
        <property name="houseName" value="Lotus Homes"/>
        <property name="mobile" value="91822222"/>
    </bean>
    <bean id="accountDeatils"
          class="com.naresh.hello.AccountDetails">
        <constructor-arg name="name" value="Dilip" />
        <constructor-arg name="balance" value="500.00" />
        <constructor-arg name="mobiles">
            <set>
                <value>8826111377</value>
                <value>8826111377</value>
                <value>+91-88888888</value>
                <value>+232388888888</value>
            </set>
        </constructor-arg>
        <constructor-arg name="customerAddress" ref="addr" />
    </bean>

```

```
</bean>
</beans>
```

- Now Test Constructor Injection Beans and Configuration.

```
package com.naresh.hello;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class SpringCoreApp {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
            "D:\\workspaces\\naresit\\spring_notes\\beans.xml");

        AccountDetails details = (AccountDetails) context.getBean("accountDetails");

        System.out.println(details.getName());
        System.out.println(details.getBalance());
        System.out.println(details.getMobiles());
        System.out.println(details.getCustomerAddress().getFlatNo());
        System.out.println(details.getCustomerAddress().getHouseName());
    }
}
```

Output:

```
Dilip
500.0
[8826111377, +91-88888888, +232388888888]
333
Lotus Homes
```

Differences Between Setter and Constructor Injection.

Setter injection and constructor injection are two common approaches for implementing dependency injection. Here are the key differences between them:

- 1. Dependency Resolution:** In setter injection, dependencies are resolved and injected into the target object using setter methods. In contrast, constructor injection resolves dependencies by passing them as arguments to the constructor.
- 2. Timing of Injection:** Setter injection can be performed after the object is created, allowing for the possibility of injecting dependencies at a later stage. Constructor injection, on the other hand, requires all dependencies to be provided at the time of object creation.

3. Flexibility: Setter injection provides more flexibility because dependencies can be changed or modified after the object is instantiated. With constructor injection, dependencies are typically immutable once the object is created.

4. Required Dependencies: In setter injection, dependencies may be optional, as they can be set to null if not provided. Constructor injection requires all dependencies to be provided during object creation, ensuring that the object is in a valid state from the beginning.

5. Readability and Discoverability: Constructor injection makes dependencies more explicit, as they are declared as parameters in the constructor. This enhances the readability and discoverability of the dependencies required by a class. Setter injection may result in a less obvious indication of required dependencies, as they are set through individual setter methods.

6. Testability: Constructor injection is generally favored for unit testing because it allows for easy mocking or substitution of dependencies. By providing dependencies through the constructor, testing frameworks can easily inject mocks or stubs when creating objects for testing. Setter injection can also be used for testing, but it may require additional setup or manipulation of the object's state.

The choice between setter injection and constructor injection depends on the specific requirements and design considerations of your application. In general, constructor injection is recommended when dependencies are mandatory and should be set once during object creation, while setter injection provides more flexibility and optional dependencies can be set or changed after object instantiation.

Bean Wiring in Spring:

Bean wiring, also known as bean configuration or bean wiring configuration, is the process of defining the relationships and dependencies between beans in a container or application context. In bean wiring, you specify how beans are connected to each other, how dependencies are injected, and how the container should create and manage the beans. This wiring process is typically done through configuration files or annotations.

```

package com.naresh.hello;

public class AreaDeatils {

    private String street;
    private String pincode;

    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
}

```

```

public String getPincode() {
    return pincode;
}
public void setPincode(String pincode) {
    this.pincode = pincode;
}
}

```

- Create Another Bean : **Address**

```

package com.naresh.hello;

public class Address {

    private int flatNo;
    private String houseName;
    private long mobile;
    private AreaDeatils area; // Dependency Of Another
Class

    public AreaDeatils getArea() {
        return area;
    }
    public void setArea(AreaDeatils area) {
        this.area = area;
    }
    public int getFlatNo() {
        return flatNo;
    }
    public void setFlatNo(int flatNo) {
        this.flatNo = flatNo;
    }
    public String getHouseName() {
        return houseName;
    }
    public void setHouseName(String houseName) {
        this.houseName = houseName;
    }
    public long getMobile() {
        return mobile;
    }
    public void setMobile(long mobile) {
        this.mobile = mobile;
    }
}

```

- Create Another Bean : **AccountDetails**

```

package com.naresh.hello;

```

```
import java.util.Set;

public class AccountDetails {

    private String name;
    private double balance;
    private Set<String> mobiles;
    private Address customerAddress;
    public AccountDetails(String name, double balance, Set<String> mobiles,
                         Address customerAddress) {
        super();
        this.name = name;
        this.balance = balance;
        this.mobiles = mobiles;
        this.customerAddress = customerAddress;
    }

    public AccountDetails() {
    }

    public Address getCustomerAddress() {
        return customerAddress;
    }

    public void setCustomerAddress(Address customerAddress) {
        this.customerAddress = customerAddress;
    }

    public String getName() {
        return name;
    }

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

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public Set<String> getMobiles() {
        return mobiles;
    }

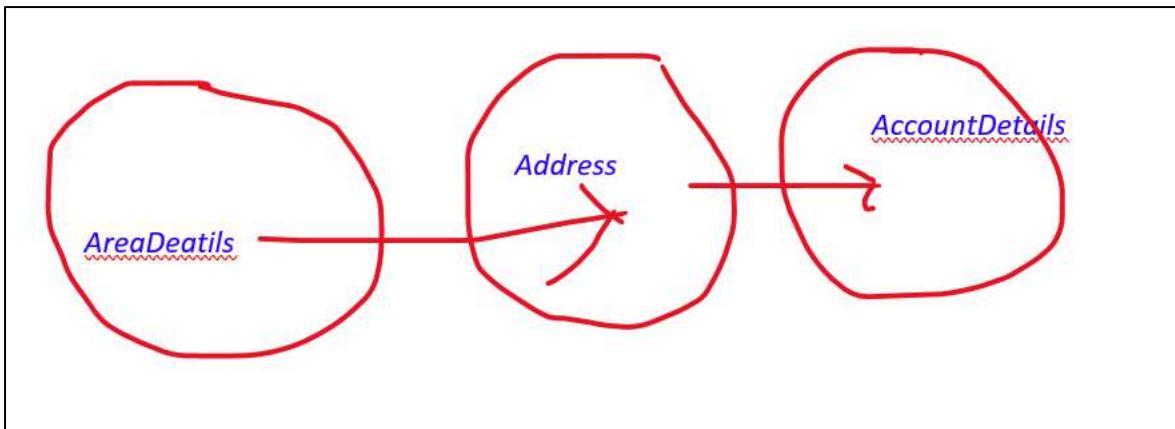
    public void setMobiles(Set<String> mobiles) {
        this.mobiles = mobiles;
    }
}
```

- Beans Configuration in spring xml file. With “ref” attribute we are configuring bean object each other internally.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="areaDetails" class="com.naresh.hello.AreaDeatils">
        <property name="street" value="Naresh It road"></property>
        <property name="pincode" value="323232"></property>
    </bean>
    <bean id="addr" class="com.naresh.hello.Address">
        <property name="flatNo" value="333"></property>
        <property name="houseName" value="Lotus Homes"></property>
        <property name="mobile" value="9182222222"></property>
        <property name="area" ref="areaDetails"></property>
    </bean>
    <bean id="accountDeatils" class="com.naresh.hello.AccountDetails">
        <constructor-arg name="name" value="Dilip" />
        <constructor-arg name="balance" value="500.00" />
        <constructor-arg name="customerAddress" ref="addr" />
        <constructor-arg name="mobiles">
            <set>
                <value>8826111377</value>
                <value>8826111377</value>
                <value>+91-8888888888</value>
                <value>+23238888888888</value>
            </set>
        </constructor-arg>
    </bean>
</beans>
```



Testing of Bean Configuration:

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class SpringCoreApp {
    public static void main(String[] args) {

        ApplicationContext context = new FileSystemXmlApplicationContext(
                "D:\\workspaces\\naresit\\spring_notes\\beans.xml");

        AccountDetails details
                = (AccountDetails) context.getBean("accountDeatils");

        System.out.println(details.getName());
        System.out.println(details.getBalance());
        System.out.println(details.getMobiles());
        //Get Address Instance
        System.out.println(details.getCustomerAddress().getFlatNo());
        //Get Area Instance
        System.out.println(details.getCustomerAddress().getArea().getPincode());
    }
}
    
```

Output:

```

Dilip
500.0
[8826111377, +91-88888888, +232388888888]
333
323232
    
```

AutoWiring in Spring:

Auto wiring feature of spring framework enables you to inject the objects dependency implicitly. It internally uses setter or constructor injection. In Spring framework, the “**autowire**” attribute is used in XML **<bean>** configuration files to enable automatic dependency injection. It allows Spring to automatically wire dependencies between beans without explicitly specifying them in the XML file.

To use autowiring in an XML bean configuration file, you need to define the “**autowire**” attribute for a bean definition. The “**autowire**” attribute accepts different values to determine how autowiring should be performed. There are many autowiring modes.

1. **no** : This is the default value. It means no autowiring will be performed, and you need to explicitly specify dependencies using the appropriate XML configuration using property or constructor tags.
2. **byName** : The byName mode injects the object dependency according to name of the bean i.e. Bean ID. In such case, property name of class and bean ID must be same. It internally calls setter method. If a match is found, the dependency will be injected.
3. **byType**: The byType mode injects the object dependency according to type i.e. Data Type of Property. So property/variable name and bean name can be different in this case. It internally calls setter method. If a match is found, the dependency will be injected. If multiple beans are found, an exception will be thrown.
4. **constructor**: The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

Here's an examples of using the “**autowire**” attribute in an XML bean configuration file.

autowire=no:

For example, Define Product Class.

```
public class Product {

    private String productId;
    private String productName;

    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
}
```

```

public String getProductName() {
    return productName;
}
public void setProductName(String productName) {
    this.productName = productName;
}
}

```

- Now Define Class **Order** which is having dependency of **Product** Object i.e. **Product** bean object should be injected to **Order**.

```

Package com.flipkart.orders;

import com.flipkart.product.Product;

public class Order {

    private String orderId;
    private double orderValue;
    private Product product;

    public Order() {
        System.out.println("Order Object Created by IOC");
    }
    public Order(String orderId, double orderValue, Product product) {
        super();
        this.orderId = orderId;
        this.orderValue = orderValue;
        this.product = product;
    }
    public String getOrderId() {
        return orderId;
    }
    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }
    public double getOrderValue() {
        return orderValue;
    }
    public void setOrderValue(double orderValue) {
        this.orderValue = orderValue;
    }
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
}

```

```
    }
}
```

- Now let's configure both Product and Order in side beans xml file.

```
<beans>
    <bean id="product" class="com.flipkart.product.Product">
        <property name="productId" value="101"></property>
        <property name="productName" value="Lenevo Laptop"></property>
    </bean>
    <bean id="order" class="com.flipkart.orders.Order" autowire="no">
        <property name="orderId" value="order1234"></property>
        <property name="orderValue" value="33000.00"></property>
    </bean>
</beans>
```

- Now Try to request Object of Order from IOC Container.

```
package com.flipkart.main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
import com.flipkart.orders.Order;
import com.flipkart.orders.OrdersManagement;
import com.flipkart.product.Product;

public class AutowiringDemo {
    public static void main(String[] args) {

        // IOC Container
        ApplicationContext context = new FileSystemXmlApplicationContext(
                "D:\\workspaces\\narexit\\bean-wiring\\beans.xml");

        Order order = (Order) context.getBean("order");
        // Product Object: Getting product Id
        System.out.println(order.getProduct().getProductId());
    }
}
```

- Now we got an exception, as shown below.

```
Product Object Created by IOC
Order Object Created by IOC
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"com.flipkart.product.Product.getProductId()" because the return value of
"com.flipkart.orders.Order.getProduct()" is null at
com.flipkart.main.AutowiringDemo.main(AutowiringDemo.java:22)
```

Means, Product and Order Object created by Spring but not injected product bean object automatically in side Order Object by IOC Container with setter injection internally. Because we used **autowire** mode as **no**. **By Default autowire value is "no" i.e.** Even if we are not given autowire attribute internally Spring Considers it as **autowire="no"**.

autowire="byName":

Now configure **autowire="byName"** in side beans xml file for order Bean configuration, because internally Product bean object should be injected to Order Bean Object. In this autowire mode, We are expecting dependency injection of objects by Spring instead of we are writing bean wiring with either using **<property>** and **<constructor-arg>** tags by using **ref** attribute. Means, eliminating logic of reference configurations.

As per **autowire="byName"**, Spring internally checks for a dependency bean objects which is matched to property names of Object. As per our example, Product is dependency for Order class.

Product class Bean ID = Property Name of Order class

```
<bean id="product" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

```
public class Order

    private String orderId;
    private double orderValue;
    private Product product;

    // setters , getters, constructors
}
```

- Internally Spring comparing as shown in above and injected product bean object to Order object.
- **Beans Configuration:**

```
<beans>
    <bean id="product" class="com.flipkart.product.Product">
        <property name="productId" value="101"></property>
        <property name="productName" value="Lenevo Laptop"></property>
    </bean>
    <bean id="order" class="com.flipkart.orders.Order" autowire="byName">
        <property name="orderId" value="order1234"></property>
        <property name="orderValue" value="33000.00"></property>
    </bean>
</beans>
```

- **Now test our application.**

```

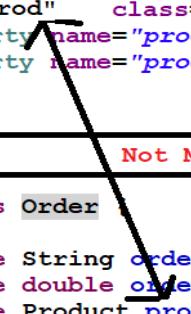
1 package com.flipkart.orders;
2
3 public class OrdersManagement {
4
5     private int noOfOrders;
6     private double totalAmount;
7     private Order order;
8
9     public OrdersManagement(int noOfOrders, double totalAmount, Order order)
10        super();
11        this.noOfOrders = noOfOrders;
12        this.totalAmount = totalAmount;
13        this.order = order;
14    }
15

```

Console x
<terminated> AutowiringDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (11-Jul-2023, 1:16:0)
Product Object Created by IOC
Order Object Created by IOC
101

So Internally Spring injected **Product object by name of bean and property name of Order class.**

Question: If property name and Bean ID are different, then Spring will not inject Product object inside Order Object. Now I made bean id as prod for Product class.



```

<bean id="prod" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>

```

Not Matching

```

public class Order {
    private String orderId;
    private double orderValue;
    private Product product;

    // setters , getters, constructors
}

```

Test Our application and check Spring injected Product object or not inside Order.

```

Product Object Created by IOC
Order Object Created by IOC
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"com.flipkart.product.Product.getProductId()" because the return value of
"com.flipkart.orders.Order.getProduct()" is null at
com.flipkart.main.AutowiringDemo.main(AutowiringDemo.java:19)

```

autowire="byType":

Now configure **autowire="byType"** in side beans xml file for Order Bean configuration, because internally Product bean object should be injected to Order Bean Object. In this

autowire mode, We are expecting dependency injected by Spring instead of we are writing bean wiring with either using `<property>` and `<constructor-arg>` tags by using `ref` attribute. Means, eliminating logic of reference configurations.

As per `autowire="byType"`, Spring internally checks for a dependency bean objects, which are matched with Data Type of property and then that bean object will be injected. In this case Bean ID and Property Names may be different. As per our example, Product is dependency for Order class.

Bean Data Type i.e. class Name = Data type of property of Order class

```

<bean id="prod" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>

```

```

public class Order {
    private String orderId;
    private double orderValue;
    private Product product;
    // setters , getters, constructors
}

```

- **Beans Configuration:**

```

<beans>
    <bean id="prod" class="com.flipkart.product.Product">
        <property name="productId" value="101"></property>
        <property name="productName" value="Lenevo Laptop"></property>
    </bean>
    <bean id="order" class="com.flipkart.orders.Order" autowire="byType">
        <property name="orderId" value="order1234"></property>
        <property name="orderValue" value="33000.00"></property>
    </bean>
</beans>

```

- **Test Our Application Now:** Dependency Injected Successfully, because only One Product Object available.

```

3* import org.springframework.context.ApplicationContext;
4
5
6 public class AutowiringDemo {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new FileSystemXmlApplicationContext(
10             "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");
11
12         Order order = (Order) context.getBean("order");
13         System.out.println(
14             order.getProduct() // Product Object
15             .getProductId() // Product object : Getting product Id
16         );
17     }
18 }
19
20
21
22 }

Console > terminated> AutowiringDemo [Java Application] D:\\software\\eclipse\\plugins\\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\\jre\\bin\\javaw.exe (11-Jul-2023, 1:38:27 pm)
Product Object Created by IOC
Order Object Created by IOC
101

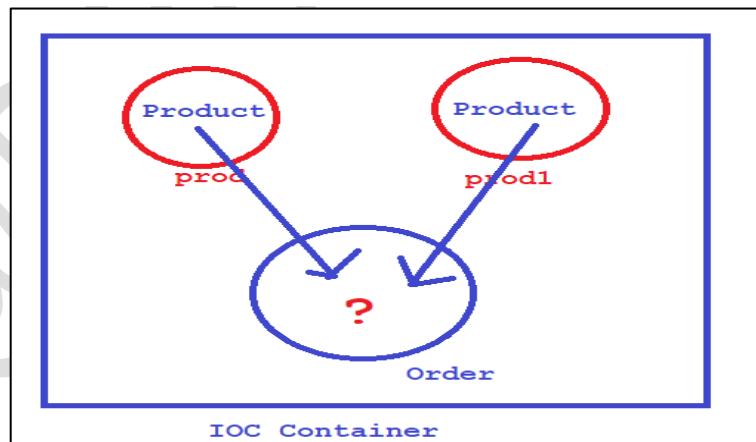
```

Question: If Product Bean configured more than one time inside beans configuration, then which Product Bean Object will be injected in side Order ?

```

<bean id="prod" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="prod2" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>

```



Test Our Application: We will get Exception while trying to inject Product Object because of ambiguity between 2 Objects.

```

Product Object Created by IOC
Product Object Created by IOC
Order Object Created by IOC
Jul 11, 2023 1:56:23 PM org.springframework.context.support.AbstractApplicationContext
refresh

```

WARNING: Exception encountered during context initialization - cancelling refresh attempt:
`org.springframework.beans.factory.UnsatisfiedDependencyException`: Error creating bean with name 'order' defined in file [D:\workspaces\naresit\bean-wiring\beans.xml]: Unsatisfied dependency expressed through bean property 'product'; nested exception is `org.springframework.beans.factory.NoUniqueBeanDefinitionException`: No qualifying bean of type 'com.flipkart.product.Product' available: expected single matching bean but found 2: prod,prod2

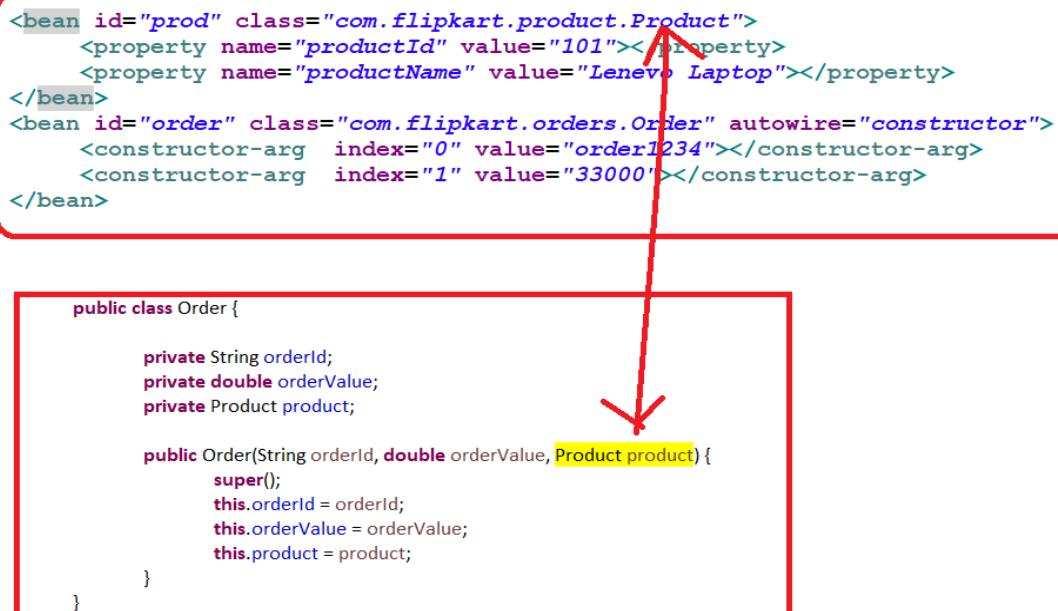
Exception in thread "main" `org.springframework.beans.factory.UnsatisfiedDependencyException`: Error creating bean with name 'order' defined in file [D:\workspaces\naresit\bean-wiring\beans.xml]: Unsatisfied dependency expressed through bean property 'product'; nested exception is `org.springframework.beans.factory.NoUniqueBeanDefinitionException`: No qualifying bean of type 'com.flipkart.product.Product' available: expected single matching bean but found 2: prod,prod2

autowire="constructor":

Now configure **autowire="constructor"** in side beans xml file for Order Bean configuration, because internally Product bean object should be injected to Order Bean Object. In this autowire mode, We are expecting dependency injected by Spring instead of we are writing bean wiring with either using `<property>` or `<constructor-arg>` tags by using `ref` attribute. Means, eliminating logic of reference configurations.

As per **autowire="constructor"**, Spring internally checks for a dependency bean objects, which are matched with **constructor argument** of same data type and then that bean object will be injected. Means, constructor autowire mode works with constructor injection not setter injection. In this case, injected Bean Type and Constructor Property Name should be same.

As per our example, Product is dependency for Order and Order class defined a constructor with parameter contains Product type.



Beans Configuration:

```

<bean id="prod" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="order" class="com.flipkart.orders.Order" autowire="constructor">
    <constructor-arg index="0" value="order1234"></constructor-arg>
    <constructor-arg index="1" value="33000"></constructor-arg>
</bean>

```

- **Test Our Application: Now Product Object Injected via Constructor.**

```

10 public class AutowiringDemo {
11
12    public static void main(String[] args) {
13        ApplicationContext context = new FileSystemXmlApplicationContext(
14            "D:\\workspaces\\nareesha\\bean-wiring\\beans.xml");
15
16        Order order = (Order) context.getBean("order");
17        System.out.println(
18            order.getProduct() // Product Object
19            .getProductId() // Product object : Getting product Id
20        );
21    }
22 }
23

```

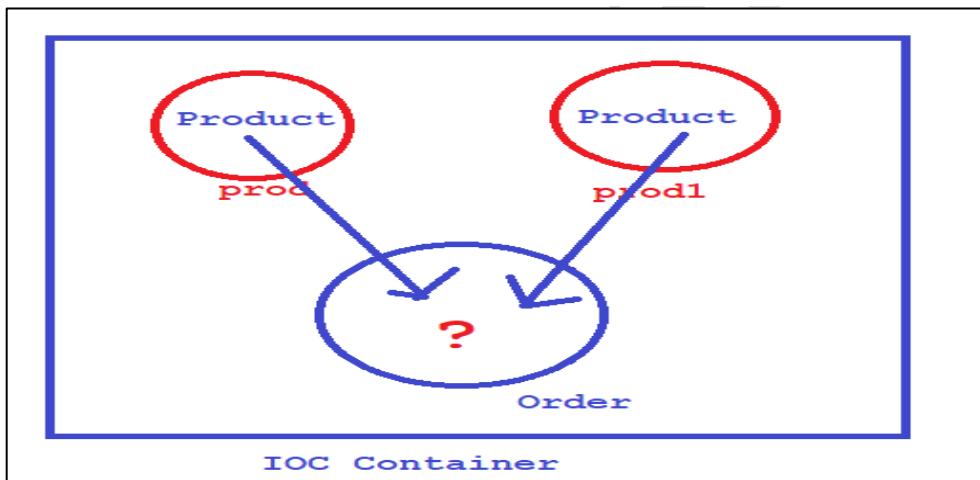
Console <terminated> AutowiringDemo [Java Application] D:\\softwares\\eclipse\\plugins\\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\\jre\\bin\\javaw.exe (11-Jul-2023)
Product Object Created by IOC
Order Object Created: With Params Constructor
101

Question: If Product Bean configured more than one time inside beans configuration, then which Product Bean Object will be injected in side Order?

when **autowire =constructor**, spring internally checks out of multiple bean ids dependency object which is matching with property name of Order class. If matching found then that specific bean object will be injected. If not found then we will get ambiguity exception as following.

As per our below configuration, both bean ids of Product are not matching with Order class property name of Product type.

```
<bean id="prod" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="prod2" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
```



- **Test Our Application:** We will get Exception while trying to inject Product Object because of ambiguity between 2 Objects.

Product Object Created by IOC

Product Object Created by IOC

Jul 11, 2023 1:56:23 PM org.springframework.context.support.AbstractApplicationContext refresh

WARNING: Exception encountered during context initialization - cancelling refresh attempt: org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'order' defined in file [D:\workspaces\naresit\bean-wiring\beans.xml]: Unsatisfied dependency expressed through bean property 'product'; nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.flipkart.product.Product' available: expected single matching bean but found 2: prod,prod2

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean
with name 'order' defined in file [D:\workspaces\naresit\bean-wiring\beans.xml]:
Unsatisfied dependency expressed through bean property 'product'; nested exception is
org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean
of type 'com.flipkart.product.Product' available: expected single matching bean but found
2: prod,prod2
```

Now if we configure one bean object of **Product** class with bean id which is matching with property name of **Order** class. Then that Specific Object will be injected. From following configuration **Product** object of bean id “**product**” will be injected.

```
<bean id="product" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="prod2" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

- **Test Our Application :**

```
10 public class AutowiringDemo {
11
12*     public static void main(String[] args) {
13         ApplicationContext context = new FileSystemXmlApplicationContext(
14             "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");
15
16         Order order = (Order) context.getBean("order");
17         System.out.println(
18             order.getProduct() // Product Object
19             .getProductId() // Product object : Getting product Id
20         );
21     }
22 }
23 *
```

Console x
terminated> AutowiringDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (11-Jul-2023)
Product Object Created by IOC
Order Object Created: With Params Constructor
101

Advantage of Autowiring:

- It requires less code because we don't need to write the code to inject the dependency explicitly.

Disadvantages of Autowiring:

- No control of the programmer.
- It can't be used for primitive and string values.

Bean Scopes in Spring:

When you start a Spring application, the Spring Framework creates beans for you. These Spring beans can be application beans that you have defined or beans that are part of the framework. When the Spring Framework creates a bean, it associates a scope with the bean. **A scope defines the life cycle and visibility of that bean within runtime application context which the bean instance is available.**

The Latest Spring Framework supports 5 scopes, last four are available only if you use Web aware of ApplicationContext i.e. inside Web applications.

1. singleton
2. prototype
3. request
4. session
5. application
6. websokcet

Defining Scope of beans syntax:

In XML configuration, we will use an attribute “scope”, inside `<bean>` tag as shown below.

```
<!-- A bean definition with singleton scope -->
<bean id = "... " class = "... " scope = "singleton">
    <!-- collaborators and configuration for this bean go here -->
</bean>
```

singleton:

This is **default scope** of a bean configuration i.e. even if we are not provided scope attribute as part of any bean configuration, then spring container internally consideres as `scope="singleton"`.

If Bean `scope=singleton`, then Spring Container creates only one object in side Spring container overall application level and Spring Container returns same instance reference always for every IOC container call i.e. `getBean()`.

```
<bean id="productOne" class="com.flipkart.product.Product" scope="singleton">
```

Above line is equal to following because default is `scope="singleton"`

```
<bean id="productOne" class="com.flipkart.product.Product">
```

Now create Bean class and call IOC container many times with same bean ID.

Product.java

```
public class Product {
    private String productId;
    private String productName;

    public Product() {
        System.out.println("Product Object Created by IOC");
    }

    //setters and getters methods
}
```

- Now configure above class in side beans xml file.

```
<bean id="product" class="com.flipkart.product.Product" scope="singleton">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

- Now call Get Bean from IOC Container for Product Bean Object. In Below, we are calling IOC container 3 times.

```
public class BeanScopesDemo {

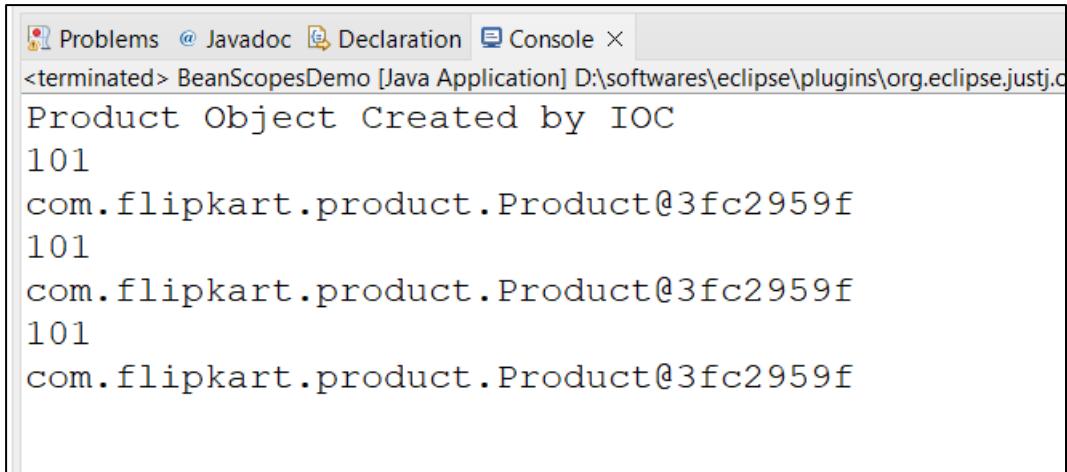
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
                "D:\\beans.xml");

        // 1. Requesting/Calling IOC Container for Product Object
        Product p1 = (Product) context.getBean("product");
        System.out.println(p1.getProductId());
        System.out.println(p1);

        // 2. Requesting/Calling IOC Container for Product Object
        Product p2 = (Product) context.getBean("product");
        System.out.println(p2.getProductId());
        System.out.println(p2);

        // 3. Requesting/Calling IOC Container for Product Object
        Product p3 = (Product) context.getBean("product");
        System.out.println(p3.getProductId());
        System.out.println(p3);
    }
}
```

Output:



```

Problems @ Javadoc Declaration Console X
<terminated> BeanScopesDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justj.oci
Product Object Created by IOC
101
com.flipkart.product.Product@3fc2959f
101
com.flipkart.product.Product@3fc2959f
101
com.flipkart.product.Product@3fc2959f

```

From above output, Spring Container created only one Object and same passed for every new container call with `getBean()` by passing bean id. Means, Singleton Object created for bean ID **product** in IOC container.

NOTE: If we created another bean id configuration for same class, then previous configuration behaviour will not applicable to current configuration i.e. every individual bean configuration or Bean Object having it's own behaviour and functionality in Spring Framework.

Create one more bean configuration of Product.

```

<bean id="product" class="com.flipkart.product.Product" scope="singleton">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="productTwo" class="com.flipkart.product.Product">
    <property name="productId" value="102"></property>
    <property name="productName" value="HP Laptop"></property>
</bean>

```

- **Testing:** In below we are requesting 2 different bean objects of Product.

```

public class BeanScopesDemo {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
            "D:\\workspaces\\\\naresit\\\\bean-wiring\\\\beans.xml");
        // 1. Requesting/Calling IOC Container for Product Object
        Product p1 = (Product) context.getBean("product");
        System.out.println(p1.getProductId());
        System.out.println(p1);
        // 2. Requesting/Calling IOC Container for Product Object
    }
}

```

```

Product p2 = (Product) context.getBean("product");
System.out.println(p2.getProductId());
System.out.println(p2);

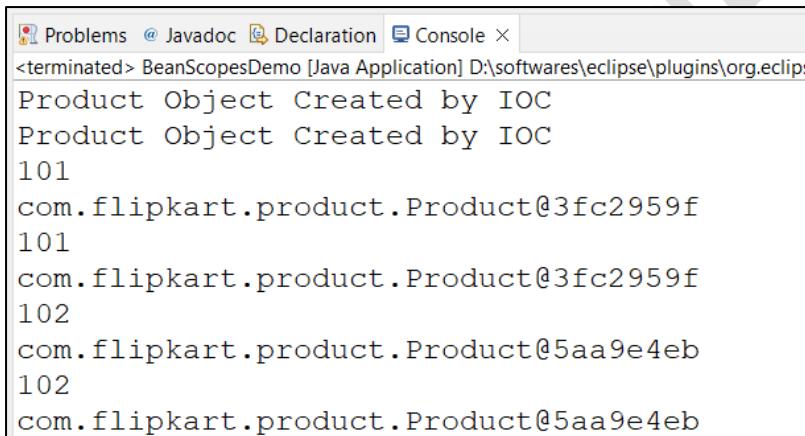
// 3. Requesting/Calling IOC Container for Product Object
Product p3 = (Product) context.getBean("productTwo");
System.out.println(p3.getProductId());
System.out.println(p3);

// 4. Requesting/Calling IOC Container for Product Object
Product p4 = (Product) context.getBean("productTwo");
System.out.println(p4.getProductId());
System.out.println(p4);
}

}

```

Output:



```

Problems @ Javadoc Declaration Console ×
<terminated> BeanScopesDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.jdt.core\src\com\flipkart\product\product.java:10: error: cannot find symbol
        System.out.println("Product Object Created by IOC");
                                         ^
  symbol:   class System
  location: class product
        System.out.println("Product Object Created by IOC");
                                         ^
  symbol:   class System
  location: class product
        101
        com.flipkart.product.Product@3fc2959f
        101
        com.flipkart.product.Product@3fc2959f
        102
        com.flipkart.product.Product@5aa9e4eb
        102
        com.flipkart.product.Product@5aa9e4eb

```

For 2 Bean configurations of Product class, 2 individual Singleton Bean Objects created.

prototype:

If Bean scope defined as “**prototype**”, a new instance of the bean is created every time it is requested from the container. It is not cached, so each request/call to IOC container for the bean will return in a new instance.

Bean class: Product.java

```

public class Product {

    private String productId;
    private String productName;

    public Product() {
        System.out.println("Product Object Created by IOC");
    }
}

```

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

XML Bean configuration:

```
<bean id="product" class="com.flipkart.product.Product" scope="prototype">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

Testing :



```

7
8 public class BeanScopesDemo {
9     public static void main(String[] args) {
10         ApplicationContext context = new FileSystemXmlApplicationContext(
11             "D:\\\\workspaces\\\\naresit\\\\bean-wiring\\\\beans.xml");
12         // 1. Requesting/Calling IOC Container for Product Object
13         Product p1 = (Product) context.getBean("product");
14         System.out.println(p1);
15         System.out.println(p1.getProductId());
16         // 2. Requesting/Calling IOC Container for Product Object
17         Product p2 = (Product) context.getBean("product");
18         System.out.println(p2);
19         System.out.println(p2.getProductId());
20     }
21 }
```

Product Object Created by IOC
com.flipkart.product.Product@5542c4ed
101
Product Object Created by IOC
com.flipkart.product.Product@1573f9fc
101

Now Spring Container created and returned every time new Bean Object for every Container call `getBean()` for same bean ID.

request:

When we apply scope as request, then for every new HTTP request Spring will creates new instance of configured bean. Only valid in the context of a web-aware Spring ApplicationContext i.e. in web/MVC applications.

session:

When we apply scope as session, then for every new HTTP session creation in server side Spring will creates new instance of configured bean. Only valid in the context of a web-aware Spring ApplicationContext i.e. in web/MVC applications.

application:

Once you have defined the application-scoped bean, Spring will create a single instance of the bean per web application context. Any requests for this bean within the same web application will receive the same instance.

It's important to note that the application scope is specific to web applications and relies on the lifecycle of the web application context. Each web application running in a container will have its own instance of the application-scoped bean.

You can use application-scoped beans to store and share application-wide state or resources that need to be accessible across multiple components within the same web application.

websocket:

This is used as part of socket programming. We can't use in our Servlet based MVC application level.

Java/Annotation Based Beans Configuration:

So far we have seen how to configure Spring beans using XML configuration file. Java-based configuration option enables you to write most of your Spring configuration without XML but with the help of annotations.

@Configuration & @Bean Annotations:

@Configuration:

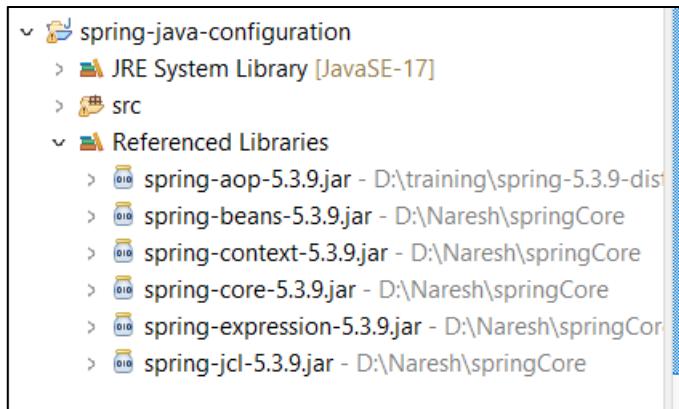
In Java Spring, the **@Configuration** annotation is used to indicate that this class is a configuration class of Beans. A configuration class is responsible for defining beans and their dependencies in the Spring application context. Beans are objects that are managed by the Spring IOC container. Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IOC container as a source of bean definitions.

Create a Java class and annotate it with **@Configuration**. This class will serve as our configuration class.

@Bean:

In Spring, the **@Bean** annotation is used to declare a method as a bean definition method within a configuration class. The **@Bean** annotation tells Spring that a method annotated with **@Bean** will return an object that should be registered as a bean in the Spring application context. The method annotated with **@Bean** is responsible for creating and configuring an instance of a bean that will be managed by the Spring IoC (Inversion of Control) container.

- Now Create a Project and add below jars to support Spring Annotations of Core Module.



NOTE: Added one extra jar file comparing with previous project setup. Because internally Spring core module using AOP functionalities to process annotations.

- Now Create a Bean class : **UserDetails**

```
package com.amazon.users;

public class UserDetails {

    private String firstName;
    private String lastName;
    private String emailld;
    private String password;
    private long mobile;

    //setters and getters
}
```

Now Create a Beans Configuration class. i.e. Class Marked with an annotation **@Configuration**. In side this configuration class, we will define multiple bean configurations with **@Bean** annotation methods.

Configuration class would be as follows:

```
package com.amazon.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.amazon.users.UserDetails;

@Configuration
public class BeansConfiguration {
```

```

@Bean("userDetails")
UserDetails getUserDetails() {
    return new UserDetails();
}
}

```

The above code will be equivalent to the following XML bean configuration –

```

<beans>
    <bean id = "userDetails"  class = "com.amazon.users.UserDetails" />
</beans>

```

Here, the method is annotated with **@Bean("userDetails")** works as bean ID is **userDetails** and Spring creates bean object with that bean ID and returns the same bean object when we call **getBean()**. Your configuration class can have a declaration for more than one **@Bean**. Once your configuration classes are defined, you can load and provide them to Spring container using **AnnotationConfigApplicationContext** as follows .

```

package com.amazon;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfiguration;
import com.amazon.users.UserDetails;

public class SpringBeanMainApp {
    public static void main(String[] args) {

        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(BeansConfiguration.class);

        UserDetails user = (UserDetails) context.getBean("userDetails");
        System.out.println(user);
        context.close();
    }
}

```

Output: Prints hash code of Object

com.amazon.users.UserDetails@34bde49d

Understand From above code :

AnnotationConfigApplicationContext:

AnnotationConfigApplicationContext is a class provided by the Spring Framework that serves as an implementation of the **ApplicationContext** interface. It is used to create an application context container by reading Java-based configuration metadata. In Spring, there

are multiple ways to configure the application context, such as XML-based configuration or Java-based configuration using annotations.

AnnotationConfigApplicationContext is specifically used for Java-based configuration. It allows you to bootstrap the Spring container by specifying one or more Spring configuration classes that contain **@Configuration** annotations.

We will provide configuration classes as Constructor parameters of **AnnotationConfigApplicationContext** or spring provided **register()** method also. We will have example here after.

- **context.getBean()**, Returns an instance, which may be shared or independent, of the specified bean.
- **context.close()**, Close this application context, destroying all beans in its bean factory.

Multiple Configuration Classes and Beans:

Now we can configure multiple bean classes inside multiple configuration classes as well as Same bean with multiple bean id's.

Now I am creating one more Bean class in above application.

```
package com.amazon.products;

public class ProductDetails {

    private String pname;
    private double price;

    public String getPname() {
        return pname;
    }
    public void setPname(String pname) {
        this.pname = pname;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}
```

Configuring above POJO class as Bean class inside Beans Configuration class.

```
package com.amazon.config;

import org.springframework.context.annotation.Bean;
import com.amazon.products.ProductDetails;

public class BeansConfigurationTwo {

    @Bean("productDetails")
    ProductDetails productDetails() {
        return new ProductDetails();
    }
}
```

➤ Testing Bean Object Created or not. Below Code loading Two Configuration classes.

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfiguration;
import com.amazon.config.BeansConfigurationTwo;
import com.amazon.products.ProductDetails;
import com.amazon.users.UserDetails;
public class SpringBeanMainApp {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context
            = new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class);           //UserDetails Bean Config.
        context.register(BeansConfigurationTwo.class); //ProductDetails Bean Config.
        context.refresh();

        UserDetails user = (UserDetails) context.getBean("userDetails");
        System.out.println(user);
        ProductDetails product = (ProductDetails) context.getBean("productDetails");
        System.out.println(product);
        context.close();
    }
}
```

Now Crate multiple Bean Configurations for same Bean class.

- Inside Configuration class: Two Bean configurations for **ProductDetails** Bean class.

```
import org.springframework.context.annotation.Bean;
import com.amazon.products.ProductDetails;

public class BeansConfigurationTwo {

    @Bean("productDetails")
    ProductDetails productDetails() {
        return new ProductDetails();
    }

    @Bean("productDetailsTwo")
    ProductDetails productTwoDetails() {
        return new ProductDetails();
    }
}
```

- Now get Both bean Objects of ProductDeatils.

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfiguration;
import com.amazon.config.BeansConfigurationTwo;
import com.amazon.products.ProductDetails;
import com.amazon.users.UserDetails;

public class SpringBeanMainApp {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class);
        context.register(BeansConfigurationTwo.class);
        context.refresh();

        UserDetails user = (UserDetails) context.getBean("userDetails");
        System.out.println(user);

        ProductDetails product = (ProductDetails) context.getBean("productDetails");
        System.out.println(product);

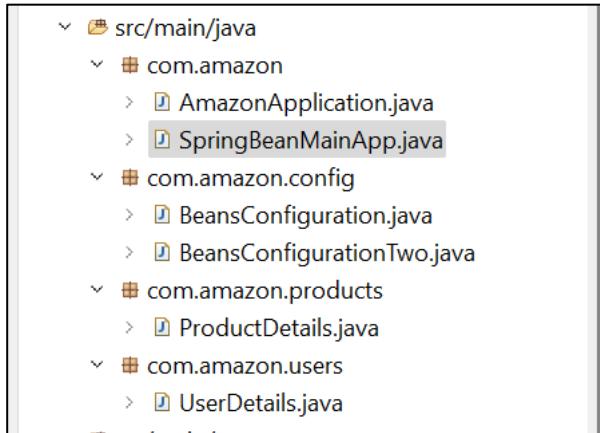
        ProductDetails productTwo = (ProductDetails) context.getBean("productDetailsTwo");
        System.out.println(productTwo);
        context.close();
    }
}
```

Output:

```
com.amazon.users.UserDetails@7d3e8655
com.amazon.products.ProductDetails@7dfb0c0f
com.amazon.products.ProductDetails@626abbd0
```

From above Output Two **ProductDetails** bean objects created by Spring Container.

Above Project Files Structure:



@Component Annotation :

Before we can understand the value of **@Component**, we first need to understand a little bit about the **Spring ApplicationContext**.

Spring **ApplicationContext** is where Spring holds instances of objects that it has identified to be managed and distributed automatically. These are called beans. Some of Spring's main features are bean management and dependency injection. Using the Inversion of Control principle, **Spring collects bean instances from our application and uses them at the appropriate time**. We can show bean dependencies to Spring without handling the setup and instantiation of those objects.

However, the base/regular spring bean definitions are explicitly defined in the XML file or configured in configuration class with **@Bean**, while the annotations drive only the dependency injection. This section describes an option for implicitly/internally detecting the candidate components by scanning the classpath. Components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration. Instead, you can use annotations (for example, **@Component**) to select which classes have bean definitions registered with the container.

We should take advantage of Spring's automatic bean detection by using stereotype annotations in our classes.

@Component: This annotation that allows Spring to detect our custom beans automatically. In other words, without having to write any explicit code, Spring will:

- Scan our application for classes annotated with `@Component`
- Instantiate them and inject any specified dependencies into them
- Inject them wherever needed

We have other more specialized stereotype annotations like `@Controller`, `@Service` and `@Repository` to serve this functionality derived, we will discuss then in MVC module level.

Define Spring Components :

1. Create a java Class and provide an annotation `@Component` at class level.

```
package com.tek.teacher;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
public class Product {

    private String pname;
    private double price;

    public Product(){
        System.out.println("Product Object Created.");
    }
    public String getPname() {
        return pname;
    }
    public void setPname(String pname) {
        this.pname = pname;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}
```

- Now Test in Main Class.

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

public class SpringComponentDemo {

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

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();

        context.scan("com.tek.teacher");
        context.refresh();

        Product details = (Product) context.getBean(Product.class);
        System.out.println(details);
    }
}
```

From Above program, **context.scan()** method, Perform a scan for **@Component** classes to instantiate Bean Objects within the specified base packages. We can pass many package names wherever we have **@Componet** Classes. Note that, **scan(basePackages)** method will scans **@Configuraion** classes as well from specified package names. Note that **refresh()** must be called in order for the context to fully process the new classes.

Spring Provided One more way which used mostly in Real time applications is using **@ComponentScan** annotation. To enable auto detection of Spring components, we shou use another annotation **@ComponentScan**.

@ComponentScan:

Before we rely entirely on **@Component**, we must understand that it's only a plain annotation. The annotation serves the purpose of differentiating beans from other objects, such as domain objects. However, Spring uses the **@ComponentScan** annotation to gather all component into its **ApplicationContext**.

@ComponentScan annotation is used to specify packages for spring to scan for annotated components. Spring needs to know which packages contain beans, otherwise you would have to register each bean individually. Hence **@ComponentScan** annotation is a supporting annotation for **@Configuration** annotation. Spring instantiate Bean Objects of components from specified packages for those classes annotated with **@Component**.

So create a beans configuration class i.e. **@Configuration** annotated class and provide **@ComponentScan** with base package name.

Ex: When we have to scan multiple packages we can pass all package names as String array with attribute **basePackages** .

```
@ComponentScan(basePackages =
    {"com.hello.spring.*", "com.hello.spring.boot.*"})
```

Or If only one base package and it's sub packages should be scanned, then we can directly pass package name.

```
@ComponentScan("com.hello.spring.*")
```

Test our component class:

- Create A **@Configuration** class with **@ComponentScan** annotation.

```
@Configuration
//making sure scanning all packages starts with com.tek.teacher
@ComponentScan("com.tek.teacher.*")
public class BeansConfiguration {
}
```

- Now Load/pass above configuration class to Application Context i.e. Spring Container.

```
package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfiguration.class);
        context.refresh();
        Product details = (Product) context.getBean(Product.class);
        System.out.println(details);
    }
}
```

- Now Run your Main class application.

Output: **ProductDetails [pname=null, price=0.0]**

From above, Spring Container detected **@Component** classes from all packages and instantiated as Bean Objects.

Now Add One More @Component class:

```
package com.tek.teacher;

import org.springframework.stereotype.Component;

@Component
public class UserDetails {

    private String firstName;
    private String lastName;
    private String emailId;
}
```

```

private String password;
private long mobile;

public UserDetails(){
    System.out.println("UserDetails Object Created");
}
public String getFirstName() {
    return firstName;
}
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
public String getLastName() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
public String getEmailId() {
    return emailId;
}
public void setEmailId(String emailId) {
    this.emailId = emailId;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
public long getMobile() {
    return mobile;
}
public void setMobile(long mobile) {
    this.mobile = mobile;
}
}

```

- Now get UserDetails from Spring Container and Test/Run our Main class.

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {

    public static void main(String[] args) {

```

```

AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class);
        context.refresh();
        //UserDetails Component
        UserDetails userDetails = context.getBean(UserDetails.class);
        System.out.println(userDetails);
    }
}

```

Output: com.tek.teacher.UserDetails@5e17553a

NOTE: In Above Logic, used **getBean(Class<UserDetails> requiredType)**, Return the bean instance that uniquely matches the given object type, if any. Means, when we are not configured any component name or don't want to pass bean name from getBean() method.

We can use any of overloaded method **getBean()** to get Bean Object as per our requirement or functionality demanding.

Can we Pass Bean Scope to @Component Classes?

Yes, Similar to **@Bean** annotation level however we are assigning scope type , we can pass in same way with **@Component** class level because Component is nothing but Bean finally.

Ex : From above example, requesting another Bean Object of type UserDetails without configuring scope at component class level.

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

        // UserDetails Component
        UserDetails userDetails = context.getBean(UserDetails.class);
        System.out.println(userDetails);

        UserDetails userTwo = context.getBean(UserDetails.class);
        System.out.println(userTwo);
    }
}

```

Output:

```
com.tek.teacher.UserDetails@5e17553a
com.tek.teacher.UserDetails@5e17553a
```

So we can say by default component classes are instantiated as singleton bean object, when there is no scope defined. Means, Internally Spring Container considering as **singleton scope**.

Question : Can we create @Bean configurations for @Component class?

Yes, We can create Bean Configurations in side Spring Configuration classes. With That Bean ID, we can request from Application Context, as usual.

Inside Configuration Class:

```
package com.tek.teacher;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.tek.*")
public class BeansConfiguration {

    @Bean("user")
    UserDetails getUserDetails() {
        return new UserDetails();
    }
}
```

➤ **Testing from Main Class:**

```
package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

        // UserDetails Component
        UserDetails userThree = (UserDetails) context.getBean("user");
        System.out.println(userThree);
    }
}
```

Output: com.tek.teacher.UserDetails@3eb91815

Question : How to pass default values to @Component class properties?

We can pass/initialize default values to a component class instance with **@Bean** method implementation inside Spring Configuration classes.

Inside Configuration Class:

```
package com.tek.teacher;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.tek.*")
public class BeansConfiguration {

    @Bean("user")
    UserDetails getUserDetails() {
        UserDetails user = new UserDetails();
        user.setEmailId("dilip@gmail.com");
        user.setMobile(8826111377l);
        return user;
    }
}
```

Main App:

```
package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

        // UserDetails Component
        UserDetails userThree = (UserDetails) context.getBean("user");
        System.out.println(userThree.getEmailId());
        System.out.println(userThree.getMobile());
    }
}
```

Output:

dilip@gmail.com
8826111377

Defining Scope of beans with Annotations:

In XML configuration, we will use an attribute “**scope**”, inside **<bean>** tag as shown below.

```
<!-- A bean definition with singleton scope -->
<bean id = "..." class = "..." scope = "singleton">
    <!-- collaborators and configuration for this bean go here -->
</bean>
```

Annotation Based Scope Configuration:

We will use **@Scope** annotation will be used to define scope type.

@Scope: A bean’s scope is set using the **@Scope** annotation. By default, the Spring framework creates exactly one instance for each bean declared in the IoC container. This instance is shared in the scope of the entire IoC container and is returned for all subsequent **getBean()** calls and bean references.

Example: Create a bean class and configure with Spring Container : **ProductDetails.java**

```
package com.amazon.products;

public class ProductDetails {
    private String pname;
    private double price;
    public String getPname() {
        return pname;
    }
    public void setPname(String pname) {
        this.pname = pname;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public void printProductDetails() {
        System.out.println("Product Details Are : .....");
    }
}
```

Now Inside Configuration class, Define Bean Creation and Configure scope value.

Singleton Scope:

A single Bean object instance created and returns same Bean instance for each Spring IoC container call i.e. **getBean()**. Inside Configuration class, **scope** value defined as **singleton**.

NOTE: If we are not defined any scope value for any Bean Configuration, then Spring Container by default considers scope as **singleton**.

```
package com.amazon.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
import com.amazon.products.ProductDetails;

@Configuration
public class BeansConfigurationThree {
    @Scope("singleton")
    @Bean("productDetails")
    ProductDetails getProductDetails() {
        return new ProductDetails();
    }
}
```

- Now Test Bean **ProductDetails** Object is singleton or not. Request multiple times **ProductDetails** Object from Spring Container by passing bean id **productDetails**.

```
package com.amazon;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BbeansConfigurationThree;
import com.amazon.products.ProductDetails;

public class SpringBeanScopeTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfigurationThree.class);
        context.refresh();
        ProductDetails productOne = (ProductDetails) context.getBean("productDetails");
        System.out.println(productOne);
        ProductDetails productTwo = (ProductDetails) context.getBean("productDetails");
        System.out.println(productTwo);
        context.close();
    }
}
```

Output:

```
com.amazon.products.ProductDetails@58e1d9d
com.amazon.products.ProductDetails@58e1d9d
```

From above output, we can see same hash code printed for both getBean() calls on Spring Container. Means, Container created singleton instance for bean id “**productDetails**”.

Prototype Scope: In side Configuration class, scope value defined as **prototype**.

```
package com.amazon.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
import com.amazon.products.ProductDetails;

@Configuration
public class BeansConfigurationThree {
    @Scope("singleton")
    @Bean("productDetails")
    ProductDetails getProductDetails() {
        return new ProductDetails();
    }
    @Scope("prototype")
    @Bean("productTwoDetails")
    ProductDetails getProductTwoDetails() {
        return new ProductDetails();
    }
}
```

- Now Test Bean **ProductDetails** Object is **prototype** or not. Request multiple times **ProductDetails** Object from Spring Container by passing bean id **productTwoDetails**.

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BbeansConfigurationThree;
import com.amazon.products.ProductDetails;

public class SpringBeanScopeTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfigurationThree.class);
        context.refresh();
        //Prototype Beans:
        ProductDetails productThree =
            (ProductDetails) context.getBean("productTwoDetails");
        System.out.println(productThree);
    }
}
```

```

        ProductDetails productFour =
                (ProductDetails) context.getBean("productTwoDetails");
        System.out.println(productFour);
        context.close();
    }
}

```

Output:

```

com.amazon.products.ProductDetails@12591ac8
com.amazon.products.ProductDetails@5a7fe64f

```

From above output, we can see different hash codes printed for both getBean() calls on Spring Container. Means, Container created new instance every time when we requested for instance of bean id “**productTwoDetails**”.

Scope of @Component classes:

If we want to define scope of with component classes and Objects externally, then we will use same **@Scope** at class level of component class similar to **@Bean** method level in previous examples.

If we are not passed any scope value via **@Scope** annotation to a component class, then Component Bean Object will be created as singleton as usually.

Now for **UserDetails** class, added scope as **prototype**.

```

@Scope("prototype")
@Component
public class UserDetails {
    //Properties
    //Setter & Getters
    // Methods
}

```

Now test from Main application class, whether we are getting new Instance or not for every request of Bean Object **UserDetails** from Spring Container.

```

package com.tek.teacher.products;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

```

```
// UserDetails Component
UserDetails userDetails = context.getBean(UserDetails.class);
System.out.println(userDetails);

UserDetails userTwo = context.getBean(UserDetails.class);
System.out.println(userTwo);
}
```

Output:

com.tek.teacher.UserDetails@74f6c5d8
 com.tek.teacher.UserDetails@27912e3

NOTE: Below four are available only if you use a web-aware **ApplicationContext** i.e. inside Web applications.

- **request**
- **session**
- **application**
- **globalsession**

Auto Wiring In Spring

- Autowiring feature of spring framework enables you to inject the object dependency implicitly.
- Autowiring can't be used to inject primitive and string values. It works with reference only.
- It requires the less code because we don't need to write the code to inject the dependency explicitly.
- Autowired is allows spring to resolve the collaborative beans in our beans. Spring boot framework will enable the automatic injection dependency by using declaring all the dependencies in the configurations.
- We will achieve auto wiring with an Annotation **@Autowired**

Auto wiring will be achieved in multiple ways/modes.

Auto Wiring Modes:

- **no**
- **byName**
- **byType**
- **constructor**

- ⊕ **no:** It is the default autowiring mode. It means no autowiring by default.
- ⊕ **byName:** The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
- ⊕ **byType:** The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
- ⊕ **constructor:** The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

In XML configuration, we will enable autowiring between Beans as shown below.

```
<bean id="a" class="org.sssit.A" autowire="byName">
    .....
</bean>
```

In Annotation Configuration, we will use **@Autowired** annotation.

We can use **@Autowired** in following methods.

1. On properties
2. On setter
3. On constructor

@Autowired on Properties

Let's see how we can annotate a property using **@Autowired**. This eliminates the need for getters and setters.

First, Let's Define a bean : Address.

```
package com.dilip.account;

import org.springframework.stereotype.Component;

@Component
public class Address {

    private String streetName;
    private int pincode;

    public String getStreetName() {
        return streetName;
    }

    public void setStreetName(String streetName) {
        this.streetName = streetName;
    }

    public int getPincode() {
```

```

        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
    @Override
    public String toString() {
        return "Address [streetName=" + streetName + ", pincode=" + pincode + "]";
    }
}

```

Now Define, Another component class **Account** and define Address type property inside as a Dependency property.

```

package com.dilip.account;

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

@Component
public class Account {

    private String name;
    private long accNumber;

    // Field/Property Level
    @Autowired
    private Address addr;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public long getAccNumber() {
        return accNumber;
    }
    public void setAccNumber(long accNumber) {
        this.accNumber = accNumber;
    }
    public Address getAddr() {
        return addr;
    }
    public void setAddr(Address addr) {
        this.addr = addr;
    }
}

```

- Create a configuration class, and define Component Scan packages to scan all packages.

```
package com.dilip.account;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.dilip.*")
public class BeansConfiguration {

}
```

- Now Define, Main class and try to get Account Bean object and check really Address Bean Object Injected or Not.

```
package com.dilip.account;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

        // UserDetails Component
        Account account = (Account) context.getBean(Account.class);
        // Getting Injected Object of Address
        Address address = account.getAddr();
        address.setPincode(500072);

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

Output: Address [streetName=null, pincode=500072]

So, Dependency Object **Address** injected in **Account** Bean Object implicitly, with **@Autowired** on property level.

Autowiring with Multiple Bean ID Configurations with Single Bean/Component Class:

Let's Create Bean class: Below class Bean Id is : **home**

```
package com.hello.spring.boot.employees;

import org.springframework.stereotype.Component;

@Component("home")
public class Addresss {

    private String streetName;
    private int pincode;

    public String getStreetName() {
        return streetName;
    }

    public void setStreetName(String streetName) {
        this.streetName = streetName;
    }

    public int getPincode() {
        return pincode;
    }

    public void setPincode(int pincode) {
        this.pincode = pincode;
    }

    public void printAddressDetails() {
        System.out.println("Street Name is : " + this.streetName);
        System.out.println("Pincode is : " + this.pincode);
    }
}
```

➤ For above Address class create a Bean configuration in Side Configuration class.

```
package com.hello.spring.boot.employees;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.hello.spring.boot.*")
public class BeansConfig {

    @Bean("hyd")
    Addresss createAddress() {
        Addresss a = new Addresss();
        a.setStreetName("Hyderabad");
        a.setPincode(500001);
        return a;
    }
}
```

```
        a.setPincode(500067);
        a.setStreetName("Gachibowli");
        return a;
    }
}
```

➤ Now Autowire Address in Employee class.

```
package com.hello.spring.boot.employees;

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

@Component("emp")
public class Employee {

    private String lastName;
    private long mobile;

    @Autowired
    private Addresss add;

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public long getMobile() {
        return mobile;
    }
    public void setMobile(long mobile) {
        this.mobile = mobile;
    }
    public Addresss getAdd() {
        return add;
    }
    public void setAdd(Addresss add) {
        this.add = add;
    }
}
```

- Now Test which Address Object Injected by Container i.e. either home or hyd bean object.

```
package com.hello.spring.boot.employees;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AutowiringTestMainApp {

    public static void main(String[] ar) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfig.class);
        context.refresh();
        Employee empemployee = (Employee) context.getBean("emp");
        Addresss empAdd = empemployee.getAdd();
        System.out.println(empAdd);
    }
}
```

We got an exception now as follows,

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'emp': Unsatisfied dependency expressed through field 'add': No qualifying bean of type 'com.hello.spring.boot.employees.Addresss' available: expected single matching bean but found 2: home,hyd
```

i.e. Spring Container unable to inject Address Bean Object into Employee Object because of Ambiguity/Confusion like in between **home** or **hyd** bean Objects of Address type.

To resolve this Spring provided one more annotation called as **@Qualifier**

@Qualifier:

By using the **@Qualifier** annotation, we can eliminate the issue of which bean needs to be injected. There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

We need to take into consideration that the qualifier name to be used is the one declared in the **@Component** or **@Bean** annotation.

Now add **@Qualifier** on **Address** filed, inside **Employee** class.

```

package com.hello.spring.boot.employees;

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

@Component("emp")
public class Employee {

    private String lastName;
    private long mobile;

    @Qualifier("hyd")
    @Autowired
    private Addresss add;

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public long getMobile() {
        return mobile;
    }

    public void setMobile(long mobile) {
        this.mobile = mobile;
    }

    public Addresss getAdd() {
        return add;
    }

    public void setAdd(Addresss add) {
        this.add = add;
    }
}

```

➤ Now Test which Address Bean Object with bean Id “hyd” Injected by Container.

```

package com.hello.spring.boot.employees;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AutowiringTestMainApp {
    public static void main(String[] ar) {

        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
    }
}

```

```

        context.register(BeansConfig.class);
        context.refresh();
        Employee empployee = (Employee) context.getBean("emp");
        Addresss empAdd = empployee.getAdd();
        System.out.println(empAdd.getPincode());
        System.out.println(empAdd.getStreetName());
    }
}

```

Output: 500067
Gachibowli

i.e. **Address** Bean Object Injected with Bean Id called as **hyd** into **Employee** Bean Object.

@Primary:

There's another annotation called **@Primary** that we can use to decide which bean to inject when ambiguity is present regarding dependency injection. This annotation **defines a preference when multiple beans of the same type are present**. The bean associated with the **@Primary** annotation will be used unless otherwise indicated.

Now add One more **@Bean** config for **Address** class inside Configuration class.

```

package com.hello.spring.boot.employees;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration
@ComponentScan("com.hello.spring.boot.*")
public class BeansConfig {
    @Bean("hyd")
    Addresss createAddress() {
        Addresss a = new Addresss();
        a.setPincode(500067);
        a.setStreetName("Gachibowli");
        return a;
    }
    @Bean("banglore")
    @Primary
    Addresss bangloreAddress() {
        Addresss a = new Addresss();
        a.setPincode(560043);
        a.setStreetName("Banglore");
        return a;
    }
}

```

In above, we made `@Bean("banglore")` as Primary i.e. by Default bean object with ID **"banglore"** should be injected out of multiple Bean definitions of Address class when `@Qualifier` is not defined with `@Autowired`.

```
package com.hello.spring.boot.employees;

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

@Component("emp")
public class Employee {

    private String lastName;
    private long mobile;

    //No @Qualifier Defined
    @Autowired
    private Addresss add;

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public long getMobile() {
        return mobile;
    }
    public void setMobile(long mobile) {
        this.mobile = mobile;
    }
    public Addresss getAdd() {
        return add;
    }
    public void setAdd(Addresss add) {
        this.add = add;
    }
}
```

Output:

560043
Banglore

I.e. **Address** Bean Object with **"banglore"** injected in **Employee** object level.

NOTE: if both the **@Qualifier** and **@Primary** annotations are present, then the **@Qualifier** annotation will have precedence/priority. Basically, **@Primary** defines a default, while **@Qualifier** is very specific to Bean ID.

Autowiring With Interface and Implemented Classes:

In Java, Interface reference can hold Implemented class Object. With this rule, We can Autowire Interface references to inject implemented component classes.

- Now Define an Interface: **Animal**

```
package com.dilip.auto.wiring;

public interface Animal {
    void printNameOfAnimal();
}
```

- Now Define A class from interface : **Tiger**

```
package com.dilip.auto.wiring;

import org.springframework.stereotype.Component;

@Component
public class Tiger implements Animal {
    @Override
    public void printNameOfAnimal() {
        System.out.println("I am a Tiger ");
    }
}
```

- Now Define a Configuration class for component scanning.

```
package com.dilip.auto.wiring;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.dilip.*")
public class BeansConfig {
```

- Now Autowire **Animal** type property in any other Component class i.e. Dependency of **Animal** Interface implemented class Object **Tiger** should be injected.

```
package com.dilip.auto.wiring;

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

@Component
public class AnimalManagement {

    @Autowired
    //Interface Type Property
    Animal animal;

}
```

- Now Test, Animal type property injected with what type of Object.

```
package com.dilip.auto.wiring;

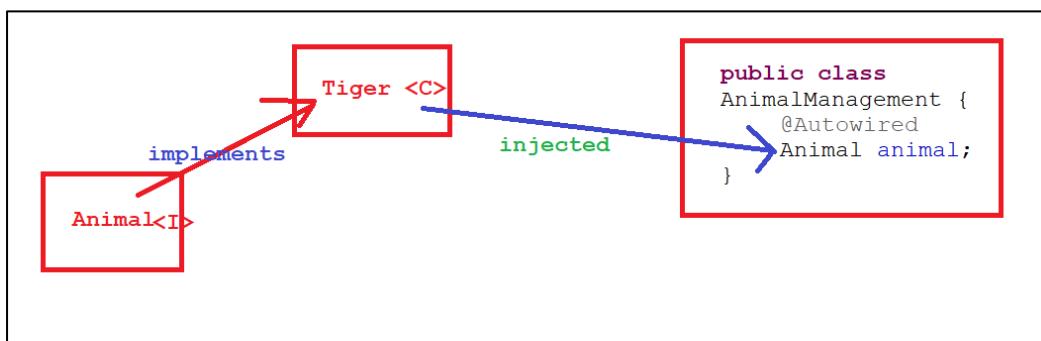
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

        // Getting AnimalManagement Bean Object
        AnimalManagement animalMgmt = context.getBean(AnimalManagement.class);
        animalMgmt.animal.printNameOfAnimal();
    }
}
```

Output: I am a Tiger

So, implicitly Spring Container Injected one and only implanted class Tiger of Animal Interface inside Animal Type reference property of AnimalManagement Object.



If we have multiple Implemented classes for same Interface i.e. Animal interface, How Spring Container deciding which implanted Bean object should Injected?

- Define one more Implementation class of Animal Interface : **Lion**

```
package com.dilip.auto.wiring;

import org.springframework.stereotype.Component;

@Component("lion")
public class Lion implements Animal {
    @Override
    public void printNameOfAnimal() {
        System.out.println("I am a Lion ");
    }
}
```

- Now Test, **Animal** type property injected with what type of Object either **Tiger** or **Lion**.

```
package com.dilip.auto.wiring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

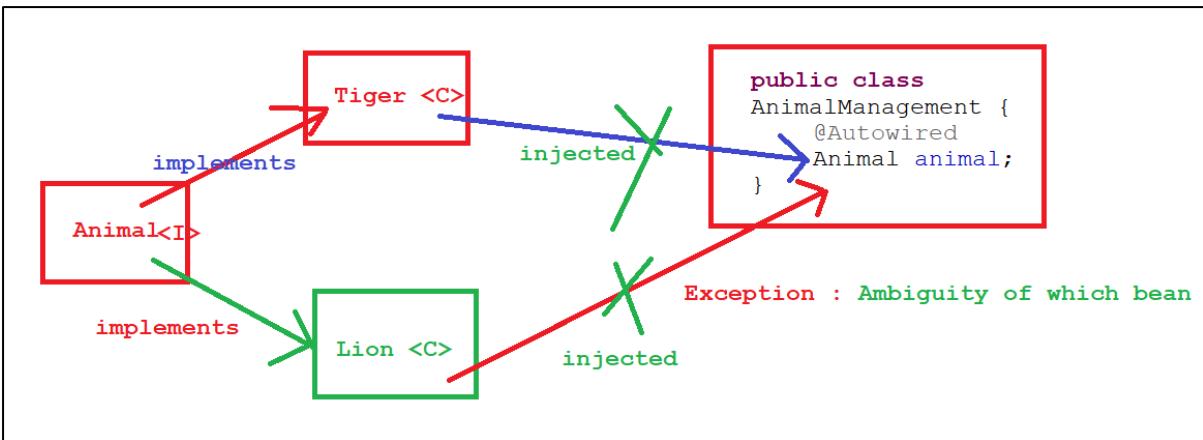
public class AutoWringDemo {
    public static void main(String[] args) {

        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext();
        context.register(BeansConfig.class);
        context.refresh();

        // Getting AnimalManagement Bean Object
        AnimalManagement animalMgmt = context.getBean(AnimalManagement.class);
        animalMgmt.animal.printNameOfAnimal();
    }
}
```

We got an Exception as,

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'animalManagement':
Unsatisfied dependency expressed through field 'animal': No
qualifying bean of type 'com.dilip.auto.wiring.Animal'
available: expected single matching bean but found 2:
lion,tiger
```



So to avoid this ambiguity again between multiple implementation of single interface, again we can use **@Qualifier** with Bean Id or Component Id.

```
package com.dilip.auto.wiring;

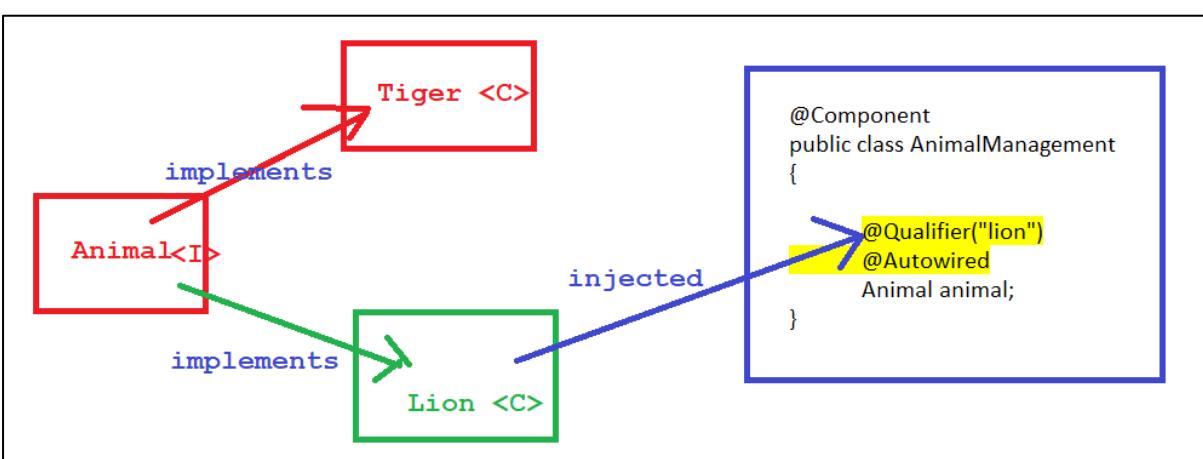
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
public class AnimalManagement {
    @Qualifier("lion")
    @Autowired
    Animal animal;
}
```

Now it will inject only **Lion** Object inside **AnimalManagement** Object as per **@Qualifier** annotation value out of **lion** and **tiger** bean objects.

Run again now **AutoWringDemo.java**

Output: I am a Lion.



Can we inject Default implemented class Object out of multiple implementation classes into Animal reference if not provided any Qualifier value?

Yes, we can inject default Implementation bean Object of Interface. We should mark one class as **@Primary**. Now I marked Tiger class as **@Primary** and removed **@Qualifier** from **AnimalManagement**.

```
package com.dilip.auto.wiring;

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

@Component
public class AnimalManagement {
    @Autowired
    Animal animal;
}
```

Run again now **AutoWringDemo.java**

Output : I am a Tiger

Types of Dependency Injection with Annotations :

The process where objects use their dependent objects without a need to define or create them is called dependency injection. It's one of the core functionalities of the Spring framework.

We can inject dependent objects in three ways, using:

Spring Framework supporting 3 types of Dependency Injection .

1. Filed/Property level Injection (Only supported Via Annotations)
2. Setter Injection
3. Constructor Injection

Filed Injection:

As the name says, the dependency is injected directly in the field, with no constructor or setter needed. This is done by annotating the class member with the **@Autowired** annotation. If we define **@Autowired** on property/field name level, then Spring Injects Dependency Object directly into field.

Requirement : Address is Dependency of Employee class.

Address.java : Create as Component class

```

package com.dilip.spring;

import org.springframework.stereotype.Component;

@Component
public class Address {

    private String city;
    private int pincode;

    public Address() {
        System.out.println("Address Object Created.");
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public int getPincode() {
        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
}

```

➤ **Employee.java** : Component class with Dependency Injection.

```

package com.dilip.spring;

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

@Component
public class Employee {

    private String empName;
    private double salary;

    //Field Injection
    @Autowired
    private Address address;

    public Employee(Address address ) {
        System.out.println("Employee Object Created");
    }
}

```

```

public String getEmpName() {
    return empName;
}
public void setEmpName(String empName) {
    this.empName = empName;
}
public double getSalary() {
    return salary;
}
public void setSalary(double salary) {
    this.salary = salary;
}
public Address getAddress() {
    return address;
}
public void setAddress(Address address) {
    System.out.println("This is Setter method of Emp of Address");
    this.address = address;
}
}

```

We are Defined **@Autowired** on **Address** type field in side **Employee** class, So Spring IOC will inject **Address** Bean Object inside **Employee** Bean Object via field directly.

Testing DI:

```

package com.dilip.spring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class DiMainAppDemo {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();
        Employee emp = context.getBean(Employee.class);
        System.out.println(emp);
        System.out.println(emp.getAddress());
    }
}

```

Output:

Address Object Created.
Employee Object Created
Address Object Created.
com.dilip.spring.Employee@791f145a
com.dilip.spring.Address@38cee291

Setter Injection Overview:

Setter injection uses the setter method to inject dependency on any Spring-managed bean. Well, the Spring IOC container uses a setter method to inject dependency on any Spring-managed bean. We have to annotate the setter method with the **@Autowired** annotation.

Let's create an interface and Impl. Classes in our project.

Interface : **MessageService.java**

```
package com.dilip.setter.injection;

public interface MessageService {
    void sendMessage(String message);
}
```

Impl. Class : **EmailService.java**

```
package com.dilip.setter.injection;

import org.springframework.stereotype.Component;

@Component("emailService")
public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println(message);
    }
}
```

We have annotated **EmailService** class with **@Component** annotation so the Spring container automatically creates a Spring bean and manages its life cycle.

Impl. Class : **SMSService.java**

```
package com.dilip.setter.injection;

import org.springframework.stereotype.Component;

@Component("smsService")
public class SMSService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println(message);
    }
}
```

We have annotated **SMSService** class with **@Component** annotation so the Spring container automatically creates a Spring bean and manages its life cycle.

MessageSender.java: In setter injection, Spring will find the **@Autowired** annotation and call the setter method to inject the dependency.

```
package com.dilip.setter.injection;

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

@Component
public class MessageSender {

    private MessageService messageService;

    //At setter method level.
    @Autowired
    public void setMessageService(@Qualifier("emailService") MessageService
        messageService) {
        this.messageService = messageService;
        System.out.println("setter based dependency injection");
    }

    public void sendMessage(String message) {
        this.messageService.sendMessage(message);
    }
}
```

@Qualifier annotation is used in conjunction with **@Autowired** to avoid confusion when we have two or more beans configured for the same type.

➤ Now create a Test class to validate, dependency injection with setter Injection.

```
package com.dilip.setter.injection;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Client {
    public static void main(String[] args) {

        String message = "Hi, good morning have a nice day!.";
        ApplicationContext context = new AnnotationConfigApplicationContext();
        context.scan("com.dilip.*");
```

```

        MessageSender messageSender = context.getBean(MessageSender.class);
        messageSender.sendMessage(message);
    }
}

```

Output:

setter based dependency injection
Hi, good morning have a nice day!.

Injecting Multiple Dependencies using Setter Injection:

Let's see how to inject multiple dependencies using Setter injection. To inject multiple dependencies, we have to create multiple fields and their respective setter methods. In the below example, the **MessageSender** class has multiple setter methods to inject multiple dependencies using setter injection:

```

package com.dilip.setter.injection;

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

@Component
public class MessageSender {

    private MessageService messageService;
    private MessageService smsService;

    @Autowired
    public void setMessageService(@Qualifier("emailService") MessageService
                                  messageService) {
        this.messageService = messageService;
        System.out.println("setter based dependency injection");
    }

    @Autowired
    public void setSmsService(MessageService smsService) {
        this.smsService = smsService;
        System.out.println("setter based dependency injection 2");
    }

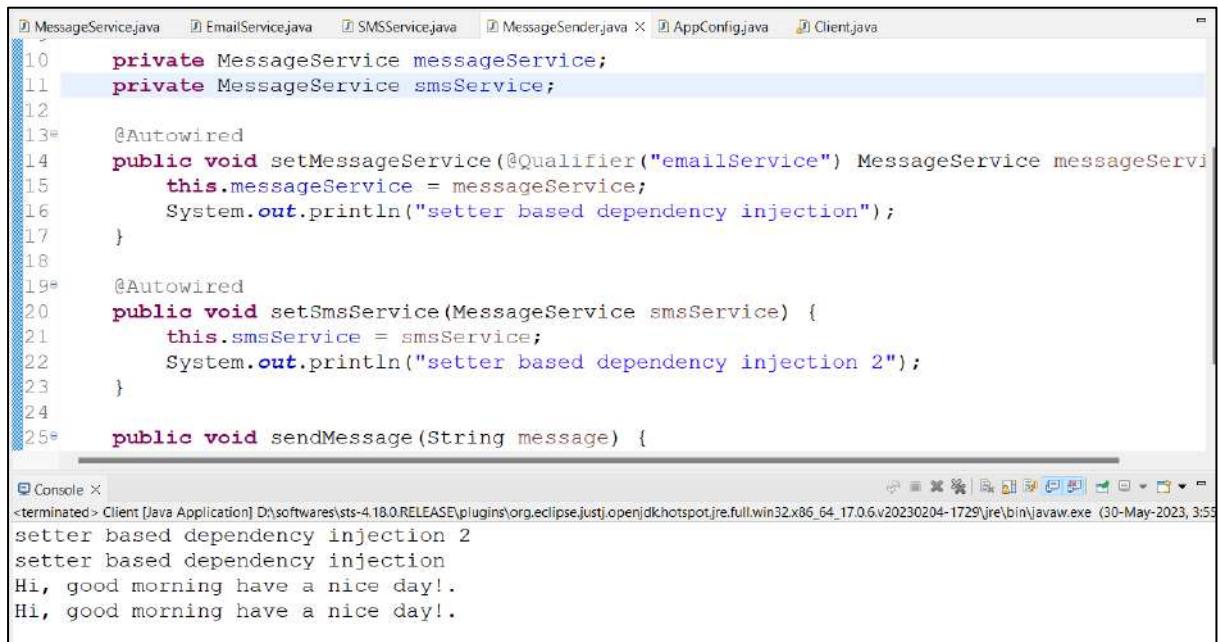
    public void sendMessage(String message) {
        this.messageService.sendMessage(message);
        this.smsService.sendMessage(message);
    }
}

```

- Now Run **Client.java**, One more time to see both Bean Objects injected or not.

Output:

```
setter based dependency injection 2
setter based dependency injection
Hi, good morning have a nice day!.
Hi, good morning have a nice day!.
```



```
10  private MessageService messageService;
11  private MessageService smsService;
12
13* @Autowired
14  public void setMessageService(@Qualifier("emailService") MessageService messageService) {
15      this.messageService = messageService;
16      System.out.println("setter based dependency injection");
17  }
18
19* @Autowired
20  public void setSmsService(MessageService smsService) {
21      this.smsService = smsService;
22      System.out.println("setter based dependency injection 2");
23  }
24
25* public void sendMessage(String message) {
```

Console X
<terminated> Client [Java Application] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.jdt.core\openjdkhotspot\jre\full\win32\x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (30-May-2023, 3:55)
setter based dependency injection 2
setter based dependency injection
Hi, good morning have a nice day!.
Hi, good morning have a nice day!.

Constructor Injection:

Constructor injection uses the constructor to inject dependency on any Spring-managed bean. Well, the Spring IOC container uses a constructor to inject dependency on any Spring-managed bean. In order to demonstrate the usage of constructor injection, let's create a few interfaces and classes.

- **MessageService.java**

```
package com.dilip.setter.injection;

public interface MessageService {
    void sendMessage(String message);
}
```

- **EmailService.java**

```
package com.dilip.setter.injection;

import org.springframework.stereotype.Component;
```

```
@Component
public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println(message);
    }
}
```

We have annotated **EmailService** class with **@Component** annotation so the Spring container automatically creates a Spring bean and manages its life cycle.

➤ **SMSService.java**

```
package com.dilip.setter.injection;

import org.springframework.stereotype.Component;

@Component("smsService")
public class SMSService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println(message);
    }
}
```

We have annotated **SMSService** class with **@Component** annotation so the Spring container automatically creates a Spring bean and manages its life cycle.

➤ **MessageSender.java**

```
package com.dilip.setter.injection;

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

@Component
public class MessageSender {

    private MessageService messageService;

    //Constructor level Auto wiring
    @Autowired
    public MessageSender(@Qualifier("emailService") MessageService
                         messageService) {
        this.messageService = messageService;
        System.out.println("constructor based dependency injection");
    }
}
```

```

    }
    public void sendMessage(String message) {
        this.messageService.sendMessage(message);
    }
}

```

➤ Now create a Configuration class: AppConfig.java

```

package com.dilip.setter.injection;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.dilip.*")
public class AppConfig {

}

```

➤ Now create a Test class to validate, dependency injection with setter Injection.

```

package com.dilip.setter.injection;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Client {

    public static void main(String[] args) {

        String message = "Hi, good morning have a nice day!.";

        ApplicationContext applicationContext =
            new AnnotationConfigApplicationContext(AppConfig.class);
        MessageSender messageSender =
            applicationContext.getBean(MessageSender.class);
        messageSender.sendMessage(message);
    }
}

```

Output:

```

constructor based dependency injection
Hi, good morning have a nice day! .

```

How to Declare Types of Autowiring with Annotations?

When we discussed of autowiring with beans XML configurations, Spring Provided 4 types autowiring configuration values for **autowire** attribute of **bean** tag.

1. no
2. byName
3. byType
4. constructor

But with annotation Bean configurations, we are not using these values directly because we are achieving same functionality with **@Autowired** and **@Qualifier** annotations directly or indirectly.

Let's compare functionalities with annotations and XML attribute values.

no : If we are not defined **@Autowired** on field/setter/constructor level, then Spring not injecting Dependency Object in side composite Object.

byType : If we define only **@Autowired** on field/setter/constructor level then, Spring injecting Dependency Object in side composite Object specific to Datatype of Bean. This works when we have only one Bean Configuration of Dependent Object.

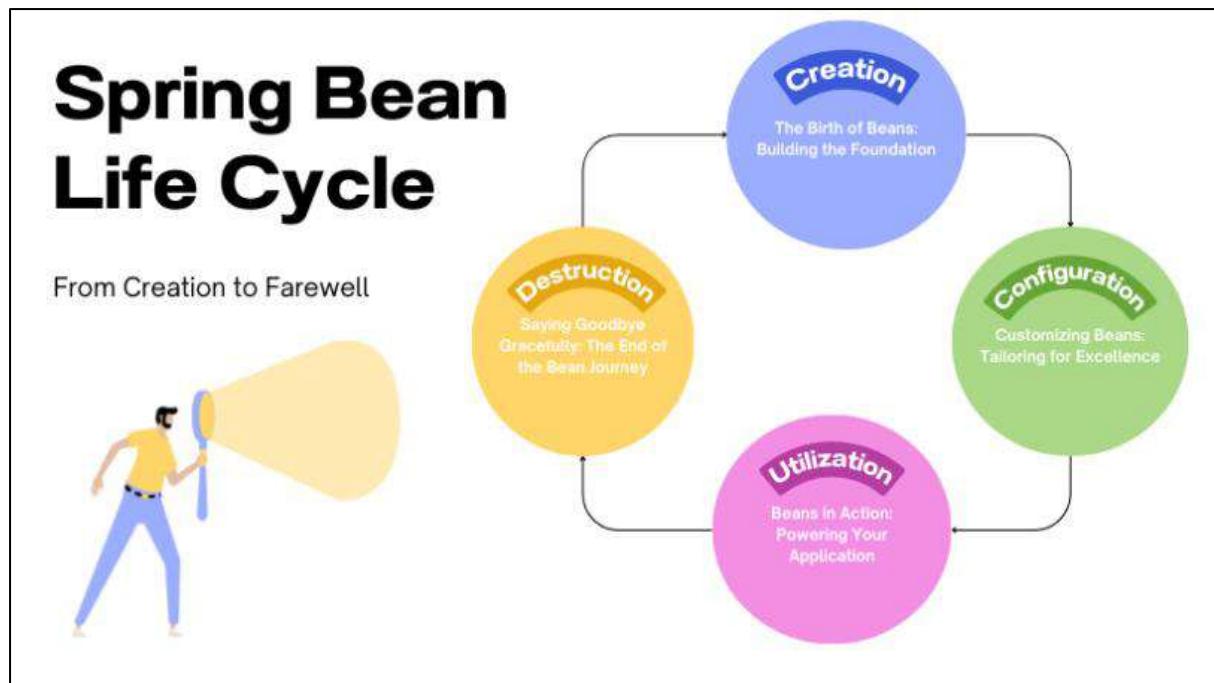
byName : If we define **@Autowired** on field/setter/constructor level along with **@Qualifier** then, Spring injecting Dependency Object in side composite Object specific to Bean ID.

constructor : when we are using **@Autowired** and **@Qualifier** along with constructor, then Spring IOC container will inject Dependency Object via constructor.

So explicitly we no need to define any autowiring type with annotation based Configurations like in XML configuration.

Bean life cycle in Java Spring:

The Spring Bean life cycle is the heartbeat of any Spring application, dictating how beans are created, initialized, and eventually destroyed. The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed.



Spring bean is a Java object managed by the Spring IoC container. These objects can be anything, from simple data holders to complex business logic components. The magic lies in Spring's ability to manage the creation, configuration, and lifecycle of these beans.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per configuration, and then dependencies are injected. After utilization of Bean Object and then finally, the bean is destroyed when the spring container is closed.

Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** methods.

Benefits of Exploring the Spring Bean Life Cycle:

- **Resource Management:** As you traverse the life cycle stages of Bean Object, you're in control of resources. This translates to efficient memory utilization and prevents resource leaks, ensuring your application runs like a well configured machine.
- **Customization:** By walking through the life cycle stages, you can inject custom logic at strategic points. This customization allows your beans to adapt to specific requirements, setting the stage for a flexible and responsive application.
- **Dependency Injection:** Understanding the stages of bean initialization also resolves the magic of dependency injection. You'll learn how beans communicate, collaborate, and share information, building a cohesive application architecture.
- **Debugging:** With a firm grasp of the life cycle, troubleshooting becomes very easy. By tracing a bean's journey through each stage, you can pinpoint issues and enhance the overall stability of your application.

Defining Bean Life Cycle Methods:

Spring allows us to attach custom actions to bean creation and destruction. We can do it by implementing the **InitializingBean** and **DisposableBean** interfaces.

InitializingBean:

InitializingBean is an interface in the Spring Framework that allows a bean to perform initialization tasks after its properties have been set. It defines a single method, **afterPropertiesSet()**, which a bean class must implement to carry out any initialization logic.

When the Spring container initializes the Bean instance, it will first set any properties configured, and then it will call the **afterPropertiesSet()** method automatically. This allows you to perform any custom initialization tasks within that method.

DisposableBean:

DisposableBean is another interface in the Spring Framework that complements the **InitializingBean**. While **InitializingBean** is used for performing initialization tasks, **DisposableBean** is used for performing cleanup or disposal tasks when a bean is being removed from the Spring container.

The **DisposableBean** interface defines a single method, **destroy()**, which a bean class must implement to carry out any cleanup logic.

When the Spring container is shutting down or removing the bean, it will call the **destroy()** method automatically, allowing you to perform any necessary cleanup tasks.

Defining Bean Life Cycle Methods with Beans:

- Create a Bean class by implementing both **InitializingBean** and **DisposableBean**.

```
package com.dilip;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;

@Component
public class Customer implements InitializingBean, DisposableBean {

    private int id;
    private String name;

    public Customer() {
        System.out.println("Customer Object is Created");
    }

    @Override
    public void afterPropertiesSet() {
        System.out.println("Customer Object is Initialized");
    }

    @Override
    public void destroy() {
        System.out.println("Customer Object is Destroyed");
    }
}
```

```

// This will be executed once instance created
@Override
public void afterPropertiesSet() throws Exception {
    System.out.println("This is Init logic from afterPropertiesSet()");
    System.out.println("Initialization logic goes here");
}

// This will be executed before container instance closing
@Override
public void destroy() throws Exception {
    System.out.println("This is destroying logic from destroy()");
    System.out.println("Cleanup logic goes here");
}

public int getId() {
    return id;
}

public void setId(int id) {
    System.out.println("Setter for injecting ID value");
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    System.out.println("Setter for injecting name value");
    this.name = name;
}
}

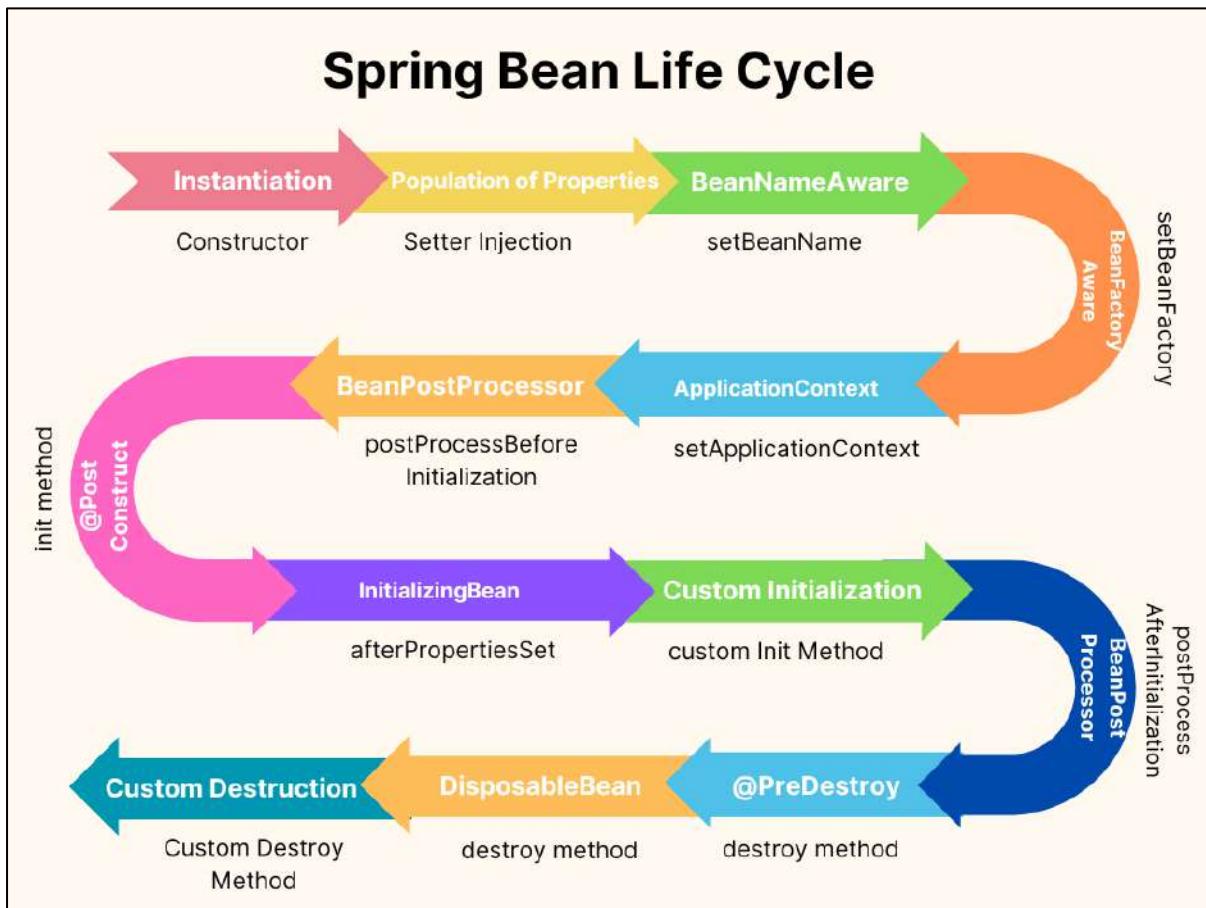
```

Spring Bean Object Life Cycle followed below steps in an order.

- Now get this Bean Object from IOC container.
- Once we created or initialized Spring Container, Container starts the process of Instantiating Bean Objects.
- Bean Object will be created/instantiated
- After Bean Object creation , Properties Values will be injected if any available.
- All Dependencies will be identified and injected
- Now Bean Initialization method i.e. **afterPropertiesSet()** method logic will be executed by IOC container one time i.e. this method will be executed once anew Object is created always by container.
- Now Bean Object is ready with all configuration values of properties and initialization values.
- We will always get current object always when we get it from container always.
- After utilization of Bean Object, when the Spring container is shutting down or removing the bean, it will call the **destroy()** method automatically for every Bean Object, allowing you to perform any necessary cleanup tasks written as part of the method.

- After executing all bean Objects **destroy()** methods then finally container got closed.

The following image shows the process flow of the Bean Object life cycle.



- Now create Spring IOC container instance and try to get Bean Object and then close the container Instance.
- Creating **Beans Configuration** class.

```
package com.dilip;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan("com.dilip")
@Configuration
public class BeansConfiguration {
```

- Now Pass above Configuration class to container.

```

package com.dilip;

import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringBeanLifeCycleDemo {
    public static void main(String[] args) {

        // Created the Container
        AnnotationConfigApplicationContext context =
            new
        AnnotationConfigApplicationContext();

        // Providing Bean Classes information to Container
        context.register(BeansConfiguration.class);

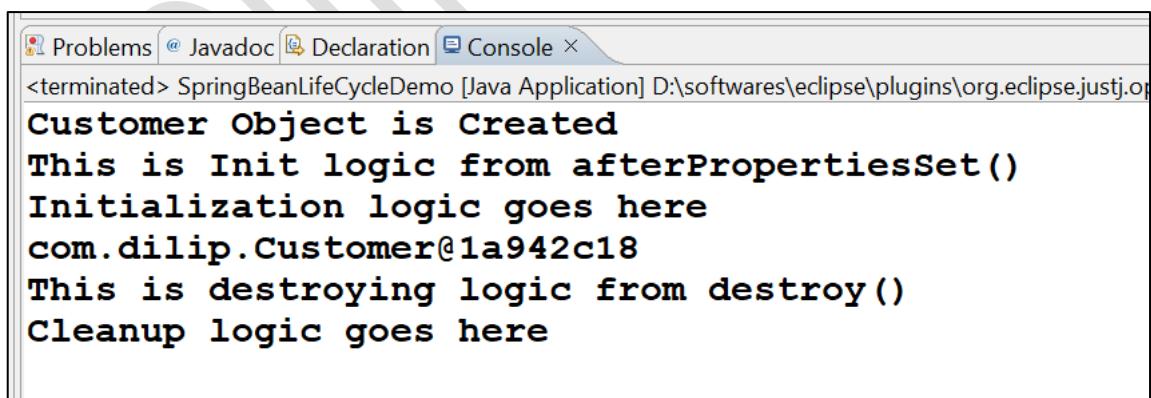
        // Create/Instantiate Bean Objects
        context.refresh();

        //Get the Bean Object from Container and Utilize it
        Customer customer = context.getBean(Customer.class);
        System.out.println(customer);

        // Closing the Container
        context.close();
    }
}

```

Output:



```

Customer Object is Created
This is Init logic from afterPropertiesSet()
Initialization logic goes here
com.dilip.Customer@1a942c18
This is destroying logic from destroy()
Cleanup logic goes here

```

- Same Process will follow by container internally for every Bean Object of class.
- Adding a Bean Method inside Configuration class for another Customer Object and then we will see same process followed for new Bean Object as usually.

BeansConfiguration.java

```

package com.dilip;

```

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan("com.dilip")
@Configuration
public class BeansConfiguration {

    @Bean(name="customer2")
    Customer getCustomer() {
        return new Customer();
    }
}

```

- Now Execute container creation and closing Lofigc again and observe initialization and destroy methods executed 2 times for 2 Customer Bean Objects creation.

```

package com.dilip;

import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringBeanLifeCycleDemo {
    public static void main(String[] args) {
        // Created the Container
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        // Providing Bean Classes information to Container
        context.register(BeansConfiguration.class);
        // Create/Instantiate Bean Objects
        context.refresh();
        // Closing the Container
        context.close();
    }
}

```

Output : For Every bean Object, executed both actions of initialization and destroy.

```

<terminated> SpringBeanLifeCycleDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.jdt.open
Customer Object is Created ✓
This is Init logic from afterPropertiesSet() ✓
Initialization logic goes here
Customer Object is Created ✓
This is Init logic from afterPropertiesSet() ✓
Initialization logic goes here
This is destroying logic from destroy() ✓
Cleanup logic goes here
This is destroying logic from destroy() ✓
Cleanup logic goes here

```

This is how we can define lifecycle methods explicitly to provide instantiation logic and destruction logic for a bean Object.

Note: Same above approach of writing Bean class with **InitializingBean** and **DisposableBean**, can be followed in Spring Beans XML configuration for a Bean class and Objects.

Question: Do we have any other ways to define life cycle methods apart from InitializingBean and DisposableBean?

Yes, we have second possibility, the **@PostConstruct** and **@PreDestroy** annotations from Java EE.

Note: Both **@PostConstruct** and **@PreDestroy** annotations are part of Java EE. Since Java EE was deprecated in Java 9, and removed in Java 11, we have to add an additional dependency in pom.xml to use these annotations.

```

<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
  <version>1.3.2</version>
</dependency>

```

Define these annotations on custom methods of Bean instantiation and destroying logic without using any Predefined Interfaces from Spring FW.

@PostConstruct:

Spring calls the methods annotated with **@PostConstruct** only once, just after the initialization of bean properties i.e. this is a replacement of **InitializingBean** and its associated abstract method implementation.

@PreDestroy:

Spring calls the methods annotated with **@PreDestroy** runs only once, just before Spring removes our bean from the application context.

Note: `@PostConstruct` and `@PreDestroy` annotated methods can have any access level, but can't be static.

Example: Bean Class: Customer.java

```
package com.dilip;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.stereotype.Component;

@Component
public class Customer {

    private int id;
    private String name;

    public Customer() {
        System.out.println("Customer Object is Created");
    }

    @PostConstruct
    public void init() {
        System.out.println("This is Init logic from init()");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("This is destroying logic from destroy()");
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

- Beans Configuration Class: BeansConfiguration.java : Created another Bean Instance

```
package com.dilip;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan("com.dilip")
@Configuration
public class BeansConfiguration {
    @Bean(name="customer2")
    Customer getCustomer() {
        return new Customer();
    }
}
```

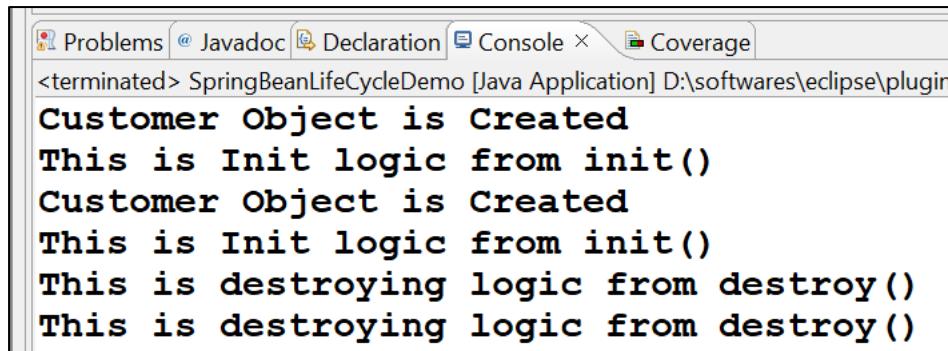
- Now Execute container creation and closing Lofigc again and observe initialization and destroy methods executed 2 times for 2 Customer Bean Objects creation.

```
package com.dilip;

import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringBeanLifeCycleDemo {
    public static void main(String[] args) {
        // Created the Container
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfiguration.class);
        context.refresh();
        // Closing the Container
        context.close();
    }
}
```

Output : For Every bean Object, executed both actions of initialization and destroy.



```

Problems @ Javadoc Declaration Console × Coverage
<terminated> SpringBeanLifeCycleDemo [Java Application] D:\softwares\eclipse\plugin
Customer Object is Created
This is Init logic from init()
Customer Object is Created
This is Init logic from init()
This is destroying logic from destroy()
This is destroying logic from destroy()

```

Question: Can we define custom methods in class for initialization and destruction of a bean object i.e. without using Pre-Defined Interfaces and Annotations ?

Yes, We can Define custom methods with user defined names of methods of both initialization and destroying actions.

Bean class: Student.java

```

package com.dilip;

public class Student {

    private int sid;

    public Student() {
        System.out.println("Student Constructor : Object Created");
    }
    public int getSid() {
        return sid;
    }
    public void setSid(int sid) {
        this.sid = sid;
    }
    // For Initialization
    public void beanInitialization() {
        System.out.println("Bean Initialization Started... ");
    }
    // For Destruction
    public void beanDestruction() {
        System.out.println("Bean Destruction Started..... ");
    }
}

```

In XML Configuration :

- Now inside Beans XML file configuration, define which method is Responsible for Bean life cycle method of initialization and destruction. Spring framework provide 2 pre-defined attributes **init-method** and **destroy-method** as part of **<bean>** tag .

- Configure custom life cycle methods in Beans.xml by using both attributes:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
  <bean id="student" class="com.dilip.Student"
        init-method="beanInitialization"
        destroy-method="beanDestruction">

  </bean>
</beans>

```

- Now Instantiate and close Spring Container and then container will execute life cycle methods as per our configuration of a bean Object.

In Annotation based Configuration :

- Spring Framework provided two attributes **initMethod** and **destroyMethod** as part of **@Bean** annotation. By using these attributes, we will define the method names as followed.

```

@Bean(initMethod = "beanInitialization", destroyMethod = "beanDestruction")
Student getStudent() {
    return new Student();
}

```

Question: Why Understand the Spring Bean Life Cycle?

Understanding the life cycle of Spring beans is like having a backstage pass to the inner workings of your Spring application. A solid grasp of the bean life cycle empowers you to effectively manage resources, configure beans, and ensure proper initialization and cleanup. With this knowledge, you can optimize your application's performance, prevent memory leaks, and implement custom logic at various stages of a bean's existence.

SpringBoot:

Spring and Spring Boot are both Java-based frameworks used for building enterprise-level applications, particularly in the realm of web and server-side development. However, they serve slightly different purposes and have some key differences:

Purpose:

Spring: Spring is a comprehensive framework for building Java-based enterprise applications. It provides a wide range of modules for various concerns such as dependency injection, aspect-oriented programming, data access, transaction management, and more. Spring is highly configurable and allows developers to pick and choose which modules they want to use in their applications.

Spring Boot: Spring Boot is built on top of the Spring framework and is designed to simplify the process of building and deploying Spring-based applications. It aims to provide an opinionated way of setting up Spring applications with minimal configuration. Spring Boot favors convention over configuration, which means it provides sensible defaults and requires less boilerplate code, making it easier to get started with Spring-based projects.

Configuration:

Spring: Spring applications typically require extensive XML or Java-based configuration to wire up components and define application behavior. Developers need to configure various aspects of the application, such as data sources, beans, and transaction management, explicitly.

Spring Boot: Spring Boot uses a "convention over configuration" approach, which means it comes with sensible defaults for many common configurations. It uses annotations and auto-configuration to automatically set up the application, reducing the need for explicit configuration. Developers can still customize the configuration when needed.

Development:

Spring: Developing with Spring often involves writing a significant amount of configuration code, XML files, and boilerplate code. It provides a lot of flexibility, but this flexibility can lead to complexity.

Spring Boot: Spring Boot encourages rapid development by providing pre-configured settings and dependencies. It's optimized for creating production-ready applications with minimal effort. Developers can focus more on writing business logic and less on infrastructure code.

Dependency Management:

Spring: Developers using Spring typically need to manage dependencies manually, which involves specifying the versions of libraries they want to use and handling conflicts.

Spring Boot: Spring Boot includes a dependency management system that simplifies the process of declaring and managing dependencies. It also provides a wide range of pre-configured dependencies for common tasks.

Embedded Containers:

Spring: Spring applications often require external web containers like Apache Tomcat or Jetty for deployment.

Spring Boot: Spring Boot includes embedded web containers like Tomcat, Jetty, or Undertow, making it easier to package and deploy applications as standalone executable JAR files.

In summary, while both Spring and Spring Boot are part of the Spring ecosystem and can be used to build Java-based applications, Spring Boot focuses on simplifying the development and deployment process by providing defaults and reducing configuration overhead, making it an attractive choice for developers looking for a faster way to build production-ready applications.

Now we are starting Spring Boot and whatever we discussed in Spring framework everything can be done in Spring boot Application because Spring Boot Internally uses Spring only.

Here are some key points to introduce Spring Boot:

Rapid Application Development: Spring Boot eliminates the need for extensive boilerplate configuration that often accompanies traditional Spring projects. It offers auto-configuration, where sensible defaults are applied based on the dependencies in your classpath. This allows developers to focus on writing business logic instead of spending time configuring various components.

Embedded Web Servers: Spring Boot includes embedded web servers, such as Tomcat, Jetty, or Undertow, which allows you to run your applications as standalone executables without requiring a separate application server. This feature simplifies deployment and distribution.

Starter POMs: Spring Boot provides a collection of "starter" dependencies, which are opinionated POMs (Project Object Model) that encapsulate common sets of dependencies for specific use cases, such as web applications, data access, security, etc. By adding these starters to your project, you automatically import the required dependencies, further reducing setup efforts.

Actuator: Spring Boot Actuator is a powerful feature that provides production-ready tools to monitor, manage, and troubleshoot your application. It exposes various endpoints, accessible via HTTP or JMX, to obtain valuable insights into your application's health, metrics, and other operational information.

Configuration Properties: Spring Boot allows you to configure your application using external properties files, YAML files, or environment variables. This decouples configuration from code, making it easier to manage application settings in different environments.

Auto-configuration: Spring Boot analyzes the classpath and the project's dependencies to automatically configure various components. Developers can override this behavior by providing their own configurations, but the auto-configuration greatly reduces the need for explicit configuration.

Overall, Spring Boot has revolutionized Java development by simplifying the creation of robust, production-ready applications. Its emphasis on convention-over-configuration, auto-configuration, and opinionated defaults makes it an excellent choice for developers seeking to build modern, scalable, and maintainable Java applications.

Starters In Spring Boot:

In Spring Boot, a "Starter" is a pre-configured set of dependencies that are commonly used together to build specific types of applications or address particular tasks. Starters simplify the process of setting up a Spring Boot application by providing a curated collection of libraries and configuration that are known to work well together for a particular use case. They help developers get started quickly without having to manually configure each individual dependency.

Here are some key points about Spring Boot Starters:

Purpose: Spring Boot Starters are designed to streamline the development process by bundling together dependencies that are commonly used for specific tasks or application types. They promote the principle of convention over configuration, reducing the amount of boilerplate code and configuration required.

Dependency Management: Starters include not only the necessary libraries but also pre-configured settings and defaults for those libraries. This simplifies dependency management, as developers don't need to worry about specifying versions or managing compatibility issues between libraries.

Spring Initializr: Spring Initializr is a web-based tool provided by the Spring team that makes it easy to generate a new Spring Boot project with the desired starters. Developers can select the starters they need, and Spring Initializr generates a project structure with the necessary dependencies and basic configuration.

Examples:

spring-boot-starter-web: This starter is commonly used for building web applications. It includes dependencies for Spring MVC, embedded web server (e.g., Tomcat, Jetty), and other web-related components.

spring-boot-starter-data-jpa: This starter is used for building applications that interact with relational databases using the Java Persistence API (JPA). It includes dependencies for Spring Data JPA, Hibernate, and a database driver.

spring-boot-starter-security: For building secure applications, this starter includes Spring Security and related dependencies for authentication and authorization.

spring-boot-starter-test: This starter includes testing libraries such as JUnit and Spring Test for writing unit and integration tests.

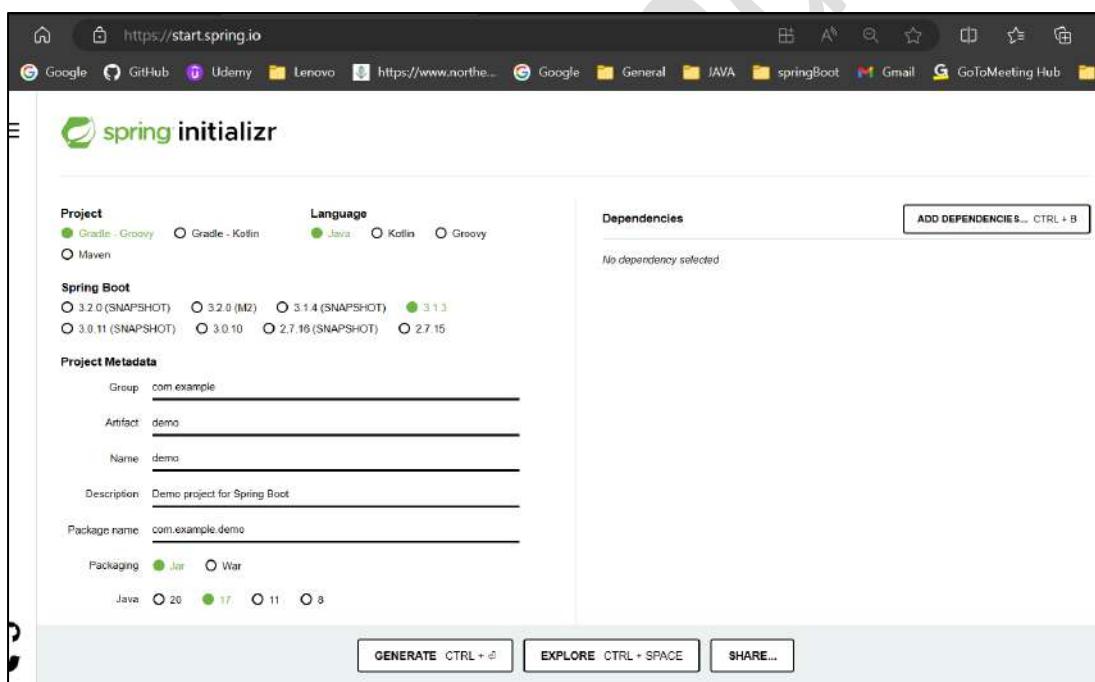
Using Spring Boot Starters can significantly accelerate the development process and improve consistency across projects by ensuring that best practices and compatible libraries are used together. It's one of the key features that makes Spring Boot a popular choice for building Java-based microservices and web applications.

Let's Create Spring application with Spring Boot.

Spring Boot is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring framework. We can create Spring Boot project mainly in 2 ways.

Using <https://start.spring.io> web portal:

- Go to <https://start.spring.io> website. This service pulls in all the dependencies you need for an application and does most of the setup for you.



- Choose **Maven** and the language **Java**, **Spring Boot Version** we want.
- Click Dependencies and select required modules.
- Now fill all details of Project Metadata like project name and package details.
- Click Generate.
- Download the resulting ZIP file, which is an archive i.e. zip file of application that is configured with your choices.
- Now you can import project inside Eclipse IDE or any other IDE's.

Creating From STS (Spring Tool Suite) IDE:

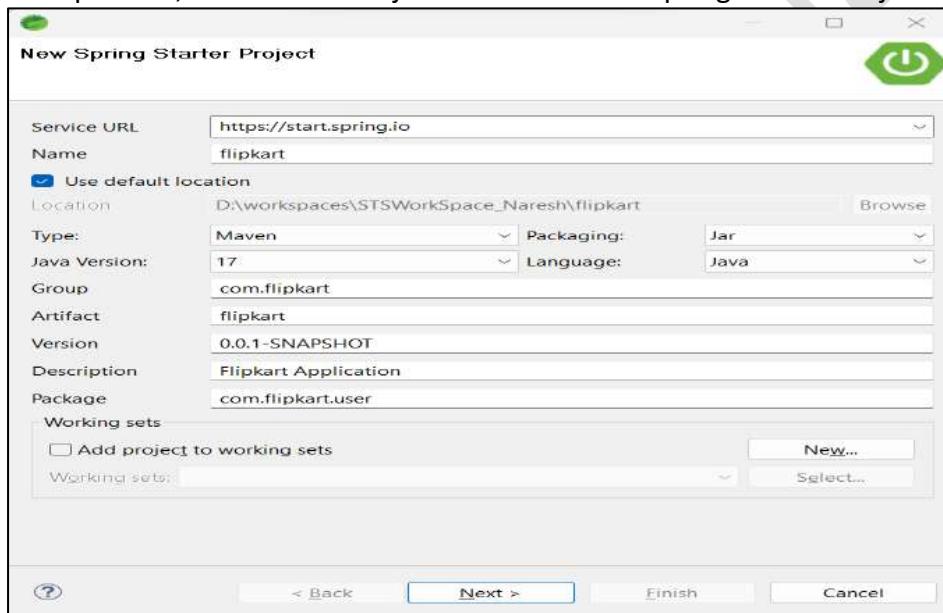
STS stands for "Spring Tool Suite." It is an integrated development environment (IDE) based on Eclipse and is specifically designed for developing applications using the Spring Framework, including Spring Boot projects. STS provides a range of tools and features that streamline the development process and enhance productivity for Spring developers.

STS is a widely used IDE for Spring development due to its rich feature set and seamless integration with the Spring Framework and related technologies. It provides a productive environment for building robust and scalable Spring applications, particularly those leveraging Spring Boot's capabilities. STS is available as a free download and is an excellent choice for developers working on Spring projects.

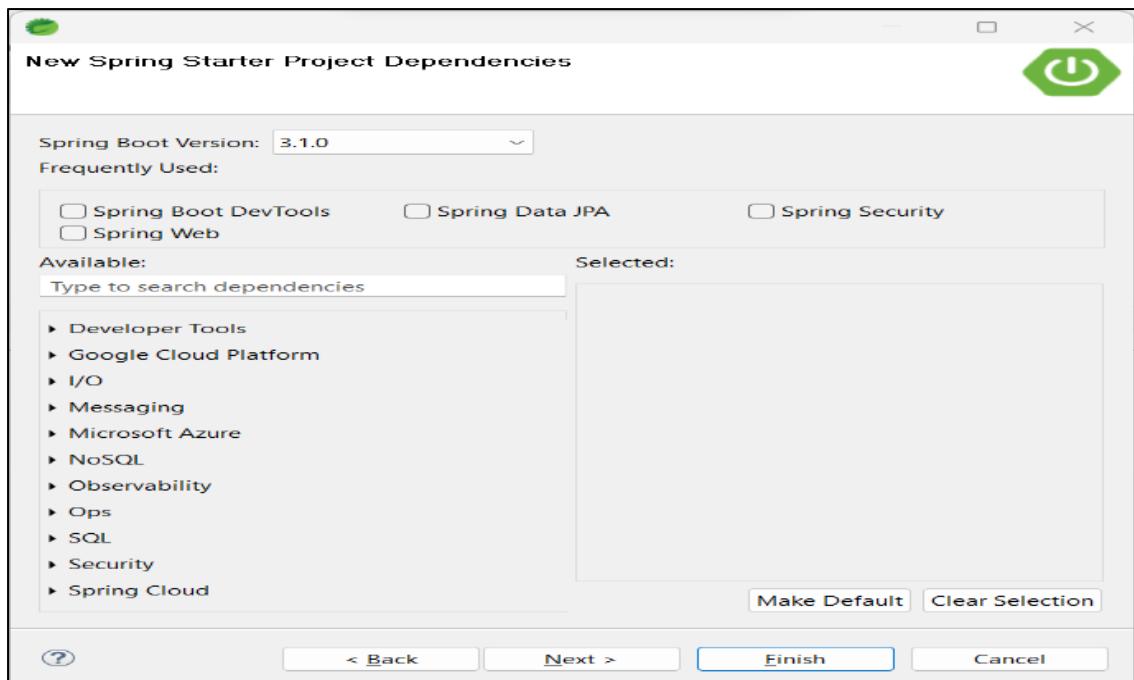
- Download STS from below link.

https://download.springsource.com/release/STS4/4.18.1.RELEASE/dist/e4.27/spring-tool-suite-4-4.18.1.RELEASE-e4.27.0-win32.win32.x86_64.self-extracting.jar

- **It will download STS as a jar file. Double click on jar, it will extract STS software.**
- Open STS, Now create Project. File -> New -> Spring Starter Project.



- **Now we can add all our dependencies, and click on finish.**



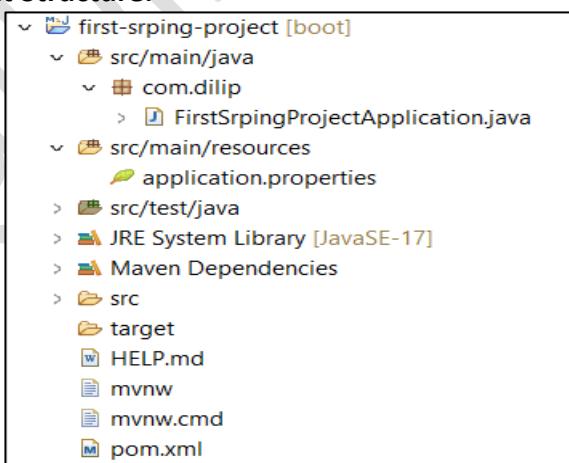
NOTE: By Default Spring Boot will support Core Module Functionalities i.e. not required to add any Dependencies in this case.

Now project created and imported to STS Directly.

If we observe, we are not added any jar files manually or externally to project like however we did in Spring Framework to work with Core Module. This is mot biggest advantage of Spring Boot Framework because in future when we are working with other modules specifically, we no need to find out jar files information and no need to add manually.

Now It's all about writing logic in project instead of thinking about configuration and project setup.

Created Project Structure:



While Project creation, By default Spring Boot will generates a main method class as shown in below.

```
1 FirstSrpingProjectApplication.java ×
2 package com.dilip;
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class FirstSrpingProjectApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(FirstSrpingProjectApplication.class, args);
10    }
11
12 }
13
14 }
```

We will discuss about this Generated class in future, but not this point because we should understand other topics before going internally.

Beans : XML based Configuration:

Create a Bean class : Student.java

```
package com.dilip.beans;

public class Student {

    private String name;
    private int studentID;
    private long mobile;

    public Student() {
        System.out.println("Student Object Created");
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getStudentID() {
        return studentID;
    }
    public void setStudentID(int studentID) {
        this.studentID = studentID;
    }
    public long getMobile() {
        return mobile;
    }
}
```

```
public void setMobile(long mobile) {
    this.mobile = mobile;
}
```

➤ Now create Benas XML file inside resources folder : beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id = "student" class="com.dilip.beans.Student">
        <property name="name" value="Dilip Singh"></property>
        <property name="studentID" value="1111"></property>
        <property name="mobile" value="8826111377"></property>
    </bean>
</beans>
```

Now Create Main Method Class:

```
package com.dilip.beans;

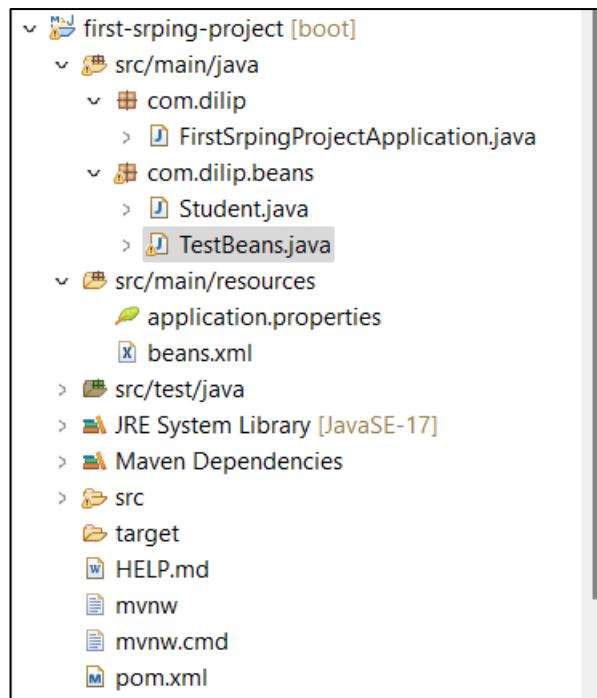
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestBeans {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
        Student s1 = (Student) context.getBean("student");
        System.out.println(s1);
    }
}
```

Now Execute Program : Output:

```
Student Object Created
com.dilip.beans.Student@1c7696c6
```

Project Folder Structure :



NOTE: If we observe above logics, we are written same code of Spring Framework completely. Means, Nothing new in spring boot w.r.to Coding/Logic point of view because Spring Boot itself a Spring Project.

So Please Practice all examples of Spring framework what we discussed previously w.r.to XML and Java Based configuration.

@Value Annotation:

This **@Value** annotation can be used for injecting values into fields in Spring-managed beans, and it can be applied at the field or constructor/method parameter level. We can read spring environment variables as well as system variables using **@Value** annotation.

Package: `org.springframework.beans.factory.annotation.Value;`

@Value - Default Value:

We can assign default value to a class property using **@Value** annotation.

```
@Value("Dilip Singh")
private String defaultName;
```

So, **defaultName** value is now **Dilip Singh**

@Value annotation argument can be a string only, but spring tries to convert it to the specified type. Below code will work fine and assign the **boolean** and **int** values to the variable.

```
@Value("true")
private boolean isJoined;

@Value("10")
```

```
private int count;
```

@Value – injecting Values from Properties File:

As part of **@Value** we should pass property name as shown in below signature along with Variables.

Syntax: **@Value("\${propertyName}")**

We will define properties in side properties/yml file, we can access them with **@Value** annotation.

application.properties:

```
bank.name=CITI BANK
bank.main.location=USA
bank.total.employees=40000
citi.db.userName=localDatabaseName
```

Now we can access any of above values in our any of Spring Bean classes with **@Value** annotation. For example, Accessing from Component class.

```
package com.bank.city;

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

@Component
public class CitiBank {

    @Value("${citi.db.userName}")
    String dbName;

    @Value("${bank.total.employees}")
    int totalEmplyeCount;

    public String getDbName() {
        return dbName;
    }

    public void setDbName(String dbName) {
        this.dbName = dbName;
    }

    public int getTotalEmplyeCount() {
        return totalEmplyeCount;
    }

    public void setTotalEmplyeCount(int totalEmplyeCount) {
        this.totalEmplyeCount = totalEmplyeCount;
    }
}
```

}

In Spring Framework Application, we have to Define an annotation **@PropertySource** on **Configuration** class level by passing properties file name externally. So then Spring framework, Annotation providing a convenient and declarative mechanism for adding a Property Sources to Spring Environment. To be used in conjunction with **@Configuration** classes.

```
package com.bank.city.config;

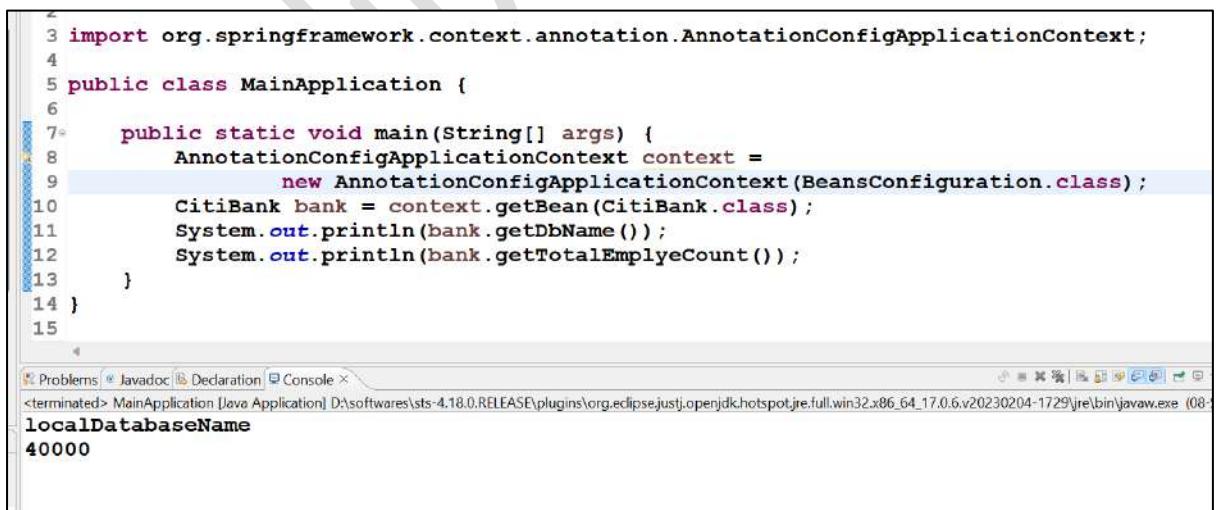
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

@ComponentScan("com.*")
@Configuration
@PropertySource("application.properties")
public class BeansConfiguration {

}
```

If it is SpringBoot Application, then we no need to add **@PropertySource**, because internally SpringBoot framework will take care of **application.properties** file by default and added to Spring environment.

Testing :



```
3 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
4
5 public class MainApplication {
6
7     public static void main(String[] args) {
8         AnnotationConfigApplicationContext context =
9             new AnnotationConfigApplicationContext(BeansConfiguration.class);
10        CitiBank bank = context.getBean(CitiBank.class);
11        System.out.println(bank.getDbName());
12        System.out.println(bank.getTotalEmployeeCount());
13    }
14 }
15
```

localDatabaseName
40000

Sometimes, we need to inject **List of values** for one property. It would be convenient to define them as comma-separated values for the single property in the properties file and to inject into an array.

List of values in Property: trainingCourses = java,python,angular

Inside Bean class, we will write below logic to inject values.

```
@Value("${trainingCourses}")
List<String> courses;
```

Map property: We can also use the `@Value` annotation to inject a Map property.

First, we'll need to define the property in the `{key: 'value'}` form in our properties file:

Note: the values in the Map must be in single quotes.

value in Property: `course.fees={java:'2000', python:'3000', oracle:'1000'}`

Now we can inject this value from the property file as a Map: Syntax change in Attribute definition of `@Value` annotation.

```
@Value("#${course.fees}")
Map<String, Integer> prices;
```

@Value with methods:

`@Value` is defined at method level, If the method has multiple arguments, then every argument value is mapped from the method annotation.

```
@Value("Test")
public void printValues(String value1, String value2){

}
```

From above, **value1 & value2** injected with value **Test**.

If we want different values for different arguments then we can use `@Value` annotation directly with the argument.

```
@Value("Test")
public void printValues(String value1, @Value("Data") String value2){

}
```

`// value1=Test, value2=Data`

Similarly, we can use `@Value` along with Constructor arguments.

@Order Annotation:

In Spring Boot, the `@Order` annotation is used to specify the order in which Spring beans should be instantiated and initialized. It's often used when you have multiple components that implement the same interface or extend the same class, and you want to

control the order in which they are injected by Spring Container as part of Collection Instances. The **@Order** annotation in Spring defines the sorting order of beans or components.

Use of @Order annotation:

Many times, we face situations where we require the beans or dependencies to be injected in a particular order. Some of the common use-cases are:

- For injecting the collection of ordered beans, components
- Sequencing the execution **CommandLineRunner** or **ApplicationRunner** in Spring Boot
- Applying the filters in an ordered way in Spring Security

Package: org.springframework.core.annotation.Order;

- Apply the **@Order** annotation to the classes you want to order. You can apply it to classes, methods, or fields, depending on your use case.

```

@Component
@Order(1)
public class MyFirstComponent {
    ...
}

@Component
@Order(2)
public class MySecondComponent {
    ...
}

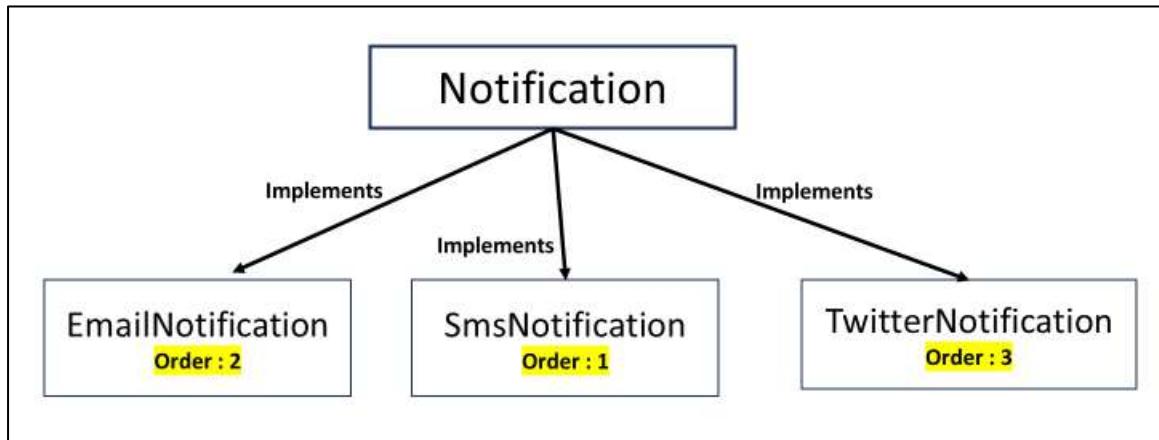
@Component
@Order(-1)
public class MyHighPriorityComponent {
    ...
}

```

In this example, **MyFirstComponent** will be injected before **MySecondComponent** because it has a lower order value (1 vs. 2) i.e. lower value takes higher precedence.

Components with a lower order value are processed before those with a higher order value. We can also use negative values if you want to indicate a higher precedence. For example, if we want a component to have the highest precedence, you can use a negative value like **-1**. If you have multiple beans with the same order value, the initialization order among them is not guaranteed.

Example: For this, I'm taking a very simple example. Here, Let's have a **Notification** interface. This interface has one method **send()**. We also have different implementations of this interface. Basically, each implementation represents a different channel or medium to send notifications.



Interface : Notification.java

```

package com.dilip.notifications;

/*
 * A Simple interface having just one method send()
 */
public interface Notification{
    void send();
}
  
```

Implemented Classes : SmsNotification.java

```

package com.dilip.notifications;

import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Component
@Order(1)
public class SmsNotification implements Notification {

    public SmsNotification() {
        System.out.println("SmsNotification Service created.");
    }
    @Override
    public void send() {
        System.out.println("Sending SMS Notification Handler");
    }
}
  
```

➤ **EmailNotification.java**

```
package com.dilip.notifications;

import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Component
@Order(2)
public class EmailNotification implements Notification {

    public EmailNotification() {
        System.out.println("EmailNotification Service Created.");
    }

    @Override
    public void send() {
        System.out.println("Sending Email Notification");
    }
}
```

➤ **TwitterNotification.java**

```
package com.dilip.notifications;

import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Component
@Order(3)
public class TwitterNotification implements Notification {

    public TwitterNotification() {
        System.out.println("TwitterNotification Service created.");
    }

    @Override
    public void send() {
        System.out.println("Sending Twitter Notification Handler");
    }
}
```

So, let's now try sending a notification to all possible channels i.e. through all available implementations of Notification. To do so, we will need **List<Notification>** to be injected.

In our ordering, we have given the highest priority to **SmsNotification** by giving **@Order(1)** as compared to others. So, that should be the first in the output. Also, we have given no **3** to **TwitterNotification** so this should come in the last always. In Below Same List Objects of Notifications component order will be followed as we defined via **@Order**.

```
package com.dilip.notifications;

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

@Component
public class AllNotifications {

    @Autowired
    private List<Notification> notifications;

    public List<Notification> getNotifications() {
        return notifications;
    }

    public void setNotifications(List<Notification> notifications) {
        this.notifications = notifications;
    }
}
```

Testing:

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.notifications.AllNotifications;

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

        //Getting All Notifications and calling send methods
        AllNotifications notifications = context.getBean(AllNotifications.class);
        notifications.getNotifications().stream().forEach(v -> v.send());
    }
}
```

Output: And here you go. It printed in the exact order we mentioned.

```
2023-09-11T12:33:03.653+05:30  INFO 11080 --- [ 
2023-09-11T12:33:03.656+05:30  INFO 11080 --- [ 
EmailNotification Service Created. 
SmsNotification Service created. 
TwitterNotification Service created. 
2023-09-11T12:33:04.090+05:30  INFO 11080 --- [ 
Sending SMS Notification 
Sending Email Notification 
Sending Twitter Notification
```

So, Spring Sorted the Notification Bean Objects while adding to List instance internally by following **@Order** value.

Note: **Ordering works at the time of injection only**

Since we have annotated with **@Order** you believe that their instantiation will also follow the same order, right?

However, that's not the case. Though the concept of **@Order** looks very simple, sometimes people get confused about its behaviour. The ordering works only at the time of injecting the beans/services, not at the time of their creation.

So, clearly we can see that the order of creation is different than order of injection from above execution.

Ordering @Bean Factory Methods:

We can also order Injecting Objects of Collections in Spring by putting **@Order** annotation on the **@Bean** methods.

Consider, there are three **@Bean** methods and each of them returns a String.

```
@Bean
@Order(2)
public String getString1() {
    return "one";
}

@Bean
@Order(3)
String getString2() {
    return "two";
}
```

```

@Bean
@Order(1)
String getString3() {
    return "three";
}

```

When we auto wire them into a **List**

```

private final List<String> stringList;

@Autowired
public FileProcessor(List<String> stringList) {
    this.stringList = stringList;
}

```

Spring sorts the beans in the specified order. Thus, printing the list we get:

```

three
one
two

```

Similarly, when we are injecting a multiple beans of same type as a collection we can set a custom sort order on the individual bean classes.

Ordered Interface:

In Spring Framework & Spring Boot, the **Ordered** interface is used to provide a way to specify the order in which objects should be processed. This interface defines a single method, **getOrder()**, which returns an integer value representing the order of the object. Objects with lower order values are processed before objects with higher order values. This is similar to **@Order** Annotation functionality.

Here's how you can use the **Ordered** interface:

1. Implement the **Ordered** interface in our component class:

```

import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;

@Component
public class MyOrderedComponent implements Ordered {

    @Override
    public int getOrder() {
        // Specify the order of this component
        // Lower values mean higher precedence
        return 1;
    }
}

```

```

    }

    // Other methods and properties of your component
}

```

In this example, the **MyOrderedComponent** class implements the **Ordered** interface and specifies an order value of **1**.

2. When Spring Boot initializes the beans, it will take into account the **getOrder()** method to determine the processing order of your component.
3. Components that implement **Ordered** interface can be used in various contexts where order matters, such as event listeners, filters, and other processing tasks.
4. To change the processing order, simply modify the return value of the **getOrder()** method. Lower values indicate higher precedence.

In this example, the **MyOrderedComponent** bean will be initialized based on the order specified in its **getOrder()** method. You can have multiple beans that implement **Ordered**, and they will be processed in order according to their **getOrder()** values. Lower values indicate higher precedence.

Runners in SpringBoot :

Runners in Spring Boot are beans that are executed after the Spring Boot application has been started. They can be used to perform any one-time initialization tasks, such as loading data, configuring components, or starting background processes.

Spring Boot provides two types of runners:

1. **ApplicationRunner** : This runner is executed after the Spring context has been loaded, but before the application has started. This means that you can use it to access any beans that have been defined in the Spring context.
2. **CommandLineRunner** : This runner is executed after the Spring context has been loaded, and after the command-line arguments have been parsed. This means that you can use it to access the command-line arguments that were passed to the application.

To implement a runner, you need to create a class that implements the appropriate interface. The **run()** method of the interface is where you will put your code that you want to execute.

CommandLineRunner:

In Spring Boot, **CommandLineRunner** is an interface that allows you to execute code after the Spring Boot application has started and the Spring Application Context has been fully initialized. We can use it to perform tasks or run code that need to be executed once the application is up and running.

Here's how you can use **CommandLineRunner** in a Spring Boot application:

1. Create a Java class that implements the **CommandLineRunner** interface. This class should override the `run()` method, where you can define the code you want to execute.

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class MyCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        // Your code to be executed after the application starts
        System.out.println("Hi from CommandLineRunner!");
        // You can put any initialization logic or tasks here.
    }
}
```

Note that **@Component** annotation is used here to make Spring automatically detect and instantiate this class as a Spring Bean.

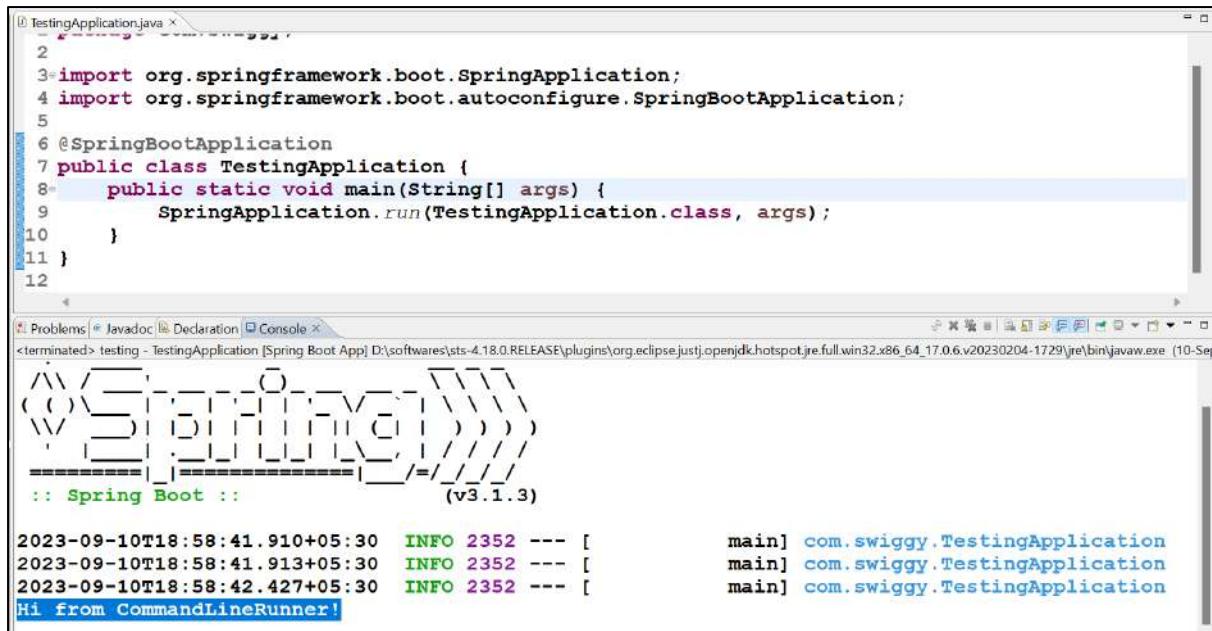
2. When we run our Spring Boot application, the `run()` method of your **CommandLineRunner** implementation will be executed automatically after the Spring context has been initialized.
3. We can have multiple **CommandLineRunner** implementations, and they will be executed in the order specified by the **@Order** annotation or the **Ordered** interface if you want to control the execution order.

Here's an example of how to run a Spring Boot application with a **CommandLineRunner**:

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

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

When we run this Spring Boot application, the `run()` method of your **MyCommandLineRunner** class (or any other **CommandLineRunner** implementations) will be executed after the application has started.



```

1 TestingApplication.java
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class TestingApplication {
8     public static void main(String[] args) {
9         SpringApplication.run(TestingApplication.class, args);
10    }
11 }
12

Problems Javadoc Declaration Console
<terminated> testing - TestingApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.jdt\openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (10-Service)
:: Spring Boot :: (v3.1.3)

2023-09-10T18:58:41.910+05:30  INFO 2352 --- [           main] com.swiggy.TestingApplication
2023-09-10T18:58:41.913+05:30  INFO 2352 --- [           main] com.swiggy.TestingApplication
2023-09-10T18:58:42.427+05:30  INFO 2352 --- [           main] com.swiggy.TestingApplication
Hi from CommandLineRunner!

```

This is a useful mechanism for tasks like database initialization, data loading, or any other setup code that should be executed once your Spring Boot application is up and running.

Similarly, we can Create Multiple **CommandLineRunner** implementation Component Classes, these all classes **run()** methods logic will be executed when we execute our Spring Boot Application. To define execution order of all **CommandLineRunner** implementation classes, we have to declare either **@Order** annotation or should implement **Ordered** interface.

Using @Order Annotation :

```

import org.springframework.boot.CommandLineRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Order(2)
@Component
public class MyCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        // Your code to be executed after the application starts
        System.out.println("Hi from CommandLineRunner!");
        // You can put any initialization logic or tasks here.
    }
}

```

Using Ordered Interface :

```

import org.springframework.boot.CommandLineRunner;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;

```

```

@Component
public class MyCommandLineRunnerTwo implements CommandLineRunner, Ordered {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("Hi from MyCommandLineRunnerTwo!");
        // You can put any initialization logic or tasks here.
    }
    @Override
    public int getOrder() {
        return 1;
    }
}

```

Testing: Run Our Spring Boot Application.

```

2023-09-10T19:30:51.173+05:30  INFO 8100 --- [ 
2023-09-10T19:30:51.176+05:30  INFO 8100 --- [ 
2023-09-10T19:30:51.684+05:30  INFO 8100 --- [ 
Hi from MyCommandLineRunner Two! ✓
Hi from CommandLineRunner!

```

ApplicationRunner:

In Spring Boot, the **ApplicationRunner** interface is part of the Spring Boot application lifecycle and is used for executing custom code after the Spring application context has been fully initialized and before the application starts running. It allows you to perform complex initialization tasks or execute code that should run just before your application starts serving requests. **ApplicationRunner** wraps the raw application arguments and exposes the **ApplicationArguments** interface, which has many convenient methods to get arguments, like **getOptionNames()** to return all the arguments' names, **getOptionValues()** to return the argument values, and raw source arguments with method **getSourceArgs()**.

In Spring Boot, both **CommandLineRunner** and **ApplicationRunner** are interfaces that allow you to execute code after the Spring application context has been fully initialized. They serve a similar purpose but differ slightly in the way they accept and handle command-line arguments.

```

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;

@Component
public class MyRunners implements ApplicationRunner {

```

```
@Override
public void run(ApplicationArguments args) throws Exception {
    System.out.println("Using ApplicationRunner");
    System.out.println("Non-option args: " + args.getNonOptionArgs());
    System.out.println("Option names: " + args.getOptionNames());
    System.out.println("Option values: " + args.getOptionValues("myOption"));
}
```

Here are the key differences between **CommandLineRunner** and **ApplicationRunner**:

Argument Handling:

CommandLineRunner: The `run()` method of **CommandLineRunner** receives an array of `String` arguments (`String... args`). These arguments are the command-line arguments passed to the application when it starts.

ApplicationRunner: The `run()` method of **ApplicationRunner** receives an **ApplicationArguments** object. This object provides a more structured way to access and work with command-line arguments. It includes methods for accessing arguments, option names, and various other features.

Use Cases:

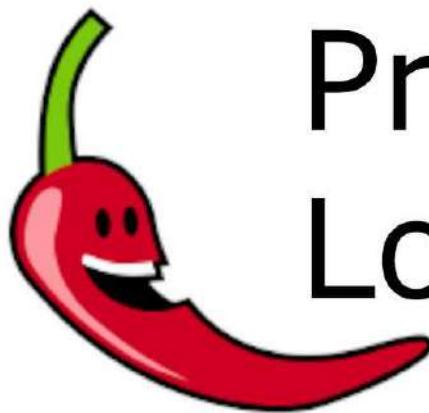
CommandLineRunner: It is suitable for simple cases where you need access to raw command-line arguments as plain strings. For example, if you want to extract specific values or flags from command-line arguments.

ApplicationRunner: It is more versatile and powerful when dealing with complex command-line argument scenarios. It provides features like option values, non-option arguments, option names, and support for argument validation. This makes it well-suited for applications with more advanced command-line parsing requirements.

Your choice between **CommandLineRunner** and **ApplicationRunner** depends on your specific requirements and the complexity of command-line argument processing in your Spring Boot application. If you need advanced features like option parsing and validation, **ApplicationRunner** is the more suitable choice. Otherwise, **CommandLineRunner** provides a simpler, more straightforward approach.

@DilipItAcademy

Project Lombok



Project Lombok

Project Lombok is a Java library that automatically plugs into your editor and build tools, spicing up your java. Never write another getter or equals method again, with one annotation your class has a fully featured builder, Automate your logging variables, and much more.

- Team Lombok

Project Lombok is a popular Java library that is designed to reduce the amount of boilerplate code that developers need to write in their Java applications. It achieves this by providing a set of annotations that can be added to Java classes to automatically generate common code structures, such as getters and setters, constructors, equals(), hashCode(), and toString() methods. Lombok helps make Java code more concise, readable, and less error-prone.

Here are some of the key features and annotations provided by Lombok:

@Getter and @Setter: These annotations generate getter and setter methods for class fields, eliminating the need to write them manually.

@ToString: Generates a `toString()` method that includes all the class's fields for easy debugging and logging.

@EqualsAndHashCode: Generates `equals()` and `hashCode()` methods based on the class's fields for object equality comparisons.

@NoArgsConstructor, @RequiredArgsConstructor, and @AllArgsConstructor: Generate constructors with no arguments, constructors with required fields, and constructors with all fields, respectively.

@Data: A shortcut annotation that combines `@Getter`, `@Setter`, `@ToString`, `@EqualsAndHashCode`, and `@RequiredArgsConstructor` into a single annotation.

@Builder: Generates a builder pattern for creating instances of a class with a fluent API, which can be particularly useful for creating complex objects with many optional parameters.

@Slf4j and other logging annotations: Simplifies the integration of logging frameworks like SLF4J by generating a logger field for the class.

Custom Annotations: Lombok allows you to create custom annotations for code generation, enabling you to automate repetitive tasks specific to your application.

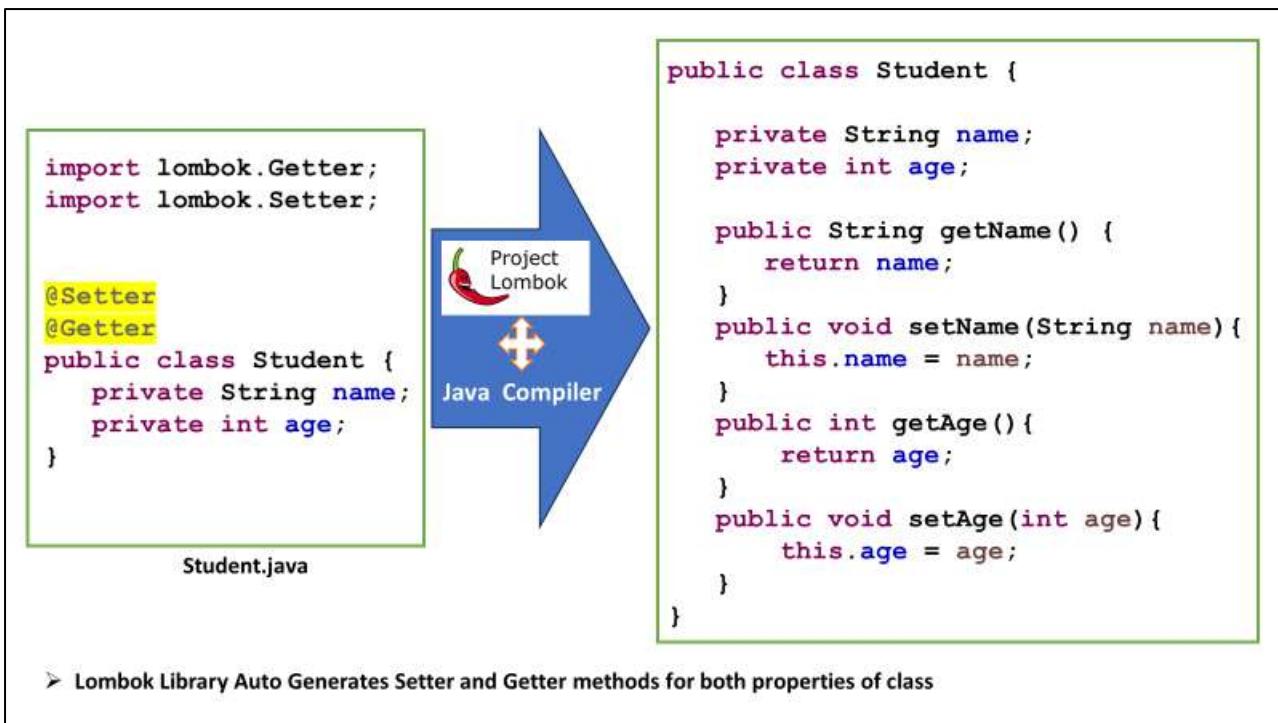
To use Lombok, you typically include it as a dependency in your project's build configuration (e.g., using Maven or Gradle) and enable annotation processing in your IDE. Lombok's annotations are processed at compile-time, which means that the generated code is automatically added to your classes during compilation. Lombok can significantly reduce the amount of boilerplate code you need to write, making your Java code cleaner and more maintainable.

Enable Lombok Annotation Processing in IDE's:

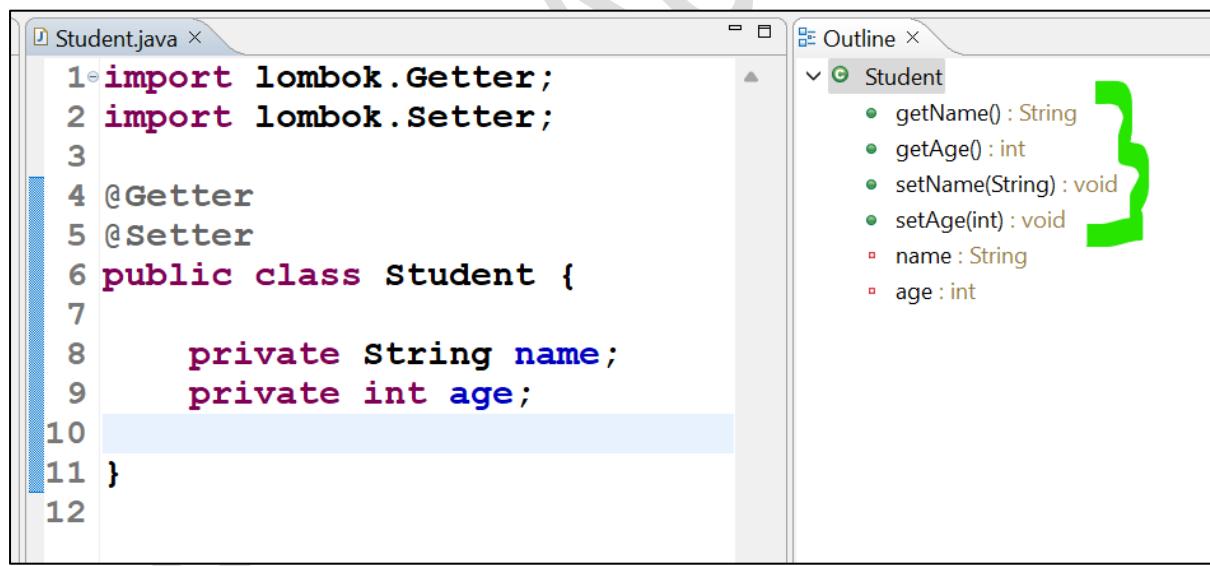
Please follow below link which will guide you to enable Lombok annotation processing.

<https://projectlombok.org/setup/eclipse>

Annotation Processing by Lombok Example:



Same we can see inside Outline View of a class in IDE:



- Similarly all other Annotations will be processed by Lombok from java Source Code.
- Here are some common features and annotations provided by Lombok:

@Getter and @Setter: These annotations generate getter and setter methods for class fields. You can apply them at the field or class level.

```

import lombok.Getter;
import lombok.Setter;

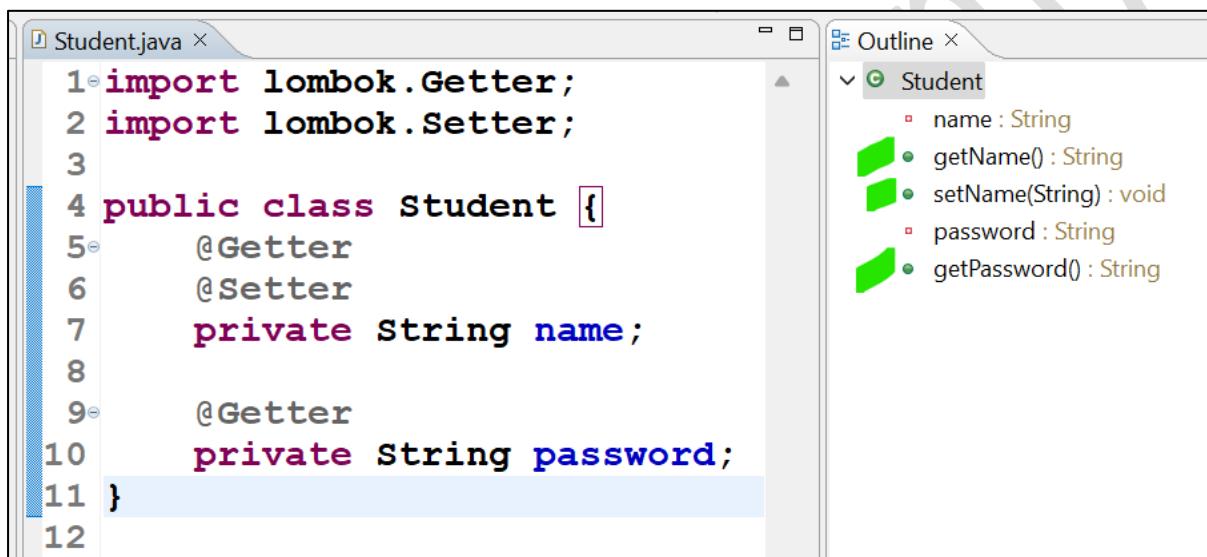
```

```
public class Student {
    @Getter
    @Setter
    private String name;

    @Getter
    private String password;
}
```

From Above Declaration,

- both setter and getter will be generated for property **name**
- Only Getter Method will Be Generated for **password**



```
Student.java
1 import lombok.Getter;
2 import lombok.Setter;
3
4 public class Student {
5     @Getter
6     @Setter
7     private String name;
8
9     @Getter
10    private String password;
11 }
12
```

Outline view:

- Student
 - name : String
 - getName() : String
 - setName(String) : void
 - password : String
 - getPassword() : String

@ToString: Generates a **toString()** method that includes all fields of the class.

```
import lombok.ToString;

@ToString
public class MyClass {
    //Properties
}
```

@EqualsAndHashCode: Generates equals and hashCode methods based on the fields of the class.

```
import lombok.EqualsAndHashCode;

@EqualsAndHashCode
public class MyClass {
    //Properties
}
```

@NoArgsConstructor, @RequiredArgsConstructor, and @AllArgsConstructor: Generate constructors with no arguments, constructors with required fields, and constructors with all fields, respectively.

```
import lombok.NoArgsConstructor;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@NoArgsConstructor
@RequiredArgsConstructor
@AllArgsConstructor
public class MyClass {
    //Properties
}
```

@Data: Combines @Getter, @Setter, @ToString, @EqualsAndHashCode, and @RequiredArgsConstructor into a single annotation.

```
import lombok.Data;

@Data
public class MyClass {
    //Properties
}
```

@Builder: Generates a builder pattern for your class, allowing you to create instances with a fluent API.

```
import lombok.Builder;

@Builder
public class MyClass {
    //Properties
}
```

We do have many other Annotations from Project Lombok. For More and clear Details please refer followed Link:

<https://objectcomputing.com/resources/publications/sett/january-2010-reducing-boilerplate-code-with-project-lombok>

Spring Boot JDBC:

Spring Boot JDBC:

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

handle transactions, and execute queries. Spring Boot builds on top of the Spring Framework and provides additional features and simplifications to make it easier to work with databases.

Here are some key aspects of Spring Boot JDBC:

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

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

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

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

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

Here are some concepts of Spring JDBC:

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

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

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

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

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

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

- Create a JdbcTemplate.
- Execute a JDBC query.

Let's see few methods of spring JdbcTemplate class.

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

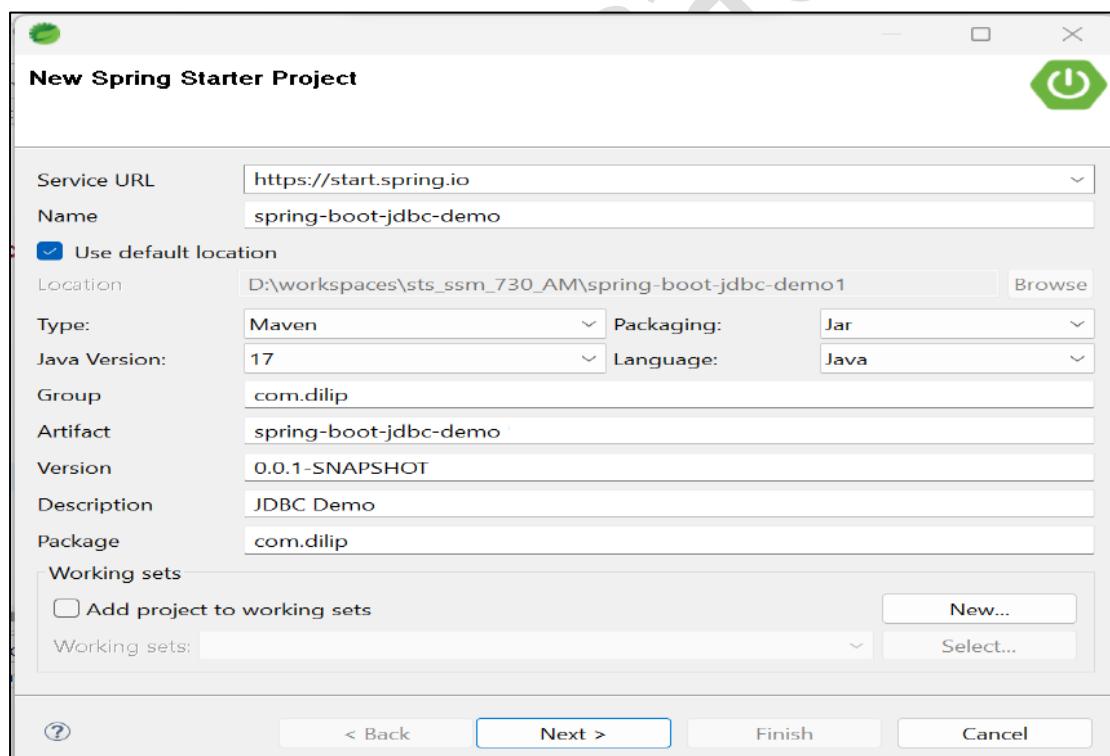
Similarly Spring / Spring Boot JDBC module provided many predefined classes and methods to perform all database operations whatever we can do with JDBC API.

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

Steps for Spring Boot JDBC Project:

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

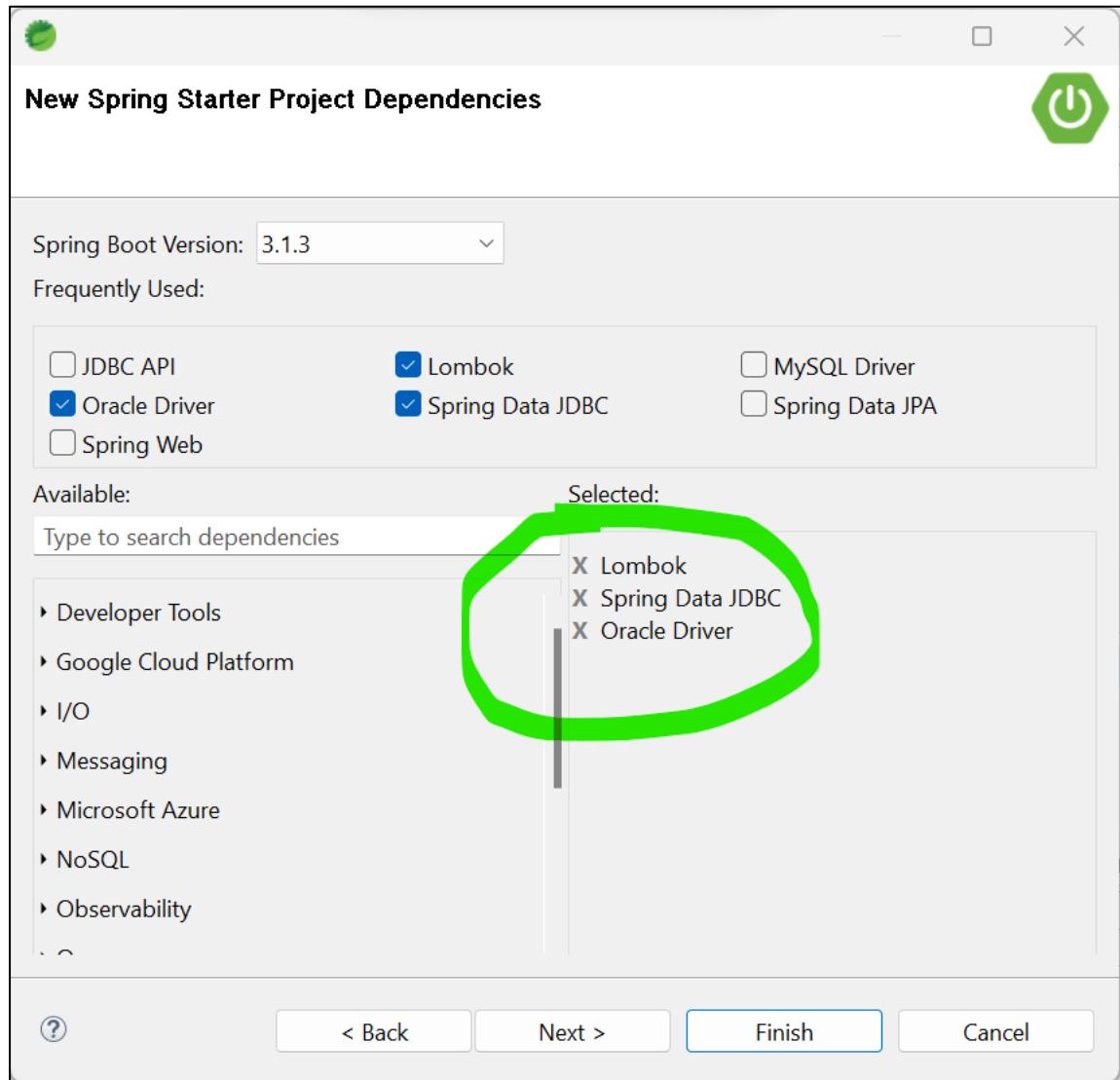
- Open STS and Create Project with Spring Boot Data JDBC dependency.



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

Oracle Driver: When developing an application that needs to interact with Oracle Database, you typically need to include the appropriate Oracle driver as a dependency in your project. The driver provides the necessary classes and methods for establishing connections, executing

SQL queries, and processing database results. How we added **ojdbc.jar** file in JDBC programming projects. Same **ojdbc.jar** file here also added by Spring Boot into project level.



Now We have to start Programming.

Requirement: **Add Product Details.**

Create Table in Database:

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

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

Note: Spring Boot reduce the JDBC driver class for most databases from the URL. If you

need to specify a specific class, you can use the **spring.datasource.driver-class-name** property.

#Oracle Database Details

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

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

```
package com.dilip;

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

@Component
public class DataBaseOperations {

    @Autowired
    JdbcTemplate jdbcTemplate;

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

- Now Call above method.

```
package com.dilip;

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

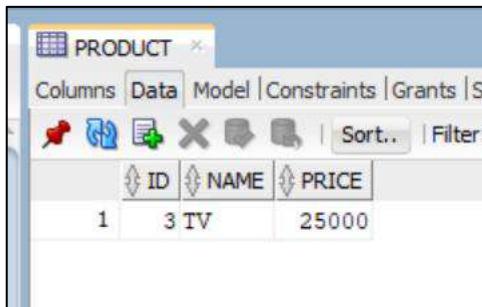
@SpringBootApplication
public class SpringBootJdbcDemoApplication {

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

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

{}

Verify in Database: Data is inserted or not.



ID	NAME	PRICE
1	3 TV	25000

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

Let's perform other database operations.

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

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

```
package com.dilip;

import lombok.Data;

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

- Create another method inside Database Operations class.

```
package com.dilip;

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

@Component
public class DataBaseOperations {

    @Autowired
    JdbcTemplate jdbcTemplate;
```

```

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

public void loadAllProducts() {

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

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

```

- Execute above logic.

```

package com.dilip;

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

@SpringBootApplication
public class SpringBootJdbcDemoApplication {

    public static void main(String[] args) {

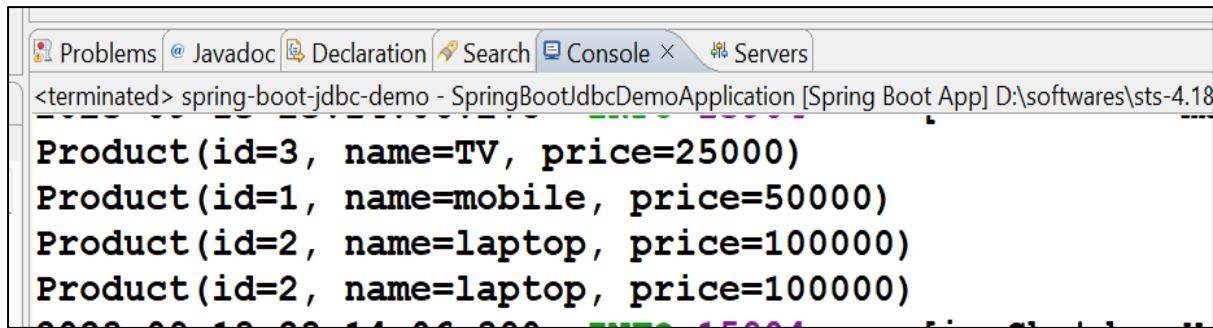
        ApplicationContext context =
            SpringApplication.run(SpringBootJdbcDemoApplication.class, args);

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

    }
}

```

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



```

Problems @ Javadoc Declaration Search Console × Servers
<terminated> spring-boot-jdbc-demo - SpringBootJdbcDemoApplication [Spring Boot App] D:\softwares\sts-4.18
Product(id=3, name=TV, price=25000)
Product(id=1, name=mobile, price=50000)
Product(id=2, name=laptop, price=100000)
Product(id=2, name=laptop, price=100000)

```

Adding few more Requirements.

```

package com.dilip;

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

@Component
public class DataBaseOperations {

    @Autowired
    JdbcTemplate jdbcTemplate;

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

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

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

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

    //delete product by id
}

```

```

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

```

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

```

package com.dilip;

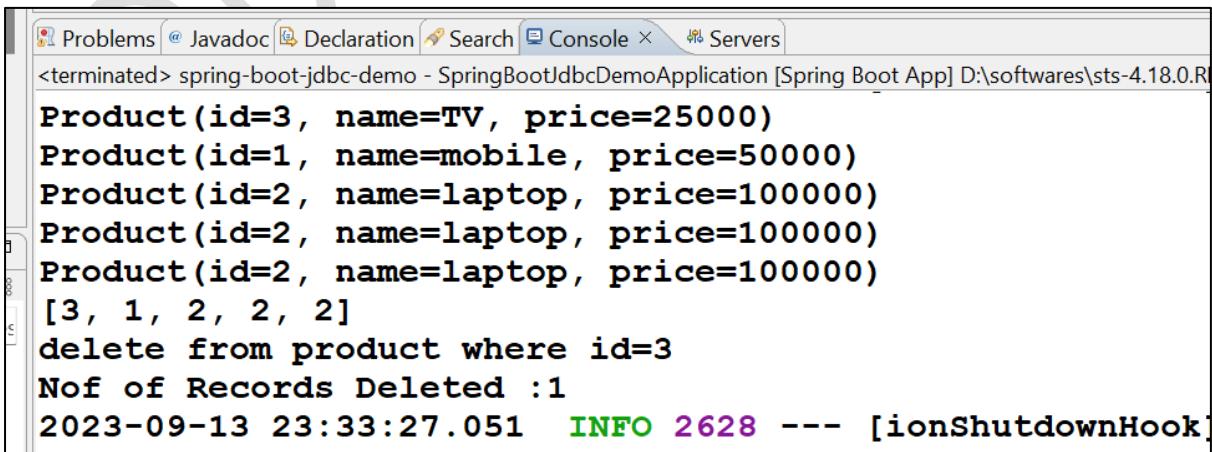
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringBootJdbcDemoApplication {

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

```

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

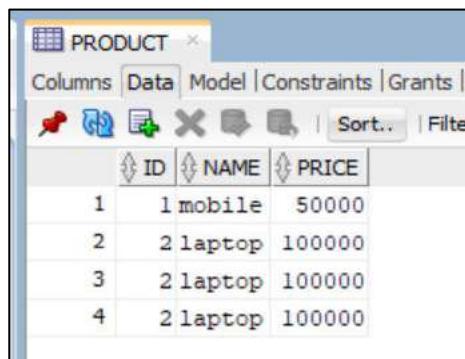


```

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

```

Inside Database: Product Id with 3 is deleted.



ID	NAME	PRICE
1	1 mobile	50000
2	2 laptop	100000
3	2 laptop	100000
4	2 laptop	100000

Positional Parameters in Query :

Requirement : Update Product Details with Product Id.

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

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

```
package com.dilip;

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

@Component
public class DataBaseOperations {

    @Autowired
    JdbcTemplate jdbcTemplate;

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

Testing:

```
package com.dilip;

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

```

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

```

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

Now Understand If we want to do same in Spring:

Steps for Spring JDBC Project:

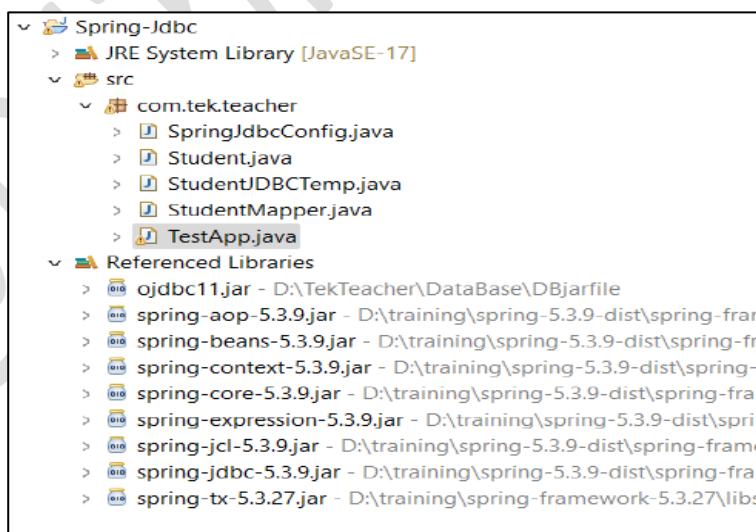
Create a Project with Spring JDBC module Functionalities: Perform below Operations on Student Table.

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

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

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

Project Structure: for jar files reference



- **Please Create Configuration class for Configuring JdbcTemplate Bean Object with DataSource Properties. So, Let's start with some simple configuration of the data source.**
- **We are using Oracle database:**
- **The DriverManagerDataSource is used to contain the information about the database such as driver class name, connection URL, username and password.**

SpringJdbcConfig.java

```
package com.tek.teacher;

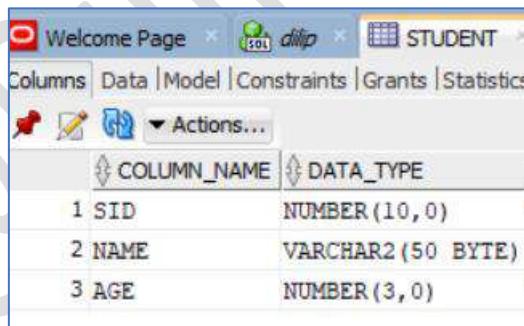
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

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

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

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

Table : Student



COLUMN_NAME	DATA_TYPE
1 SID	NUMBER(10,0)
2 NAME	VARCHAR2(50 BYTE)
3 AGE	NUMBER(3,0)

- Student.java POJO class:

```
package com.tek.teacher;

public class Student {

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

```

private Integer sid;

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

```

Mapping Query Results to Java Object:

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

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

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

StudentMapper.java

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

```

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

public class StudentMapper implements RowMapper<Student>{

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

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

        return student;
    }
}

```

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

StudentJDBCTemp.java

```

package com.tek.teacher;

import java.util.List;

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

@Component
public class StudentJDBCTemp {

    @Autowired
    private JdbcTemplate jdbcTemplateObject;

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

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

```

```

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

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

```

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

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

TestApp.java

```

package com.tek.teacher;

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

public class TestApp {

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

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

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

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

        // Load all Students
        List<Student> students = template.listStudents();
        students.stream().forEach(System.out::println);
        //students.stream().map(st -> st.getId()).forEach(System.out::println);
    }
}

```

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

Spring Data JPA

JPA and Spring Data JPA:

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

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

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

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

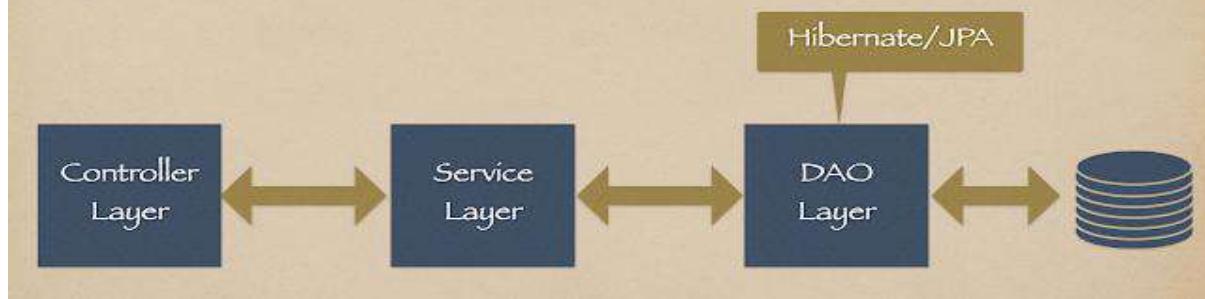
Java/Jakarta Persistence API (JPA) :

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

Features:

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

Application Architecture



ORM(Object-Relational Mapping))

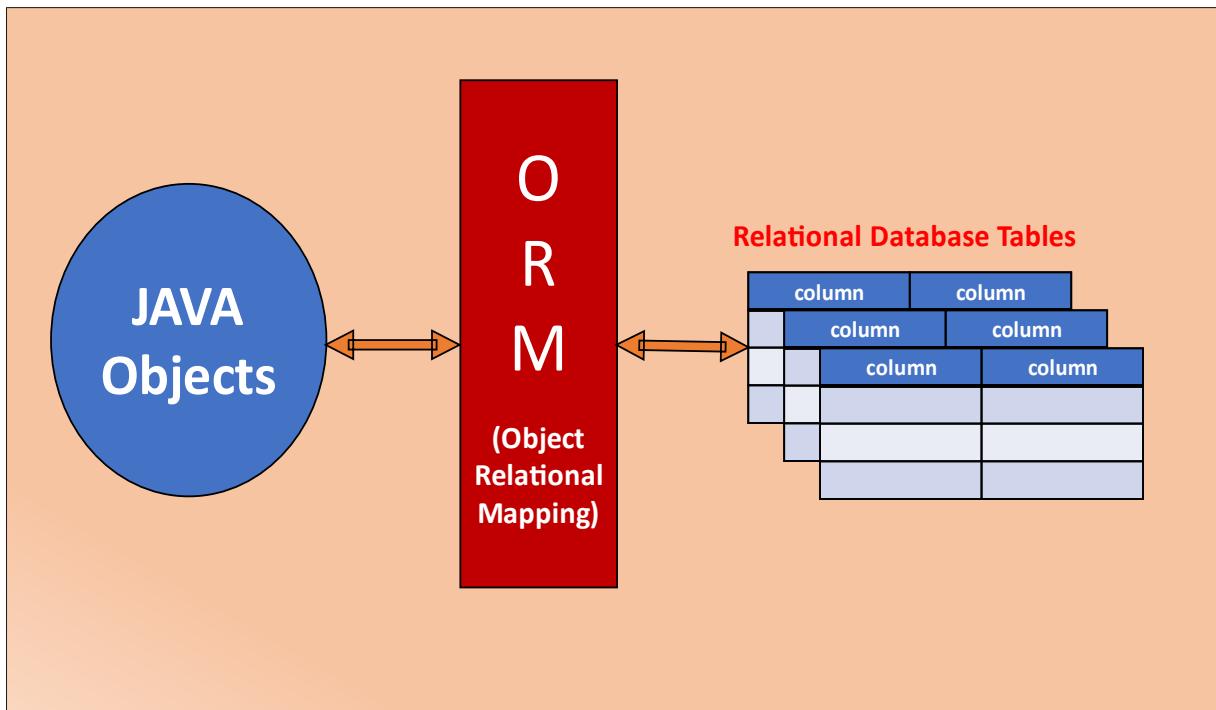
ORM(Object-Relational Mapping) is the method of querying and manipulating data from a database using an object-oriented paradigm/programming language. By using this method, we are able to interact with a relational database without having to use SQL. Object Relational Mapping (ORM) is a functionality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

We are going to implement Entity classes to map with Database Tables.

What is Entity Class?

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

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



Annotations Used In Entity Class:

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

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

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

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

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

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

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

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

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

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

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

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

Field-Level Annotation: We can place **@Id** annotation directly on a field in our entity class.

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

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

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

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

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

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

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

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

```

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

    @Id
    @Column
    private Long id;

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

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

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

```

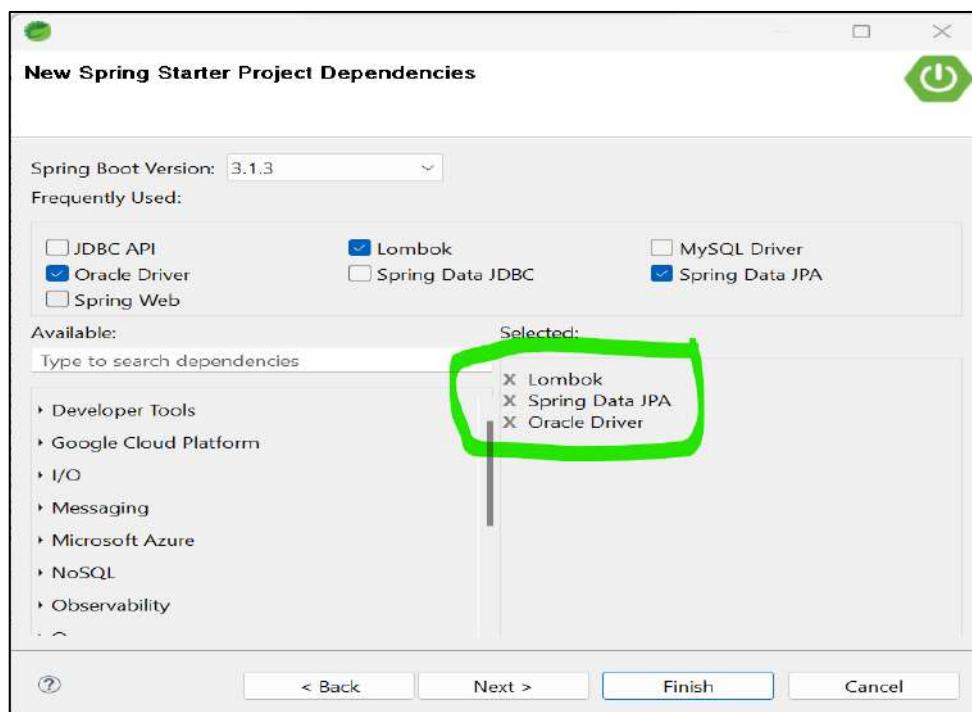
@Column(name = "product_name", length = 100, nullable = false): The **@Column** annotation is used to specify the mapping of entity class fields to table columns. In this example, it indicates that the “**name**” field should be mapped to a column named “**product_name**” in the database table. Additional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

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

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

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

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



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

application.properties:

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

➤ **Create Entity Class:**

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

Table Name : flipkart_orders

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

Entity Class: FlipakartOrder.java

```

package com.flipkart.dao;

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

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

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

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

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

    public long getOrderID() {
        return orderID;
    }

    public void setOrderID(long orderID) {
        this.orderID = orderID;
    }

    public String getProductName() {
        return productName;
    }

    public void setProductName(String productName) {
        this.productName = productName;
    }

    public float getTotalAmount() {
        return totalAmount;
    }

    public void setTotalAmount(float totalAmount) {
        this.totalAmount = totalAmount;
    }
}

```

```
}
```

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

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

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

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

```
private int pincode;
```

Spring JPA Repositories:

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

Spring JPA Provided 2 Types of Repositories

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

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

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

Create Spring JPA Repository:

```
package com.flipkart.dao;

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

```
public interface FlipkartOrderRepository extends JpaRepository<FlipkartOrder, Long> {
}
```

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

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

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

Benefits of using Spring JPA Repositories:

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

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

Here's how repositories work in Spring Data JPA:

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

- **Use Repository Methods:** You can now use the methods defined in the repository interface to perform database operations.

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

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

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

```
package com.flipkart.dao;

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

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

    @Autowired
    FlipkartOrderRepository flipkartOrderRepository;

    public void addOrderDetails(FlipkartOrder order) {
        flipkartOrderRepository.save(order);
    }
}
```

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

```
package com.flipkart;

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

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

        // Created Entity Object
        FlipkartOrder order = new FlipkartOrder();
    }
}
```

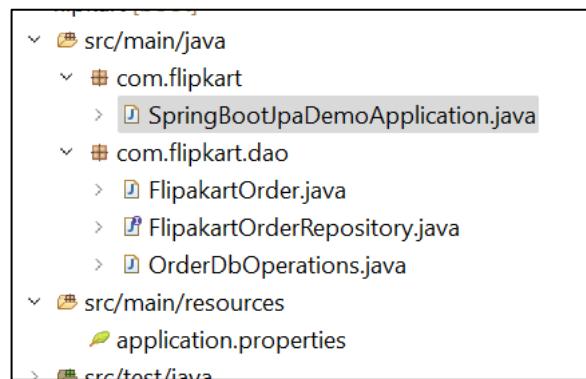
```

        order.setOrderID(9988);
        order.setProductName("Book");
        order.setTotalAmount(333.00f);

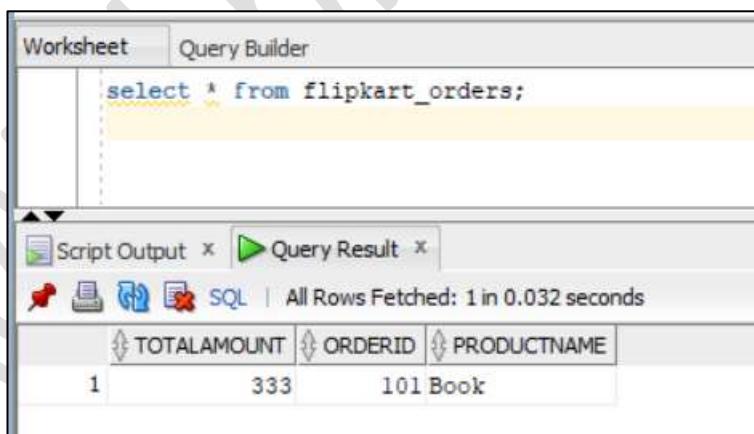
        // Pass Entity Object to Repository Method
        OrderDbOperations dbOperation
            = context.getBean(OrderDbOperations.class);
        dbOperation.addOrderDetails(order);
    }
}

```

Project Structure:



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



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

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

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

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

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

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

true: Hibernate will log all SQL statements to the console.

false: Hibernate will not log any SQL statements.

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

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

Add below Property In : application.properties

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

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

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

Add Logic in : OrderDbOperations.java

```
package com.flipkart.dao;

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

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

    @Autowired
    FlipkartOrderRepository flipkartOrderRepository;
```

```

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

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

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

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

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

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

flipkartOrderRepository.saveAll(orders);
}
}

```

➤ Execute above logic.

```

package com.flipkart;

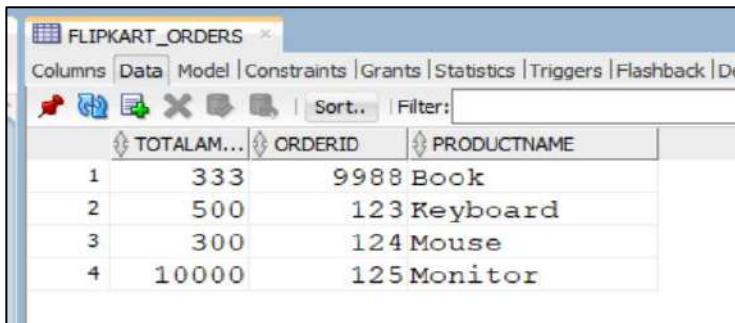
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.flipkart.dao.OrderDbOperations;

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

        OrderDbOperations dbOperation = context.getBean(OrderDbOperations.class);
        dbOperation.addListOfOrders();
    }
}

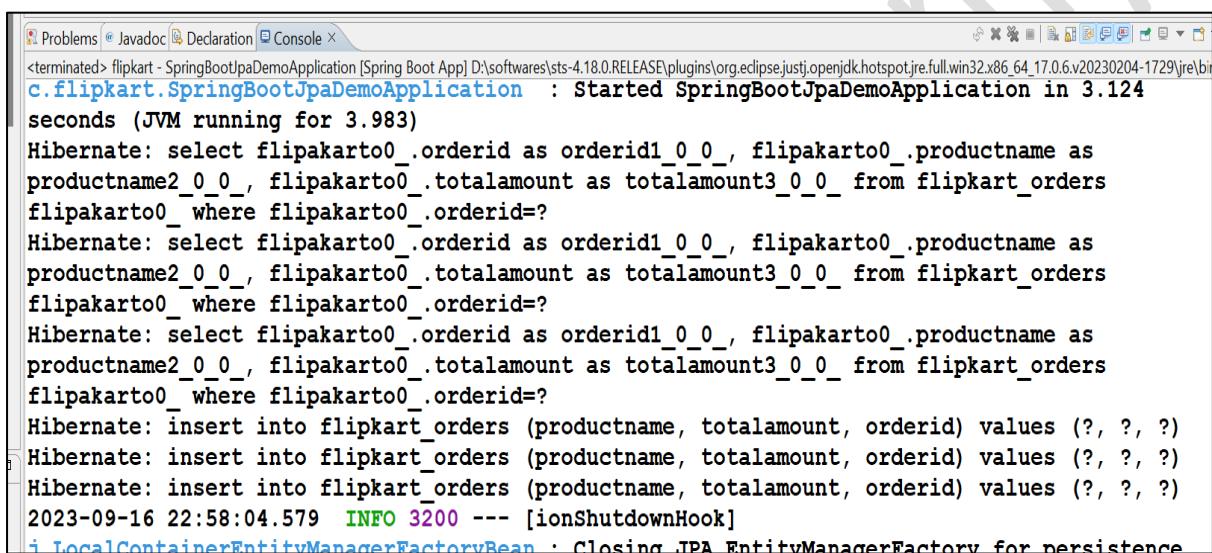
```

- Now Three records will be inserted in database.



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

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



```

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

```

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

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

```

public void loadAllOrders() {
    List<FlipkartOrder> allOrders = flipkartOrderRepository.findAll();
    for(FlipkartOrder order : allOrders) {
        System.out.println(order.getOrderId());
        System.out.println(order.getProductName());
        System.out.println(order.getTotalAmount());
    }
}

```

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

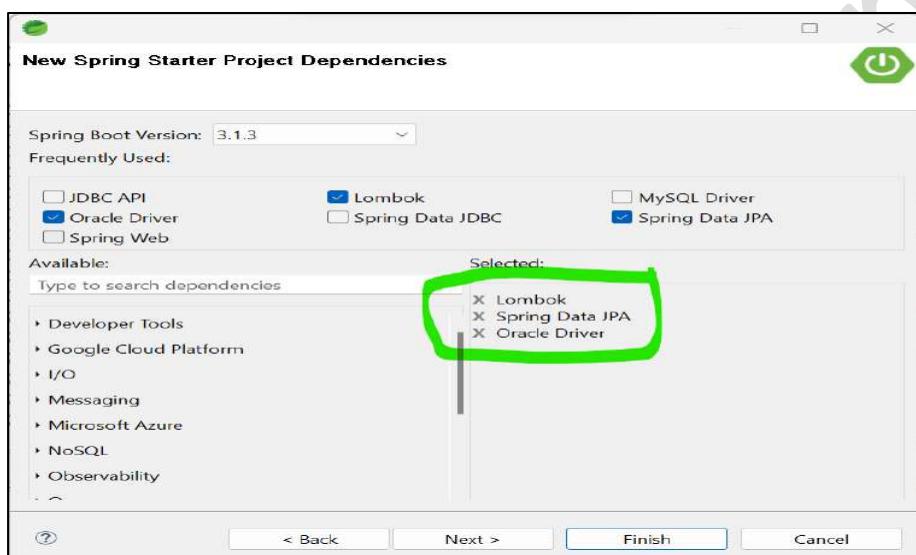
- **New Requirement** : Let's Have Patient Information as follows.

- Name

- Age
- Gender
- Contact Number
- Email Id

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

1. Create Spring Boot Project with JPA and Oracle Driver



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

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

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

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

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

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

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

3. Create Entity Class

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

```
package com.dilip.dao;

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

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

    @Column
    private String name;

    @Column
    private int age;

    @Column
    private String gender;

    @Column
    private String contact;
```

```
{}
```

4. Create A JPA Repository Now : PatientRepository.java

```
package com.dilip.dao;

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

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

}
```

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

```
package com.dilip.dao;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

}
```

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

Requirement : Add Single Patient Details

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

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

Add Below Method in PatientOperations.java:

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

Now Test it : From Main Class : PatientApplication.java

```
package com.dilip;

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

```

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

@SpringBootApplication
public class PatientApplication {

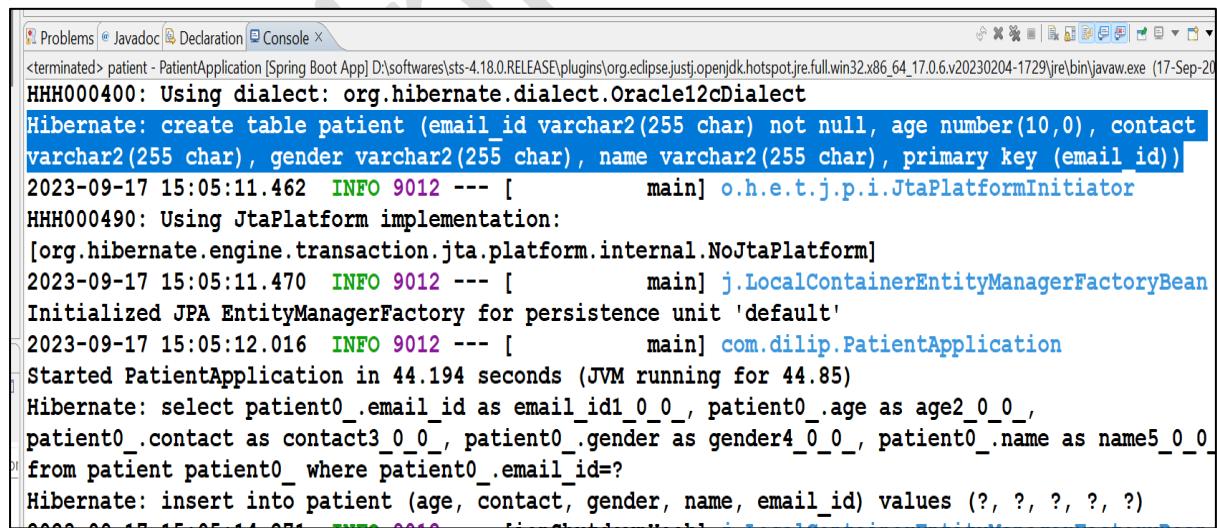
    public static void main(String[] args) {
        ApplicationContext context
            = SpringApplication.run(PatientApplication.class, args);
        PatientOperations ops = context.getBean(PatientOperations.class);

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

        ops.addPatient(p);
    }
}

```

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



```

Problems Javadoc Declaration Console
<terminated> patient - PatientApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (17-Sep-2023)
HHH000400: Using dialect: org.hibernate.dialect.Oracle12cDialect
Hibernate: create table patient (email_id varchar2(255 char) not null, age number(10,0), contact varchar2(255 char), gender varchar2(255 char), name varchar2(255 char), primary key (email_id))
2023-09-17 15:05:11.462  INFO 9012 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator
HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
2023-09-17 15:05:11.470  INFO 9012 --- [           main] j.LocalContainerEntityManagerFactoryBean
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-09-17 15:05:12.016  INFO 9012 --- [           main] com.dilip.PatientApplication
Started PatientApplication in 44.194 seconds (JVM running for 44.85)
Hibernate: select patient0_.email_id as email_id1_0_0_, patient0_.age as age2_0_0_, patient0_.contact as contact3_0_0_, patient0_.gender as gender4_0_0_, patient0_.name as name5_0_0_ from patient patient0_ where patient0_.email_id=?
Hibernate: insert into patient (age, contact, gender, name, email_id) values (?, ?, ?, ?, ?)

```

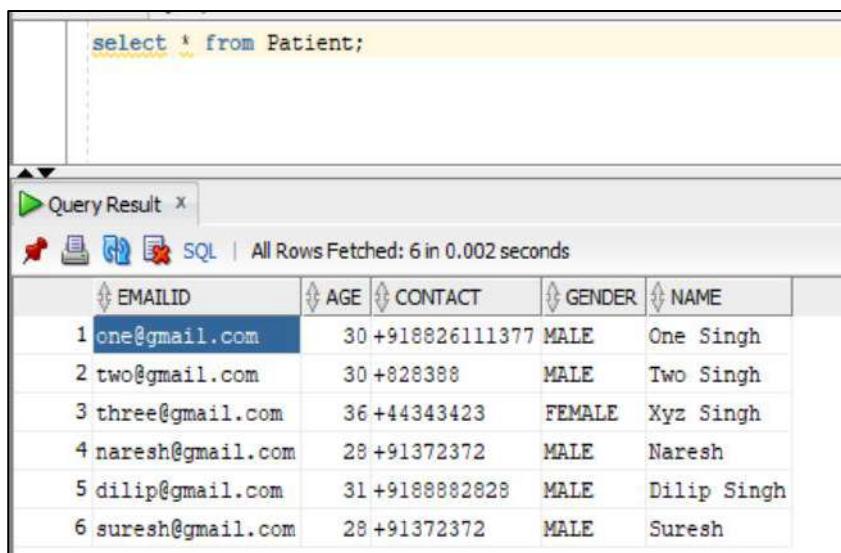
Requirement : Update Patient Details

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

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

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

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



EMAILID	AGE	CONTACT	GENDER	NAME
1	30	+918826111377	MALE	One Singh
2	30	+828388	MALE	Two Singh
3	36	+44343423	FEMALE	Xyz Singh
4	28	+91372372	MALE	Naresh
5	31	+9188882828	MALE	Dilip Singh
6	28	+91372372	MALE	Suresh

Add Below Method in PatientOperations.java:

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

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

```
package com.dilip;

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

@SpringBootApplication
public class PatientApplication {
```

```

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

```

Verify In DB :

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

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

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

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

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

Add Below Method in PatientOperations.java:

```

package com.dilip.dao;

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

@Component
public class PatientOperations {

```

```

@.Autowired
PatientRepository repository;

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

public void deletePatient(String email) {
    repository.deleteById(email);
}

}

```

Testing from Main Class:

```

package com.dilip;

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

@SpringBootApplication
public class PatientApplication {

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

```

Before Execution/Deletion:

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

After Execution/Deletion:



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

Requirement: Get Patient Details Based on Email Id.

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

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

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

```
package com.dilip.dao;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

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

    public void deletePatient(String email) {
        repository.deleteById(email);
    }

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

Testing from Main Class:

```
package com.dilip;

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

```

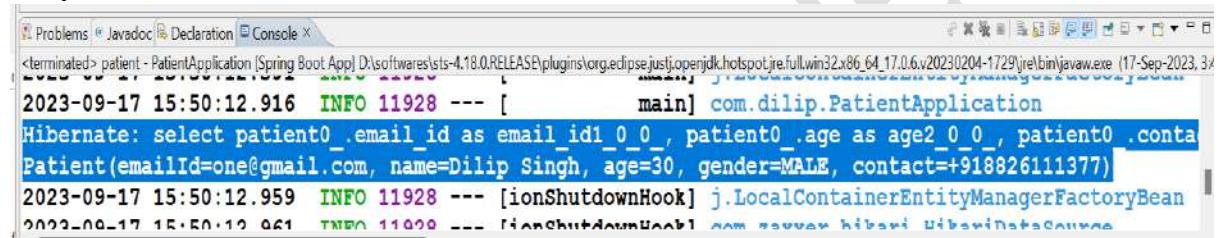
import org.springframework.context.ApplicationContext;
import com.dilip.dao.Patient;
import com.dilip.dao.PatientOperations;

@SpringBootApplication
public class PatientApplication {

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

```

Output in Console:



```

2023-09-17 15:50:12.916 INFO 11928 --- [           main] com.dilip.PatientApplication
Hibernate: select patient0_.email_id as email_id1_0_0_, patient0_.age as age2_0_0_, patient0_.conta
Patient(emailId=one@gmail.com, name=Dilip Singh, age=30, gender=MALE, contact=+918826111377)
2023-09-17 15:50:12.959 INFO 11928 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean
2023-09-17 15:50:12.961 INFO 11928 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource

```

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

For example :

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

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

Spring Data JPA Custom Query Repository Methods:

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

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

```

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

```

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

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

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

Requirement: **Get Details of Patients by Age i.e. Single Column.**

Result we will get More than One record i.e. List of Entity Objects. So return type is `List<Patient>`

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

```
package com.dilip.repository;

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

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

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

```
package com.dilip.operations;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

    // Non- primary Key column
    public List<Patient> fetchByAge(int age) {

```

```

        return repository.findByAge(age);
    }
}

```

Step 3: Now Test It from Main Class.

```

package com.dilip;

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

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

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

```

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

```

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

```

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

Fetching Data with Multiple Columns:

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

Requirement: Fetch Data with Age and Gender Columns.

- Age is 28
- Gender is Female

Step 1: Create a Custom method inside Repository.

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

```
package com.dilip.repository;

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

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

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

```
package com.dilip.operations;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

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

Step 3: Now Test It from Main Class.

```
package com.dilip;

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

```

import com.dilip.dao.PatientOperations;

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

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

```

Table Data:

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

Expected Output:

```

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

```

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

```

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

```

Both query methods will return only first 3 records.

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

List of keywords used to write custom repository methods:

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

Some of the examples for Method Names formations:

```
public List<Student> findFirst3ByName(String name);
public List<Student> findByNameIs(String name);
public List<Student> findByNameEquals(String name);
public List<Student> findByNameRollNumber(String rollNumber);
public List<Student> findByNameUniversity(String university);
public List<Student> findByNameAndRollNumber(String name, String rollNumber);
public List<Student> findByNameIn(List<String> rollNumbers);
public List<Student> findByNameNotIn(List<String> rollNumbers);
public List<Student> findByNameBetween(String start, String end);
public List<Student> findByNameNot(String name);
public List<Student> findByNameContainingIgnoreCase(String name);
public List<Student> findByNameLike(String name);
public List<Student> findByNameGreaterThanOrEqual(String rollnumber);
public List<Student> findByNameLessThanOrEqual(String rollnumber);
```

@GeneratedValue Annotation:

In Java Persistence API (JPA), the **@GeneratedValue** annotation is used to specify how primary key values for database entities should be generated. This annotation is typically used in conjunction with the **@Id** annotation, which marks a field or property as the primary key of an entity class. The **@GeneratedValue** annotation provides options for automatically generating primary key values when inserting records into a database table.

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

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

strategy:

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

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

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

2. **GenerationType.IDENTITY:** The primary key value is generated by the database system itself (e.g., auto-increment in MySQL or identity columns in SQL Server, SERIAL in PostgreSQL).
3. **GenerationType.SEQUENCE:** The primary key value is generated using a database sequence. Tells JPA to use a database sequence for ID generation. This strategy does currently not provide full portability. Sequences are supported by Oracle and PostgreSQL. When this value is used then **generator** filed is mandatory to specify the generator.
4. **GenerationType.TABLE:** The primary key value is generated using a database table to maintain unique key values. Tells JPA to use a separate table for ID generation. This strategy provides full portability. When this value is used then generator filed is mandatory to specify the generator.
5. **GenerationType.UUID:** Jakarta EE 10 now adds the GenerationType for a **UUID**, so that we can use Universally Unique Identifiers (UUIDs) as the primary key values. Using the GenerationType.UUID strategy, This is the easiest way to generate **UUID** values. Simply annotate the primary key field with the **@GeneratedValue** annotation and set the strategy attribute to **GenerationType.UUID**. The persistence provider will automatically generate a UUID value for the primary key column.

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

Examples For All Strategies:

GenerationType.AUTO:

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

When to Use GenerationType.AUTO?

The **GenerationType.AUTO** strategy is a convenient choice for most applications because it eliminates the need to explicitly specify a generation strategy of primary key values. It is particularly useful when you are using a database that supports both

GenerationType.IDENTITY and **GenerationType.SEQUENCE**, as the persistence provider will automatically select the most efficient strategy for your database.

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

Benefits of **GenerationType.AUTO**

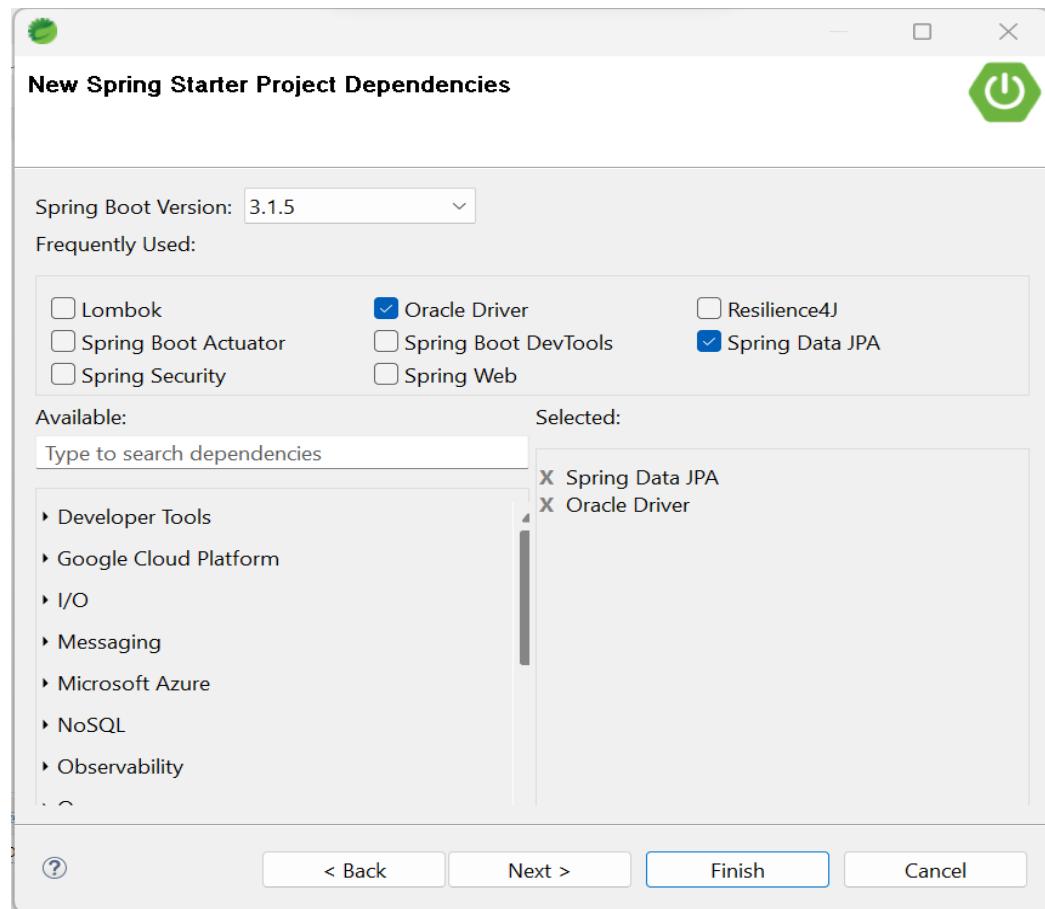
- **Convenience:** It eliminates the need to explicitly specify a generation strategy.
- **Automatic selection:** It selects the most appropriate strategy for the underlying database.
- **Compatibility:** It is compatible with a wide range of databases.

Limitations of **GenerationType.AUTO**

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

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

- Create Spring Boot Data JPA project



➤ Now Add Database and JPA properties in application.properties file:

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

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

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

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

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

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

@Entity
```

```

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

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

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

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

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

```

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

```

package com.tek.teacher.data;

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

public interface PatientRepository extends JpaRepository<Patient, Long> {
}

```

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

```

package com.tek.teacher.data;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

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

```

- Call/Execute above method for persisting data.

```

package com.tek.teacher;

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

@SpringBootApplication
public class SpringBootJpaGeneartedvalueApplication {

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

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

```

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

```

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

```

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

Verify Data Inside Table now.

```
select * from patient_details;
```

Query Result x

SQL | All Rows Fetched: 1 in 0.004 seconds

PATIENT_ID	PATIENT_AGE	PATIENT_NAME
1	44	Dilip Singh

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

```

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

```

Table Result :

```
select * from patient_details;
```

Query Result x

SQL | All Rows Fetched: 2 in 0.002 seconds

PATIENT_ID	PATIENT_AGE	PATIENT_NAME
1	44	Dilip Singh
2	44	Dilip Singh

➤ **GenerationType.IDENTITY:**

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

➤ **Entity class with IDENTITY Type:**

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

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

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

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

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

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

    public long getPateintId() {
        return pateintId;
    }

    public void setPateintId(long pateintId) {
        this.pateintId = pateintId;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }
}
```

```

    }

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

```

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

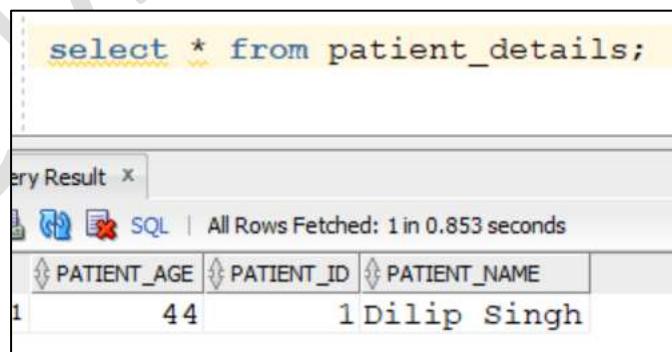
```

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

```

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

Table Data :



The screenshot shows a MySQL Workbench interface with a query editor and a results grid. The query is:

```
select * from patient_details;
```

The results grid shows one row:

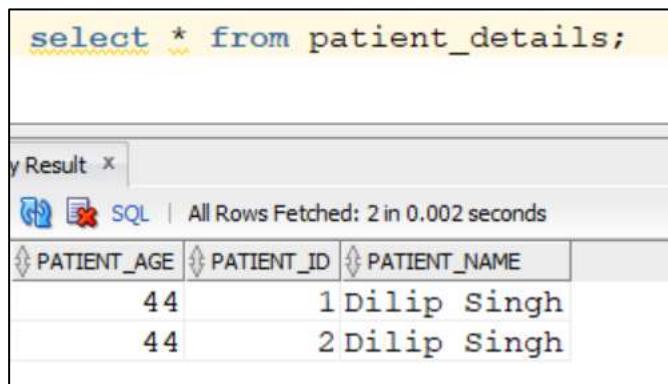
	PATIENT_ID	PATIENT_NAME
1	44	Dilip Singh

Below the grid, it says "All Rows Fetched: 1 in 0.853 seconds".

Execute Again :

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

Table Result:



select * from patient_details;		
Result		
PATIENT_ID	PATIENT_NAME	PATIENT_AGE
1	Dilip Singh	44
2	Dilip Singh	44

GenerationType.SEQUENCE:

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

➤ Entity class with IDENTITY Type:

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

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

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

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

    @Column(name = "patient_name")
```

```

private String name;

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

public long getPateintId() {
    return pateintId;
}

public void setPateintId(long pateintId) {
    this.pateintId = pateintId;
}

public String getName() {
    return name;
}

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

public int getAge() {
    return age;
}

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

```

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

```

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

```

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

Table Data :

select * from patient_details		
Query Result		
PATIENT_ID	PATIENT_NAME	PATIENT_NAME
1	44	Dilip Singh

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

```
2023-11-10T10:44:12.000+05:30 [INFO] 4404 --- [main] t.sprin...plicati...app: 
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)
```

Table Data:

select * from patient_details		
Query Result		
PATIENT_ID	PATIENT_NAME	PATIENT_NAME
1	44	Dilip Singh
2	33	Suresh Singh

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

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

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

generator:

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

@SequenceGenerator Annotation:

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

Attributes:

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

Example with Custom Sequence Generator:

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

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

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

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

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

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

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

public long getPatientId() {
    return patientId;
}

public void setPatientId(long patientId) {
    this.patientId = patientId;
}

public String getName() {
    return name;
}

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

public int getAge() {
    return age;
}

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

Data Persisting in Table:

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

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

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

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

```

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

```

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

Table Result :

select * from patient_details		
Result		
PATIENT_ID	PATIENT_NAME	PATIENT_NAME
1000	33	Dilip Singh

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

select * from patient_details		
Query Result		
PATIENT_ID	PATIENT_NAME	PATIENT_AGE
1000	33 Dilip Singh	
1001	33 Dilip Singh	
1002	33 Dilip Singh	
1003	33 Dilip Singh	

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

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

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

create sequence patient_id_values start with 20000 increment by 2;
Query Result Script Output Task completed in 0.129 seconds
Sequence PATIENT_ID_VALUES created.

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

<TBD>

GenerationType.TABLE:

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

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

Example With GenerationType.TABLE:

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

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

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

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

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

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

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

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

- In this example, the **@GeneratedValue** annotation is used with the **GenerationType.TABLE** strategy to indicate that the **id** field of **Entity** should have its values generated using a separate table.
- Now Try to insert data inside table **patient_details** from JPA Repository.

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

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

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

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

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

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

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

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

Table Data:

```
select * from patient_details;
```

Query Result x

SQL | All Rows Fetched: 1 in 0.809 seconds

PATEINT_ID	PATIENT_AGE	PATIENT_NAME
1	33	Dilip Singh

Primary Key Table: **hibernate_sequences**

HIBERNATE_SEQUENCES

Data Model Constraints Grants Statistics Triggers

Sort... Filter:

NEXT_VAL	SEQUENCE_NAME
50	default

Execute Same Logic Again and Again With New Patients Data:

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

```
select * from patient_details;
```

Query Result x

SQL | All Rows Fetched: 6 in 0.003 seconds

PATEINT_ID	PATIENT_AGE	PATIENT_NAME
1	33	Dilip Singh
2	25	tek teacher
3	52	tek teacher
4	102	tek teacher
5	152	tek teacher
6	202	tek teacher

Note:

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

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

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

JPA @TableGenerator Annotation:

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

Attributes:

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

Example: Create Entity Class: Patient.java

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

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

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

    @Id
    @Column
    @TableGenerator(name = "pat_id_generator", table = "pat_id_values",
```

```

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

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

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

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

```

- In this example, the **@GeneratedValue** annotation is used with the **GenerationType.TABLE** strategy along with custom Table **generator** value.
- Now Try to insert data inside table **patient_details** from JPA Repository.

```

package com.tek.teacher.data;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;
}

```

```

public void addPatient() {

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

```

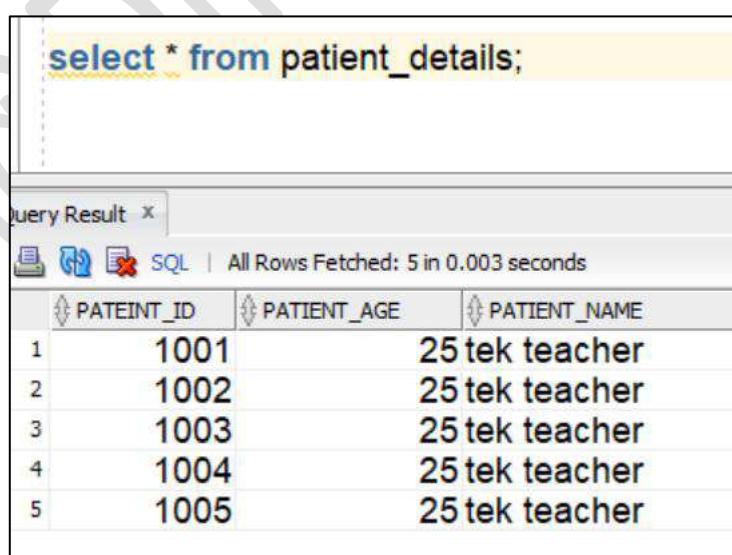
- Now execute Logic and try to monitor in application console logs, how JPA created our own primary key values table instead of default table data and format.
- JPA created table with custom values provided as part of `@TableGenerator` annotation and attributes values as shown in console logs.

```

Hibernate: create table pat_id_values (next_val number(19,0), pat_id varchar2(255 char) not null, primary key (pat_id))
Hibernate: insert into pat_id_values(pat_id, next_val) values ('pat_id_nxt_value',1000)
Hibernate: create table patient_details (patient_age number(10,0), pateint_id number(19,0) not null, patient_name varchar2(255 char), primary key (pateint_id))
2023-11-16T14:42:36.964+05:30  INFO 12304 --- [           main] j.LocalContainerEntityManagerFactoryBean :
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-16T14:42:37.274+05:30  INFO 12304 --- [           main] t.SpringBootJpaGeneartedvalueApplication :
Started SpringBootJpaGeneartedvalueApplication in 4.898 seconds (process running for 5.542)
Hibernate: select tbl.next_val from pat_id_values tbl where tbl.pat_id=? for update
Hibernate: update pat_id_values set next_val=? where next_val=? and pat_id?
Hibernate: insert into patient_details (patient_age,patient_name,pateint_id) values (?,?,?)
2023-11-16T14:42:37.531+05:30  INFO 12304 --- [Thread-1] i.LocalContainerEntityManagerFactoryBean :

```

Tables Result: After multiple Executions of different patients records.



The screenshot shows a MySQL Workbench interface. The SQL editor contains the query: `select * from patient_details;`. The results are displayed in a table titled "Query Result" with the following data:

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

PAT_ID_VALUES	
Data Model Constraints Grants Statistics Triggers	
	
	
Sort..	Filter:
 NEXT_VAL	 PAT_ID
1005	pat_id_nxt_value

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

- **GeneratedValue** is used only to get the generated values of column. The two arguments **strategy** and **generator** are used to define how the value is created or gained. We can define to use the database sequence or value from table which is used to store increasing sequence values. But to specify database sequence or table generators name, we specify the named generators to **generator** argument.
- **SequenceGenerator/TableGenerator** is used to define named generators, to map a user defined sequence generator with your JPA session. This is used to give a name to database sequence or database value of table or any kind of generators. This name can be now referred by the **generator** argument of **GeneratedValue**.

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

GenerationType.UUID:

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

What is UUID?

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

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

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

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

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

Create Entity Class with UUID Generator Strategy:

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

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

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

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

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

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

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

- Now execute Logic and try to monitor in application console logs, how JPA working with **GenerationType.UUID** strategy of **GeneratedValue**.
- **Execute Same Logic Again and Again With New Patients Data:**

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

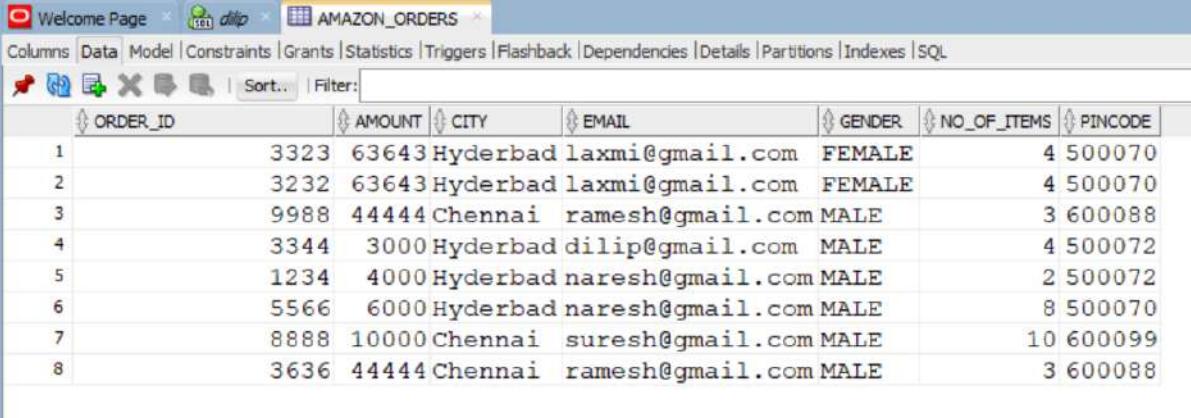
SELECT * FROM PATIENT_DETAILS		
ts 1 ×		
PATEINTID	PATIENT_AGE	PATIENT_NAME
a3335f0a-82ef-47ae-a7e1-1d5c5c3bc4e4	44	naresh Singh
003e48fd-c1cd-40a1-9aae-1b828efc1397	33	Dilip Singh

Sorting and Pagination in JPA:

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



Table and Data:



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

 **Requirement:** Get Details by Email Id with Sorting

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

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

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

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

```
package com.dilip.dao;

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

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

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

```

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

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

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

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

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

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

```

➤ Now Create A Repository.

```

package com.dilip.dao;

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

public interface AmazonOrderRepository extends JpaRepository<AmazonOrders, Integer> {
}

```

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

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

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

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

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

```
package com.dilip.dao;

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

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

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

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

➤ **Now Execute above Logic**

```
package com.dilip;

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

@SpringBootApplication
public class SpringBootJpaSortingPaginationApplication {

    public static void main(String[] args) {

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

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

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

[

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

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

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

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

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

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

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

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

1

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

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

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

```
package com.dilip.dao;

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

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;
```

```

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

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

```

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

```

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

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

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

Pagination:

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

time, making it easier to browse, consume, and interact with large datasets or content collections.



Key features and concepts related to pagination include:

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

Page Number: Pagination is typically associated with a page number, starting from 1 and incrementing as users navigate through the data. Users can move forward or backward to view different segments of the dataset.

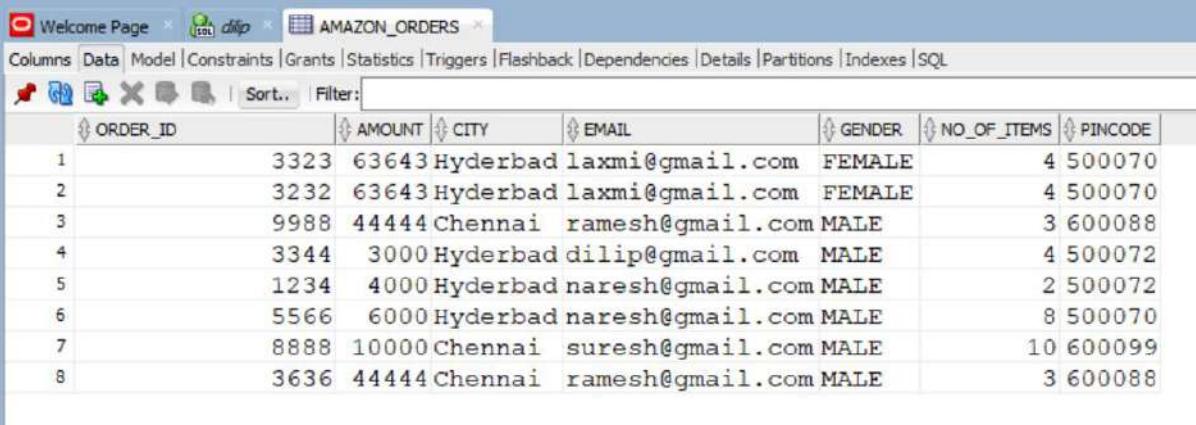
Navigation Controls: Pagination is usually accompanied by navigation controls, such as "Previous" and "Next" buttons or links. These controls allow users to move between pages easily.

Total Number of Pages: The total number of pages in the dataset is determined by dividing the total number of items by the page size. For example, if there are 100 items and the page size is 10, there will be 10 pages.

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

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

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



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

In Spring Data JPA, **Pageable** is an interface that allows you to paginate query results easily. It provides a way to specify the page number, the number of items per page (page size), and optional sorting criteria for your query results. This is particularly useful when you need to retrieve a large set of data from a database and want to split it into smaller pages.

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

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

```
package com.dilip.dao;

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

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

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

➤ Now execute above logic

```
package com.dilip;

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

```

import org.springframework.context.ApplicationContext;
import com.dilip.dao.OrdersOperations;

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

```

Output: We got first 2 records of table.

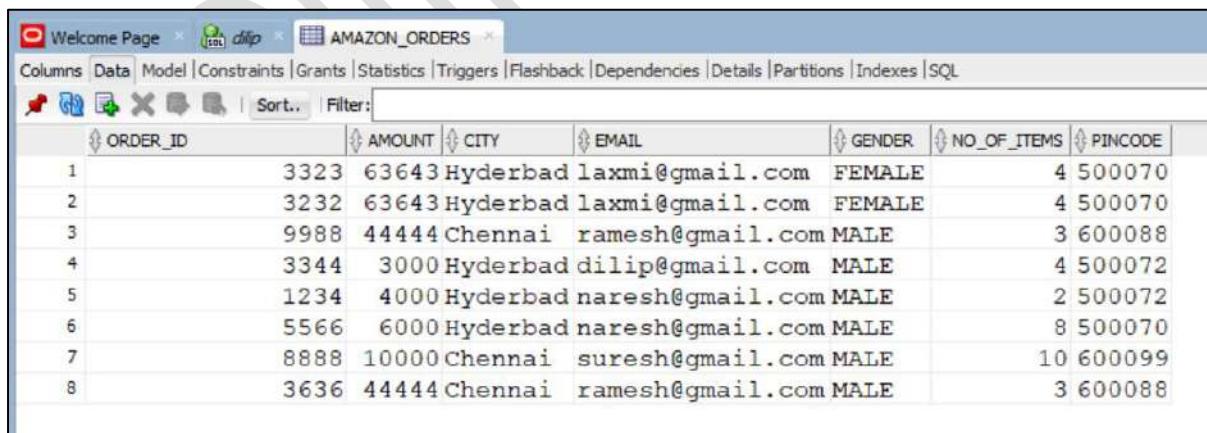
```

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

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

```

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



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

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

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

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

```
package com.dilip.dao;

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

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

    public void getRecordsByPageIdAndNoOfRecords(int pagId, int noOfReorcds) {

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

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

    }
}
```

➤ Now execute above logic

```
package com.dilip;

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

import com.dilip.dao.OrdersOperations;

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

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

```
[  
AmazonOrders(orderId=3344, noOfItems=4, amount=3000.0, email=dilip@gmail.com,  
pincode=500072, city=Hyderabad, gender=MALE),  
  
AmazonOrders(orderId=1234, noOfItems=2, amount=4000.0, email=naresh@gmail.com,  
pincode=500072, city=Hyderabad, gender=MALE),  
  
AmazonOrders(orderId=5566, noOfItems=8, amount=6000.0, email=naresh@gmail.com,  
pincode=500070, city=Hyderabad, gender=MALE)  
]
```

 **Requirement:** Pagination with Sorting:

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

```
package com.dilip.dao;  
  
import java.util.List;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.domain.PageRequest;  
import org.springframework.data.domain.Sort;  
import org.springframework.data.domain.Sort.Direction;  
import org.springframework.stereotype.Component;  
  
@Component  
public class OrdersOperations {  
  
    @Autowired  
    AmazonOrderRepository repository;  
  
    public void getDataByPaginationAndSorting(int pagId, int noOfRecords) {  
  
        List<AmazonOrders> allOrders =  
            repository.findAll(PageRequest.of(pagId, noOfRecords,  
                Sort.by(Direction.DESC, "noOfItems"))).getContent();  
  
        System.out.println(allOrders);  
    }  
}
```

➤ **Execute Above Logic**

```
package com.dilip;
```

```

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

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

```

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

```

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

```

Native Queries & JPQL Queries with Spring JPA:

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

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

```

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

```

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

JPQL Query:

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

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

```
// JPQL Query in Repository Layer
@Query(value = "Select u from Users u")
List<Users> getUsers();
```

JPQL can perform:

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

Native SQL Query's:

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

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

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

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

1. Indexed Query Parameters
2. Named Query Parameters

By using Indexed Query Parameters:

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

Example: Get All Records Of Table

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

Example: Get All Records Of Table where city is matching

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

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

Example with more indexed parameters: users from either **city** or **pincode** matches.

Example: Get All Records Of Table where city or pincode is matching

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

Examples:

Requirement:

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

Step 1: Define Methods Inside Repository with Native Queries:

```
package com.dilip.repository;

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

```

import com.dilip.entity.Patient;

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

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

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

    //Get All Patients with Age and Gender
    @Query(value = "select * from patient where age=?1 and gender=?2", nativeQuery = true)
    public List<Patient> getPatientDetailsByAgeAndGender(int age, String gender);
}

```

➤ **Step 2: Call Above Methods from DB Operations Class**

```

package com.dilip.operations;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

    //Select all patients
    public List<Patient> getPatientDetails() {
        return repository.getAllPatients();
    }

    //by email Id
    public Patient getPatientDetailsbyEmailId(String email) {
        return repository.getDetailsByEmail(email);
    }

    //age and gender
    public List<Patient> getPatientDetailsbyAgeAndGender(int age, String gender) {

```

```

        return repository.getPatientDetailsByAgeAndGender(age, gender);
    }
}

```

➤ **Step 3: Testing From Main Method class**

```

package com.dilip;

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

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

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

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

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

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

```

Output:

```

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

Hibernate: select * from patient where emailid=?
Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com]
***** PAteints with Age and gender***** 

Hibernate: select * from patient where age=? and gender=?
[Patient [name=Dilip Singh, age=31, gender=MALE, contact=+9188882828, emailId=dilip@gmail.com]]

```

By using Named Query Parameters:

We can also pass values of method parameters to the query using named parameters i.e. we are providing We define these using the **@Param** annotation inside our repository method declaration. Each parameter annotated with **@Param** must have a value string matching the corresponding **JPQL or SQL** query parameter name. A query with named parameters is easier to read and is less error-prone in case the query needs to be refactored.

```

@Query(value = "select * from flipkart_users where city=:cityName and pincode=:pincode", nativeQuery = true)
List<Users> getUsersByCityAndPincode(@Param("cityName") String city, @Param("pincode") String pincode);

```

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

Requirement:

1. Insert Patient Data

Step 1: Define Method Inside Repository with Native Query:

```

package com.dilip.dao;

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

```

```

@Component

```

```

public interface PatientRepository extends CrudRepository<Patient, String> {

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

```

Step 2: Call Above Method from DB Operations Class

```

package com.dilip.dao;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

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

```

Step 3: Test it From Main Method class.

```

package com.dilip;

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

```

```

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

```

Output:

```

□ *** Adding Patient *****
Hibernate: INSERT INTO patient VALUES (?, ?, ?, ?, ?, ?)

```

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

@Transactional:

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

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

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

@Modifying:

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

and insert data in the database. It is often used in conjunction with the **@Transactional** annotation to ensure that the data is updated or deleted in a safe and consistent manner.

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

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

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

JPQL Queries Execution:

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

Requirement:

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

Step1: Define Repository Methods along with JPQL Queries.

```
package com.dilip.repository;

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

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

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

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

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

Step 2: Call Above Methods From DB Operations class.

```
package com.dilip.operations;

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

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

    // All Patients
    public List<Patient> getAllpatients() {
        return repository.getAllPatients();
    }

    // All Patients Names
    public List<String> getAllpatientsNames() {
        return repository.getAllPatientsNames();
    }

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

Step 3: Test it From Main Method class.

```
package com.dilip;

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

@SpringBootApplication
public class PatientApplication {
```

```

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

```

Output:

```

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

```

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

JPQL Query Guidelines

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

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

- The HAVING clause: The HAVING clause specifies the conditions that the groups must meet in order to be included in the results of the query.
- The ORDER BY clause: The ORDER BY clause specifies the order in which the results of the query will be returned.

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

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

Relationship Mapping with JPA:

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

Here are JPA relationship mappings:

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

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

Example: An Employee has one Address.

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

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

Example: An Employee has more than one Address.

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

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

Example: Many Employees belong to one Role.

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

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

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

5. Unidirectional and Bidirectional Relationships:

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

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

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

Cascading and Cascade Types:

What Is Cascading?

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

When we perform some action on the target entity, the same action will be applied to the associated entity. Cascading is the way to achieve this. To enable this behaviour, we had use “**CascadeType**” attribute with mappings. To establish a dependency between related entities, JPA provides `jakarta.persistence.CascadeType` enumerated types that define the cascade operations. These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many.

JPA Cascade Types:

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

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

Type	Description
------	-------------

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

CascadeType.PERSIST:

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

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

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

CascadeType.MERGE:

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

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

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

CascadeType.REMOVE:

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

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

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

CascadeType.REFRESH:

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

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

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

CascadeType.DETACH:

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

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

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

CascadeType.ALL:

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

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

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

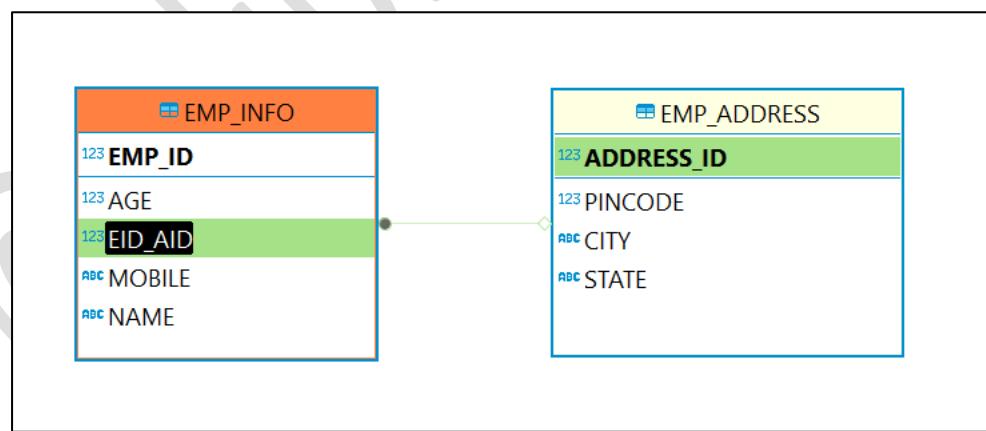
One-to-One (1:1) Relationship:

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

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

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

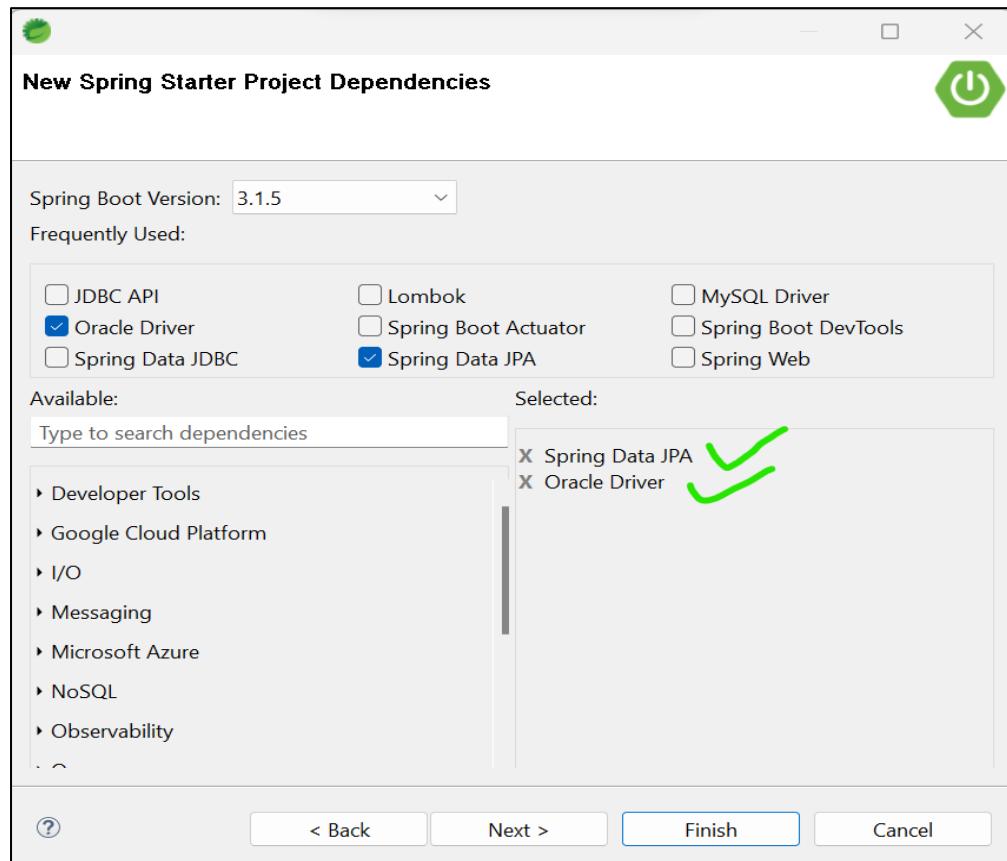
Requirement: Add Employee and Address Details with One to One Relationship as shown in below Entity Relationship Diagram.



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

Implementation:

- Now Create a Spring Boot JPA Project.



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

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

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

- Now Create Entity Classes as details shown ER diagram.

Entity Class : Address.java

```
package com.dilip.entity;

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

@Entity
```

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

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

    @Column
    private String city;

    @Column
    private int pincode;

    @Column
    private String state;

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

Entity Class : Employye.java

```
package com.dilip.entity;
```

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

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

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

    @Column
    private String name;

    @Column
    private int age;

    @Column
    private String mobile;

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

    public Long getEmpld() {
        return empld;
    }

    public void setEmpld(Long empld) {
        this.empld = empld;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }
}
```

```

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

```

@JoinColumn:

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

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

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

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

- Create JPA Repository's for Employye Entity Classes.

Repository Interface : EmployyeRepository.java

```

package com.dilip.repository;

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

```

```
import com.dilip.entity.Employee;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long>{}
```

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

EmployeeService.java

```
package com.dilip.service;

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

@Component
public class EmployeeService {

    @Autowired
    EmployeeRepository repository;

    public String addNewEmployee() {

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

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

        // Setting Address Entity Object inside Employee
        employee.setAddress(address);

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

- Execute above Service class methods to do Database Operations.

```
package com.dilip;

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

@SpringBootApplication
public class SpringBootJpaMappingsApplication {

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

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

Results:

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

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

Data in Tables:

SELECT * FROM EMP_INFO						Enter a SQL expression to filter results (use Ctrl+Space)
	EMP_ID	NAME	MOBILE	AGE	EID_AID	
1	1	Dilip Singh	+91-8826111377	30	1	

SELECT * FROM EMP_ADDRESS					Enter a SQL expression to filter results (use Ctrl+Space)
	ADDRESS_ID	PINCODE	CITY	STATE	
1	1	500,072	Hyderabad	Telangana	

- Similarly Try to Insert Few More Records.

SELECT * FROM EMP_INFO ei					Enter a SQL expression to filter results (use Ctrl+Space)
EMP_ID	NAME	AGE	EID_AID	MOBILE	
1	Dilip Singh	30	1	+91-8826111377	
2	Naresh	44	2	+91-8125262702	
3	Anusha	22	3	+1772727727	

SELECT * FROM EMP_ADDRESS ea					Enter a SQL expression to filter results (use Ctrl+Space)
ADDRESS_ID	PINCODE	CITY	STATE		
2	400,000	Banglore	Karnatka		
3	300,555	texas	USA		
1	500,072	Hyderabad	Telangana		

Project Directory Structure:

spring-boot-jpa-mappings-one-to-one [boot]
src/main/java
com.dilip
SpringBootJpaMappingsApplication.java
com.dilip.entity
Address.java
Employye.java
com.dilip.repository
EmployyeRepository.java
com.dilip.service
EmployyeService.java
src/main/resources
static
templates
application.properties

Delete Owner Entity Records:

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

Delete One Employee Details From Employee Table:

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

```
package com.dilip.service;

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

@Component
public class EmployeeService {

    @Autowired
    EmployeeRepository repository;

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

- Execute Above Method From Main Method:

```
package com.dilip;

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

@SpringBootApplication
public class SpringBootJpaMappingsApplication {

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

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

```

EMP_INFO ei left join emp_address ea on ei.address_id = ea.address_id where ei.emp_id = 2
Hibernate: delete from emp_info where emp_id=?
Hibernate: delete from emp_address where address_id=?
  
```

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

SELECT * FROM EMP_INFO ei					SELECT * FROM EMP_ADDRESS							
1	X	* FROM EMP_INFO ei Enter a SQL expression to filter results (use Ctrl + F)					X	* FROM EMP_ADDRESS Enter a SQL expression to filter results (use Ctrl + F)				
EMP_ID ↑	ABC NAME ↓	ABC MOBILE ↓	123 AGE ↓	123 EID_AID ↓	ADDRESS_ID ↑	123 PINCODE ↓	ABC CITY ↓	ABC STATE ↓				
1	Dilip Singh	+91-8826111377	30	1	1	500,072	Hyderabad	Telangana				
3	Anusha	+1772727727	22	3	3	300,555	texas	USA				

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

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

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

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

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

- Define Entity Classes with One to Many Relationships.

Entity class : Employee.java

```

package com.dilip.entity;

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

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

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

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

    @Column
    private String name;

    @Column
    private int age;

    @Column
    private String mobile;

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

    public Long getEmplId() {
        return emplId;
    }

    public void setEmplId(Long emplId) {
        this.emplId = emplId;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public String getMobile() {
        return mobile;
    }
}
```

```

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

```

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

Entity class : Address.java

```

package com.dilip.entity;

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

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

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

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

    public int getAddressId() {
        return addressId;
    }
}

```

```

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

```

➤ Now Add More Addresses with One Employee Object.

```

package com.dilip.service;

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

@Service
public class EmployeeService {

    @Autowired
    EmployeeRepository repository;

    public String addNewEmployee() {

        List<Address> empAddresses = new ArrayList<>();
        Address home = new Address();
        home.setCity("HYDERBAD");
        home.setPincode(500072);
    }
}

```

```

        home.setState("TELANGANA");
        empAddresses.add(home);

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

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

        employye = repository.save(employye);

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

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

```

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

```

package com.dilip;

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

@SpringBootApplication
public class SbootJpaOneToManyMapping {

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

        EmployeeService employyeService = context.getBean(EmployeeService.class);
        employyeService.addNewEmployye();
    }
}

```

```
        }
    }
```

- If we observe Application console logs, both Employee and Address tables are created with Foreign Key relationship.
- One Record inside Employee and Two Records inside Address Tables are inserted.

* FROM EMP_INFO ei			
AGE	EMP_ID	MOBILE	NAME
30	1	+918826111377	Dilip Singh

* FROM EMP_ADDRESS ea					
ADDRESS_ID	PINCODE	EID_AID	CITY	STATE	
1	500,072	1	HYDERBAD	TELANGANA	
2	400,072	1	BANGLORE	KARNATAKA	

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

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

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

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

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

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

Creating Employee Entity Class: Employee.java

```
package com.dilip.entity;

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

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

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

    @Column
    private String name;

    @Column
    private int age;

    @Column
    private String mobile;

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

    public Long getEmplId() {
        return emplId;
    }

    public void setEmplId(Long emplId) {
        this.emplId = emplId;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
```

```

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

```

Creating Department Entity Class: Department.java

```

package com.dilip.entity;

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

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

    @Id
    private Long id;

    private String name;

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

```

```
        this.name = name;
    }
}
```

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

```
package com.dilip.repository;

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

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

}
```

```
package com.dilip.repository;

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

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

}
```

- Now Create Data like More Employees of same Department.

```
package com.dilip.service;

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

@Service
public class EmployeeService {

    @Autowired
    EmployeeRepository employeRepository;

    @Autowired
    DepartmentRepository departmentRepository;
}
```

```

DepartmentRepository departmentRepository;

public void addNewEmployye() {

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

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

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

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

}
}

```

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

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

- Now Execute Above Logic.

```

package com.dilip;

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

@SpringBootApplication
public class SpringJpaApplication {

```

```

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

```

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

```

Hibernate: create table departments (id number(19,0) not null, name varchar2(255 char), primary key (id))
Hibernate: create table emp_info (age number(10,0), dept_id number(19,0), emp_id number(19,0) generated as identity,
mobile varchar2(255 char), name varchar2(255 char), primary key (emp_id))
Hibernate: alter table emp_info add constraint FKci9jim8lqk0nt9dkaisv4mv foreign key (dept_id) references departments
2023-12-29T18:54:50.860+05:30  INFO 19896 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized
JPA EntityManagerFactory for persistence unit 'default'
2023-12-29T18:54:51.187+05:30  INFO 19896 --- [           main] com.dilip.SpringJpaNotesApplication      : Started
SpringJpaNotesApplication in 4.103 seconds (process running for 4.714)
Hibernate: select d1_0.id,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into departments (name,id) values (?,?)✓
Hibernate: select null,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into emp_info (age,dept_id,mobile,name,emp_id) values (?,?,?,?,default)✓
Hibernate: select null,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into emp_info (age,dept_id,mobile,name,emp_id) values (?,?,?,?,default)✓

```

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

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

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

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

Employee
Id (Primary Key)
name
email

Role:
Id (Primary Key)
name

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

Junction Table: employee_roles

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

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

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

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

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

Employee Entity: Employee.java

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

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

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

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

    String name;
```

```

String email;

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

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

```

Role Entity: Role.java

```

package com.tek.teacher.employye;

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

@Entity

```

```

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

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

    String name;

    @ManyToMany
    List<Employee> employees;

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

```

➤ **Create JPA Repository : EmployeRepository.java**

```

package com.tek.teacher.employye;

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

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

```

➤ Now Create a class and Persist Data inside tables.

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

```

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

@Component
public class EmployyeOperations {

    @Autowired
    EmployyeRepository repository;

    public void addEmployye() {
        Employee e = new Employee();
        e.setEmail("Naresh@gmail.com");
        e.setName("Naresh Singh");

        Role r1 = new Role();
        r1.setName("DEVLOPER");
        Role r2 = new Role();
        r2.setName("USER");

        List<Role> roles = new ArrayList<Role>();
        roles.add(r1);
        roles.add(r2);
        e.setRoles(roles);
        repository.save(e);
    }
}

```

➤ Now Execute Above Logic

```

Hibernate: create table employee_roles (employee_id number(19,0) not null, role_id number(19,0) not null)
Hibernate: create table employees (id number(19,0) generated as identity, email varchar2(255 char), name
varchar2(255 char), primary key (id))
Hibernate: create table roles (id number(19,0) generated as identity, name varchar2(255 char), primary
key (id))
Hibernate: alter table employee_roles add constraint FK398vvu81xw154mvv3g2eytscn foreign key (role_id)
references roles
Hibernate: alter table employee_roles add constraint FK3uwwaxeiuvcfixgd45etkjgmg foreign key
(employee_id) references employees
2024-01-02T18:52:17.969+05:30  INFO 17312 --- [           main] j.LocalContainerEntityManagerFactoryBean
: Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-01-02T18:52:18.378+05:30  INFO 17312 --- [           main] t.SpringBootJpaGeneartedvalueApplication
: Started SpringBootJpaGeneartedvalueApplication in 4.375 seconds (process running for 4.819)
Hibernate: insert into employees (email, name, id) values (?, ?, default)
Hibernate: insert into roles (name, id) values (?, default)
Hibernate: insert into roles (name, id) values (?, default)
Hibernate: insert into employee_roles (employee_id, role_id) values (?, ?)
Hibernate: insert into employee_roles (employee_id, role_id) values (?, ?)

```

➤ From the above Console Logs, if we observe a joining Table is created along with Employee and Roles. Data is inserted in total 3 tables and **employee_roles** table maintaining the relation between **Employees** and **Roles** of **Many to Many** Association.

Database Data of 3 Tables:

	123 ID	ABC EMAIL	ABC NAME
1	1	Naresh@gmail.com	Naresh Singh
2	2	Dilip@gmail.com	Dilip Singh

	123 ID	ABC NAME
1	1	DEVELOPER
2	2	USER
3	3	DEVELOPER
4	4	USER
5	5	MANAGER

	123 EMPLOYEE_ID	123 ROLE_ID
1	1	1
2		1
3		2
4		3
5		4
		5

Spring Framework JPA Project Configuration:

- For using Spring Data JPA, first of all we have to configure **DataSource** bean. Then we need to configure **LocalContainerEntityManagerFactoryBean** bean.

- The next step is to configure bean for transaction management. In our example it's **JpaTransactionManager**.
- **@EnableTransactionManagement**: This annotation allows users to use transaction management in application.
- **@EnableJpaRepositories("com.flipkart.*")**: indicates where the repositories classes are present.

Configuring the DataSource Bean:

- Configure the database connection. We need to set the name of the the JDBC url, the username of database user, and the password of the database user.

Configuring the Entity Manager Factory Bean:

We can configure the entity manager factory bean by following steps:

- Create a new **LocalContainerEntityManagerFactoryBean** object. We need to create this object because it creates the JPA **EntityManagerFactory**.
- Configure the Created DataSource in Previous Step.
- Configure the Hibernate specific implementation of the **HibernateJpaVendorAdapter** . It will initialize our configuration with the default settings that are compatible with Hibernate.
- Configure the packages that are scanned for entity classes.
- Configure the JPA/Hibernate properties that are used to provide additional configuration to the used JPA provider.

Configuring the Transaction Manager Bean:

Because we are using JPA, we have to create a transaction manager bean that integrates the JPA provider with the Spring transaction mechanism. We can do this by using the **JpaTransactionManager** class as the transaction manager of our application.

We can configure the transaction manager bean by following steps:

- Create a new JpaTransactionManager object.
- Configure the entity manager factory whose transactions are managed by the created JpaTransactionManager object.

```

package flipkart.entity;

import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;

```

```

@Configuration

```

```

@EnableJpaRepositories("flipkart.*")
public class SpringJpaConfiguration {

    //DB Details
    @Bean
    public DataSource getDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
        dataSource.setUsername("c##dilip");
        dataSource.setPassword("dilip");
        return dataSource;
    }

    @Bean("entityManagerFactory")
    LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {
        LocalContainerEntityManagerFactoryBean factory
            = new LocalContainerEntityManagerFactoryBean();

        // 1. Setting Datasource Object // DB details
        factory.setDataSource(getDataSource());
        // 2. Provide package information of entity classes
        factory.setPackagesToScan("flipkart.*");
        // 3. Providing Hibernate Properties to EM
        factory.setJpaProperties(hibernateProperties());
        // 4. Passing Predefined Hibernate Adaptor Object EM
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        factory.setJpaVendorAdapter(adapter);

        return factory;
    }

    @Bean("transactionManager")
    public PlatformTransactionManager createTransactionManager() {
        JpaTransactionManager transactionManager = new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(createEntityManagerFactory()
            .getObject());
        return transactionManager;
    }

    // these are all from hibernate FW , Predefined properties : Keys
    Properties hibernateProperties() {

        Properties hibernateProperties = new Properties();
        hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "create");

        //This is for printing internally genearted SQL Quries
        hibernateProperties.setProperty("hibernate.show_sql", "true");
    }
}

```

```

        return hibernateProperties;
    }
}

```

Now create another Component class For Performing DB operations as per our Requirement:

```

package flipkart.entity;

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

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

    @Autowired
    FlipkartOrderRepository flipkartOrderRepository;

    public void addOrderDetails(FlipkartOrder order) {
        flipkartOrderRepository.save(order);
    }
}

```

Now Create a Main method class for creating Spring Application Context for loading all Configurations and Component classes.

```

package flipkart.entity;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("flipkart.*");
        context.refresh();

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

        // Pass Entity Object to Repository METHOD
        OrderDbOperations dbOperation = context.getBean(OrderDbOperations.class);
    }
}

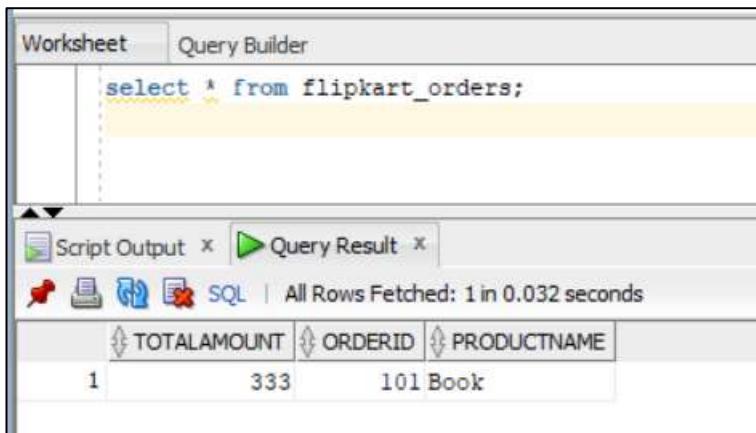
```

```

        dbOperation.addOrderDetails(order);
    }
}

```

Execute The Programme Now. Verify In Database Now.



The screenshot shows the Eclipse IDE interface with the 'Worksheet' tab active. The query `select * from flipkart_orders;` is entered in the worksheet. Below the worksheet, the 'Query Result' tab is active, showing the following data:

	TOTALAMOUNT	ORDERID	PRODUCTNAME
1	333	101	Book

Below the table, it says 'All Rows Fetched: 1 in 0.032 seconds'.

In Eclipse Console Logs, Printed internally generated SQL Queries to perform insert operation.

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

Internally, Based on Repository method `save()` of `flipkartOrderRepository.save(order)`, JPA internally generates implementation code, SQL queries and will be executed internally.

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

We learned to configure the persistence layer of a Spring application that uses Spring Data JPA and Hibernate. Let's create few more examples to do CRUD operations on Db table. From Spring JPA Configuration class, we have used two properties related to Hibernate FW.

hibernate.hbm2ddl.auto: The `'hibernate.hbm2ddl.auto'` property is used to configure the automatic schema/tables generation and management behaviour of Hibernate. This property allows you to control how Hibernate handles the database schema based on your entity classes.

Here are the possible values for the `hibernate.hbm2ddl.auto` property:

none: No action is performed. The schema will not be generated.

validate: The database schema will be validated using the entity mappings. This means that Hibernate will check to see if the database schema matches the entity mappings. If there are any differences, Hibernate will throw an exception.

update: The database schema will be updated by comparing the existing database schema with the entity mappings. This means that Hibernate will create or modify tables in the database as needed to match the entity mappings.

create: The database schema will be created. This means that Hibernate will create all of the tables needed for the entity mappings.

create-drop: The database schema will be created and then dropped when the SessionFactory is closed. This means that Hibernate will create all of the tables needed for the entity mappings, and then drop them when the SessionFactory is closed.

Note: In Spring Boot, we are using property **spring.jpa.hibernate.ddl-auto** with same values for same functionalities as we discussed.

hibernate.show_sql: The **hibernate.show_sql** property is a Hibernate configuration property that controls whether or not Hibernate will log the SQL statements that it generates. The possible values for this property are:

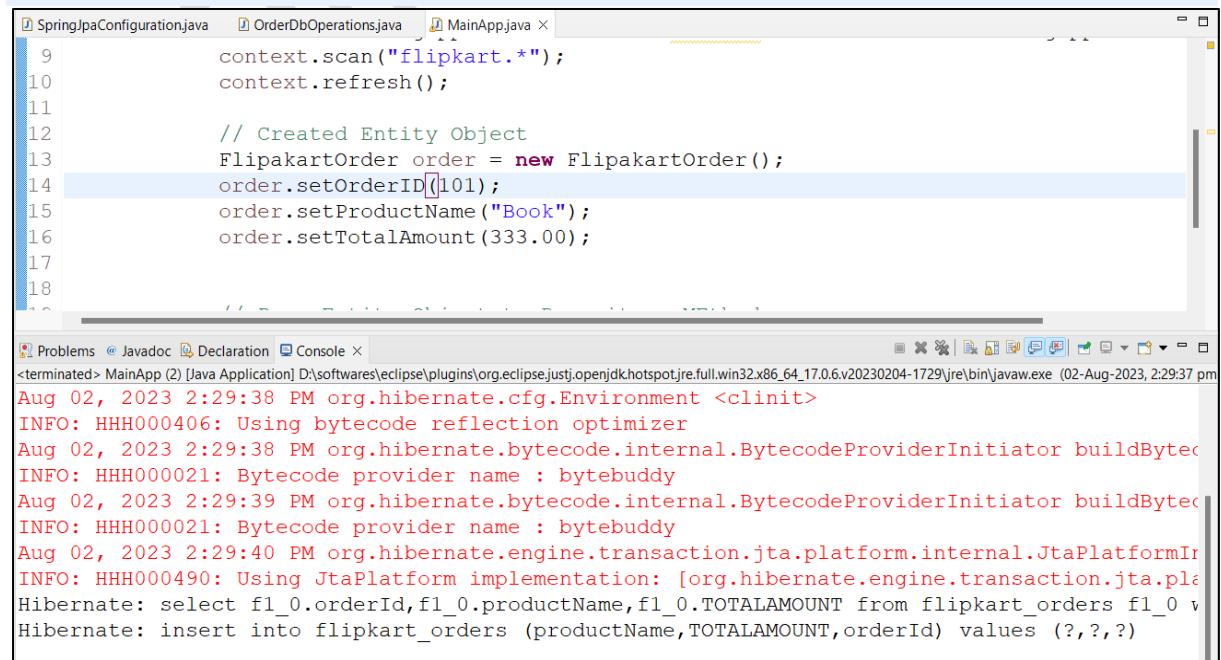
true: Hibernate will log all SQL statements to the console.

false: Hibernate will not log any SQL statements.

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

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

Note: In Spring Boot, we are using property **spring.jpa.show-sql** with same values for same functionalities as we discussed.



```

9         context.scan("flipkart.*");
10        context.refresh();
11
12        // Created Entity Object
13        FlipkartOrder order = new FlipkartOrder();
14        order.setOrderID(101);
15        order.setProductName("Book");
16        order.setTotalAmount(333.00);
17
18

```

Aug 02, 2023 2:29:38 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000406: Using bytecode reflection optimizer
Aug 02, 2023 2:29:38 PM org.hibernate bytecode.internal.BytecodeProviderInitiator buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : bytebuddy
Aug 02, 2023 2:29:39 PM org.hibernate bytecode.internal.BytecodeProviderInitiator buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : bytebuddy
Aug 02, 2023 2:29:40 PM org.hibernate transaction.jta.platform.internal.JtaPlatformInitiator
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate transaction.jta.platform.internal.JtaPlatform]
Hibernate: select f1_0.orderId,f1_0.productName,f1_0.TOTALAMOUNT from flipkart_orders f1_0
Hibernate: insert into flipkart_orders (productName,TOTALAMOUNT,orderId) values (?,?,?)

Creation of New Spring JPA Project:

Requirement : Patient Information

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

Requirements:

- Add Single Patient Details
- Add More Than One Patient Details
- Update Patient Details
- Select Single Patient Details
- Select More Patient Details
- Delete Patient Details

1. Create Simple Maven Project and Add Required Dependencies

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.dilip</groupId>
  <artifactId>spring-jpa-two</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>6.0.11</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>6.0.11</version>
    </dependency>
    <dependency>
      <groupId>com.oracle.database.jdbc</groupId>
      <artifactId>ojdbc8</artifactId>
      <version>21.9.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
    </dependency>
  </dependencies>

```

```

        <version>6.0.11</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.2.6.Final</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
        <version>3.1.2</version>
    </dependency>
</dependencies>
</project>

```

2. Now Create Spring JPA Configuration

```

package com.dilip;

import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;

@Configuration
@EnableJpaRepositories("com.*")
public class SpringJpaConfiguration {

    //DB Details
    @Bean
    public DataSource getDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
        dataSource.setUsername("c##dilip");
        dataSource.setPassword("dilip");
        return dataSource;
    }

    @Bean("entityManagerFactory")
    LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {
        LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();

        // 1. Setting Datasource Object // DB details
        factory.setDataSource(getDataSource());
    }
}

```

```

// 2. Provide package information of entity classes
factory.setPackagesToScan("com.*");

// 3. Providing Hibernate Properties to EM
factory.setJpaProperties(hibernateProperties());

// 4. Passing Predefined Hiberante Adaptor Object EM
HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
factory.setJpaVendorAdapter(adapter);

return factory;
}

//Spring JPA: configuring data based on your project req.
@Bean("transactionManager")
public PlatformTransactionManager createTransactionManager() {
JpaTransactionManager transactionManager = new JpaTransactionManager();
transactionManager.setEntityManagerFactory(createEntityManagerFactory().getObject());
return transactionManager;
}

// these are all from hibernate FW , Predefined properties : Keys
Properties hibernateProperties() {
Properties hibernateProperties = new Properties();
hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");
//This is for printing internally generated SQL Queries
hibernateProperties.setProperty("hibernate.show_sql", "true");
return hibernateProperties;
}

}

```

Note: Until this Step/Configuration in Spring Framework, we have written manually of JPA configuration. From here onwards, Rest of functionalities implementations are as it is like how we implemented in Spring Boot.

Means, The above 2 Steps are automated/ auto configured in Spring Boot internally. We just need to provide database details and JPA properties inside **application.properties**.

3. Create Entity Class

NOTE : Configured **hibernate.hbm2ddl.auto** value as **update**. So Table Creation will be done by JPA internally.

```

package com.dilip.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;

```

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

@Entity
@Table
public class Patient {

    @Id
    @Column
    private String emailId;

    @Column
    private String name;

    @Column
    private int age;

    @Column
    private String gender;

    @Column
    private String contact;

    public Patient() {
        super();
        // TODO Auto-generated constructor stub
    }

    public Patient(String name, int age, String gender, String contact, String emailId) {
        super();
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.contact = contact;
        this.emailId = emailId;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public String getGender() {
        return gender;
    }

}
```

```

public void setGender(String gender) {
    this.gender = gender;
}
public String getContact() {
    return contact;
}
public void setContact(String contact) {
    this.contact = contact;
}
public String getEmailId() {
    return emailId;
}
public void setEmailId(String emailId) {
    this.emailId = emailId;
}

@Override
public String toString() {
    return "Patient [name=" + name + ", age=" + age + ", gender=" + gender +",
    contact=" + contact + ", emailId=" +
    + emailId + "]";
}
}

```

4. Create A Repository Now

```

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

```

5. Create a class for DB operations

```

@Component
public class PatientOperations {
    @Autowired
    PatientRepository repository;
}

```

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

Requirement : Add Single Patient Details

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

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

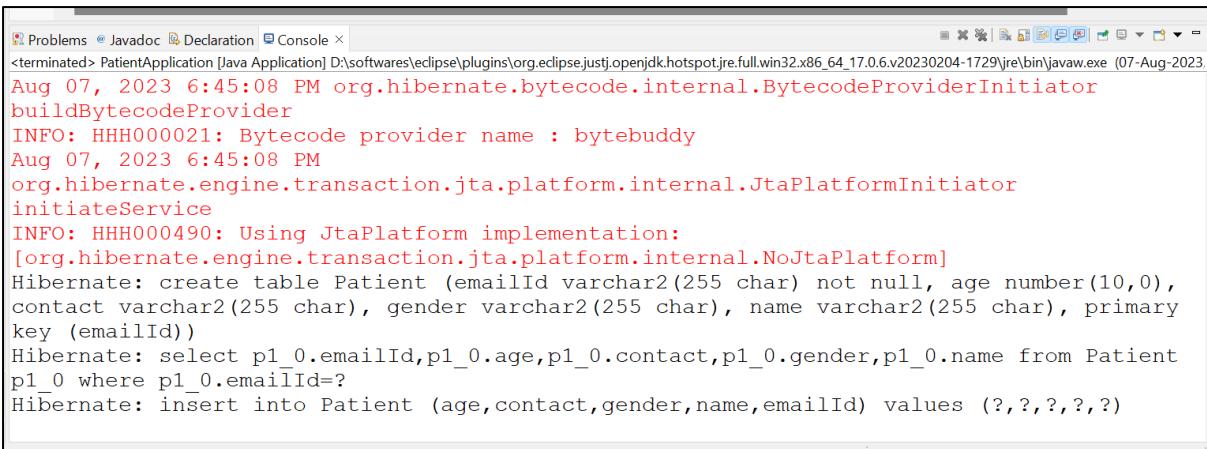
Add Below Method in PatientOperations.java:

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

Now Test it : Create Main method class

```
package com.dilip.operations;  
  
import java.util.ArrayList;  
import java.util.List;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
import com.dilip.entity.Patient;  
  
public class PatientApplication {  
  
    public static void main(String[] args) {  
  
        AnnotationConfigApplicationContext context =  
            new AnnotationConfigApplicationContext();  
        context.scan("com.*");  
        context.refresh();  
  
        PatientOperations ops = context.getBean(PatientOperations.class);  
  
        // Add Single Patient  
        Patient p = new Patient();  
        p.setEmailId("one@gmail.com");  
        p.setName("One Singh");  
        p.setContact("+918826111377");  
        p.setAge(30);  
        p.setGender("MALE");  
  
        ops.addPatient(p);  
    }  
}
```

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



The screenshot shows the Eclipse IDE's Console tab with the following log output:

```

Problems Javadoc Declaration Console
<terminated> PatientApplication [Java Application] D:\softwares\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (07-Aug-2023)
Aug 07, 2023 6:45:08 PM org.hibernate.bytecode.internal.BytecodeProviderInitiator
buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : bytebuddy
Aug 07, 2023 6:45:08 PM
org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator
initiateService
INFO: HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Hibernate: create table Patient (emailId varchar2(255 char) not null, age number(10,0),
contact varchar2(255 char), gender varchar2(255 char), name varchar2(255 char), primary
key (emailId))
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient
p1_0 where p1_0.emailId=?
Hibernate: insert into Patient (age,contact,gender,name,emailId) values (?, ?, ?, ?, ?)

```

Requirement : Add multiple Patient Details at a time

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

saveAll() : This is for adding List of Objects at a time. We should pass List Object of Patient Type.

Add Below Method in PatientOperations.java:

```

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

```

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

```

package com.dilip.operations;

import java.util.ArrayList;
import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();

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

        // Adding More Patients
        Patient p1 = new Patient();
        p1.setEmailId("two@gmail.com");
    }
}

```

```

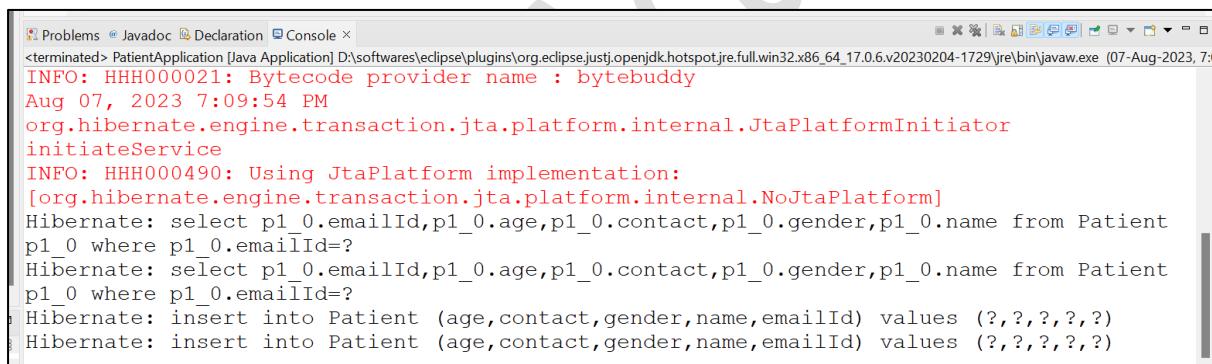
        p1.setName("Two Singh");
        p1.setContact("+828388");
        p1.setAge(30);
        p1.setGender("MALE");

        Patient p2 = new Patient();
        p2.setEmailId("three@gmail.com");
        p2.setName("Xyz Singh");
        p2.setContact("+44343423");
        p2.setAge(36);
        p2.setGender("FEMALE");

        List<Patient> allPatients = new ArrayList<>();
        allPatients.add(p1);
        allPatients.add(p2);
        ops.addMorePatients(allPatients);
    }
}

```

Execution Output:

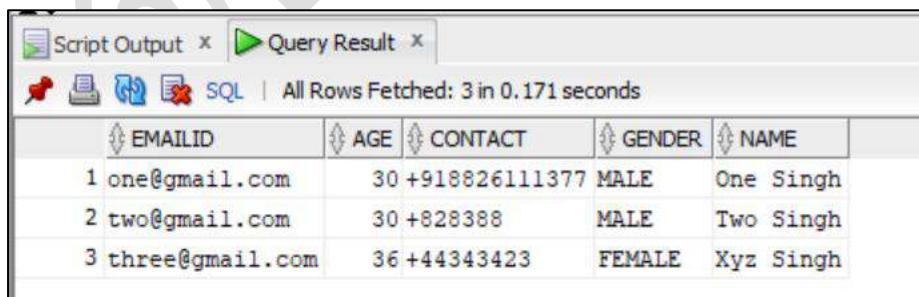


```

Problems Javadoc Declaration Console
<terminated> PatientApplication [Java Application] D:\softwares\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (07-Aug-2023, 7:09:54 PM)
INFO: HHH000021: Bytecode provider name : bytebuddy
INFO: HHH0000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.emailId=?
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.emailId=?
Hibernate: insert into Patient (age,contact,gender,name,emailId) values (?,?,?,?,?)
Hibernate: insert into Patient (age,contact,gender,name,emailId) values (?,?,?,?,?)

```

Verify In DB Table:



EMAILID	AGE	CONTACT	GENDER	NAME
1 one@gmail.com	30	+918826111377	MALE	One Singh
2 two@gmail.com	30	+828388	MALE	Two Singh
3 three@gmail.com	36	+44343423	FEMALE	Xyz Singh

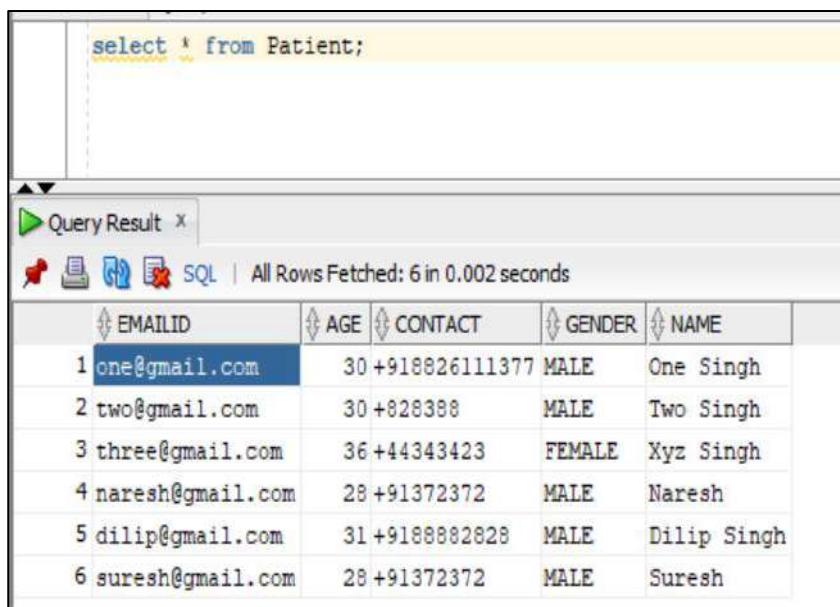
Requirement : Update Patient Details

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

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

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

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

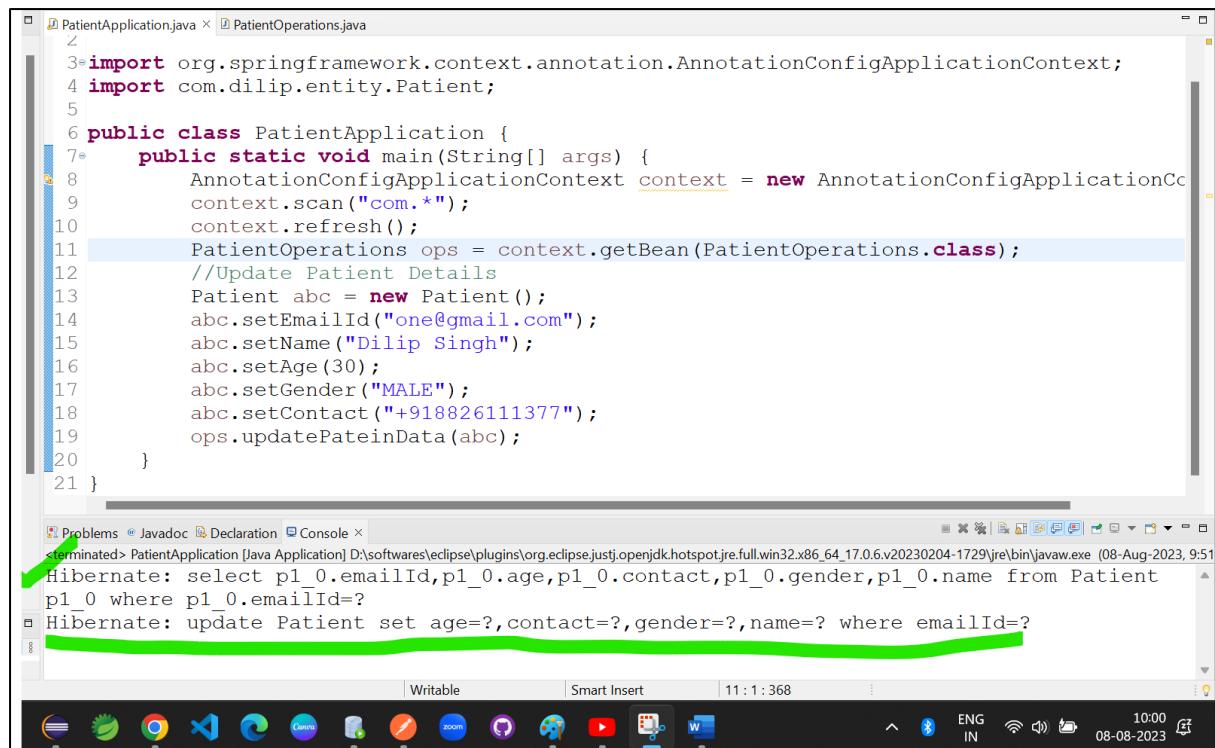


EMAILID	AGE	CONTACT	GENDER	NAME
1 one@gmail.com	30+918826111377	MALE	One Singh	
2 two@gmail.com	30+828388	MALE	Two Singh	
3 three@gmail.com	36+44343423	FEMALE	Xyz Singh	
4 naresh@gmail.com	28+91372372	MALE	Naresh	
5 dilip@gmail.com	31+9188882828	MALE	Dilip Singh	
6 suresh@gmail.com	28+91372372	MALE	Suresh	

Add Below Method in PatientOperations.java:

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

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



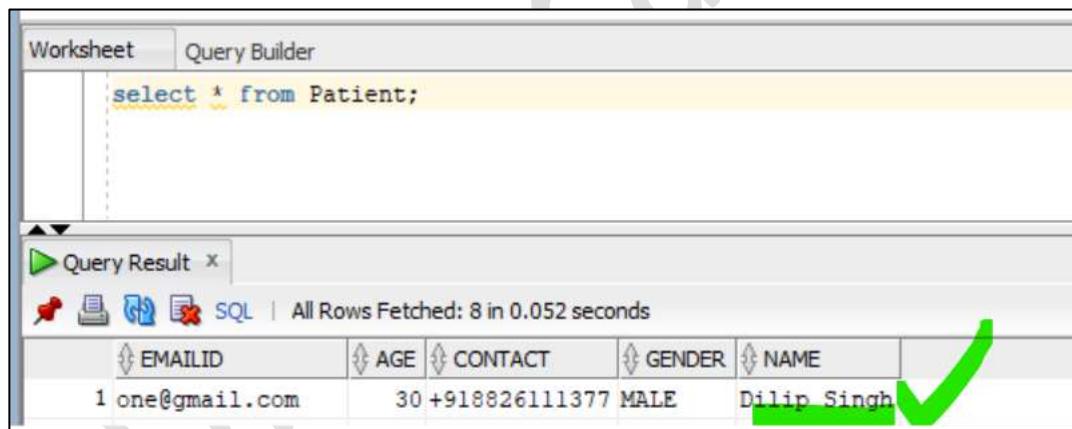
```

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

```

Problems Declaration Console X
 terminated> PatientApplication [Java Application] D:\softwares\eclipse\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (08-Aug-2023, 9:51)
 Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.emailId=?
 Hibernate: update Patient set age=?,contact=?,gender=?,name=? where emailId=?

Verify In DB :



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

Requirement: Deleting Patient Details based on Email ID.

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

deleteById():

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

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

Add Below Method in PatientOperations.java:

```
public void deletePatient(String email) {
    repository.deleteById(email);
}
```

Testing from Main Class:

```
package com.dilip.operations;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();

        PatientOperations ops = context.getBean(PatientOperations.class);
        //Delete Patient Details
        ops.deletePatientWithEmailID("two@gmail.com");
    }
}
```

Before Execution/Deletion:

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

After Execution/Deletion:

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

Requirement: Get Patient Details Based on Email Id.

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

Add Below Method in PatientOperations.java:

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

Testing from Main Class:

```
package com.dilip.operations;

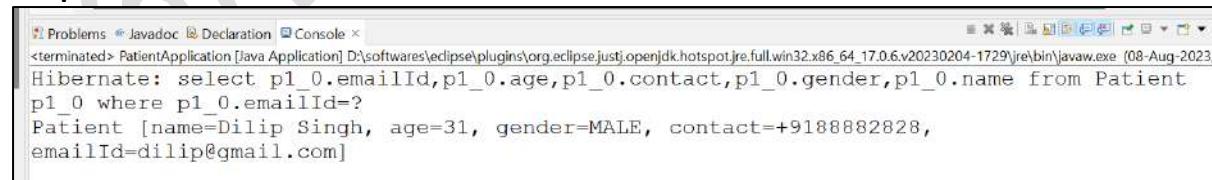
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {

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

        PatientOperations ops = context.getBean(PatientOperations.class);
        //Fetch Patient Details By Email ID
        Patient patient = ops.fetchByEmailId("dilip@gmail.com");
        System.out.println(patient);
    }
}
```

Output:



```
<terminated> PatientApplication [Java Application] D:\softwares\eclipse\plugins\org.eclipse.jdt\openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (08-Aug-2023)
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient
p1_0 where p1_0.emailId=?
Patient [name=Dilip Singh, age=31, gender=MALE, contact=+9188882828,
emailId=dilip@gmail.com]
```

Remaining all functionalities are similar like we discussed in Spring BOOT.

@DilipItAcademy

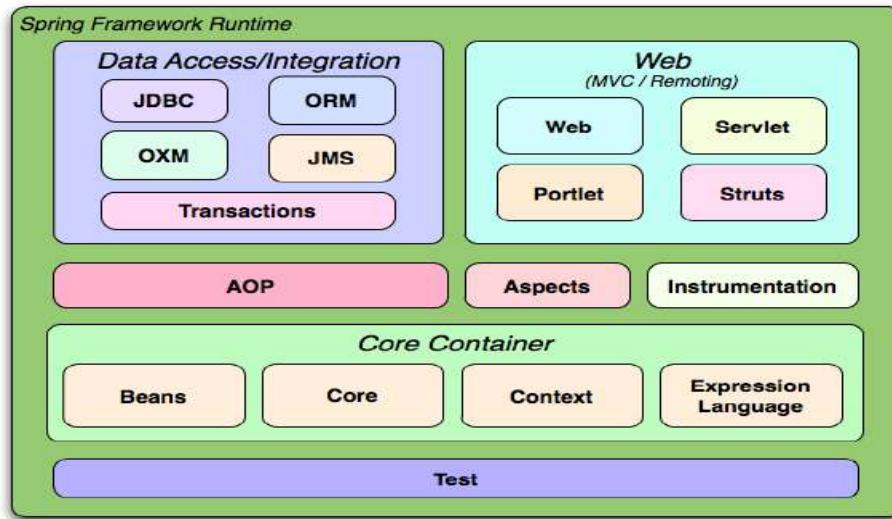
Spring / Spring Boot Web/MVC

Modules

Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning. The formal name, "Spring Web MVC," comes from the name of its source module (spring-webmvc), but it is more commonly known as "Spring MVC". A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

A Spring MVC provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet. Here, DispatcherServlet is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the SpringBoot-starter-web module to get up and running quickly.

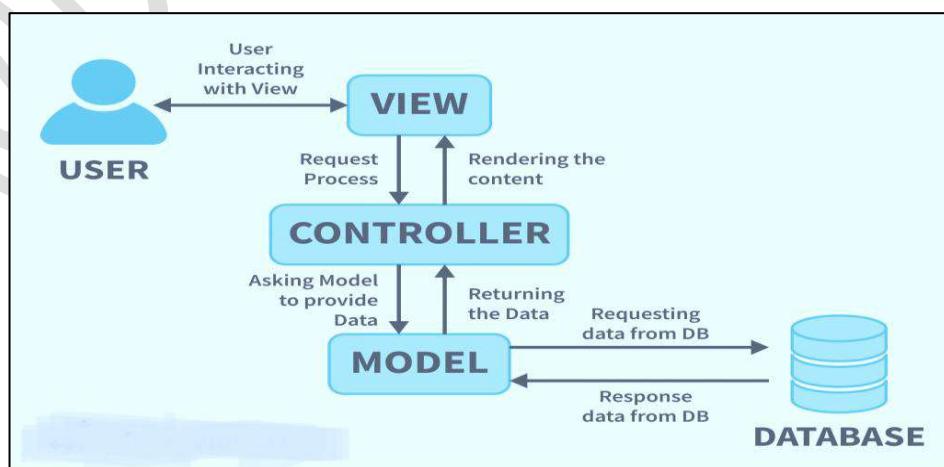


What is MVC?

MVC stands for Model-View-Controller, and it is a widely used architectural pattern in software development, particularly for building user interfaces and web applications. MVC is designed to separate an application into three interconnected components, each with a specific responsibility:

MVC Architecture becomes so popular that now most of the popular frameworks follow the MVC design pattern to develop the applications. Some of the popular Frameworks that follow the MVC Design pattern are:

- **JAVA Frameworks:** Sprint, Spring Boot.
- **Python Framework:** Django.
- **NodeJS (JavaScript):** ExpressJS.
- **PHP Framework:** Cake PHP, Phalcon, PHPixie.
- **Ruby:** Ruby on Rails.
- **Microsoft.NET:** ASP.net MVC.



Model:

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it back to the database or use it to render data.

View:

The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

Controller:

Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

Here's how the MVC pattern works in a typical scenario:

1. A user interacts with the View (e.g., clicks a button or submits a form).
2. The View forwards the user input to the Controller.
3. The Controller processes the input, potentially querying or updating the Model.
4. The Model is updated if necessary, and the Controller retrieves data from the Model.
5. The Controller sends the updated data to the View.
6. The View renders the data and presents it to the user.

Advantages of Spring MVC Framework

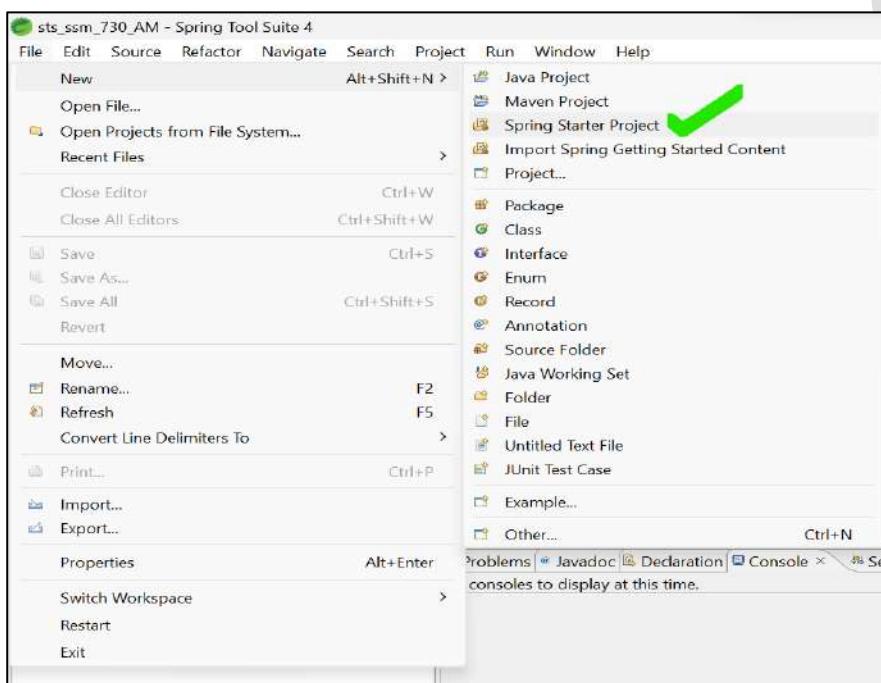
- **Separate roles** - The Spring MVC separates each role, where the model object, controller, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
- **Light-weight** - It uses light-weight servlet container to develop and deploy your application.
- **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
- **Rapid development** - The Spring MVC facilitates fast and parallel development.
- **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
- **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.

- **Flexible Mapping** - It provides the specific annotations that easily redirect the page.

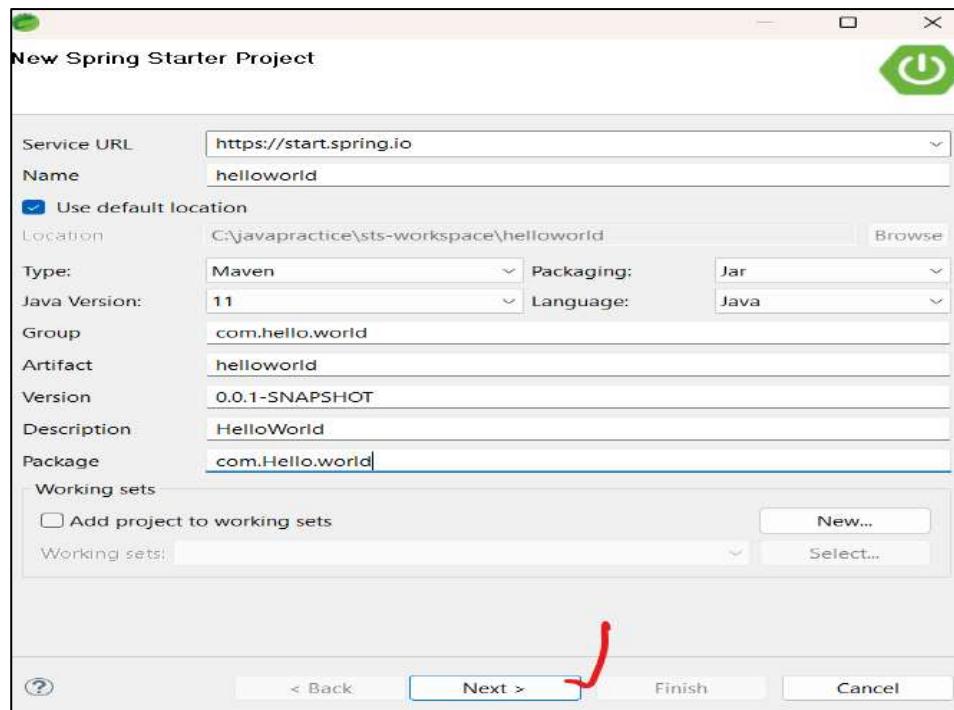
MVC is a foundational pattern used in various software development paradigms, including desktop applications, web applications, and mobile applications. Different platforms and frameworks may implement MVC in slightly different ways, such as Model-View-Presenter (MVP) or Model-View-View-Model (MVVM), but the core principles of separation of concerns and data flow remain consistent.

Creating SpringBoot Web Application:

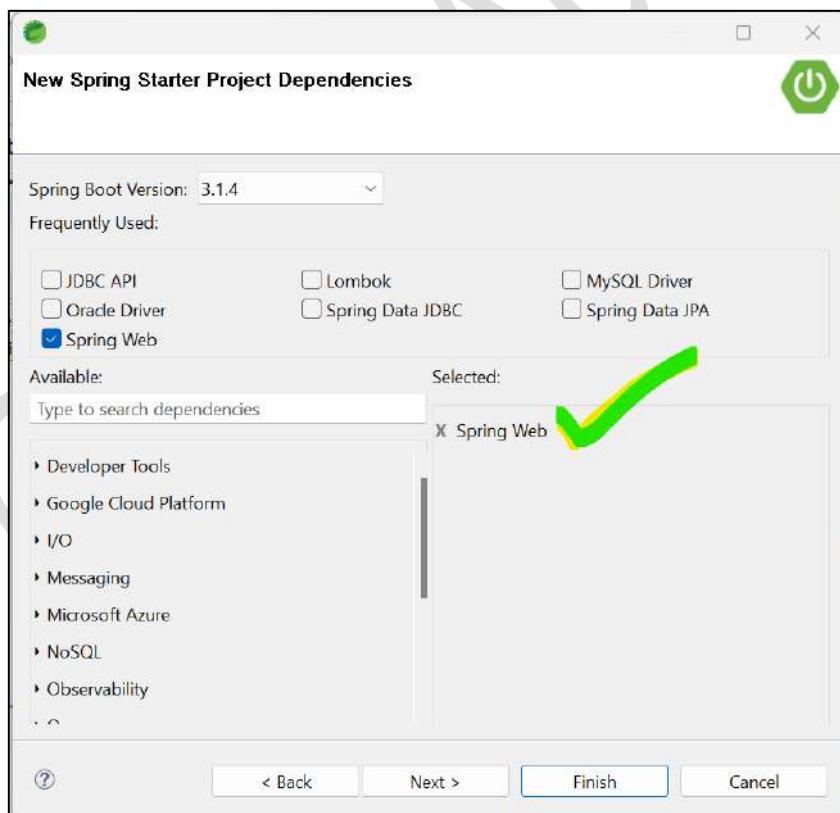
- **Open STS : File-> New > Spring Starter Project**



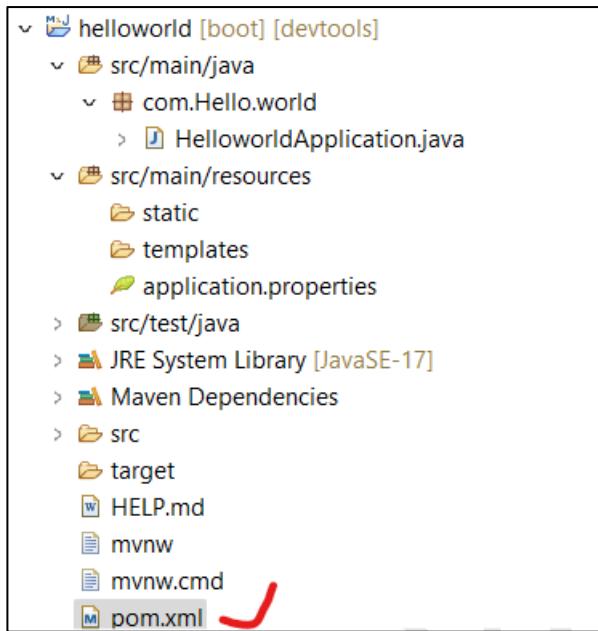
- **Fill All Project details as shown below and click on Next.**



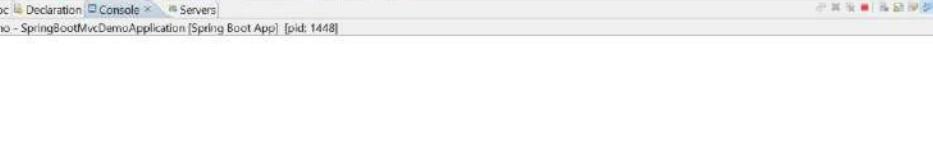
⊕ In Next Page, Add Spring Boot Modules/Starter as shown below and click on finish.



⊕ After finish the project look like this all your dependencies in pom.xml file



- Now Run your Application as **Spring Boot App / java application** from Main Method Class.



sts_2021_7_30_AM - spring-boot-mvc-demo/src/main/resources/application.properties - Spring Tool Suite 4

File Edit Source Navigate Search Project Run Window Help

Problems Javadoc Declaration Console Servers

spring-boot-mvc-demo - SpringBootMvcDemoApplication [Spring Boot App] [pid: 1448]

```
om.dilip.SpringBootMvcDemoApplication : Starting SpringBootMvcDemoApplication using Java 17.0.6 with PID 1
om.dilip.SpringBootMvcDemoApplication : No active profile set, falling back to 1 default profile: "default"
.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
.apache.catalina.core.StandardService : Starting service [Tomcat]
.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.13]
.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 974 ms
.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
om.dilip.SpringBootMvcDemoApplication : Started SpringBootMvcDemoApplication in 2.047 seconds (process run
```

- Integrated Server Started with Default Port : **8080** with context path '. i.e., if we won't give any port number then default port number will be **8080**. If we want to change default port number then, we should add a property and its value in **application.properties**.
 - By Default Spring Boot application will be deployed with empty context path ''. If we want to change default context path then, we should add a property and its value in **application.properties**.

So our application base URL will be always : <http://localhost:8080/>

Requirement: Now Let me add an Endpoint/URL to print Hello World Message.

Controller Class:

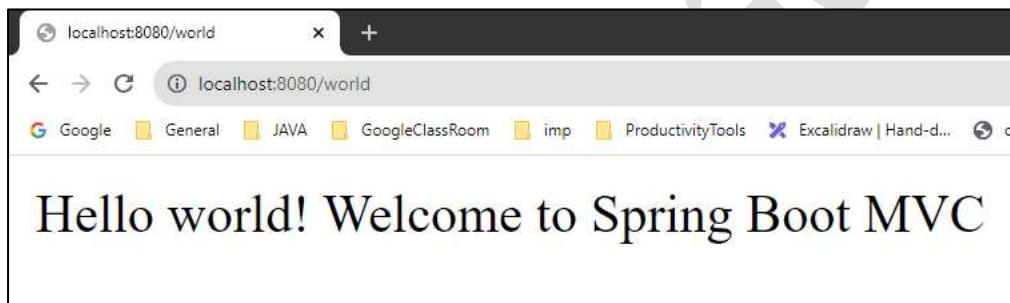
```

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HelloWorldController {
    @GetMapping("/world")
    @ResponseBody
    public String printHelloWorld() {
        return "Hello world! Welcome to Spring Boot MVC";
    }
}

```

Execute/Call above Endpoint: We will access Endpoints/URLs from Http Clients like Browsers.



Changing Default Port Number and Context path of Application:

Now open **application. Properties** file and add below predefined properties and provide required values.

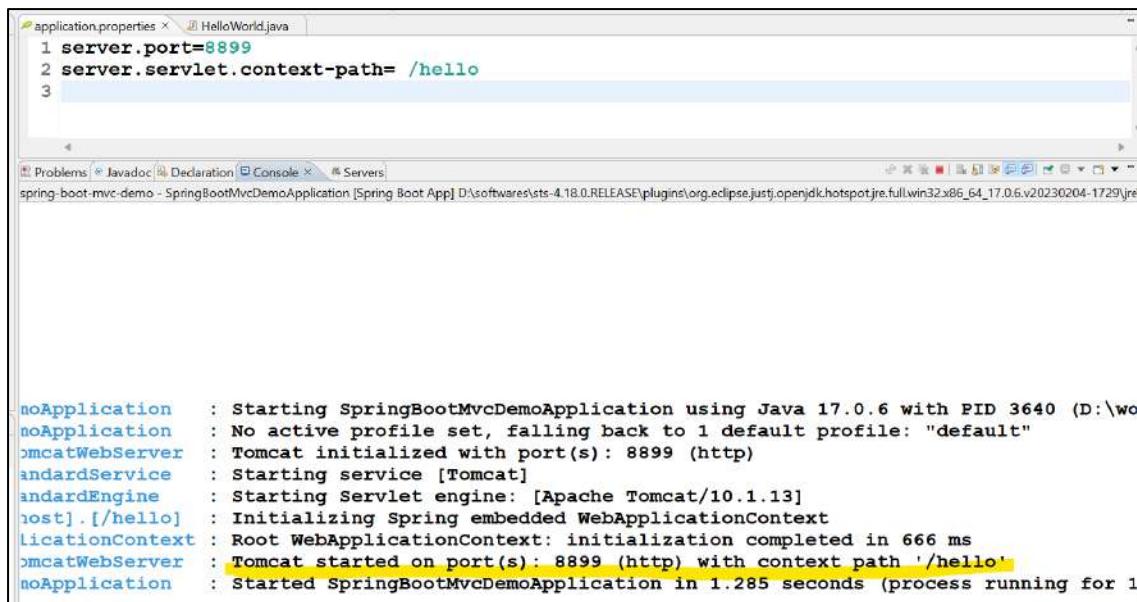
```

server.port=8899
server.servlet.context-path= /hello

```

- Restart our application again, application started on port(s): 8899 (http) with context path '/hello'

So our application base URL will become now always : <http://localhost:8899/hello>



```
application.properties
1 server.port=8899
2 server.servlet.context-path= /hello
3

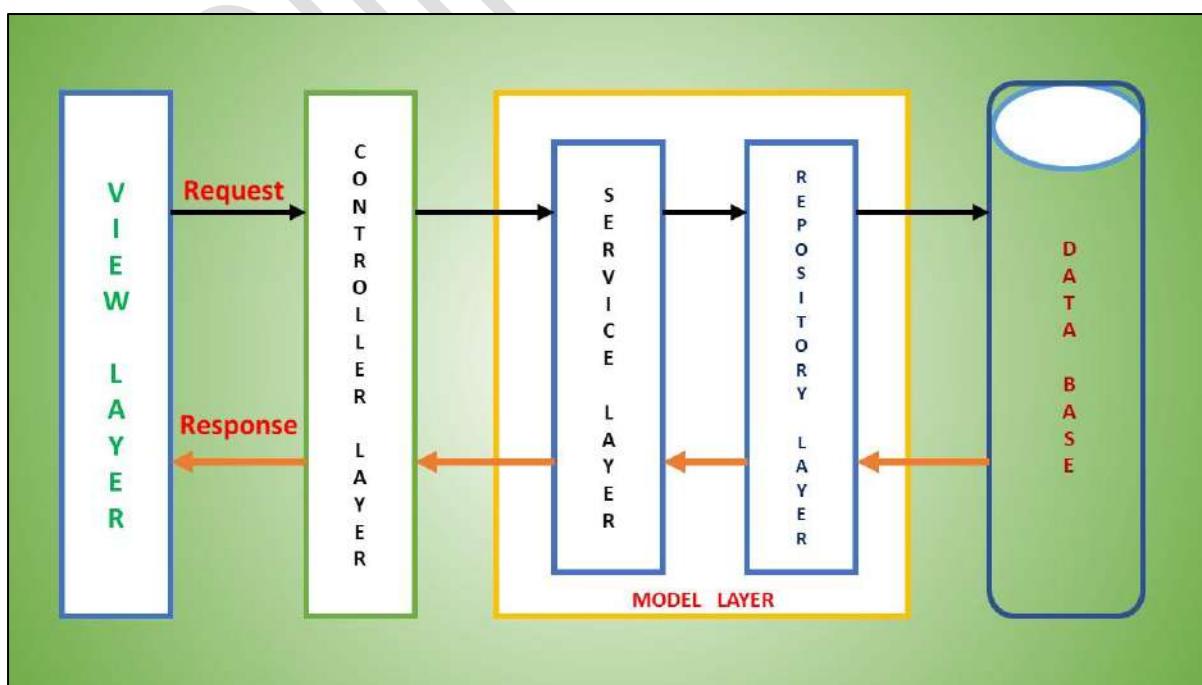
noApplication      : Starting SpringBootMvcDemoApplication using Java 17.0.6 with PID 3640 (D:\wo
noApplication      : No active profile set, falling back to 1 default profile: "default"
tomcatWebServer    : Tomcat initialized with port(s): 8899 (http)
standardService     : Starting service [Tomcat]
standardEngine      : Starting Servlet engine: [Apache Tomcat/10.1.13]
host] . [/hello]    : Initializing Spring embedded WebApplicationContext
applicationContext  : Root WebApplicationContext: initialization completed in 666 ms
tomcatWebServer     : Tomcat started on port(s): 8899 (http) with context path '/hello'
noApplication      : Started SpringBootMvcDemoApplication in 1.285 seconds (process running for 1
```

Output: Call “/world” endpoint : <http://localhost:8899/hello/world>



Spring MVC Application Internal Workflow:

Usually, Spring MVC Application follows below architecture on high level.



Internal Workflow of Spring MVC Application i.e., Request & Response Handling:

The Spring Web **Model-View-Controller** (MVC) framework is designed around a **Front Controller Design Pattern** i.e. **DispatcherServlet** that handles all the HTTP requests and responses across Spring Web application. The request and response processing workflow of the Spring Web MVC **DispatcherServlet** is illustrated in the following diagram.

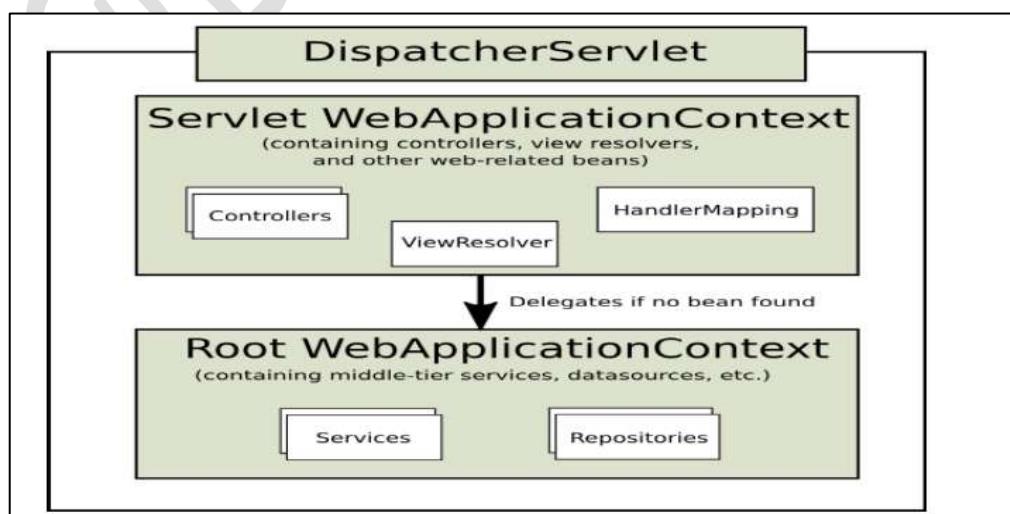
Front Controller:

A **front controller** is defined as a controller that handles all requests for a Web Application. **DispatcherServlet** servlet is the front controller in Spring MVC that intercepts every request and then dispatches requests to an appropriate controller. The **DispatcherServlet** is a Front Controller and one of the most significant components of the Spring MVC web framework. A Front Controller is a typical structure in web applications that receives requests and delegates their processing to other components in the application. The **DispatcherServlet** acts as a single entry point for client requests to the Spring MVC web application, forwarding them to the appropriate Spring MVC controllers for processing. **DispatcherServlet** is a front controller that also helps with view resolution, error handling, locale resolution, theme resolution, and other things.

Request: The first step in the MVC flow is when a request is received by the Dispatcher Servlet. The aim of the request is to access a resource on the server.

Response: Response is made by a server to a client. The aim of the response is to provide the client with the resource it requested, or inform the client that the action it requested has been carried out; or else to inform the client that an error occurred in processing its request.

Dispatcher Servlet: Now, the **DispatcherServlet** with the help of Handler Mappings understands the Controller class name associated with the received request. Once the **DispatcherServlet** knows which Controller will be able to handle the request, it will transfer the request to it. **DispatcherServlet** expects a **WebApplicationContext** (an extension of a plain **ApplicationContext**) for its own configuration. **WebApplicationContext** has a link to the **ServletContext** and the Servlet with which it is associated.



The **DispatcherServlet** delegates to special beans to process requests and render the appropriate responses.

All the above-mentioned components, i.e. **HandlerMapping**, Controller, and **ViewResolver** are parts of **WebApplicationContext** which is an extension of the plain **ApplicationContext** with some extra features necessary for web applications.

HandlerMapping:

In Spring MVC, the **DispatcherServlet** acts as front controller – receiving all incoming HTTP requests and processing them. Simply put, the processing occurs by passing the requests to the relevant component with the help of handler mappings.

HandlerMapping is an interface that defines a mapping between requests and handler objects. The HandlerMapping component parses a Request and finds a Handler that handles the Request, which is generally understood as a method in the Controller.

Now Define Controller classes inside our Spring Boot MVC application:

- >Create a controller class : IphoneController.java

```
package com.apple.iphone.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class IphoneController {

    @GetMapping("/message")
    @ResponseBody
    public String printIphoneMessage() {
        //Logic of Method
        return " Welcome to Iphone World.";
    }

    @GetMapping("/cost")
    @ResponseBody
    public String printIphone14Cost() {
        return " Price is INR : 150000";
    }
}
```

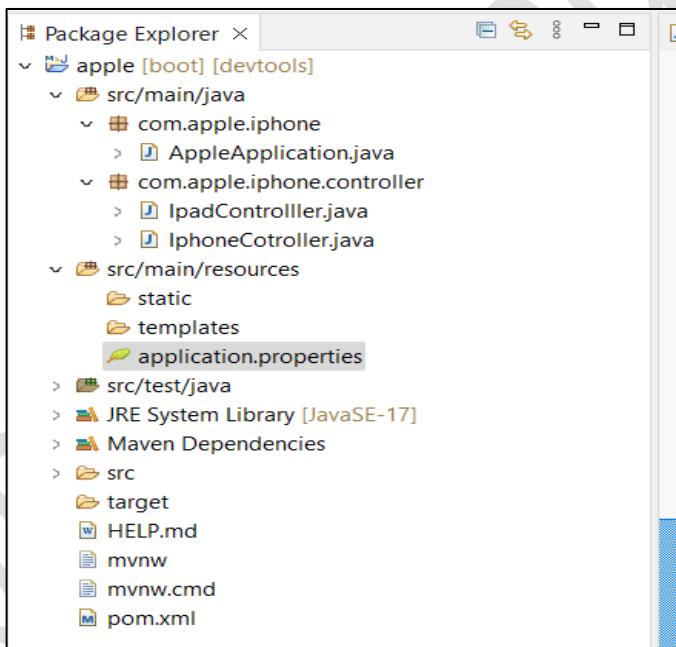
✍ **Create another Controller class : IpadController.java**

```
package com.apple.iphone.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class IpadController {
    @GetMapping("/ipad/cost")
    @ResponseBody
    public String printIPadCost() {
        return " Ipad Price is INR : 200000";
    }
}
```

Project Folder and File Structure:

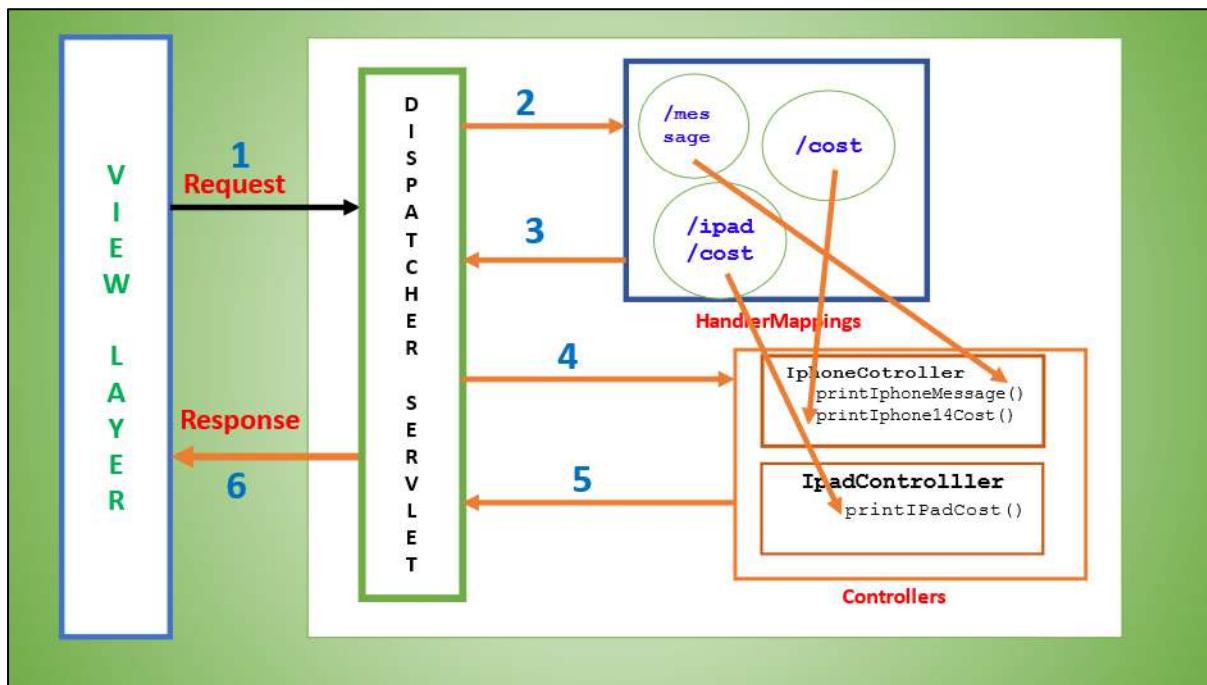


Now when we start our project as Spring Boot Application, Internally Project deployed to tomcat server and below steps will be executed.

- When we are started/deployed out application, Spring MVC internally creates **WebApplicationContext** i.e. Spring Container to instantiate and manage all Spring Beans associated to our project.
- Spring instantiates Pre-Defined Front Controller class called as **DispatcherServlet** as well as **WebApplicationContext** scans all our packages for **@Component**, **@Controller** etc.. and other Bean Configurations.

- Spring MVC **WebApplicationContext** will scan all our Controller classes which are marked with **@Controller** and starts creating Handler Mappings of all URL patterns defined in side controller classes with Controller and endpoint method names mappings.

In our App level, we created **2 controller classes** with **total 3 endpoints/URL-patterns**.



After Starting our Spring Boot Application, when we are sending a request, Following is the sequence of events happens corresponding to an incoming HTTP request to **DispatcherServlet**:

For example, we sent a request to our endpoint from browser:

<http://localhost:6655/apple/message>

- After receiving an HTTP request, **DispatcherServlet** consults the **HandlerMappings** by passing URI (**/message**) and HTTP Method type GET.
- Then **HandlerMappings** will checks all mappings information with above Details, If details are mapped then **HandlerMapping** will returns Controller Class Name and Method Name.
- If Details are not mapped/found in mappings, then **HandlerMappings** will provide an error message to **DispatcherServlet** with Error Details.
- After **DispatcherServlet** Receiving appropriate Controller and its associated method of endpoint URI, then **DispatcherServlet** forwards all request body and parameters to controller method and executes same.
- The Controller takes the request from **DispatcherServlet** and calls the appropriate service methods.
- The service method will set model data based on defined business logic and returns result or response data to Controller and from Controller to **DispatcherServlet**.
- If We configured **ViewResolver**, The **DispatcherServlet** will take help from **ViewResolver** to pick up the defined view i.e. JSP files to render response of for that specific request.

- Once view is finalized, The **DispatcherServlet** passes the model data to the view which is finally rendered on the browser.
- **If no ViewResolver configured** then Server will render the response on Browser or ANY Http Client as default test/JSON format response.

NOTE: As per REST API/Services, we are not integrating Frontend/View layer with our controller layer i.e. We are implementing individual backend services and shared with Frontend Development team to integrate with Our services. Same Services we can also share with multiple third party applications to interact with our services to accomplish the task. So We are continuing our training with REST services implantation point of view because in Microservices Architecture communication between multiple services happens via REST APIS integration across multiple Services.

Controller Class:

In Spring Boot, the controller class is responsible for processing incoming HTTP web requests, preparing a model, and returning the view to be rendered as a response on client. The controller classes in Spring are annotated either by the **@Controller** or the **@RestController** annotation.

@Controller: `org.springframework.stereotype.Controller`

The **@Controller** annotation is a specialization of the generic stereotype **@Component** annotation, which allows a class to be recognized as a Spring-managed component. **@Controller** annotation indicates that the annotated class is a controller. It is a specialization of **@Component** and is autodetected through class path/component scanning. It is typically used in combination with annotated handler methods based on the **@RequestMapping** annotation.

@ResponseBody:

Package: `org.springframework.web.bind.annotation.ResponseBody`

We annotated the request handling method with **@ResponseBody**. This annotation enables automatic serialization of the return object into the **HttpServletResponse**. This indicates a method return value should be bound to the web response i.e. **HttpServletResponse** body. Supported for annotated handler methods. The **@ResponseBody** annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the **HttpServletResponse** object.

@RequestMapping:

Package: `org.springframework.web.bind.annotation.RequestMapping`

This Annotation for mapping web requests onto methods in request-handling classes i.e. controller classes with flexible method signatures. **@RequestMapping** is Spring MVC's most common and widely used annotation.

This Annotation has the following optional attributes.

Attribute Name	Data Type	Description
name	String	Assign a name to this mapping.
value	String[]	The primary mapping expressed by this annotation.
method	RequestMethod[]	The HTTP request methods to map to, narrowing the primary mapping: GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE.
headers	String[]	The headers of the mapped request, narrowing the primary mapping.
path	String[]	The path mapping URIs (e.g. "/profile").
consumes	String[]	media types that can be consumed by the mapped handler. Consists of one or more media types one of which must match to the request Content-Type header. code examples: consumes = "text/plain" consumes = {"text/plain", "application/*"} consumes = MediaType.TEXT_PLAIN_VALUE
produces	String[]	mapping by media types that can be produced by the mapped handler. Consists of one or more media types one of which must be chosen via content negotiation against the "acceptable" media types of the request. code examples: produces = "text/plain" produces = {"text/plain", "application/*"} produces = MediaType.TEXT_PLAIN_VALUE produces = "text/plain; charset=UTF-8"
params	String[]	The parameters of the mapped request, narrowing the primary mapping. Same format for any environment: a sequence of "myParam=myValue" style expressions, with a request only mapped if each such parameter is found to have the given value.

Note: This annotation can be used both at the class and at the method level. In most cases, at the method level, applications will prefer to use one of the HTTP method specific variants **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**.

 **Example:** `@RequestMapping` without any attributes with method level

```
package com.apple.iphone.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class IphoneController {
    @RequestMapping("/message")
    @ResponseBody
    public String printIphoneMessage() {
        return " Welcome to Iphone World.";
    }
}
```

@RequestMapping("/message"):

1. If we are not defined in **method** type attribute and value, then same handler method will be executed for all HTTP methods along with endpoint.
2. `@RequestMapping("/message")` is equivalent to `@RequestMapping(value="/message")` or `@RequestMapping(path="/message")`
i.e. **value** and **path** are same type attributes to configure URI path of handler method. We can use either of them i.e. **value** is an alias for **path**.

 **Example :** With method attribute and one value:

```
@RequestMapping(value="/message", method = RequestMethod.GET)
@ResponseBody
public String printIphoneMessage() {
    return " Welcome to Iphone World.";
}
```

Now above handler method will work only for HTTP **GET** request call. If we try to request with any HTTP methods other than **GET**, we will get error response as

```
"status": 405,
"error": "Method Not Allowed",
```

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Oct 03 17:35:31 IST 2023

There was an unexpected error (type=Method Not Allowed, status=405).

 **Example :** method attribute having multiple values i.e. Single Handler method

```

@RequestMapping(value="/message",
                method = {RequestMethod.GET, RequestMethod.POST})
@ResponseBody
public String printIphoneMessage() {
    return " Welcome to Iphone World.";
}

```

Now above handler method will work only for HTTP **GET** and **POST** requests calls. If we try to request with any HTTP methods other than **GET, POST** we will get error response as:

```

"status": 405,
"error": "Method Not Allowed",

```

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Oct 03 17:35:31 IST 2023

There was an unexpected error (type=Method Not Allowed, status=405).

i.e. we can configure one URL handler method with multiple HTTP methods request.

 **Example :** With Multiple URI values and method values:

```

@RequestMapping(value = { "/message", "/msg/iphone" },
                method = { RequestMethod.GET, RequestMethod.POST })
@ResponseBody
public String printIphoneMessage() {
    return " Welcome to Iphone World.";
}

```

Above handler method will support both **GET** and **POST** requests of URI's mappings **"/message", "/msg/iphone"**.

RequestMethod:

RequestMethod is Enumeration(Enum) of HTTP request methods. Intended for use with the **RequestMapping.method()** attribute of the **RequestMapping** annotation.

ENUM Constant Values : GET, POST, PUT, DELETE, HEAD, OPTIONS, PATCH, TRACE

 **Example :** multiple Handler methods with same URI and different HTTP methods.

We can Define Same URI with multiple different handler/controller methods for different HTTP methods. Depends on incoming HTTP method request type specific handler method will be executed.

```

@RequestMapping(value = "/mac", method = RequestMethod.GET)
@ResponseBody
public String printMacMessage() {
    return " Welcome to MAC World.";
}

@RequestMapping(value = "/mac", method = RequestMethod.POST)
@ResponseBody
public String printMac2Message() {
    return " Welcome to MAC 2 World.";
}

```

Declaring @RequestMapping at Class Level:

We can use **@RequestMapping** with class definition level to create the base URI of that specific controller i.e. All URI mappings of that controller will be preceded with class level URI value always.

For example:

```

package com.apple.iphone.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/ipad")
public class IpadController {

    @GetMapping("/cost")
    @ResponseBody
    public String printIPadCost() {
        return " Ipad Price is INR : 200000";
    }

    @GetMapping("/model")
    @ResponseBody
    public String printIPadModel() {

```

```

        return " Ipad Model is 2023 Mode";
    }
}

```

From above example, class level Request mapping value ("`/ipad`") will be base URI for all handler method URI values. Means All URIs starts with `/ipad` of the controller URI's as shown below.

`http://localhost:6655/apple/ipad/model`
`http://localhost:6655/apple/ipad/cost`

@GetMapping:

Package: `org.springframework.web.bind.annotation.GetMapping`

This Annotation used for mapping HTTP **GET** requests onto specific handler methods. The **@GetMapping** annotation is a composed version of **@RequestMapping** annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.GET)`.

The **@GetMapping** annotated methods handle the HTTP GET requests matched with the given URI value.

Similar to this annotation, we have other Composed Annotations to handle different HTTP methods.

@PostMapping:

This Annotation used for mapping HTTP POST requests onto specific handler methods. **@PostMapping** is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.POST)`.

@PutMapping:

This Annotation used for mapping HTTP PUT requests onto specific handler methods. **@PutMapping** is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.PUT)`.

@DeleteMapping:

This Annotation used for mapping HTTP DELETE requests onto specific handler methods. **@DeleteMapping** is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.DELETE)`.

View Layer / JSP files Integration in Spring Boot MVC:

- Create A Spring Boot Web Application
- By default embedded tomcat server will not support JSP functionalities inside a Spring Boot MVC application. So, In order to work with JSP, we need to add below dependency in Spring boot MVC.

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

JSP ViewResolver Configuration in application.properties file:

What is ViewResolver?

In Spring MVC, a **ViewResolver** is an essential component responsible for resolving logical view names returned by controller methods into actual view implementations that can be rendered and returned to the client. It plays a crucial role in the web application's flow by mapping logical view names to views, which can be JSP pages, HTML templates, or any other type of view technology supported by Spring.

InternalResourceViewResolver:

InternalResourceViewResolver is a class in the Spring Framework used for view resolution in a Spring web application. It's typically used when you are working with Java Server Pages (JSP) as your view technology. **InternalResourceViewResolver** helps map logical view names returned by controllers to physical JSP files within your application.

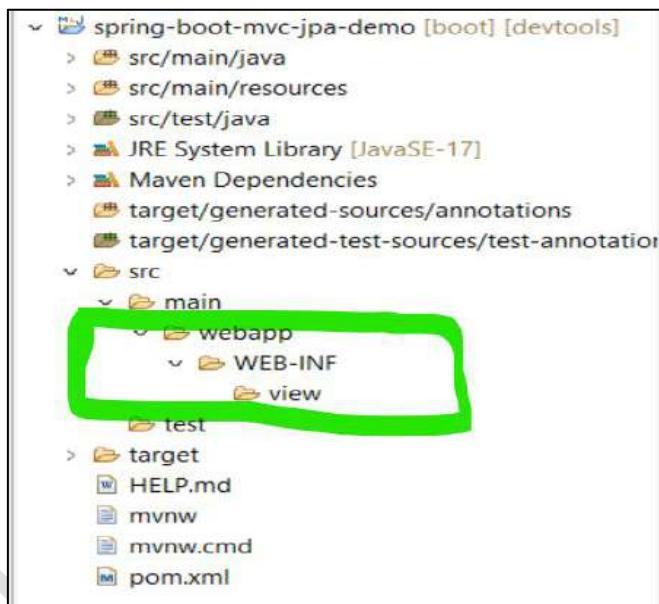
Here's how **InternalResourceViewResolver** works:

1. **Controller Returns a Logical View Name:** In a Spring web application, when a controller method processes an HTTP request and returns a logical view name (e.g., "home" or "dashboard"), this logical view name is returned to the Spring MVC framework.
2. **InternalResourceViewResolver Resolves the View:** The **InternalResourceViewResolver** is configured in the Spring application context, and it is responsible for resolving logical view names. It combines a prefix and suffix with the logical view name to construct the actual path to the JSP file. For example, if the prefix is **"/WEB-INF/view/"** and the suffix is **".jsp"**, and the logical view name is **"home"**, then the resolver constructs the view path as **"/WEB-INF/view/home.jsp"**
3. **JSP Is Rendered:** Once the view path is resolved, the JSP file at that path is executed. Any dynamic data is processed, and the JSP generates HTML content that is sent as a response to the client's browser.

- Add below View resolver properties in **application.properties** file to configure view names i.e. JSP files.

```
spring.mvc.view.prefix=/WEB-INF/view/
spring.mvc.view.suffix=.jsp
```

- Based on above configuration of property **prefix** value, we have to create folders inside our Spring Boot MVC application.
- Create a folder **webapp** inside **src -> main**
- Inside **webapp**, create another folder **WEB-INF**
- Inside **WEB-INF**, create another folder **view**
- Inside **view** Folder, We will create our JSP files.



- Now create a JSP file inside view folder and invoke it from Controller Method.

Create JSP file : **hello.jsp**

```
<html>
<head>
<title>Spring Boot MVC</title>
</head>
<body>
    <h1>Welcome to Spring Boot MVC with JSP</h1>
</body>
</html>
```

- Now create a controller method and invoke above jsp file.

```
package com.facebook.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

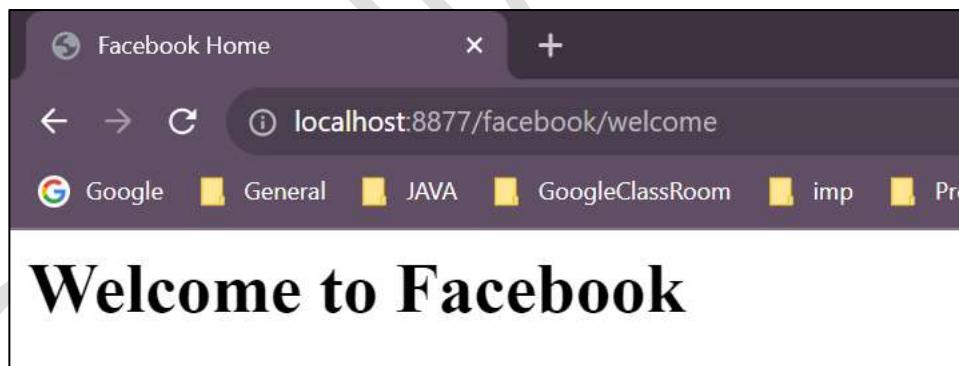
@Controller
public class UserController {
    @GetMapping("/welcome")
    public String sayHello() {
        return "hello";
    }
}
```

Testing: Send a Request to above URI method:

- Internally, **DispatcherServlet** will forwards the request to jsp file as per our Internal Resource View Resolver configuration data, i.e. inside folder **/WEB-INF/view/** with suffix **.jsp** by including jsp file name **“hello”**.

prefix + view name + suffix = /WEB-INF/view/hello.jsp

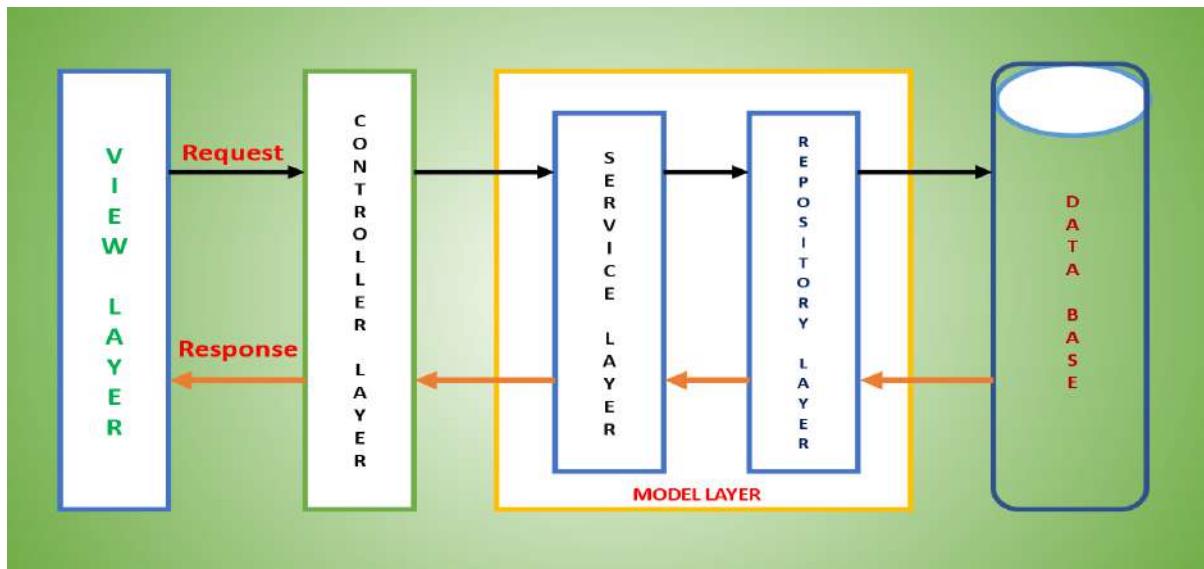
- Above JSP loaded as Response now in client/browser side.



Service Layer in Spring MVC:

A service layer is a layer in an application that facilitates communication between the controller and the persistence/repository layer. Additionally, business logic should be written inside service layer. It defines which functionalities you provide, how they are accessed, and what to pass and get in return. Even for simple CRUD cases, introduce a service layer, which at least translates from DTOs to Entities and vice versa. A Service Layer defines an application's boundary and its set of available operations from the perspective of interfacing

client layers. It encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations.



Spring MVC application Architecture

We are going to annotate with **@Service** is annotated on class to say spring, this is my Service Layer.

Create An Example with Service Layer:

Controller Class:

```
@Controller
@RequestMapping("/admission")
public class UniversityAdmissionsController {

    //Logic

}
```

Service Class:

```
@Service
public class UniversityAdmissionsService {

    //Logic

}
```

Now integrate Service Layer class with Controller Layer i.e. injecting Service class Object into Controller class Object. So we will use **@Autowired** annotation to inject service in side controller.

```

@Controller
public class UniversityAdmissionsController {

    //Injecting Service Class Object : Dependency Injection
    @Autowired
    UniversityAdmissionsService service;

    //Logic
}

```

From above, We are integrated controller with service layer. Now inside Service class, we will write Business Logic and then data should be passed to persistence layer.

Now return values of service class methods are passed to Controller class level. This is how we are using service layer with controller layer. Now we should integrate Service layer with DAO Layer to Perform DB operations. We will have multiple examples together of all three layer.

Repository Layer:

Repository Layer is mainly used for managing the data with database in a Spring Application. A huge amount of code is required for working with the databases, which can be easily reduced by Spring Data modules. It consists of JPA and JDBC modules. There are many Spring applications that use JPA technology, so these development procedures can be easily simplified by Spring Data JPA. As we discussed earlier in JPA functionalities, Now we have to integrate JPA Module to our existing application.

Repository Interface:

```

@Repository
public interface AdmissionsRepository extends JpaRepository {
    //JPA Methods
}

```

Repository Integration with Service Class:

```

@Service
public class UniversityAdmissionsService {

    @Autowired
    AdmissionsRepository repository;

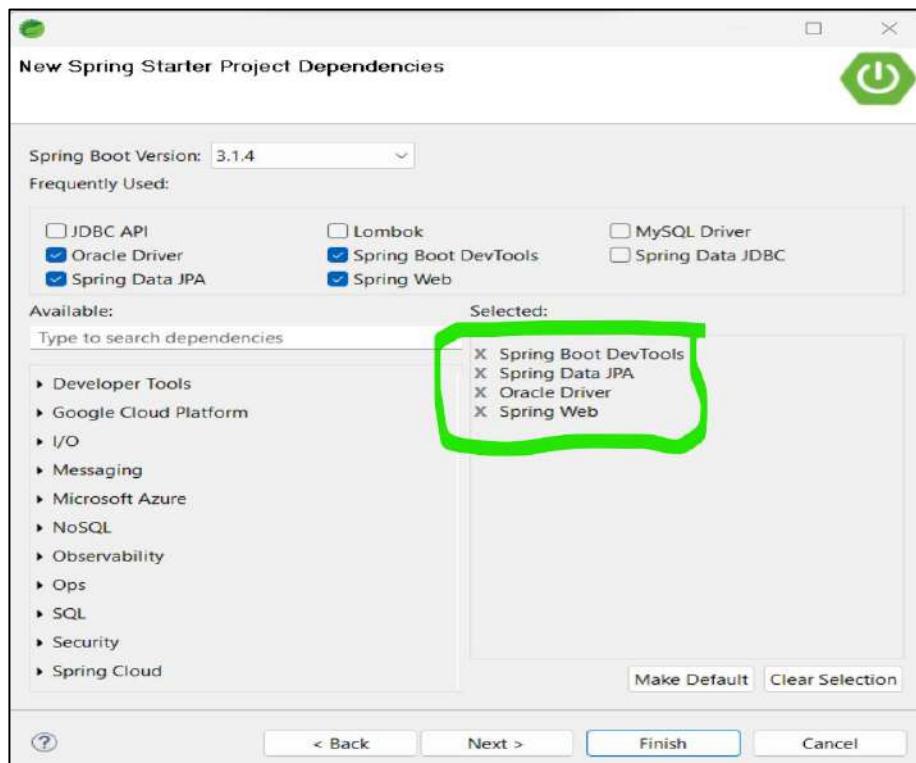
    //Logic
}

```

Spring Boot Web/MVC + JSP+ JPA Example:

Requirement: Create a Project and implement User Registration and Login Flows.

- Create a Spring Project with Web and JPA Modules.



- Add Database details and server port details inside **application.properties**.

```
server.port=9999
server.servlet.context-path=/tekteacher

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

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none
```

- Now Add Dependency of JSP inside **pom.xml** file.

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

- Now Add **ViewResolver** properties of JSP inside **application.properties** file.

```

server.port=9999
server.servlet.context-path=/tekteacher

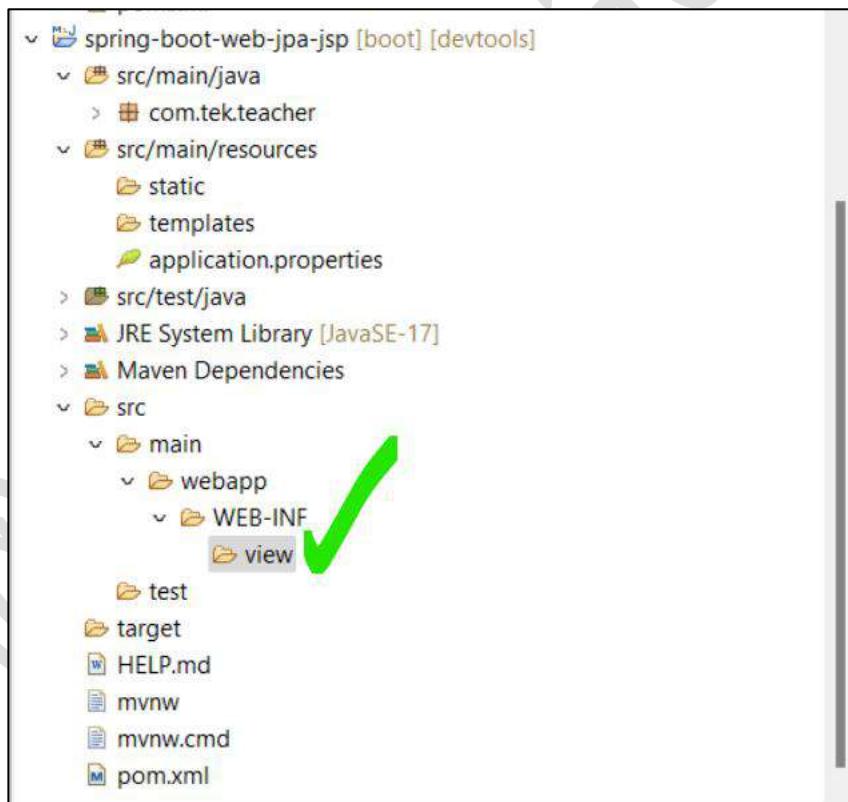
#DB Properties.
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip

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

#JSP
spring.mvc.view.prefix=/WEB-INF/view/
spring.mvc.view.suffix=.jsp

```

- Create **view** folder as per prefix value inside our application.



- >Create a JSP file for User Registration Form User Interface : **user-register.jsp**

```

<html>
<head>
    <title> User Register</title>
</head>
<body>
    <form action="user/register" method="POST">
        Name : <input type="text" name="name" /><br />
        Email Id : <input type="text" name="email" /><br />
        Contact Number : <input type="text" name="contact" /><br />
        Password : <input type="password" name="pwd" /><br />
        <input type="submit" value="Register" /><br />
    </form>
</body>
</html>

```

- Create another JSP file for User Registration Result Message, whether User Account Created or Not : **result.jsp**

```

<html>
<head>
    <title> Result</title>
</head>
<body>
    ${message}
</body>
</html>

```

- Create a DTO class for retrieving details from **HttpServletRequest** Object in side Controller method.

```

package com.tek.teacher.dto;

public class UserReigtserDto {

    private String name;
    private String emailId;
    private String contact;
    private String password;

    public String getName() {
        return name;
    }

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

```

```

    }
    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
    public String getContact() {
        return contact;
    }
    public void setContact(String contact) {
        this.contact = contact;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}

```

- Create Controller class and Methods for loading User Registration Page and reading data from Registration page. Once Receiving Data at controller we should store it inside database.

Controller Class : **UserController.java**

```

package com.tek.teacher.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.servlet.ModelAndView;
import com.tek.teacher.dto.UserReigtserDto;
import com.tek.teacher.service.UserService;
import jakarta.servlet.http.HttpServletRequest;

@Controller
public class UserController {

    @Autowired
    UserService userService;
}

```

```

//for loading HTML UI Form
@GetMapping("register")
public String sayHello() {
    return "register";
}

// From Action Endpoint for User Registration
@PostMapping("user/register")
public ModelAndView registerUser(HttpServletRequest request) {

    //Extracting Data From HttpServletRequest to DTO
    UserReigtserDto userReigtserDto = new UserReigtserDto();
    userReigtserDto.setName(request.getParameter("name"));
    userReigtserDto.setEmailId(request.getParameter("email"));
    userReigtserDto.setContact(request.getParameter("contact"));
    userReigtserDto.setPassword(request.getParameter("pwd"));

    String result = userService.userRegistration(userReigtserDto);

    ModelAndView modelAndView = new ModelAndView();
    //setting result jsp file name
    modelAndView.setViewName("result");
    modelAndView.addObject("message", result);

    return modelAndView;
}

```

- Now Create Service Layer class and respective method for storing User Information inside database.

```

package com.tek.teacher.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.tek.teacher.dto.UserReigtserDto;
import com.tek.teacher.entity.UsersInfo;
import com.tek.teacher.repository.UserRepository;

@Service
public class UserService {

    @Autowired
    UserRepository repository;

    public String userRegistration(UserReigtserDto userReigtserDto) {

```

```

    // convert dto instance to entity object
    UsersInfo user = new UsersInfo();
    user.setContact(userReigtserDto.getContact());
    user.setEmailId(userReigtserDto.getEmailId());
    user.setName(userReigtserDto.getName());
    user.setPassword(userReigtserDto.getPassword());
    repository.save(user);

    return "User Registration Successful.";
}
}

```

- Now create JPA Entity class for Database Operations, with columns related to User Details.

```

package com.tek.teacher.entity;

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

@Entity
@Table
public class UsersInfo{

    @Id
    @Column
    private String emailId;

    @Column
    private String name;

    @Column
    private String contact;

    @Column
    private String password;

    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
    public String getName() {
        return name;
    }
}

```

```

    }
    public void setName(String name) {
        this.name = name;
    }
    public String getContact() {
        return contact;
    }
    public void setContact(String contact) {
        this.contact = contact;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}

```

- Now create a JPA Repository.

```

package com.tek.teacher.repository;

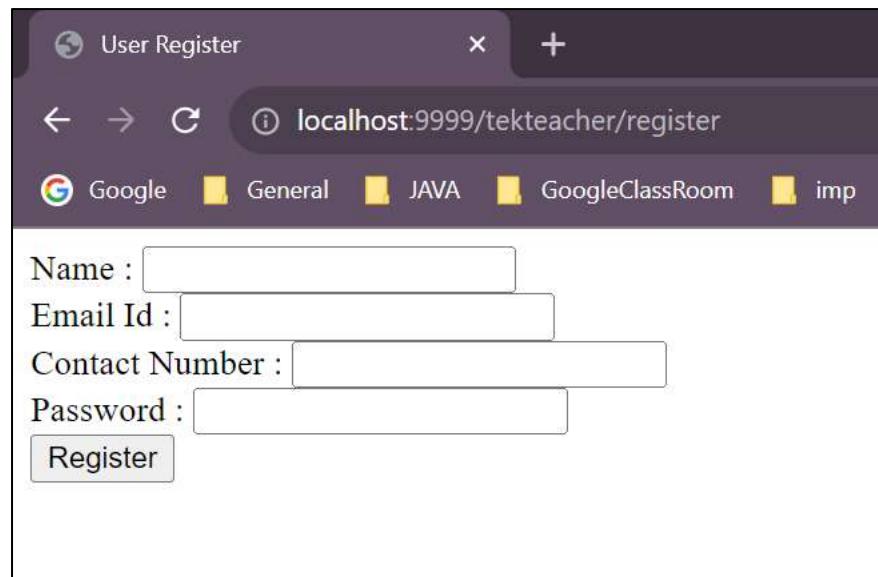
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.tek.teacher.entity.UserInfo;

@Repository
public interface UserRepository extends JpaRepository<UserInfo, String>{
}

```

Testing/Execution: Start Our Spring Boot Application

- Now Access register URI : <http://localhost:9999/tekteacher/register>
- This URL will load Form for entering User Details as followed.



User Register

localhost:9999/tekteacher/register

Name :

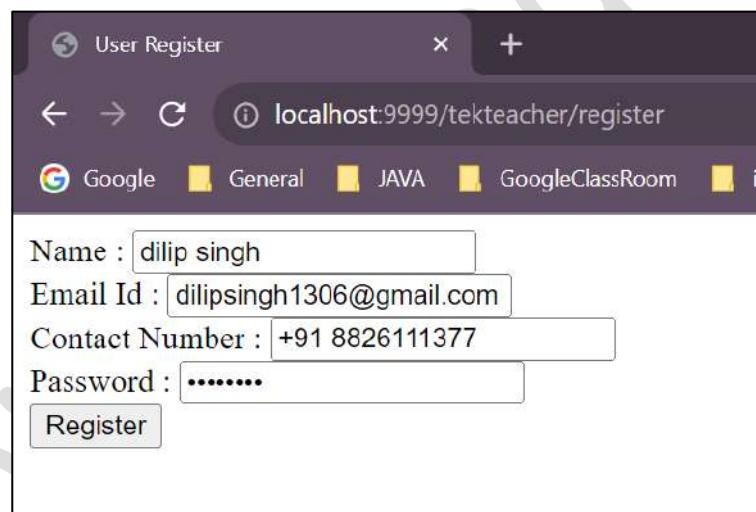
Email Id :

Contact Number :

Password :

Register

- Now Enter User Information and then click on Register button. Internally it will trigger another endpoint “**user/register**”



User Register

localhost:9999/tekteacher/register

Name : dilip singh

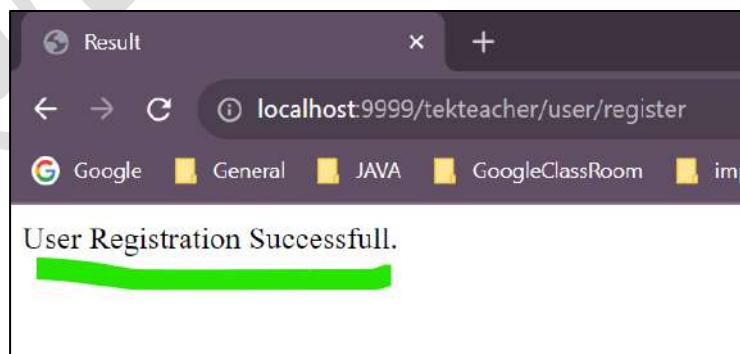
Email Id : dilipsingh1306@gmail.com

Contact Number : +91 8826111377

Password :

Register

Response :



Result

localhost:9999/tekteacher/user/register

User Registration Successfull.

Finally User Information Successfully Stored Inside Database.

Requirement: Login of User

 **Create Login UI Form : login.jsp**

```

<html>
<head>
<title>Login User</title>
</head>
<body>
    <form action="/loginCheck" method="POST">
        Email : <input type="text" name="email" /> <br />
        Password : <input type="password" name="pwd" /> <br />
        <input type="submit" value="Login" /> <br />
    </form>
</body>
</html>

```

 **Now Create Controller Method For Login Page/Form Loading.**

```

@GetMapping("login")
public ModelAndView loadLoginPage() {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("login");
    return modelAndView;
}

```

 **Now Create Controller Method For Receiving Login Form Data and validation of User Details.**

```

@PostMapping("/loginCheck")
public ModelAndView validateUser(HttpServletRequest request) {

    String result = userService.validateUser(request.getParameter("email"),
                                              request.getParameter("pwd"));

    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("result");
    modelAndView.addObject("message", result);
    return modelAndView;
}

```

 **Create Another Method in Service Class to connect with Repository layer**

```
public String validateUser(String emailId, String password) {
    // Verify in data base
    List<FacebookUsers> users = repository.findByEmailIdAndPassword(emailId,
                                                                    password);
    if (users.size() == 0) {
        return "Invalid Credentials. Please Try again";
    } else {
        return "Welcome to FaceBook, " + emailId;
    }
}
```

 **Create a custom findBy.. JPA method inside Repository Interface.**

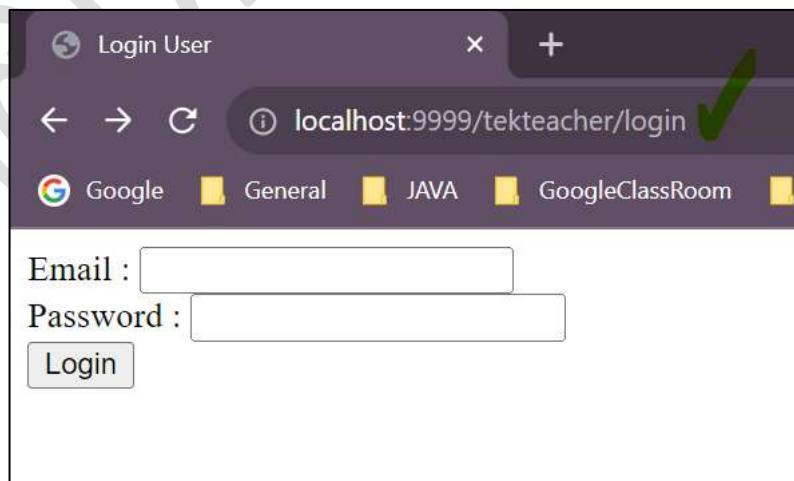
```
package com.facebook.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.facebook.entity.FacebookUsers;

@Repository
public interface UserRepository extends JpaRepository<FacebookUsers, String>{

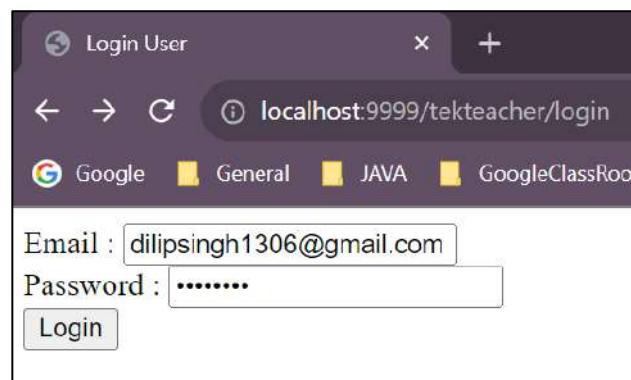
    List<FacebookUsers> findByEmailIdAndPassword(String emailId, String password);
}
```

Testing: Load Login Form.



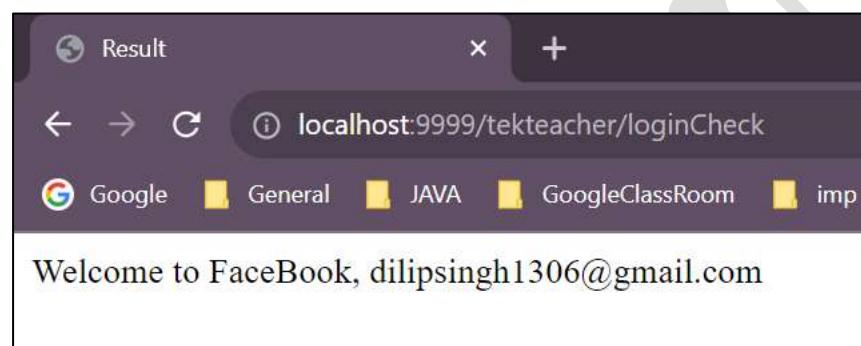
Now enter User Login Information.

Valid User Credentials: Entered Valid Email Id and Password



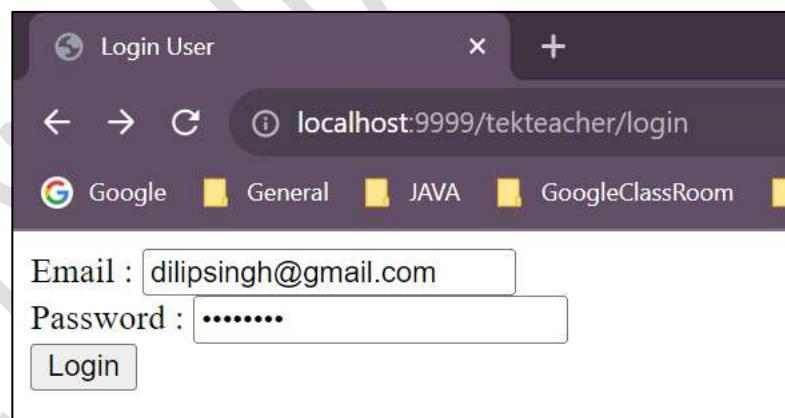
A screenshot of a browser window titled "Login User". The address bar shows "localhost:9999/tekteacher/login". The page contains a form with fields for "Email" (dilipsingh1306@gmail.com) and "Password" (redacted). A "Login" button is at the bottom.

Response :



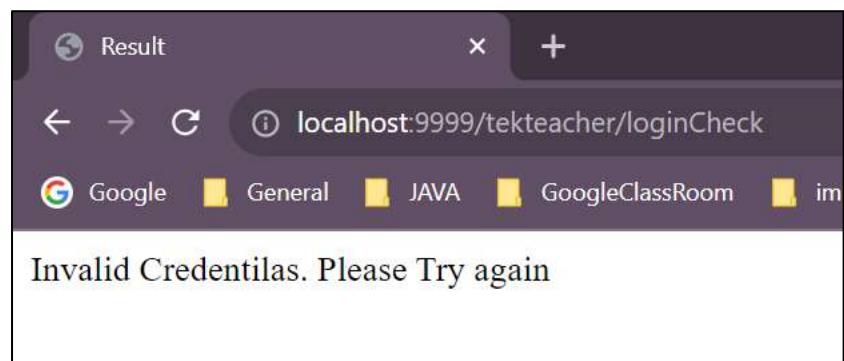
A screenshot of a browser window titled "Result". The address bar shows "localhost:9999/tekteacher/loginCheck". The page displays the message "Welcome to FaceBook, dilipsingh1306@gmail.com".

Invalid User Credentials: Entered Invalid Email Id and Password



A screenshot of a browser window titled "Login User". The address bar shows "localhost:9999/tekteacher/login". The page contains a form with fields for "Email" (dilipsingh@gmail.com) and "Password" (redacted). A "Login" button is at the bottom.

Response :



A screenshot of a browser window titled "Result". The address bar shows "localhost:9999/tekteacher/loginCheck". The page displays the message "Invalid Credentilas. Please Try again".

How to Choose HTTP methods?

Generally we will choose HTTP method depends on Data Base Operations of the requirement i.e. When we are implementing Handler methods finally as part of implantation which database query is executed as explained as follows.

CRUD Operations vs HTTP methods:

Create, Read, Update, and Delete — or **CRUD** — are the four major functions used to interact with database applications. The acronym is popular among programmers, as it provides a quick reminder of what data manipulation functions are needed for an application to feel complete. Many programming languages and protocols have their own equivalent of **CRUD**, often with slight variations in how the functions are named and what they do. For example, SQL — a popular language for interacting with databases — calls the four functions **Insert, Select, Update, and Delete**. **CRUD** also maps to the major HTTP methods.

Although there are numerous definitions for each of the **CRUD** functions, the basic idea is that they accomplish the following in a collection of data:

NAME	DESCRIPTION	SQL EQUIVALENT
Create	Adds one or more new entries	Insert
Read	Retrieves entries that match certain criteria (if there are any)	Select
Update	Changes specific fields in existing entries	Update
Delete	Entirely removes one or more existing entries	Delete

Generally most of the time we will choose HTTP methods of an endpoint based on Requirement Functionality performing which operation out of **CRUD** operations. This is a best practice of creating REST API's.

CRUD	HTTP
CREATE	POST
READ	GET
UPDATE	PUT
DELETE	DELETE

Webservices:

Web services are a standardized way for different software applications to communicate and exchange data. They enable interoperability between various systems, regardless of the programming languages or platforms they are built on. Web services use a set of protocols and technologies to enable communication and data exchange between different applications, making it possible for them to work together without any issues.

Web services are used to integrate different applications and systems, regardless of their platform or programming language. They can be used to provide a variety of services, such as:

- Information retrieval
- Transaction processing
- Data exchange
- Business process automation

There are two main types of web services:

1. SOAP (Simple Object Access Protocol) Web Services:

SOAP is a protocol for exchanging structured information using XML. It provides a way for applications to communicate by sending messages in a predefined format. SOAP web services offer a well-defined contract for communication and are often used in enterprise-level applications due to their security features and support for more complex scenarios.

2. REST (Representational State Transfer) Web Services:

REST is an architectural style that uses HTTP methods (GET, POST, PUT, DELETE) to interact with resources in a stateless manner. RESTful services are simple, lightweight, and widely used due to their compatibility with the HTTP protocol. They are commonly used for building APIs that can be consumed by various clients, such as web and mobile applications. The choice of web service type depends on factors such as the nature of the application, the level of security required, the complexity of communication, and the preferred data format.

REST API:

RESTful API is an interface that two computer systems use to exchange information securely over the internet. Most business applications have to communicate with other internal and third-party applications to perform various tasks.

API: An API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information consumer. An application programming interface (API) defines the rules that you must follow to communicate with other software systems. Developers expose or create APIs so that other applications can communicate with their applications programmatically. For example, the ICICI application exposes an API that asks for banking users, Card Details , Name,

CVV etc.. When it receives this information, it internally processes the users data and returns the payment status.

REST is a set of architectural style but not a protocol or a standard. API developers can implement REST in a variety of ways. When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information or representation is delivered in one of several formats like JSON or XML via HTTP Protocol.

JSON is the most generally popular format to use because, despite its name, it's language-agnostic, as well as readable by both humans and machines.

REST API architecture that imposes conditions on how an API should work. REST was initially created as a guideline to manage communication on a complex network like the internet. You can use REST-based architecture to support high-performing and reliable communication at scale. You can easily implement and modify it, bringing visibility and cross-platform portability to any API system.

Clients: Clients are users who want to access information from the web. The client can be a person or a software system that uses the API. For example, developers can write programs that access weather data from a weather system. Or you can access the same data from your browser when you visit the weather website directly.

Resources: Resources are the information that different applications provide to their clients/users. Resources can be images, videos, text, numbers, or any type of data. The machine that gives the resource to the client is also called the server. Organizations use APIs to share resources and provide web services while maintaining security, control, and authentication. In addition, APIs help them to determine which clients get access to specific internal resources.

API developers can design APIs using several different architectures. APIs that follow the REST architectural style are called REST APIs. Web services that implement REST architecture are called RESTful web services. The term RESTful API generally refers to RESTful web APIs. However, you can use the terms REST API and RESTful API interchangeably.

The following are some of the principles of the REST architectural style:

Uniform Interface: The uniform interface is fundamental to the design of any RESTful webservice. It indicates that the server transfers information in a standard format. The formatted resource is called a representation in REST. This format can be different from the internal representation of the resource on the server application. For example, the server can store data as text but send it in an HTML representation format.

Statelessness: In REST architecture, statelessness refers to a communication method in which the server completes every client request independently of all previous requests. Clients can request resources in any order, and every request is stateless or isolated from other requests.

This REST API design constraint implies that the server can completely understand and fulfil the request every time.

Layered system: In a layered system architecture, the client can connect to other authorized intermediate services between the client and server, and it will still receive responses from the server. Sometimes servers can also pass on requests to other servers. You can design your RESTful web service to run on several servers with multiple layers such as security, application, and business logic, working together to fulfil client requests. These layers remain invisible to the client. We can achieve this as part of Micro Services Design.

What are the benefits of RESTful APIs?

RESTful APIs include the following benefits:

Scalability: Systems that implement REST APIs can scale efficiently because REST optimizes client-server interactions. Statelessness removes server load because the server does not have to store past client request information.

Flexibility: RESTful web services support total client-server separation. Platform or technology changes at the server application do not affect the client application. The ability to layer application functions increases flexibility even further. For example, developers can make changes to the database layer without rewriting the application logic.

Platform and Language Independence: REST APIs are platform and language independent, meaning they can be consumed by a wide range of clients, including web browsers, mobile devices, and other applications. As long as the client can send HTTP requests and understand the response, it can interact with a REST API regardless of the technology stack used on the server side. You can write both client and server applications in various programming languages without affecting the API design. We can also change the technology on both sides without affecting the communication.

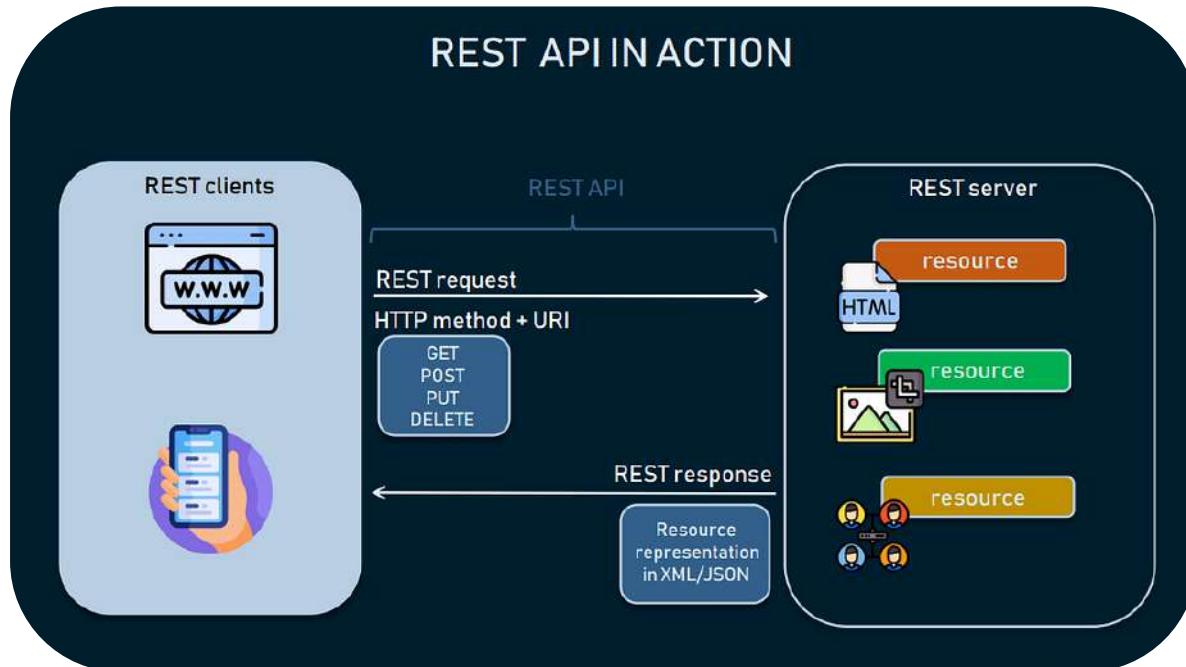
Overall, REST APIs provide a simple, scalable, and widely supported approach to building web services. These advantages in terms of simplicity, platform independence, scalability, flexibility, and compatibility make REST as a popular choice for developing APIs in various domains, from web applications to mobile apps and beyond.

How RESTful APIs work?

The basic function of a RESTful API is the same as browsing the internet. The client contacts the server by using the API when it requires a resource. API developers explain how the client should use the REST API in the server application with API documentation. These are the general steps for any REST API call integration:

1. The client sends a request to the server. The client follows the API documentation to format the request in a way that the server understands.
2. The server authenticates the client and Request and confirms that the client has the right to make that request.
3. The server receives the request and processes it internally.

4. The server returns a response to the client. The response contains information that tells the client whether the request was successful. The response also includes any information that the client requested.



The REST API request and response details are vary slightly depending on how the API developers implemented the API.

What does the RESTful API client request contain?

RESTful APIs require requests to contain the following main components:

URI (Unique Resource Identifier) : The server identifies each resource with unique resource identifiers. For REST services, the server typically performs resource identification by using a Uniform Resource Locator (URL). The URL specifies the path to the resource. A URL is similar to the website address that you enter into your browser to visit any webpage. The URL is also called the request endpoint and clearly specifies to the server what the client requires.

HTTP Method: Developers often implements RESTful APIs by using the Hypertext Transfer Protocol (HTTP). An HTTP method tells the server what it needs to do with the resource. The following are four common HTTP methods:

- **GET:** Clients use GET to access resources that are located at the specified URL on the server.
- **POST:** Clients use POST to send data to the server. They include the data representation with the request body. Sending the same POST request multiple times has the side effect of creating the same resource multiple times.

- **PUT:** Clients use PUT to update existing resources on the server. Unlike POST, sending the same PUT request multiple times in a RESTful web service gives the same result.
- **DELETE:** Clients use DELETE request to remove the resource.

HTTP Headers: Request headers are the metadata exchanged between the client and server.

Data: REST API requests might include data for the POST, PUT, and other HTTP methods to work successfully.

Parameters: RESTful API requests can include parameters that give the server more details about what needs to be done. The following are some different types of parameters:

- **Path parameters** that specify URL details.
- **Query/Request parameters** that request more information about the resource.
- **Cookie parameters** that authenticate clients quickly.

What does the RESTful API server response contain?

REST principles require the server response to contain the following main components:

Status line: The status line contains a three-digit status code that communicates request success or failure.

- 2XX codes indicate success
- 4XX and 5XX codes indicate errors
- 3XX codes indicate URL redirection.

The following are some common status codes:

- 200: Generic success response
- 201: POST method success response as Created Resource
- 400: Incorrect/Bad request that the server cannot process
- 404: Resource not found

Message body: The response body contains the resource representation. The server selects an appropriate representation format based on what the request headers contain i.e. like JSON/XML formats. Clients can request information in XML or JSON formats, which define how the data is written in plain text. For example, if the client requests the name and age of a person named John, the server returns a JSON representation as follows:

```
{
  "name": "John",
  "age": 30
}
```

Headers: The response also contains headers or metadata about the response. They give more context about the response and include information such as the server, encoding, date, and content type.

As per REST API creation Guidelines, we should choose HTTP methods depends on the Database Operation performed by our functionality, as We discussed previously.

REST Services Implementation in Spring MVC:

Spring MVC is a popular framework for creating web applications in Java. Implementing RESTful web services in Spring MVC involves using the Spring framework to create endpoints that follow the principles of the REST architectural style. It can be used to create RESTful web services, which are web services that use the REST architectural style.

RESTful services allow different software systems to communicate over the internet using standard HTTP methods, like GET, POST, PUT, and DELETE. These services are based on a set of principles that emphasize simplicity, scalability, and statelessness.

In REST Services implementation, Data will be represented as **JSON/XML** type most of the times. Now a days JSON is most popular data representational format to create and produce REST Services.

So, we should know more about JSON.

JSON:

JSON stands for **JavaScript Object Notation**. JSON is a **text format** for storing and transporting data. JSON is "self-describing" and easy to understand.

This example is a JSON string is :

```
{  
  "name": "John",  
  "age": 30,  
  "car": null  
}
```

JSON is a lightweight data-interchange format. JSON is plain text written in JavaScript object notation. JSON is used to exchange data between multiple applications/services. JSON is language independent.

JSON Syntax Rules:

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

Example: **"name" : "John"**

In JSON, values must be one of the following data types:

- a string
- a number
- an object
- an array
- a Boolean
- null

JSON vs XML:

Both JSON and XML can be used to receive data from a web server. The following JSON and XML examples both define an employee's object, with an array of 3 employees:

JSON Example

```
{
  "employees": [
    {
      "firstName": "John",
      "lastName": "Doe"
    },
    {
      "firstName": "Anna",
      "lastName": "Smith"
    },
    {
      "firstName": "Peter",
      "lastName": "Jones"
    }
  ]
}
```

XML Example:

```
<employees>
  <employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName>
```

```

<lastName>Smith</lastName>
</employee>
<employee>
  <firstName>Peter</firstName>
  <lastName>Jones</lastName>
</employee>
</employees>

```

JSON is Like XML Because

- Both JSON and XML are "self-describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages

When A Request Body Contains JSON/XML data Format, then how Spring MVC/JAVA language handling Request data?

Here, We should **Convert JSON/XML data to JAVA Object** while Request landing on Controller method, after that we are using JAVA Objects in further process. Similarly, Sometimes we have to send Response back as either JSON or XML format i.e. JAVA Objects to JSON/XML Format.

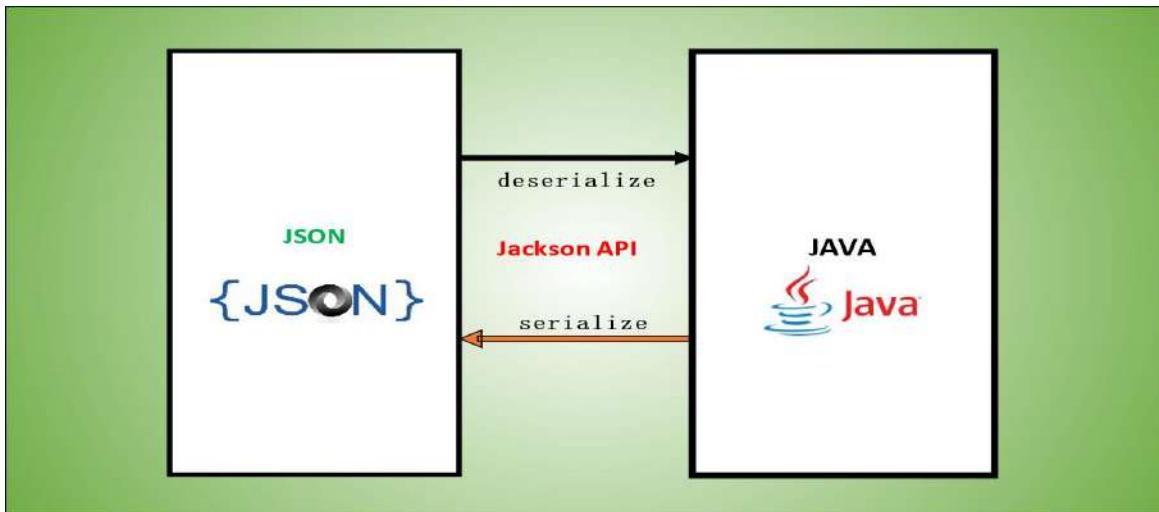
For these conversions, we have few pre-defined solutions in Market like Jackson API, GSON API, JAXB etc..

JSON to JAVA Conversion:

Spring MVC supports Jackson API which will take care of un-marshalling/mapping JSON request body to Java objects. We should add **Jackson API dependencies** explicitly In Spring MVC, If It is Spring Boot MVC application Jackson API jars will be added internally. We can use **@RequestBody** Spring MVC annotation to deserialize/un-marshall JSON string to Java object. Similarly, java method return data will be converted to JSON format i.e. Response of Endpoint by using an annotation **@ResponseBody**.

And as you have annotated with **@ResponseBody** of endpoint method, we no need to do explicitly JAVA to JSON conversion. Just return a POJO and Jackson serializer will take care of converting to Json format. It is equivalent to using **@ResponseBody** when used with **@Controller**. Rather than placing **@ResponseBody** on every controller method we place **@RestController** instead of **@Controller** and **@ResponseBody** by default is applied on all resources in that controller.

Note: we should carte Java POJO classes specific to JSON payload structure, to enable auto conversion between JAVA and JSON.



JSON with Array of String values:

JSON Payload: Below Json contains ARRY of String Data Type values

```
{
  "student": [
    "Dilip",
    "Naresh",
    "Mohan",
    "Laxmi"
  ]
}
```

Java Class: JSON Array of String will be takes as `List<String>` with JSON key name.

```
import java.util.List;

public class StudentDetails {

    private List<String> student;

    public List<String> getStudent() {
        return student;
    }

    public void setStudent(List<String> student) {
        this.student = student;
    }

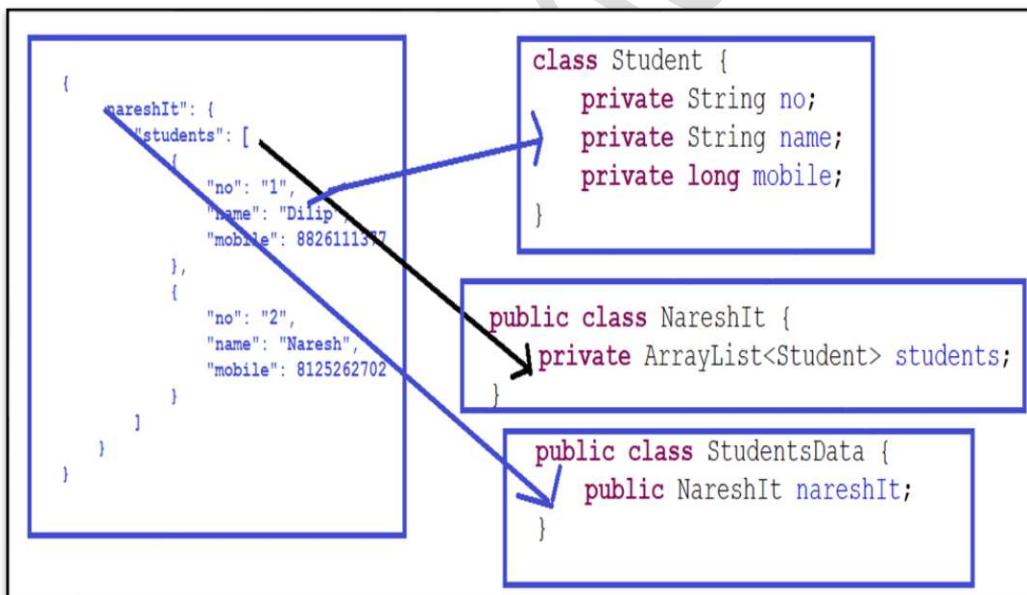
    @Override
    public String toString() {
        return "StudentDetails [student=" + student + "]";
    }
}
```

JSON payload with Array of Student Object Values:

Below JSON payload contains array of Student values.

```
{
  "nareshIt": {
    "students": [
      {
        "no": "1",
        "name": "Dilip",
        "mobile": 8826111377
      },
      {
        "no": "2",
        "name": "Naresh",
        "mobile": 8125262702
      }
    ]
  }
}
```

Below picture showing how are creating JAVA classes from above payload.



Created Three java files to wrap above JSON payload structure:

Student.java

```
public class Student {
```

```

private String no;
private String name;
private long mobile;

//Setters and Getters
}

```

Add Student as List in class **NareshIt.java**

```

import java.util.ArrayList;
public class NareshIt {
    private ArrayList<Student> students;

    //Setters and Getters

}

```

StudentsData.java

```

public class StudentsData {
    public NareshIt nareshIt;

    //Setters and Getters
}

```

Now we will use **StudentsData** class to bind our JSON Payload.

➤ [Let's Take another Example of JSON to JAVA POJO class:](#)

JSON PAYLOAD: Json with Array of Student Objects

```

{
    "student": [
        {
            "firstName": "Dilip",
            "lastName": "Singh",
            "mobile": 88888,
            "pwd": "Dilip",
            "emailID": "Dilip@Gmail.com"
        },
        {
            "firstName": "Naresh",
            "lastName": "It",
            "mobile": 232323,
            "pwd": "Naresh",
            "emailID": "Naresh@Gmail.com"
        }
    ]
}

```

}

For the above Payload, JAVA POJO'S are:

```
import com.fasterxml.jackson.annotation.JsonProperty;

public class StudentInfo {

    private String firstName;
    private String lastName;
    private long mobile;
    private String pwd;
    @JsonProperty("emailID")
    private String email;

    //Setters and Getters
}
```

Another class To Wrap above class Object as List with property name student as per JSON.

```
import java.util.List;

public class Students {

    List<StudentInfo> student;

    public List<StudentInfo> getStudent() {
        return student;
    }
    public void setStudent(List<StudentInfo> student) {
        this.student = student;
    }
}
```

From the above JSON payload and JAVA POJO class, we can see a difference for one JSON property called as **emailID** i.e. in JAVA POJO class property name we taken as **email** instead of emailID. In Such case to map JSON to JAVA properties with different names, we use an annotation called as `@JsonProperty("jsonPropertyName")`.

@JsonProperty:

The `@JsonProperty` annotation is used to specify the property name in a JSON object when serializing or deserializing a Java object using the Jackson API library. It is often used when the JSON property name is different from the field name in the Java object, or when the JSON property name is not in camelCase.

If you want to serialize this object to JSON and specify that the JSON property names should be "first_name", "last_name", and "age", you can use the `@JsonProperty` annotation like this:

```
public class Person {
    @JsonProperty("first_name")
    private String firstName;
    @JsonProperty("last_name")
    private String lastName;
    @JsonProperty
    private int age;

    // getters and setters go here
}
```

As a developer, we should always create POJO classes aligned to JSON payload to bind JSON data to Java Object with `@RequestBody` annotation.

To implement REST services in Spring MVC, you can use the `@RestController` annotation. This annotation marks a class as a controller that returns data to the client in a RESTful way.

@RestController:

Spring introduced the `@RestController` annotation in order to simplify the creation of RESTful web services. `@RestController` is a specialized version of the controller. It's a convenient annotation that combines `@Controller` and `@ResponseBody`, which eliminates the need to annotate every request handling method of the controller class with the `@ResponseBody` annotation.

Package: org.springframework.web.bind.annotation.RestController;

For example, When we mark class with `@Controller` and we will use `@ResponseBody` at request mapping method level.

`@RestController = @Controller + @ResponseBody`

```
@Controller
public class MAcBookController {

    @GetMapping(path = "/mac/details")
}
```

```

@RequestBody
public String getMacBookDetail() {
    return "MAC Book Details : Price 200000. Model 2022";
}

@GetMapping(path = "/iphone/details")
@RequestBody
public String getIphoneDetail() {
    return "Iphone Details : Price 150000. Model 15 Pro";
}

```

- If we Used **@RestController** with controller class, removed **@ResponseBody** at all handler mapping methods level.

```

@RestController
public class MAcBookController {

    @GetMapping(path = "/mac/details")
    public String getMacBookDetail() {
        return "MAC Book Details : Price 200000. Model 2022";
    }

    @GetMapping(path = "/iphone/details")
    public String getIphoneDetail() {
        return "Iphone Details : Price 150000. Model 15 Pro";
    }
}

```

That's all, **@RestController** is just like a shortcut annotation to avoid declaring **@ResponseBody** on every controller handler mapping method inside a class.

@RequestBody Annotation:

The **@RequestBody** annotation in Spring is used to bind the HTTP request body to a method parameter. This means that Spring will automatically deserialize the request body into a Java object and that object is then passed to the method as a parameter. The **@RequestBody** annotation can be used on controller methods.

For example, the following controller method accepts a User object as a parameter:

```

@PostMapping("/users")
public User createUser(@RequestBody User user) {
}

```

```
// ...  
}
```

- When you send a POST request to the `/users` endpoint with a JSON body containing the user data, Spring will automatically deserialize the JSON into a User object and pass it to the `createUser()` method.
- The `@RequestBody` annotation is a powerful tool that makes it easy to work with HTTP request bodies in Spring. It is especially useful for developing REST APIs.

Here are some additional things to keep in mind about the `@RequestBody` annotation:

- The request body must be in a supported media type, such as JSON, XML.
- Spring will use an appropriate HTTP message converter to deserialize the request body.
- If the request body cannot be serialized, Spring will throw a `HttpMessageNotReadableException`.

Postman API Testing Tool:



Postman is a popular and widely used API (Application Programming Interface) testing and development tool. It provides a user-friendly interface for sending HTTP requests to APIs and inspecting the responses. Postman offers a range of features that make it valuable for developers, testers, and anyone working with APIs:

Some of the key features of Postman API Tools include:

- **API client:** Postman provides a powerful API client that allows you to send HTTP requests to any API and inspect the responses. The API client supports a wide range of authentication protocols and response formats.
- **API testing:** Postman provides a powerful API testing framework that allows you to create and execute tests for your APIs. Postman tests can be used to validate the functionality, performance, and security of your APIs.
- **API design:** Postman can be used to design your API specifications in OpenAPI, RAML, GraphQL, or SOAP formats. Postman's schema editor makes it easy to work with specification files of any size, and it validates specifications with a built-in linting engine.

- **API documentation:** Postman can be used to generate documentation for your APIs in a variety of formats, including HTML, Markdown, and PDF. Postman documentation is automatically generated from your API requests, so it is always up-to-date.
- **API monitoring:** Postman can be used to monitor your APIs for performance and availability issues. Postman monitors can be configured to send alerts when your APIs are unavailable or when they are not performing as expected.

Postman is a powerful tool that can help you to streamline your API development workflow. It is used by developers and teams of all sizes to build, test, document, and monitor APIs.

Here are some examples of how Postman API Tools can be used:

- A developer can use Postman to explore a new API and learn how to use it.
- A QA engineer can use Postman to create and execute tests for an API.
- A DevOps engineer can use Postman to monitor an API for performance and availability issues.
- A product manager can use Postman to generate documentation for an API.
- A sales engineer can use Postman to demonstrate an API to a customer.

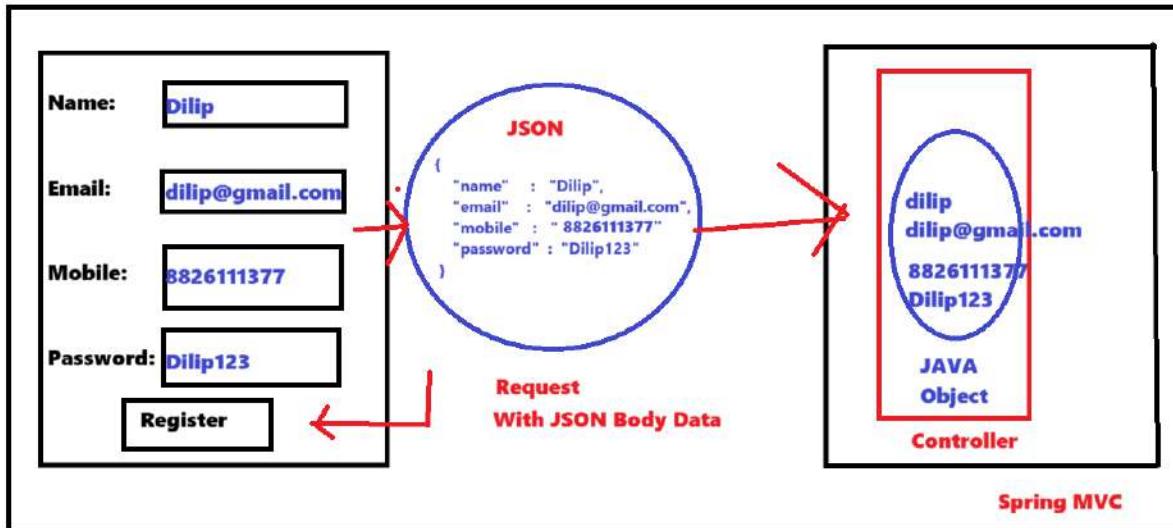
Project Setup:

1. Create Spring Boot Web Module Project
2. Now Create an endpoints or REST Services for below requirement.

Requirement: Write a Rest Service for User Registration. User Details Should Be :

- **User Name**
 - **Email Id**
 - **Mobile**
 - **Password**
- Create a JSON Mapping for above Requirement with dummy values.

```
{
  "name" : "Dilip",
  "email" : "dilip@gmail.com",
  "mobile" : "+91 73777373",
  "password" : "Dilip123"
}
```



3. Before Creating Controller class, we should Create JAVA POJO class which is compatible with JSON Request Data. So create a JAVA class, as discussed previously. Which is Responsible for holding Request Data of JSON.

```

package com.swiggy.user.request;

public class UserRegisterRequest {

    private String name;
    private String email;
    private String mobile;
    private String password;

    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 getMobile() {
        return mobile;
    }
    public void setMobile(String mobile) {
        this.mobile = mobile;
    }
    public String getPassword() {
    }
}

```

```

        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}

```

4. Now Create A controller and inside an endpoint for User Register Request Handling.

```

package com.swiggy.user.controller;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.swiggy.user.request.UserRegisterRequest;

@RestController
@RequestMapping("/user")
public class UserController {

    @PostMapping("/register")
    public String getUserDetails(@RequestBody UserRegisterRequest request){
        System.out.println(request.getEmail());
        System.out.println(request.getName());
        System.out.println(request.getPassword());
        return "User Created Successfully";
    }
}

```

In Above, We are used **@RequestBody** for binding/mapping incoming JSON request to JAVA Object at method parameter layer level. Means, Spring MVC internally maps JSON to JAVA with help of Jackson API jar files.

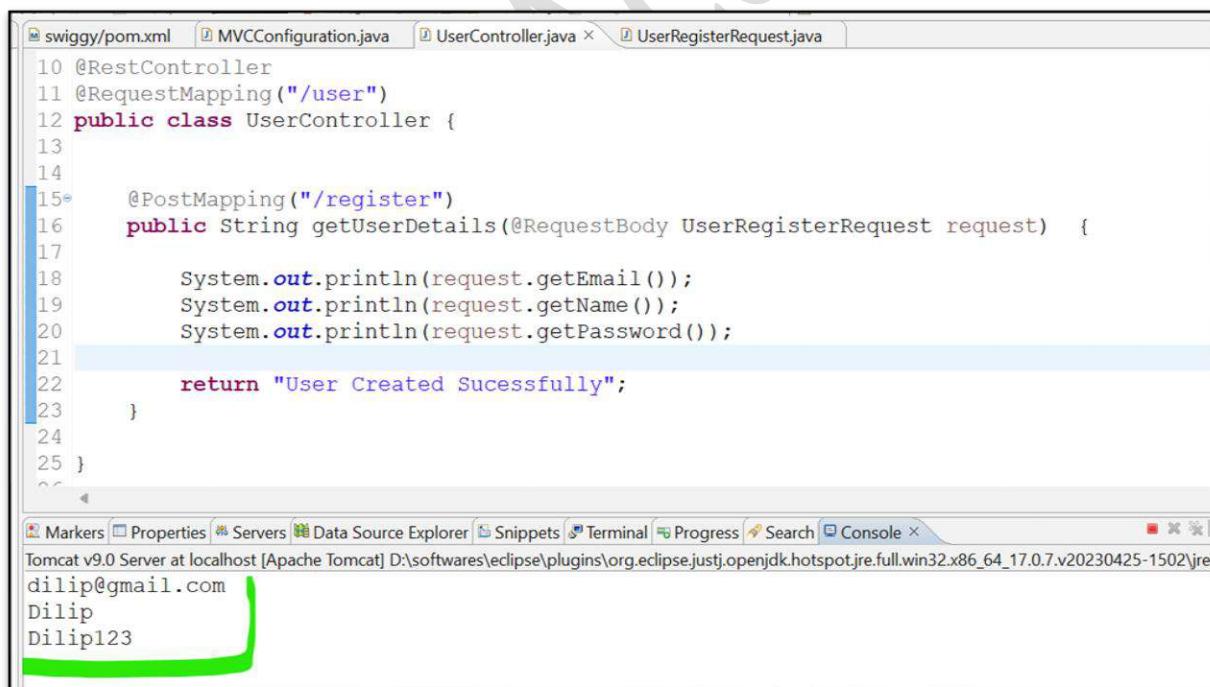
5. Deploy Your Application in Server Now. After Deployment we have to Test now weather it Service is working or not.
6. Now We are Taking Help of **Postman** to do API/Services Testing.

- Open Postman
- Now Click on Add request
- Select Your Service HTTP method
- And Enter URL of Service
- Select Body
- Select raw
- Select JSON
- Enter JSON Body as shown in Below.



After Clicking on **Send** Button, Summited Request to Spring MVC REST Service Endpoint method and we got Response back with **200 Ok** status Code.

In Below, We can See in Server Console, Request Data is printed what we Received from Client/Postman level as JSON data.



Path Variables:

Path variable is a template variable called as place holder of URI, i.e. this variable path of URI. **@PathVariable** annotation can be used to handle template variables in the request URI mapping, and set them as method parameters. Let's see how to use **@PathVariable** and its various attributes. We will define path variable as part of URI in side curly braces{}.

Package of Annotation: org.springframework.web.bind.annotation.PathVariable;

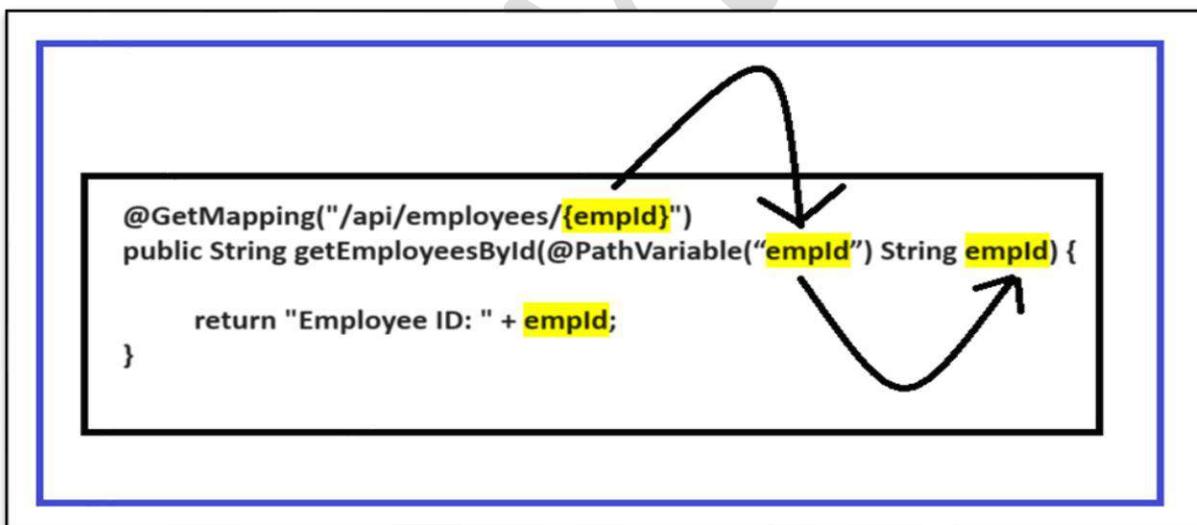
Examples:

URI with Template Path variables : /location/{locationName}/pincode/{pincode}

URI with Data replaced : /location/Hyderabad/pincode/500072

Example for endpoint URI mapping in Controller : /api/employees/{empId}

```
@GetMapping("/api/employees/{empId}")
public String getEmployeesById(@PathVariable("empId") String empId) {
    return "Employee ID: " + empId;
}
```



Example 2: Path variable Declaration



Requirement : Get User Details with User Email Id.

In these kind of requirements, like getting Data with Resource ID's. We can Use Path Variable as part of URI instead of JSON mapping and equivalent Request Body classes. So Create a REST endpoint with a Path Variable of Email ID.

UserController.java : Add Below Logic In existing User Controller.

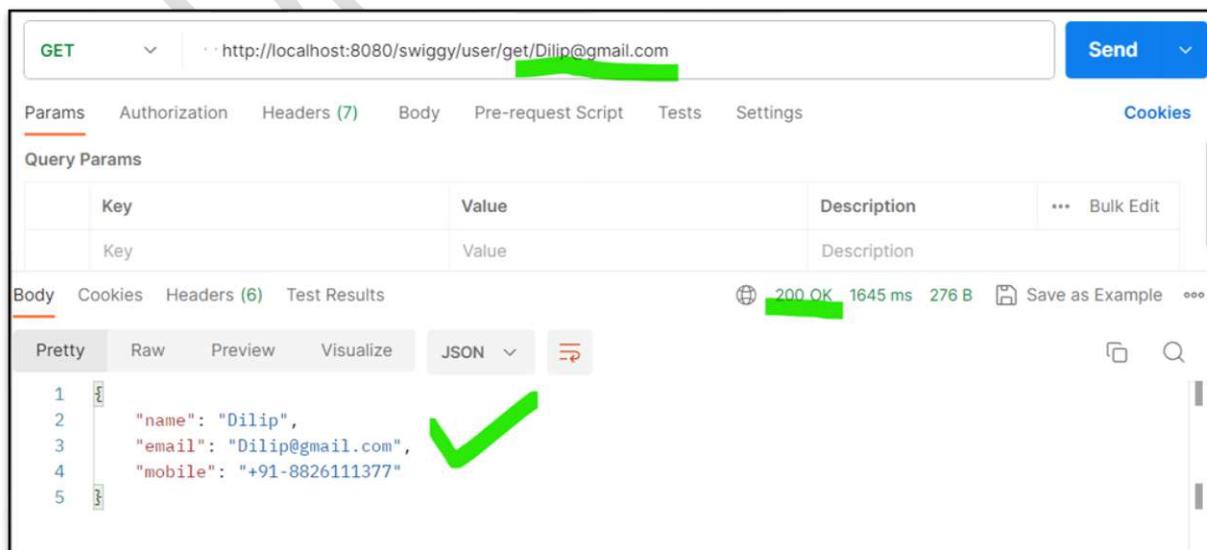
```
@RequestMapping(value = "/get/{emailId}", method = RequestMethod.GET)
public UserRegisterResponse getUserByEmailId(@PathVariable("emailId") String email) {
    return userService.getUserDetails(email);
}
```

➤ Now Add Method in Service Class for interacting with Repository Layer.

Method inside Service Class : UserService.java

```
public UserRegisterResponse getUserDetails(String email) {
    SwiggyUsers user = repository.findById(email).get();
    //Entity Object to DTO object
    UserRegisterResponse response = new UserRegisterResponse();
    response.setEmail(user.getEmail());
    response.setMobile(user.getMobile());
    response.setName(user.getName());
    return response;
}
```

Testing: Pass Email Value in place of PATH variable



The screenshot shows a Postman request for a GET endpoint. The URL is `http://localhost:8080/swiggy/user/get/Dilip@gmail.com`. The response status is 200 OK, and the response body is a JSON object:

```

1  "name": "Dilip",
2  "email": "Dilip@gmail.com",
3  "mobile": "+91-8826111377"

```

Multiple Path Variable as part of URI:

We can define more than one path variables as part of URI, then equal number of method parameters with **@PathVariable** annotation defined in handler mapping method.

NOTE: We no need to define value inside **@PathVariable** when we are taking method parameter name as it is URI template/Path variable.

Syntax : /{pathvar1}/{pathvar2}

Example: /pharmacy/{location}/pincode/{pincode}

Requirement : Add Order Details as shown in below.

- Order ID
- Order status
- Amount
- Email Id
- City

After adding Orders, Please Get Order Details based on **Email Id** and **Order Status**.

In this case, we are passing values of Email ID and Order Status to find out Order Details. Now we can take **Path variables** here to fulfil this requirement.

- Create an endpoints for adding Order Details and Getting Order Details with Email ID and Order Status.

Create Request, Response and Entity Classes.

OrderRequest.java

```
package com.swiggy.order.request;

public class OrderRequest {

    private String orderID;
    private String orderstatus;
    private double amount;
    private String emailId;
    private String city;

    public String getOrderID() {
        return orderID;
    }

    public void setOrderID(String orderID) {
        this.orderID = orderID;
    }

    public String getOrderstatus() {
        return orderstatus;
    }
}
```

```

    }
    public void setOrderstatus(String orderstatus) {
        this.orderstatus = orderstatus;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
}

```

➤ OrderResponse.java

```

package com.swiggy.order.response;

public class OrderResponse {

    private String orderID;
    private String orderstatus;
    private double amount;
    private String emailId;
    private String city;

    public OrderResponse() {
    }
    public OrderResponse(String orderID, String orderstatus, double amount,
                        String emailId, String city) {
        this.orderID = orderID;
        this.orderstatus = orderstatus;
        this.amount = amount;
        this.emailId = emailId;
        this.city = city;
    }
    public String getOrderID() {

```

```

        return orderID;
    }
    public void setOrderID(String orderID) {
        this.orderID = orderID;
    }
    public String getOrderstatus() {
        return orderstatus;
    }
    public void setOrderstatus(String orderstatus) {
        this.orderstatus = orderstatus;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
}

```

➤ Entity Class : [SwiggyOrders.java](#)

```

package com.swiggy.order.entity;

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

@Entity
@Table(name = "swiggy_orders")
public class SwiggyOrders {

    @Id
    @Column
    private String orderID;
}

```

```

    @Column
    private String orderstatus;
    @Column
    private double amount;
    @Column
    private String emailId;
    @Column
    private String city;

    public String getOrderID() {
        return orderID;
    }
    public void setOrderID(String orderID) {
        this.orderID = orderID;
    }
    public String getOrderstatus() {
        return orderstatus;
    }
    public void setOrderstatus(String orderstatus) {
        this.orderstatus = orderstatus;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
}

```

➤ **Controller class : OrderController.java**

```

package com.swiggy.order.controller;

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

```

```

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.swiggy.order.request.OrderRequest;
import com.swiggy.order.response.OrderResponse;
import com.swiggy.order.service.OrderService;

@RestController
@RequestMapping("/order")
public class OrderController {

    @Autowired
    OrderService orderService;

    @PostMapping(value = "/create")
    public String createOrder(@RequestBody OrderRequest request) {
        return orderService.createOrder(request);
    }

    @GetMapping("/email/{emailId}/status/{status}")
    public List<OrderResponse> getOrdersByemailIDAndStatus(@PathVariable String
        emailId, @PathVariable("status") String orderStatus ){
        List<OrderResponse> orders =
            orderService.getOrdersByemailIDAndStatus(emailId, orderStatus);

        return orders;
    }
}

```

➤ Now create methods in Service layer.

```

package com.swiggy.order.service;

import java.util.List;
import java.util.stream.Collectors;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.swiggy.order.entity.SwiggyOrders;
import com.swiggy.order.repository.OrderRepository;
import com.swiggy.order.request.OrderRequest;
import com.swiggy.order.response.OrderResponse;

@Service
public class OrderService {

```

```

@Autowired
OrderRepository repository;

public String createOrder(OrderRequest request) {
    SwiggyOrders order = new SwiggyOrders();
    order.setAmount(request.getAmount());
    order.setCity(request.getCity());
    order.setEmailId(request.getEmailId());
    order.setOrderID(request.getOrderID());
    order.setOrderstatus(request.getOrderstatus());
    repository.save(order);
    return "Order Created Successfully";
}

public List<OrderResponse> getOrdersByEmailIDAndStatus(String emailId,
                                                       String orderStatus) {
    List<SwiggyOrders> orders =
        repository.findByEmailIdAndOrderstatus(emailId, orderStatus);

    List<OrderResponse> allOrders = orders.stream().map(
        v -> new OrderResponse(
            v.getOrderID(),
            v.getOrderstatus(),
            v.getAmount(),
            v.getEmailId(),
            v.getCity()
        ).collect(Collectors.toList());
}

    return allOrders;
}
}

```

➤ **Create Repository : OrderRepository.java**

Add JPA Derived Query findBy() Method for Email Id and Order Status.

```

package com.swiggy.order.repository;

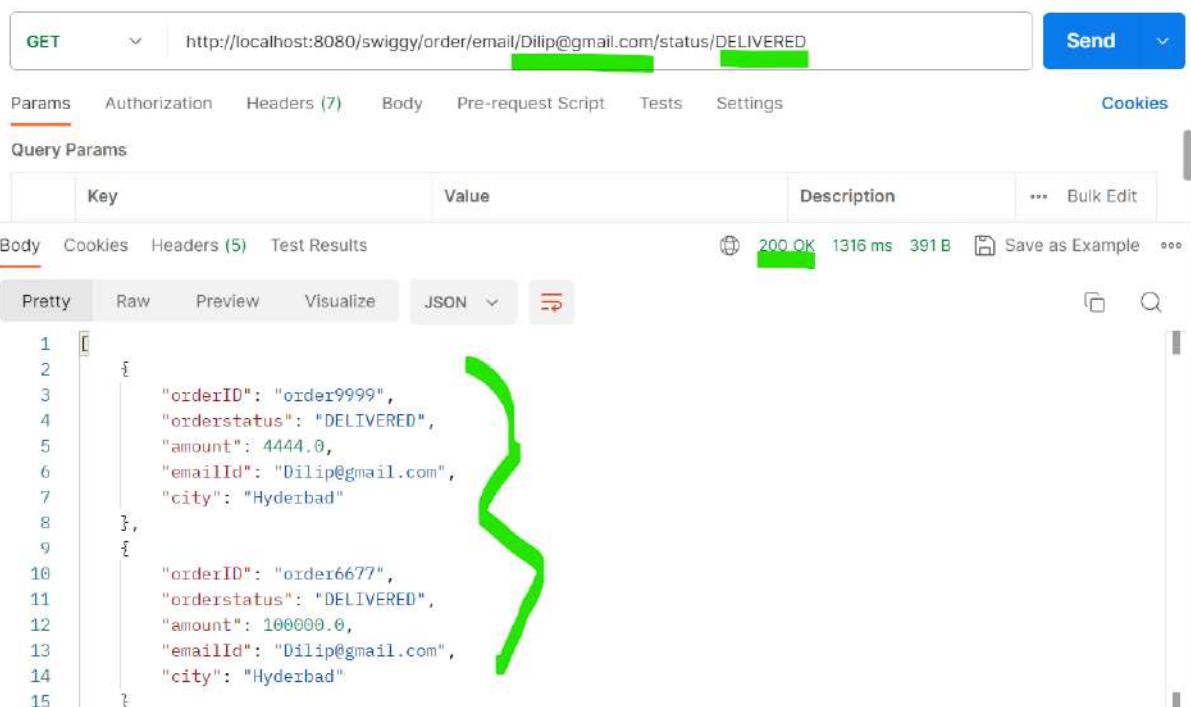
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.swiggy.order.entity.SwiggyOrders;

@Repository
public interface OrderRepository extends JpaRepository<SwiggyOrders, String>{
    List<SwiggyOrders> findByEmailIdAndOrderstatus(String emailId, String orderStatus);
}

```

{}

From Postman Test end point: URL formation, replacing Path variables with real values



The screenshot shows a Postman test endpoint. The URL is `http://localhost:8080/swiggy/order/email/Dilip@gmail.com/status/DELIVERED`. The response is a JSON array with two elements, each representing an order. The JSON structure is as follows:

```

1  [
2   {
3     "orderID": "order9999",
4     "orderstatus": "DELIVERED",
5     "amount": 4444.0,
6     "emailId": "Dilip@gmail.com",
7     "city": "Hyderabad"
8   },
9   {
10    "orderID": "order6677",
11    "orderstatus": "DELIVERED",
12    "amount": 100000.0,
13    "emailId": "Dilip@gmail.com",
14    "city": "Hyderabad"
15  }
]

```

- We can also handle more than one Path Variables of URI by using a method parameter of type `java.util.Map<String, String>`.

```

@GetMapping("/pharmacy/{location}/pincode/{pincode}")
public String getPharmacyByLocationAndPincode(@PathVariable Map<String, String>
                                             values) {
    String location = values.get("location"); // Key is Path variable
    String pincode = values.get("pincode");

    return "Location Name : " + location + ", Pin code: " + pincode;
}

```

Query String and Query Parameters:

Query string is a part of a uniform resource locator (URL) that assigns values to specified parameters. A query string commonly includes fields added to a base URL by a Web browser or other client application. Let's understand this statement in a simple way by an example. Suppose we have filled out a form on websites and if we have noticed the URL something like as shown below as follows:

http://internet.org/process-homepage?number1=23&number2=12

So in the above URL, the query string is whatever follows the question mark sign (“?”) i.e (number1=23&number2=12) this part. And “number1=23”, “number2=12” are Query Parameters which are joined by a connector “&”.

Let us consider another URL something like as follows:

http://internet.org?title=Query_string&action=edit

So in the above URL, the query string is “title=Query_string&action=edit” this part. And “title=Query_string”, “action=edit” are Query Parameters which are joined by a connector “&”.

Now we are discussing the concept of the query string and query parameter from the Spring MVC point of view. Developing Spring MVC application and will understand how query strings and query parameters are generated.

@RequestParam:

In Spring, we use **@RequestParam** annotation to extract the id of query parameters. Assume we have Users Data, and we should get data based on email Id.

Example : URL : **/details?email=<value-of-email>**

```
@GetMapping("/details")
public String getUserDetails(@RequestParam String email) {
    //Now we can pass Email Id to service layer to fetch user details
    return "Email Id of User : " + email;
}
```

Example with More Query Parameters :

Requirement: Please Get User Details by using either email or mobile number

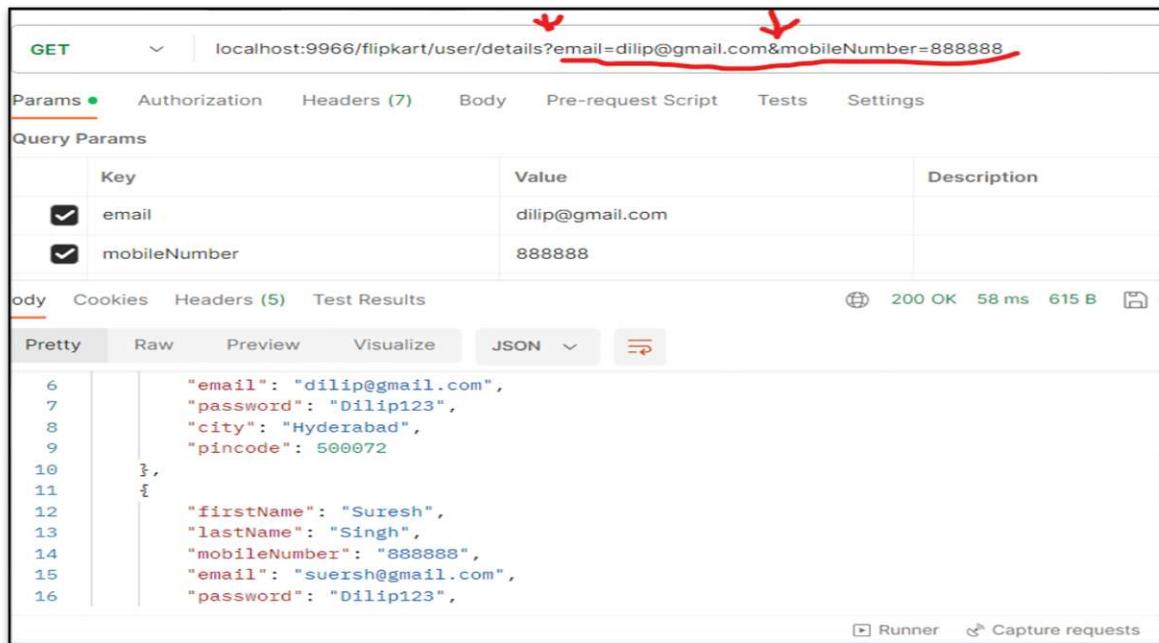
Method in controller:

```
@GetMapping("/details")
public List<Users> getUsersByEmailOrMobile(@RequestParam String email,
                                              @RequestParam String mobileNumber) {

    //Now we can pass Email Id and Mobile Number to service layer to fetch user details
    List<Users> response = service.getUsersByEmailOrMobile(email, mobileNumber);
    return response;
}
```

NOTE: Add Service, Repository layers.

URI with Query Params: **details?email=<value>&mobileNumber=<value>**



GET <localhost:9966/flipkart/user/details?email=dilip@gmail.com&mobileNumber=888888>

Params • Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

Key	Value	Description
<input checked="" type="checkbox"/> email	dilip@gmail.com	
<input checked="" type="checkbox"/> mobileNumber	888888	

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

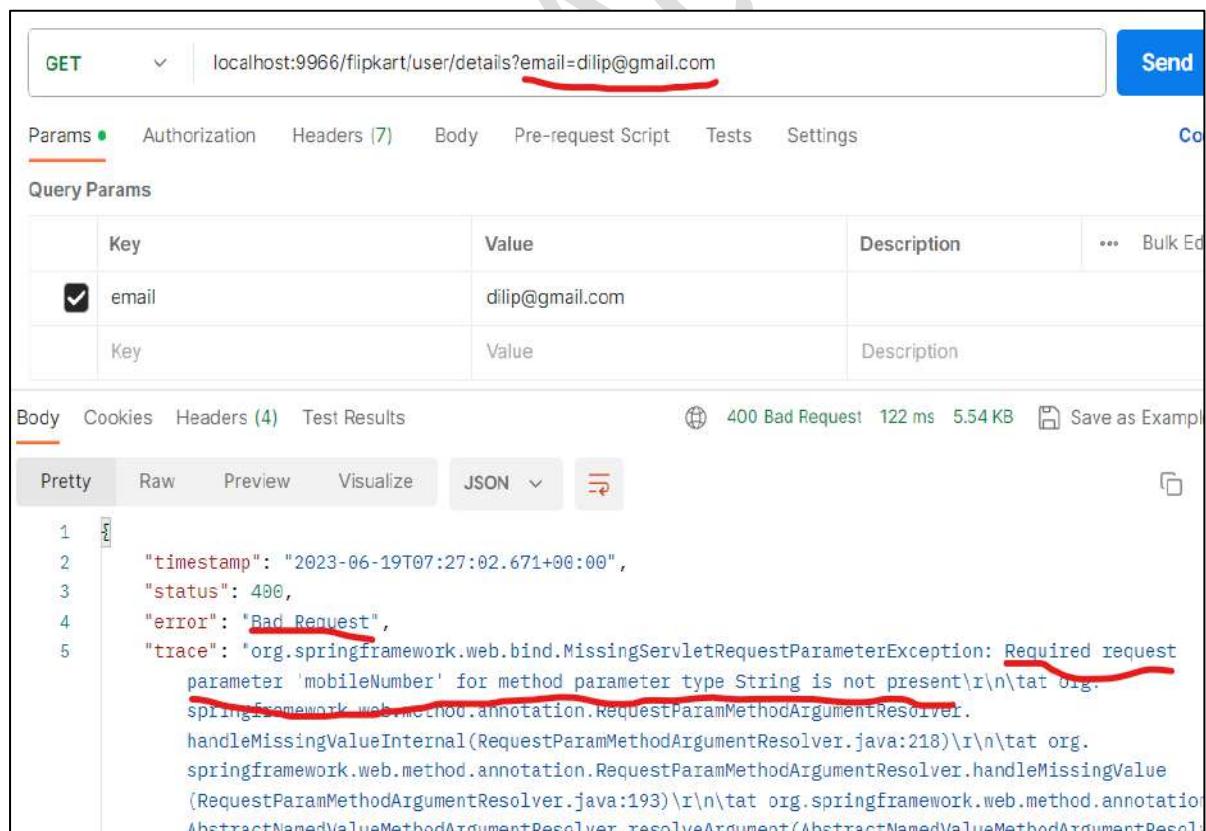
```

6   "email": "dilip@gmail.com",
7   "password": "Dilip123",
8   "city": "Hyderabad",
9   "pincode": 500072
10  },
11  {
12   "firstName": "Suresh",
13   "lastName": "Singh",
14   "mobileNumber": "888888",
15   "email": "suresh@gmail.com",
16   "password": "Dilip123",

```

Runner Capture requests

Note: By Default every Request Parameter variable is Required i.e. we should pass Query Parameter and its value as part of URL always. If we are missed any parameter, then we will get bad request.



GET <localhost:9966/flipkart/user/details?email=dilip@gmail.com>

Send

Params • Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

Key	Value	Description	...	Bulk Ed
<input checked="" type="checkbox"/> email	dilip@gmail.com			
Key	Value	Description		

Body Cookies Headers (4) Test Results

400 Bad Request 122 ms 5.54 KB Save as Example

Pretty Raw Preview Visualize JSON

```

1  {
2   "timestamp": "2023-06-19T07:27:02.671+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "trace": "org.springframework.web.bind.MissingServletRequestParameterException: Required request
parameter 'mobileNumber' for method parameter type String is not present\r\n\tat org.
springframework.web.method.annotation.RequestParamMethodArgumentResolver.
handleMissingValueInternal(RequestParamMethodArgumentResolver.java:218)\r\n\tat org.
springframework.web.method.annotation.RequestParamMethodArgumentResolver.handleMissingValue
(RequestParamMethodArgumentResolver.java:193)\r\n\tat org.springframework.web.method.annotation.
AbstractNamedValueMethodArgumentResolver.resolveArgument(AbstractNamedValueMethodArgumentResol

```

If we want to make sure any request parameter as optional, then we have to use attribute `required=false` in side `@RequestParam` annotation. Now let's make Request Parameter `mobileNumber` as an Optional in controller.

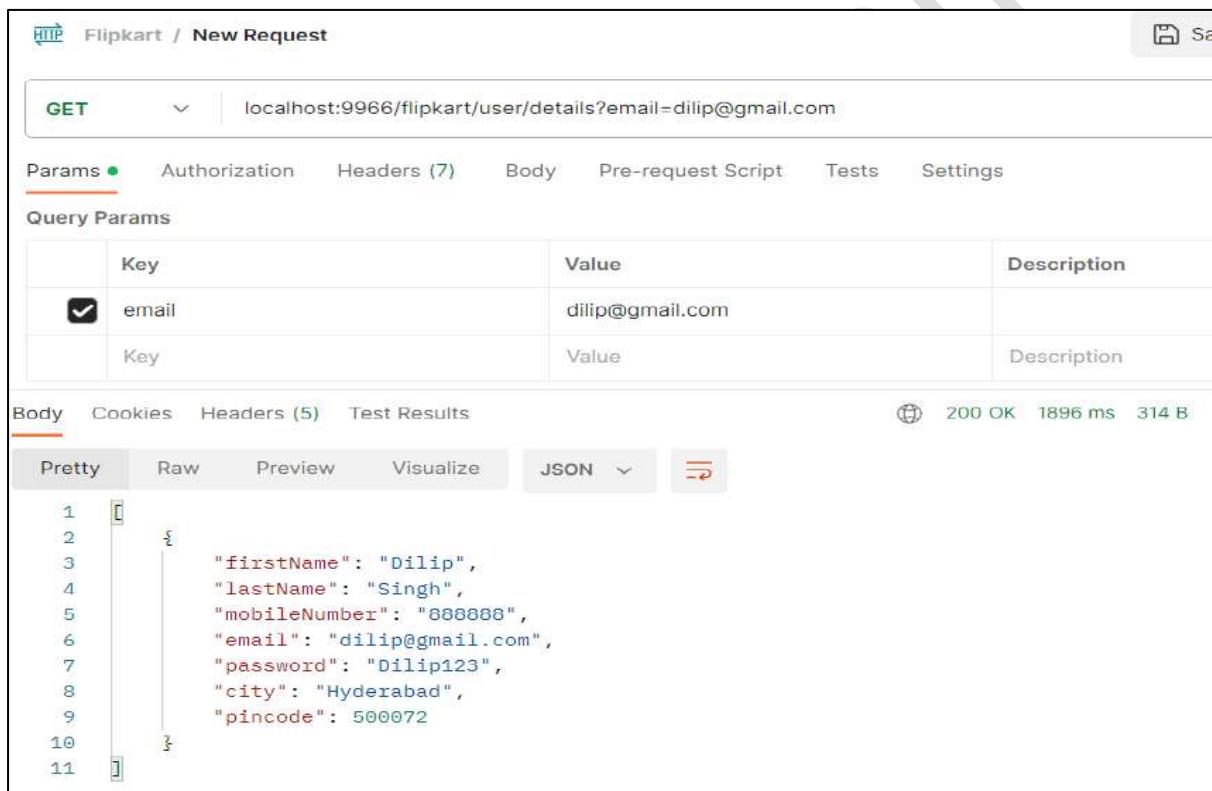
```

@GetMapping("/details")
public List<Users> getUsersByEmailOrMobile(@RequestParam String email,
                                             @RequestParam(required = false) String mobileNumber) {

    List<Users> response = service.getUsersByEmailOrMobile(email, mobileNumber);
    return response;
}

```

Testing Endpoint: Now `mobileNumber` Request Parameter is missing in URI, but still our endpoint is working only with one Request parameter `email`.



HTTP Flipkart / New Request

GET localhost:9966/flipkart/user/details?email=dilip@gmail.com

Params • Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

	Key	Value	Description
<input checked="" type="checkbox"/>	email	dilip@gmail.com	
	Key	Value	Description

Body Cookies Headers (5) Test Results 200 OK 1896 ms 314 B

Pretty Raw Preview Visualize JSON

```

1  [
2   {
3     "firstName": "Dilip",
4     "lastName": "Singh",
5     "mobileNumber": "888888",
6     "email": "dilip@gmail.com",
7     "password": "dilip123",
8     "city": "Hyderabad",
9     "pincode": "500072"
10    }
11  ]

```

Mapping a Multi-Value Parameter: A single `@RequestParam` can have multiple values:

```

@GetMapping("/api")
@ResponseBody
public String getUsers(@RequestParam List<String> id) {
    return "IDs are " + id;
}

```

And Spring MVC will map a comma-delimited id parameter:

URI: /api?id=1,2,3

Or we can pass a list of separate id parameters as part of URL

URI : /api?id=1&id=2

Mapping All Parameters:

We can also have multiple parameters without defining their names or count by just using a Map:

```
@GetMapping("/api")
public String getUsers(@RequestParam Map<String, String> allParams) {
    return "Parameters are " + allParams.entrySet();
}
```

Now we can read all Request Params from Map Object as Key and Value Pairs and we will utilize as per requirement.

When to use Query Param vs Path Variable:

As a best practice, almost of developers are recommending following way. If you want to identify a resource, you should use Path Variable. But if you want to sort or filter items on data, then you should use query parameters. So, for example you can define like this:

```
/users                                # Fetch a list of users
/users?occupation=programmer&skill=java # Fetch a list of java programmers

/users/123                             # Fetch a user who has id 123
```

Swagger UI With SpringBoot Applications:

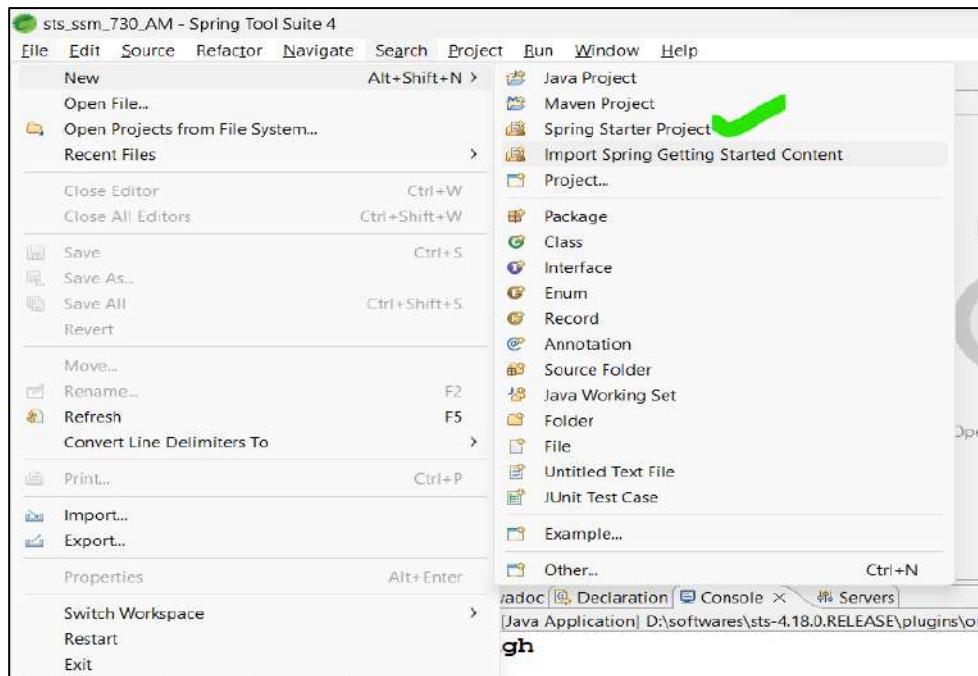
“Swagger is a set of rules, specifications and tools that help us document our APIs.”
“By using Swagger UI to expose our API’s documentation, we can save significant time.”

Swagger UI allows anyone — be it your development team or your end consumers — to visualize and interact with the API’s resources without having any of the implementation logic in place. It’s automatically generated from your OpenAPI (formerly known as Swagger) Specification, with the visual documentation making it easy for back end implementation and client side consumption.

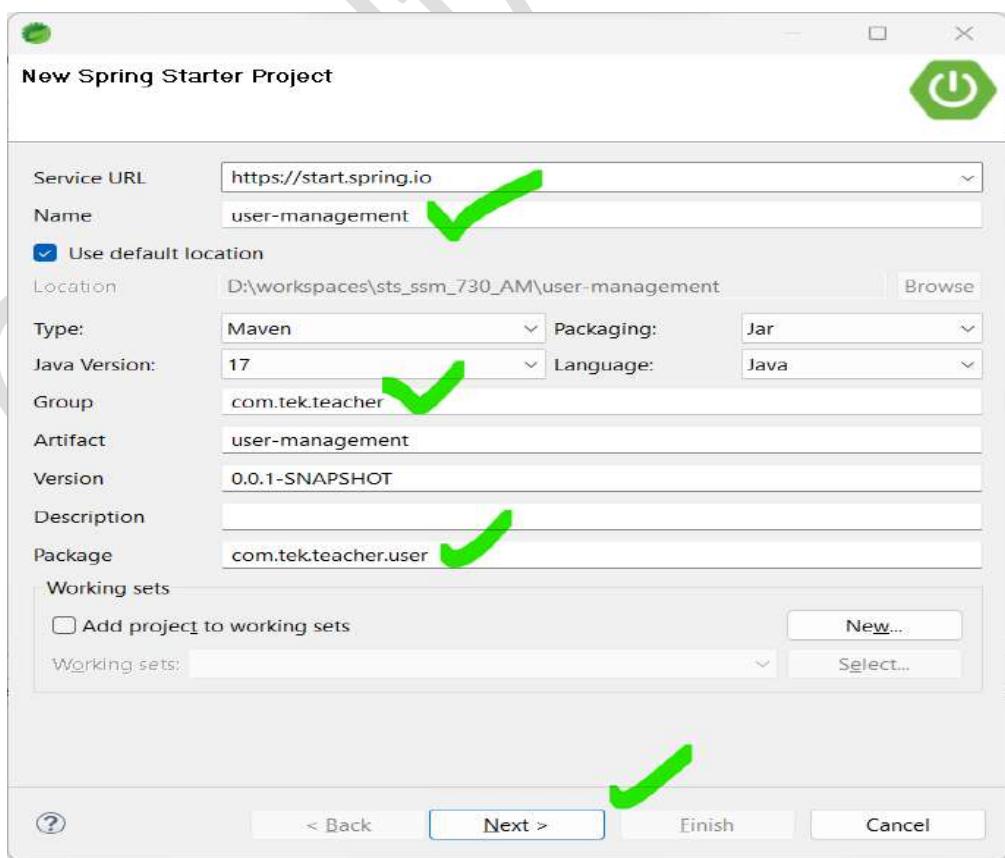
Swagger UI is one of the platform’s attractive tools. In order for documentation to be useful, we will need it to be browsable and to be perfectly organized for easy access. It is for this reason that writing good documentation may be tedious and use a lot of the developers’ time.

Create Spring Boot Web Application:

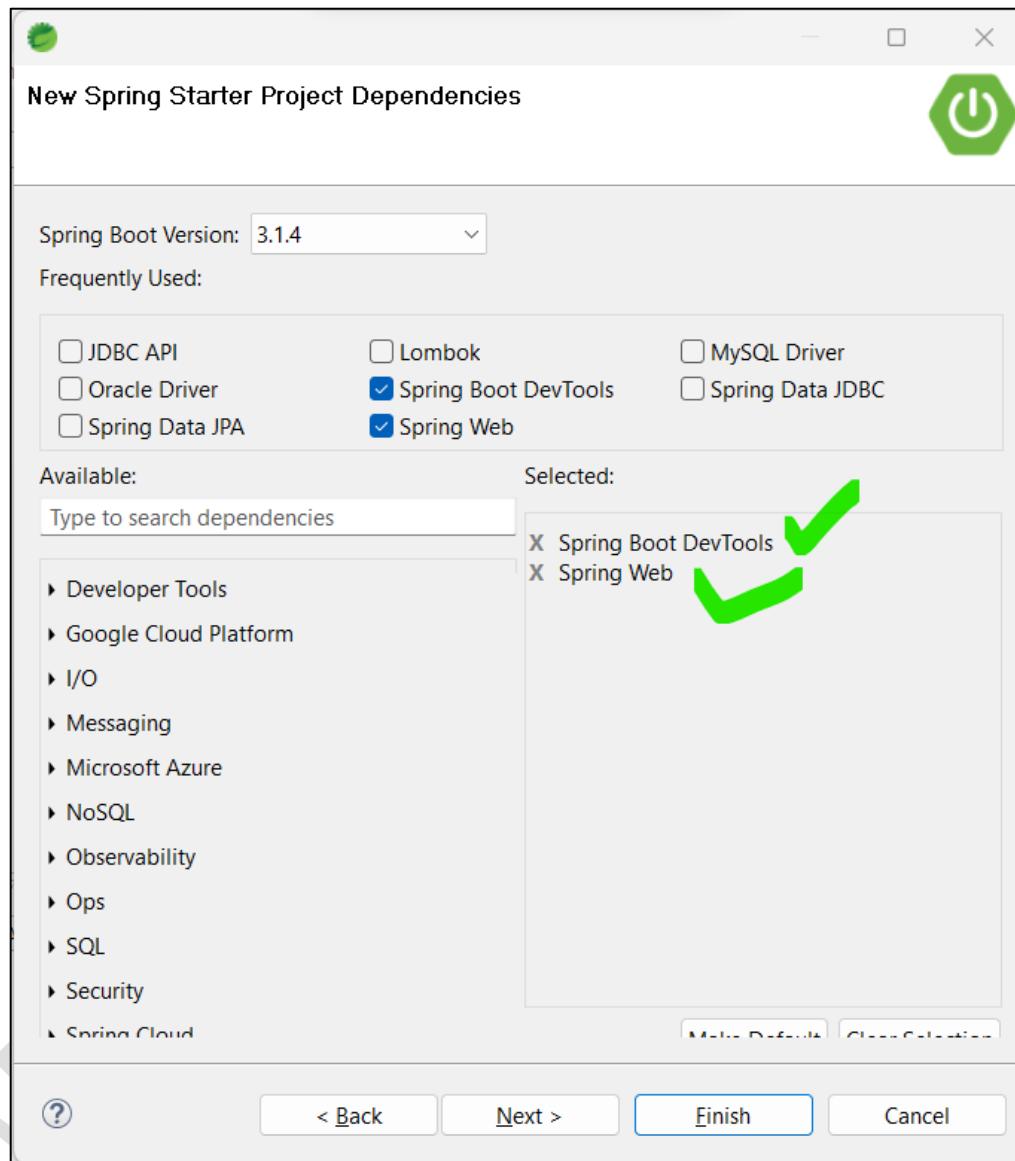
1. Open STS -> File-> New > Spring Starter Project



2. Fill All Project details as shown below and click on Next.



3. In Next Page, Add Spring Boot Modules/Starters as shown below and click on finish.
NOTE: Spring Web is mandatory, because REST Service Documentation we should do with Swagger.



4. After Project Creation, Open **pox.xml** file and add below dependency starter in side dependencies section and save.

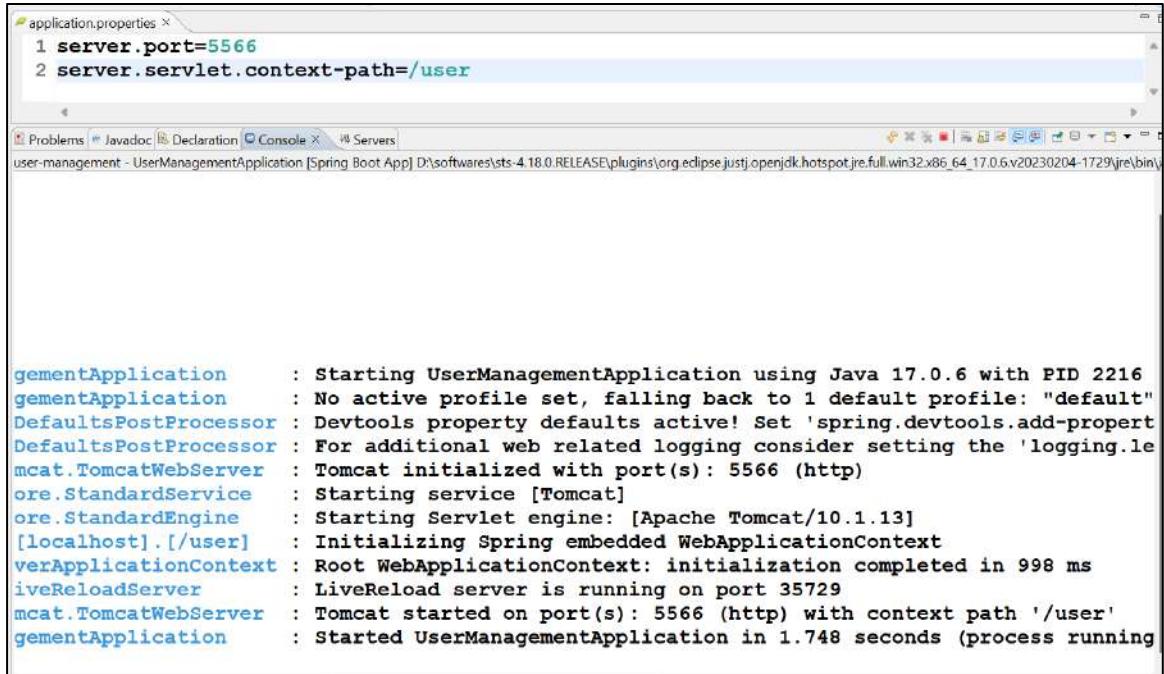
```
<!--Swagger/OpenAPI Documentation starter-->
```

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.1.0</version>
</dependency>
```

5. Now open **application.properties** file and add below two properties and save. With these properties application started on port : 5566 with context path '/user'.

```
server.port=5566
server.servlet.context-path=/user
```

6. Now start your spring Boot application



The screenshot shows the Eclipse IDE interface. The top part displays the `application.properties` file with the following content:

```
server.port=5566
server.servlet.context-path=/user
```

The bottom part shows the Eclipse logs (Console tab) with the following output:

```
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Starting UserManagementApplication using Java 17.0.6 with PID 2216
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : No active profile set, falling back to 1 default profile: "default"
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Devtools property defaults active! Set 'spring.devtools.add-properties' to false to disable
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : For additional web related logging consider setting the 'logging.level.web' property to "DEBUG"
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Tomcat initialized with port(s): 5566 (http)
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Starting service [Tomcat]
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Starting Servlet engine: [Apache Tomcat/10.1.13]
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Initializing Spring embedded WebApplicationContext
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Root WebApplicationContext: initialization completed in 998 ms
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : LiveReload server is running on port 35729
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Tomcat started on port(s): 5566 (http) with context path '/user'
2023-02-04 17:29:46.221  INFO 2216 --- [main] o.s.boot.SpringApplication : Started UserManagementApplication in 1.748 seconds (process running in 2216ms)
```

7. Enter URL in Browser for OpenAPI Swagger Documentation of Web services. Then you can see Swagger UI page with empty Services List. Because Our application not contained any web services.

NOTE: The Swagger UI page will be available at

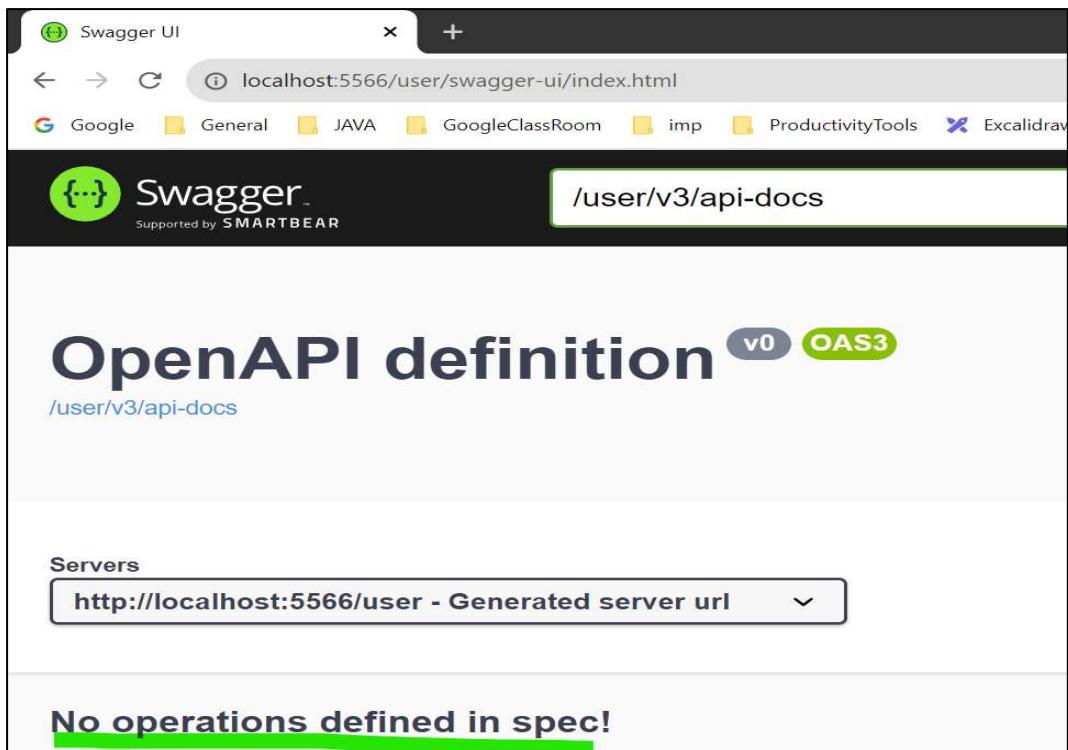
<http://server:port/context-path/swagger-ui.html> and the OpenAPI description will be available at the following URL for JSON format: <http://server:port/context-path/v3/api-docs>. Documentation can be available in YAML format as well, on the following path : [/v3/api-docs.yaml](#)

server: The server name, hostname or IP

port: The server port

context-path: The context path of the application

Swagger UI URL : <http://localhost:5566/user/swagger-ui/index.html>



Successfully Our SpringBoot Application Integrated with OpenAPI/Swagger documentation.

[Adding REST Services to our application, to see Swagger API documentation:](#)

- ✍ Now Create A controller Class in Our Application : [UserController.java](#)
- ✍ Adding REST Services inside controller class.

```
package com.tek.teacher.user.controller;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    @RequestMapping(method=RequestMethod.POST, path = "/create")
    public String createUser(@RequestBody CreateUserRequest request) {
        return "User Created Successfully ";
    }

    @RequestMapping(method=RequestMethod.GET, path = "/id/{userID}")
    public CreateUserResponse createUser(@PathVariable String userID) {

        System.out.println("Loading Values of user : " + userID);
    }
}
```

```
        CreateUserResponse response = new CreateUserResponse();
        response.setEmail("Tekk.Teacher@gmail.com");
        response.setFirstName("Tek");
        response.setLastName("Teacher");

        return response;
    }
}
```

- Create Request and Response DTO classes: [CreateUserRequest.java](#)

```
package com.tek.teacher.user.controller;

public class CreateUserRequest {

    private String firstName;
    private String lastName;
    private String email;
    private String password;
    private long mobile;
    private float income;
    private String gender;

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public long getMobile() {
```

```
        return mobile;
    }
    public void setMobile(long mobile) {
        this.mobile = mobile;
    }
    public float getIncome() {
        return income;
    }
    public void setIncome(float income) {
        this.income = income;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
}
```

 **CreateUserResponse.java**

```
package com.tek.teacher.user.controller;

public class CreateUserResponse {

    private String firstName;
    private String lastName;
    private String email;
    private long mobile;
    private float income;
    private String gender;

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getEmail() {
        return email;
    }
}
```

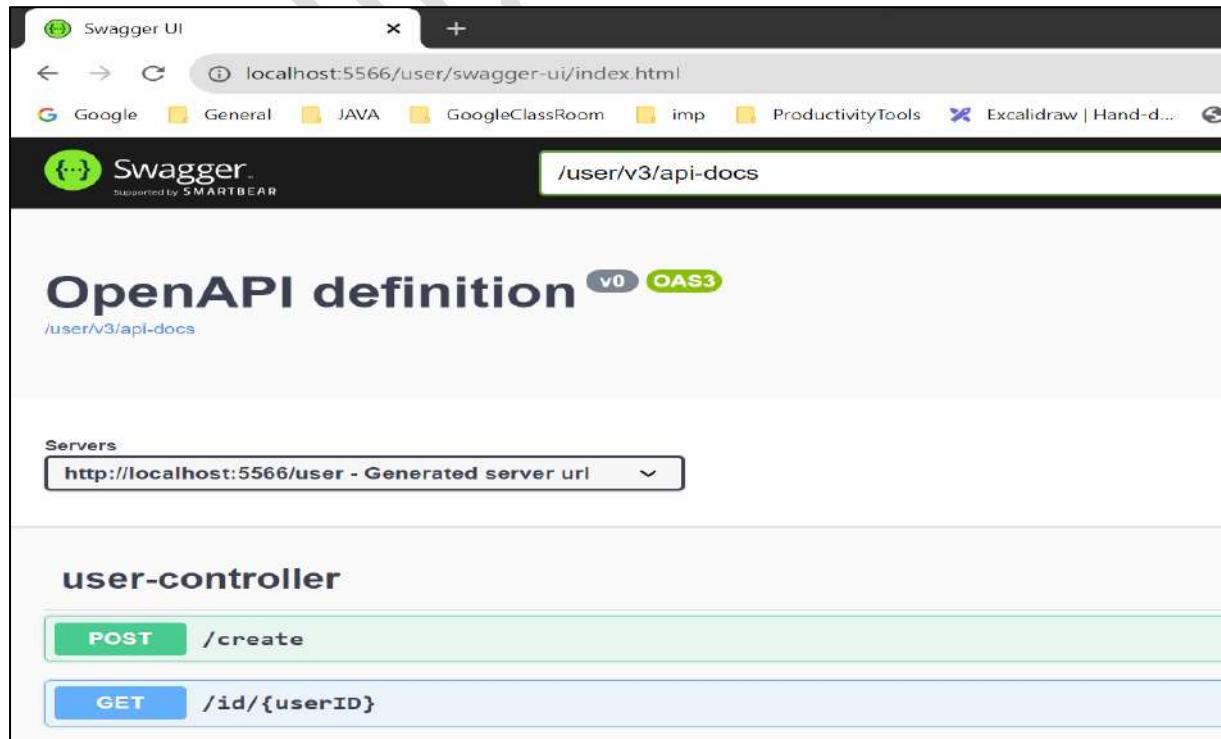
```

public void setEmail(String email) {
    this.email = email;
}
public long getMobile() {
    return mobile;
}
public void setMobile(long mobile) {
    this.mobile = mobile;
}
public float getIncome() {
    return income;
}
public void setIncome(float income) {
    this.income = income;
}
public String getGender() {
    return gender;
}
public void setGender(String gender) {
    this.gender = gender;
}
}

```

Now Start Your Spring Boot App. After application started, Now please enter swagger URL in browser. You can see all endpoints/services API request and response format Data.

Swagger UI URL : <http://localhost:5566/user/swagger-ui/index.html>



The screenshot shows the Swagger UI interface for a Spring Boot application. The title bar says "Swagger UI" and the address bar shows "localhost:5566/user/swagger-ui/index.html". The main header says "Swagger" and "Supported by SMARTBEAR". The URL is "/user/v3/api-docs".

The main content area is titled "OpenAPI definition" with "v0 OAS3" sub-titles. Below this, there is a "Servers" section with a dropdown menu containing "http://localhost:5566/user - Generated server url".

Under the "user-controller" section, there are two API endpoints listed:

- A green "POST" button next to the URL "/create".
- A blue "GET" button next to the URL "/id/{userID}".

- You can expand above both endpoints and look for payload details.

user-controller

POST /create

Parameters

No parameters

Request body required

application/json

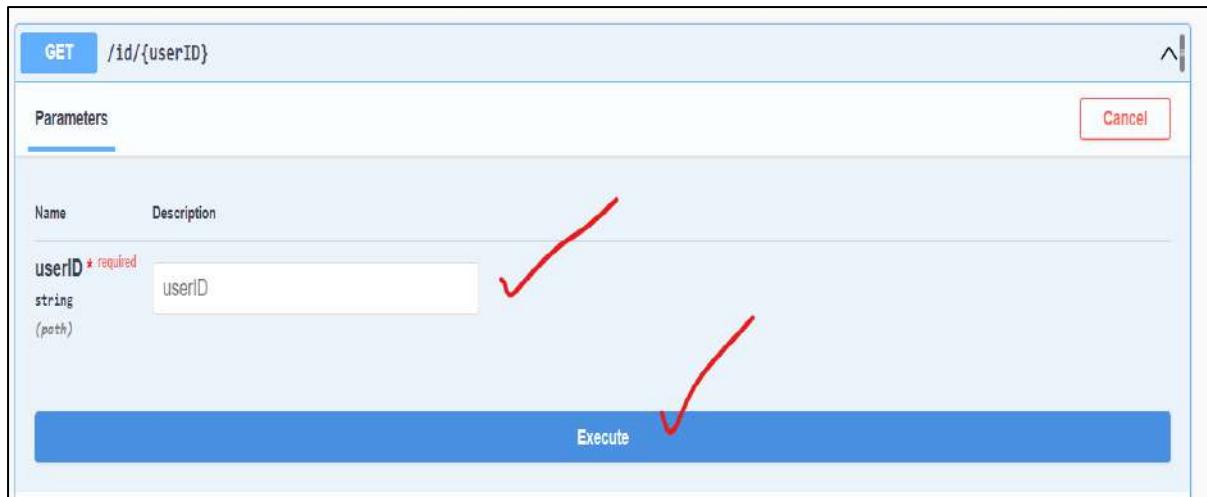
Example Value | Schema

```
{
  "firstName": "string",
  "lastName": "string",
  "email": "string",
  "password": "string",
  "mobile": 0,
  "income": 0,
  "gender": "string"
}
```

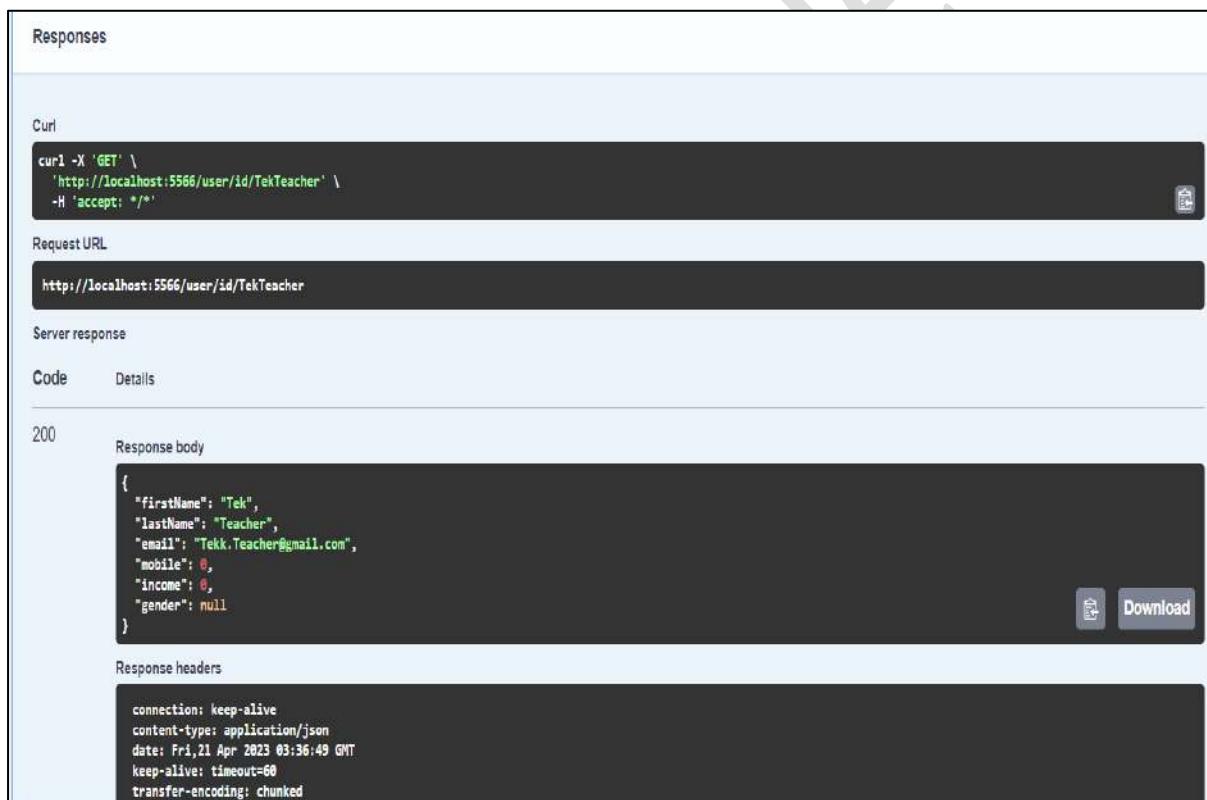
Responses

Code	Description	Links
200	OK	No links
	Media type */* <small>Controls Accept header.</small>	
	Example Value Schema string	

- We can Test API calls from Swagger UI, Please click on **Try it Out** button then it will you pass values to parameters/properties. In Below, I am passing **userId** value in **GET API service** and then click on **Execute**.



- After clicking on **Execute** You will get Response in response Section.



Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5566/user/id/TekTeacher' \
  -H 'accept: */'
```

Request URL

<http://localhost:5566/user/id/TekTeacher>

Server response

Code Details

200 Response body

```
{
  "firstName": "Tek",
  "lastName": "Teacher",
  "email": "Tekk.Teacher@gmail.com",
  "mobile": 0,
  "income": 0,
  "gender": null
}
```

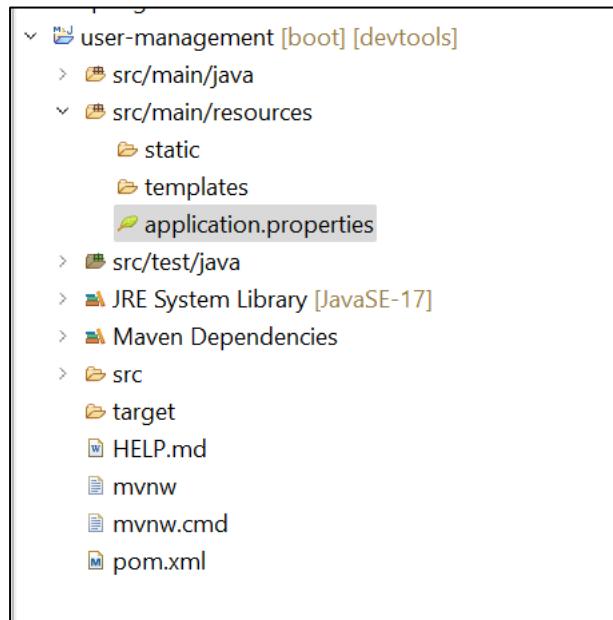
Response headers

```
connection: keep-alive
content-type: application/json
date: Fri,21 Apr 2023 03:36:49 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

This is how we can integrate and use swagger UI API Documentation Tool with our applications to share all REST API data in UI format.

Spring Boot – application.yml / application.yaml File:

In Spring Boot, whenever we create a new Spring Boot Application in spring starter, or inside an IDE (Eclipse or STS) a file is located inside the src/main/resources folder named as application.properties file.



So in a spring boot application, **application.properties** file is used to write the application-related property into that file. This file contains the different configuration values which is required to run the application. The type of property like changing the port, database connectivity and many more.

In place of **properties** file, we can use **YAML/YML** based configuration files to achieve same behaviour.

What is this YAML/YML file?

YAML stands for **Yet Another Markup Language**. YAML is a data serialization language that is often used for writing configuration files. So YAML configuration file in Spring Boot provides a very convenient syntax for storing logging configurations in a hierarchical format. The application.properties file is not that readable. So most of the time developers choose application.yml file over application.properties file. YAML is a superset of JSON, and as such is a very convenient format for specifying hierarchical configuration data. YAML is more readable and it is good for the developers to read/write configuration files.

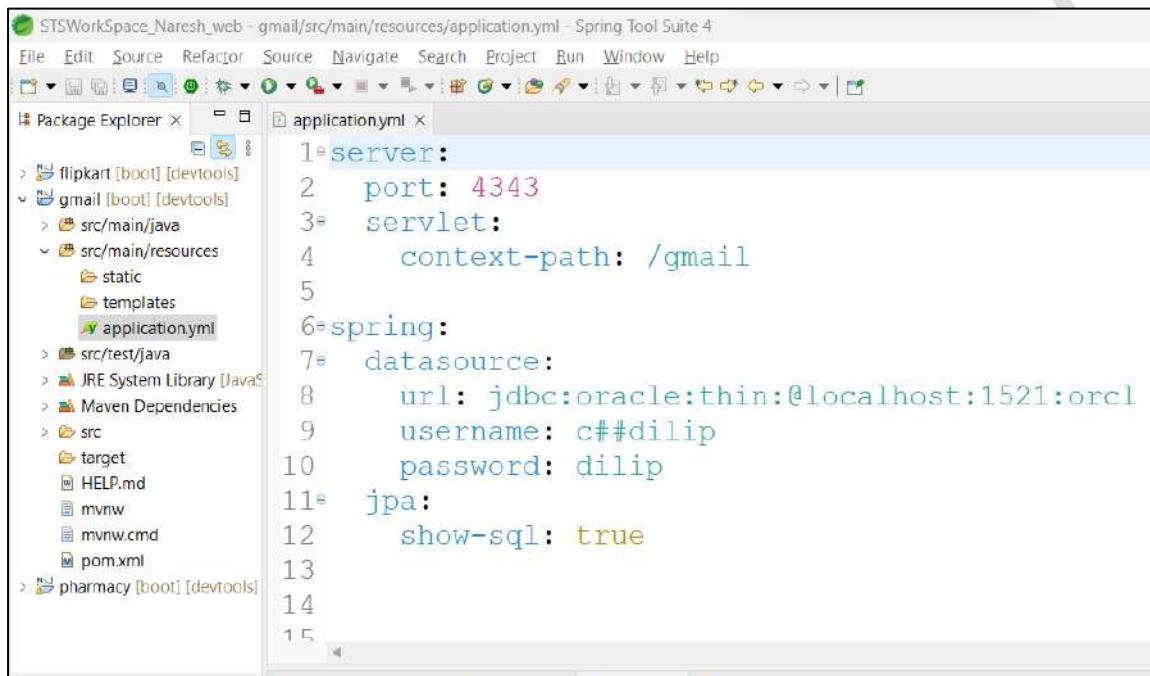
- Comments can be identified with a pound or hash symbol (#). YAML does not support multi-line comment, each line needs to be suffixed with the pound character.
- YAML files use a **.yml** or **.yaml** extension, and follow specific syntax rules.

Now let's see some examples for better understanding :

If it is **application.properties** file :

```
server.port=4343
server.servlet.context-path=/gmail
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
spring.jpa.show-sql=true
```

If we written same properties content in **application.yml**:



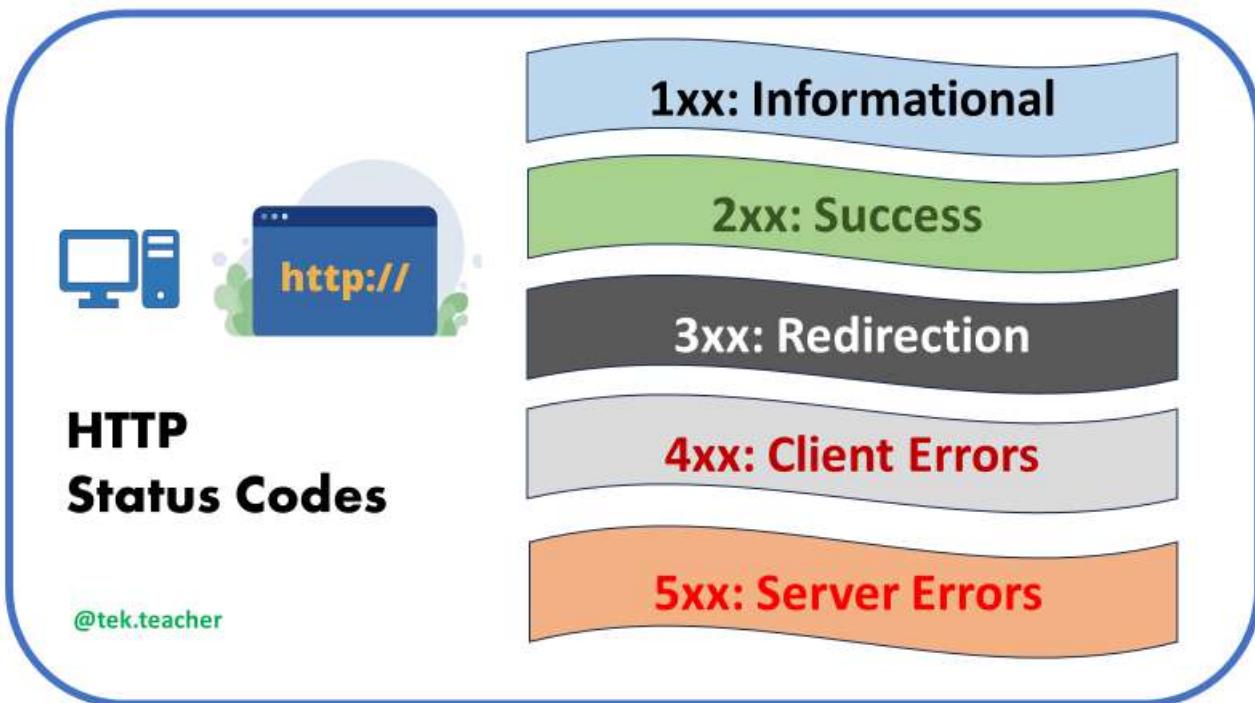
```
server:
  port: 4343
  servlet:
    context-path: /gmail
spring:
  datasource:
    url: jdbc:oracle:thin:@localhost:1521:orcl
    username: c##dilip
    password: dilip
  jpa:
    show-sql: true
```

- ✚ Similarly, we can add all other Properties in same format always as per YAML script rules and regulations.
- ✚ Now we can start our Spring Boot application as usual and continue development and testing activities.

HTTP status codes in building RESTful API's:

HTTP status codes are three-digit numbers that are returned by a web server in response to a client's request made to a web page or resource. These codes indicate the outcome of the request and provide information about the status of the communication between the client (usually a web browser) and the server. They are an essential part of the HTTP (Hypertext Transfer Protocol) protocol, which is used for transferring data over the internet. HTTP defines these standard status codes that can be used to convey the results of a client's request.

The status codes are divided into five categories.



Status Codes Series	Description
1xx: Informational	Communicates transfer protocol-level information.
2xx: Success	Indicates that the client's request was accepted successfully.
3xx: Redirection	Indicates that clients must take some additional action in order to complete their request.
4xx: Client Error	This category of error status codes points the finger at clients.
5xx: Server Error	The server takes responsibility for these error status codes.

Some of HTTP status codes summary being used mostly in REST API creation

1xx Informational:

This series of status codes indicates informational content. This means that the request is received and processing is going on. Here are the frequently used informational status codes:

100 Continue:

This code indicates that the server has received the request header and the client can now send the body content. In this case, the client first makes a request (with the `Expect: 100-continue` header) to check whether it can start with a partial request. The server can then respond either with `100 Continue (OK)` or `417 Expectation Failed (No)` along with an appropriate reason.

101 Switching Protocols:

This code indicates that the server is OK for a protocol switch request from the client.

102 Processing:

This code is an informational status code used for long-running processing to prevent the client from timing out. This tells the client to wait for the future response, which will have the actual response body.

2xx Success:

This series of status codes indicates the successful processing of requests. Some of the frequently used status codes in this class are as follows.

200 OK:

This code indicates that the request is successful and the response content is returned to the client as appropriate.

201 Created:

This code indicates that the request is successful and a new resource is created.

204 No Content:

This code indicates that the request is processed successfully, but there's no return value for this request. For instance, you may find such status codes in response to the deletion of a resource.

3xx Redirection:

This series of status codes indicates that the client needs to perform further actions to logically end the request. A frequently used status code in this class is as follows.

304 Not Modified:

This status indicates that the resource has not been modified since it was last accessed. This code is returned only when allowed by the client via setting the request headers as If-Modified-Since or If-None-Match. The client can take appropriate action on the basis of this status code.

4xx Client Errors:

This series of status codes indicates an error in processing the request. Some of the frequently used status codes in this class are as follows:

400 Bad Request:

This code indicates that the server failed to process the request because of the malformed syntax in the request. The client can try again after correcting the request.

401 Unauthorized:

This code indicates that authentication is required for the resource. The client can try again with appropriate authentication.

403 Forbidden:

This code indicates that the server is refusing to respond to the request even if the request is valid. The reason will be listed in the body content if the request is not a HEAD method.

404 Not Found:

This code indicates that the requested resource is not found at the location specified in the request.

405 Method Not Allowed:

This code indicates that the HTTP method specified in the request is not allowed on the resource identified by the URI.

408 Request Timeout:

This code indicates that the client failed to respond within the time window set on the server.

409 Conflict:

This code indicates that the request cannot be completed because it conflicts with some rules established on resources, such as validation failure.

5xx Server Errors:

This series of status codes indicates server failures while processing a valid request. Here is one of the frequently used status codes in this class:

500 Internal Server Error:

This code indicates a generic error message, and it tells that an unexpected error occurred on the server and that the request cannot be fulfilled.

501 (Not Implemented):

The server either does not recognize the request method, or it cannot fulfil the request. Usually, this implies future availability (e.g., a new feature of a web-service API).

REST API Specific HTTP Status Codes:

Generally we will have likewise below scenarios and respective status codes in REST API services. For Example,

POST - Create : **201 Created** : Successfully Request Completed.

PUT - Update : **200 Ok** : Successfully Updated Data

If not i.e. Resource Not Found Data

404 Not Found : Successfully Processed but Data Not available

GET - Read : **200 Ok** : Successfully Retrieved Data

If not i.e. Resource Not Found Data

404 Not Found : Successfully Processed but Data Not available

DELETE - Delete : **204 No Content** : Successfully Deleted Data

If not i.e. Resource Not Found Data

404 Not Found : Successfully Processed but Data Not available

Binding HTTP status codes and Response in Spring:

To bind response data and relevant HTTP status code with endpoint in side controller class, we will use predefined Spring provided class **ResponseEntity**.

ResponseEntity:

ResponseEntity represents the whole HTTP response: status code, headers, and body. As a result, we can use it to fully configure the HTTP response. If we want to use it, we have to return it from the endpoint, Spring takes care of the rest. ResponseEntity is a generic type. Consequently, we can use any type as the response body. This will be used in Controller methods as well as in RestTemplate.

This can be used as return value from an Controller URI method:

```
@RequestMapping("/handle")
public ResponseEntity<T> handle() {
    // Logic
    return new ResponseEntity<T>(responseData, responseHeaders, statusCode);
}
```

Points to be noted:

1. We should Define **ResponseEntity<T>** with Response Object Data Type at method declaration as Return type of method.
2. We should bind actual Response Data Object with Http Status Codes by passing as Constructor Parameters of ResponseEntity class, and then we returning that ResponseEntity Object to HTTP Client.

Few Examples of Controller methods with ResponseEntity:

```
@RestController
public class NetBankingController {

    @PostMapping("/create")
    @ResponseStatus(value = HttpStatus.CREATED) //Using Annotation
    public String createAccount(@Valid @RequestBody AccountDetails accountDetails) {

        return "Created Net banking Account. Please Login.";
    }

    @PostMapping("/create/loan") //Using Class
    public ResponseEntity<String> createLoan(@Valid @RequestBody AccountDetails details) {

        return new ResponseEntity<>("Created Loan Account.", HttpStatus.CREATED);
    }
}
```

Another Example:

```

@RestController
public class OrdersController {

    @RequestMapping(value = "/product/order", method = RequestMethod.PUT)
    public ResponseEntity<String> updateOrders(@RequestBody OrderUpdate request) {

        String result = orderService.updateOrders(request);
        if (result.equalsIgnoreCase("Order ID Not found")) {
            return new ResponseEntity<String>(result, HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<String>(result, HttpStatus.OK);
    }

    @GetMapping("/orders")
    public ResponseEntity<Order> getOrders(@RequestParam("orderID") String orderID) {
        Order response = service.getOrders(orderID);
        return new ResponseEntity<Order>(response, HttpStatus.OK);
    }
}

```

This is how can write any Response Status code in REST API Service implementation. Please refer RET API Guidelines for more information at what time which HTTP status code should be returned to client.

Headers in Spring MVC:

HTTP headers are part of the Hypertext Transfer Protocol (HTTP), which is the foundation of data communication on the World Wide Web. They are metadata or key-value pairs that provide additional information about an HTTP request or response. Headers are used to convey various aspects of the communication between a client (typically a web browser) and a server.

HTTP headers can be classified into two main categories: request headers and response headers.

Request Headers:

Request headers are included in an HTTP request sent by a client to a server. They provide information about the client's preferences, the type of data being sent, authentication credentials, and more. Some common request headers include:

- **User-Agent:** Contains information about the user agent (usually a web browser) making the request.
- **Accept:** Specifies the media types (content types) that the client can process.

- **Authorization:** Provides authentication information for accessing protected resources.
- **Cookie:** Sends previously stored cookies back to the server.
- **Content-Type:** Specifies the format of the data being sent in the request body.

Response Headers:

Response headers are included in an HTTP response sent by the server to the client. They convey information about the server's response, the content being sent, caching directives, and more. Some common response headers include:

- **Content-Type:** Specifies the format of the data in the response body.
- **Content-Length:** Specifies the size of the response body in bytes.
- **Set-Cookie:** Sets a cookie in the client's browser for managing state.

HTTP headers are important for various purposes, including negotiating content types, enabling authentication, handling caching, managing sessions, and more. They allow both clients and servers to exchange additional information beyond the basic request and response data. Proper understanding and usage of HTTP headers are essential for building efficient and secure web applications.

Spring MVC provides mechanisms to work with HTTP headers both in requests and responses. Here's how you can work with HTTP headers in Spring MVC.

Handling Request Headers:

Accessing Request Headers: Spring provides the **@RequestHeader** annotation that allows you to access specific request headers in your controller methods. You can use this annotation as a method parameter to extract header values.

In Spring Framework's MVC module, **@RequestHeader** is an annotation used to extract values from HTTP request headers and bind them to method parameters in your controller methods. This annotation is part of Spring's web framework and is commonly used to access and work with the values of specific request headers.

```

@GetMapping("/endpoint")
public ResponseEntity<String> handleRequest(@RequestHeader("Header-Name") String
                                             headerValue) {
    // Do something with the header and other values
}

```

Example: Define a header **user-name** inside request:

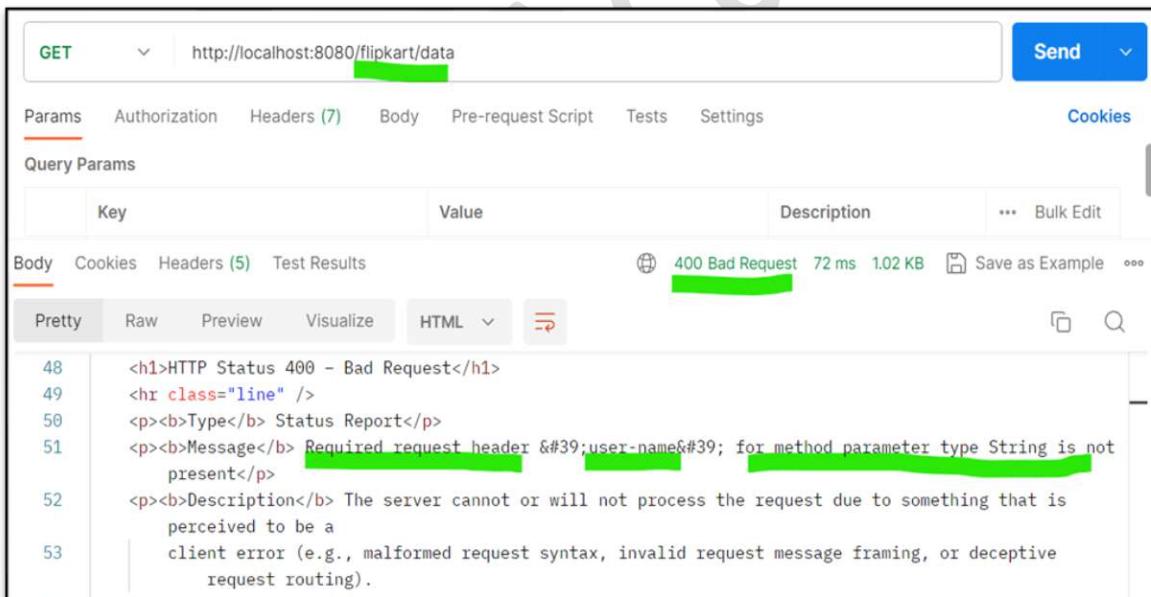
- Header and its Value should come from Client while they are triggering this endpoint.

```
package com.flipkart.controller;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;

@RestController
public class OrderController {
    //Header is Part of Request, i.e. Should be Come from Client Side.
    @GetMapping("/data")
    public String testHeaders(@RequestHeader("user-name") String userName) {
        return "Connected User : " + userName;
    }
}
```

Testing: Without Sending Header and Value from Client, Sending Request to Service.



The screenshot shows a Postman request for a GET method at `http://localhost:8080/flipkart/data`. The 'Headers' tab is selected, showing a single header `user-name` with an empty value. The response status is 400 Bad Request, with a detailed error message in the body:

```

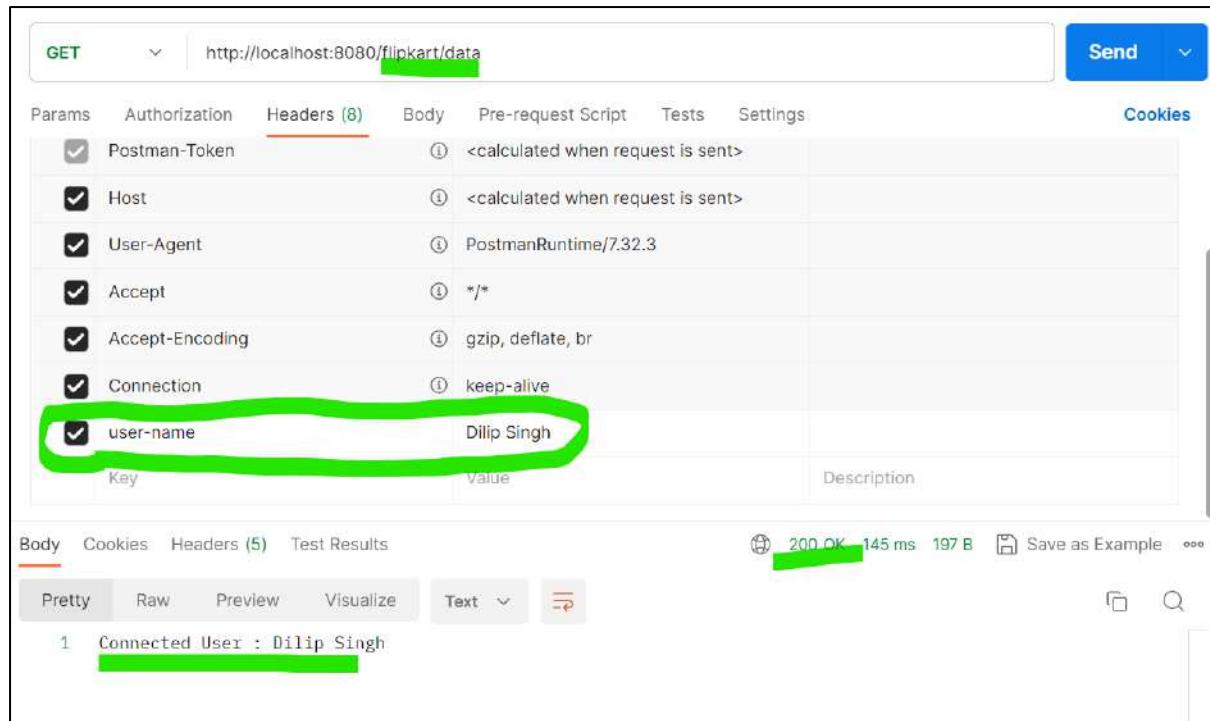
48 <h1>HTTP Status 400 – Bad Request</h1>
49 <hr class="line" />
50 <p><b>Type</b> Status Report</p>
51 <p><b>Message</b> Required request header &#39;user-name&#39; for method parameter type String is not present</p>
52 <p><b>Description</b> The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).
53

```

Result : We Got Bad request like Header is Missing i.e. Header is Mandatory by default if we defined in Controller method.

Setting Header in Client: i.e. In Our Case From Postman:

In Postman, Add header **user-name** and its value under Headers Section. Now request is executed Successfully.



Optional Headers:

If we want to make Header as an Optional i.e. non mandatory. we have to add an attribute of **required** and Its value as **false**.

```
package com.flipkart.controller;

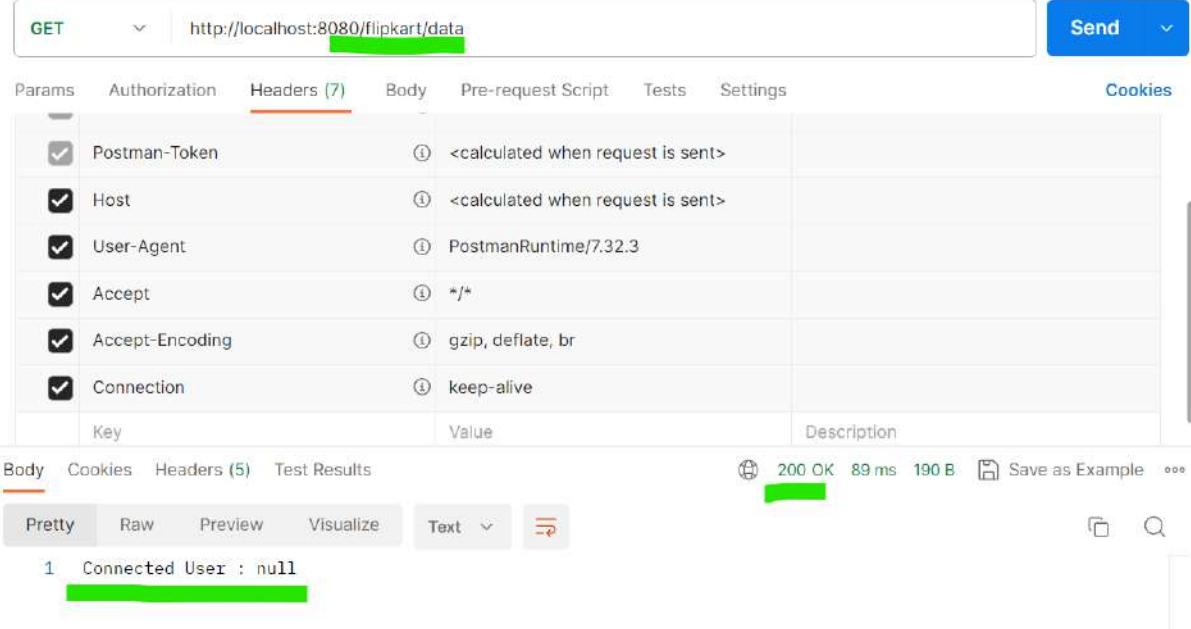
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    //Header is Part of Request, i.e. Should be Come from Client Side.
    @GetMapping("/data")
    public String testHeaders(@RequestHeader(name = "user-name",
                                             required = false) String userName) {
        return "Connected User : " + userName;
    }
}
```

Testing:

- No header Added, So Header value is null.



Default Value Of Header:

- We can Set Header Default Value also in case if we are not getting it from Client. Add an attribute **defaultValue** and its value.

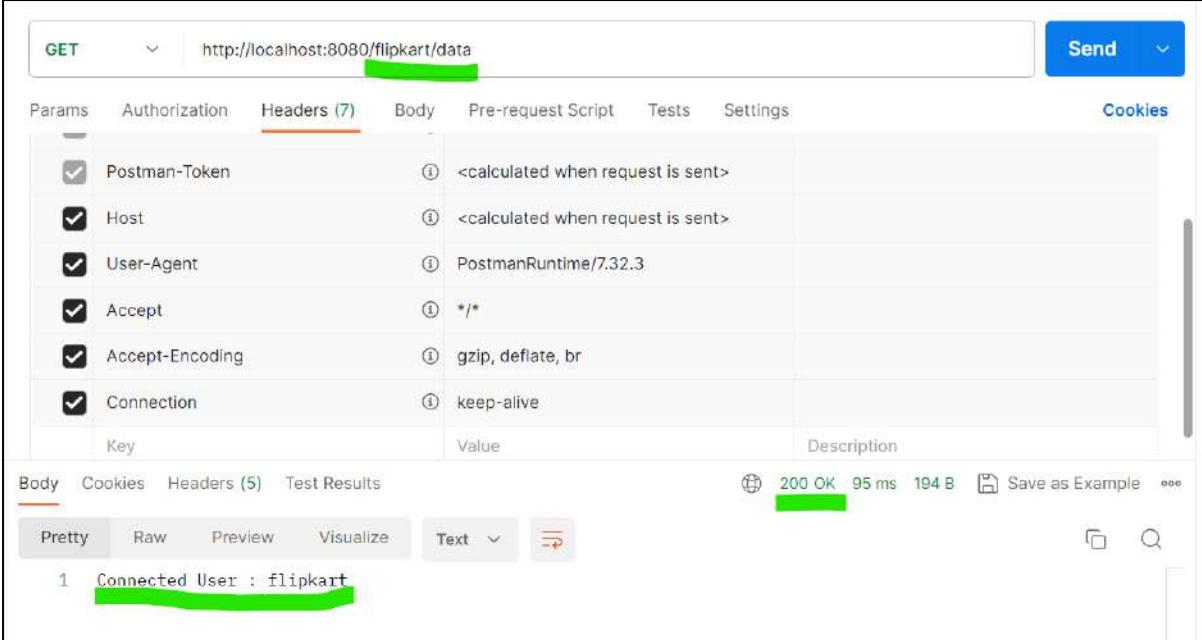
```
package com.flipkart.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    //Header is Part of Request, i.e. Should be Come from Client Side.
    @GetMapping("/data")
    public String testHeaders(@RequestHeader(name = "user-name", required = false,
                                            defaultValue = "flipkart") String userName) {
        return "Connected User : " + userName;
    }
}
```

Testing: Without adding Header and its value, triggering Service. Default Value of Header **user-name** is **flipkart** is considered by Server as per implementation.



Setting Response Headers:

In Spring MVC, response headers can be set using the **HttpServletResponse** object or the **ResponseEntity** class.

Here are some of the commonly used response headers in Spring MVC:

Content-Type: The MIME type of the response body.

Expires: The date and time after which the response should no longer be cached.

Last-Modified: The date and time when the resource was last modified.

The **HttpServletResponse** object is the standard way to set headers in a servlet-based application. To set a header using the **HttpServletResponse** object, you can use the **addHeader()** method.

For example:

```
HttpServletResponse response = request.getServletResponse();
response.addHeader("Content-Type", "application/json");
```

The **ResponseEntity** class is a more recent addition to Spring MVC. It provides a more concise way to set headers, as well as other features such as status codes and body content. To set a header using the **ResponseEntity** class, you can use the **headers()** method.

For example:

```
ResponseEntity<String> response = new ResponseEntity<>("Hello, world!", HttpStatus.OK);
response.headers().add("Content-Type", "application/json");
```

In another approach, We can create **HttpHeaders** instance and we can add multiple Headers and their values. After that, we can pass **HttpHeaders** instance to **ResponseEntity** Object.

HttpHeaders:

In Spring MVC, the **HttpHeaders** class is provided by the framework as a convenient way to manage HTTP headers in both request and response contexts. **HttpHeaders** is part of the **org.springframework.http** package, and it provides methods to add, retrieve, and manipulate HTTP headers. Here's how you can use the **HttpHeaders** class in Spring MVC:

In a Response:

You can use **HttpHeaders** to set custom headers in the HTTP response. This is often done when you want to include specific headers in the response to provide additional information to the client.

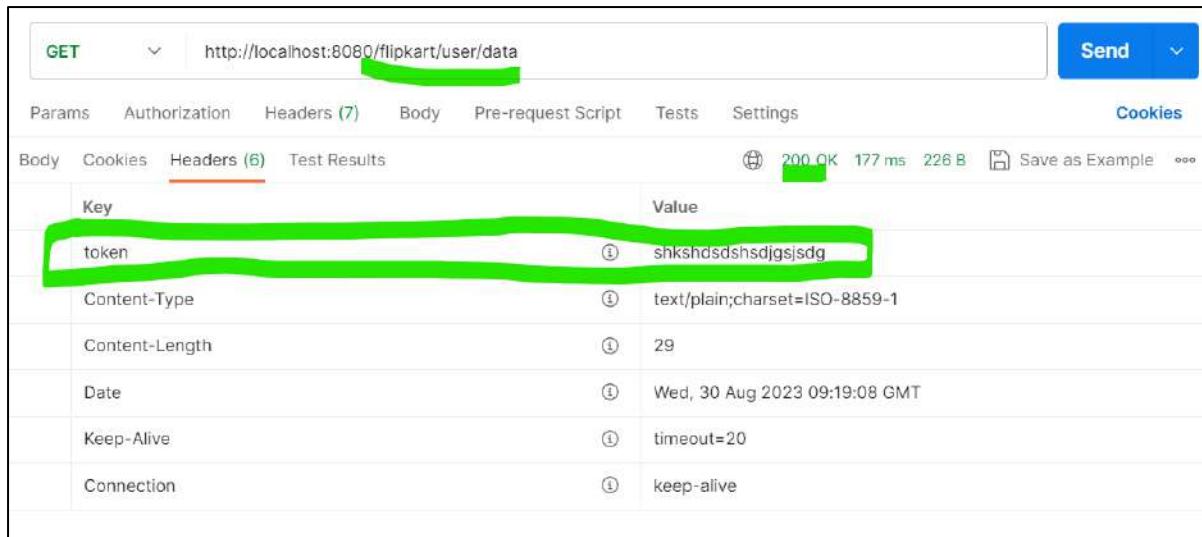
Example: Sending a Header and its value as part of response Body.

```
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    // Header is Part of Response, i.e. Should be set from Server Side.
    @GetMapping("/user/data")
    public ResponseEntity<String> testResponseHeaders() {
        HttpHeaders headers = new HttpHeaders();
        headers.set("token", "shkshdsdshsdjgsjsdg");
        return new ResponseEntity<String>("Sending Response with Headers", headers,
                                         HttpStatus.OK);
    }
}
```

Testing: Trigger endpoint from Client: Got Token and its value from Service in Headers.



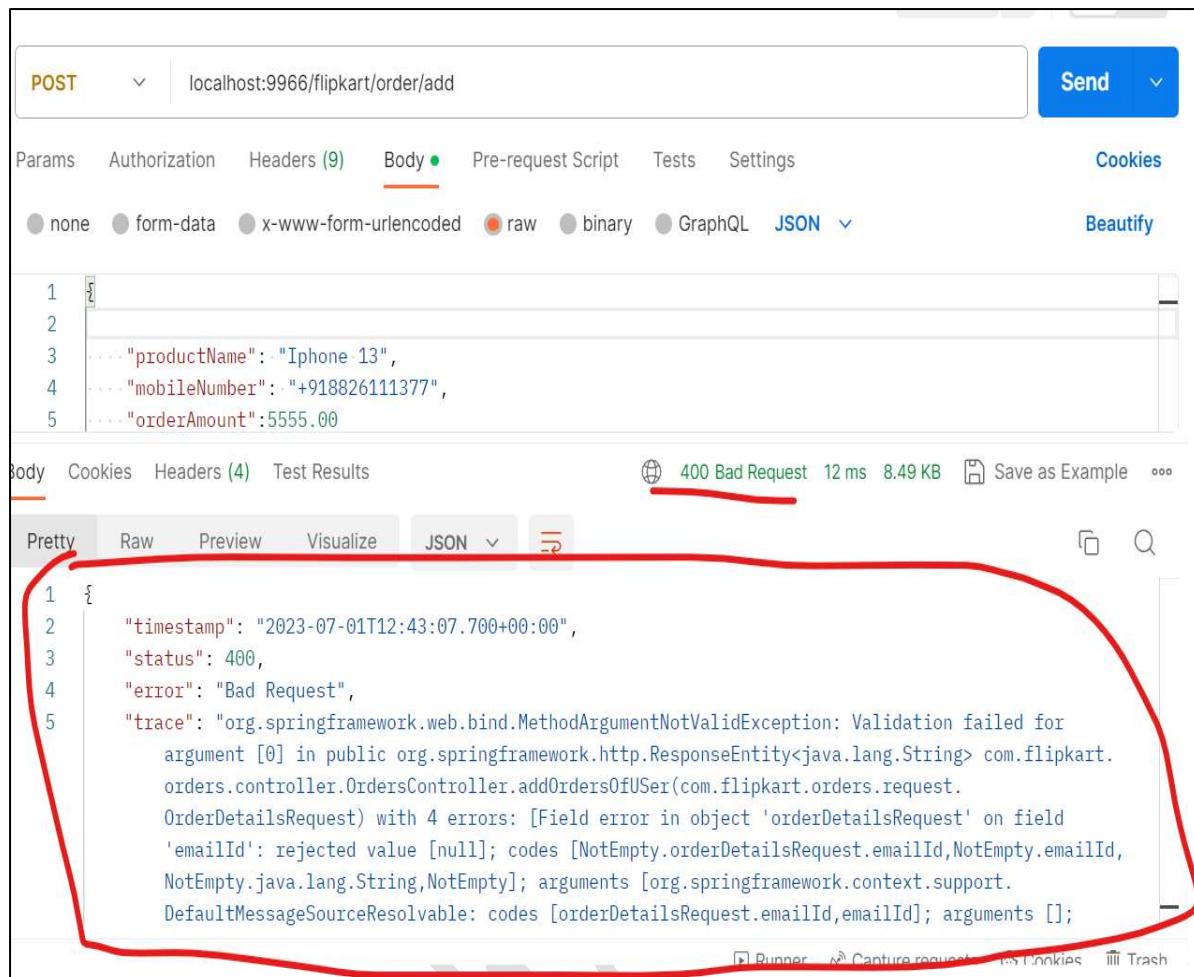
Exception Handling in Spring MVC Controllers:

What I have to do with Errors or Exceptions ?



Spring brought **@ExceptionHandler** & **@ControllerAdvice** annotations for handling Exceptions thrown at controller layer. So we can handle exceptions and will be forwarded meaning full Error response messages with response status code to HTTP clients.

If we are not handled exceptions then we will see Exception stack trace as shown in below at HTTP client level as a response. As a Best Practice we should show meaningful Error Response messages.



POST localhost:9966/flipkart/order/add

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Body (4) Test Results

400 Bad Request 12 ms 8.49 KB Save as Example

Pretty Raw Preview Visualize JSON

```

1 {
2   "timestamp": "2023-07-01T12:43:07.700+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "trace": "org.springframework.web.bind.MethodArgumentNotValidException: Validation failed for argument [0] in public org.springframework.http.ResponseEntity<java.lang.String> com.flipkart.orders.controller.OrdersController.addOrdersOfUser(com.flipkart.orders.request.OrderDetailsRequest) with 4 errors: [Field error in object 'orderDetailsRequest' on field 'emailId': rejected value [null]; codes [NotEmpty.orderDetailsRequest.emailId,NotEmpty.emailId,NotEmpty.java.lang.String,NotEmpty]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [orderDetailsRequest.emailId,emailId]; arguments []];"

```

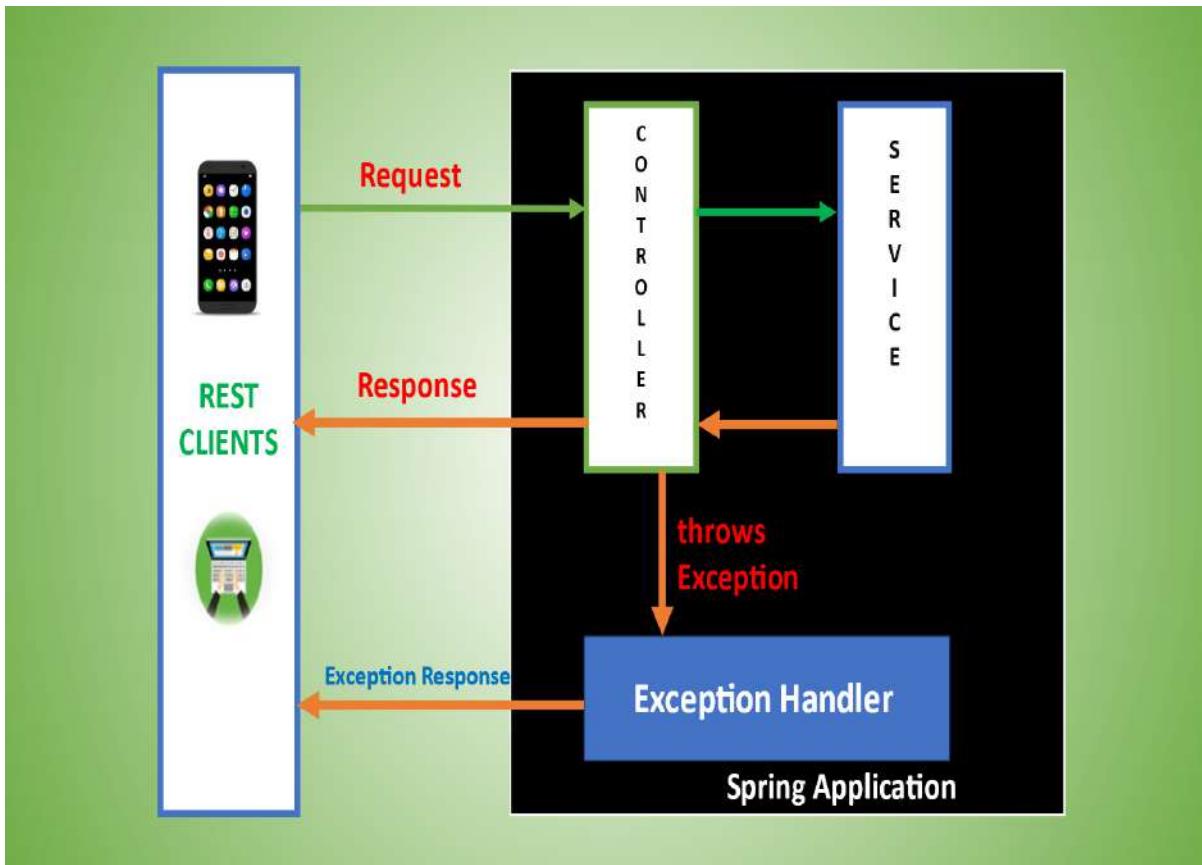
Note: Spring provided other ways as well to handle exceptions but controller advice and Exception handler will provide better way of exception handling.

@ExceptionHandler is a Spring annotation that provides a mechanism to treat exceptions thrown during execution of handlers (controller operations). This annotation, if used on methods of controller classes, will serve as the entry point for handling exceptions thrown within this controller only.

Altogether, the most common implementation is to use **@ExceptionHandler** on methods of **@ControllerAdvice** classes so that the Spring Boot exception handling will be applied globally or to a subset of controllers.

ControllerAdvice is an annotation in Spring and, as the name suggests, is “advice” for all controllers. It enables the application of a single **ExceptionHandler** to multiple controllers. With this annotation, we can define how to treat an exception in a single place, and the system will call this exception handler method for thrown exceptions on classes covered by this **ControllerAdvice**.

By using **@ExceptionHandler** and **@ControllerAdvice**, we'll be able to define a central point for treating exceptions and wrapping them in an Error object with the default Spring Boot error-handling mechanism.



Solution 1: Controller-Level @ExceptionHandler:

The first solution works at the `@Controller` level. We will define a method to handle exceptions and annotate that with `@ExceptionHandler` i.e. We can define Exception Handler Methods in side controller classes. This approach has a major drawback: The `@ExceptionHandler` annotated method is only active for that particular Controller, not globally for the entire application. But better practice is writing a separate controller advice classes dedicatedly handle different exception at one place.

```
@RestController
public class FooController{

    // Endpoint Methods

    @ExceptionHandler({ ExceptionName.class, ExceptionName.class })
    public void handleException() {
        //
    }
}
```

Solution 2: @ControllerAdvice:

The `@ControllerAdvice` annotation allows us to consolidate multiple Exception Types with `ExceptionHandler`s into a single, global error handling component level.

The actual mechanism is extremely simple but also very flexible:

- It gives us full control over the body of the response as well as the status code.
- It provides mapping of several exceptions to the same method, to be handled together.
- It makes good use of the newer RESTful **ResponseEntity** response.

One thing to keep in our mind here is to match the exceptions declared with `@ExceptionHandler` to the exception used as the argument of the method.

Example of Controller Advice class : Controller Advice With Exception Handler methods

```

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import jakarta.servlet.http.HttpServletRequest;

@ControllerAdvice
public class OrderControllerExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?>
    handleMethodArgumentException(MethodArgumentNotValidException ex,
                                   HttpServletRequest rq) {

        List<String> messages = ex.getFieldErrors().stream().map(e ->
            e.getDefaultMessage()).collect(Collectors.toList());
        return new ResponseEntity<>( messages, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(NullPointerException.class)
    public ResponseEntity<?> handleNullpointerException(NullPointerException ex,
                                                       HttpServletRequest request) {

        return new ResponseEntity<>("Please Check data, getting as null values",
                                     HttpStatus.INTERNAL_SERVER_ERROR);
    }

    @ExceptionHandler(ArithmetricException.class)
    public ResponseEntity<?> handleArithmetricException(ArithmetricException ex,
                                                       HttpServletRequest request) {
}

```

```

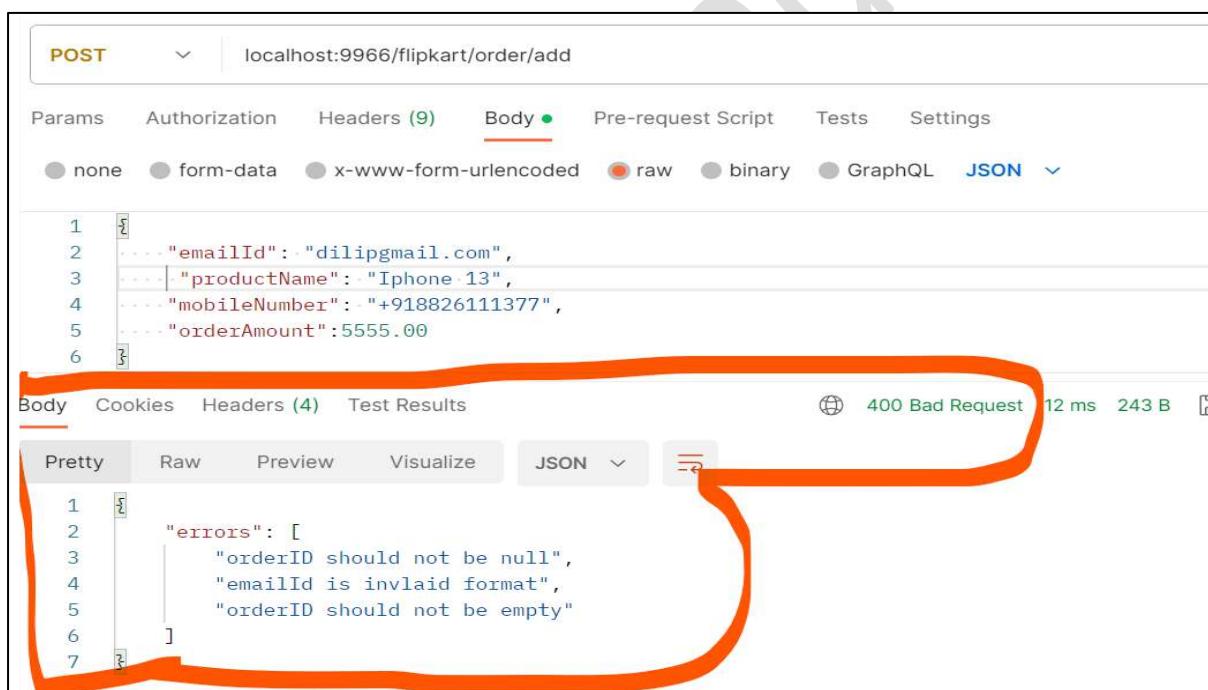
        return new ResponseEntity<>("Please Exception Details",
                                      HttpStatus.INTERNAL_SERVER_ERROR);
    }

// Below Exception handler method will work for all child exceptions when we are not
// handled those specifically.
    @ExceptionHandler(Exception.class)
    public ResponseEntity<?> handleException(Exception ex, HttpServletRequest
                                                request) {

        return new ResponseEntity<>("Please check Exception Details. ",
                                      HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

- Now see How we are getting Error response with meaningful messages when Request Body validation failed instead of complete Exception stack trace.



The screenshot shows a POST request to `localhost:9966/flipkart/order/add`. The request body is a JSON object with fields: `emailId`, `productName`, `mobileNumber`, and `orderAmount`. The response is a 400 Bad Request with the following JSON error message:

```

{
  "errors": [
    "orderID should not be null",
    "emailId is invalid format",
    "orderID should not be empty"
  ]
}

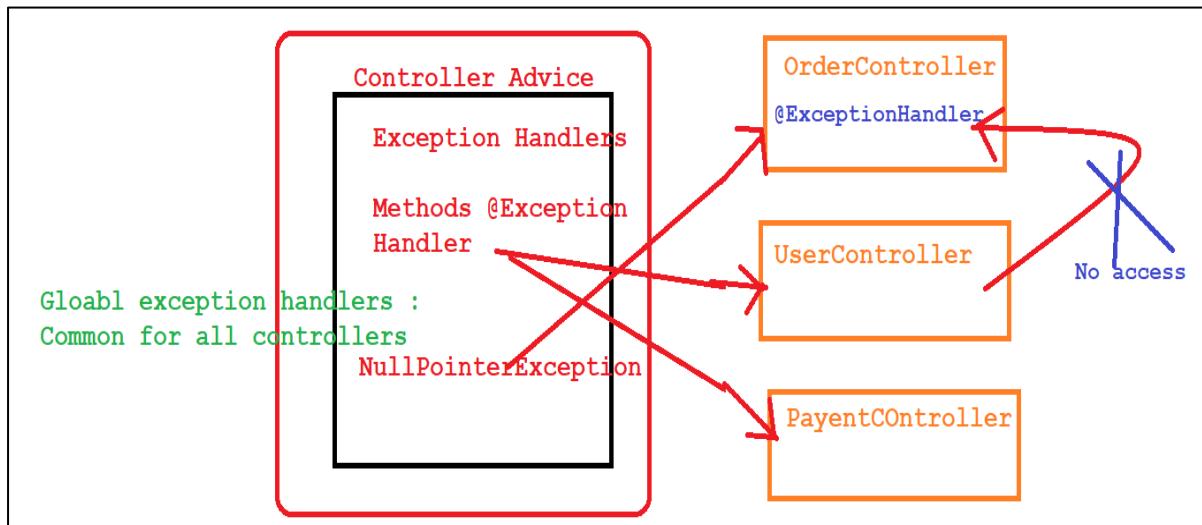
```

How it is working?

Whenever an exception occurred at controller layer due to any reason, immediately controller will check for relevant exceptions handled as part of Exception Handler or not. If handled, then that specific exception handler method will be executed and response will be forwarded to clients. If not handled, then entire exception error stack trace will be forwarded to client as it's not suggestable.

Which Exception takes Priority if we defined Child and Parent Exceptions Handlers?

From above example, if **NullPointerException** occurred then **handleNullpointerException()** method will be executed even though we have logic for parent **Exception** handling i.e. Priority given to child exception if we handled and that will be returned as response data. Similarly we can define multiple controller advice classes with different types of Exceptions along with relevant Http Response Status Codes.



@RequestMapping consumes and produces attributes:

Spring, by default, configures Jackson for parsing Java objects to JSON and converting JSON to Java objects as part of REST API request-response handling. When we want to support other Request and Response Data Formats in REST Services implementation, then we should define those respective data formats with help of **consumes** and **produces** attributes inside **@RequestMapping** annotation with endpoint methods. Same attributes and respective functionalities are applicable to shortcut annotations like **@GetMapping**, **@PostMapping** etc..

consumes:

Using a **consumes** attribute to narrow the mapping by the content type. You can declare a shared **consumes** attribute at the class level i.e. applicable to all controller methods. Unlike most other request-mapping attributes, however, when used at the class level, a method-level **consumes** attribute overrides rather than extends the class-level declaration.

The **consumes** attribute also supports negation expressions – for example, **!text/plain** means any content type other than **text/plain**.

MediaType class provides constants for commonly used media/content types, such as **APPLICATION_JSON_VALUE** and **APPLICATION_XML_VALUE** etc..

Now let's have an example, as below shown. Created an endpoint method, which accepts only JSON data Request by providing **consumes = "application/json"**.

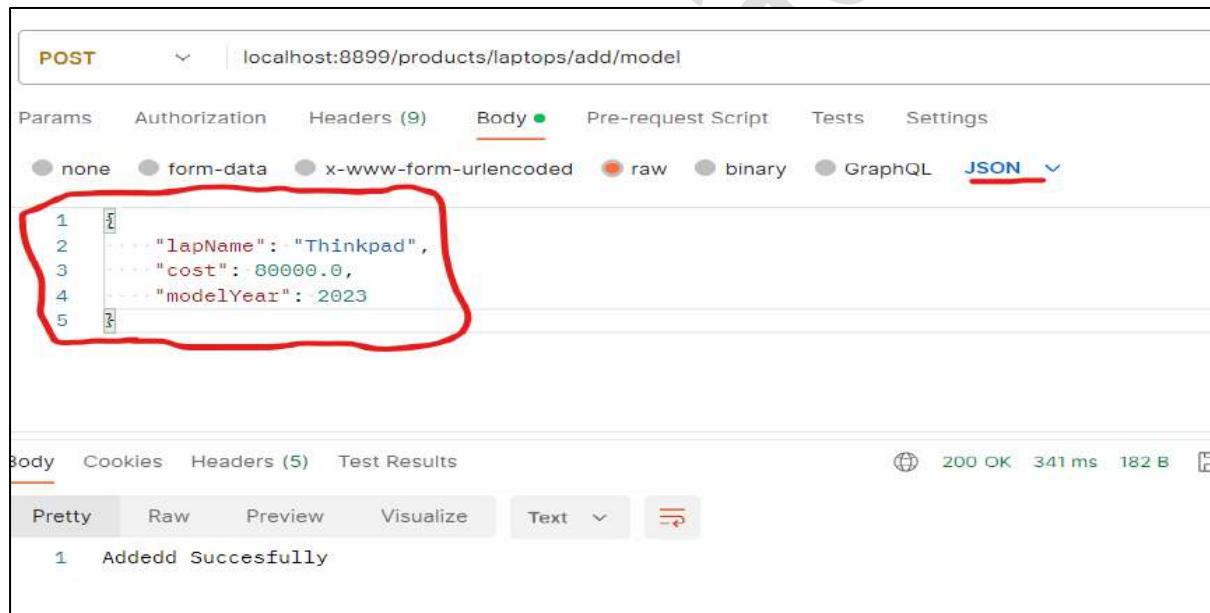
```
@RequestMapping(path = "/add/model", consumes = "application/json",
    method = RequestMethod.POST)
public String addLaptopDetails(@RequestBody LaptopDetails details) {
    return "Adddedd Succesfully";
}
```

LaptopDetails.java : To Bind Request Body of JSON

```
public class LaptopDetails {
    private String lapName;
    private double cost;
    private int modelYear;

    //Setters and Getters
}
```

Now Trigger Endpoint with JSON data in Request Body.



Now try to trigger same endpoint with XML Request Body.

We will get an exception/error response as shown below.

```
"error": "Unsupported Media Type",
"trace": "org.springframework.web.HttpMediaTypeNotSupportedException: Content-Type 'application/xml'"
```

Creating Endpoint which accepts only XML data Request Body:

To support XML request Body, we should follow below configurations/steps. Spring boot, by default, configures Jackson for parsing Java objects to JSON and converting JSON to Java objects as part of REST API request-response handling. To accept XML requests and send XML responses, there are two common approaches.

- Using Jackson XML Module
- Using JAXB Module

Start with adding Jackson's XML module by including the `jackson-dataformat-xml` dependency. Spring boot manages the library versions, so the following declaration is enough. Add below both dependencies in `POM.xml` file of application.

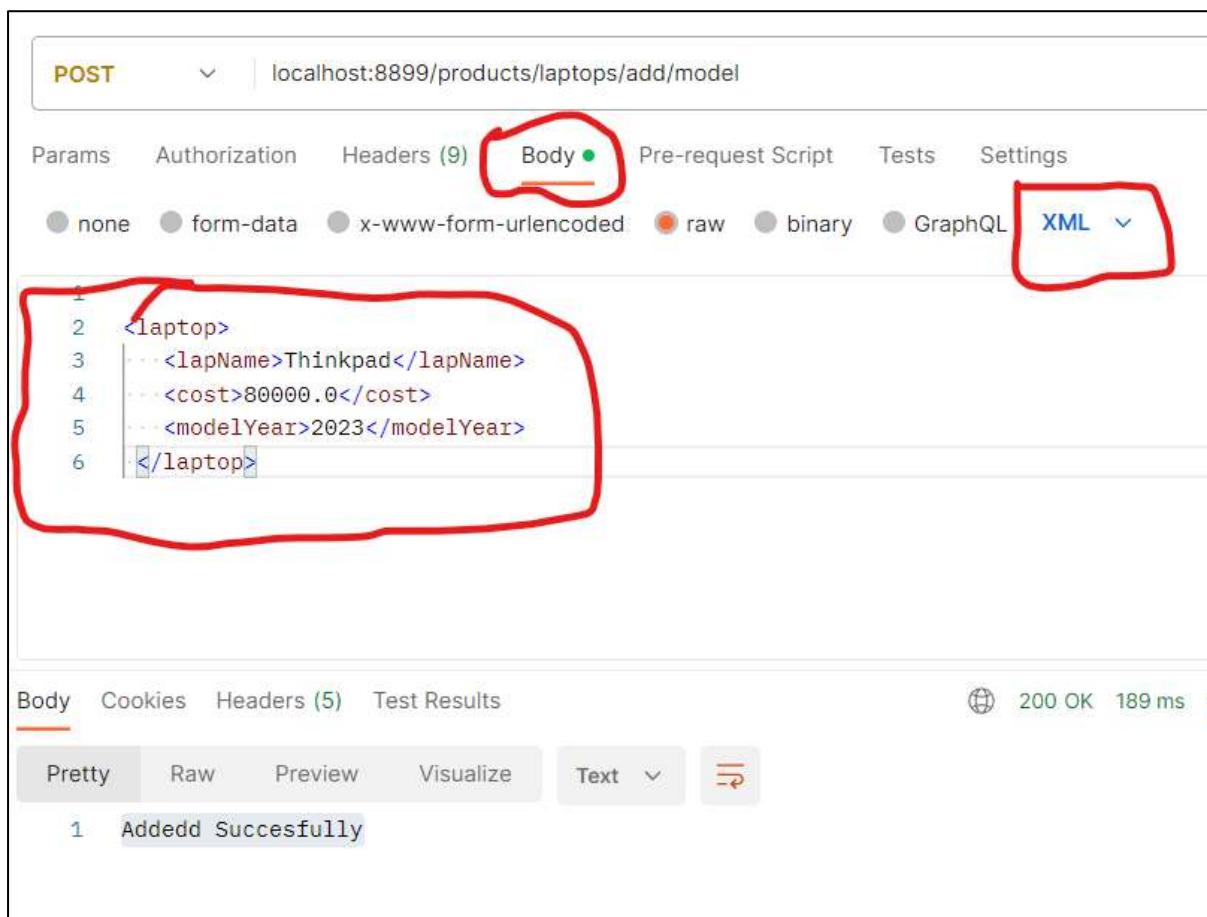
```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Now we can define and access REST API Services with XML data format.

- Creating a service which accepts only XML Request Body i.e. endpoint accepts now only XML request body but not JSON.

```
@RequestMapping(path = "/add/model", method = RequestMethod.POST,
                consumes = "application/xml")
public String addLaptopDetails(@RequestBody LaptopDetails details) {
    return "Added Successfully";
}
```

- Now Trigger Endpoint with XML Request data in Body.



POST | localhost:8899/products/laptops/add/model

Params Authorization Headers (9) **Body** (green dot) Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **XML**

```

1 <laptop>
2   <lapName>Thinkpad</lapName>
3   <cost>80000.0</cost>
4   <modelYear>2023</modelYear>
5 </laptop>

```

Body Cookies Headers (5) Test Results 200 OK 189 ms

Pretty Raw Preview Visualize Text

1 Added Succesfully

- Create endpoint which supports both JSON and XML Request Body.

Below URI Request Mapping will support both XML and JSON Requests. We can pass multiple data types **consumes** attribute with array of values.

```

@RequestMapping(path = "/add/model", method = RequestMethod.POST,
                consumes = {"application/json", "application/xml"})
public String addLaptopDetails(@RequestBody LaptopDetails details) {
    return "Added Succesfully";
}

```

Spring Provided a class **MediaType** with Constant values of different Medi Types. We will use **MediaType** in **consumes** and **produces** attributes values.

consumes ={"application/json", "application/xml"}
 is equals to
 consumes ={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE}

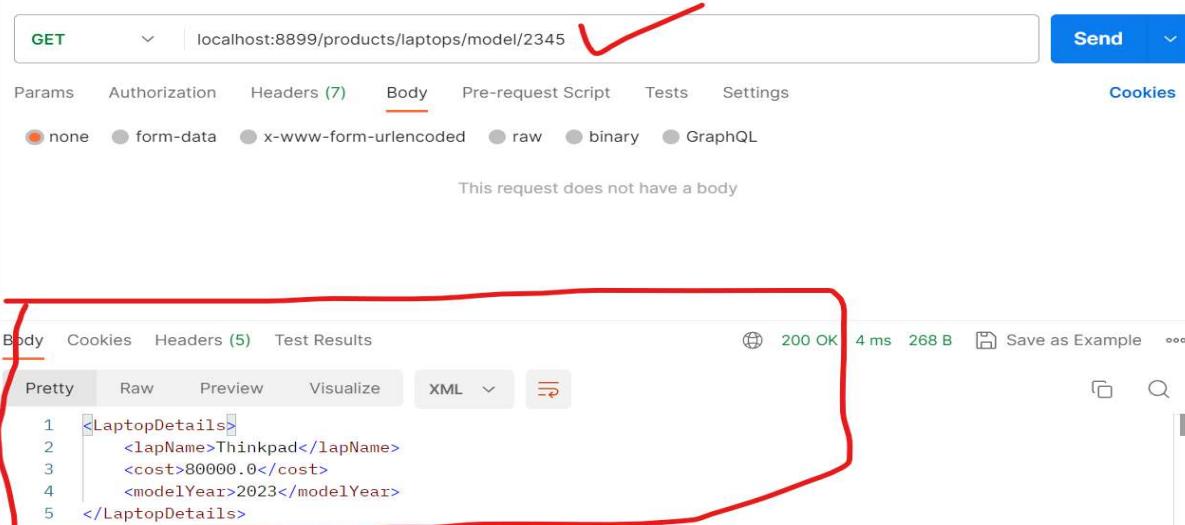
produces: with **produces** attributes, we can configure which type of Response data should be generated from Response object.

Endpoint Producing Only XML response:

Configure Request mapping with **produces = MediaType.APPLICATION_XML_VALUE**. So that now it will generate only XML response.

```
@RequestMapping(path = "/model/2345", method = RequestMethod.GET,
produces = MediaType.APPLICATION_XML_VALUE)
public LaptopDetails getLaptopDetails() {
    LaptopDetails lap = new LaptopDetails();
    lap.setCost(80000.00);
    lap.setLapName("Thinkpad");
    lap.setModelYear(2023);
    return lap;
}
```

Above endpoint generates only XML response for every incoming request.



The screenshot shows a Postman request for a GET method to the URL `localhost:8899/products/laptops/model/2345`. The 'Body' tab is selected, showing the response content as XML. The XML output is:

```

1 <LaptopDetails>
2   <lapName>Thinkpad</lapName>
3   <cost>80000.0</cost>
4   <modelYear>2023</modelYear>
5 </LaptopDetails>

```

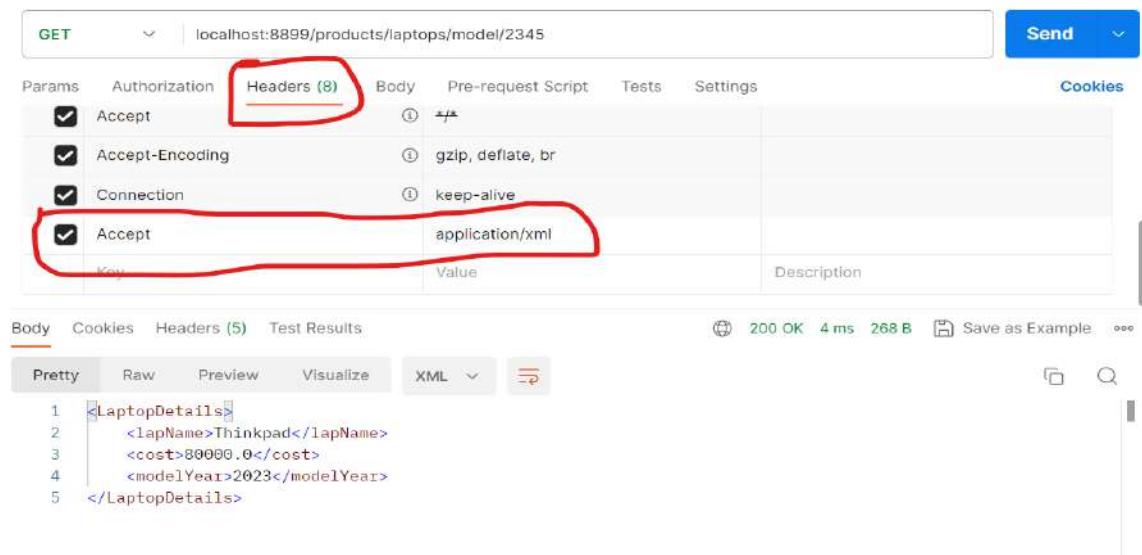
Creating an Endpoint Producing both JSON and XML response.

Configure Request mapping with **produces** attribute supporting both Media Types values i.e. array of values. So that now this endpoint generates either XML or JSON response depends on header **Accept** and its value. The HTTP **Accept** header is a request type header. The Accept header is used to inform the server by the client that which content type is understandable by the client.

```
@RequestMapping(path = "/model/2345", method = RequestMethod.GET, produces = {  
    MediaType.APPLICATION_XML_VALUE, MediaType.APPLICATION_JSON_VALUE})
```

```
public LaptopDetails getLaptopDetails() {  
    LaptopDetails lap = new LaptopDetails();  
    lap.setCost(80000.00);  
    lap.setLapName("Thinkpad");  
    lap.setModelYear(2023);  
    return lap;  
}
```

Request for XML response: Add Header **Accept** and value as **application/xml** as shown.



GET localhost:8899/products/laptops/model/2345

Headers (8)

Accept	application/xml
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
Accept	application/xml

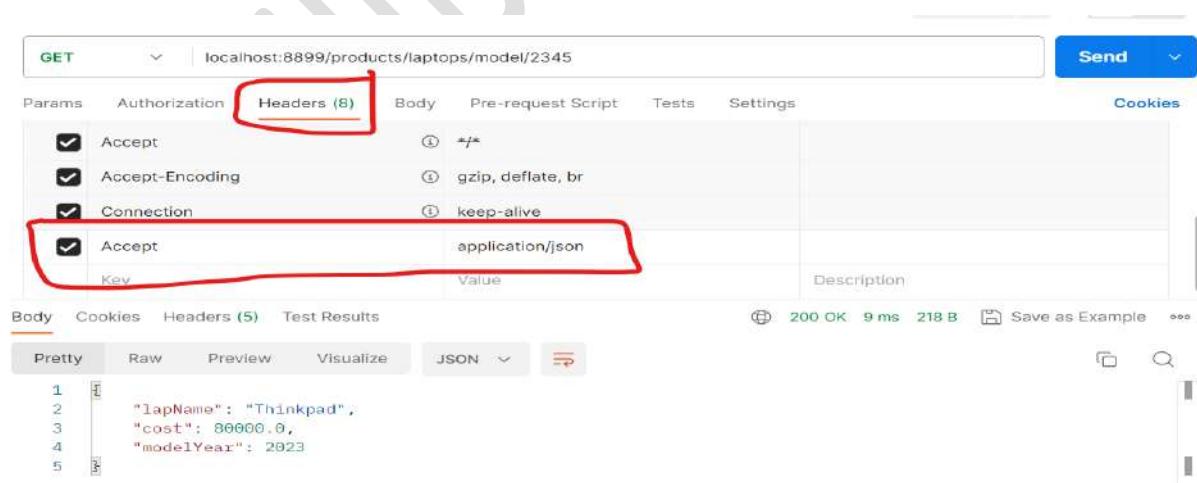
Body

```

1 <LaptopDetails>
2   <lapName>Thinkpad</lapName>
3   <cost>80000.0</cost>
4   <modelYear>2023</modelYear>
5 </LaptopDetails>

```

Request for JSON response: Add Header **Accept** and value as **application/json** as shown.



GET localhost:8899/products/laptops/model/2345

Headers (8)

Accept	application/json
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
Accept	application/json

Body

```

1 {
2   "lapName": "Thinkpad",
3   "cost": 80000.0,
4   "modelYear": 2023
5 }

```

Producing and Consuming REST API services:

Producing REST Services:

Producing REST services is nothing but creating Controller endpoint methods i.e. Defining REST Services on our own logic. As of Now we are created/produced multiple REST API Services with different examples by writing controller layer and URI mapping methods.

Consuming REST Services:

Consuming REST services is nothing but integrating/calling other application REST API services from our application logic.

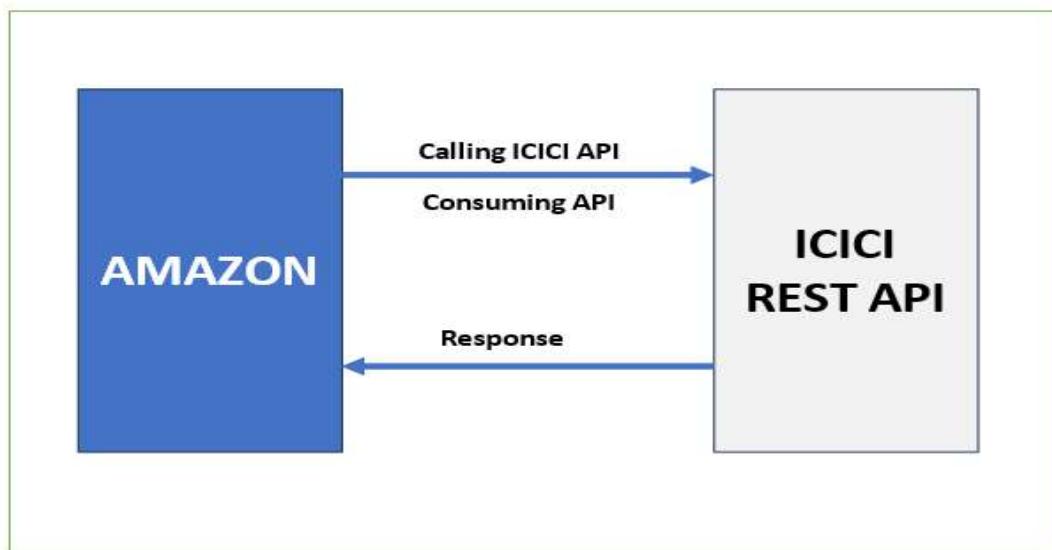
For Example,

ICICI bank will produce API services to enable banking functionalities. Now Amazon application integrated with ICICI REST services for performing Payment Options.

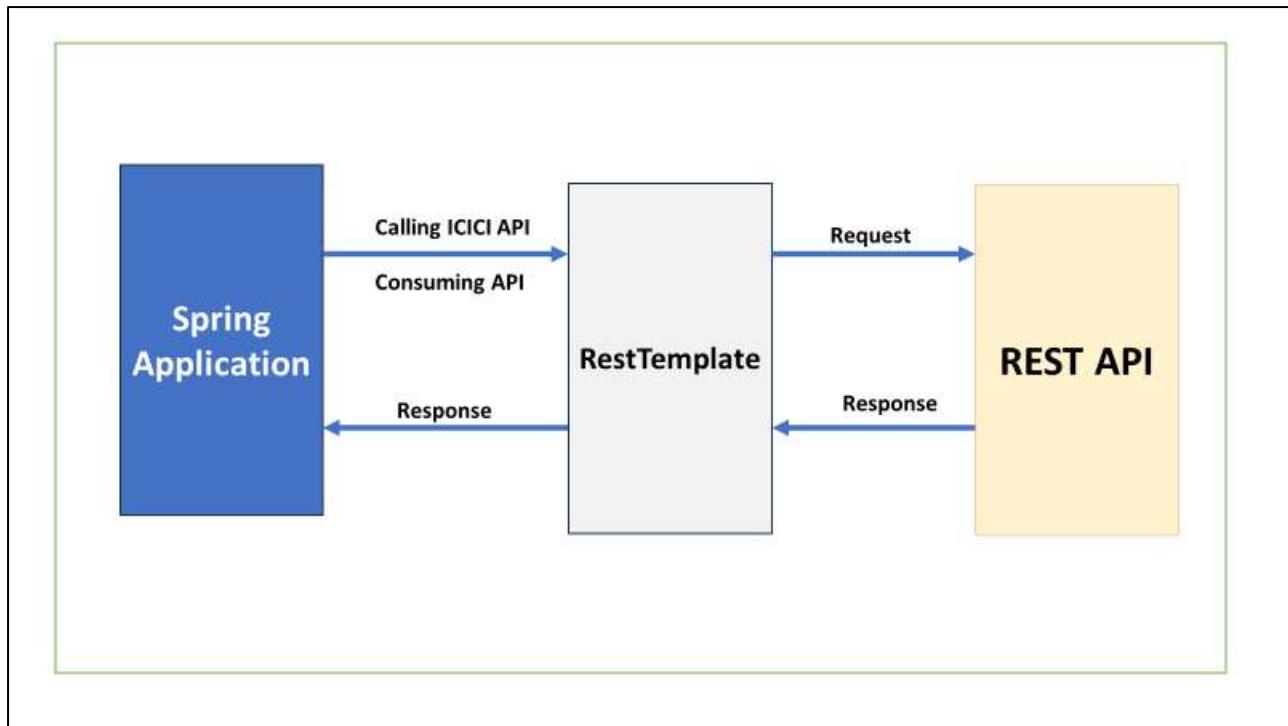
In This case:

Producer is : ICICI

Consumer is : Amazon



In Spring MVC, Spring Provided an HTTP or REST client class called as **RestTemplate** from package **org.springframework.web.client**. **RestTemplate** class provided multiple utility methods to consume REST API services from one application to another application.



RestTemplate is used to create applications that consume RESTful Web Services. You can use the **exchange()** or specific http methods to consume the web services for all HTTP methods.

Now we are trying to call Pharmacy Application API from our Spring Boot Application Flipkart i.e. **Flipkart consuming Pharmacy Application REST API**.

Now I am giving only API details of Pharmacy Application as swagger documentation. Because in Realtime Projects, swagger documentation or Postman collection data will be shared to Developers team, but not source code. So we will try to consume by seeing Swagger API documentation of tother application. When you are practicing also include swagger documentation to other application and try to implement by seeing swagger document only.

NOTE: Please makes sure other application running always to consume REST services.

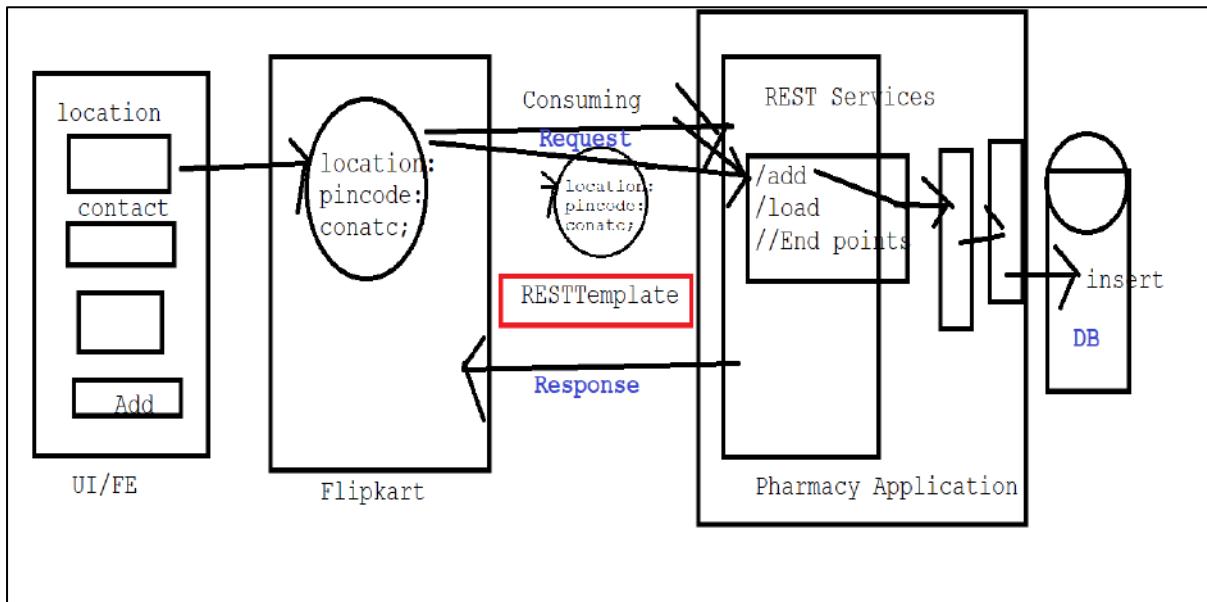
Below snap shows what are all services available in side pharmacy application.



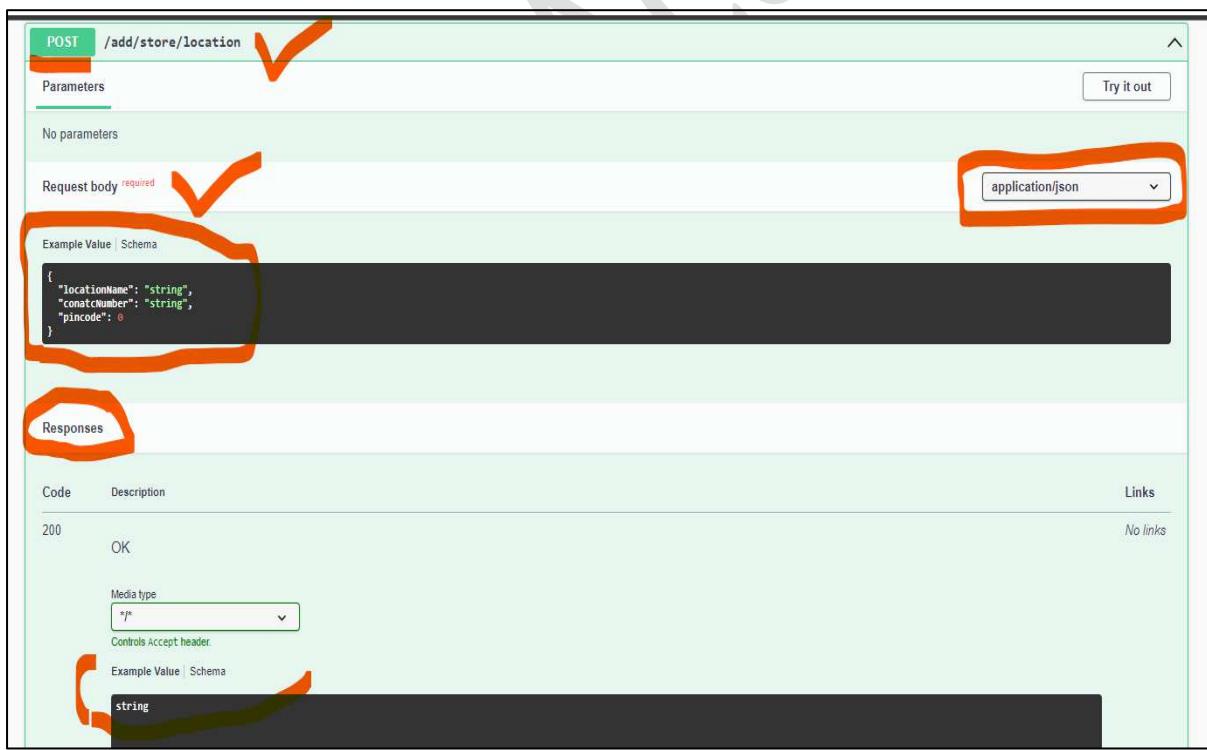
Method	Path
POST	/add/store/location
GET	/location/{locationName}
GET	/load/location
GET	/load/contact

Consuming REST Services with Request Body:

Requirement : Now I want to call Rest service `/add/store/location` of pharmacy application from my Flipkart application.



Go to swagger and expand details of `/add/store/location` in swagger documentation.



The screenshot shows the Swagger UI for the `/add/store/location` endpoint. It is a POST request. The 'Parameters' section shows 'No parameters'. The 'Request body' section is marked as 'required' and has a 'Schema' example:

```

{
  "locationName": "string",
  "conatcNumber": "string",
  "pincode": "string"
}

```

The 'Responses' section shows a 200 OK response with a 'Media type' of `*/*` and an 'Example Value' of `string`.

From above swagger snap, we should understand below points for that API call.

1. URL : <http://localhost:6677/pharmacy/add/store/location>
2. HTPP method: POST
3. Request Body Should contain below payload structure

```
{  
  "locationName": "hyderabad",  
  "conatcNumber": "323332323",  
  "pincode": 500099  
}
```

4. Response Receiving as String Format.

Same we can see in Postman as shown below.



POST <localhost:6677/pharmacy/add/store/location> Send

Params Authorization Headers (10) Body **Pre-request Script** Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL **JSON** x Beautify

```
1 --- "locationName": "hyderabad",
2 --- "conatctNumber": "323332323",
3 --- "pincode": 500099
4
5
```

Body Cookies Headers (5) Test Results

200 OK 48 ms 191 B Save as Example

Pretty Raw Preview Visualize Text copy

1 Added Details Successfully.

Now based on above data, we are going to write logic of **RestTemplate** to consume in our application flipkart.

Now assume we are receiving data from UI/Frontend to Flipkart application and that data we are transferring to Pharmacy API with help of **RestTemplate**.

NOTE : All code changes will happen only in flipkart application.

```
@RestController
@RequestMapping("/pharmacy")
public class PharmacyController {

    @Autowired
    PharmacyService pharmacyService;

    @PostMapping("/add/location")
    public String addPharmacyDetails(@RequestBody PharmacyLocation request) {
        return pharmacyService.addPharmacyDetails(request);
    }
}
```

- Now in Service class, we should write logic of integrating Pharmacy endpoint for adding store details as per swagger notes.

- Create a POJO class which is equal to JSON Request payload of Pharmacy API call.

Request body required

[Example Value](#) | [Schema](#)

```
{
  "locationName": "string",
  "conatcNumber": "string",
  "pincode": 0
}
```

PharmacyData.java : This Object will be used as Request Body

```
public class PharmacyData {

    private String locationName;
    private String conatcNumber;
    private int pincode;

    //setters and getters
}
```

HttpEntity:

HttpEntity class is used to represent an HTTP request or response entity. It encapsulates/binds the HTTP message's headers and body. You can use **HttpEntity** to customize the headers and body of the HTTP request before sending it using **RestTemplate**. It provides more control and flexibility over the request or response compared to simpler methods like `getForEntity()`, `postForObject()`, etc.

Here's how you can use **HttpEntity** in **RestTemplate**:

- Now In service layer, Please map data from controller layer to API request body class.

```
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpMethod;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
import com.flipkart.dto.PharmacyData;
import com.flipkart.pharmacy.request.PharmacyLocation;

@Service
public class PharmacyService {

    public String addPharmacyDetails(PharmacyLocation location) {
        //complete URL of pharmacy endpoint.
        String url = "http://localhost:6677/pharmacy/add/store/location";
    }
}
```

```

//Mapping from flipkart request object to JSON payload Object of class i.e.
PharmacyData
// Java Object which should be aligned to Pharmacy POST end point Request body.
PharmacyData data = new PharmacyData();
data.setConatcNumber(location.getContact());
data.setLocationName(location.getLocation());
data.setPincode(location.getPincode());

// converting our java object to HttpEntity : i.e. Request Body
HttpEntity<PharmacyData> body = new HttpEntity<PharmacyData>(data);
RestTemplate restTemplate = new RestTemplate();
return restTemplate.exchange(url, HttpMethod.POST, body, String.class).getBody();
}
}

```

- Now Test it from Postman and check pharmacy API call triggered or not i.e. check data is inserted in DB or not from pharmacy application.

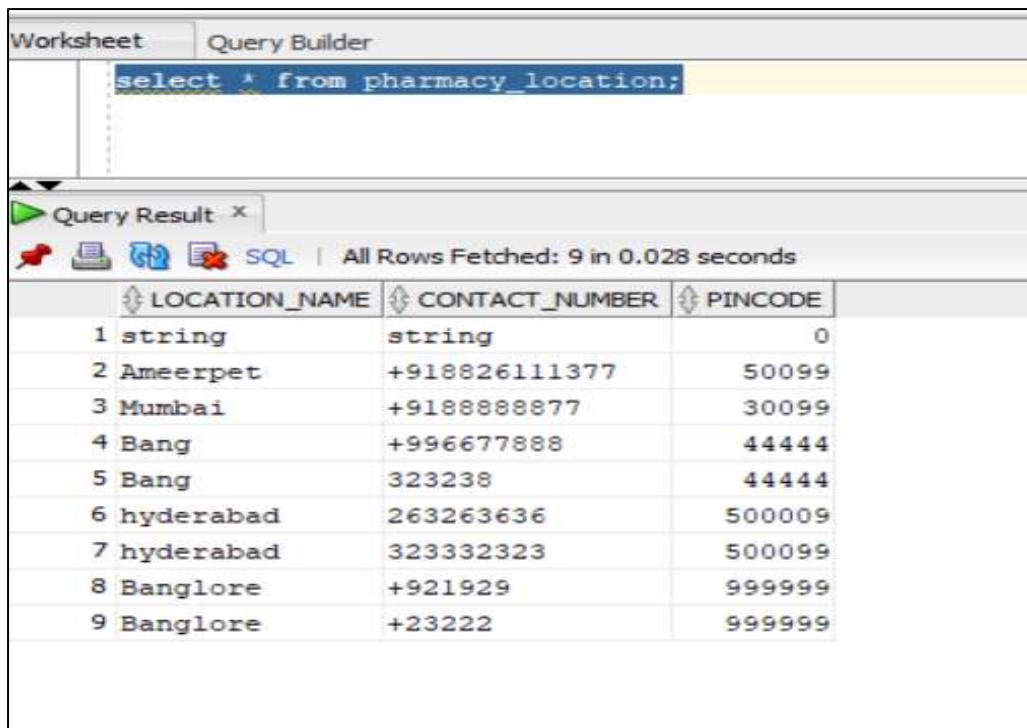
flipkart URL : **localhost:9966/flipkart/pharmacy/add/location**

- **Now create Request body as per our controller request body class.**

```
{
  "location": "pune",
  "contact": "+918125262702",
  "pincode": 500088
}
```

- Before executing from post man, please check DB data. In my table I have below data right now.





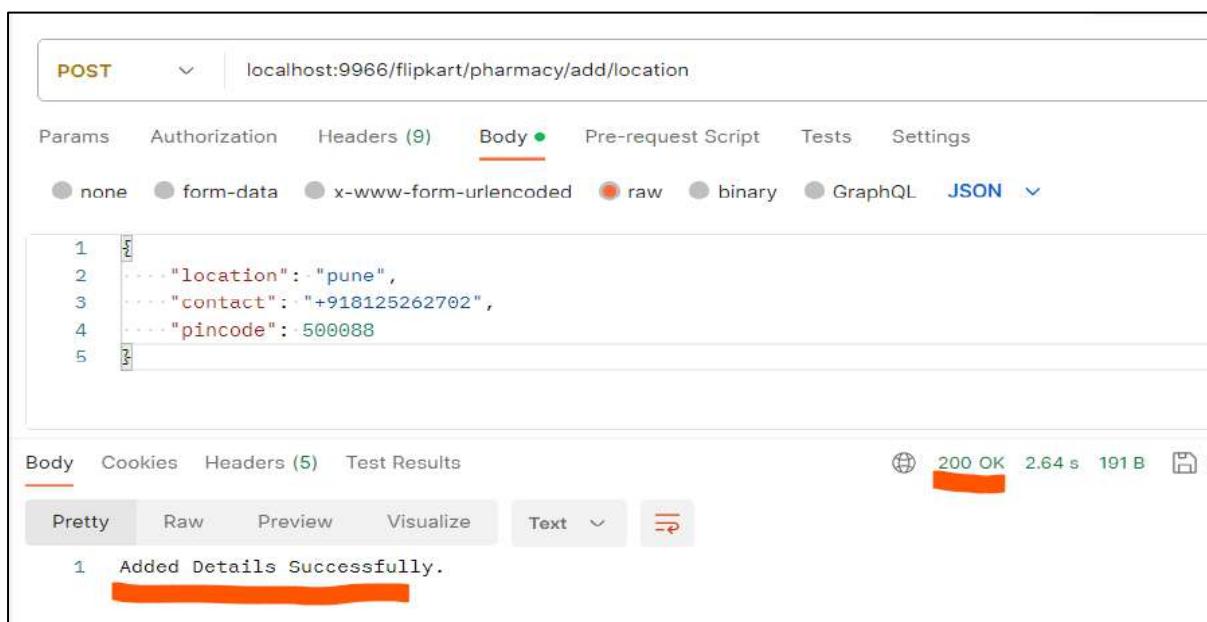
Worksheet | Query Builder

```
select * from pharmacy_location;
```

Query Result | All Rows Fetched: 9 in 0.028 seconds

	LOCATION_NAME	CONTACT_NUMBER	PINCODE
1	string	string	0
2	Ameerpet	+918826111377	50099
3	Mumbai	+9188888877	30099
4	Bang	+996677888	44444
5	Bang	323238	44444
6	hyderabad	263263636	500009
7	hyderabad	323332323	500099
8	Banglore	+921929	999999
9	Banglore	+23222	999999

- Now from postman send request as per flipkart controller method.



POST | localhost:9966/flipkart/pharmacy/add/location

Params | Authorization | Headers (9) | **Body** | Pre-request Script | Tests | Settings

Body Type: JSON

```

1 {
2   "location": "pune",
3   "contact": "+918125262702",
4   "pincode": 500088
5 }
```

Body | Cookies | Headers (5) | Test Results

200 OK | 2.64 s | 191 B | 

Pretty | Raw | Preview | Visualize | Text | 

1 Added Details Successfully.

- Request executed successfully and you got response from Pharmacy API of post REST API call what we integrated. Verify In Database record inserted or not. It's inserted.

Worksheet Query Builder

```
select * from pharmacy_location;
```

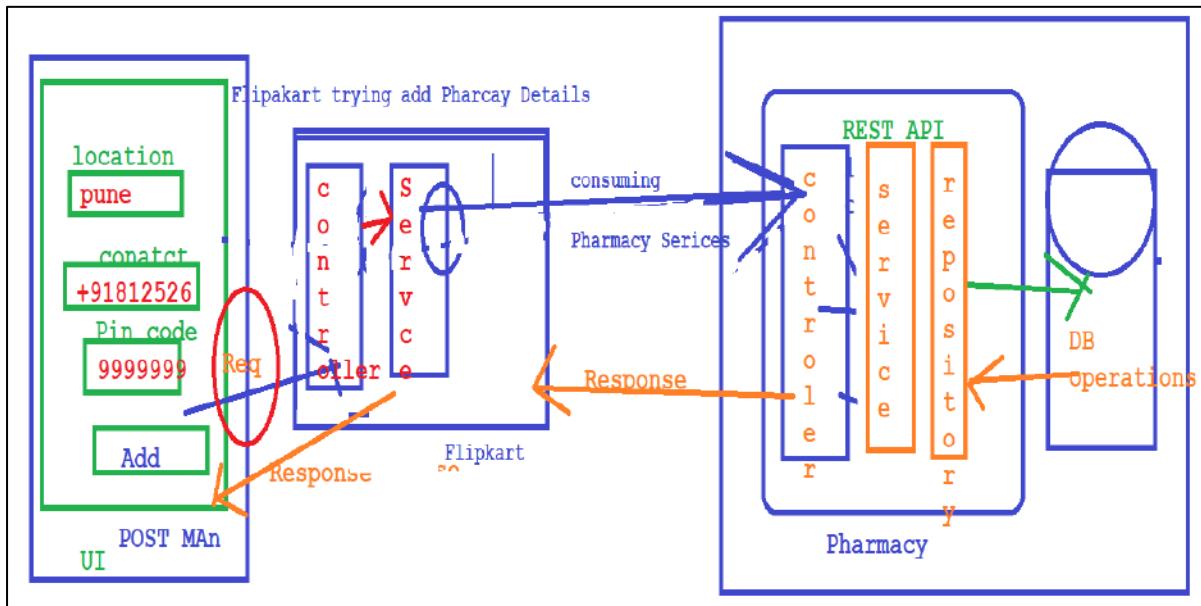
Query Result x

SQL | All Rows Fetched: 10 in 0.001 seconds

LOCATION_NAME	CONTACT_NUMBER	PINCODE
1 string	string	0
2 pune	+918125262702	500088
3 Ameerpet	+918826111377	50099
4 Mumbai	+91888888877	30099
5 Bang	+996677888	44444
6 Bang	323238	44444
7 hyderabad	263263636	500009
8 hyderabad	323332323	500099
9 Banglore	+921929	999999
10 Banglore	+23222	999999

Internal Execution/Workflow:

When we are sending data to flipkart app, now flipkart app forwarded data to pharmacy application via REST API call.



Now Let's integrate Path variable and Query Parameters REST API Services:

Consuming API Services with Query Parameters:

Example1 : Consume below Service which contains Query String i.e. Query Parameters.

Servers
http://localhost:6677/pharmacy - Generated server url

pharmacy-controller

POST /add/store/location

GET /location

Parameters

Name	Description
locationName * required string (query)	locationName

Responses

Code	Description
200	OK

Media type
/
Controls Accept header.

Example Value | Schema

```
[  
  {  
    "locationName": "string",  
    "conatcNumber": "string",  
    "Pincode": 0  
  }  
]
```

Consuming GET API Service with Query Parameter:

In **RestTemplate**, to handle Query Parameters Spring provided flexibility with **HashMap** Object i.e. Configuring Query parameters with values key and values. Above Service Producing Response as JSON array of Objects. So create Response Class.

PharmacyResponse.java:

```
public class PharmacyResponse {  
  
    private String locationName;  
    private String conatcNumber;  
    private int pincode;  
  
    //Setters and Getters  
}
```

- From above API details, we have one Request Parameter : **locationName**.

```
public List<PharmacyResponse> loadDetailsByLocationName(String location) {  
  
    String url = "http://localhost:6677/pharmacy/location?locationName={locationName}";
```

```
// For Query parameters
Map<String, String> values = new HashMap<>();
values.put("locationName", location); // passing value with location

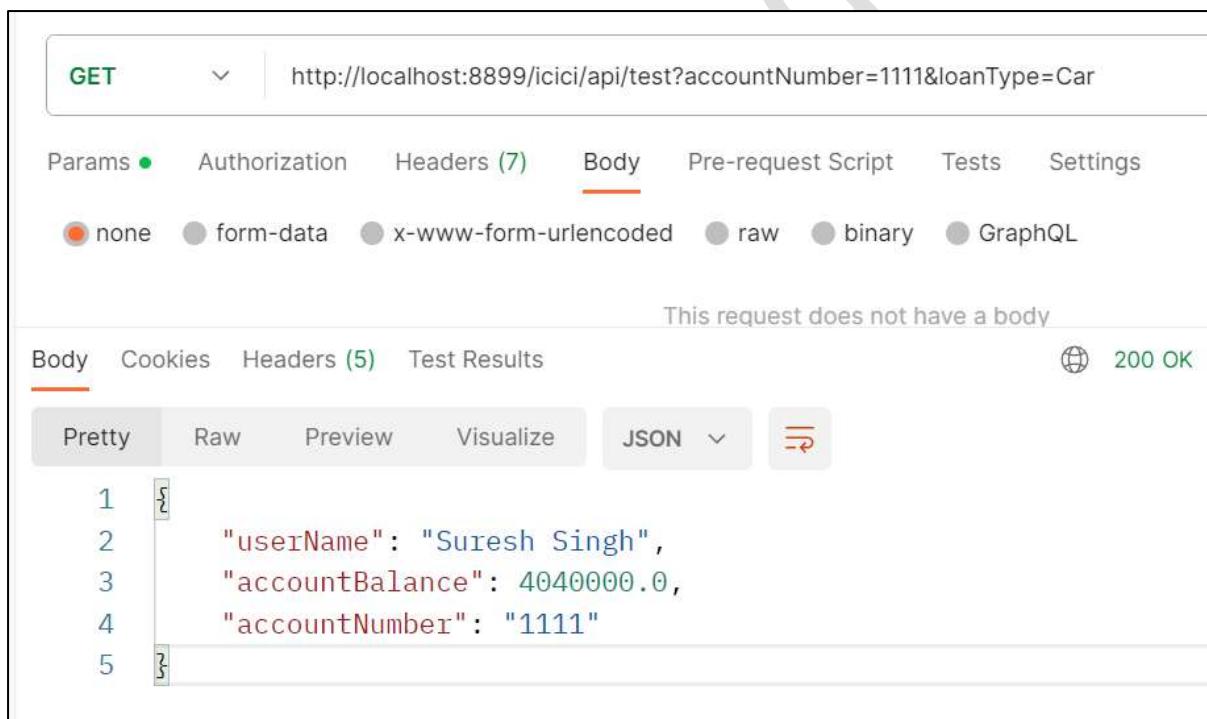
RestTemplate restTemplate = new RestTemplate();
List<PharmacyResponse> response = restTemplate.exchange(url, HttpMethod.GET,
    null, List.class, values).getBody();

return response;
}
```

Consuming Another Example with Query Parameters:

- Consume Below REST Service which contains Query Parameters From our Application.

Source of REST Service to be consumed:



GET http://localhost:8899/icici/api/test?accountNumber=1111&loanType=Car

Params • Authorization Headers (7) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (5) Test Results 200 OK

Pretty Raw Preview Visualize JSON

```

1 {
2   "userName": "Suresh Singh",
3   "accountBalance": 4040000.0,
4   "accountNumber": "1111"
5 }
```

- Logic for Consumption:

```
package com.flipkart.service;

import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.UriComponentsBuilder;

public class RestTemplateExample {
```

```

public static void main(String[] args) {

    // Create a RestTemplate instance
    RestTemplate restTemplate = new RestTemplate();

    // Define the base URL of the API
    String baseUrl = "http://localhost:8899/icici/api/test";

    // Create query parameters using UriComponentsBuilder
    UriComponentsBuilder builder = UriComponentsBuilder.fromHttpUrl(baseUrl)
        .queryParam("accountNumber", "122334455")
        .queryParam("loanType", "House");

    // Build the final URL with query parameters
    String finalUrl = builder.toUriString();

    // Make a GET request to the API
    ResponseEntity<String> response = restTemplate.getForEntity(finalUrl, String.class);

    // Process the response
    if (response.getStatusCode().is2xxSuccessful()) {
        String responseBody = response.getBody();
        System.out.println(responseBody);
    } else {
        System.err.println("Request failed with status code: " + response.getStatusCode());
    }
}
}

```

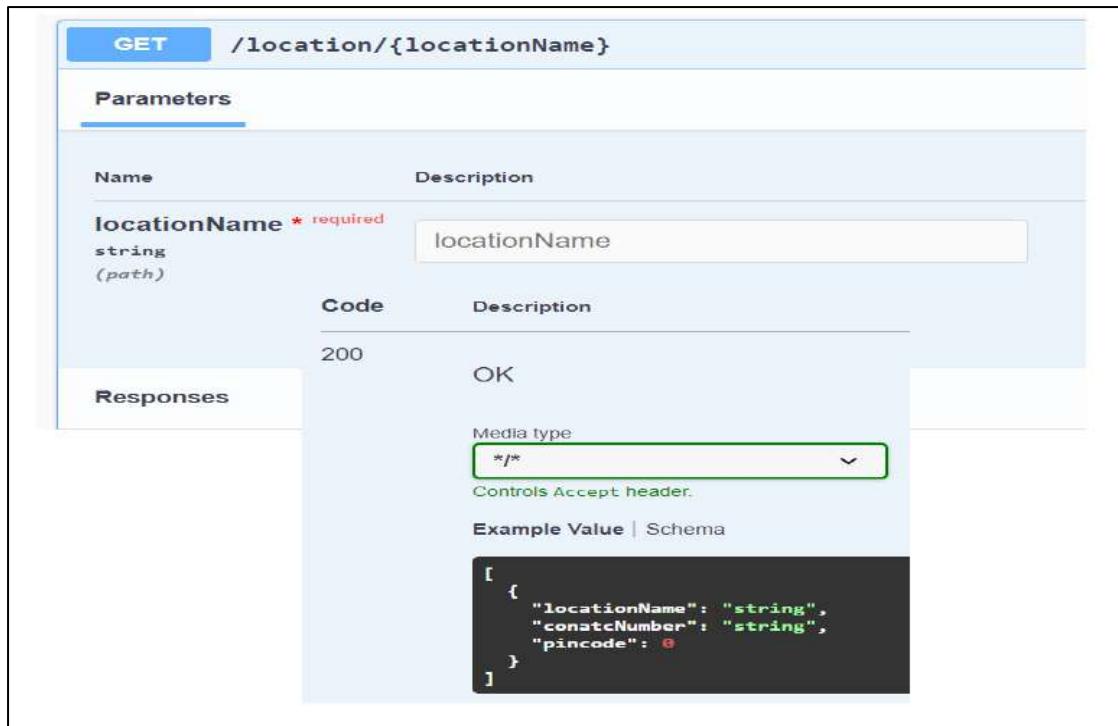
Output:

```
{
  "userName": "Suresh Singh",
  "accountBalance": 4040000.0,
  "accountNumber": "122"
}
```

Consuming GET API Service with Path Parameter:

Example : Consume below Service which contains Path Variable.

In **RestTemplate**, to handle Path Parameters Spring provided flexibility with Hashmap Object or Object Type Variable Arguments as part of exchange() i.e. Configuring Path parameters with values. Above Service Producing Response as JSON array of Objects. So create Response Class.



GET /location/{locationName}

Parameters

Name	Description
locationName <small>* required</small> string (path)	locationName

Responses

Code	Description
200	OK

Media type: `/*`
Controls Accept header.

Example Value | Schema

```
[  
  {  
    "locationName": "string",  
    "conatcNumber": "string",  
    "pincode": 0  
  }  
]
```

PharmacyResponse.java

```
public class PharmacyResponse {  
  
    private String locationName;  
    private String conatcNumber;  
    private int pincode;  
  
    //Setters and Getters  
}
```

- From above API details, we have one Path Parameter : `locationName`.

```
public List<PharmacyResponse> loadByLocationName(String location) {  
  
    String url = "http://localhost:6677/pharmacy/location/{locationName}";  
  
    Map<String, String> values = new HashMap<>();  
    values.put("locationName", location);  
  
    RestTemplate restTemplate = new RestTemplate();  
    List<PharmacyResponse> response = restTemplate.exchange(url, HttpMethod.GET,  
                                              null, List.class, values).getBody();  
  
    return response;  
}
```

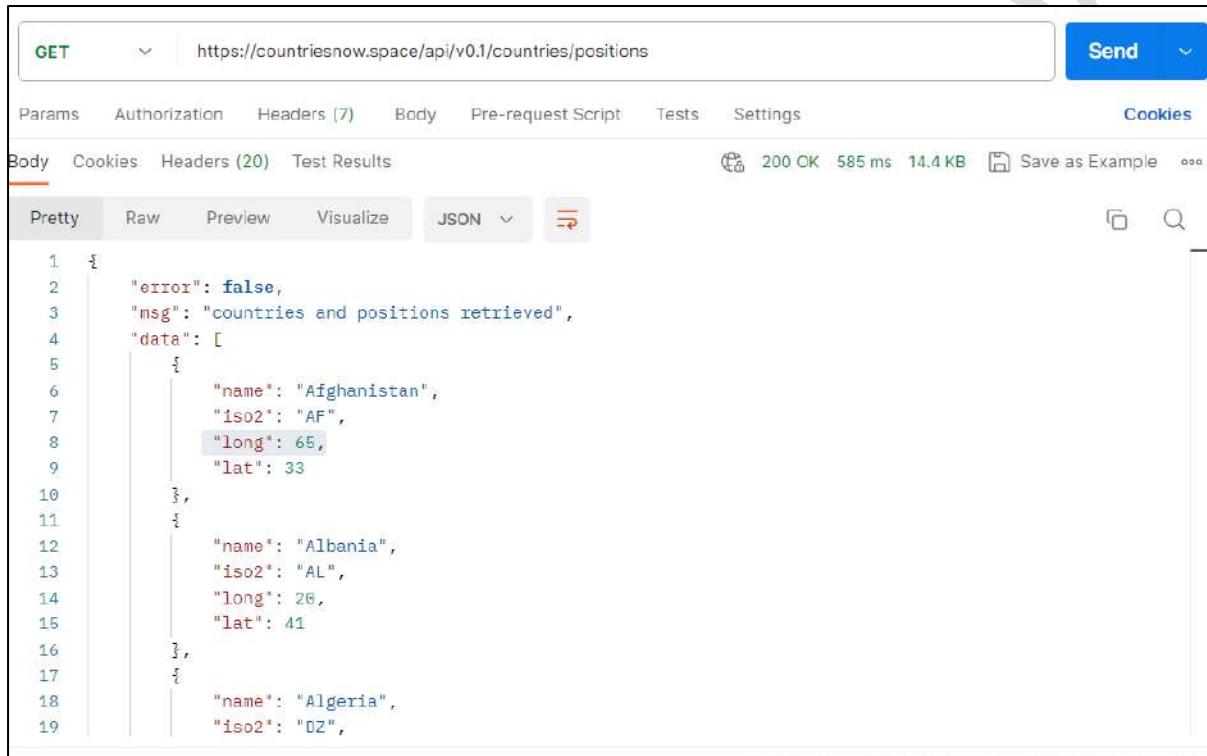
NOTE: We can handle both Path variable and Query Parameters of a single URI with Hashmap Object. i.e. We are passing values to keys. Internally spring will replace values specifically.

Integration of One More REST API Service:

- **Example 3:** We are Integrating one Real time API service from Online.

REST API GET URL: <https://countriesnow.space/api/v0.1/countries/positions>

Above API call, Producing JSON Response, as shown in below Postman. Depends on Response we should create JAVA POJO classes to serialize data from JAVA to JSON and vice versa.



```

1  {
2      "error": false,
3      "msg": "countries and positions retrieved",
4      "data": [
5          {
6              "name": "Afghanistan",
7              "iso2": "AF",
8              "long": 65,
9              "lat": 33
10         },
11         {
12             "name": "Albania",
13             "iso2": "AL",
14             "long": 20,
15             "lat": 41
16         },
17         {
18             "name": "Algeria",
19             "iso2": "DZ",
20         }
21     ]
22 }
  
```

- Based on API call Response, we should create Response POJO classes aligned to JSON Payload.

Country.java

```

public class Country {
    private String name;
    private String iso2;
    private int lat;

    //Setters and Getters
}
  
```

CountriesResponse.java

```

public class CountriesResponse {
  
```

```
private boolean error;
private String msg;
private List<Country> data;

//Setters and Getters
}
```

API Consuming Logic:

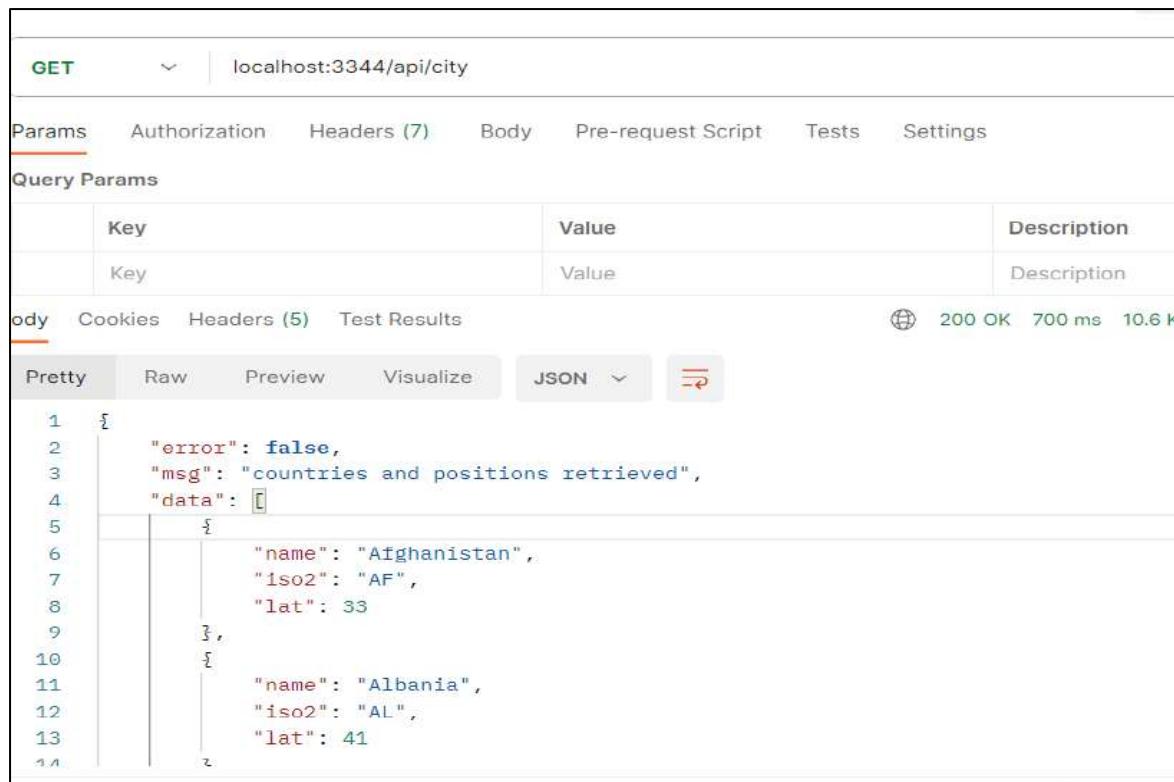
```
public CountriesResponse loadCities() {

    String url = "https://countriesnow.space/api/v0.1/countries/positions";

    RestTemplate restTemplate = new RestTemplate();
    CountriesResponse respone = restTemplate.exchange(url, HttpMethod.GET,
            null, CountriesResponse.class).getBody();

    System.out.println(respone);
    return respone;
}
```

- Testing from our Application Postman:



GET localhost:3344/api/city

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```

1  {
2      "error": false,
3      "msg": "countries and positions retrieved",
4      "data": [
5          {
6              "name": "Afghanistan",
7              "iso2": "AF",
8              "lat": 33
9          },
10         {
11             "name": "Albania",
12             "iso2": "AL",
13             "lat": 41
14         }
15     ]
16 }

```

How to Pass Headers with RestTemplate when Consuming REST Services:

In Spring's **RestTemplate**, we can work with HTTP headers by using the **HttpHeaders** class. You can add, retrieve, and manipulate headers in both requests and responses. Here's how we can work with headers in **RestTemplate**:

Adding Headers to a Request:

In this example, we create an **HttpHeaders** object and set custom headers. We can add headers to your HTTP request before sending it using RestTemplate. Here's an example of how to add headers to a request:

```

import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class RestTemplateHeadersExample {
    public static void main(String[] args) {

        // Create a RestTemplate instance
        RestTemplate restTemplate = new RestTemplate();

        // Define the request URL
        String url = "https://api.example.com/api/resource";
    }
}

```

```

// Create an HttpHeaders object to set custom headers
HttpHeaders headers = new HttpHeaders();
headers.set("Authorization", "Bearer yourAccessToken");
headers.set("Custom-Header", "Custom-Value");
headers.add("token", "dss3232444gt54t5tgrgtry54y5ydsdsdsdsdsd");

HttpEntity<Object> entity = new HttpEntity<Object>(headers);

// Create a HttpEntity with the custom headers
 ResponseEntity<String> responseEntity =
        restTemplate.exchange(url, HttpMethod.POST, entity, String.class);

// Process the response
if (responseEntity.getStatusCode().is2xxSuccessful()) {
    String responseBody = responseEntity.getBody();
    System.out.println("Response: " + responseBody);
} else {
    System.err.println("Request failed with status code: " +
        responseEntity.getStatusCode());
}
}
}

```

Accessing Headers in a Response:

We can access response headers when you receive a response from the server. Here's an example:

```

import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class RestTemplateResponseHeadersExample {

    public static void main(String[] args) {
        // Create a RestTemplate instance
        RestTemplate restTemplate = new RestTemplate();

        // Define the request URL
        String url = "https://api.example.com/api/resource";

        // Send a GET request and receive the entire ResponseEntity for the response
        ResponseEntity<String> responseEntity = restTemplate.getForEntity(url, String.class);

        // Access response headers
        HttpHeaders responseHeaders = responseEntity.getHeaders();
        String contentType = responseHeaders.getFirst("Content-Type");
        long contentLength = responseHeaders.getContentLength();
    }
}

```

```

System.out.println("Content-Type: " + contentType);
System.out.println("Content-Length: " + contentLength);

// Access the response body
String responseBody = responseEntity.getBody();
System.out.println("Response Body: " + responseBody);
}
}

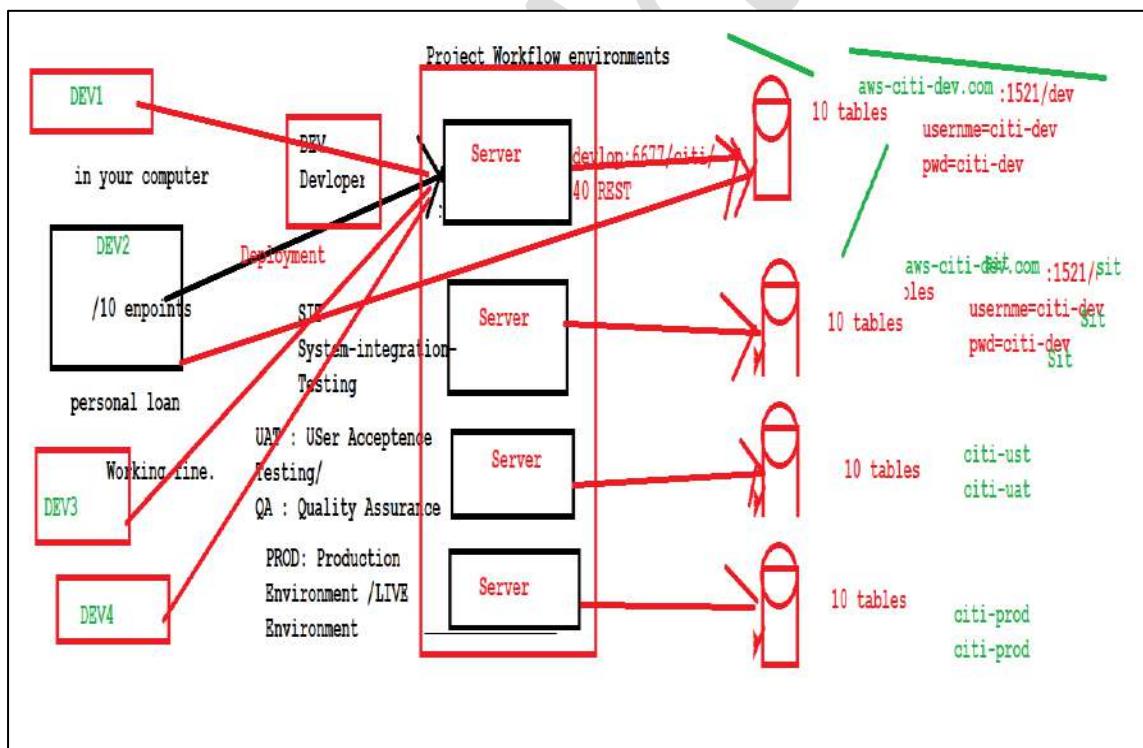
```

In this example, we use **responseEntity.getHeaders()** to access the response headers and then retrieve specific header values using **responseHeaders.getFirst("Header-Name")**.

Working with headers allows you to customize your requests and process responses more effectively in your **RestTemplate** interactions.

Spring Boot Profiles:

Every enterprise application has many environments, like: Dev, Sit, UAT, Prod. Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments.



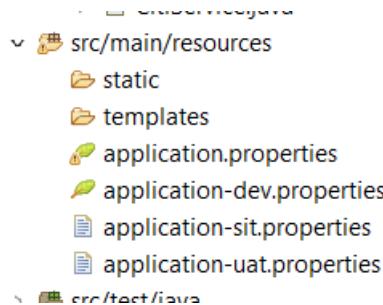
Each environment requires a setting that is specific to them. For example, in DEV, we do not need to constantly check database consistency. Whereas in UAT and PROD, we need to.

These environments host specific configurations called Profiles.

How Do we Maintain Profiles?

This is simple — properties files!

We make properties files for each environment and set the profile in the application accordingly, so it will pick the respective properties file. Don't worry, we will see how to set it up.



In this demo application, we will see how to configure different databases at runtime based on the specific environment by their respective profiles.

As the DB connection is better to be kept in a property file, it remains external to an application and can be changed. We will do so here. But, Spring Boot — by default — provides just one property file (**application.properties**). So, how will we segregate the properties based on the environment?

The solution would be to create more property files and add the "**profile**" name as the suffix and configure Spring Boot to pick the appropriate properties based on the profile.

Then, we need to create three `application-<profile>.properties`:

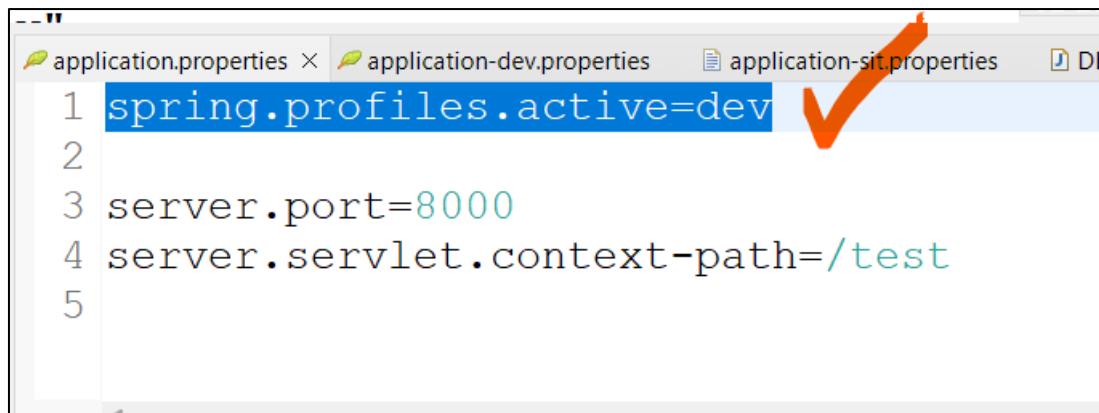
- `application-dev.properties`
- `application-sit.properties`
- `application-uat.properties`

Of course, the **application.properties** will remain as a master properties file, but if we override any key in the profile-specific file, then it will take priority.

Generally profile files will be created specific to Environments in projects. So we will configure properties and value which are really related to that environment. For example, In Real time Projects implementation, we will have different databases i.e. different database hostnames, user name and passwords for different environments. We will define common properties and values across all environments in side main **application.properties** file.

How to run Application with Specific Profile:

In side **application.properties** file, we have to add a property called **spring.profiles.active** with profile value.



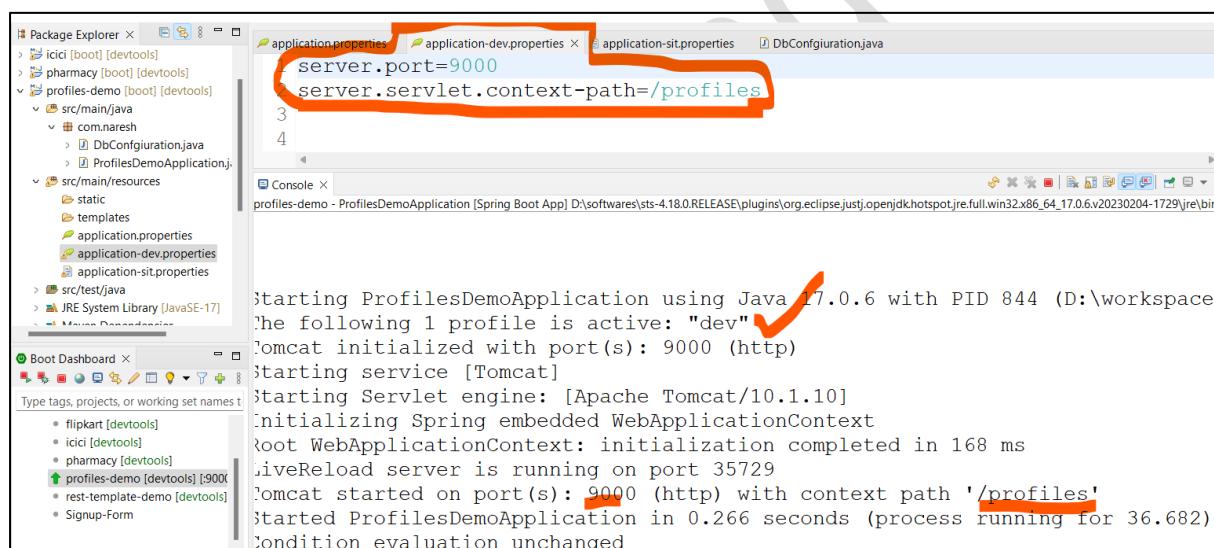
```

application.properties x application-dev.properties application-sit.properties
1 spring.profiles.active=dev
2
3 server.port=8000
4 server.servlet.context-path=/test
5

```

Now run application as SpringBoot or Java application, SpringBoot will load by default properties of **application.properties** and loads configured profiles properties file **application-dev.properties** file.

NOTE: Whenever we have same property in main **application.properties** and **application-<profile>.properties**, priority given to profile specific property and it's value.



application.properties

```

server.port=9000
server.servlet.context-path=/profiles

```

Console

```

starting ProfilesDemoApplication using Java 17.0.6 with PID 844 (D:\workspace
The following 1 profile is active: "dev"
Tomcat initialized with port(s): 9000 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/10.1.10]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 168 ms
LiveReload server is running on port 35729
Tomcat started on port(s): 9000 (http) with context path '/profiles'
Started ProfilesDemoApplication in 0.266 seconds (process running for 36.682)
Condition evaluation unchanged

```

This is how we are running application with specific profile i.e. loading specific profiles properties file.

Now, we are done with properties files. Let's configure in the Configuration classes to pick the correct properties.

For Example, Database Connection should be created for specific profile or environment from configuration class.

```
import org.springframework.beans.factory.annotation.Value;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
public class DbConfiguration {

    @Value("${db.hostName}")
    String hostName;

    @Value("${db.userName}")
    String userName;

    @Value("${db.password}")
    String password;

    @Profile("sit")
    @Bean
    public String getSitDBConnection() {
        System.out.println("SIT Creating DB Connection");
        System.out.println(hostName);
        System.out.println(userName);
        System.out.println(password);
        return "SIT DB Connection Successfu.";
    }

    @Profile("dev")
    @Bean
    public String getDevDBConnection() {
        System.out.println("Creating DEV DB Connection");
        System.out.println(hostName);
        System.out.println(userName);
        System.out.println(password);

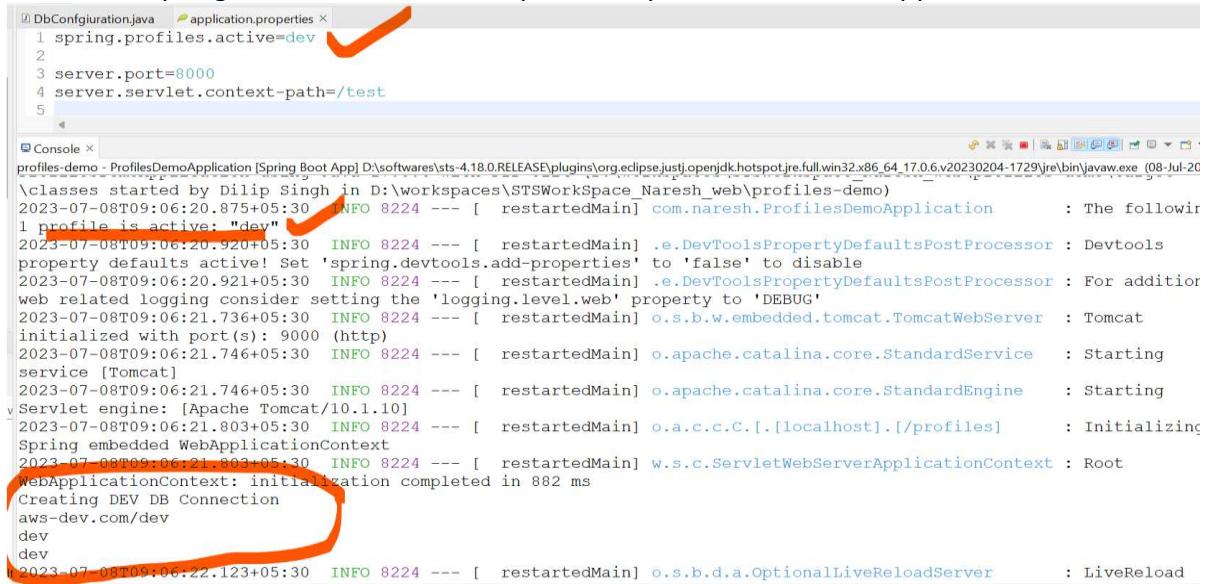
        return "DEV DB Connection Successfu.";
    }
}
```

We have used the `@Profile("dev")` and `@Profile("sit")` for specific profiles to pickup properties and create specific bean Objects. So when we start our application with "dev" profile, only `@Profile("dev")` bean object will be created not `@Profile("sit")` object i.e. The other profile beans will not be created at all.

How application knows that this is DEV or SIT profile? how do we do this?

We will use `application.properties` with property `spring.profiles.active=<profile>`

From here, Spring Boot will know which profile to pick. Let's run the application now!



```

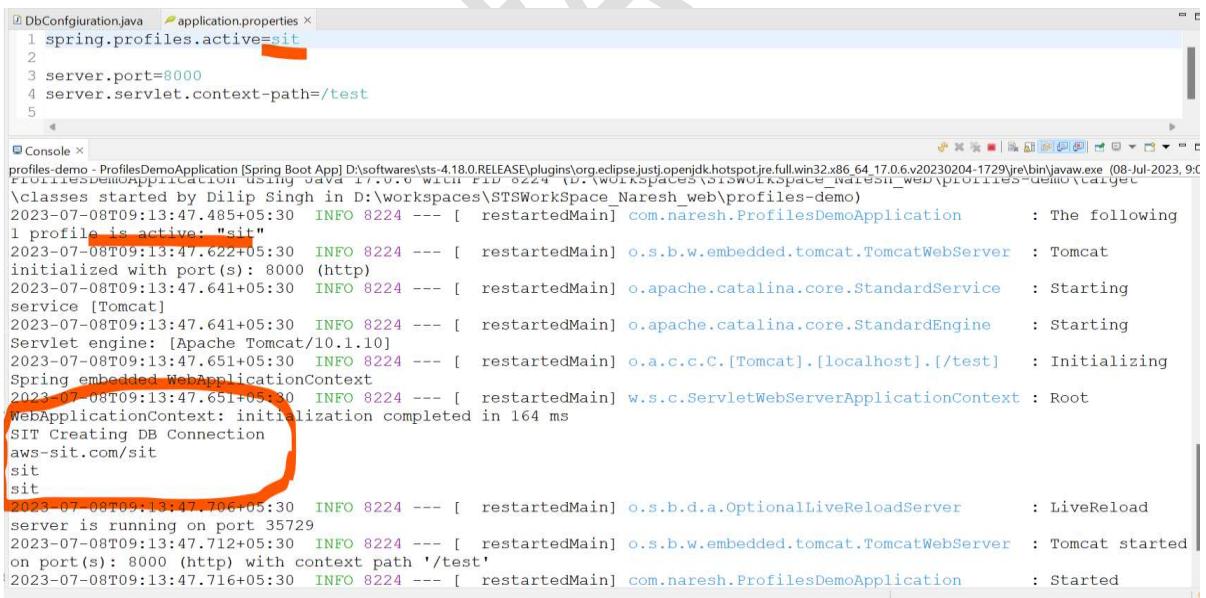
DbConfiguration.java application.properties
1 spring.profiles.active=dev
2
3 server.port=8000
4 server.servlet.context-path=/test
5

Console x
profiles-demo - ProfilesDemoApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.jst.jdt.openjdk.hotspot.jre.full.win32.x86_64.17.0.6.v20230204-1729\jre\bin\javaw.exe (08-Jul-2023-07-08T09:06:20.875+05:30) INFO 8224 --- [ restartedMain] com.naresh.ProfilesDemoApplication : The following 1 profile is active: "dev"
2023-07-08T09:06:20.920+05:30 INFO 8224 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
2023-07-08T09:06:20.921+05:30 INFO 8224 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to 'DEBUG'
2023-07-08T09:06:21.736+05:30 INFO 8224 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 9000 (http)
2023-07-08T09:06:21.746+05:30 INFO 8224 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2023-07-08T09:06:21.746+05:30 INFO 8224 --- [ restartedMain] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.10]
2023-07-08T09:06:21.803+05:30 INFO 8224 --- [ restartedMain] o.a.c.c.C.[localhost].[/profiles] : Initializing Spring embedded WebApplicationContext
2023-07-08T09:06:21.803+05:30 INFO 8224 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 882 ms
Creating DEV DB Connection
aws-dev.com/dev
dev
dev
2023-07-08T09:06:22.123+05:30 INFO 8224 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload

```

We are not seeing any details of **sit** profile bean configuration i.e. skipped Bean creation because active profile is **dev**.

Now Let's change our active profile to **sit** and observe which Bean object created and which are ignored by Spring.



```

DbConfiguration.java application.properties
1 spring.profiles.active=sit
2
3 server.port=8000
4 server.servlet.context-path=/test
5

Console x
profiles-demo - ProfilesDemoApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.jst.jdt.openjdk.hotspot.jre.full.win32.x86_64.17.0.6.v20230204-1729\jre\bin\javaw.exe (08-Jul-2023, 9:00:00 AM)
profiles-demo - ProfilesDemoApplication using Java 17.0.6 with PID 8224 (D:\workspaces\STSWorkSpace_Naresh_web\profiles-demo\target\classes started by Dilip Singh in D:\workspaces\STSWorkSpace_Naresh_web\profiles-demo)
2023-07-08T09:13:47.485+05:30 INFO 8224 --- [ restartedMain] com.naresh.ProfilesDemoApplication : The following 1 profile is active: "sit"
2023-07-08T09:13:47.622+05:30 INFO 8224 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8000 (http)
2023-07-08T09:13:47.641+05:30 INFO 8224 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2023-07-08T09:13:47.641+05:30 INFO 8224 --- [ restartedMain] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.10]
2023-07-08T09:13:47.651+05:30 INFO 8224 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/test] : Initializing Spring embedded WebApplicationContext
2023-07-08T09:13:47.651+05:30 INFO 8224 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 164 ms
SIT Creating DB Connection
aws-sit.com/sit
sit
sit
2023-07-08T09:13:47.706+05:30 INFO 8224 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2023-07-08T09:13:47.712+05:30 INFO 8224 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8000 (http) with context path '/test'
2023-07-08T09:13:47.716+05:30 INFO 8224 --- [ restartedMain] com.naresh.ProfilesDemoApplication : Started

```

That's it! We just have to change it in **application.properties** to let Spring Boot know which environment the code is deployed in, and it will do the magic with the setting.

Spring Boot Dev Tools:

Spring Boot provides another module called Spring Boot DevTools. DevTools stands for **Developer Tool**. The aim of the module is to try and improve the development time while working with the Spring Boot application. Spring Boot DevTools pick up the changes and restart the application.

We can implement the DevTools in our project by adding the following dependency in the pom.xml file.

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

Some of the key features include:

Automatic Restart: DevTools enables automatic restart of the application when changes are detected in the classpath. This means that developers don't need to manually stop and restart the application every time they make changes to the code.

Live Reload: DevTools integrates with live reload functionality, allowing changes to static resources (such as HTML, CSS, or JavaScript files) to be immediately reflected in the browser without requiring a manual page refresh.

Remote Development: DevTools supports remote development, allowing developers to connect to a remote JVM and trigger automatic restarts and live reloads remotely.

Additional Developer Tools: DevTools also includes additional developer-friendly features such as embedded H2 database console, which provides a web-based interface to interact with an H2 in-memory database, and property defaults, which automatically configures default properties for the development environment.

Overall, Spring Boot DevTools significantly improves the developer experience by reducing the time spent on manual tasks like restarting the application and refreshing the browser and by providing additional tools to aid in the development process.

Spring Boot Actuator:

In Spring Boot, an actuator is a set of endpoints that provides various production-ready features to help monitor and manage your application. It exposes useful endpoints that give insights into your application's health, metrics, environment, and more. Actuators are essential for monitoring and managing your Spring Boot application in production environments.

To enable the Spring Boot Actuator, you need to add the relevant dependencies to your project. In most cases, you'll want to include the `spring-boot-starter-actuator` dependency in your pom.xml (Maven) or build.gradle (Gradle) file.

For Maven:

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

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the **health** endpoint provides basic application health information.

The built-in endpoints are auto-configured only when they are available. Most applications choose exposure over HTTP, where the ID of the endpoint and a prefix of **/actuator** is mapped to a URL. For example, by default, the health endpoint is mapped to **/actuator/health**

Some of endpoints are:

ID	Description
beans	Displays a complete list of all the Spring beans in your application.
health	Shows application health information.
info	Displays arbitrary application info.
loggers	Shows and modifies the configuration of loggers in the application.

Exposing Endpoints:

By default, only the **health** endpoint is exposed. Since Endpoints may contain sensitive information, you should carefully consider when to expose them. To change which endpoints are exposed, use the following specific **include** and **exclude** properties:

Property

```
management.endpoints.web.exposure.exclude=<endpoint>,<endpoint>
management.endpoints.web.exposure.include=<endpoint>,<endpoint>
```

* can be used to select all endpoints. For example, to expose everything over HTTP except the env and beans endpoints, use the following properties:

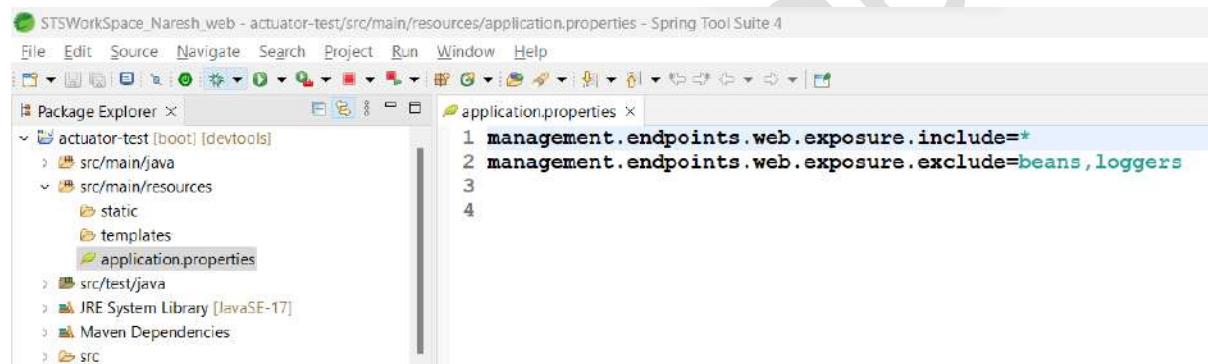
Properties:

```
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=env,beans
```

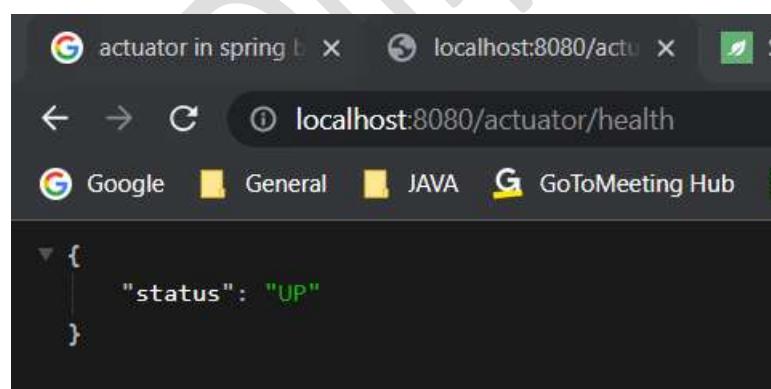
For security purposes, only the **/health** endpoint is exposed over HTTP by default. You can use the **management.endpoints.web.exposure.include** property to configure the endpoints that are exposed.

Before setting the **management.endpoints.web.exposure.include**, ensure that the exposed actuators do not contain sensitive information, are secured by placing them behind a firewall, or are secured by something like Spring Security.

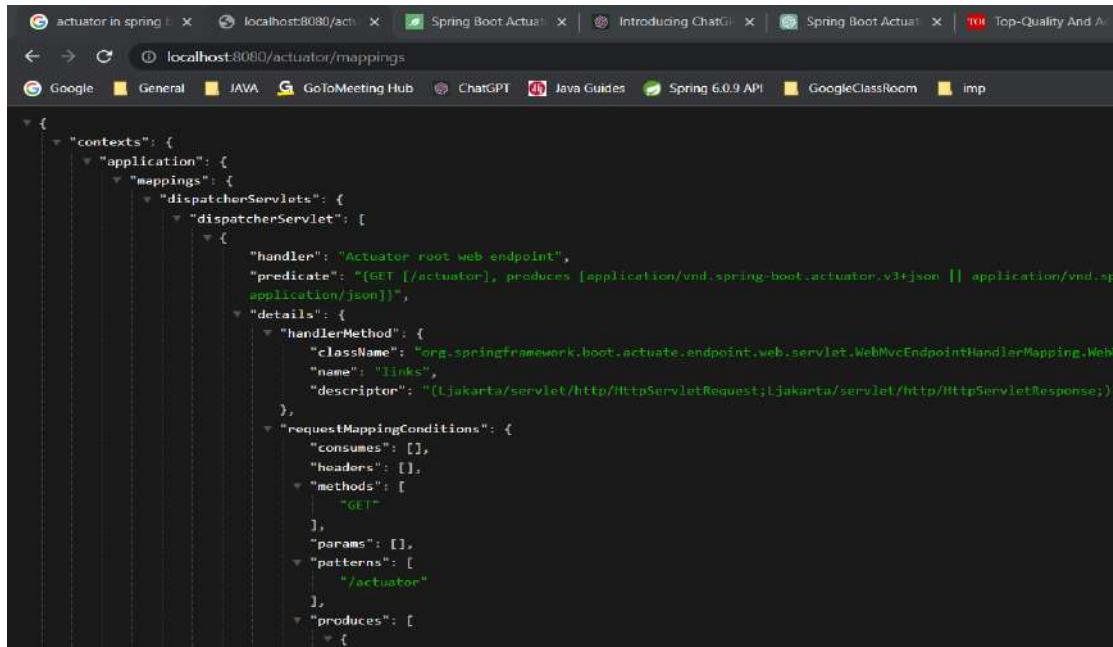
Configure Properties in **application.properties**:



Accessing Health Endpoint : Getting status as UP i.e. Application Started and Deployed Successfully.



Similarly, <http://localhost:8080/actuator/mappings>



```

{
  "contexts": {
    "application": {
      "mappings": {
        "dispatcherServlet": [
          {
            "handler": "Actuator root web endpoint",
            "predicate": "[{GET [/actuator]}, produces [application/vnd.spring-boot.actuator.v3+json || application/vnd.spring-boot.actuator.v2+json]]",
            "details": {
              "handlerMethod": {
                "className": "org.springframework.boot.actuate.endpoint.web.servlet.WebMvcEndpointHandlerMapping.WebMvcEndpointHandler",
                "name": "links",
                "descriptor": "(Ljakarta/servlet/http/HttpServletRequest;Ljakarta/servlet/http/HttpServletResponse;)V"
              },
              "requestMappingConditions": {
                "consumes": [],
                "headers": [],
                "methods": [
                  "GET"
                ],
                "params": [],
                "patterns": [
                  "/actuator"
                ],
                "produces": [
                  {
                    "mediaType": "application/vnd.spring-boot.actuator.v3+json"
                  },
                  {
                    "mediaType": "application/vnd.spring-boot.actuator.v2+json"
                  }
                ]
              }
            }
          }
        ]
      }
    }
  }
}
  
```

Similar to above actuator endpoints, we can enable and access regards to their specifications.

Logging in Spring Boot:

Logging is the activity of keeping a log of events that occur in a computer system or application, such as problems, errors or just information. A message or log entry is recorded for each such event. These log messages can be used to monitor and understand the operation of the system, to debug problems, or during an audit i.e. Logs serves as a valuable tool for developers, system administrators, and support teams to understand the behaviours of the application, diagnose issues, track user activities, and monitor system health. Logging is particularly important in multi-user software, to have a central overview of the operation of the system. Logging in software applications refers to the good practice of capturing and recording important events, workflows, activities, and messages that occur within an application during runtime.

In the simplest case, messages are written to a file, called a log file. Alternatively, the messages may be written to a dedicated logging system or to a log management software, where it is stored in a database or on a different computer system. A good logging infrastructure is necessary for any software project as it not only helps in understanding what's going on with the application but also to traces any unusual incident or error present in the project.

Here are some key points of logging in software applications:

- Log Levels:** Logs are typically categorized into different levels based on their importance and severity. Common log levels include DEBUG, INFO, WARNING, ERROR, and FATAL.

- Each level provides different levels of detail, allowing developers to filter and focus on specific types of information.
2. **Log Messages:** Logs contain messages that describe the workflow, activity, data being logged. These messages should be clear, concise, and meaningful to provide relevant information for troubleshooting and analysis wise.
 3. **Timestamps:** Each log entry should include a timestamp indicating when the event occurred. Timestamps are crucial for understanding the sequence of workflows and identifying correlations between different log entries and layers.
 4. **Contextual Information:** It's important to include contextual information in logs, such as user IDs, session IDs, request IDs, and other relevant metadata. This helps us correlating logs across different components and tracking the flow of activities within the application.
 5. **Log Storage and Retention:** Logs should be stored in a centralized location or log management system in application.
 6. **Log Analysis and Monitoring:** Log analysis tools and techniques, such as log aggregators, search capabilities, and real-time monitoring, can help identify patterns, detect errors, and gain insights into the application's behaviours and performance.
 7. **Security Considerations:** Logs may contain sensitive information, such as user credentials or personal data, so it's crucial to handle them securely. Implement measures like encryption in logs.

By implementing efficient logging practices, developers and system administrators can gain visibility into the application's behaviour, so we can troubleshoot issues efficiently, and improve the overall performance and reliability of the software.

Log levels are used to categorize the severity or importance of log messages in a software application. Each log level represents a different level of detail or criticality, allowing developers and system administrators to filter and focus on specific types of information based on their needs. The following are commonly used log levels are, listed in increasing order of severity:

INFO: The INFO level represents informational messages that highlight important events or workflows in the application's lifecycle. These logs provide useful information about the application's overall state, such as startup and shutdown events, major configuration changes, or significant user interactions. INFO-level logs are generally enabled in production environments to provide essential information without overwhelming the logs.

DEBUG: The DEBUG level provides detailed information that is primarily intended for debugging and troubleshooting during development or testing. It helps developers understand the internal workings of the application, including variable values, control flow, and specific events. DEBUG-level logs are generally used during development and are usually disabled in production to reduce noise and improve performance.

TRACE: The TRACE level provides the most detailed and fine-grained information about the application's execution flow. It is typically used for debugging purposes and is useful when you need to trace the exact sequence of method calls or track specific variables' values. TRACE-level logs are typically disabled in production environments due to their high volume and potential impact on performance.

WARN: The WARN level indicates potential issues or warnings that do not prevent the application from functioning but require attention. It signifies that something unexpected or incorrect has occurred, but the application can continue its operation. WARN-level logs are typically used to capture non-fatal errors, unusual conditions, or situations that might lead to problems if left unaddressed.

ERROR: The ERROR level represents errors or exceptional conditions that indicate a problem in the application's execution. It signifies that something has gone wrong and needs immediate attention in such area. ERROR-level logs captured critical issues/Exceptions that may affect the application's functionality or cause it to behave unexpectedly. These logs often trigger alerts or notifications to the development or support teams to investigate and resolve the problem.

FATAL: The FATAL level represents the most severe log level, indicating a critical error that leads to the application's termination or an unrecoverable/unreachable state. FATAL-level logs are used to capture exceptional situations that render the application unusable or significantly impact its operation. These logs are typically reserved for severe errors that require immediate attention.

By using different log levels appropriately in your application, you can control the amount of information generated by logs and focus on the severity and importance of different events. This enables effective debugging, troubleshooting, and monitoring of your software system.

In order of urgency, ERROR is the most urgent while TRACE is the least urgent log. The default log level in Spring Boot is INFO when no manual configuration is set. By Default Spring Boot using log level as INFO, so we are able to see INFO logs in Server console when we started our application as shown below.

```
2023-06-25T09:01:58.766+05:30      INFO      17340      ---      [restartedMain]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9966 (http) with
context path '/flipkart'
```

```
2023-06-25T09:01:58.783+05:30      INFO      17340      ---      [restartedMain]
com.flipkart.FlipkartApplication Started FlipkartApplication in 8.308 seconds (process running
for 9.187)
```

Spring Boot makes use of Apache Commons' Logging for its system logs by default. Additionally, by default you can use any of the logging frameworks under the SLF4J API in SpringBoot directly without any external configuration.

Logging in Spring Boot, using SLF4J, as well as log levels:

To enable logging in Spring, import **Logger** and **LoggerFactory** from the org.slf4j API library. Writing Logs in Our own implemented Classes. As Part of **LoggerFactory** we should pass current class Class object to print logs specifically. Usually we will follow below style to enable logs of that class i.e. in side getLogger() method we should pass Class object of same class.

Ex : `Logger = LoggerFactory.getLogger(UsersController.class);`

UsersController.java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UsersController {

    Logger logger = LoggerFactory.getLogger(UsersController.class);

    @GetMapping("/logs")
    public void checkLogLevels() {
        logger.error("Error message");
        logger.warn("Warning message");
        logger.info("Info message");
        logger.debug("Debug message");
        logger.trace("Trace message");
    }
}
```

When we called above endpoint, we can see by default which logs are printed out of all log types/levels.



```
:30 INFO 17340 --- [nio-9966-exec-2] o.a.c.c.C.[.localhost].[/flipkart] : Initializing Spring application
:30 INFO 17340 --- [nio-9966-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
:30 INFO 17340 --- [nio-9966-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 0 ms
:30 ERROR 17340 --- [nio-9966-exec-2] c.f.user.controller.UsersController : Error message
:30 WARN 17340 --- [nio-9966-exec-2] c.f.user.controller.UsersController : Warning message
:30 INFO 17340 --- [nio-9966-exec-2] c.f.user.controller.UsersController : Info message
```

So By default INFO, WARN, ERROR messages are printed in console. If we want to see other level logs then we should enable or configure in our properties file.

- Another Example for Adding Loggers in other classes i.e. create logger by passing class Object of Service class.

```
public class UserService {
```

`Logger logger = LoggerFactory.getLogger(UserService.class);`

```
public List<Users> getUsersByCityName(String city) {  
    logger.info("City Name is :" + city);  
    List<Users> users = getUsersByCityName(city);  
    logger.info("Response is: " + users);  
    return users;  
}  
}
```

Configuring Log Levels in Spring Boot:

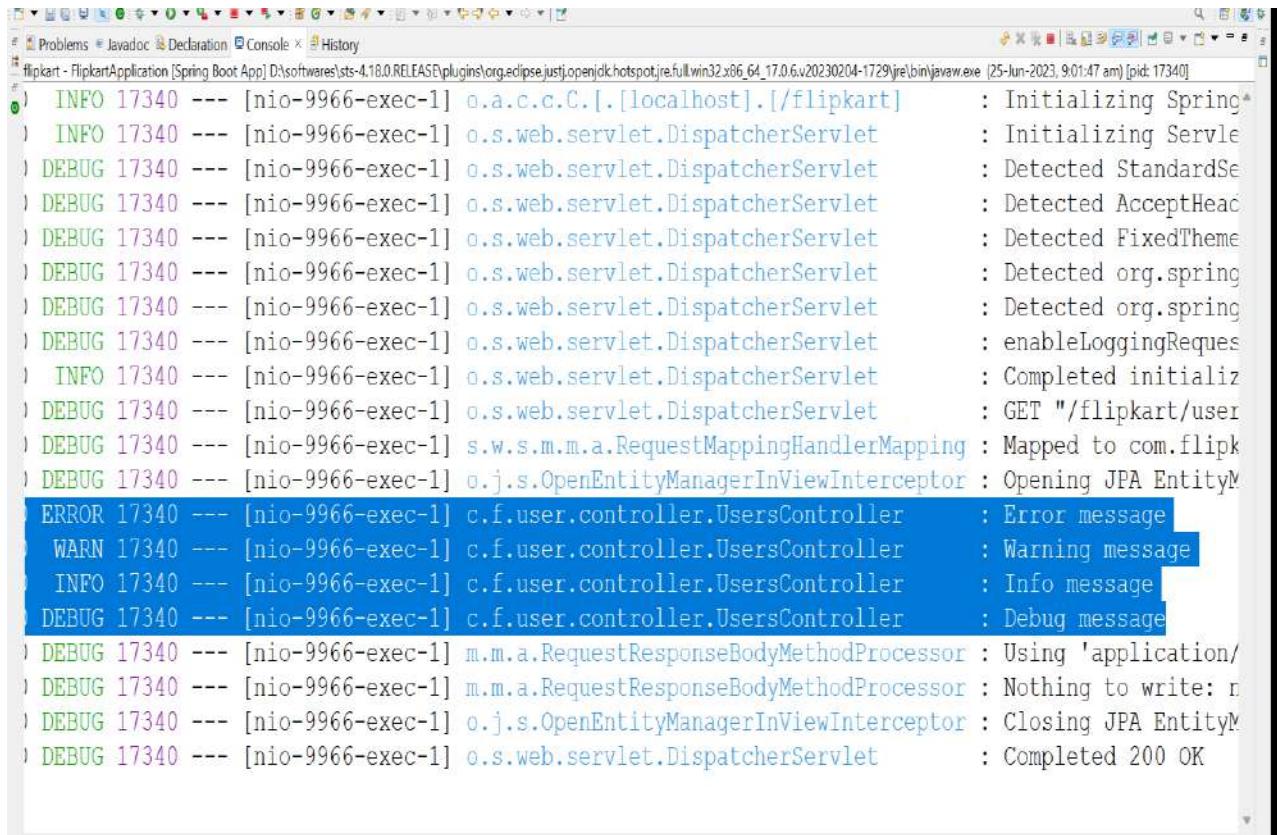
Log levels can be set in the Spring environment by setting its configurations in **application.properties** file. The format to set the log level configuration is **logging.level.[classpath] = [level]**. The classpath is specified because different components of the application can be configured with different log levels, which is especially useful for code isolation and debugging. To specify a log level for all classes that don't have their own log level settings, the root logger can be set using with property **logging.level.root**.

application.properties

```
#Setting Log level For All Spring Modules  
logging.level.org.springframework=debug  
#Setting Log level For our Project Classes  
logging.level.com.flipkart=DEBUG
```

Now Start our Application and try to execute above existing endpoint, Observer the logs. All Spring Framework modules will print log messages including DEBUG level. As well as we set LOG level for our own application classes level by providing below Property.

```
logging.level.com.flipkart=DEBUG
```



```

flipkart - FlipkartApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\java.exe [25-Jun-2023, 9:01:47 am] [pid: 17340]
: Initializing Spring
: Initializing Servle
: Detected StandardSe
: Detected AcceptHead
: Detected FixedTheme
: Detected org.spring
: Detected org.spring
: enableLoggingReques
: Completed initializ
: GET "/flipkart/user
: Mapped to com.flipk
: Opening JPA EntityM
: Error message
: Warning message
: Info message
: Debug message
: Using 'application/
: Nothing to write: n
: Closing JPA EntityM
: Completed 200 OK

```

Log Groups:

Log groups is a useful way to set logger configurations to a group of classes with different classpaths/packages. An example is if you want to set log levels to DEBUG in one go for different packages in our application. This is possible using the configuration of **logging.group.[groupName]**:

```

# Define log group
logging.group.test=com.test, com.user, com.product

# Set log level to above log group
logging.level.test=DEBUG

```

With this approach, we no need to individually set the log level of all related components all the time.

Conclusion of Logging In Application:

Knowing about the different log levels is important especially in situations like debugging in production environments. Let's say a major bug has been exposed in production, and the current logs do not have enough information to diagnose the root cause of the problem. By changing the log level to DEBUG or TRACE, the logs will show much-needed information to pinpoint crucial details that may lead towards the fix of issue.

In Spring, the log level configurations can be set in the **application.properties** file which is processed during runtime. Spring supports 5 default log levels, ERROR, WARN, INFO, DEBUG, and TRACE, with INFO being the default log level configuration.

@DilipItAcademy

Spring Boot Security Module

Spring Security:

Spring Security is a powerful and highly customizable authentication and access-control framework. Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements. It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application. The main goal of Spring Security is to make it easy to add security features to your applications. It follows a modular design, allowing you to choose and configure various components according to your specific requirements. Some of the key features of Spring Security include:

Authentication: Spring Security supports multiple authentication mechanisms, such as form-based, HTTP Basic, HTTP Digest, and more. It integrates seamlessly with various authentication providers, including in-memory, JDBC, LDAP, and OAuth.

Authorization: Spring Security enables fine-grained authorization control based on roles, permissions, and other attributes. It provides declarative and programmatic approaches for securing application resources, such as URLs, methods, and domain objects.

Session Management: Spring Security offers session management capabilities, including session fixation protection, session concurrency control, and session timeout handling. It allows you to configure session-related properties and customize session management behaviour.

Security Filters: Spring Security leverages servlet filters to intercept and process incoming requests. It includes a set of predefined filters for common security tasks, such as authentication, authorization, and request/response manipulation. You can easily configure and extend these filters to meet your specific needs.

Integration with Spring Framework: Spring Security seamlessly integrates with the Spring ecosystem. It can leverage dependency injection and aspect-oriented programming features provided by the Spring Framework to enhance security functionality.

Customization and Extension: Spring Security is highly customizable, allowing you to override default configurations, implement custom authentication/authorization logic, and integrate with third-party libraries or existing security infrastructure.

Overall, Spring Security simplifies the process of implementing robust security features in Java applications. It provides a flexible and modular framework that addresses common security concerns and allows developers to focus on building secure applications.

This module targets two major areas of application are **authentication** and **authorization**.

What is Authentication?

Authentication in Spring refers to the process of verifying the identity of a user or client accessing a system or application. It is a crucial aspect of building secure applications to ensure that only authorized individuals can access protected resources.

In the context of Spring Security, authentication involves validating the credentials provided by the user and establishing their identity. Spring Security offers various authentication mechanisms and supports integration with different authentication providers.

Here's a high-level overview of how authentication works in Spring Security:

User provides credentials: The user typically provides credentials, such as a username and password, in order to authenticate themselves.

Authentication request: The application receives the user's credentials and creates an authentication request object.

Authentication manager: The authentication request is passed to the authentication manager, which is responsible for validating the credentials and performing the authentication process.

Authentication provider: The authentication manager delegates the actual authentication process to one or more authentication providers. An authentication provider is responsible for verifying the user's credentials against a specific authentication mechanism, such as a user database, LDAP server, or OAuth provider.

Authentication result: The authentication provider returns an authentication result, indicating whether the user's credentials were successfully authenticated or not. If successful, the result typically contains the authenticated user details, such as the username and granted authorities.

Security context: If the authentication is successful, Spring Security establishes a security context for the authenticated user. The security context holds the user's authentication details and is associated with the current thread.

Access control: With the user authenticated, Spring Security can enforce access control policies based on the user's granted authorities or other attributes. This allows the application to restrict access to certain resources or operations based on the user's role or permissions.

Spring Security provides several authentication mechanisms out-of-the-box, including form-based authentication, HTTP Basic/Digest authentication, JWT token, OAuth-based authentication. Spring also supports customization and extension, allowing you to integrate with your own authentication providers or implement custom authentication logic to meet your specific requirements.

By integrating Spring Security's authentication capabilities into your application, you can ensure that only authenticated and authorized users have access to your protected resources, helping to safeguard your application against unauthorized access.

What is Authorization?

Authorization, also known as access control, is the process of determining what actions or resources a user or client is allowed to access within a system or application. It involves enforcing permissions and restrictions based on the user's identity, role, or other attributes. Once a user is authenticated, authorization is used to control their access to different parts of the application and its resources.

Here are the key concepts related to authorization in Spring Security:

Roles: Roles represent a set of permissions or privileges granted to a user. They define the user's high-level responsibilities or functional areas within the application. For example, an application may have roles such as "admin," "user," or "manager."

Permissions: Permissions are specific actions or operations that a user is allowed to perform. They define the granular level of access control within the application. For example, a user with the "admin" role may have permissions to create, update, and delete resources, while a user with the "user" role may only have read permissions.

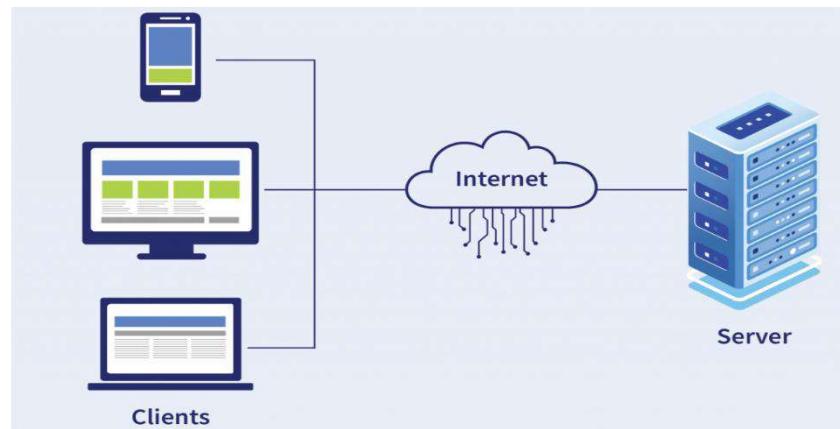
Security Interceptors: Spring Security uses security interceptors to enforce authorization rules. These interceptors are responsible for intercepting requests and checking whether the user has the required permissions to access the requested resource. They can be configured to protect URLs, methods, or other parts of the application.

Role-Based Access Control (RBAC): RBAC is a common authorization model in which access control is based on roles. Users are assigned roles, and permissions are associated with those roles. Spring Security supports RBAC by allowing you to define roles and assign them to users.

By implementing authorization in your Spring application using Spring Security, you can ensure that users have appropriate access privileges based on their roles and permissions. This helps protect sensitive resources and data from unauthorized access and maintain the overall security and integrity of your application.

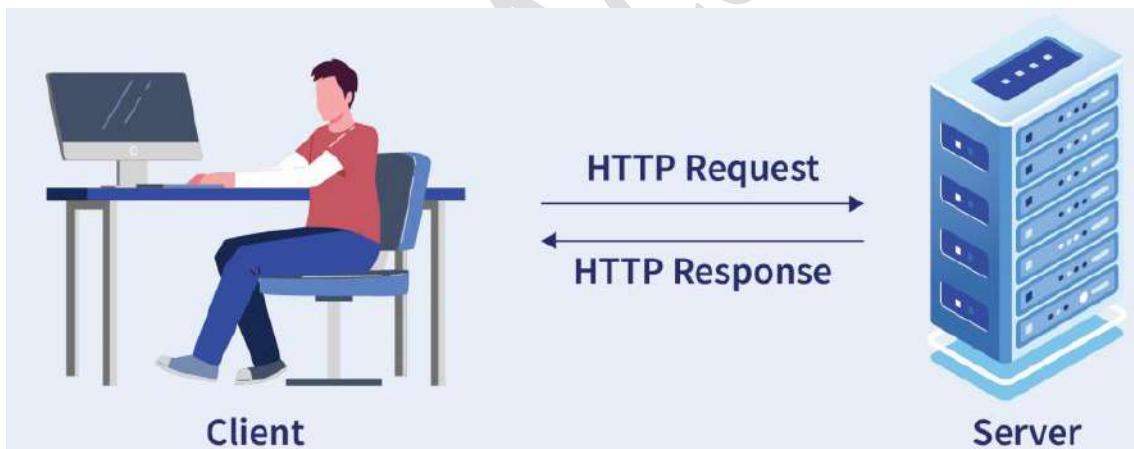
Stateless and Stateful Protocols:

In the context of the protocol, "stateless" and "stateful" refer to different approaches in handling client-server interactions and maintaining session information. Let's explore each concept:



Stateless:

In a stateless protocol, such as HTTP, the server does not maintain any information about the client's previous interactions or session state. Each request from the client to the server is considered independent and self-contained. The server treats each request as if it is the first request from the client.



Stateless protocols have the following characteristics:

- No client session information is stored on the server.
- Each request from the client must contain all the necessary information for the server to process the request.
- The server responds to each request independently, without relying on any prior request context.

HTTP is primarily designed as a stateless protocol. When a client makes an HTTP request, the server processes the request and sends back a response. However, the server does not maintain any information about the client after the response is sent. This approach simplifies the server's implementation and scalability but presents challenges for handling user sessions and maintaining continuity between multiple requests.

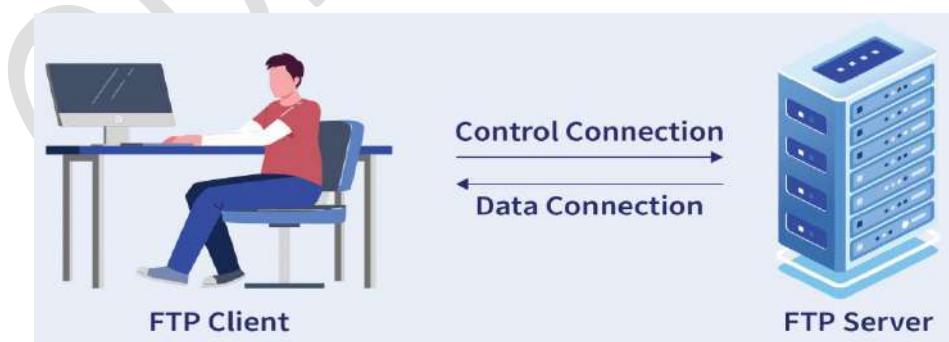
Stateless Protocol (Technical)



Stateful:

In contrast, a stateful protocol maintains information about the client's interactions and session state between requests. The server stores client-specific information and uses it to provide personalized responses and maintain continuity across multiple requests.

However, the major feature of stateful is that it maintains the state of all its sessions, be it an authentication session, or a client's request for information. Stateful are those that may be used repeatedly, such as online banking or email. They're carried out in the context of prior transactions in which the states are stored, and what happened in previous transactions may have an impact on the current transaction. Because of this, stateful apps use the same servers every time they perform a user request. An example of stateful is FTP (File Transfer Protocol) i.e. File transferring between servers. For FTP session, which often includes many data transfers, the client establishes a Control Connection. After this, the data transfer takes place.



Stateful protocols have the following characteristics:

- The server keeps track of client session information, typically using a session identifier

- The session information is stored on the server.
- The server uses the session information to maintain context between requests and responses.

While HTTP itself is stateless, developers often implement mechanisms to introduce statefulness. For example, web applications often use cookies or tokens to maintain session state. These cookies or tokens contain session identifiers that the server can use to retrieve or store client-specific data.

By introducing statefulness, web applications can provide a more personalized and interactive experience for users. However, it adds complexity to the server-side implementation and may require additional considerations for scalability and session management.

It's important to note that even when stateful mechanisms are introduced, each individual HTTP request-response cycle is still stateless in nature. The statefulness is achieved by maintaining session information outside the core HTTP protocol, typically through additional mechanisms like cookies, tokens, or server-side session stores.

Q&A:

What is the difference between stateful and stateless?

The major difference between stateful and stateless is whether or not they store data regarding their sessions, and how they respond to requests. Stateful services keep track of sessions or transactions and respond to the same inputs in different ways depending on their history. Clients maintain sessions for stateless services, which are focused on activities that manipulate resources rather than the state.

Is stateless better than stateful?

In most cases, stateless is a better option when compared with stateful. However, in the end, it all comes down to your requirements. If you only require information in a transient, rapid, and temporary manner, stateless is the way to go. Stateful, on the other hand, might be the way to go if your app requires more memory of what happens from one session to the next.

Is HTTP stateful or stateless?

HTTP is stateless because it doesn't keep track of any state information. In HTTP, each order or request is carried out in its own right, with no awareness of the demands that came before it.

Is REST API stateless or stateful?

REST APIs are stateless because, rather than relying on the server remembering previous requests, REST applications require each request to contain all of the information necessary for the server to understand it. Storing session state on the server violates the REST architecture's stateless requirement. As a result, the client must handle the complete session state.

Security Implementation:

Stateless Security and **Stateful Security** are two approaches to handling security in systems, particularly in the context of web applications. Let's explore the differences between these two approaches:

Stateless Security:

Stateless security refers to a security approach where the server does not maintain any session state or client-specific information between requests. It is often associated with stateless protocols, such as HTTP, where each request is independent and self-contained. Stateless security is designed to provide security measures without relying on server-side session state.

In the context of web applications and APIs, stateless security is commonly implemented using mechanisms such as JSON Web Tokens (JWT) or OAuth 2.0 authentication scheme. These mechanisms allow authentication and authorization to be performed without the need for server-side session storage.

Here are the key characteristics and advantages of stateless security:

- **No server-side session storage:** With stateless security, the server does not need to maintain any session-specific information for each client. This eliminates the need for server-side session storage, reducing the overall complexity and resource requirements on the server side.
- **Scalability:** Stateless security simplifies server-side scaling as there is no need to replicate session state across multiple instances of application deployed to multiple servers. Each server can process any request independently, which makes it easier to distribute the load and scale horizontally.
- **Decentralized authentication:** Stateless security allows for decentralized authentication, where the client sends authentication credentials (such as a JWT token) with each request. The server can then validate the token's authenticity and extract necessary information to authorize the request.
- **Improved performance:** Without the need to perform expensive operations like session lookups or database queries for session data, stateless security can lead to improved performance. Each request carries the necessary authentication and authorization information, reducing the need for additional server-side operations.

It's important to note that while stateless security simplifies server-side architecture and offers advantages in terms of scalability and performance, it also places additional responsibilities on the client-side. The client must securely store and transmit the authentication token and include it in each request.

Stateless security is widely adopted in modern web application development, especially in distributed systems and microservices architectures, where scalability, performance, and decentralized authentication are important considerations.

In stateless security:

- **Authentication:** The client provides credentials (e.g., username and password or a token) with each request to prove its identity. The server verifies the credentials and grants access based on the provided information.
- **Authorization:** The server evaluates each request independently, checking if the user has the necessary permissions to access the requested resource.

Cons of Stateless Security:

- **Increased overhead:** The client needs to send authentication information with each request, which can increase network overhead, especially when the authentication mechanism involves expensive cryptographic operations.

Stateful Security:

Stateful security involves maintaining session state on the server. Once the client is authenticated, the server stores session information and associates it with the client. The server refers to the session state to validate subsequent requests and provide appropriate authorization.

In stateful security:

- **Authentication:** The client typically authenticates itself once using its credentials (e.g., username and password or token). After successful authentication, the server generates a session identifier or token and stores it on the server.

Session Management: The server maintains session-specific data, such as user roles, permissions, and other contextual information. The session state is referenced for subsequent requests to determine the user's authorization level.

Pros of Stateful Security:

- **Enhanced session management:** Session state allows the server to maintain user context, which can be beneficial for handling complex interactions and personalized experiences.
- **Reduced overhead:** Since the client doesn't need to send authentication information with each request, there is a reduction in network overhead.

Cons of Stateful Security:

- **Scalability challenges:** The server needs to manage session state, which can be a scalability bottleneck. Sharing session state across multiple servers or implementing session replication techniques becomes necessary.
- **Complexity:** Implementing stateful security requires additional effort to manage session state and ensure consistency across requests.

The choice between stateless security and stateful security depends on various factors, including the specific requirements of the application, performance considerations, and the desired level of session management and personalization. Stateless security is often preferred

for its simplicity and scalability advantages, while stateful security is suitable for scenarios requiring more advanced session management capabilities.

JWT Authentication & Authorization:

JWTs or JSON Web Tokens are most commonly used to identify an authenticated user. They are issued by an authentication server and are consumed by the client-server (to secure its APIs).

What is a JWT?

JSON Web Token is an open industry standard used to share information between two entities, usually a client (like your app's frontend) and a server (your app's backend). They contain JSON objects which have the information that needs to be shared. Each JWT is also signed using cryptography (hashing) to ensure that the JSON contents (also known as JWT claims) cannot be altered by the client or a malicious party.

A token is a string that contains some information that can be verified securely. It could be a random set of alphanumeric characters which point to an ID in the database, or it could be an encoded JSON that can be self-verified by the client (known as JWTs).

Structure of a JWT:

A JWT contains three parts:

- **Header:** Consists of two parts:
 - The signing algorithm that's being used.
 - The type of token, which, in this case, is mostly "JWT".
- **Payload:** The payload contains the claims or the JSON object of clients.
- **Signature:** A string that is generated via a cryptographic algorithm that can be used to verify the integrity of the JSON payload.

In general, whenever we generated token with JWT, token generated in the format of `<header>.<payload>.<signature>` in side JWT.

Example:

`eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpEBnbWFpbC5jb20iLCJleHAiOiE2ODk1MjI5OTcsImlhdCI6MTY4OTUyMjY5N30.bjFnipeNqiZ5dyrXZHk0qTPciChw0Z0eNoX5fu5uAmj6SE9mLIGD4L1_3QeGfxjZqvv8K1Je2pmTseT4g8ZSIA`

Following image showing details of Encoded Token.

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpCE
BnbWFpbC5jb20iLCJleHAiOjE2ODk1MjI50TcsI
mlhdCI6MTY4OTUyMjY5N30.bjFnipeNqiZ5dyrX
ZHk0qTPciChw0Z0eNoX5fu5uAmj6SE9mLIGD4L1
_3QeGfXjZqvv8K1Je2pmTseT4g8ZSIA
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE	
<pre>{ "alg": "HS512" }</pre>	
PAYLOAD: DATA	
<pre>{ "sub": "dilip@gmail.com", "exp": 1689522997, "iat": 1689522697 }</pre>	
VERIFY SIGNATURE	
<pre>HMACSHA512(base64UrlEncode(header) + ".", base64UrlEncode(payload), <input type="text" value="your-256-bit-secret"/>) <input type="checkbox"/> secret base64 encoded</pre>	

JWT Token Creation and Validation:

We are using Java JWT API for creation and validation of Tokens.

- Create A Maven Project
- Add Below both dependencies, required for java JWT API.

```
<dependencies>
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
  </dependency>
  <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
  </dependency>
</dependencies>
```

- Now Write a Program for creating, claiming and validating JWT tokens : **JSONWebToken.java**

```
import java.util.Date;
import java.util.concurrent.TimeUnit;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtBuilder;
import io.jsonwebtoken.SignatureAlgorithm;
```

```

//JWT Token Generation
public class JSONWebToken {

    static String key = "ZOMATO";

    public static void main(String ar[]) {

        // Creating/Producing Tokens
        String token = Jwts.builder()
            .setSubject("dilipsingh1306@gmail.co") // User ID
            .setIssuer("ZOMATOCOMPANY")
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + TimeUnit.MINUTES.toMillis(1)))
            .signWith(SignatureAlgorithm.HS256, key.getBytes())
            .compact();

        System.out.println(token);

        // Reading/Parsing Token Details
        claimToken(token);

        //Checking Expired or not.
        boolean isExpired = isTokenExpired(token);
        System.out.println("Is It Expired? " + isExpired);

    }

    public static void claimToken(String token) {

        // Claims : Reading details from generated token by passing secret
        Claims claim = Jwts.parser().setSigningKey(key.getBytes()).parse(token).getBody();

        Date createdDateTime = claim.getIssuedAt();
        Date expDateTime = claim.getExpiration();
        String issuer = claim.getIssuer();
        String subject = claim.getSubject();

        System.out.println("Token Provider : " + issuer);
        System.out.println("Token Generated for User : " + subject);
        System.out.println("Token Created Time : " + createdDateTime);
        System.out.println("Token Expired Time : " + expDateTime);
    }

    public static boolean isTokenExpired(String token) {

```

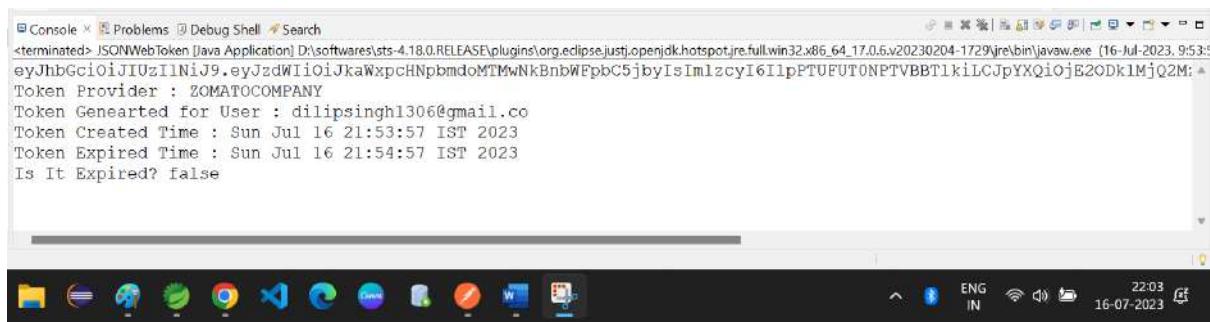
```

Claims           claim           =           (Claims)
Jwts.parser().setSigningKey(key.getBytes()).parse(token).getBody();
    Date expDateTime = claim.getExpiration();
    return expDateTime.before(new Date(System.currentTimeMillis()));
}

}

```

Output:



```

Console Problems Debug Shell Search
<terminated> JSONWebToken [Java Application] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justmydk.hotspot\re\full\win32\x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe [16-Jul-2023, 9:53:53]
eyJhbGciOiJIUzI1NiJ9.eyJdWIiOiJkaWxpchNpbmdoMTMwNkBnbWFpbC5jbyIsImIzcyI6IlpPTUUFUTONPTVBBT1k1LCJpYXQiOjE2ODk1MjQ2M
Token Provider : ZOMATOCOMPANY
Token Generated for User : dilipsingh1306@gmail.co
Token Created Time : Sun Jul 16 21:53:57 IST 2023
Token Expired Time : Sun Jul 16 21:54:57 IST 2023
Is It Expired? false

```

The above program written for understanding of how tokens are generated and how we are parsing/claiming details from JSON token.

Now we will re-use above logic as part of SpringBoot Security Implementation. Let's start SpringBoot Security with JWT.

GitHub Repository Link : <https://github.com/dilipsingh1306/javaJWTToken>

SpringBoot Security + (JSON WEB TOKEN):

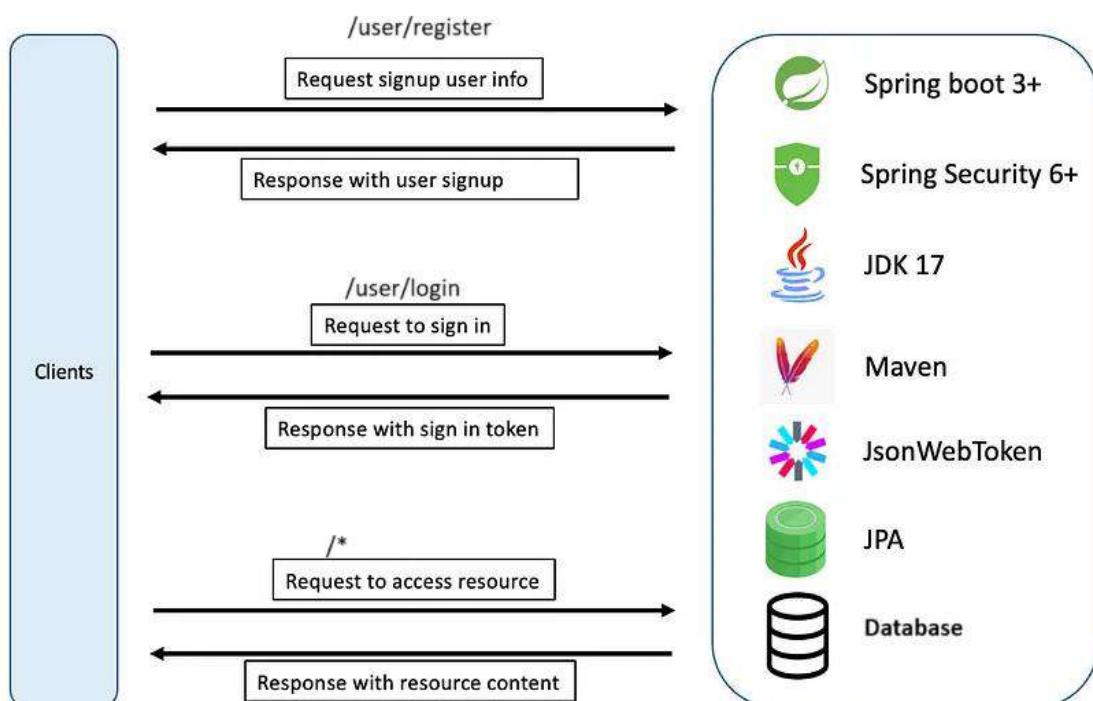
The application we are going to develop will handle user authentication and authorization with JWT's for securing an exposed REST API Services.

NOTE: We are using SpringBoot Version 3.1.1 in our training. SpringBoot 3.X Internally uses or implemented with Spring Framework Version 6. Spring 6 Security API Updated with new Classes and Methods comparing with Spring 5.



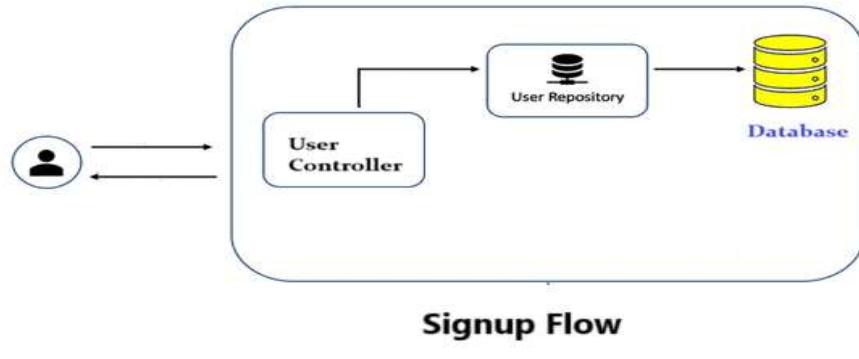
Application Architecture: Scenarios:

- User makes a request to the service, for create an account.
- User submits login request to the service to authenticate their account.
- An authenticated user sends a request to access resources/services.



Sign Up Process:

Step 1: Implement Logic for User Sign Up Process. The Sign-up process is very simple. Please understand following Signup Flow Diagram.



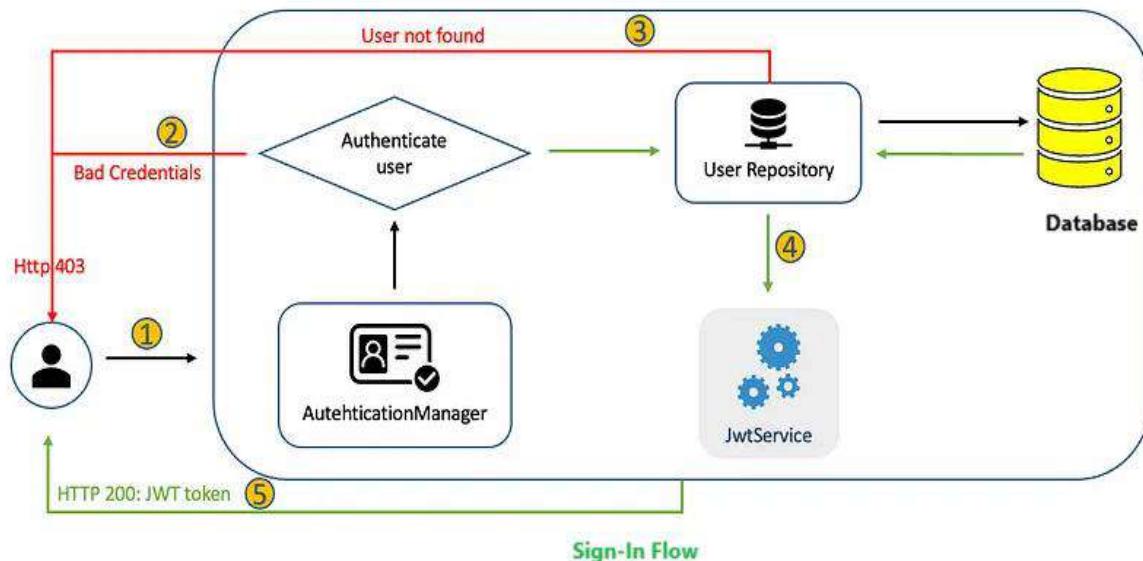
- The process starts when a user submits a request to our service. A user object is then generated from the request data, and we should encode password before storing inside Database. The password being encoded by using Spring provided Password Encoders.

It is important that we must inform Spring about the specific password encoder utilized in the application, In this case, we are using **BCryptPasswordEncoder**. This information is necessary for Spring to properly authenticate users by decoding their passwords. We will have more information about Password Encoder further.

In our application requirement is, For User Sign-up provide details of email ID, Password, Name and Mobile Number. Email ID and Password are inputs for Sign-In operation.

Sign-In Activity:

Internal Process and Logic Implementation:



1. The process begins when a user sends a sign-in request to the Service. An Authentication object called **UsernamePasswordAuthenticationToken** is then generated, using the provided username and password.
2. The **AuthenticationManager** is responsible for authenticating the Authentication object, handling all necessary tasks. If the **username or password is incorrect**, an exception is thrown as Bad Credentials, and a response with HTTP Status 403 is returned to the user.
3. **After successful authentication**, Once we have the user information, we call the JwtService to generate the JWT for that User Id.
4. The JWT is then encapsulated in a JSON response and returned to the user.

Two new concepts are introduced here, and I'll provide a brief explanation for each.

UsernamePasswordAuthenticationToken: A type of Authentication object which can be created from a *username* and *password* that are submitted.

AuthenticationManager: Processes authentication object and will do all authentication jobs for us.

Resource/Services Accessibility:

When User tries to access any other resources/REST services of application, then we will apply security rules and after success authentication and authorization of user, we will allow to access/execute services. If Authentication failed, then we will send Specific Error Response codes usually 403 Forbidden.

Internally how we are going to enabling Security with JSON web token:

This process is secured by Spring Security, Let's define its flow as follows.

1. When the Client sends a request to the Service, The request is first intercepted by **JWTTokenFilter**, which is a custom filter integrated into the **SecurityFilterChain**.
2. As the API is secured, if the JWT is missing as part of Request Body header, a response with HTTP Status 403 is sent to the client.
3. When an existing JWT is received, **JWTTokenFilter** is called to extract the user ID from the JWT. If the user ID cannot be extracted, a response with HTTP Status 403 is sent to the user.
4. If the user ID can be extracted, it will be used to query the user's authentication and authorization information via **UserDetailsService** of Spring Security.
5. If the user's authentication and authorization information does not exist in the database, a response with HTTP Status 403 is sent to the user.

6. If the JWT is expired, a response with HTTP Status 403 is sent to the user.
7. After successful authentication, the user's details are encapsulated in a **UsernamePasswordAuthenticationToken** object and stored in the **SecurityContextHolder**.
8. The Spring Security Authorization process is automatically invoked.
9. The request is dispatched to the controller, and a successful JSON response is returned to the user.

This process is a little bit tricky because involving some new concepts. Let's have some information about all new items.

SecurityFilterChain: In Spring Security, the **SecurityFilterChain** is responsible for managing a chain of security filters that process and enforce security rules for incoming requests in order to decide whether rules applies to that request or not. It plays a crucial role in handling authentication, authorization, and other security-related tasks within a Spring Security-enabled application. The **SecurityFilterChain** interface represents a single filter chain configuration. If we want, we can define multiple **SecurityFilterChain** instances to handle different sets of URLs or request patterns, allowing over security rules based on specific requirements.

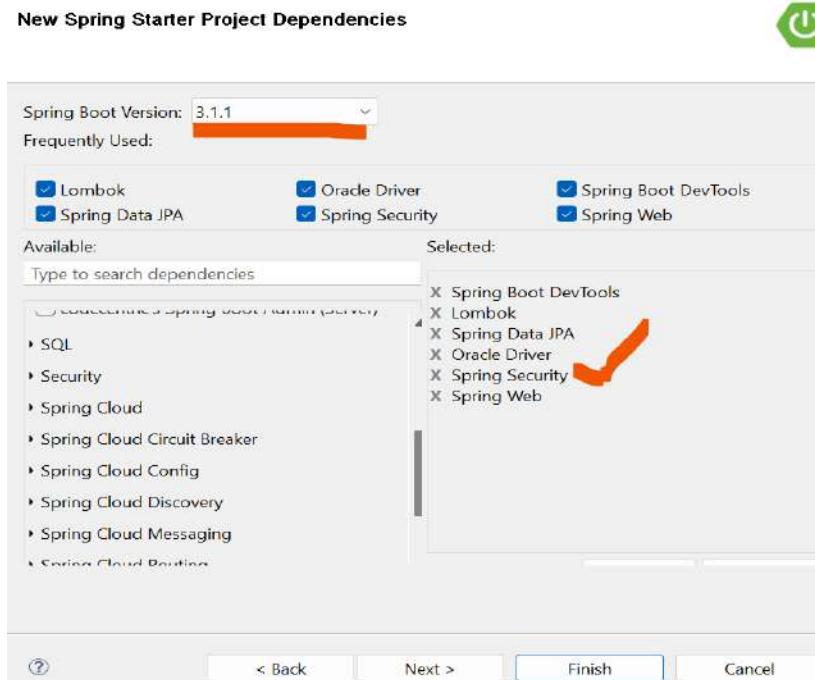
SecurityContextHolder: The **SecurityContextHolder** class is responsible for managing the **SecurityContext** object, which holds the security-related information. The **SecurityContext** contains the Authentication object representing the current user's authentication details, such as their principal (typically a user object) and their granted authorities. You can access the **SecurityContext** using the static methods of **SecurityContextHolder**.

UserDetailsService: In Spring Security, the **UserDetailsService** interface is used to retrieve user-related data during the authentication process. It provides a mechanism for Spring Security to load user details (such as username, password, and authorities) from database or any other data source. The **UserDetailsService** interface defines a single method:

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

The **loadUserByUsername()** method is responsible for retrieving the user details for a given username. It returns an implementation of the **UserDetails** interface, which represents the user's security-related data.

Security Logic Implementation: Now Create Spring Boot Application with Security API:



- After Successful Project creation, we should add JWT librarys dependecis inside Maven pom.xml file because by default SpringBoot not providing support of JWT.

```

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>

```

Now Configure Application Port Number, Context-Path along with Database Details inside **application.properties** file.

```

#App Details
server.port=8877
server.servlet.context-path=/zomato

#DB Details
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orc
1
spring.datasource.username=c##dilip
spring.datasource.password=dilip

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

```

Now create Controller, Service and Repository Layers.

Note: In this application, we are using Lombok library so we are using Lombok annotations instead of writing setters, getters and constructors. Please add import statements for used annotations.

- Defining Request and Response Classes for Signup nad SingIn user services.

UserRegisterRequest.java

```

@getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@ToString
public class UserRegisterRequest {
    private String emailId;
    private String password;
    private String name;
    private long mobile;
}

```

UserRegisterResponse.java

```

@Data
@getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class UserRegisterResponse {
    private String emailId;
    private String message;
}

```

UserLoginRequest.java

```

@getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@ToString
public class UserLoginRequest {
    private String emailId;
    private String password;
}

```

UserLoginResponse.java

```

import lombok.AllArgsConstructor;
import lombok.Data;

```

```

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Data
@Getters
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class UserLoginResponse {
    private String token;
    private String emailId;
}

```

➤ Now Add Signup and Sign-in Services in Controller class with Authentication Layer.

```

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.dilip.jwt.token.JWTTokenHelper;
import com.dilip.user.request.UserLoginRequest;
import com.dilip.user.request.UserRegisterRequest;
import com.dilip.user.response.UserLoginResponse;
import com.dilip.user.response.UserRegisterResponse;
import com.dilip.user.service.UsersRegisterService;

@RestController
@RequestMapping("/user")
public class UserController {

    Logger logger = LoggerFactory.getLogger(UserController.class);

    @Autowired
    UsersRegisterService usersRegisterService;

    @Autowired

```

```
JWTTokenHelper jwtTokenHelper;

@Autowired
AuthenticationManager authenticationManager;

@Autowired
BCryptPasswordEncoder passwordEncoder;

@GetMapping("/hello")
public String sayHello() {
    return "Welcome to Security";
}

// User Sign-Up Operation
@PostMapping("/register")
public ResponseEntity<UserRegisterResponse> createUserAccount(@RequestBody
UserRegisterRequest request) {

    request.setPassword(passwordEncoder.encode(request.getPassword()));

    String result = usersRegisterService.createUserAccount(request);
    return ResponseEntity.ok(new UserRegisterResponse(request.getEmailId(),
result));
}

// 2. Login User
@PostMapping("/login")
public ResponseEntity<UserLoginResponse> loginUser(@RequestBody
UserLoginRequest request) {

    // logic for authentication of user login time
    this.doAuthenticate(request.getEmailId(), request.getPassword());

    String token = this.jwtTokenHelper.generateToken(request.getEmailId());
    return ResponseEntity.ok(new UserLoginResponse(token, request.getEmailId()));

}
private void doAuthenticate(String emailId, String password) {

    logger.info("Authentication of User Credentials");

    UsernamePasswordAuthenticationToken authentication = new
    UsernamePasswordAuthenticationToken(emailId, password);
    try {
        authenticationManager.authenticate(authentication);
    } catch (BadCredentialsException e) {
        throw new RuntimeException("Invalid UserName and Password");
    }
}
```

```

    }
}

}

```

From the above controller class, we defined login service and as part of that we are enabling authentication with **AuthenticationManager** as part of Security Module by passing **UsernamePasswordAuthenticationToken** with requester user Id and password.

➤ Now create Service Layer for User Registration Logic: **UsersRegisterService.java**

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.dilip.user.entity.UserRegister;
import com.dilip.user.repository.UsersRegisterRepository;
import com.dilip.user.request.UserRegisterRequest;

@Service
public class UsersRegisterService {

    @Autowired
    UsersRegisterRepository usersRegisterRepository;

    public String createUserAccount(UserRegisterRequest request) {

        UserRegister register = UserRegister.builder() // Initilizing based on builder
            .emailId(request.getEmailId())
            .password(request.getPassword())
            .name(request.getName())
            .mobile(request.getMobile())
            .build(); // Create instance with provided values

        usersRegisterRepository.save(register);
        return "Registered Successfully";
    }
}

```

Logic Implementation:

- Create Custom Entity Class by implanting **UserDetails** Interface of Spring Security API. So that we can directly Store Repository Data of User Credentials and roles in side **UserDetails**. Now same will be utilized by Spring Authentication and Authorization Modules internally.

```

import java.util.Collection;
import org.springframework.security.core.GrantedAuthority;

```

```
import org.springframework.security.core.userdetails.UserDetails;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Table(name = "user_register")
@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserRegister implements UserDetails {

    @Id
    private String emailId;
    private String password;
    private String name;
    private long mobile;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return null;
    }

    @Override
    public String getUsername() {
        return emailId;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
```

```

        return true;
    }

}

```

- Add a method in Repository, to retrieve User details based on user ID i.e. email ID in our case. This method will be used as part of Authentication Service Implementation in following steps.

```

import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.dilip.user.entity.UserRegister;

@Repository
public interface UsersRegisterRepository extends JpaRepository<UserRegister, String>{

    Optional<UserRegister> findByEmailId(String emailId);
}

```

- Now Define Authentication UserService i.e. Implementation **UserDetailsService** interface from Spring Security to retrieve User Details from database for internal use by Authentication Security Filters.

```

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.dilip.user.entity.UserRegister;
import com.dilip.user.repository.UsersRegisterRepository;

@Service
public class UserAuthenticationServiceImpl implements UserDetailsService{

    Logger logger = LoggerFactory.getLogger(UserAuthenticationServiceImpl.class);

    @Autowired
    UsersRegisterRepository repository;

    @Override
    public UserDetails loadUserByUsername(String emailId) throws UsernameNotFoundException {
        logger.info("Fetching UserDetails");
        return repository.findByEmailId(emailId);
    }
}

```

```

UserRegister user = repository.findByEmailId(emailId).orElseThrow(() -> new
UsernameNotFoundException("Invalid User Name"));

return user;
}
}

```

- Create **JWTTOKENHelper.java** as Component, So SpringBoot will create bean Object. This is responsible for all JWT operations i.e. creation and validation of tokens.

```

import java.util.Date;
import org.springframework.stereotype.Component;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

@Component
public class JWTTOKENHelper {

    long JWT_TOKEN_VALIDITY_MILLIS = 5 * 60000; // 5 mins
    String secret = "fasfafacadasfasxASFACASDFACASDFASFASFASDAADSCSDFADCVSGCFVADX";

    //retrieve username from jwt token
    public String getUsernameFromToken(String token) {
        return
    Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody().getSubject();
    }

    //check if the token has expired
    private Boolean isTokenExpired(String token) {
        final Date expiration = Jwts.parser()
            .setSigningKey(secret)
            .parseClaimsJws(token)
            .getBody()
            .getExpiration();
        return expiration.before(new Date());
    }

    //generate JWT using the HS512 algorithm and secret key.
    public String generateToken(String userName) {
        return Jwts
            .builder()
            .setSubject(userName)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() +
JWT_TOKEN_VALIDITY_MILLIS))
            .signWith(SignatureAlgorithm.HS512, secret)
    }
}

```

```

        .compact();
    }

//validate token i.e. user name and time interval validation.
public Boolean validateToken(String token, String userName) {
    final String username = getUsernameFromToken(token);
    return (username.equals(userName) && !isTokenExpired(token));
}
}

```

- Now Define a custom filter by extending OncePerRequestFilter to handle JWT token for every incoming new request.
- This New Filter is responsible for checking like token available or not as part of Request
- If token available, This Filter validates token w.r.to user Id and Expiration time interval.
- This Filter is responsible for cross checking User Id of Token with Database user details.

```

import java.io.IOException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import com.dilip.jwt.token.JWTTokenHelper;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@Component
public class JWTTokenFilter extends OncePerRequestFilter {

    Logger logger = LoggerFactory.getLogger(JWTTokenFilter.class);

    @Autowired
    JWTTokenHelper jwtTokenHelper;

    @Autowired

```

```

UserAuthenticationServiceImpl authenticationService;

@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain) throws ServletException,
IOException {

    logger.info("validation of JWT token by OncePerRequestFilter");

    String token = request.getHeader("Authorization");

    logger.info("JWT token : " + token);
    String userName = null;

    if (token != null) {

        userName = this.jwtTokenHelper.getUsernameFromToken(token);
        logger.info("JWT token USer NAme : " + userName);
    } else {
        logger.info("ToKen is Misisng. Please Come with Token");
    }

    if (userName != null && SecurityContextHolder.getContext().getAuthentication() == null) {

        // fetch user detail from username
        UserDetails userDetails = this.authenticationService.loadUserByUsername(userName);
        Boolean isValidToken = this.jwtTokenHelper.validateToken(token,
userDetails.getUsername());

        if (isValidToken) {

            UsernamePasswordAuthenticationToken authenticationToken = new
UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());

            authenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authenticationToken);
        }
    }
    filterChain.doFilter(request, response);
}
}

```

- Now create a Authentication Configuration class responsible for Creating **AuthenticationManager**, **PasswordEncryptor** and **SecurityFilterChain** to define strategies

of incoming requests like which should be authenticated with JW and which should be ignored by Security layer.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
public class AppSecurityConfig {

    Logger logger = LoggerFactory.getLogger(AppSecurityConfig.class);

    @Autowired
    JWTTokenFilter jwtTokenFilter;

    @Bean
    AuthenticationManager getAuthenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {
        logger.info("Initilizing Bean AuthenticationManager");
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    BCryptPasswordEncoder getBCryptPasswordEncoder() {
        logger.info("Initilizing Bean BCryptPasswordEncoder");
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain getSecurityFilterChain(HttpSecurity security) throws Exception {
        logger.info("Configuring SecurityFilterChain Layer of URL patterns");

        security.csrf(csrf -> csrf.disable())
            .cors(cors -> cors.disable())
            .authorizeHttpRequests(
                auth -> auth.requestMatchers("/user/login", "/user/register")
                    .permitAll()
                    .anyRequest()
            );
    }
}
```

```

        .authenticated()
    )
.addFilterBefore(this.jwtTokenFilter,UsernamePasswordAuthenticationFilter.class);

    return security.build();
}

}

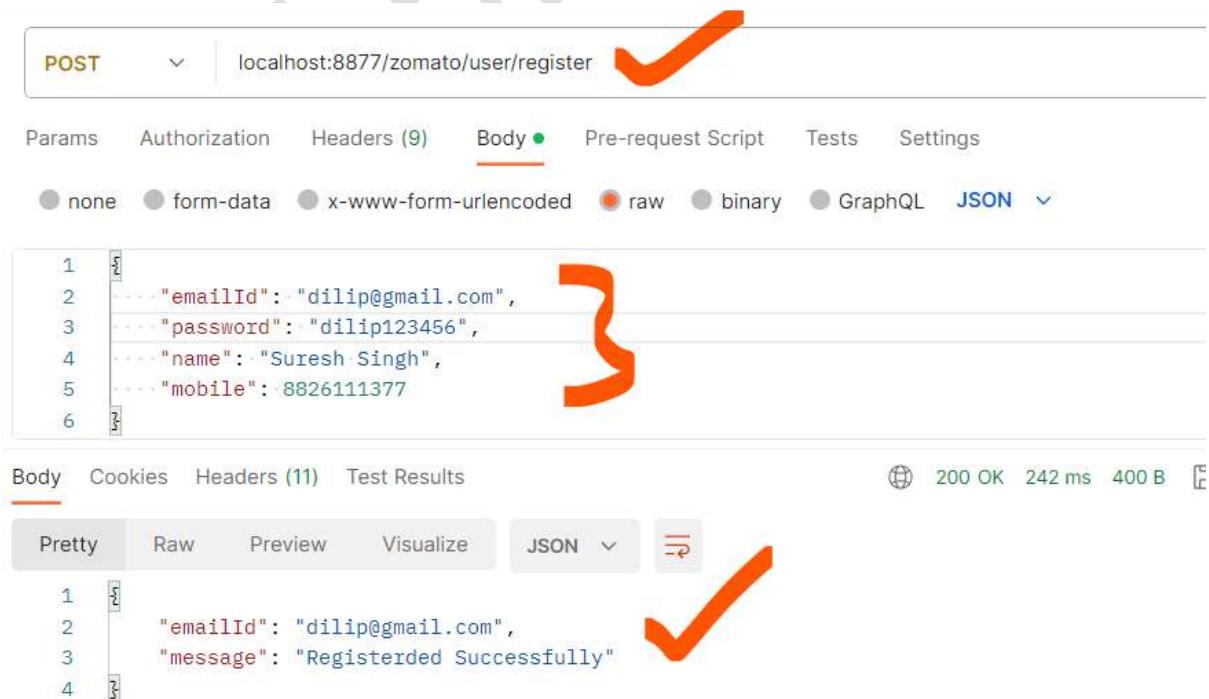
```

In Above Configuration Class, We created **SecurityFilterChain** Bean with Security rules defined for every incoming request. We are defined Security Configuration as, permitting all incoming requests without JWT token validation for both URI mappings of **"/user/login","/user/register"**. Apart from these 2 mappings , any other request should be authenticated with JWT Token i.e. every request should come with valid token always then only we are allowing to access actual Resources or Services.

Testing : Let's Test our application as per our requirement and security configuration.

Case 1: Now Create User Accounts with localhost:8877/zomato/user/register

This is Open API Service means security not applicable for this. i.e. to execute no need to provide JWT.



POST localhost:8877/zomato/user/register

Params Authorization Headers (9) Body **JSON** Tests Settings

Body (Pretty, Raw, Preview, Visualize, JSON) Headers (11) Test Results

```

1 {
2   "emailId": "dilip@gmail.com",
3   "password": "dilip123456",
4   "name": "Suresh Singh",
5   "mobile": 8826111377
6 }

```

Body Cookies Headers (11) Test Results

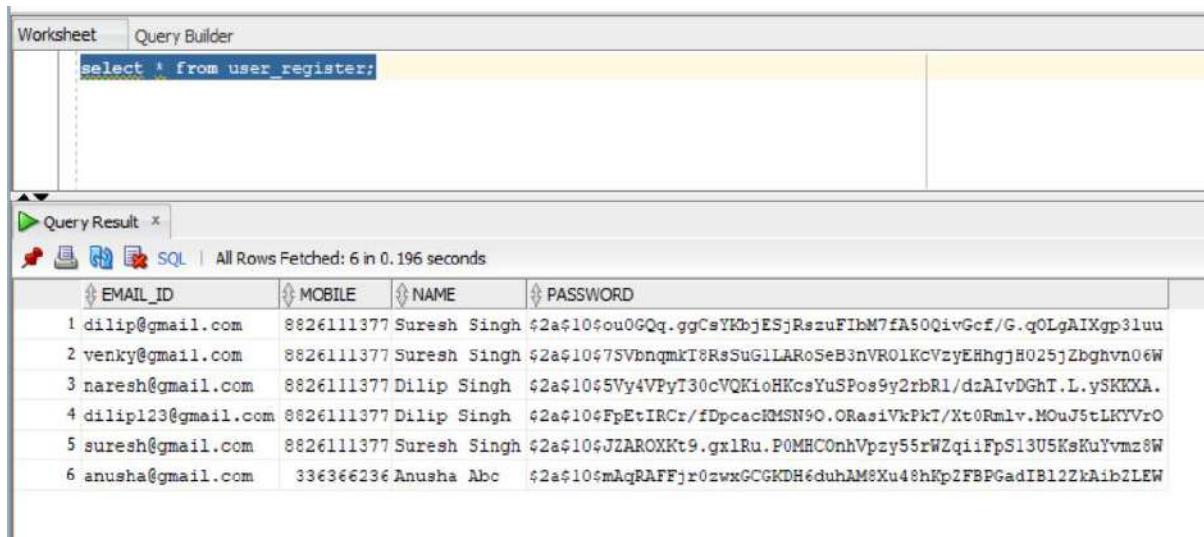
200 OK 242 ms 400 B

```

1 {
2   "emailId": "dilip@gmail.com",
3   "message": "Registered Successfully"
4 }

```

Now verify password value in database how it is stored because we encoded it.



Worksheet Query Builder

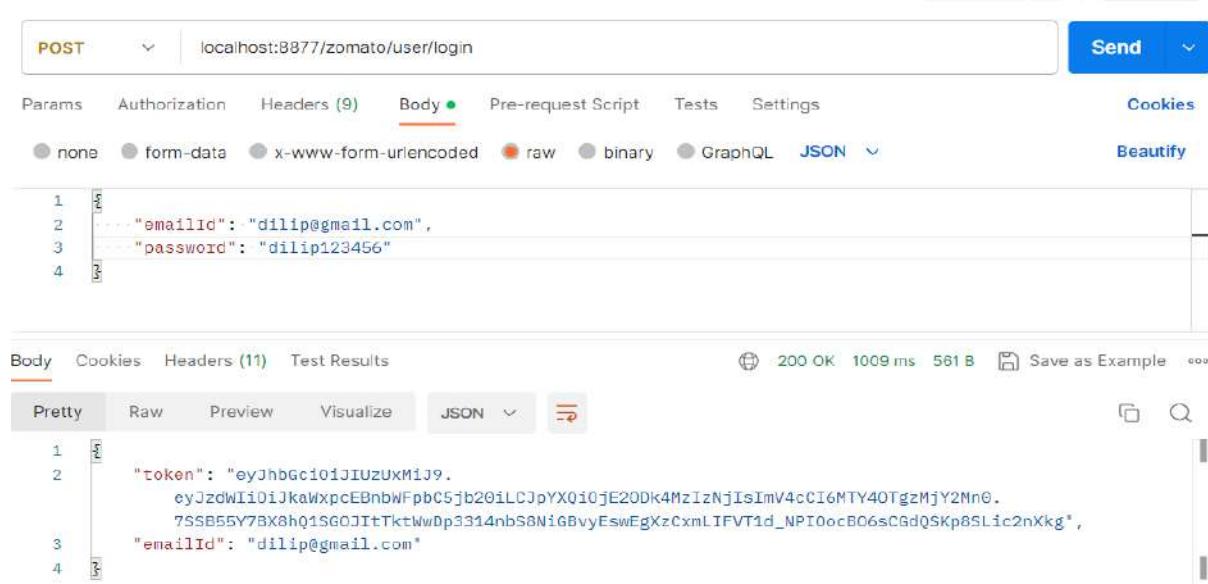
```
select * from user_register;
```

Query Result | All Rows Fetched: 6 in 0.196 seconds

EMAIL_ID	MOBILE	NAME	PASSWORD
1 dilip@gmail.com	8826111377	Suresh Singh	\$2a\$10\$ou0GQq.ggCsYKbjESjRszuFIbM7fa50QivGcf/G.qOLgAIXgp3luu
2 venky@gmail.com	8826111377	Suresh Singh	\$2a\$10\$7SVbnqmkT8RsSuG1LARoSeB3nVR01KcVzyEHhgjH025jZbghvn06w
3 naresh@gmail.com	8826111377	Dilip Singh	\$2a\$10\$5Vy4VPyT30cVQKioHKcsYuSPos9y2rbR1/dzA1vDGHt.L.ySKKXA.
4 dilip123@gmail.com	8826111377	Dilip Singh	\$2a\$10\$FpEtIRCr/fDpcacKMSN90.ORasiVkBkT/Xt0Rmlv.MOuJ5tLKYVrO
5 suresh@gmail.com	8826111377	Suresh Singh	\$2a\$10\$JZAROXKt9.gx1Ru.P0MHC0nhVpzy55rWZqiiFpS13U5KsKuYvmz8W
6 anusha@gmail.com	336366236	Anusha Abc	\$2a\$10\$mAqRAFFjr0zwGCGKDH6duhAM8Xu48hKp2FBPGadIB12ZkAibZLEW

Passwords are stored as encoded format. Now Spring also takes care of decoded while authentication because we are created Bean of Password Encryptor.

Case 2: Now try to login with User Details. localhost:8877/zomato/user/login



POST localhost:8877/zomato/user/login Send

Params Authorization Headers (9) Body (1) Pre-request Script Tests Settings Cookies

Body

```

1 {
2   "emailId": "dilip@gmail.com",
3   "password": "dilip123456"
4 }
```

Body Cookies Headers (11) Test Results 200 OK 1009 ms 561 B Save as Example

Pretty Raw Preview Visualize JSON

```

1 {
2   "token": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpCEBnbWFpbC5jb20iLCJpYXQiOjE20Dk4MzIzNjIsImV4cCI6MTY4OTgzMjY2Mn0.7SSB5Y7BX8h01SG0JITTkWwDp3314nbS8NiGBvyEswEgXzCxmLIFVT1d_NPI0ocB06sCGdQSKp8SLic2nXkg",
3   "emailId": "dilip@gmail.com"
4 }
```

On Successful validation, We received JSON Web token.

If we provide Wrong Credentials: Entered Wrong Password.

The screenshot shows a Postman request to `localhost:8877/zomato/user/login`. The **Body** tab contains the following JSON:

```

1
2   ...
3     "emailId": "dilip@gmail.com",
4     "password": "dilip1234"
5

```

The response status is **403 Forbidden** with a **232 ms** duration and **300 B** size. The response body is empty.

Now we got expected and default Response as Forbidden with status code 403:

Forbidden meaning is: **not allowed; banned**.

Case 3: Access Other URI or Services with JWT: `localhost:8877/zomato/user/hello`

Here we will pass Token as **Authorization** Header as part of Request. Copy Token from login response from previous call and pass an value to **Authorization** Header as following.

The screenshot shows a Postman request to `localhost:8877/zomato/user/hello`. The **Headers** tab is selected, showing the following headers:

- Accept: `/*`
- Accept-Encoding: `gzip, deflate, br`
- Connection: `keep-alive`
- Authorization: `eyJhbGciOiJIUzUxMiJ9.eyJzdWJlOiJkaWxp...`

The response status is **200 OK** with a **72 ms** duration and **353 B** size. The response body is `Welcome to Security`.

Now received respected Response value and Status code as 200 OK. i.e. Internally JWT is validated with user and database as per our logic implemented. We can see in Logs of application.

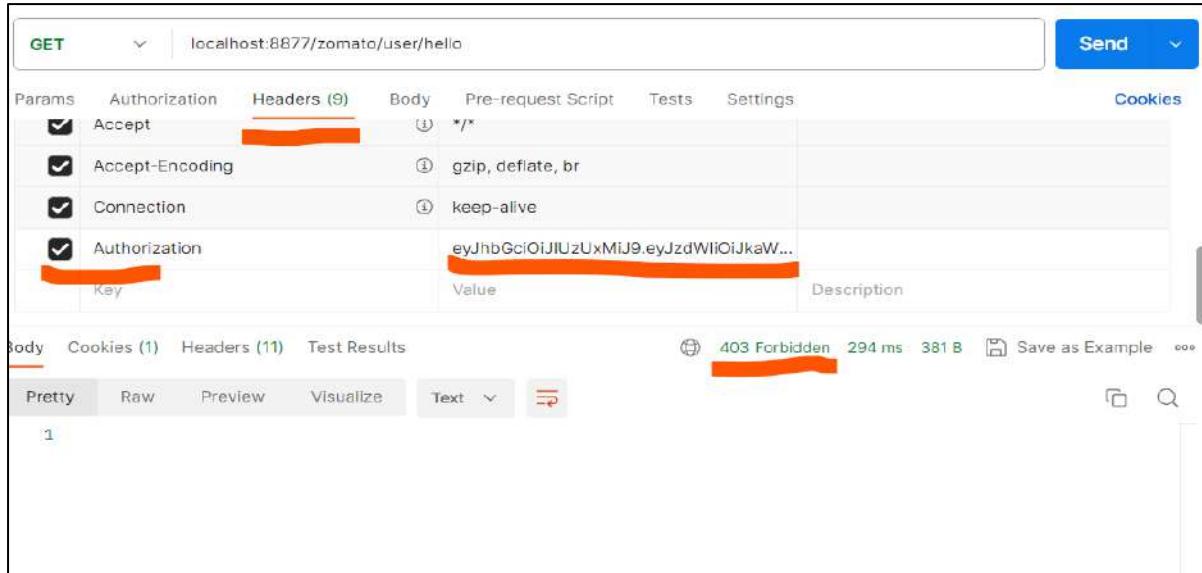
```

2023-07-20T11:54:36.084+05:30  INFO 17780 --- [nio-8877-exec-9] com.dilip.security.JWTTokenFilter : validation of JWT token by OncePerRequestFilter
2023-07-20T11:54:36.084+05:30  INFO 17780 --- [nio-8877-exec-9] com.dilip.security.JWTTokenFilter : JWT token
2023-07-20T11:54:36.084+05:30  INFO 17780 --- [nio-8877-exec-9] com.dilip.security.JWTTokenFilter : JWT token User Name : dilip@gmail.com
2023-07-20T11:54:36.087+05:30  INFO 17780 --- [nio-8877-exec-9] c.d.s.UserAuthenticationServiceImpl : Fetching UserDetails
2023-07-20T11:54:36.087+05:30  INFO 17780 --- [nio-8877-exec-9] c.d.s.UserAuthenticationServiceImpl : Hibernate: select ul_0.email_id,ul_0.mobile,ul_0.name,ul_0.password from user_register ul_0 where ul_0.email_id=?

```

Case 4: Access Other URI or Services with Invalid JWT: localhost:8877/zomato/user/hello

i.e. Provided Expired Token.



The screenshot shows a Postman request for a GET method to the URL `localhost:8877/zomato/user/hello`. The Headers tab is selected, showing the following configuration:

- Accept: `application/json, text/plain, */*`
- Accept-Encoding: `gzip, deflate, br`
- Connection: `keep-alive`
- Authorization: `eyJhbGciOiJIUzUxMiJ9.eyJzdWJkaWxpcEBnbWFpbC5jb20iLCJpYXQiOjE2ODk4MzQyNTYsImV4cCI6MTY4OTgzNDU1Nn0.Bkf3aPnx6rUY0uykrtIDt23xGRdlvtrc5sxiplPPcUxnJnb5TddpIGgPEiMoBCszYGMrDjRnq2kiVC1zAvylnQ` (redacted)

The response status is **403 Forbidden** with a **294 ms** duration and **381 B** size. The Body tab shows a single character `1`.

Token validated and found as Expired, so Server application returns Response as 403 Forbidden at client level.

Check Server Level logs:



```

Properties Console x
spring-boot-security-jwt - SpringBootSecurityJwtApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe
2023-07-20T12:04:30.796+05:30  INFO 17780 --- [nio-8877-exec-3] com.dilip.security.JWTTokenFilter : validation of JWT token by OncePerRequestFilter
2023-07-20T12:04:30.796+05:30  INFO 17780 --- [nio-8877-exec-3] com.dilip.security.JWTTokenFilter : JWT token :
eyJhbGciOiJIUzUxMiJ9.eyJzdWJkaWxpcEBnbWFpbC5jb20iLCJpYXQiOjE2ODk4MzQyNTYsImV4cCI6MTY4OTgzNDU1Nn0.Bkf3aPnx6rUY0uykrtIDt23xGRdlvtrc5sxiplPPcUxnJnb5TddpIGgPEiMoBCszYGMrDjRnq2kiVC1zAvylnQ
2023-07-20T12:04:30.797+05:30 ERROR 17780 --- [nio-8877-exec-3] o.a.c.c.C.[.].[dispatcherServlet] : Servlet.service() for servlet [dispatcherServlet] in context with path [/zomato] threw exception

io.jsonwebtoken.ExpiredJwtException: JWT expired at 2023-07-20T11:59:16Z. Current time: 2023-07-20T12:04:30Z, a difference of 314797 milliseconds. Allowed clock skew: 0 milliseconds.
    at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:385) ~[jjwt-0.9.1.jar:0.9.1]
    at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:481) ~[jjwt-0.9.1.jar:0.9.1]

```

This is how we are applying security layer to our Spring Boot Web Application with JWT exchanging.

I have uploaded this entire working copy project in GitHub.

GitHub Repository Link: <https://github.com/DilipItAcademy/spring-boot-3-security-jwt>

Micro Services With Spring Boot

**MicroServices
With
SpringBoot**



Microservices, also known as the microservice architecture, is an architectural style used in software development to design and build applications as a collection of small, independent, and loosely coupled services. Each service represents a specific business capability and can be developed, deployed, and maintained independently of other services.

In a traditional monolithic application, all functionality is bundled together into a single large codebase, making it difficult to scale, maintain, and update. Microservices, on the other hand, promote a more modular approach, breaking down the application into smaller, manageable components.

Key characteristics of microservices include:

Decentralization: Each microservice operates independently and has its own database (if required) and codebase.

Loose coupling: Microservices communicate with each other through well-defined interfaces, usually using lightweight protocols like HTTP/REST or messaging systems.

Independently deployable: Since each microservice is self-contained, updates or bug fixes can be made to a single service without affecting the others, reducing the risk of unintended side effects.

Scalability: Specific services can be scaled independently based on demand, optimizing resource utilization.

Resilience: If one microservice fails, the entire application doesn't necessarily collapse. The other services can continue to function as long as they don't depend on the failed service.

Technology diversity: Microservices allow the use of different programming languages, frameworks, and data storage systems for each service, as long as they can communicate effectively.

Focused on business capabilities: Each microservice is designed to address a specific business capability or function, making it easier to understand and maintain.

Though microservices offer several advantages, they also introduce complexities like distributed system management, inter-service communication, and the need for robust monitoring. Implementing microservices requires careful planning, and it is essential to choose the right architecture based on the specific needs and goals of the application or project. When done right, microservices can lead to a more agile, scalable, and maintainable software development process.

Some examples of applications that use microservices architectures include:

Amazon: Amazon uses microservices to build its e-commerce platform. Each microservice is responsible for a specific function, such as product search, checkout, or order fulfilment.

Netflix: Netflix uses microservices to stream movies and TV shows. Each microservice is responsible for a specific content type, such as movies, TV shows, or documentaries.

Uber: Uber uses microservices to power its ride-hailing platform. Each microservice is responsible for a specific aspect of the ride-hailing process, such as finding drivers, matching riders with drivers, and tracking rides.

If you are considering using a microservices architecture for your next application, there are a few things you should keep in mind:

- Microservices can be complex to design and implement. It is important to have a clear understanding of your application's requirements before you start designing your microservices architecture.
- Microservices require a good DevOps culture. You need to have a way to automate the deployment, scaling, and monitoring of your microservices.
- Microservices can be more expensive to maintain than monolithic architectures. You need to have a way to track and manage the dependencies between your microservices.

Overall, microservices architectures can be a good choice for applications that need to be scalable, resilient, and evolvable. However, they can be complex to design and implement, so you need to carefully consider your application's requirements before you decide to use a microservices architecture.

Difference Between Monolithic and Microservices:

Monolithic and microservices are two different architectural styles used in software development. They have distinct characteristics that impact how applications are designed, built, deployed, and maintained. Here are the key differences between monolithic and microservices:

Architecture:

Monolithic: In a monolithic architecture, the entire application is built as a single, cohesive unit. All the components, functionalities, and services are tightly integrated and deployed together.

Microservices: In a microservices architecture, the application is divided into smaller, independent services that communicate with each other through APIs. Each service is developed and deployed independently, and they can be written in different programming languages.

Scalability:

Monolithic: Scaling a monolithic application often involves replicating the entire application, including all its components, even if only a specific part of the application requires more resources.

Microservices: Microservices offer finer-grained scalability. You can scale individual services independently based on their specific resource needs, allowing for more efficient resource utilization.

Deployment and Release Cycle:

Monolithic: Since the entire application is deployed as one unit, making changes to specific parts requires deploying the whole application. This can result in longer release cycles and a higher risk of introducing bugs.

Microservices: Microservices enable continuous deployment and faster release cycles. Changes to a specific service can be made and deployed independently, without affecting the rest of the application. This makes it easier to release updates and bug fixes more frequently.

Development Team Organization:

Monolithic: In a monolithic architecture, all developers typically work on the same codebase. As the application grows, it may become more challenging for developers to work simultaneously without causing conflicts.

Microservices: Microservices promote a decentralized development approach. Different development teams can work on separate services, which fosters autonomy and faster development cycles.

Technology Diversity:

Monolithic: Monolithic applications usually stick to a single technology stack since they are all built as a single unit.

Microservices: In microservices, different services can use different technology stacks that are best suited for their specific tasks. This allows for greater flexibility and choice in technology.

Fault Isolation and Resilience:

Monolithic: A failure in one part of the monolithic application can bring down the entire system since all components are tightly integrated.

Microservices: Microservices are designed for fault isolation. If a single service fails, it does not necessarily affect the other services, increasing the overall system's resilience.

Complexity and Maintenance:

Monolithic: As a monolithic application grows, it can become more complex and challenging to maintain, especially when many developers are working on it simultaneously.

Microservices: While each microservice is simpler in structure, managing and coordinating multiple services can introduce its own complexities.

Both monolithic and microservices architectures have their advantages and challenges, and the choice between them depends on various factors, including the specific requirements of the application, team structure, scalability needs, and development philosophy.

Microservices with Spring Boot:

Microservices with Spring Boot is a popular combination for building scalable and flexible applications based on the microservices architectural style. Spring Boot is a powerful framework from the Spring ecosystem that simplifies the development of Java-based applications, while microservices is an architectural approach that breaks down applications into smaller, loosely-coupled services that can be developed, deployed, and maintained independently.

Here are some key aspects of building microservices with Spring Boot:

Service Creation: Spring Boot provides a convenient way to create microservices using various starters and auto-configuration. You can quickly set up a new microservice project with just a few lines of code.

Independence: Each microservice developed using Spring Boot is an independent unit of functionality, allowing developers to focus on specific business capabilities without being affected by the implementation details of other services.

Communication: Microservices need to communicate with each other to fulfill complex business processes. Spring Boot offers various ways to implement communication between services, such as RESTful APIs, messaging systems (e.g., RabbitMQ or Apache Kafka), and gRPC.

Spring Cloud: Spring Cloud is an extension of the Spring ecosystem that provides additional tools and libraries to simplify the development of microservices. It offers features like service discovery (Eureka), load balancing (Ribbon), centralized configuration management (Spring Cloud Config), circuit breakers (Hystrix), and more.

Containerization: Microservices are often deployed using containerization technologies like Docker and container orchestration platforms like Kubernetes. Spring Boot applications are well-suited for containerization due to their lightweight nature and easy deployment.

Scalability: Spring Boot microservices can be scaled individually, allowing you to allocate resources based on the specific needs of each service. This fine-grained scalability is one of the advantages of the microservices architecture.

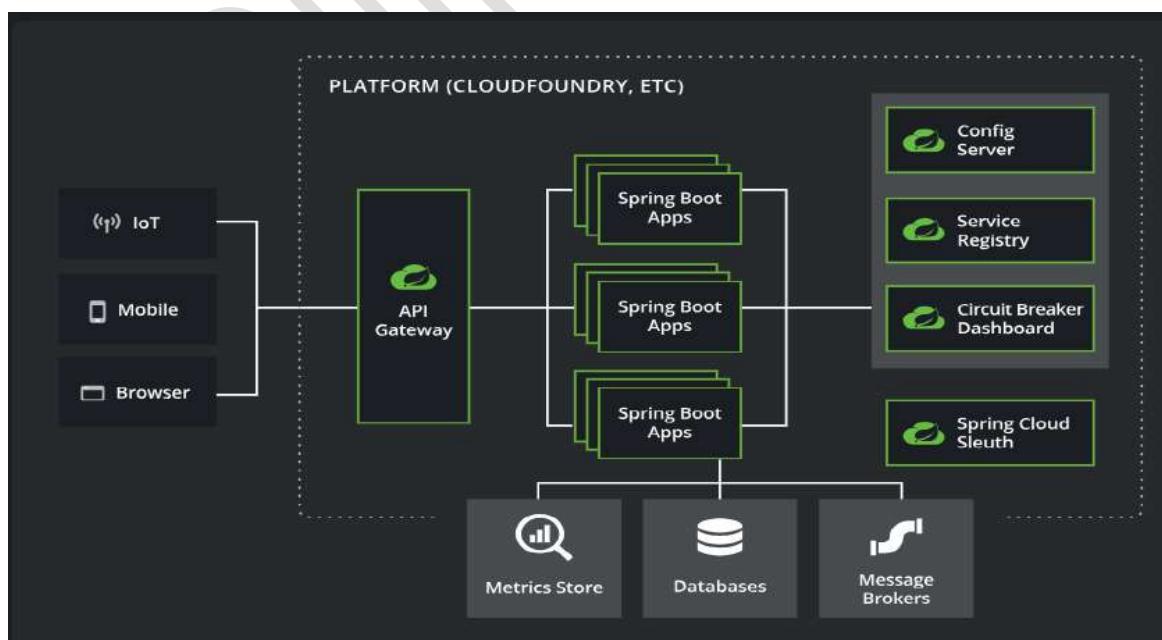
Monitoring and Logging: Monitoring and logging are crucial for maintaining the health of microservices. Spring Boot provides integration with popular logging frameworks like Logback and supports metrics and monitoring tools like Spring Actuator and Micrometer.

Testing: Spring Boot offers a testing framework that makes it easier to write unit tests and integration tests for microservices. You can test each service independently, ensuring that it functions correctly in isolation and when interacting with other services.

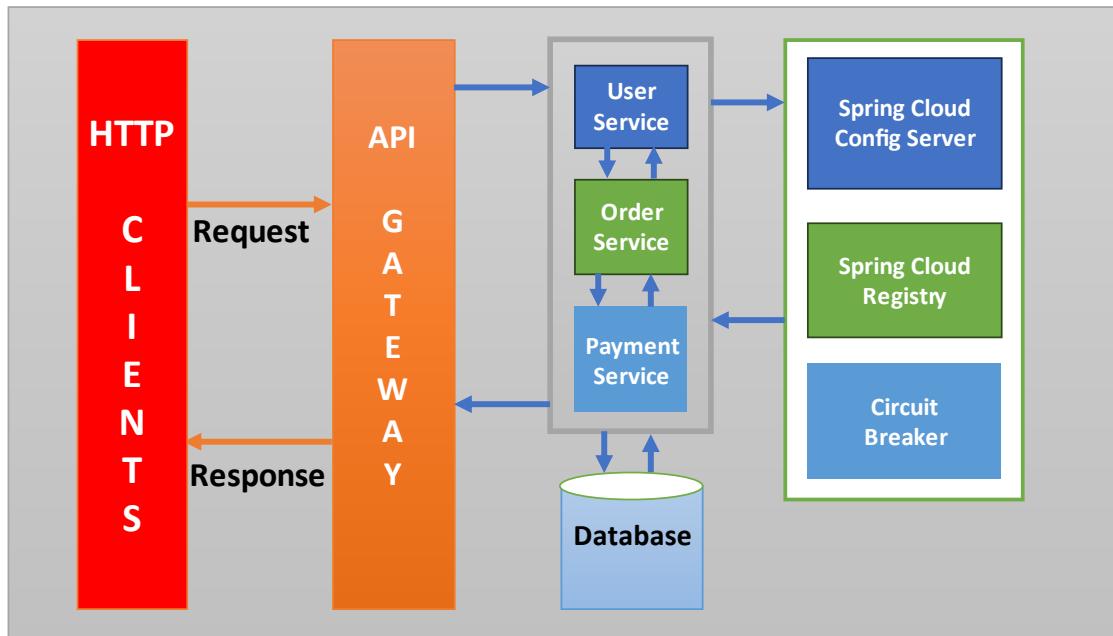
Remember that while Spring Boot simplifies many aspects of microservices development, building a successful microservices architecture still requires careful design and consideration of factors such as service boundaries, communication patterns, data consistency, and deployment strategies. Spring Boot's many purpose-built features make it easy to build and run your microservices in production at scale. And don't forget, no microservice architecture is complete without Spring Cloud API.

SpringBoot and Microservices Architecture:

The distributed nature of microservices brings challenges. Spring helps you mitigate these. With several ready-to-run cloud patterns, Spring Cloud can help with service discovery, load-balancing, circuit-breaking, distributed tracing, and monitoring. It can even act as an API gateway.



Here is what a typical microservice architecture. For example, consider this microservice architecture for a simple shopping cart application. It has different services like **User service**, **Order service**, and **Payment service**, and these are the independent and loosely coupled services in the microservices projects.



API Gateway: Spring Cloud Gateway is a part of the Spring Cloud ecosystem and is an API gateway built on top of Spring Framework and Spring Boot. An API gateway is a server that acts as an intermediary between clients (such as web or mobile applications) and the microservices that provide various functionalities. It is a crucial component in a microservices architecture.

Spring Cloud Config Server: Spring Cloud Config Server is another component of the Spring Cloud ecosystem. It provides a centralized configuration management solution for microservices-based applications. In a microservices architecture, you typically have multiple instances of services running on different servers, and managing their configurations can become challenging.

Circuit Breaker: A Circuit Breaker is a software design pattern used in distributed systems to handle failures and prevent cascading failures across interconnected micro services. It is an important component in building resilient and fault-tolerant microservices architectures.

Implementation of Micro Services:

1. Primarily we will Create 3 Spring Boot applications as per our architecture. After Creation of Micro Services, we will integrate those with other components as defined in Architecture.

```

user :
  Port : 8001
  Context Path: /user

order :
  Port: 8002
  Context Path: /order

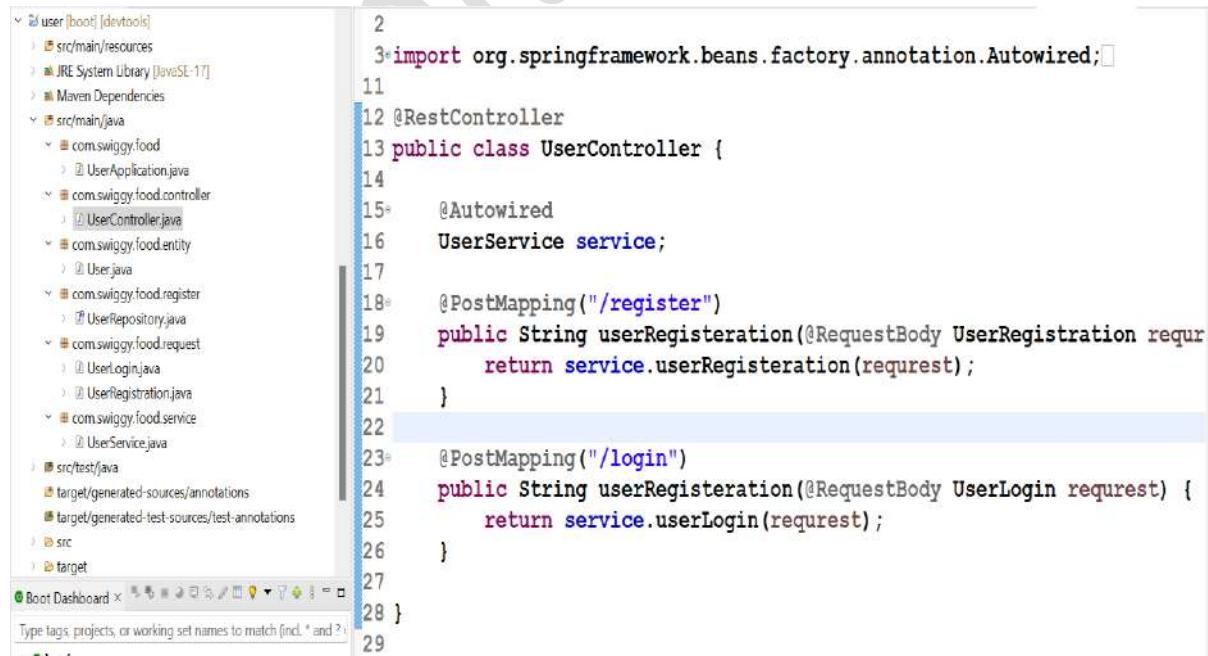
payment:
  Port: 8003
  Context Path: /payment

```

User Micro Service

Please implement a Spring Boot Web Application for User Functionalities.

Server Port : 8001
Context Path: /user
Application Name: user-service



The screenshot shows a Java code editor with a code editor and a file browser on the left. The code editor displays the following Java code for a UserController:

```

2
3import org.springframework.beans.factory.annotation.Autowired;
11
12@RestController
13public class UserController {
14
15    @Autowired
16    UserService service;
17
18    @PostMapping("/register")
19    public String userRegistration(@RequestBody UserRegistration request)
20        return service.userRegistration(request);
21    }
22
23    @PostMapping("/login")
24    public String userLogin(@RequestBody UserLogin request) {
25        return service.userLogin(request);
26    }
27
28}
29

```

The file browser on the left shows the project structure:

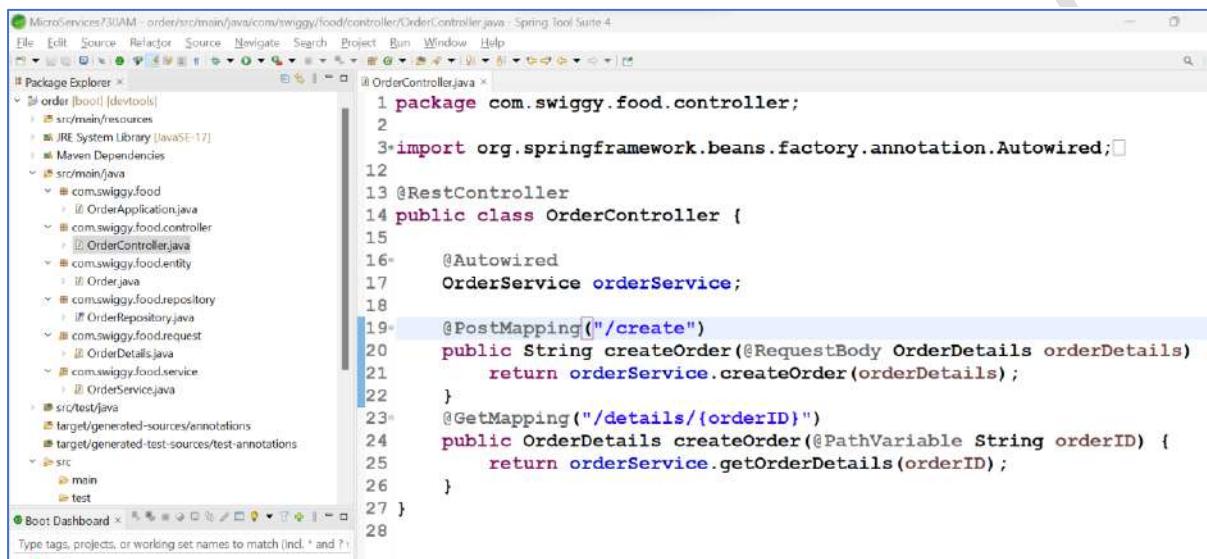
- src/main/java
 - com.swiggy.food
 - UserApplication.java
 - UserController.java
 - User.java
 - UserRepository.java
 - UserLogin.java
 - UserRegistration.java
 - com.swiggy.food.entity
 - com.swiggy.food.register
 - com.swiggy.food.service
- src/test/java
- target/generated-sources/annotations
- target/generated-test-sources/test-annotations
- src
- target

Similarly Implement two other Spring Boot Web Applications.

Order Micro Service :

order :

Server Port: 8002
Context Path: /order
Application Name: order-service



The screenshot shows the Spring Tool Suite 4 interface with the 'order' project selected in the Package Explorer. The 'OrderController.java' file is open in the editor. The code defines a REST controller with two endpoints: a POST endpoint for creating an order and a GET endpoint for getting an order by ID. The controller is annotated with @RestController and @Autowired, and it uses @PostMapping and @GetMapping annotations.

```

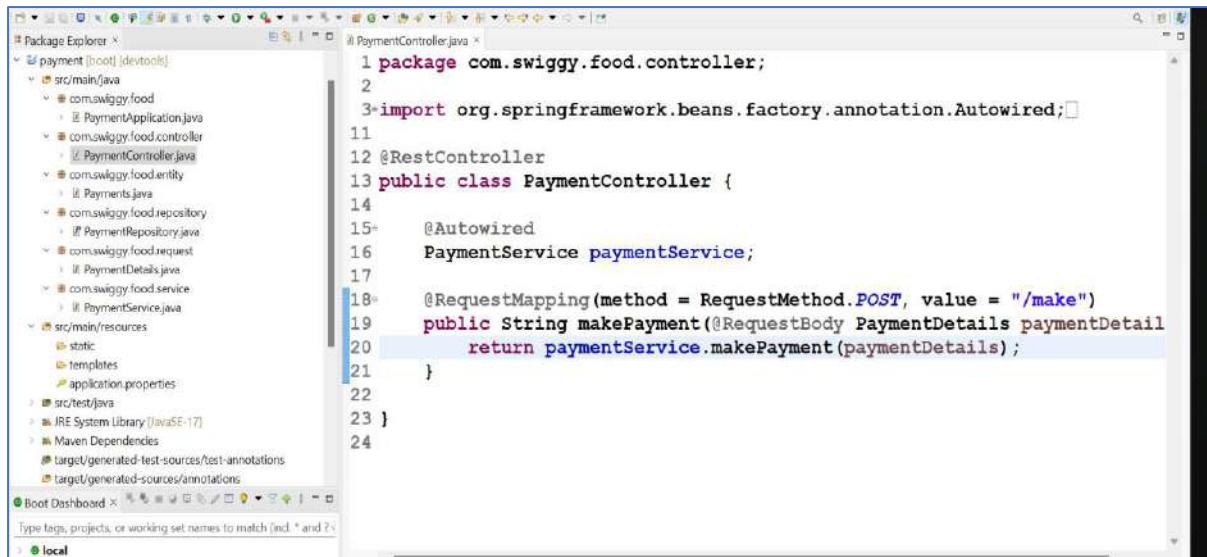
1 package com.swiggy.food.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 @RestController
6 public class OrderController {
7
8     @Autowired
9     OrderService orderService;
10
11     @PostMapping("/create")
12     public String createOrder(@RequestBody OrderDetails orderDetails) {
13         return orderService.createOrder(orderDetails);
14     }
15
16     @GetMapping("/details/{orderID}")
17     public OrderDetails createOrder(@PathVariable String orderID) {
18         return orderService.getOrderDetails(orderID);
19     }
20
21 }
22
23
24
25
26
27
28

```

Payment Micro Service :

payment:

Server Port : 8003
Context Path : /payment
Application Name: payment-service



```

1 package com.swiggy.food.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 @RestController
6 public class PaymentController {
7
8     @Autowired
9     PaymentService paymentService;
10
11     @RequestMapping(method = RequestMethod.POST, value = "/make")
12     public String makePayment(@RequestBody PaymentDetails paymentDetail)
13         return paymentService.makePayment(paymentDetail);
14
15 }
16
17
18 }
19
20
21 }
22
23 }
24

```

Now We are ready with 3 Micro Services, Let's integrate with Micro Services Architecture. As part of MicroServices Architecture, primarily we should implement API Gateway application.

API Gateway:

Spring Cloud Gateway is a library for building API gateways on top of Spring and Java. It provides a flexible way of routing requests based on a number of criteria, as well as focuses on cross-cutting concerns such as security, resiliency, and monitoring.

Here are some of the key features of Spring Cloud Gateway:

Routing: Spring Cloud Gateway can route requests to different microservices based on a variety of criteria, such as the request path, the HTTP method, or the request headers.

Filtering: Spring Cloud Gateway can filter requests before they are routed to the microservices. This can be used to add security, logging, or other functionality.

Resiliency: Spring Cloud Gateway can provide resilience to your microservices by using circuit breakers and other mechanisms. This can help to prevent your microservices from becoming unavailable if they are overloaded or experiencing errors.

Monitoring: Spring Cloud Gateway can be monitored using the Spring Boot Actuator. This allows you to track the performance of your gateway and the microservices that it routes to.

Here are some of the benefits of using Spring Cloud Gateway:

Ease of use: Spring Cloud Gateway is easy to use, even for developers who are not familiar with API gateways.

Flexibility: Spring Cloud Gateway is very flexible and can be used to route requests in a variety of ways.

Performance: Spring Cloud Gateway is performant and can handle a high volume of requests.

Reliability: Spring Cloud Gateway is reliable and can help to prevent your microservices from becoming unavailable.

Here are some of the drawbacks of using Spring Cloud Gateway:

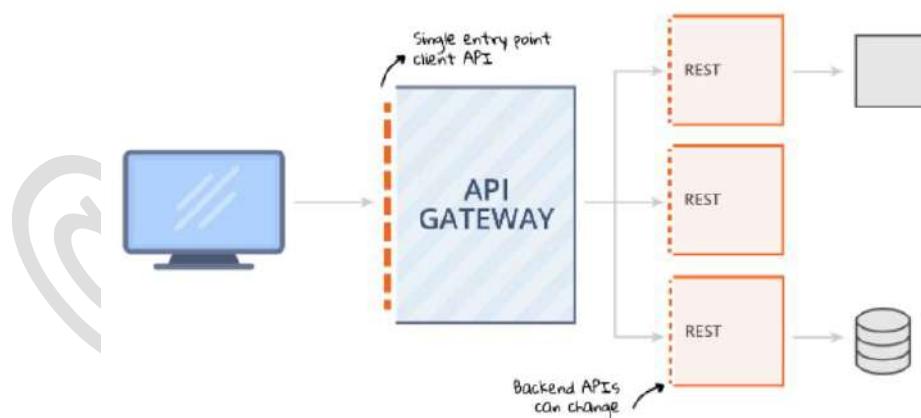
Complexity: Spring Cloud Gateway can be complex to configure, especially for large and complex applications.

Performance: Spring Cloud Gateway can have a negative impact on the performance of your microservices, especially if you are using a lot of filters.

Security: Spring Cloud Gateway does not provide any security out of the box. You will need to implement your own security mechanisms.

Overall, Spring Cloud Gateway is a powerful and flexible API gateway that can be used to build scalable and reliable microservices applications. However, it is important to be aware of the potential drawbacks of using Spring Cloud Gateway before you decide to use it in your application.

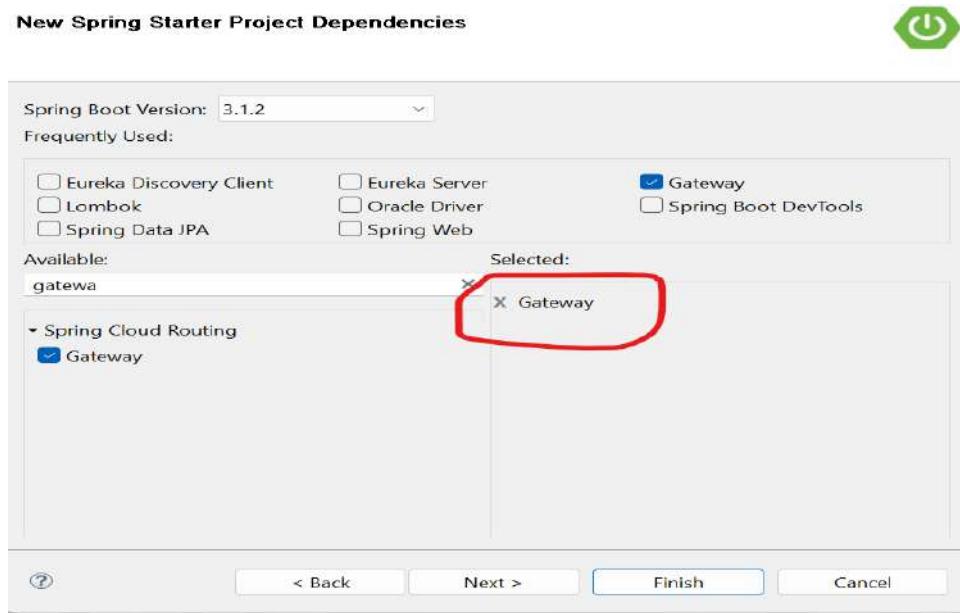
An API Gateway acts as a single entry point for a collection of microservices. Any external client cannot access the microservices directly but can access them only through the application gateway.



We are going to configure our three Miro Services with Gateway application i.e. All clients Request and response of our three MicroServices will be accessed via Gateway application instead of providing direct access. With help of API gateway application, we are bringing all our micro services under one port number instead of individual ports.

How to Create Spring Cloud API Gateway:

While creating Spring Boot Gateway Application, From Sprint Boot Starters add **Gateway starter** and finish.



Now our Spring Boot API gateway application is Ready, So Let' configure our micro services with Gateway.

Micro Services individual Port Numbers and Context paths what we created in previous steps.

```

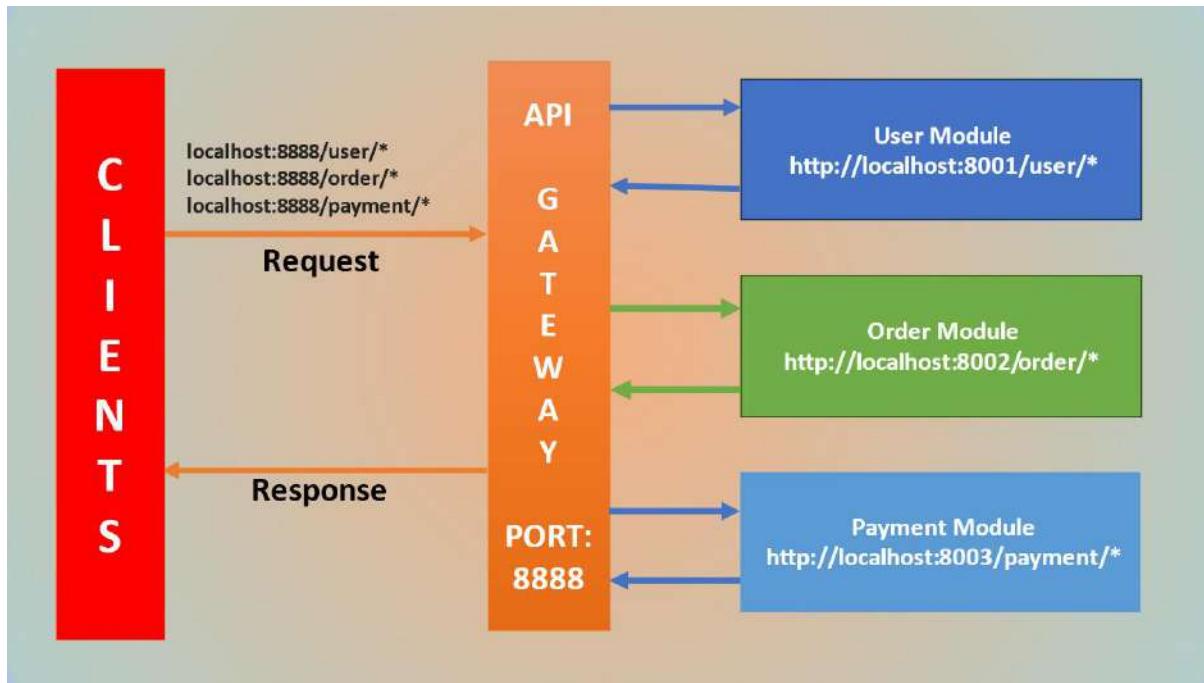
users-mgmt  :
  Port: 8001
  Context Path: /user

order-mgmt  :
  Port: 8002
  Context Path: /order

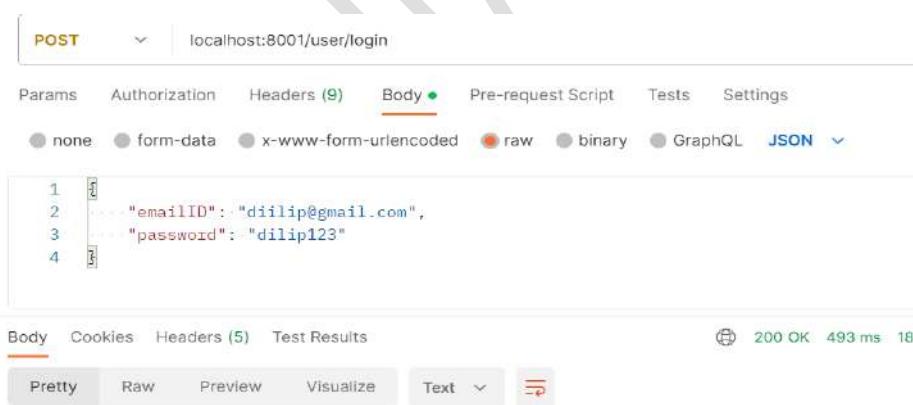
payment-mgmt  :
  Port: 8003
  Context Path: /payment
  
```

How to Configure Micro Services with Gateway application?

Following image explains how External HTTP clients will access our Micro Services endpoints via Gateway application. So here we should configure our micro services information inside Gateway application. This Process will be called as **Routing Configuration**.



Generally, If we want to access REST services User Micro Service, then we will access with User Micro Service Port number **8001** in URL as followed.



If we follow similar approach, then for every individual micro Service access we should use individual port numbers, it is a tedious process to HTTP clients to manage multiple port numbers while they are integrating or consuming our all MicroServices REST Services. In this scenario, we should make sure only port number being used across multiple micro services access from our application.

So Now we will configure our micro services with Gateway application, in a way as our Micro Services are accessible by only Gateway port number instead of their individual port numbers.

What is Routing in Gateway?

In Spring Boot API Gateway, routing is the process of determining which microservice to send a request to. The gateway uses a routing table to store information about the microservices it is connected to and the paths between them. When a request arrives at the gateway, the gateway looks up the destination path in its routing table and determines the microservice that the request should be forwarded to.

Routing is an important part of Spring Boot API Gateway. It allows the gateway to route requests to the correct microservices, which can improve the performance, reliability, and security of network communication.

There are two main ways to configure routing in Spring Boot API Gateway:

1. Declaratively:

This is done by configuring the routing table in the **application.properties** file. The routing table can be configured using a number of different predicates and filters.

Here is an example of how to configure routing declaratively in Spring Boot API Gateway:

```
spring.cloud.gateway.routes[0].id=user-route
spring.cloud.gateway.routes[0].uri=http://localhost:8080/users
spring.cloud.gateway.routes[0].predicates=Path=/api/users/**
```

These properties creates a route with the ID "**user-route**" that routes requests of the **/api/users/**** path to the microservice running at **http://localhost:8080/users**

2. Programmatically:

This is done by creating a **RouteLocator** bean and registering it with the Spring Boot application. The **RouteLocator** bean is responsible for creating and managing the routing table. Here is an example of how to configure routing programmatically in Spring Boot API Gateway:

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("user-route", r -> r.path("/api/users/**"))
        .uri("http://localhost:8080/users"))
        .build();
}
```

This code creates a route with the ID "**user-route**" that routes requests of the **/api/users/**** path to the microservice running at **http://localhost:8080/users**

Now Let's configure our three micro services with Gateway Declaratively i.e. in Properties file Level.

MicroServices with Gateway Configuration:

#Gateway Application Details

```

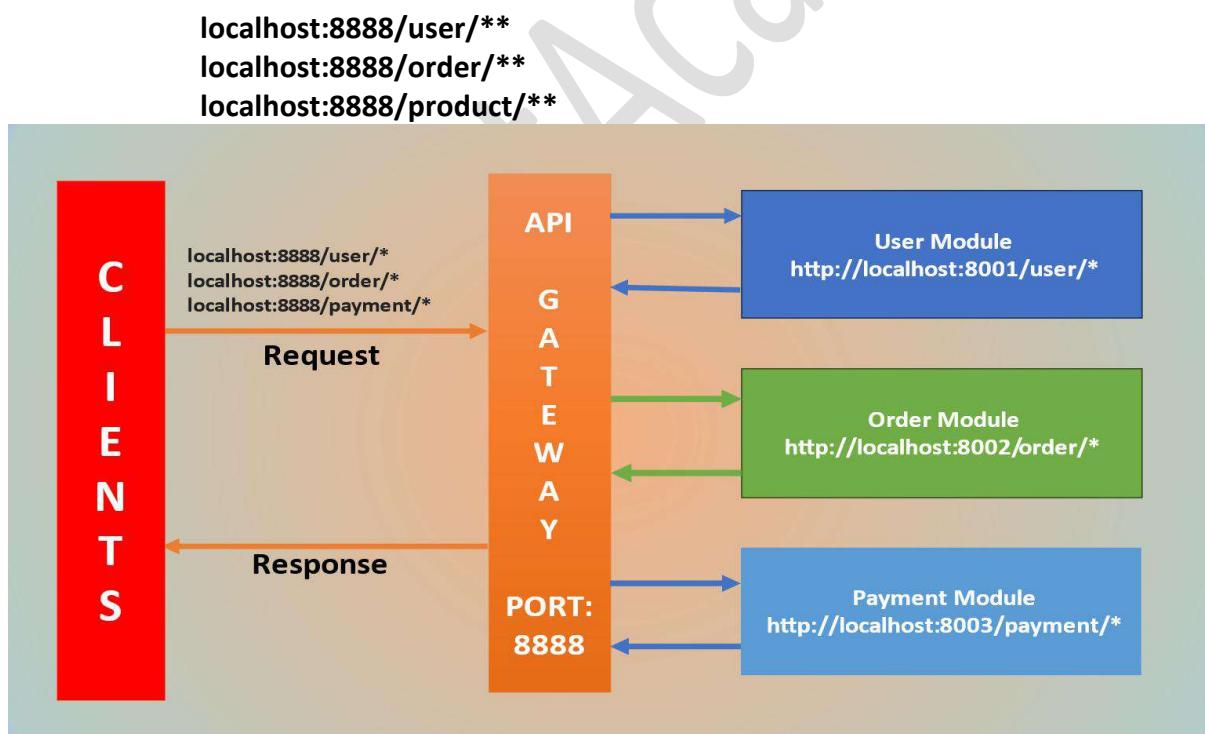
server.port=8888
spring.application.name= swiggy-gateway

#user api mapping
spring.cloud.gateway.routes[0].id=user
spring.cloud.gateway.routes[0].uri=http://localhost:8001
spring.cloud.gateway.routes[0].predicates[0]=Path=/user/**

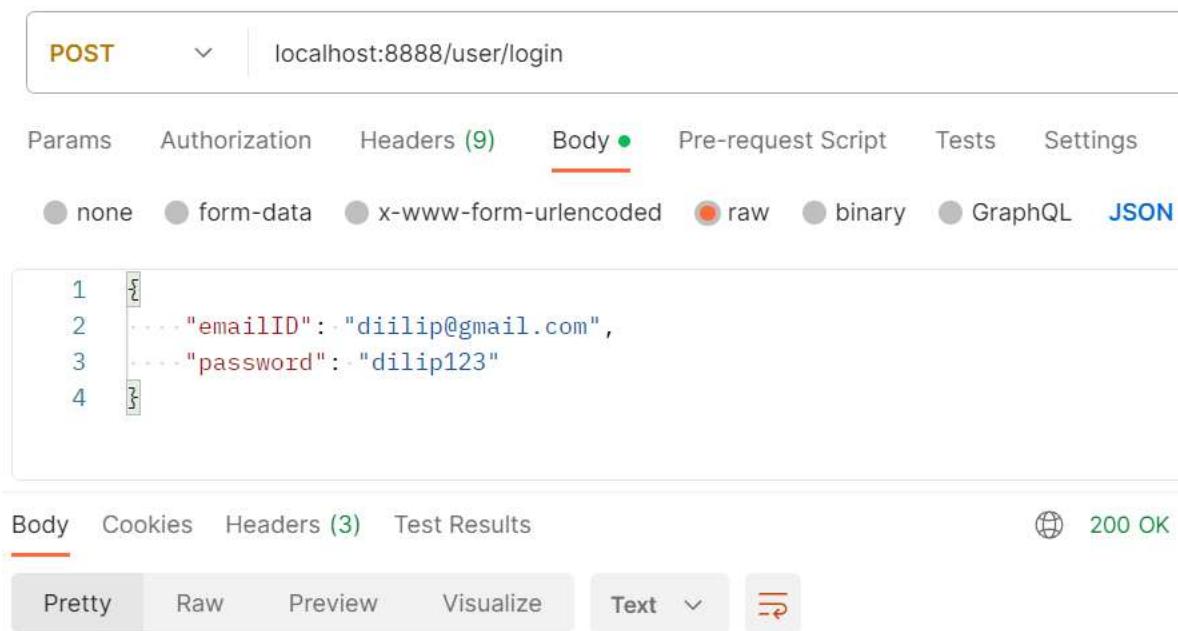
#order api mapping
spring.cloud.gateway.routes[1].id=order
spring.cloud.gateway.routes[1].uri=http://localhost:8002
spring.cloud.gateway.routes[1].predicates[0]=Path=/order/**

#payment api mapping
spring.cloud.gateway.routes[2].id=payment
spring.cloud.gateway.routes[2].uri=http://localhost:8003
spring.cloud.gateway.routes[2].predicates[0]=Path=/payment/**
```

With above Configuration, we can access our MicroServices with below URL patterns i.e. via Gateway Application.



Testing: Testing User Login Endpoint with gateway Port.



POST localhost:8888/user/login

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```

1  {
2    "emailID": "diilip@gmail.com",
3    "password": "dilip123"
4  }

```

Body Cookies Headers (3) Test Results 200 OK

Pretty Raw Preview Visualize Text

1 diilip@gmail.com

Now Our Gateway Application Is Ready.

Service Registry and Discovery:

Service registry and discovery are two important concepts in microservices architecture. Service registry is a centralized repository that stores information about all the microservices in a system. This information includes the microservice's name, address, and port number. Service discovery is the process of finding the location of a microservice based on its name.

Spring Boot provides support for service registry and discovery using the Netflix Eureka project. Eureka is a service registry that provides a REST API for registering and discovering microservices.

When a Spring Boot application registers with Eureka, it provides the following information:

- The application's name
- The application's address
- The application's port number
- The application's health status

Other Spring Boot applications can discover the location of a microservice by querying Eureka. The Eureka REST API provides a number of different endpoints for querying the service registry.

Service registry and discovery are essential for microservices architecture. They allow microservices to find each other and communicate with each other. This makes it possible to build large, complex applications that are composed of many small, independent microservices.

Here are some of the benefits of using service registry and discovery in Spring Boot microservices:

Scalability: Service registry and discovery make it easy to scale microservices. When you need to add more instances of a microservice, you simply register the new instances with Eureka. Eureka will then distribute requests to the new instances.

Fault tolerance: Service registry and discovery make it easy to handle failures in microservices. If a microservice fails, Eureka will remove it from the service registry. This will prevent other microservices from trying to communicate with the failed microservice.

Load balancing: Service registry and discovery can be used to load balance requests across multiple instances of a microservice. This can improve the performance of microservices by distributing requests evenly across the available instances.

Service Registry:

A Service Registry is a centralized directory where microservices can register themselves and provide metadata about their location and capabilities. In other words, it acts as a database of all available services in the system. Each microservice registers its network location (IP address and port) and any other relevant information (e.g., service name, version, health status) with the registry.

In Spring Boot, **Netflix Eureka** is a popular service registry implementation. Eureka provides a server component to run the registry and a client library to enable microservices to register themselves and discover other services.

Service Discovery:

Service Discovery is the process by which a microservice finds the network location (IP address and port) of other services it wants to communicate with. Instead of hardcoding the locations of other services, microservices can dynamically discover them through the service registry.

In Spring Boot, the Eureka client library is used to implement service discovery. When a microservice starts up, it registers itself with the Eureka server. Likewise, when a microservice needs to communicate with another service, it queries the Eureka server to get the location information of the required service.

Step-by-step explanation of how Service Registry and Discovery work in Spring Boot:

Service Registration:

1. Each microservice (e.g., Service A, Service B) running in the system registers itself with the Service Registry (Eureka Server).
2. The registration typically occurs when a microservice starts up. It sends a registration request to the Eureka server, providing its metadata (service name, IP address, port, health status, etc.).
3. The Eureka server maintains a registry of all the registered services and their metadata.

Service Discovery:

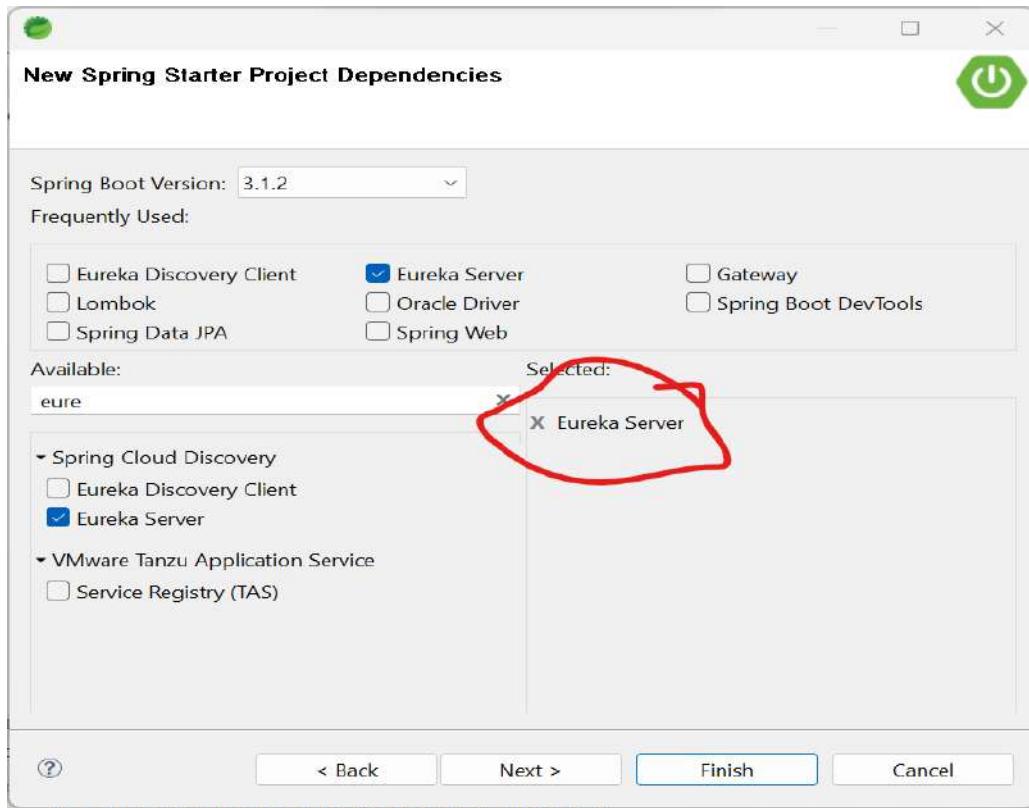
- When a microservice (e.g., Service A) needs to communicate with another service (e.g., Service B), it uses the Eureka client library to perform service discovery.
- The Eureka client in Service A queries the Eureka server to get the network location (IP address and port) of Service B.
- The Eureka server responds with the location information for Service B.
- Service A can then use this location information to communicate with Service B over the network.

Spring Boot provides excellent integration with Eureka through the **spring-cloud-starter-netflix-eureka-client** dependency for client-side service discovery and the **spring-cloud-starter-netflix-eureka-server** dependency for setting up the Eureka server.

It's worth noting that while Eureka is one of the popular choices for Service Registry and Discovery in Spring Boot, there are other alternatives like Consul, ZooKeeper etc..

Create Our Own Eureka Server for Services Registry:

Create SpringBoot Application With Eureka Server Dependency as followed.



- ⊕ After Application Created, Please add an annotation with `@SpringBootApplication` in main Spring Boot Application class, we need to add `@EnableEurekaServer` annotation.
- ⊕ The `@EnableEurekaServer` annotation is used to make your Spring Boot application acts as a Eureka Server.

```
package com.swiggy.eureka;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
public class SwiggyEurekaServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(SwiggyEurekaServerApplication.class, args);
    }

}
```

- ⊕ By default, the Eureka Server registers itself into the discovery. You should add the below given configuration into your application.properties file or application.yml file.

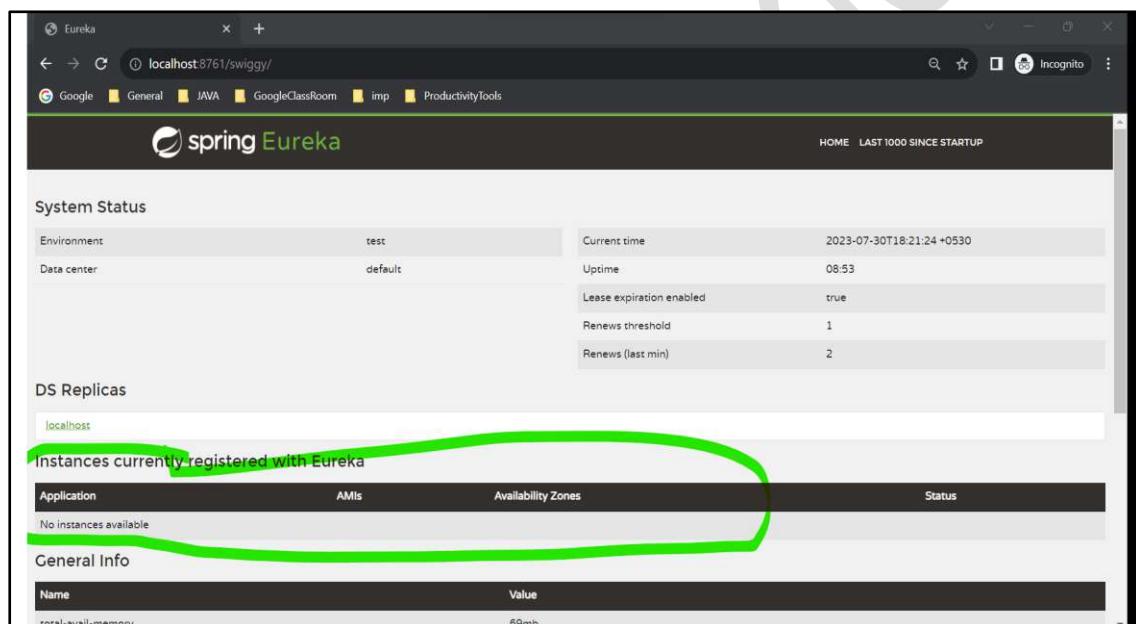
```
server.port=8761
server.servlet.context-path=/swiggy

#stopping self registration as a client again
eureka.client.register-with-eureka=false
#Fetching other Micro Services registration
eureka.client.fetch-registry=false
```

Now you can start Eureka Server Application. It will be started on port 8761 always but we can provide any context path.

Once Spring Cloud Eureka Server is started , Please access below URL for Discovered Services of MicroServices.

Eureka Server URL : <http://localhost:8761/swiggy/>



In Highlighted section we are seeing as **No Instances available**.

Now, It's time to make our Micro Services as Discovery Clients to register with Eureka Server.

In our MicroServices Project, we should make 3 Micro Services as Eureka Clients.

- User
- Order
- Payment

Step 1 : We should add a dependency in our existing Micro Services Applications or add while creating application.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2022.0.3</version>
</dependency>
```

Step 2 : After adding it We should add an annotation in every MicroService Main Spring Boot Application class level a followed.

```
package com.swiggy.food;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@SpringBootApplication
public class UserApplication {

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

Step 3: Add Eureka Server Details in Client application Properties file to register with it.

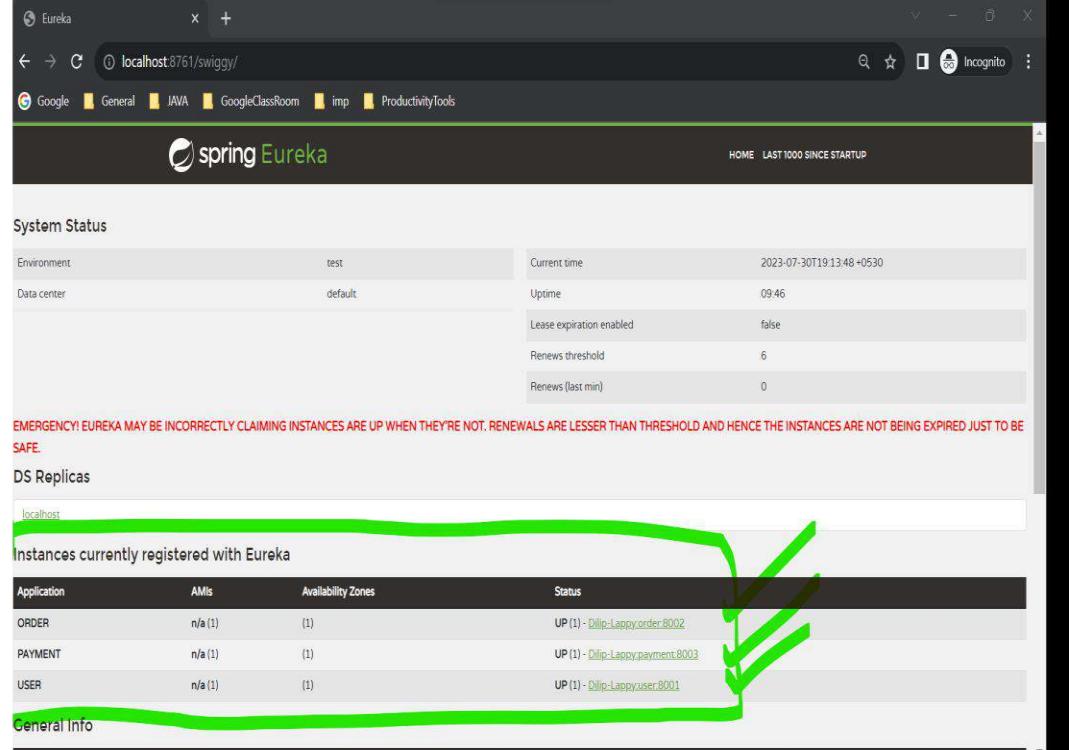
```
#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka
```

NOTE: These are basic steps for any Eureka client application. So please repeat same Steps for all Micro Services User, Order and Payment.

NOTE: Please start Eureka Server Application first and then Clients Applications.

- Start Eureka Server Application
- Now Start User, Order and Payment One by One.

Now Access Eureka Server URL again : <http://localhost:8761/swiggy/>



The screenshot shows the Spring Eureka UI on a browser. The top navigation bar includes tabs for Google, General, JAVA, GoogleClassRoom, imp, and ProductivityTools. The main content area is titled 'spring Eureka' and 'HOME: LAST1000 SINCE STARTUP'. The 'System Status' section displays various configuration parameters. A red warning message at the bottom states: 'EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.' The 'DS Replicas' section shows three registered instances: ORDER, PAYMENT, and USER, each with its status as 'UP (1) - Dilip-Lappy.order:8002', 'UP (1) - Dilip-Lappy.payment:8003', and 'UP (1) - Dilip-Lappy.user:8001' respectively. A green arrow points to the 'Instances currently registered with Eureka' table.

Now In Eureka Server our 3 MicroService Applications are Registered and same Details We can find above.

API Gateway Integration with Eureka Server:

With integration of Gateway application and Eureka Server, we can achieve auto routing to our Micro Service Instances. With the Discovery Locator enabled in API Gateway, you do not have to manually configure the routes of individual micro services unless absolutely needed. The way API Gateway knows which Eureka Service to route the incoming request.

There are several advantages of connecting Eureka server with gateway in microservices.

Centralized service discovery: Eureka server provides a centralized service discovery registry that allows the gateway to discover the microservices that are available. This makes it easy for the gateway to route requests to the appropriate microservices.

Load balancing: Eureka server can also be used to load balance requests across multiple instances of a microservice. This can help to improve the performance of the microservices architecture.

Fault tolerance: If a microservice instance fails, Eureka server will remove the instance from the registry. The gateway will then be able to route requests to the remaining instances of the microservice. This helps to ensure that the microservices architecture remains available even if some of the microservice instances fail.

Monitoring: Eureka server can also be used to monitor the health of the microservices. This information can be used to identify microservices that are not performing well or that are experiencing errors.

Overall, connecting Eureka server with gateway in microservices can provide a number of advantages, including centralized service discovery, load balancing, fault tolerance, and monitoring.

[Adding Discovery Client Functionalities to Gateway Application:](#)

As usual we will follow same Eureka Client Process as followed.

Step 1 : We should add a dependency in our existing Micro Services Applications or add while creating application.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2022.0.3</version>
</dependency>
```

Step 2 : After adding it We should add an annotation in every Gateway Main Spring Boot Application class level a followed.

```
package com.swiggy.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@SpringBootApplication
public class SwiggyGatewayApplication {

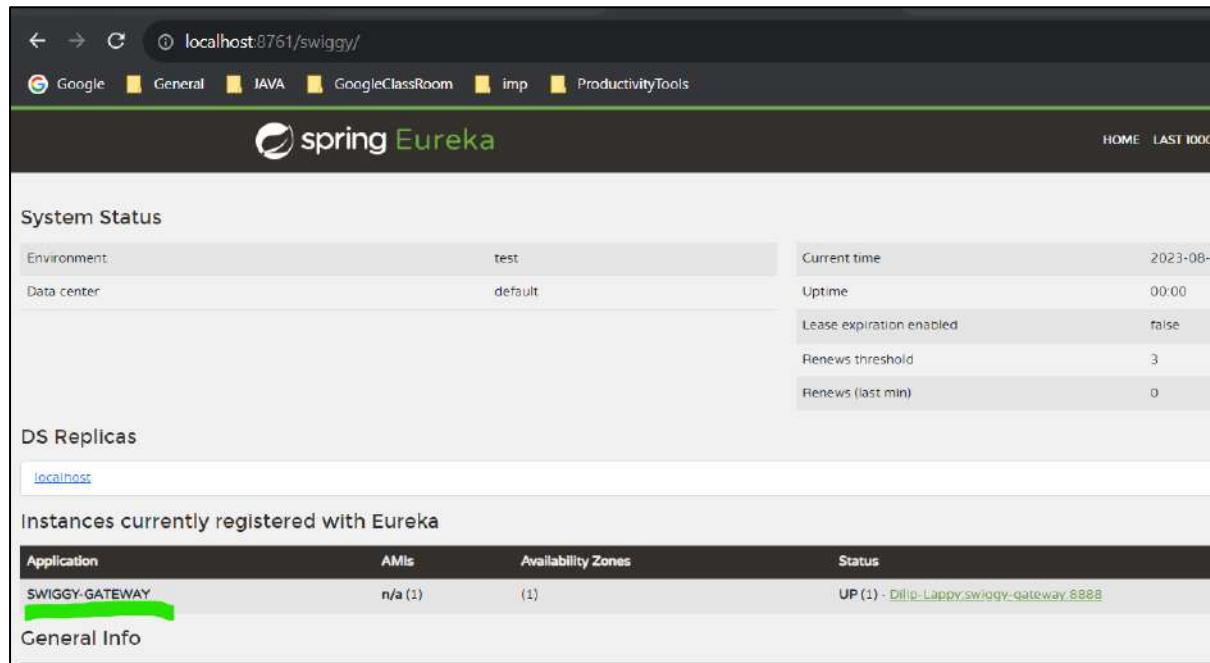
    public static void main(String[] args) {
        SpringApplication.run(SwiggyGatewayApplication.class, args);
    }
}
```

Step 3: Add Eureka Server Details in application.properties file to register with it.

#Eureka Server details

eureka.client.serviceUrl.defaultZone=<http://localhost:8761/swiggy/eureka>

With This changes in Gateway, It's ready to register and discoverable. Please Start Eureka Server and after Gateway application. We can find Gateway application details in Eureka server.



Environment	test	Current time	2023-08-01 10:00:00
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	3
		Renews (last min)	0

Application	AMIs	Availability Zones	Status
SWIGGY-GATEWAY	n/a (1)	(1)	UP (1) - Dilip-Lappy.swiggy-gateway:8080

Now We can trigger Micro Services with Gateway and Eureka Server. To access we should follow below URL format.

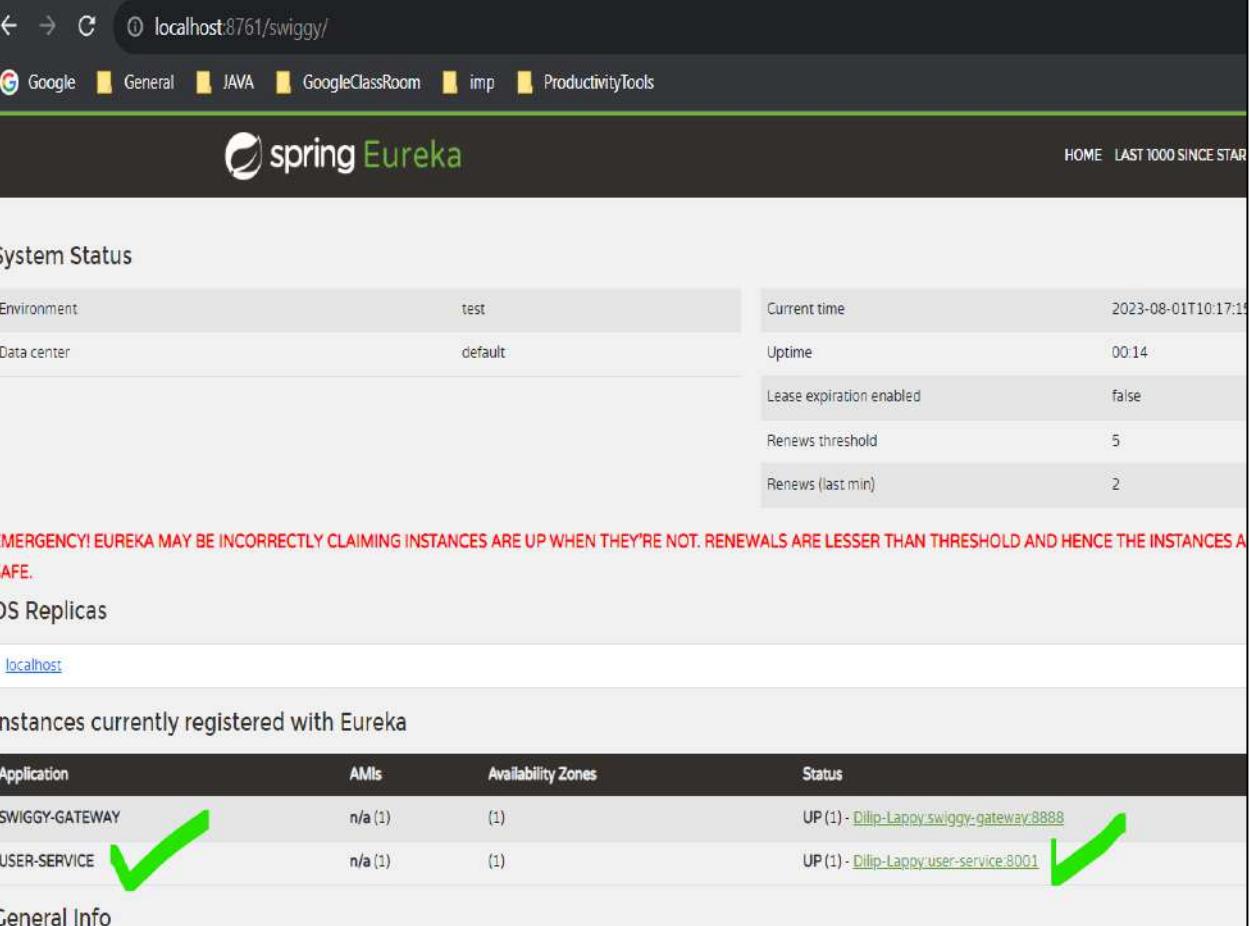
<http://{ApiGatewayHost}:{port}/{EurekaServiceId}/{ActualEndpoint}>

What is Eureka Service Id: The Micro Service Application Name what we given individually In every application.properties file. With those names every Eureka client application will be registered with Eureka Server.

For Example, For User MicroService provided name as

`spring.application.name=user-service`

After starting User Service, open Eureka server home page and we can see details as followed.



System Status

Environment	test	Current time	2023-08-01T10:17:11
Data center	default	Uptime	00:14
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	2

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE SAFE.

OS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
SWIGGY-GATEWAY	n/a (1)	(1)	UP (1) - Dilip-Lappy-swiggy-gateway:8888
USER-SERVICE	n/a (1)	(1)	UP (1) - Dilip-Lappy-user-service:8001

General Info

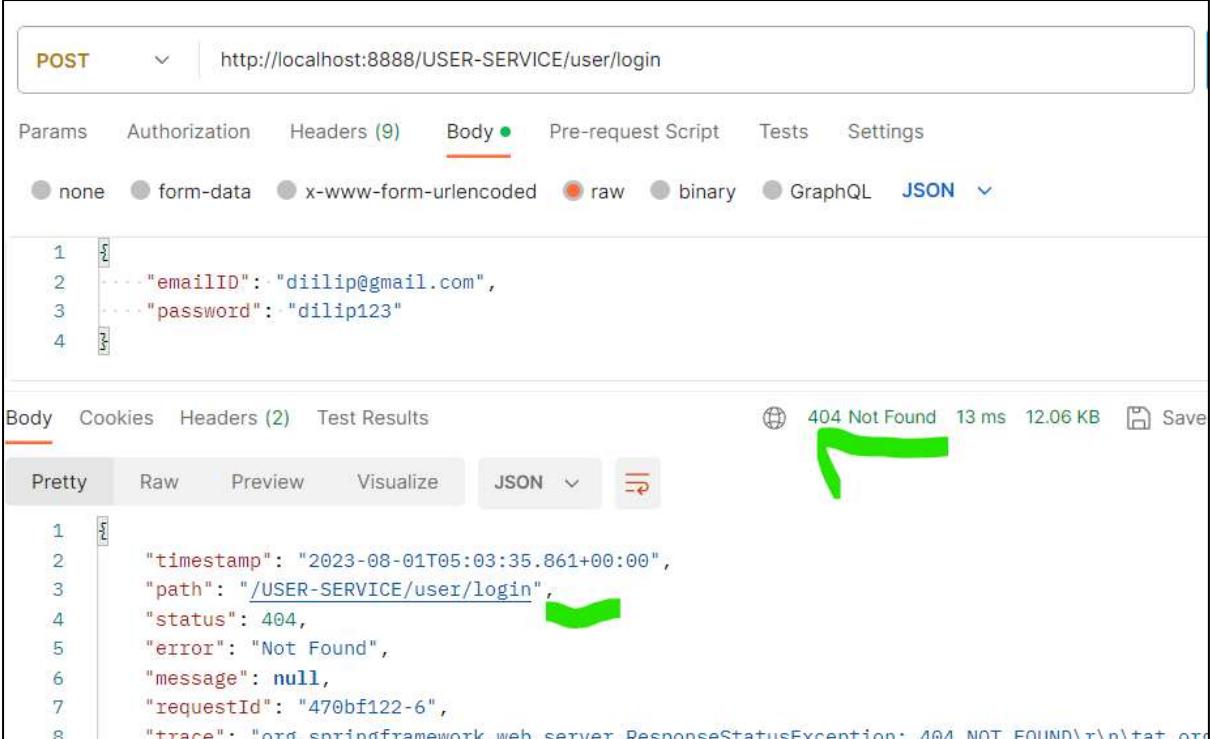
NOTE: Eureka sever will maintain Service ID'S in UPPERCASE always by default as shown in above image.

Similarly every Micro Service, will be provided with a name and if it registered with Eureka server, we will consider it as Eureka Service Id.

Access User Service With Gateway and Eureka Service ID:

Gateway : localhost:8888
Service Id : **USER-SERVICE**
Actual URI: /user/login

URL: <http://localhost:8888/USER-SERVICE/user/login>



The screenshot shows a Postman request to `http://localhost:8888/USER-SERVICE/user/login` using a POST method. The request body is a JSON object with `emailID` and `password` fields. The response status is 404 Not Found, with a timestamp of 2023-08-01T05:03:35.861+00:00, path `/USER-SERVICE/user/login`, and message `Not Found`.

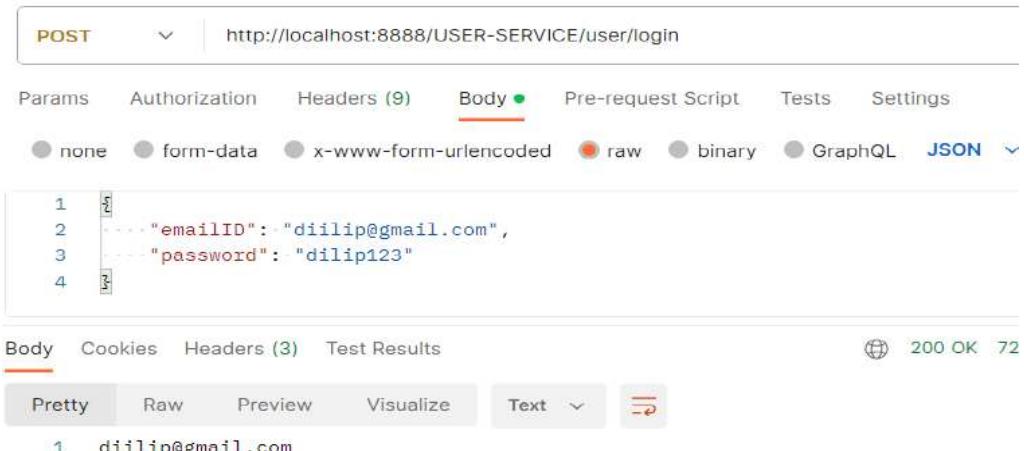
We got as **Not Found**. That means, by default Eureka server will not allow locating Micro Service details to route request internally.

To enable this functionality of locating Micro Services Information from Eureka Server we should add a property in **Gateway** application. With the Discovery Locator enabled in API Gateway, you do not have to manually configure the routes unless absolutely needed.

Add Below Property in Gateway Application Properties file.

```
spring.cloud.gateway.discovery.locator.enabled=true
```

Restart Gateway Application and try to access same URL again.



POST http://localhost:8888/USER-SERVICE/user/login

Body (JSON)

```

1 {
2   "emailID": "diilip@gmail.com",
3   "password": "dilip123"
4 }

```

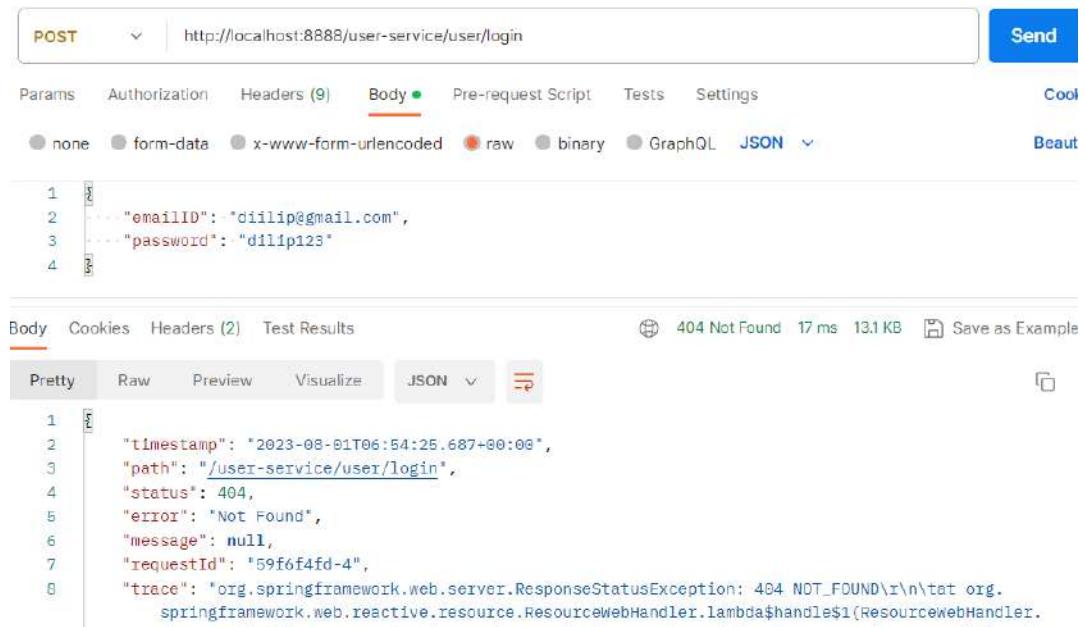
Body Cookies Headers (3) Test Results 200 OK 72

Pretty Raw Preview Visualize Text

1 diilip@gmail.com

Now Eureka Server allowing Gateway Application for locating micro service details based on Eureka Server Registered **Service ID** provided in URL. Now with that information, Gateway routes current request to User MicroService instances.

If we observe URL , Service ID given in Upper case. If we try to give lower case what will happen?



POST http://localhost:8888/user-service/user/login Send

Body (JSON)

```

1 {
2   "emailID": "diilip@gmail.com",
3   "password": "dilip123"
4 }

```

Body Cookies Headers (2) Test Results 404 Not Found 17 ms 13.1 KB Save as Example

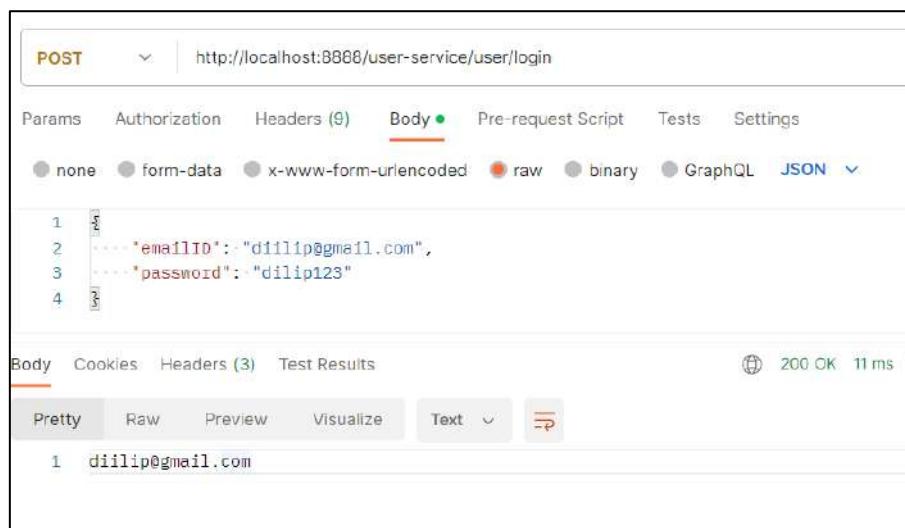
Pretty Raw Preview Visualize JSON

1 {
2 "timestamp": "2023-08-01T06:54:25.687+00:00",
3 "path": "/user-service/user/login",
4 "status": 404,
5 "error": "Not Found",
6 "message": null,
7 "requestId": "59f6f4fd-4",
8 "trace": "org.springframework.web.server.ResponseStatusException: 404 NOT_FOUND\r\n\tat org.springframework.web.reactive.resource.ResourceWebHandler.lambda\$1(ResourceWebHandler."

That means Eureka Server allowing only with Upper Case Service Id of A Micro Service always from Gateway Application by default. If we want to send service-id in Lower Case then we should add a property in Gateway allocation properties file.

spring.cloud.gateway.discovery.locator.lower-case-service-id=true

Now Access URL with Lower Case Service ID.



After lower case Property enabled, we can access only with Lower case Service Id but Not With Upper Case ID.

Gateway Integration with Eureka Server Advantages:

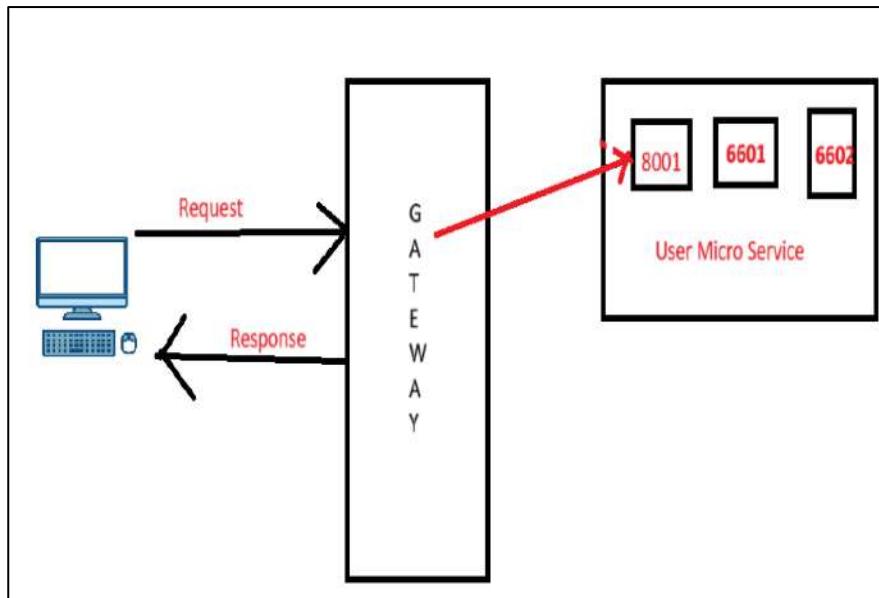
Case 1: Gateway Without Eureka Server:

Use Case : Assume User MicroService running on multiple port numbers like 8001 , 6601, 6602 etc..

Before Eureka Server implementation, we provided routing configuration in Gateway application for User MicroService to make sure Requests are always entered via Gateway to User Application.

```
spring.cloud.gateway.routes[0].id=user
spring.cloud.gateway.routes[0].uri=http://localhost:8001
spring.cloud.gateway.routes[0].predicates[0]=Path= /user/**
```

From above configuration, Request will be transferred to an instance running on port 8001 always by Gateway. **Then what about other instances running on 6601, 6602 etc. Ports. Those are not used anytime.**



To utilize all instances of a Micro Service application, then you have to do multiple routing configurations in Gateway for individual port numbers. But this is not looking good because in real time we may increase or decrease number of instances of Micro service application i.e. auto scaling.

Case 2: Gateway With Eureka Server Integration :

Use Case: Assume User MicroService running on multiple port numbers like 8001, 6601, 6602 etc..

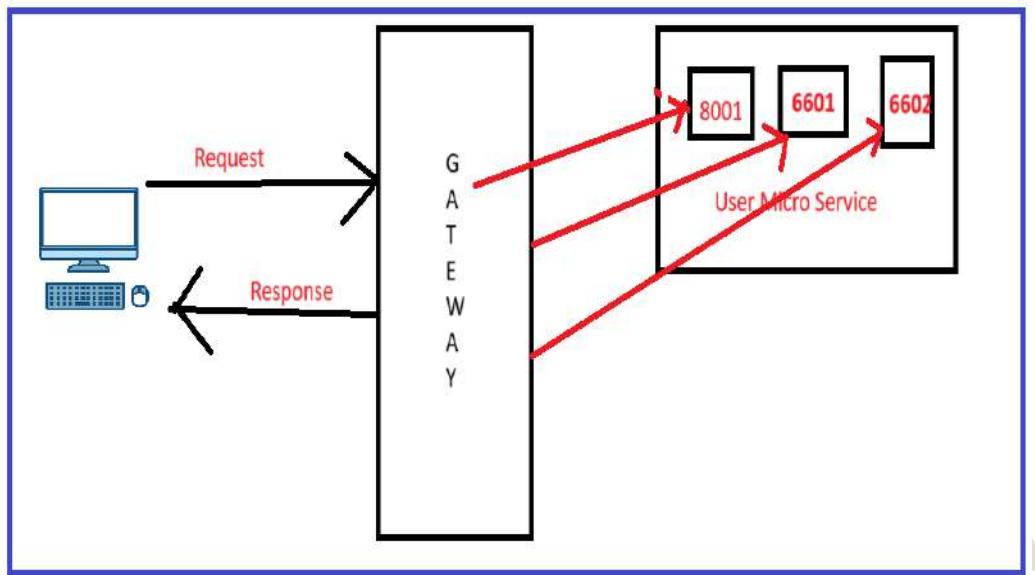
Now whenever we send a request to User Micro Service application via Gateway with Eureka Service Id as followed.

URL : <http://localhost:8888/USER-SERVICE/user/login>

Now Gateway application will locate all details of User MicroService application from Eureka Server like what are current running instances and port numbers. These Details are registered with Eureka Server while every Micro service instance we started.

In this Case, Gateway gets data of User MicroService application current running instances and port numbers like **8001, 6601, 6002** etc.. Now Gateway dynamically/automatically routes the Requests to less load instances internally. This auto routing will be taken care by load Balancer by default in Gateway.

Conclusion: Same functionality will be applicable for all Micro Services application in a project. Over all achievement now here is no manual configuration of routing in Gateway i.e. we can remove routing logic from Gateway applications for all Micro Services.



Micro Services Communication:

Feign Clients and **RestTemplate** both are Java libraries commonly used for making HTTP requests in microservices applications. They provide a way for your application to communicate with external services, APIs, or other microservices over the network. However, they have different approaches and features, which I'll explain below:

RestTemplate: RestTemplate is a synchronous HTTP client that is part of the Spring Framework. It provides a straightforward way to make HTTP requests to external services or APIs. You can use RestTemplate to send HTTP GET, POST, PUT, DELETE, and other types of requests.

Key features of RestTemplate:

Synchronous: RestTemplate operates in a synchronous manner, meaning that when you make a request, your application waits for the response before continuing execution.

Blocking: Since RestTemplate is synchronous, making multiple requests concurrently can lead to blocking behaviour and potentially reduced performance.

Flexibility: RestTemplate offers a variety of methods to customize request headers, parameters, and handling of responses.

Widely Used: RestTemplate has been a popular choice for making HTTP requests in Spring-based applications.

Feign Client: Feign is a declarative web service client also **developed by Spring Cloud**. It simplifies the process of making HTTP requests by allowing you to define interfaces with annotated methods that describe the API endpoints you want to call. Feign dynamically

generates the implementation for these interfaces, abstracting away the actual HTTP request details.

Key features of Feign Client:

Declarative: With Feign, you define an interface with annotations, and Feign generates the implementation automatically. This promotes a more concise and clean approach to API consumption.

Integration with Service Discovery: Feign can integrate seamlessly with service discovery mechanisms, such as Netflix Eureka, making it easy to call other microservices without hardcoding URLs.

Load Balancing: Feign can work in conjunction with load balancers, distributing requests across multiple instances of a service.

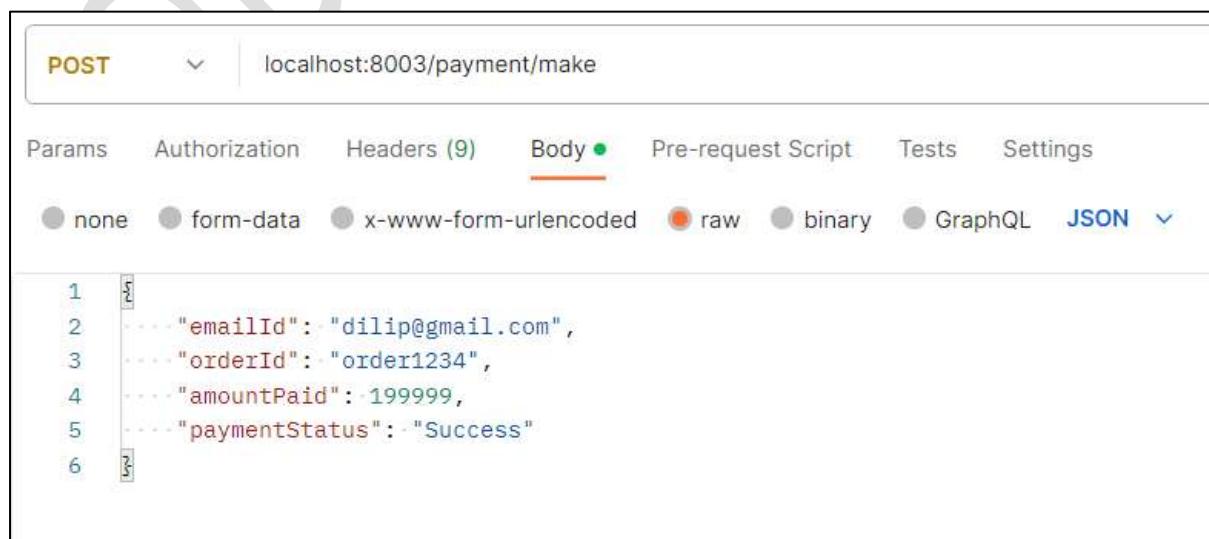
Integration with Spring Cloud: Feign is often used in Spring Cloud-based applications as part of the broader microservices ecosystem.

In summary, both Feign Client and RestTemplate are viable options for making HTTP requests in a microservices architecture. The choice between them depends on your project's specific requirements, development style, and whether you prefer a more declarative approach (Feign) or a more traditional, programmatic approach (RestTemplate). Additionally, consider factors like integration with service discovery, load balancing, and asynchronous support when making your decision.

Earlier As Part of Spring Boot Training, We Used RestTemplate for consuming micro services. Now We will continue with **Feign Clients**.

Requirement : Consume Payment MicroService REST Services From User Services.

REST API call of Payment Service:



POST localhost:8003/payment/make

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```

1  {
2    "emailId": "dilip@gmail.com",
3    "orderId": "order1234",
4    "amountPaid": 199999,
5    "paymentStatus": "Success"
6  }

```

Consumer : User Service
Producer: Payment Services

So, Now we should make changes in User Micro Service.

How To Enable Feign Clients:

Add Dependencies: In **pom.xml** file, add necessary dependency for Feign Client

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Enable Feign Client: In your main application class, add the **@EnableFeignClients** annotation to enable Feign Client functionality in Micro Service.

```
package com.swiggy.food;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.openfeign.EnableFeignClients;

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

Define Feign Client Interface: Create an interface that defines the API endpoints you want to call. Annotate the interface with **@FeignClient** and provide the **name of the service** you're communicating with. Also, define the methods corresponding to the endpoints you want to interact with. Similar or Looks Like to Controller Layer Endpoint Methods.

As per Payment Service, we should create Request Body class and we should define same in Feign client level.

```
package com.swiggy.food.feign.clients;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
```

```

import org.springframework.web.bind.annotation.RequestMethod;
import com.swiggy.food.request.PaymentDetails;

//Service ID/Name Producer

@FeignClient("payment-service")
public interface PaymentMicroServiceFeignCleint {

    @RequestMapping(method = RequestMethod.POST, value = "/payment/make")
    public String makePayment(@RequestBody PaymentDetails paymentDetails);

}

```

From the above Feign Client Configuration of REST service of Payment, we have Request Body class. So Create a POJO of request to pass the values with properties given in Postman.

```

@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class PaymentDetails {
    private String emailId;
    private String orderId;
    private double amountPaid;
    private String paymentStatus;
}

```

With this we are done with Feign Client Setup for Payment Service Endpoint. If we want to trigger another endpoint of Payment Micro Service then we will add another abstract method in Feign Client Interface.

Now Test It From User Service i.e. trigger Payment Endpoint of Feign Client.

I am creating an endpoint in User Controller with Request Body and same Request data I will forward to Request Body of Payment endpoint.

4. Autowire Feign Client in Controller
5. Call Feign Client Method with Request Data

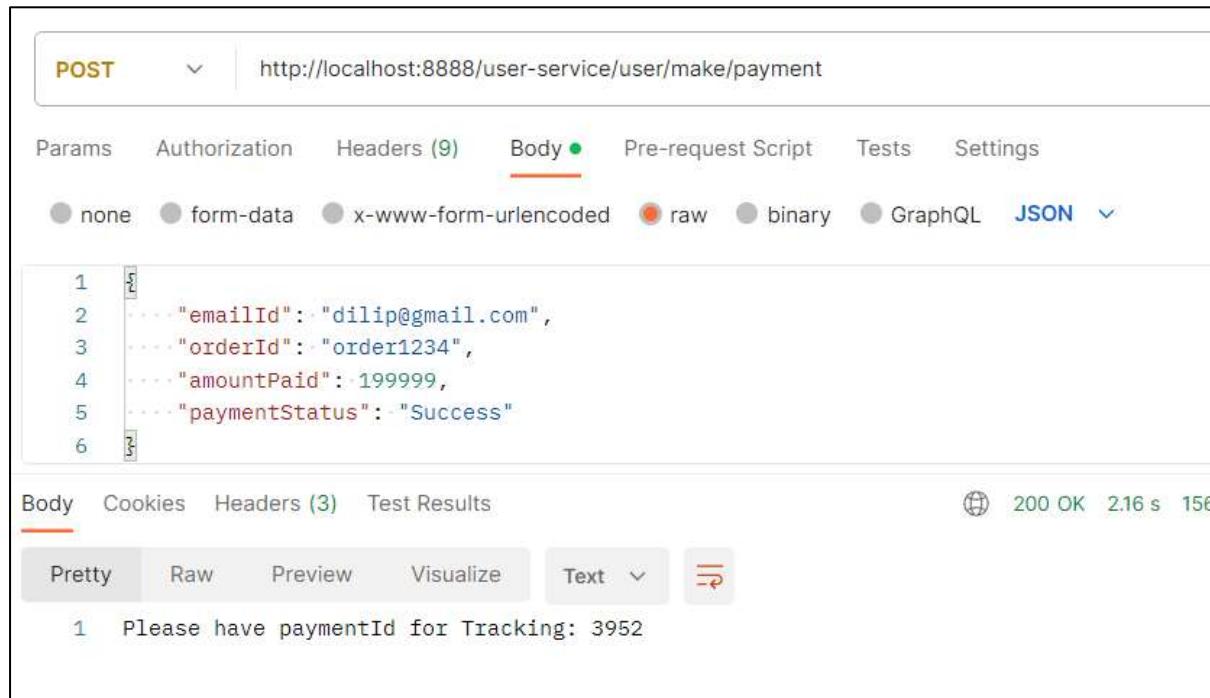
```

@Autowired
PaymentMicroServiceFeignCleint paymentClinet;

@PostMapping("/make/payment")
public String paymentStatus(@RequestBody PaymentDetails details) {
    return paymentClinet.makePayment(details);
}

```

Now Trigger Request to User Controller.



POST <http://localhost:8888/user-service/user/make/payment>

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```

1 {
2   "emailId": "dilip@gmail.com",
3   "orderId": "order1234",
4   "amountPaid": 199999,
5   "paymentStatus": "Success"
6 }
```

Body Cookies Headers (3) Test Results

200 OK 2.16 s 156

Pretty Raw Preview Visualize Text

1 Please have paymentId for Tracking: 3952

We got Response From Payment Endpoint internally and same forwarded as response to Client.

Internally Feign Client Implementation will trigger Payment Service Endpoint. Feign Client Logic internally Connected to Eureka Server with Service name what we configured with Feign Client. Eureka server will provide all available instances of Payment Mirco Services like hostname and port details. Finally Feign client contains all details of Payment Service including URI of endpoint. Internally Feign client triggers request with Full URL of endpoint.

Spring Cloud Config Server & Config Clients:

Spring Cloud **Config Server** and **Config Clients** are two components of Spring Cloud Config, which is a Spring Boot starter that provides server-side and client-side support for externalized configuration in a distributed system.

Spring Cloud Config Server and Config Clients are a powerful way to manage configuration properties in a distributed system. They make it easy to store configuration properties in a central location, and they allow you to consume configuration properties from your applications without having to store them in individual application configuration files.

The Config Server is a centralized repository for configuration properties that can be accessed by applications throughout the system. This makes it easy to manage configuration

properties for all of your applications in one place, and it also makes it easy to change configuration properties without having to redeploy your applications.

The Config Client is a Spring Boot starter that allows applications to consume configuration properties from the Config Server. This means that your applications can get their configuration properties from a central location, rather than having to store them in individual application configuration files.

There are two ways to configure the Spring Cloud Config Server:

- **Git:** The Config Server can be configured to store configuration properties in a Git repository. This is the most common way to configure the Config Server, as it allows you to version your configuration properties and easily roll back to a previous version if necessary.
- **JDBC:** The Config Server can also be configured to store configuration properties in a JDBC database. This is less common than the Git configuration, but it can be a good option if you already have a JDBC database that you want to use for configuration.

Once the Config Server is configured, you can start consuming configuration properties from it in your MicroService applications. To do this, you need to add the Spring Cloud Config Client starter to your client application's pom.xml or build.gradle file. You also need to specify the URL of the Config Server in your client application's configuration.

Once you have configured the Config Client, your application will be able to consume configuration properties from the Config Server. The configuration properties will be loaded into the Spring Environment, and you can access them using the @Value annotation as well.

Here are some of the benefits of using Spring Cloud Config Server and Config Clients:

- **Centralized configuration:** All of your configuration properties can be stored in a central location, making it easy to manage them.
- **Version control:** Configuration properties can be versioned in the same way as your application code, making it easy to roll back to a previous version if necessary.
- **Dynamic configuration:** Configuration properties can be changed at runtime, without having to redeploy your applications.
- **Support for multiple environments:** The same configuration properties can be used for different environments, such as development, staging, and production.
- **Easy to use:** Spring Cloud Config Server and Config Clients are easy to configure and use.

If you are looking for a way to manage configuration properties in a distributed system, then Spring Cloud Config Server and Config Clients are a good option. They are powerful, flexible, and easy to use.

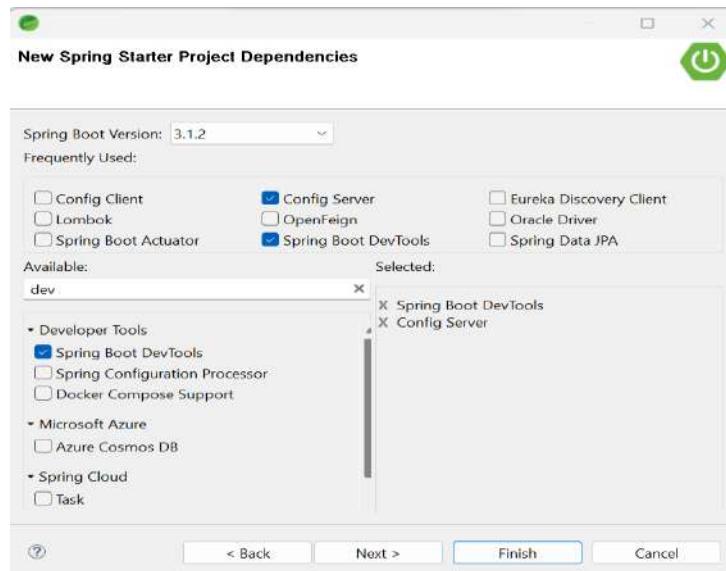
To set up Spring Cloud Config, you typically follow these steps:

- Create a Spring Boot application for the Config Server.

- Configure the Config Server to connect to a version-controlled repository (e.g., Git) where your configuration files are stored.
- Create Spring Boot applications for your microservices that will act as Config Clients.
- Configure the Config Clients to fetch their configuration from the Config Server.

Config Server Setup: With GitHub:

Step 1: Create Config Server with Below Dependency



Step 2: Add An annotation called as `@EnableConfigServer` on Spring Boot Application class level.

```
package com.swiggy.config;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

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

Step 3: Configure GitHub Details inside Config Server Properties file. By Default GitHub integration supported by Config Server.

Now We are chosen All configuration Properties should be available in Github Repository.

So create A Github Repository. Created Public Repository of below.

<https://github.com/dilipsingh1306/swiggy-config-server-data.git>

Now Configure the GitHub Repository Details in Config Server Properties File.

- **spring.cloud.config.server.git.uri** : URL of The GitHub Repository.
- **spring.cloud.config.server.git.skip-ssl-validation** : Skipping SSL Certificate Validation. The configuration server's validation of the Git server's SSL certificate can be disabled by setting property to **true** (default is false).
- **spring.cloud.config.server.git.default-label** : The default label i.e. Branch Name to be used with the remote repository.

application.properties

```
server.port=9988
server.servlet.context-path=/config
spring.application.name=config-server

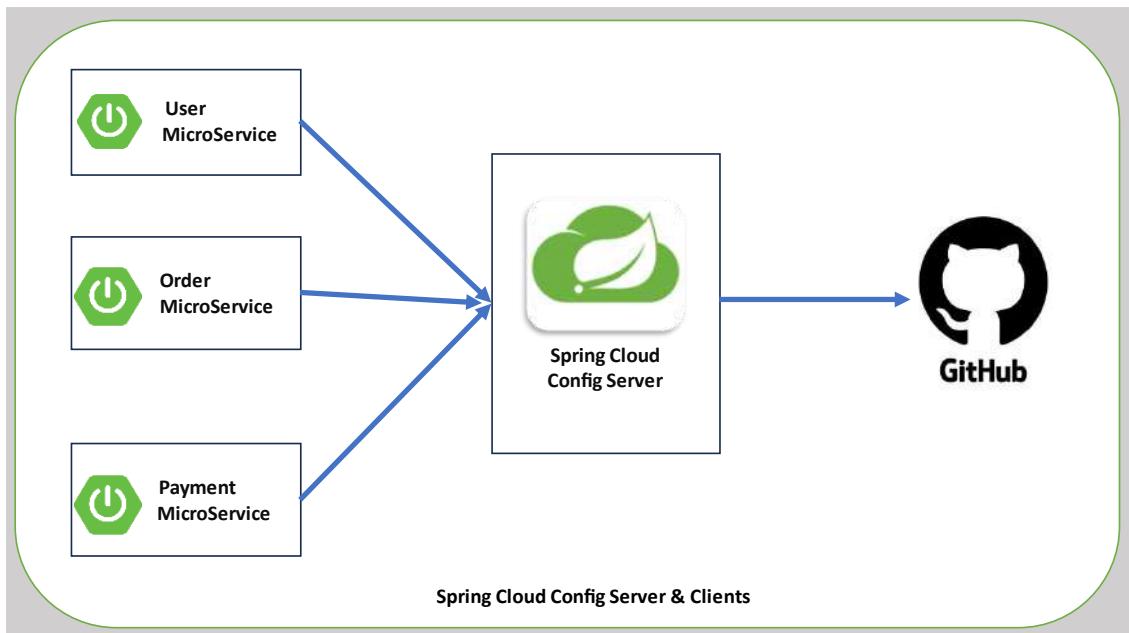
#github properties
spring.cloud.config.server.git.uri=https://github.com/dilipsingh1306/swiggy-config-server-data.git
spring.cloud.config.server.git.skip-ssl-validation=true
spring.cloud.config.server.git.default-label=master
```

Now We can Start Our Config Server. With this Configuration server Setup is completed with Backend as GitHub Repository.

Let's Work on How to Maintain Configuration of MicroServices Properties.

In our Case we have total three MicroServices **user, order and payment**. So we have to move Configuration from MicroService level to GitHub repositories and access via Config Server.

We should make **user, order and payment** as Config Clients as a first step.



Enabling Spring Cloud Config Clients to Our Micro Services:

Repeat Steps For All Three Micro Services.

Step 1: Add Dependency of Spring Cloud Config Client Starter to pom.xml file.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

Step 2: Add Config Server URL to MicroService application.properties file.

```
spring.config.import=optional:configserver:http://localhost:9888/config
```

That's all Our Three Micro Services are Ready to Fetch Configuration from Config Server.

Configuring MicroServices Properties inside Config Server:

Means we should Create Configuration Properties inside GitHub Repository. Those configuration properties pulled by Config Server internally and distributed to Respective MicroServices.

In general, We should maintain MicroServices Configuration as follows.

- Common Properties of All or Multiple MicroServices.
- MicroService Individual/Specific Properties.

Common Properties Configuration:

As per Config Server, Common Properties Should be maintained inside **application.properties** file and that should be created inside GitHub Repository.

First Create **application.properties** file inside GitHub. This properties File is common for all Micro Services Configuration i.e. We have to maintain properties only common across multiple MicroServices.

Current Configuration of Our Micro Services:

User MicroService

```
#Project Info
server.port = 8001
server.servlet.context-path=/user
spring.application.name=user-service

#Database
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka

spring.config.import=optional:configserver:http://localhost:9988/config
```

Order MicroService:

```
#Project Info
server.port = 8002
server.servlet.context-path=/order
spring.application.name=order-service

#Database
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka

spring.config.import=optional:configserver:http://localhost:9988/config
```

Payment MicroService:

```

#Project Info
server.port = 8003
server.servlet.context-path=/payment
spring.application.name=payment-service

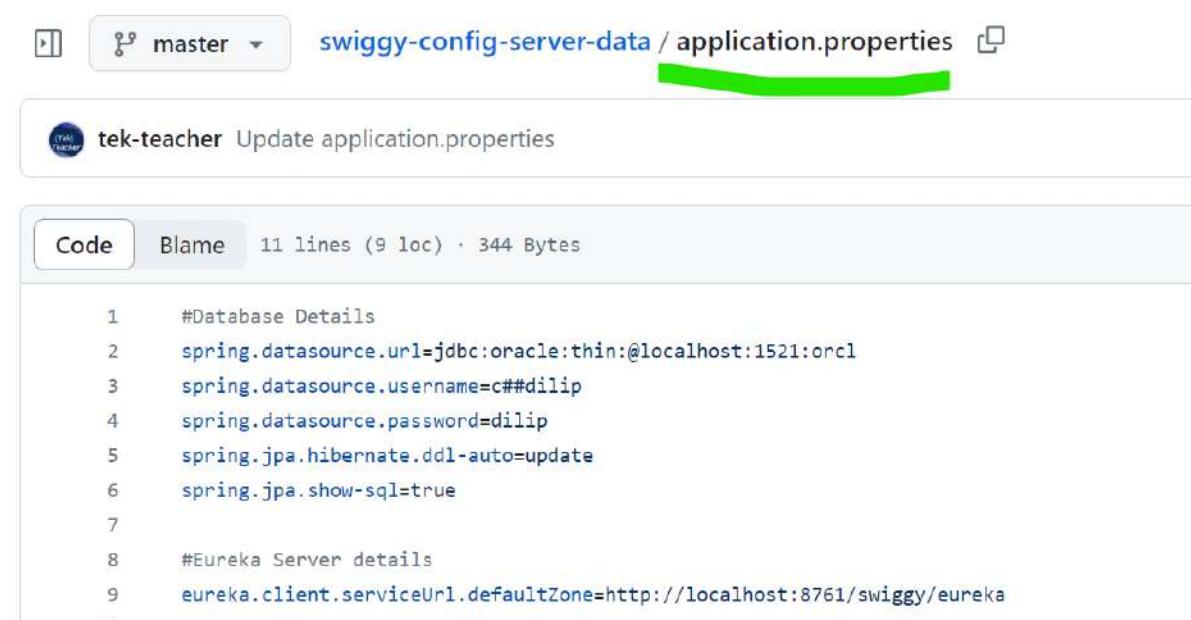
#Database
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka

spring.config.import=optional:configserver:http://localhost:9988/config

```

From Above Three MicroServices Properties, If we observe **Database and Eureka Server Details are common across apart from Config Server Property.** These Type of data should be maintained/moved inside **application.properties** in GitHub Repository which is integrated with Cloud Config Sever.



Code

```

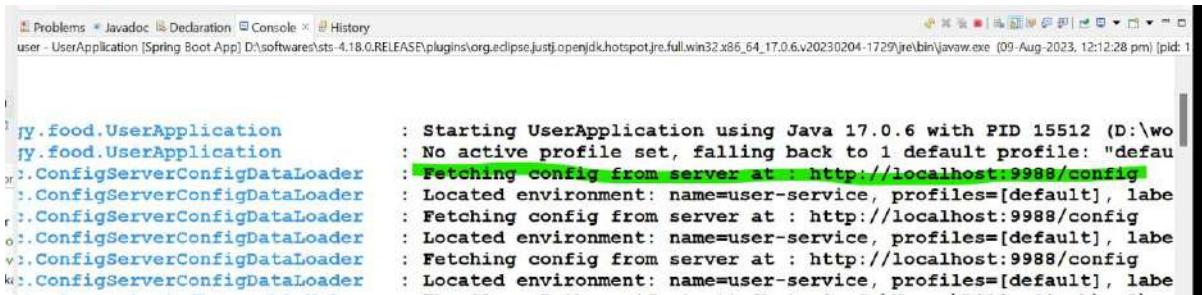
1 #Database Details
2 spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
3 spring.datasource.username=c##dilip
4 spring.datasource.password=dilip
5 spring.jpa.hibernate.ddl-auto=update
6 spring.jpa.show-sql=true
7
8 #Eureka Server details
9 eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka

```

Delete Same Properties from Individual Micro services and try to start up Micro Services.

Note: Please Start Config Server First and then MicroServices i.e. Config Clients.

While Starting Config Clients i.e. MicroServices in Console Logs printed as Configuration Fetching From Config Server and Details.



```

    : Starting UserApplication using Java 17.0.6 with PID 15512 (D:\wo
    : No active profile set, falling back to 1 default profile: "defau
    : Fetching config from server at : http://localhost:9988/config
    : Located environment: name=user-service, profiles=[default], labe
    : Fetching config from server at : http://localhost:9988/config
    : Located environment: name=user-service, profiles=[default], labe
    : Fetching config from server at : http://localhost:9988/config
    : Located environment: name=user-service, profiles=[default], labe
  
```

All 3 Micro services are Up and Running, that means Database and Eureka Server Details Are fetched from Config Server.

So in future If we have any common configuration properties and data either predefined or user-defined we have to Define inside **application.properties** so that all MicroServices will fetch same.

MicroServices Individual/Specific Properties Configuration:

For example, Every MicroService will have it's own context path and Port number. In such case, we should not maintain those properties inside **application.properties** because same context path and port will be assigned to all micro services while starting and fails.

For these kind of Scenarios, like property is same but value is individual to each MicroService then Config Server provided an option like creating individual properties files with application/service names.

Now we have to create 3 properties files with Micro Service name(i.e. value of **spring.application.name**) what we provided inside the properties file.

Current Configuration of Our Micro Services:

User MicroService

```

#Project Info
server.port = 8001
server.servlet.context-path=/user
spring.application.name=user-service

spring.config.import=optional:configserver:http://localhost:9988/config
  
```

Order MicroService:

```

#Project Info
server.port = 8002
server.servlet.context-path=/order
spring.application.name=order-service
  
```

```
spring.config.import=optional:configserver:http://localhost:9988/config
```

Payment MicroService:

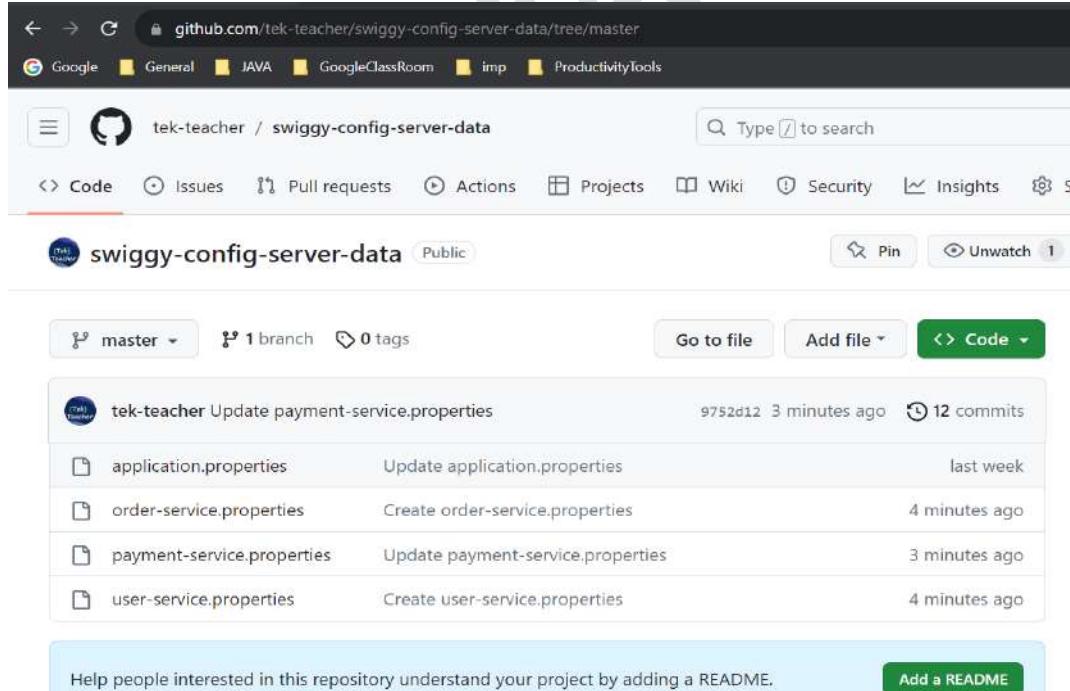
```
#Project Info
server.port = 8003
server.servlet.context-path=/payment
spring.application.name=payment-service

spring.config.import=optional:configserver:http://localhost:9988/config
```

From Above Three MicroServices Properties, If we observe **Port and Context Details are different.** These Type of data should be maintained/moved inside **<service-name>.properties** in GitHub Repository which is integrated with Cloud Config Sever.

So create 3 individual Micro Services properties files in side GitHub Repository. After that move properties of context path and port to GitHub properties files level.

- **user-service.properties**
- **order-service.properties**
- **payment-service.properties**



The screenshot shows a GitHub repository page for 'swiggy-config-server-data'. The repository is public and has 1 branch (master) and 0 tags. The commits are as follows:

- tek-teacher Update payment-service.properties 9752d12 3 minutes ago 12 commits
- application.properties Update application.properties last week
- order-service.properties Create order-service.properties 4 minutes ago
- payment-service.properties Update payment-service.properties 3 minutes ago
- user-service.properties Create user-service.properties 4 minutes ago

At the bottom, there is a message: "Help people interested in this repository understand your project by adding a README." and a "Add a README" button.

[swiggy-config-server-data / user-service.properties](#) 

tek-teacher Create user-service.properties

[Code](#)[Blame](#)

2 lines (2 loc) · 53 Bytes

```
1 server.port = 8001
2 server.servlet.context-path=/user
```

[swiggy-config-server-data / order-service.properties](#) 

tek-teacher Create order-service.properties

[Code](#)[Blame](#)

2 lines (2 loc) · 54 Bytes

```
1 server.port = 8002
2 server.servlet.context-path=/order
```

[swiggy-config-server-data / payment-service.properties](#) 

tek-teacher Update payment-service.properties

[Code](#)[Blame](#)

2 lines (2 loc) · 56 Bytes

```
1 server.port = 8003
2 server.servlet.context-path=/payment
```

Current Configuration of Our Micro Services in Local:**User MicroService**

```
spring.application.name=user-service
```

```
spring.config.import=optional:configserver:http://localhost:9988/config
```

Order MicroService:

```
spring.application.name=order-service
```

```
spring.config.import=optional:configserver:http://localhost:9988/config
```

Payment MicroService:

```
spring.application.name=payment-service
```

```
spring.config.import=optional:configserver:http://localhost:9988/config
```

Now Start Your Services in An Order.

- Config Server
- Eureka Server
- Gateway
- User
- Order
- Payment

Now we can access All Services REST Services with Gateway.

Can we define our own User Defined Properties in side Config Server i.e. GitHub Repository?

Yes, We can Define user defined properties inside **application.properties** or individual MicroService Properties File level. i.e. When we need a properties across multiple services level, then we have to define in common **application.properties** file level. If Properties are Specific to MicroServices, then define in individual Micro Service Properties file Level.

Profile Based Properties Files in Config Server:

application.properties	->	All Micro Services + All profiles
application-dev.properties	->	All Micro Services + Only DEV profile
application-sit.properties	->	All Micro Services + Only SIT profile
user.properties	->	User MicroService + All profiles
user-dev.properties	->	User MicroService + Only DEV profile
user-sit.properties	->	User MicroService + Only SIT profile
order.properties	->	Order MicroService + All profiles
order-dev.properties	->	Order MicroService + Only DEV profile
order-sit.properties	->	Order MicroService + Only SIT profile

payment.properties	->	Payment MicroService + All profiles
payment-dev.properties	->	Payment MicroService + Only DEV profile
payment-sit.properties	->	Payment MicroService + Only SIT profile

If a Property available in common, individual, profile based properties file of Micro Services, then which Property will be loaded dynamically?

1st Priority : Property Loaded from Micro Service Profile Based

2nd Priority : Property Loaded from Micro Service Properties file

3rd Priority : Property Loaded from Common Properties file

Circuit Breaker in MicroServices:

In a microservice architecture, it's common for a service to call another service. And there is always the possibility that the other service being called is unavailable or unable to respond. So, what can we do when this happens?

A circuit breaker is a pattern used to protect a system from cascading failures. It works by monitoring the number of failures for a particular operation and, if the failure rate exceeds a threshold, it stops making calls to that operation. This prevents the failure from spreading to other parts of the system. In Spring microservices, the circuit breaker pattern can be implemented using the Spring Cloud Circuit Breaker project. This project provides an abstraction layer across different circuit breaker implementations, so you can choose the one that best suits your needs.

This pattern comes into the picture while **communicating between services**. Let's take a simple scenario. Let's say we have two services: Service A and B. Service A is calling Service B(API call) to get some information needed. When Service A is calling to Service B, if Service B is down due to some infrastructure outage, what will happen? **Service A is not getting a result and it will be hang by throwing an exception**. Then another request comes and it also faces the same situation. Like this request threads will be blocked/hanged until Service B is coming up! As a result, the network resources will be exhausted with low performance and bad user experience. **Cascading failures** also can happen due to this.

In such scenarios, we can use this **Circuit Breaker pattern to solve the problem**. It is giving us a way to handle the situation without bothering the end user or application resources.

How the pattern works?

Basically, it will behave same as an electrical circuit breaker. **When the application gets remote service call failures more than a given threshold, circuit breaker trips for a particular time period. After this timeout expires, the circuit breaker allows a limited number of requests to go through it. If those requests are getting succeeded, then circuit breaker will**

be closed and normal operations are resumed. Otherwise, if they are failing, timeout period starts again and do the rest as previous.

Let's figure out this using the upcoming example scenario that I'm going to explain.

Life Cycle of Pattern States:

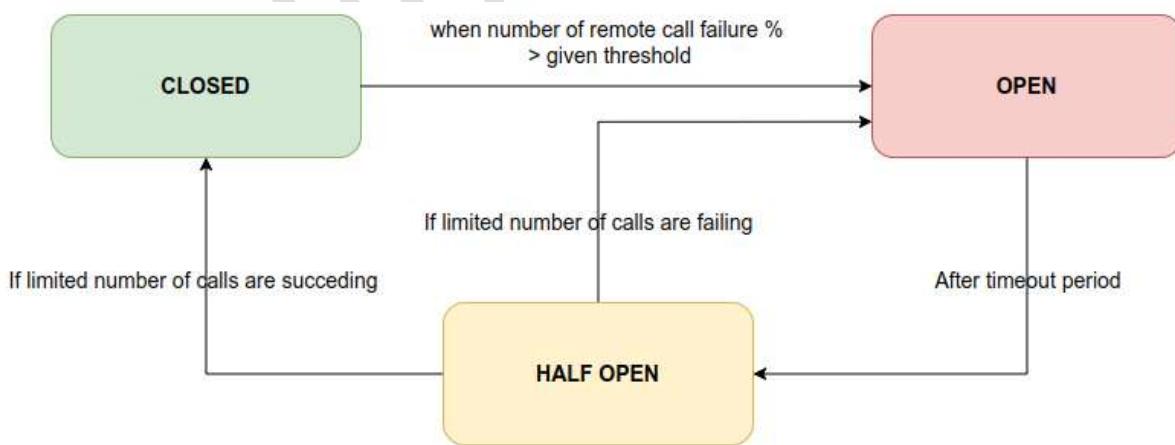
There are 3 main states discussed in Circuit Breaker pattern. They are:

1. CLOSED
2. OPEN
3. HALF OPEN

CLOSED: When both services which are interacting are up and running, circuit breaker is CLOSED. Circuit breaker is counting the number of remote API calls continuously.

OPEN: As soon as the percentage of failing remote API calls is exceeding the given threshold, circuit breaker changes its state to OPEN state. Calling micro service will fail immediately, and an exception will be returned. That means, the flow is interrupted.

HALF OPEN: After staying at OPEN state for a given timeout period, breaker automatically turns its state into HALF OPEN state. In this state, only a LIMITED number of remote API calls are allowed to pass through. If the failing calls count is greater than this limited number, breaker turns again into OPEN state. Otherwise it is CLOSED.



To demonstrate the pattern practically, I will use Spring Boot framework to create the micro services. Resilience4j library is used to implement the circuit breaker.

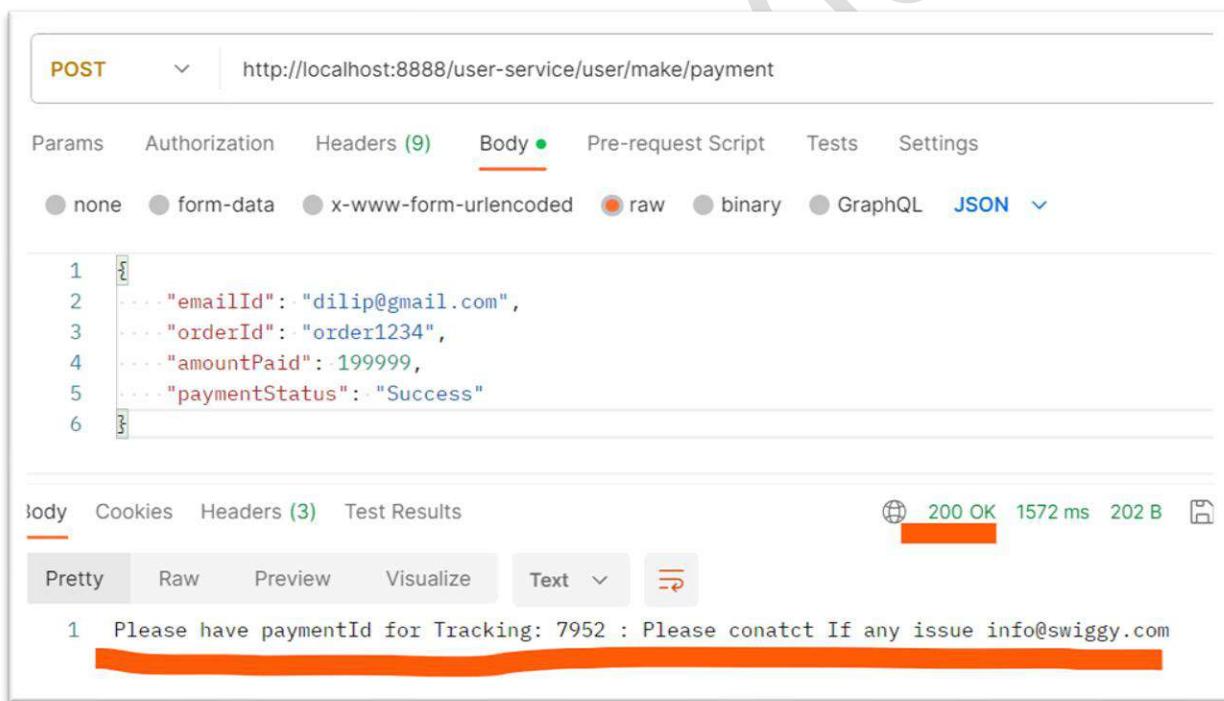
What is Resilience4j?

Resilience4j is a lightweight, easy-to-use fault tolerance library inspired by Netflix Hystrix. It provides various features. If you are looking for a lightweight and easy-to-use fault tolerance library for your Java application, Resilience4j is a good choice.

Here are some of the use cases of Resilience4j:

- To protect your application from cascading failures.
- To improve the availability of your application.
- To reduce the impact of failures on your application.
- To improve the performance of your application.
- To make your application more resilient to unexpected events.

- Now In Our existing Micro Services, User Service calling Payment Service API's. When Payment API Services are Good and working as expected, we get below Response.



The screenshot shows a Postman request for a POST method to the URL `http://localhost:8888/user-service/user/make/payment`. The request body is a JSON object:

```

1  {
2    "emailId": "dilip@gmail.com",
3    "orderId": "order1234",
4    "amountPaid": 199999,
5    "paymentStatus": "Success"
6  }

```

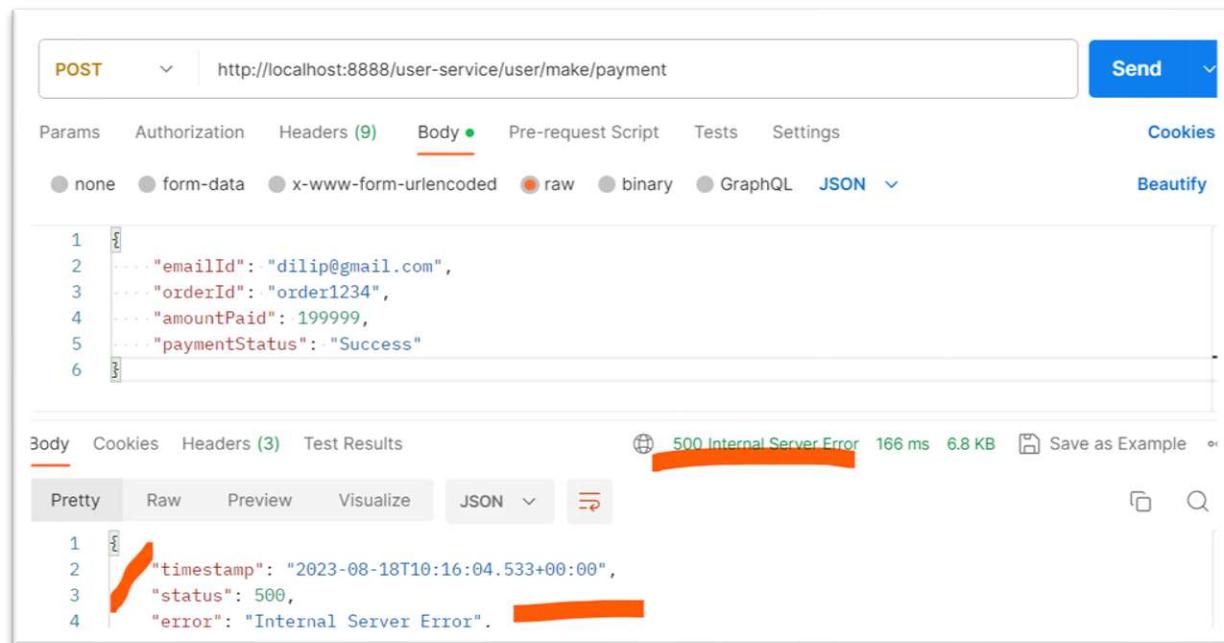
The response status is 200 OK, with a response time of 1572 ms and a response size of 202 B. The response body is:

```

1  Please have paymentId for Tracking: 7952 : Please conatct If any issue info@swiggy.com

```

- In case, Payment Service is Down then we will get an Exception as shown below.



POST http://localhost:8888/user-service/user/make/payment

Body (9) **Body** Pre-request Script Tests Settings Cookies

Params Authorization Headers (9) Body (9) Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```

1 {
2   "emailId": "dilip@gmail.com",
3   "orderId": "order1234",
4   "amountPaid": 199999,
5   "paymentStatus": "Success"
6 }

```

Body Cookies Headers (3) Test Results 500 Internal Server Error 166 ms 6.8 KB Save as Example

Pretty Raw Preview Visualize JSON

```

1 {
2   "timestamp": "2023-08-18T10:16:04.533+00:00",
3   "status": 500,
4   "error": "Internal Server Error".

```

In these cases, we should integrate or enable Circuit Breaker, to handle fault tolerance. This Breaker should be enabled in Consumer Service Side.

Steps To Enable Circuit Breaker:

- Add Below Starter Dependencies in Side User Service to enable Resilience4J Circuit Breaker.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>

```

Here, aop functionalities used by circuitbreaker-resilience4j internally so added. Similarly, actuator should be required to enable endpoint of circuit breaker to monitor circuit breaker state and statistics.

- Add Circuit Breaker Configuration in Properties of User Micro Service.

```

management.health.circuitbreakers.enabled=true
management.endpoints.web.exposure.include=health
management.endpoint.health.show-details=always

#Resilience Circuit Breaker
resilience4j.circuitbreaker.instances.user-service.registerHealthIndicator=true
resilience4j.circuitbreaker.instances.user-service.eventConsumerBufferSize=10
resilience4j.circuitbreaker.instances.user-service.failureRateThreshold=50
resilience4j.circuitbreaker.instances.user-service.minimumNumberOfCalls=5
resilience4j.circuitbreaker.instances.user-
service.automaticTransitionFromOpenToHalfOpenEnabled=true
resilience4j.circuitbreaker.instances.user-service.waitDurationInOpenState=6s
resilience4j.circuitbreaker.instances.user-
service.permittedNumberOfCallsInHalfOpenState=3
resilience4j.circuitbreaker.instances.user-service.slidingWindowSize=10
resilience4j.circuitbreaker.instances.user-service.slidingWindowType=COUNT_BASED

```

- Now Create Circuit Breaker and fall back method, where actually we are integrated Payment API Service from User Service.

What is fallback Method?

The **fallbackMethod** attribute in the **@CircuitBreaker** annotation specifies a method that will be called if the circuit breaker is open. This method is called the fallback method.

The fallback method can be used to provide a graceful degradation of service when the circuit breaker is open. For example, the fallback method could return a default value, or it could retry the call to the protected method a few times before giving up.

The fallback method must have the same method signature as the protected method, with the addition of one extra parameter: the exception that caused the circuit breaker to open.

The fallback method is a powerful tool that can be used to protect your application from cascading failures. By providing a fallback method, you can ensure that your application will continue to function even if the protected method fails.

Here are some of the things to keep in mind when using the fallback method:

- The fallback method should be lightweight and fast. It should not do anything that could cause the circuit breaker to open again.
- The fallback method should be used to provide a graceful degradation of service. It should not be used to provide the same level of service as the protected method.

UserController.java

```

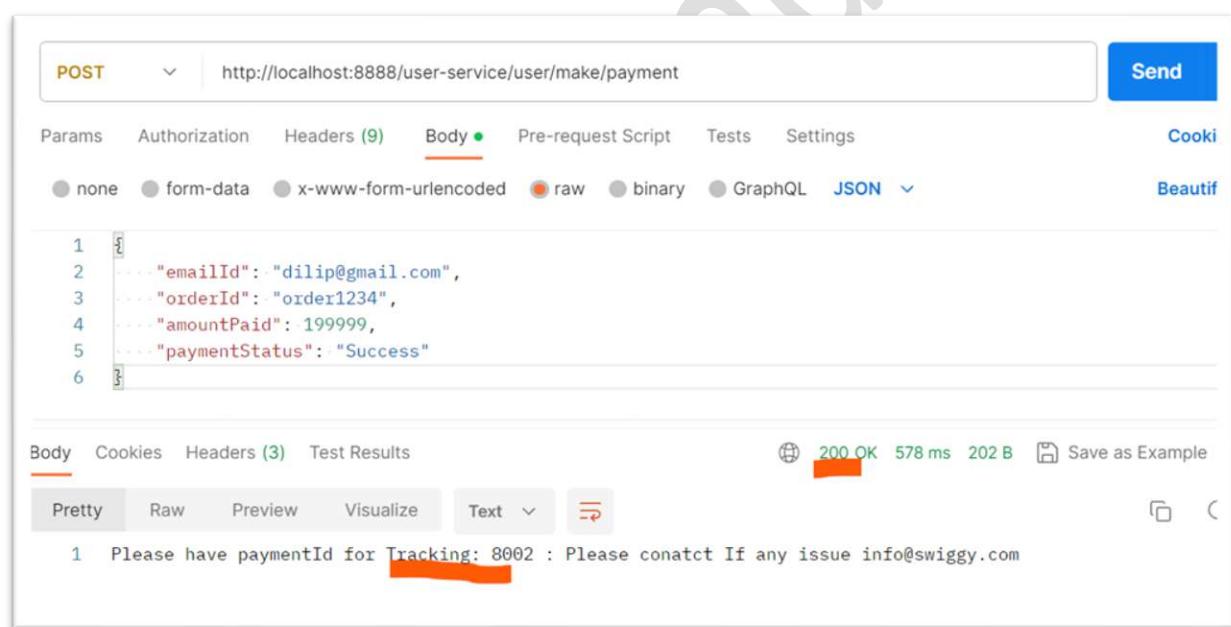
@PostMapping("/make/payment")
@CircuitBreaker( name="user-service" , fallbackMethod = "paymentStatus")
public String paymentStatus(@RequestBody PaymentDetails details) {
    return paymentClinet.makePayment(details) + " : Please contact If any issue " + emailId;
}

//fallbackMethod
public String paymentStatus(Throwable ex) {
    return "Backend Baking Systems are Not functioning. Please Contact Admin";
}

```

Now Test Above endpoint and check circuit breaker status.

When Payment Service is UP: Getting Actual response from Payment Service



POST <http://localhost:8888/user-service/user/make/payment> Send

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings Cookies Beautif

Body (Pretty, Raw, Preview, Visualize, Text, JSON)

```

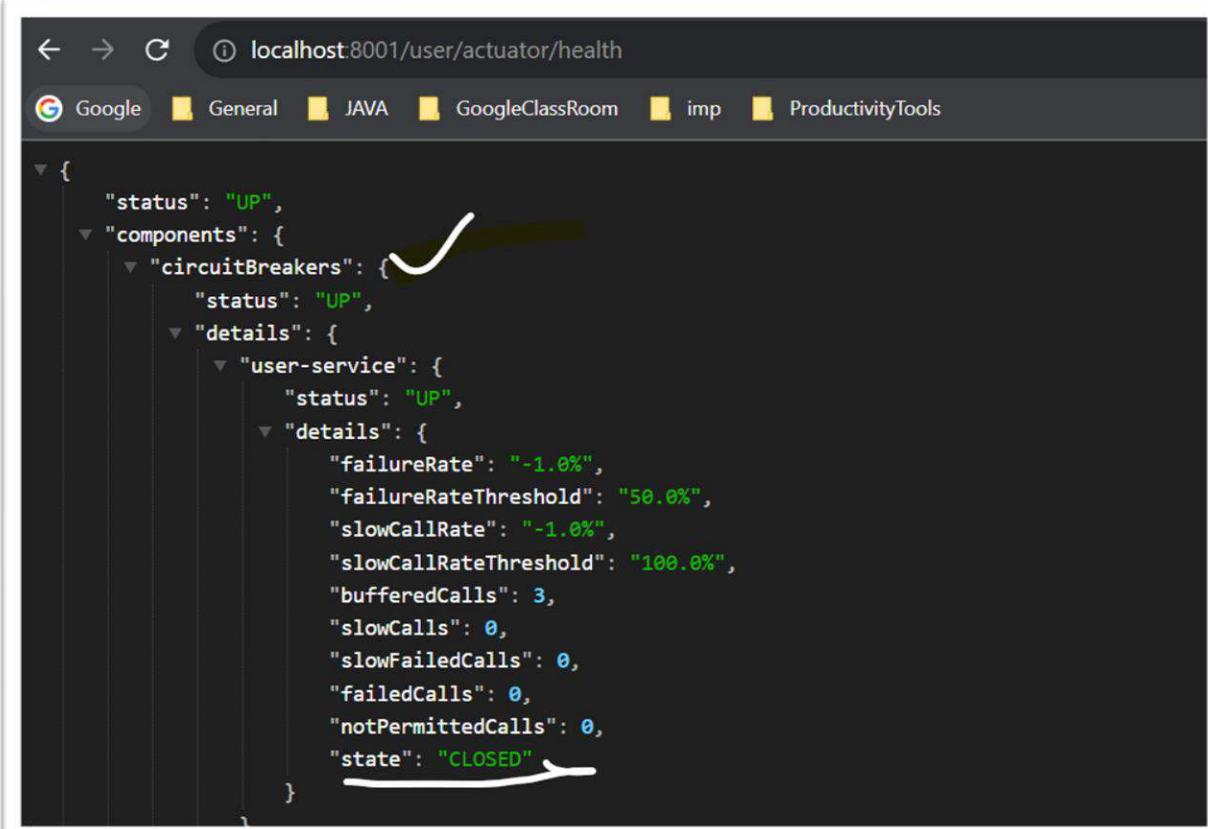
1 {
2   "emailId": "dilip@gmail.com",
3   "orderId": "order1234",
4   "amountPaid": 199999,
5   "paymentStatus": "Success"
6 }

```

Body Cookies Headers (3) Test Results 200 OK 578 ms 202 B Save as Example

1 Please have paymentId for Tracking: 8002 : Please contact If any issue info@swiggy.com

Now Access actuator health endpoint, for Watching Status of Circuit Breaker. Showing as Closed.

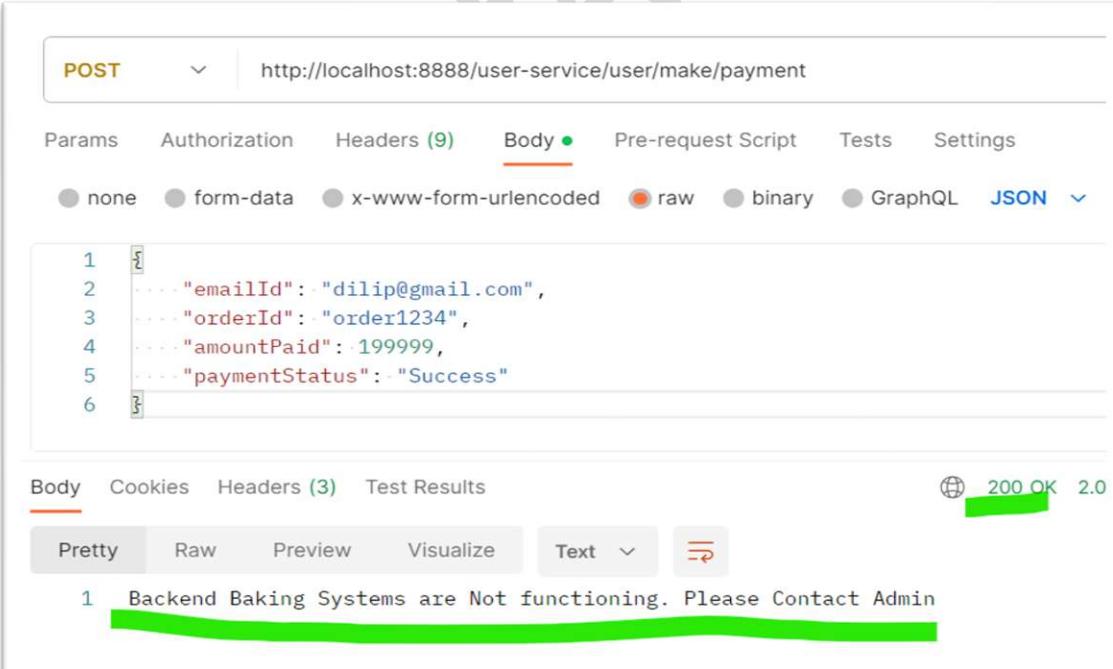


```

{
  "status": "UP",
  "components": {
    "circuitBreakers": {
      "status": "UP",
      "details": {
        "user-service": {
          "status": "UP",
          "details": {
            "failureRate": "-1.0%",
            "failureRateThreshold": "50.0%",
            "slowCallRate": "-1.0%",
            "slowCallRateThreshold": "100.0%",
            "bufferedCalls": 3,
            "slowCalls": 0,
            "slowFailedCalls": 0,
            "failedCalls": 0,
            "notPermittedCalls": 0,
            "state": "CLOSED"
          }
        }
      }
    }
  }
}

```

- When Payment Service is Down: Bring Down Payment Service.



POST <http://localhost:8888/user-service/user/make/payment>

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```

1  {
2    "emailId": "dilip@gmail.com",
3    "orderId": "order1234",
4    "amountPaid": 199999,
5    "paymentStatus": "Success"
6  }

```

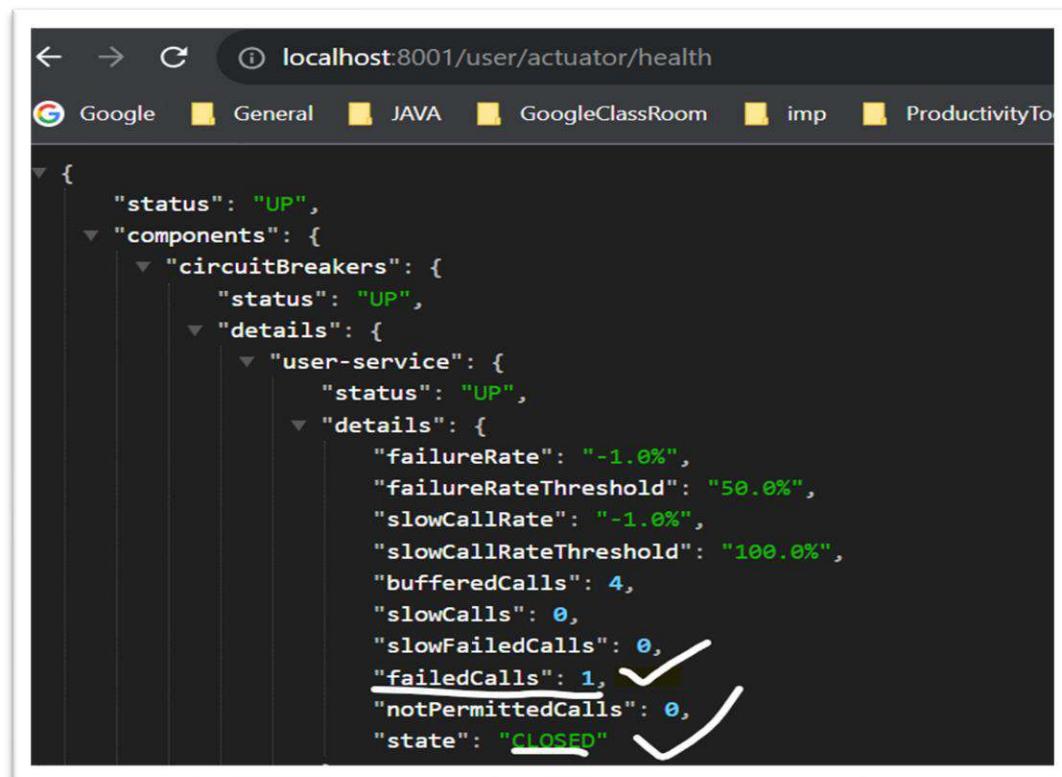
Body Cookies Headers (3) Test Results 200 OK 2.0

Pretty Raw Preview Visualize Text 

1 Backend Baking Systems are Not functioning. Please Contact Admin

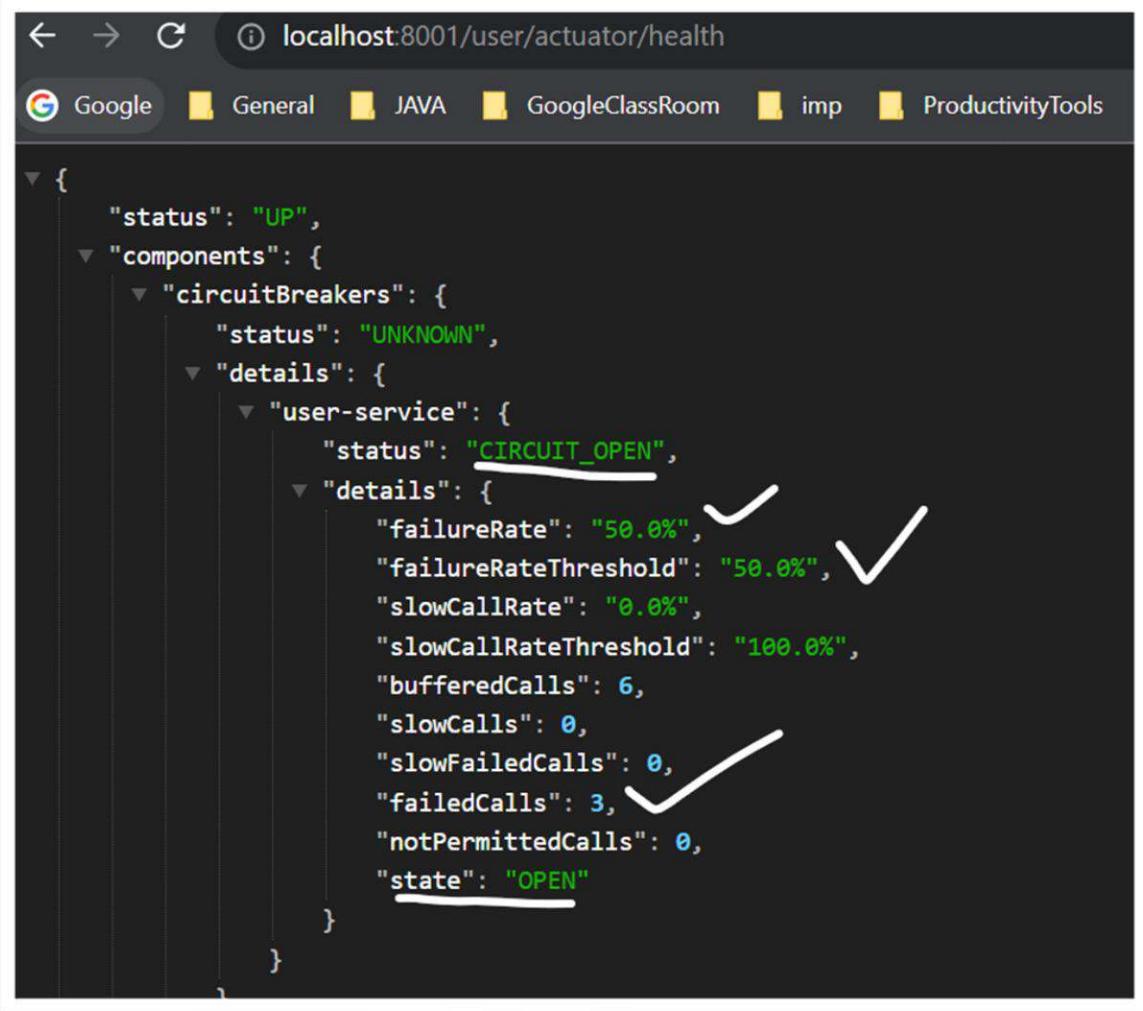
In Above, even though Payment Service is Down, instead of Internal Server Error, we are getting response of fallback method with status 200 Ok.

Now Check Circuit Breaker Status from actuator health endpoint.



```
{
  "status": "UP",
  "components": {
    "circuitBreakers": {
      "status": "UP",
      "details": {
        "user-service": {
          "status": "UP",
          "details": {
            "failureRate": "-1.0%",
            "failureRateThreshold": "50.0%",
            "slowCallRate": "-1.0%",
            "slowCallRateThreshold": "100.0%",
            "bufferedCalls": 4,
            "slowCalls": 0,
            "slowFailedCalls": 0,
            "failedCalls": 1, ✓
            "notPermittedCalls": 0,
            "state": "CLOSED" ✓
          }
        }
      }
    }
  }
}
```

After 50% of threshold calls, Circuit opened as per our configuration in properties files.



```
localhost:8001/user/actuator/health

{
  "status": "UP",
  "components": {
    "circuitBreakers": {
      "status": "UNKNOWN",
      "details": {
        "user-service": {
          "status": "CIRCUIT_OPEN",
          "details": {
            "failureRate": "50.0%",
            "failureRateThreshold": "50.0%",
            "slowCallRate": "0.0%",
            "slowCallRateThreshold": "100.0%",
            "bufferedCalls": 6,
            "slowCalls": 0,
            "slowFailedCalls": 0,
            "failedCalls": 3,
            "notPermittedCalls": 0,
            "state": "OPEN"
          }
        }
      }
    }
  }
}
```

Until Actual Payment Service is Up and Running and getting original Response Data, Circuit breaker will be opened. Once Actual service is Good, then Circuit breaker will be closed.

This is how we are going to implement Circuit Breaker to achieve fault tolerance where we have API service calls are integrated in any Micro Service.

Note: If we want to move Circuit Breaker Configuration from local to Config Server i.e. GitHub, we can do that.

All latest Configuration and Micro Services are located in GitHub.

<https://github.com/DilipItAcademy/SpringBoot-MicroServices>

<https://github.com/DilipItAcademy/swiggy-config-server-data>

Thank you
Dilip Singh