Spring Boot Framework

By – **Dilip Singh**

dilipsingh1306@gmail.com

dilipsingh1306

Spring Boot-Core Module

Spring Boot Introduction:

Before Starting with Spring Boot we should have basic understanding of Spring Framework. Because Spring is widely used for creating web applications. For web applications Spring provides Spring MVC which is a widely used module of spring which is used to create scalable web applications. But main disadvantage of spring projects is that configuration is really time-consuming and can be a bit difficult for the new developers. Making the application is production-ready, takes some time if you are new to the spring.

Solution to this is Spring Boot. Spring Boot is built on the top of the spring framework and contains all the features of spring. And is becoming favourite of developer's these days because of it's a rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the project configuration and project set up.

Spring Boot Framework is more compatible with microservice-based and making production-ready application in it takes very less time comparing with Spring.

Prerequisite for Spring Boot is the basic knowledge Spring framework.

Spring Boot is an open-source framework for building and deploying Java-based applications. It is a part of the larger Spring ecosystem, which provides various tools and libraries for enterprise-level Java development. Spring Boot is widely used for creating all kinds of applications, from small microservices to large-scale enterprise systems. It significantly simplifies the development process and allows developers to focus on building business logic rather than dealing with infrastructure and configuration complexities. Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". Most Spring Boot applications need minimal Spring configuration. Automatically configure Spring and 3rd party libraries whenever possible.

So before starting with Spring Boot, Let's have some basic discussion of Spring framework like how it is designed, implemented and how we are using in Project level.

Please Keep in mind as, Spring Boot is a wrapper framework on Spring Farmwork i.e. Internally Spring Boot Uses Spring Framework only.

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 etcc..
- 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 applications 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.

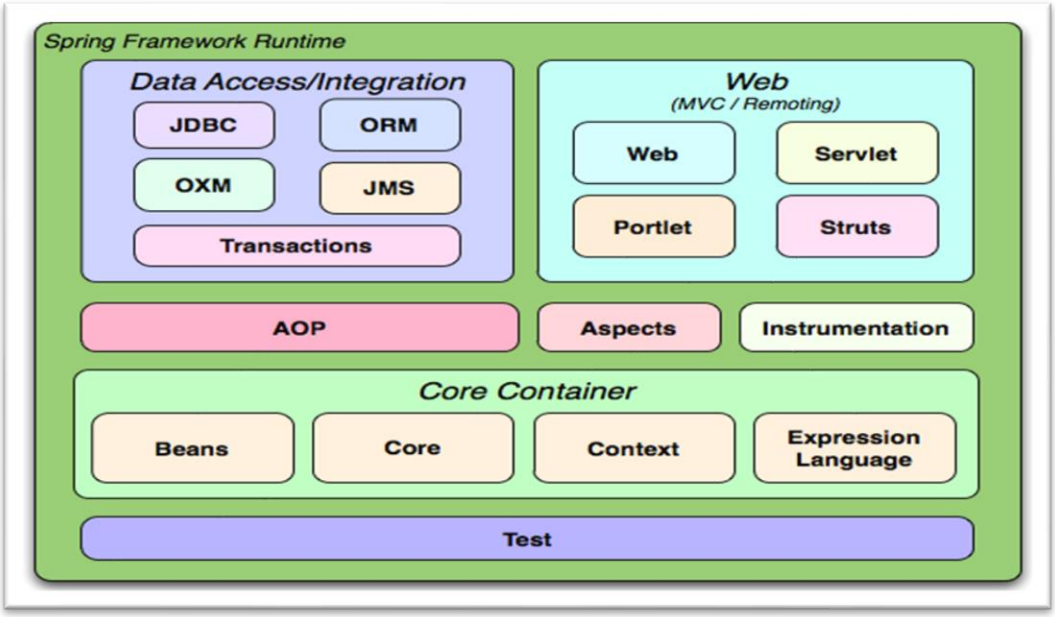
Overall, Spring Framework has become one of the most popular Java frameworks due to its ease of use, modularity, and extensive features. It is widely used in enterprise applications, web applications, and other types of Java-based projects.

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.

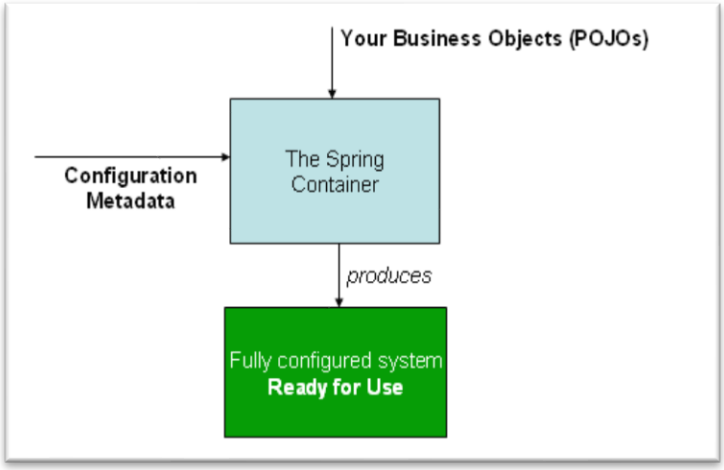


Figure :The Spring IoC container

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 FileSystemXmlApplicatio nContext 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.

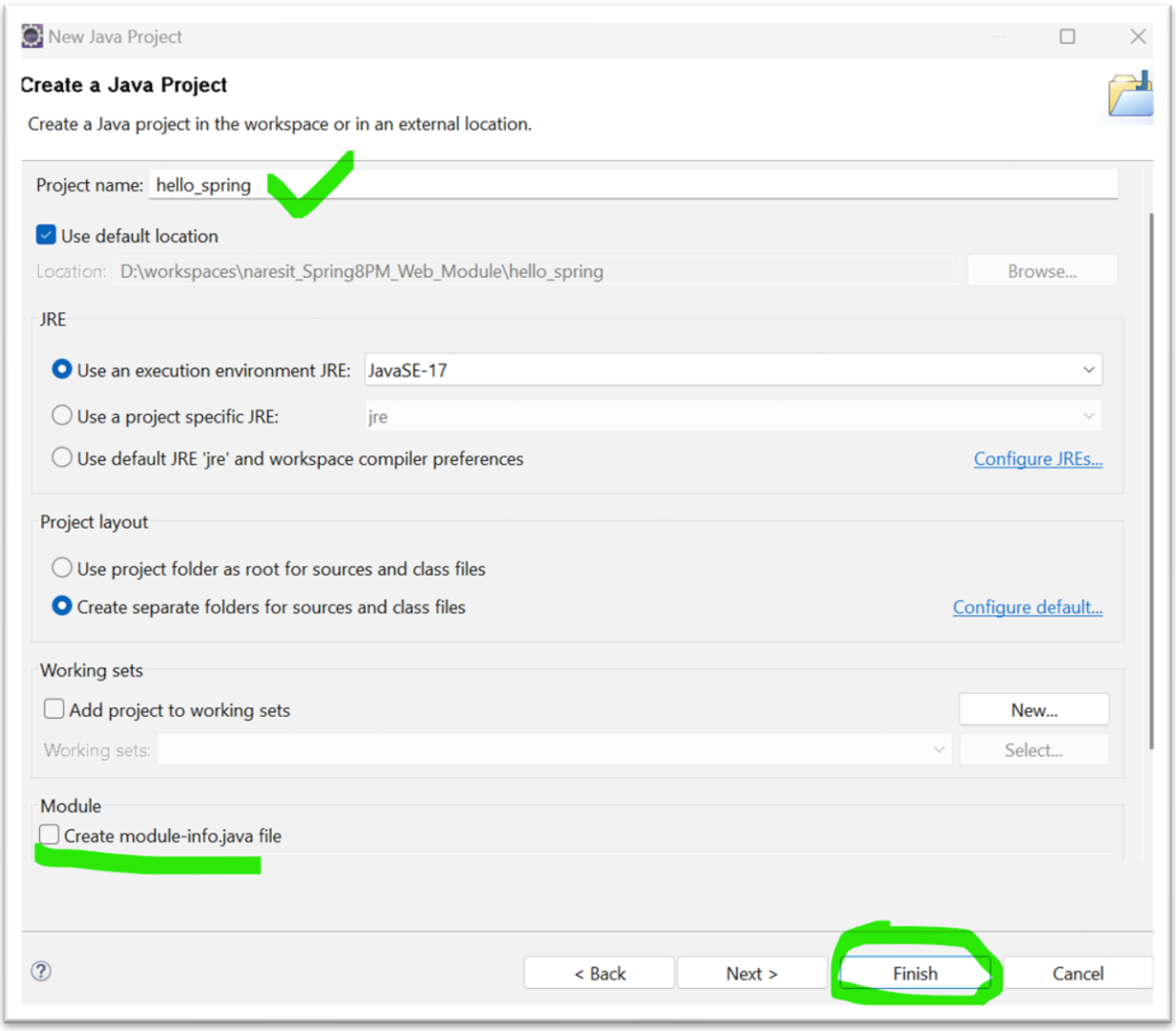
1. Manually Downloading Spring JAR files and Copying/Configuring Build Path
2. By Using Maven Project Setup, Configuring Spring JAR files in Maven. This is preferred in Real Time practice.

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.**

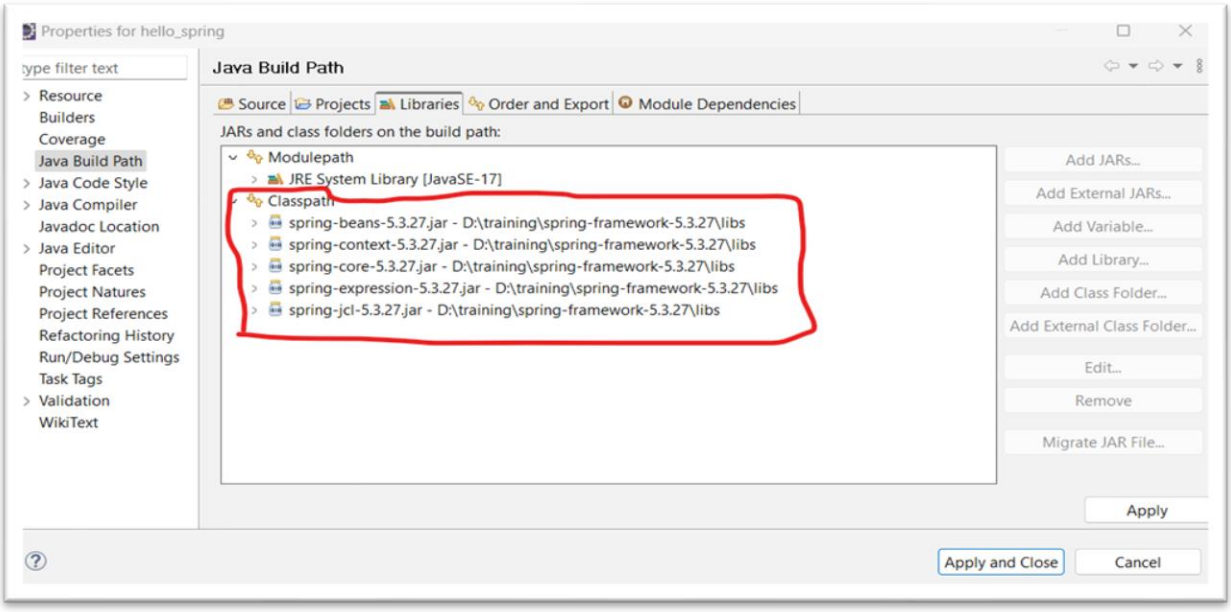
   I have uploaded copy of all Spring JAR files uploaded in Google Drive. You can download from below link directly.

   https://drive.google.com/file/d/1FnbtP3yqjTN5arlEGeoUHCrIJcdcBgM7/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.

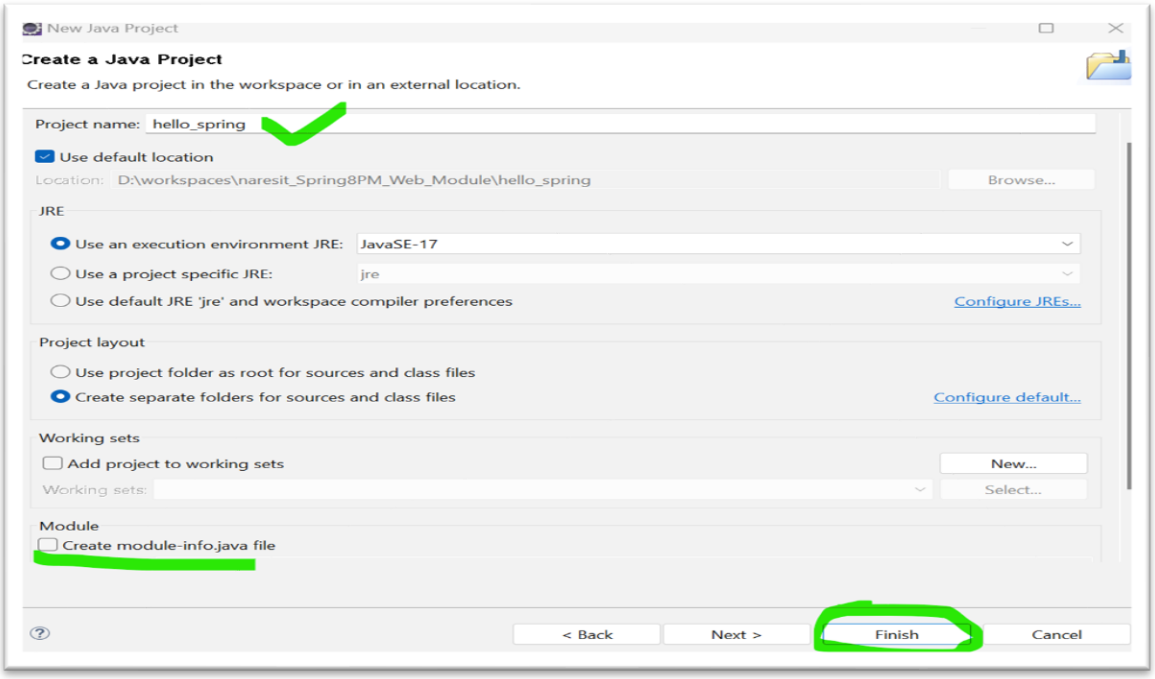
Right Click on Project -> Build Path -> Configure Build Path -> Libraries -> ClassPath



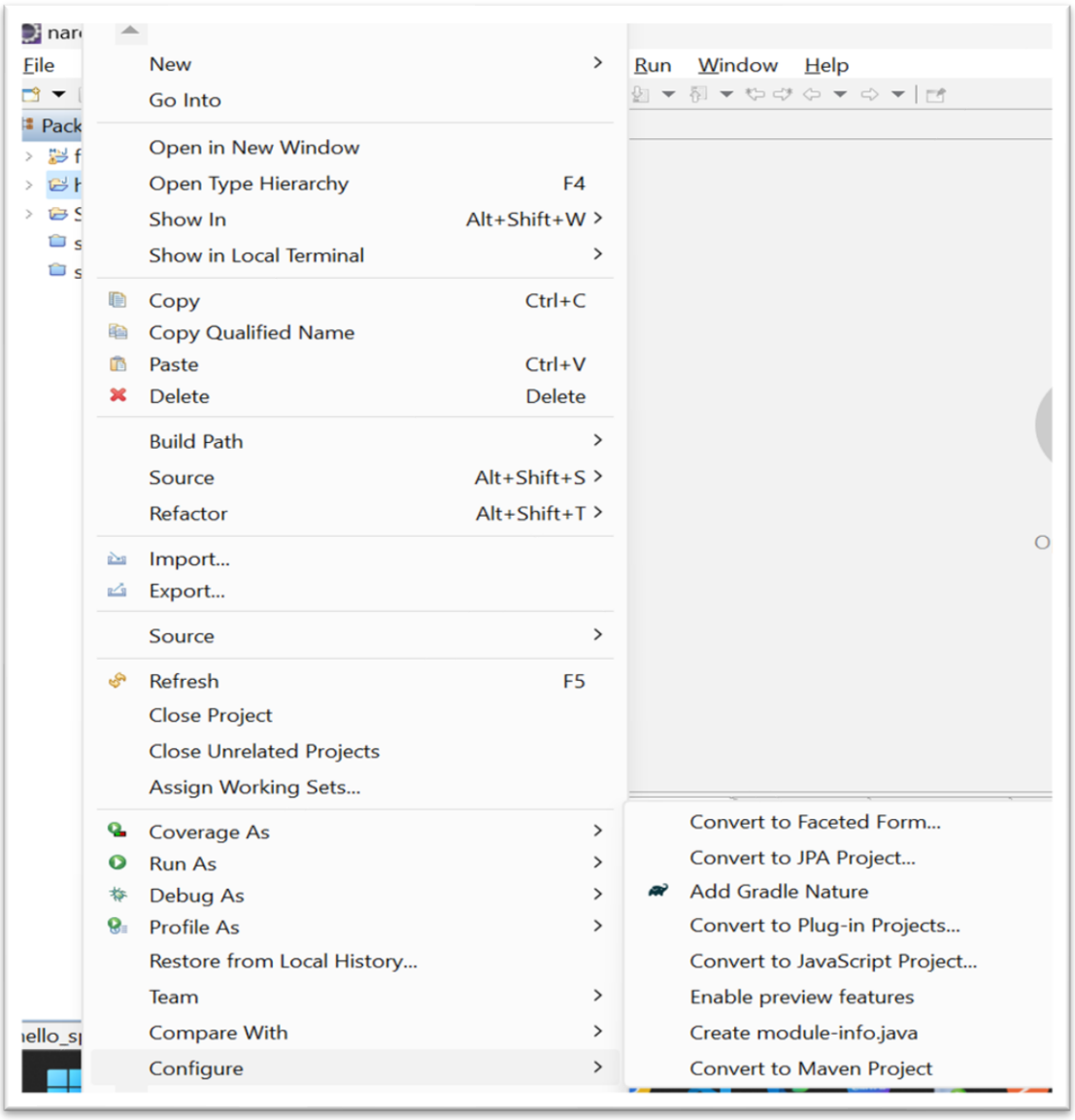
With This Our Java Project is Supporting Spring Core Module Functionalities. We can Continue with Spring Core Module Functionalities.

Creating Spring Core Project with Maven Configuration:

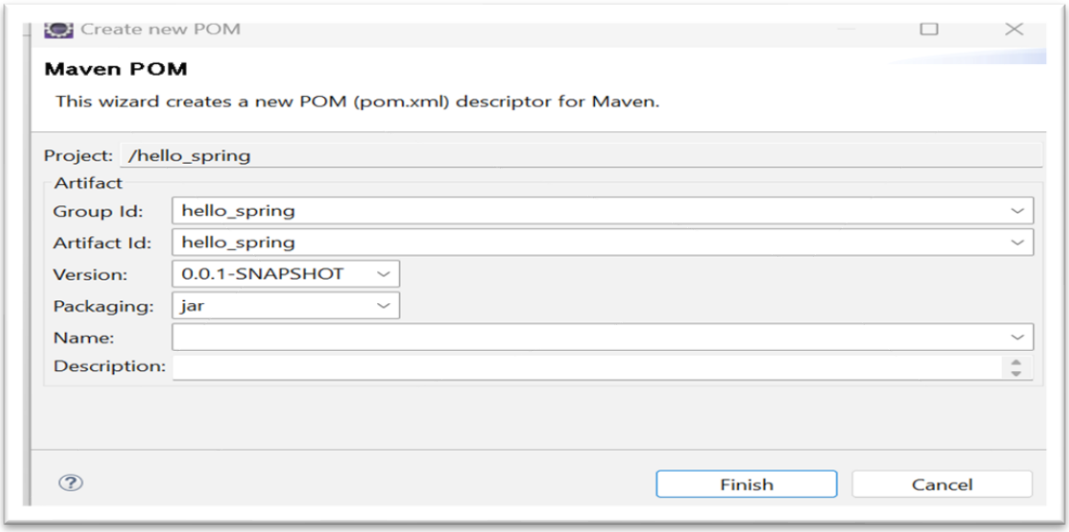
1. **Create Java Project.**
   Open Eclipse -> File -> new -> Project -> Java Project -> Enter Project Name -> Un-Select Create Module-Info -> Click Finish.



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



➢ **Immediately It will show below details and click on Finish.**

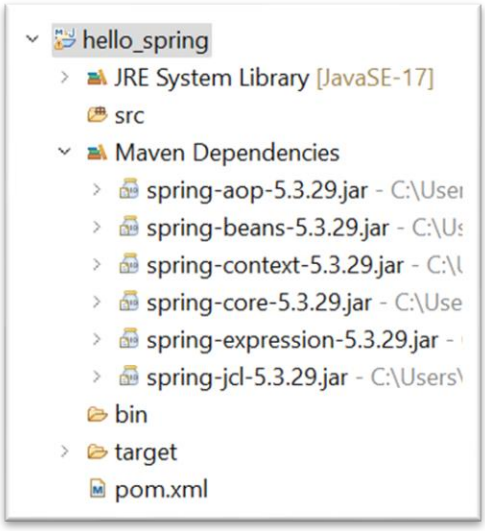


3. **Now With Above Step, Java Project Supporting Maven functionalities. Created a default pom.xml as well. Project Structure shown as below.**

Now Open pox.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.

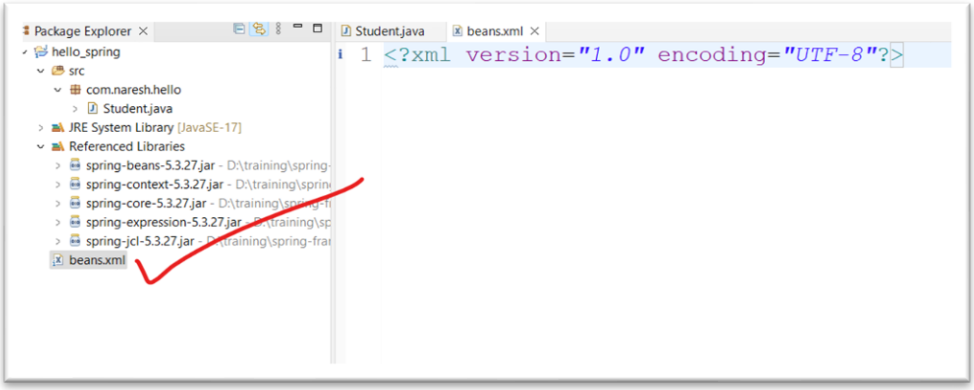
NOTE: Below Steps are now common across our Spring Core Project created by either Manual Jar files or Maven Configuration.

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:

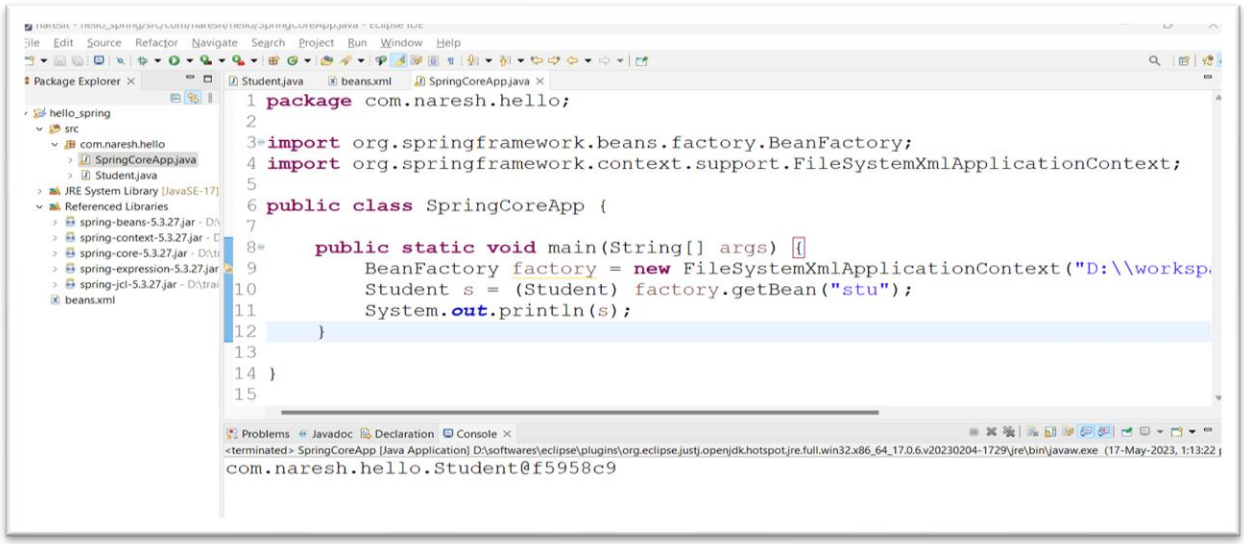
- Every class will be configured with **<bean>** tag, we can call it as Bean class.
- The **id** attribute is a string that identifies the individual bean name in Spring IOC Container i.e. similar to Object Name or Reference.
- The **class** attribute is fully qualified class name our class i.e. class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



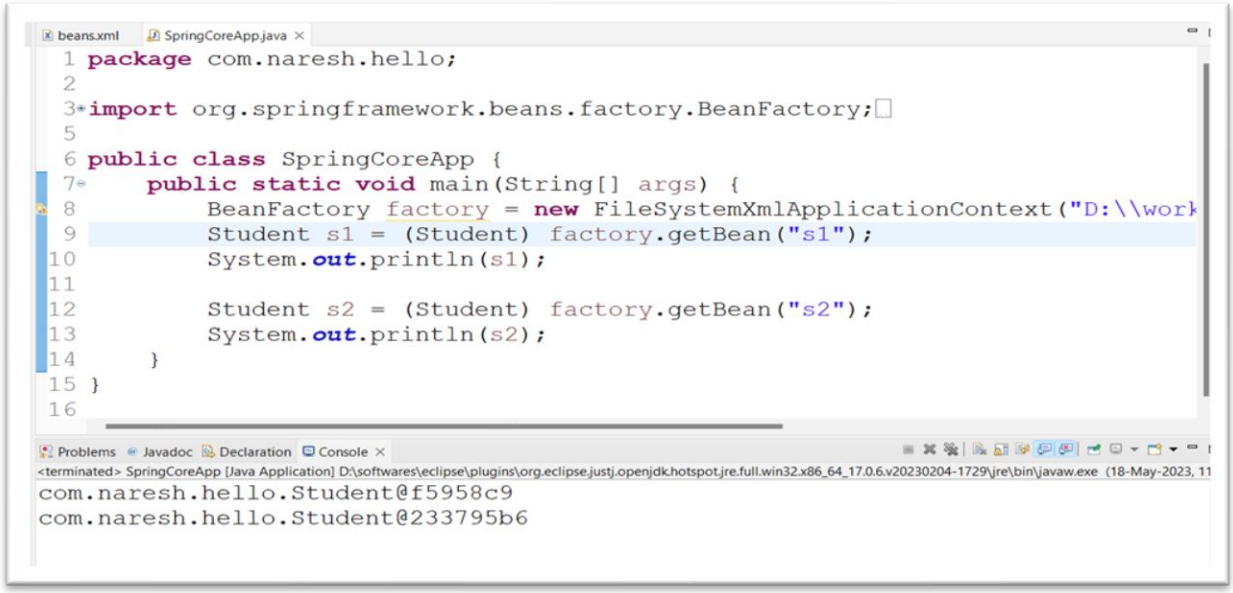
In above example Student Object Created by Spring IOC container and we got it by using **getBean()** method. If you observe, we are not written code for Student Object Creation i.e. using new operator.

- **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.**



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.

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\\naresit\\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

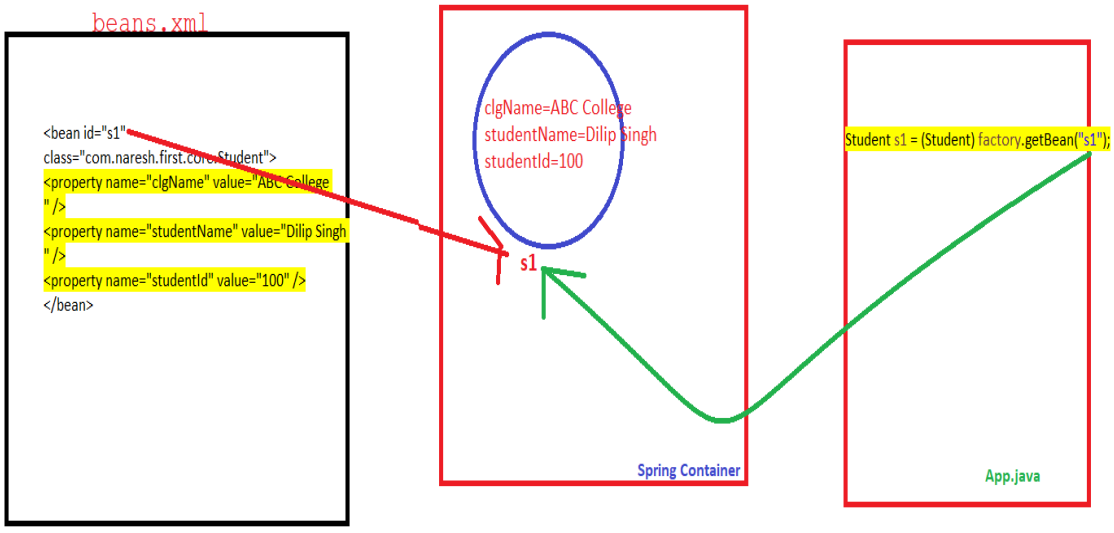
Internal Workflow/Execution of Above Program.

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 setIsSelected(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.**

```
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}
```

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.

- **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>
                <property name="houseName" value="Lotus Homes"></property>
                <property name="mobile" value="91822222"></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="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>
</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("accountDeatils");

                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.java**

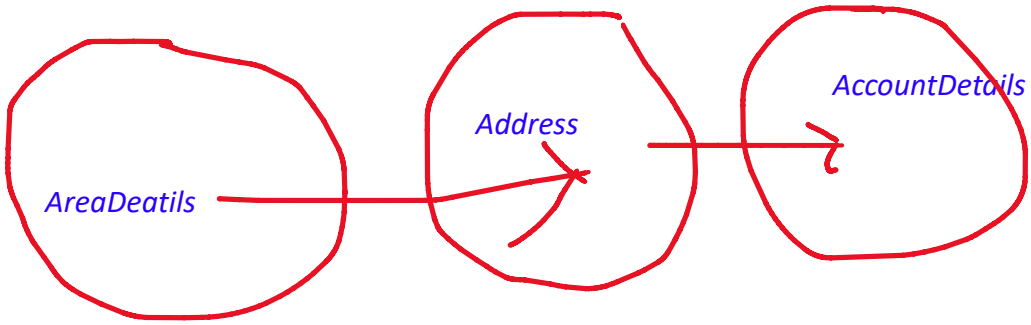
```
package com.naresh.hello;

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="91822222"></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>+91-88888888</value>
                                <value>+232388888888</value>
                        </set>
                </constructor-arg>
        </bean>
</beans>
```



Testing of 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");

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

                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().getArea().getPincode());

        }
}
```

Output:

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

SpringBoot:

Now we are starting Spring Boot and whatever we discussed in above 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.

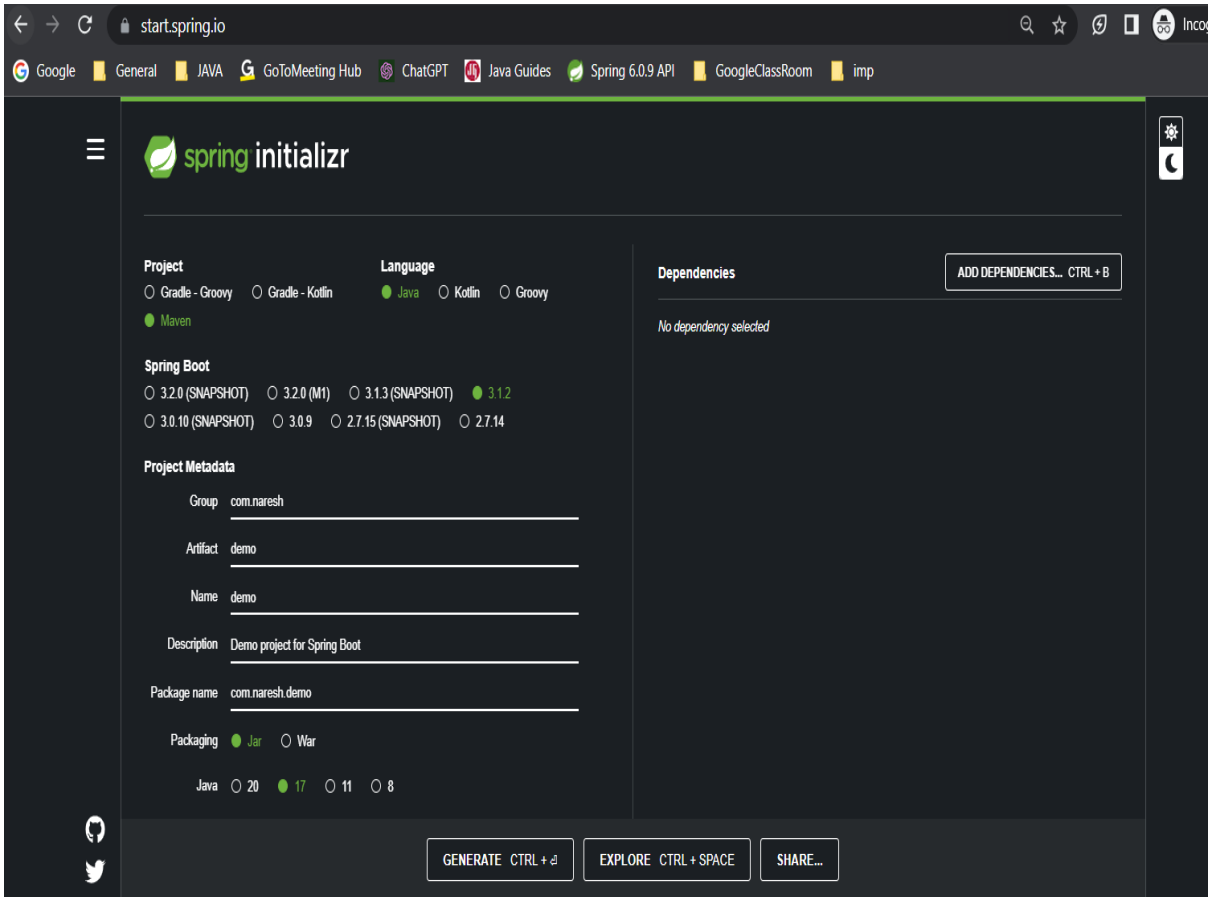
Overall, Spring Boot has revolutionized Java development by simplifying the creation of robust, production-ready applications. Its emphasis on convention-over-configuration, auto-configuration, and opinionated defaults makes it an excellent choice for developers seeking to build modern, scalable, and maintainable Java applications.

Let's Create 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.

1. **Using https://start.spring.io 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 it from Eclipse IDE or any other IDE's.

2. 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.

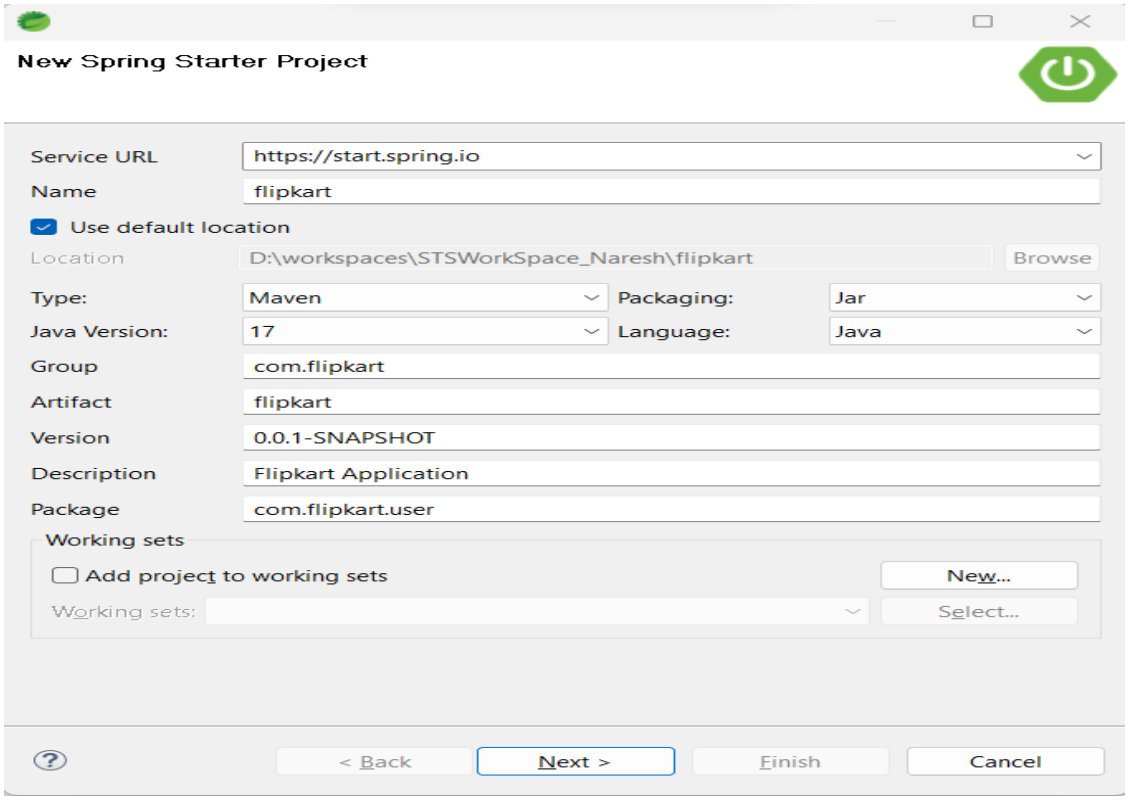
STS is a widely used IDE for Spring development due to its rich feature set and seamless integration with the Spring Framework and related technologies. It provides a productive environment for building robust and scalable Spring applications, particularly those leveraging Spring Boot's capabilities. 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

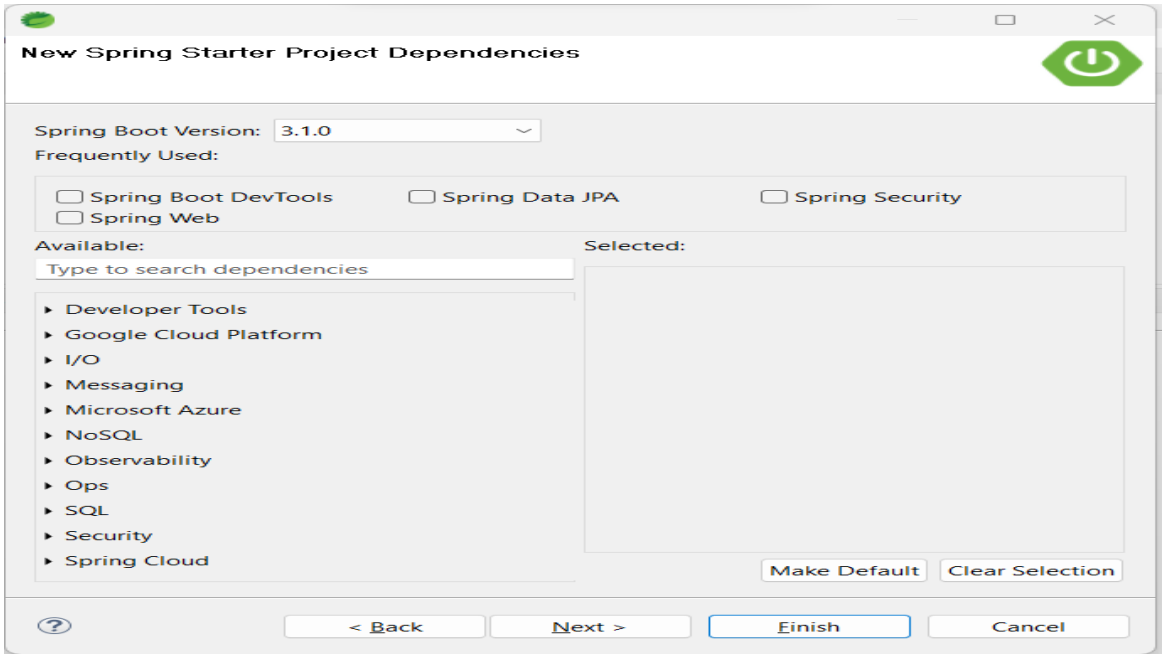
➢ **It will download STS as a jar file. Double click on jar, it will extract STS software.**

➢ 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. no t required to add any Dependencies in this case.

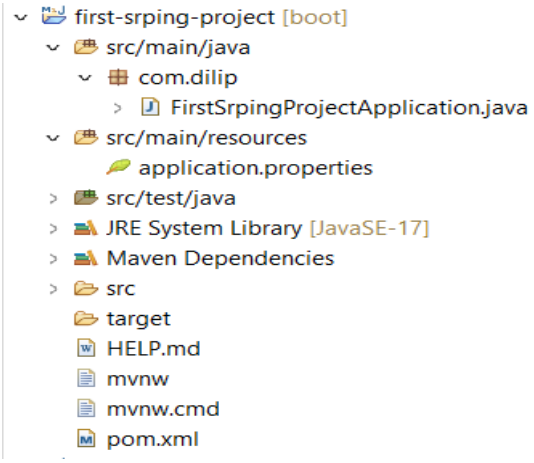


Now project created and imported to STS Directly.

If we observer we are not added any jar files manually or externally to project like 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 file 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.

FirstSrpingProjectApplication.java

```java
package com.dilip;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class FirstSrpingProjectApplication {

    public static void main(String[] args) {
        SpringApplication.run(FirstSrpingProjectApplication.class, args);
    }

}
```

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**

```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:

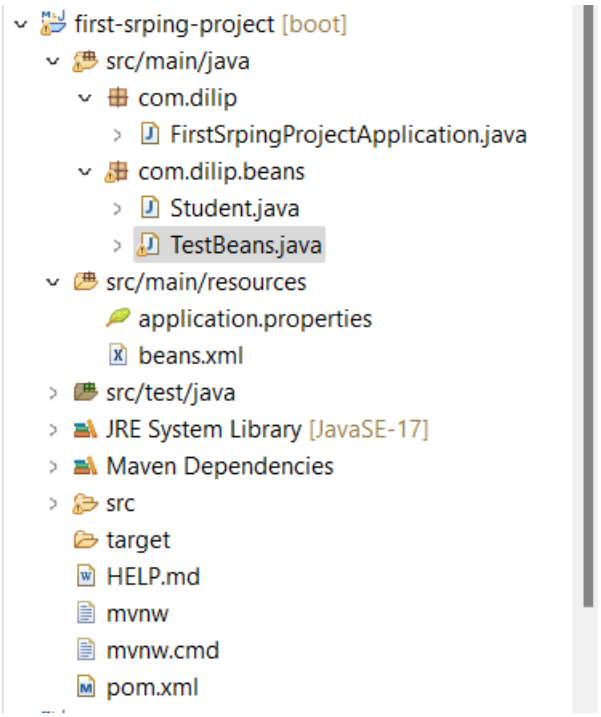
```
package com.dilip.beans;

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 Structure :



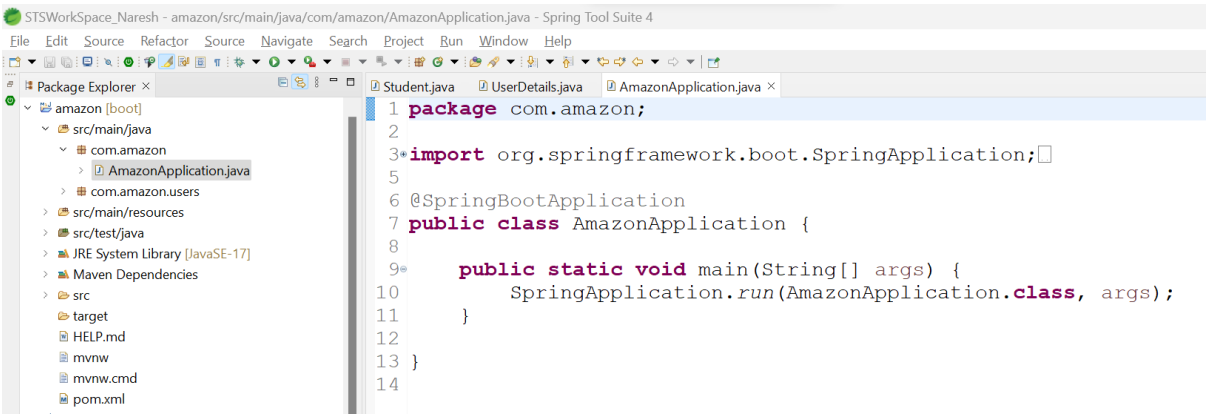
NOTE: If we observe above logics, we are written same code of Spring Framework completely. Means, Nothing new in spring boot w.r.to Coding/Logic point of view because Spring Boot itself a Spring Project.

So Please Practice all examples of Spring framework what we discussed previously w.r.to XML configuration. We will continue with JAVA/Annotation Based Configuration.

Create Beans and Configure Beans: Annotation Based Configuration.

Java-based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations.

- Create A SpringBoot Core Module project.



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

```
package com.amazon.users;

public class UserDetails {

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

	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;
	}
}
```

- To Define Beans Configuration, Spring/Spring Boot Provided few annotations. We should have knowledge of Annotations here.

@Configuration:

When we annotate a class with **@Configuration**, Spring treats it as a source of bean definitions. These bean definitions are processed by the Spring container during the application startup, and the corresponding beans are created and managed accordingly.

To define beans in Configuration class we will use **@Bean** annotations.

@Bean:

@Bean is an annotation used to define a bean and its configuration details explicitly. Beans are objects managed by the Spring IoC (Inversion of Control) container, and they represent various components in your application.

When you annotate a method with **@Bean**, you are telling Spring that the method will create and configure an instance of a bean. The return value of the method represents the bean instance that will be registered in the Spring container. The container will then manage the lifecycle and dependencies of this bean.

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 name is annotated with **@Bean** works as bean ID and it creates and returns the actual bean. 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();

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

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

		context.close();
	}
}
```

Output: `com.amazon.users.UserDetails@34bde49d`

From above code :

- **`AnnotationConfigApplicationContext`** is a standalone application context which accepts annotated classes as input. For instance, `@Configuration` or `@Component`. Beans can be looked up with `scan` or registered with `register`.
- **context.register(),** Registers one or more Configuration/ component classes to be processed. We can register multiple configuration classes with register method.
  Example:

```
ctx.register(AppConfig.class, OtherConfig.class);
ctx.register(AdditionalConfig.class);
```

- **context.refresh(),** Loads or refresh the representation of the configurations, which might be from Java-based configuration, an XML file, a properties file, a relational database schema, or some other format.
- **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 configuration classes as well as Same bean with multiple bean id's.

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

```
package com.amazon.products;

public class ProductDetails {

        private String pname;
        private double price;
        public ProductDetails(){
            System.out.println("ProductDetails Object Created");
        }

        //setters and getters
}
```

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 class Object Created or not. Below Code loading Two Configuration classes.

```
package com.amazon;

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.

```
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();
        }

        @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 {
```

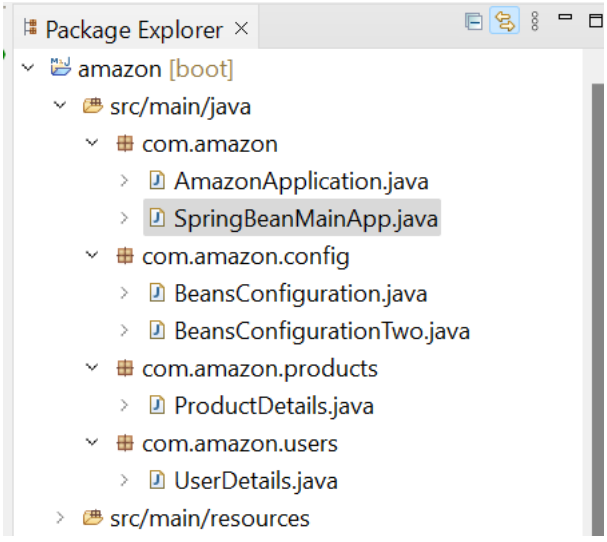
```
        public static void main(String[] args) {
                AnnotationConfigApplicationContext context = new
                AnnotationConfigApplicationContext();

                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.

Project Files Structure:



Bean Scopes:

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 six scopes, four of which are available only if you use a web-aware ApplicationContext i.e. inside Web applications.

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

singleton: (Default) Scopes a single bean definition to a single object instance for each Spring IoC container.

prototype: Scopes a single bean definition to any number of object instances.

request: Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean. Only valid in the context of a web-aware Spring ApplicationContext.

session: Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.

application: Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

globalsession: Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

Defining Scope of beans:

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.
Bean Class: ProductDetails

```
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(). In side 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 ProdcutDetails 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.BeansConfigurationThree;
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 ProdcutDetails Object is prototype or not. Request multiple times ProductDetails Object from Spring Container by passing bean id productTwoDetails.

```
package com.amazon;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfigurationThree;
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**".

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

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

@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. Candidate 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.amazon.products;

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

@Component("product")
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;
	}
	@Override
	public String toString() {
		return "ProductDetails [pname=" + pname + ", price=" + price + "]";
	}
}
```

Now Test in Main Class.

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

public class SpringComponentDemo {

	public static void main(String[] args) {
		AnnotationConfigApplicationContext context = new
		AnnotationConfigApplicationContext();
		context.refresh();
		ProductDetails details =  (ProductDetails) context.getBean("product");
		System.out.println(details);

	}
}
```

Now we got an exception as,

```
Exception in thread "main"
org.springframework.beans.factory.NoSuchBeanDefinitionExceptio
n: No bean named 'product' available
```

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 components from specified packages for those classes annotated with **@Component**, **@Controller**, @Service, and **@Repository.**

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

basePackages: Packages to specify scanning for all component classes and configuration classes from those provided package names.

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.*")
```

Alternatively, We can also use **scan(String... basePackages)** method with **AnnotationConfigApplicationContext** instance in place of **@ComponentScan** at Configuration class level. This method will accept **basePackages** similar to **@ComponetScan** annotations and scans Component Classes as well as Configuration classes resides in side the base packages. This method Performs a scan within the specified base packages. Note that refresh() must be called in order for the context to fully process the new classes. If we use scan() method, then no need to write Configuration classes to represent **ComponentScan** again.

Test our component class:

- Create A Configuration class with @ComponentScan annotation.

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

}
```

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

```
package com.amazon.products;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {

	public static void main(String[] args) {
		AnnotationConfigApplicationContext context = new
		AnnotationConfigApplicationContext();

		//context.scan("com.amazon.*");
		context.register(BeansConfiguration.class);
		context.refresh();
```

```
                ProductDetails details =  (ProductDetails) context.getBean("product");
                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.amazon.users;

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 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.**

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

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.amazon.users.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.amazon.products;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.amazon.users.UserDetails;

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.amazon.users.UserDetails@5e17553a
com.amazon.users.UserDetails@5e17553a
```

So we can say by default component classes are instantiated as singleton bean object, when there is scope defined. Means, Internally Spring Container considering as singleton scope.

If we want pass, scope value externally then we can use @scope at class level of component class.

Now for Above 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 Bena Object **UserDetails** from Spring Container.

```
package com.amazon.products;

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

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.amazon.users.UserDetails@74f6c5d8
com.amazon.users.UserDetails@27912e3
```

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.amazon.products;

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

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

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

}
```

Testing from Main Class:

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

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.amazon.users.UserDetails@3eb91815
```

How to pass default values to @Component class properties?

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

Inside Configuration Class:

```
package com.amazon.products;

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

```
@Configuration
@ComponentScan("com.amazon.*")
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.amazon.products;

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

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
88261113774
```

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 here 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 auto wring 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. This is called as Field/Property injection only supported by Annotation based configuration.

First, Let's Define a bean : Address.java

```
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.scan(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 of Single Bean/Component Class:

Let's Create Bean class: 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.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 empployee = (Employee) context.getBean("emp");
		Addresss empAdd = empployee.getAdd();
		System.out.println(empAdd);

	}
}
```

We got an exception now as,

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependen
cyException: 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 @Q 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 @Qulifier 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;

}
```

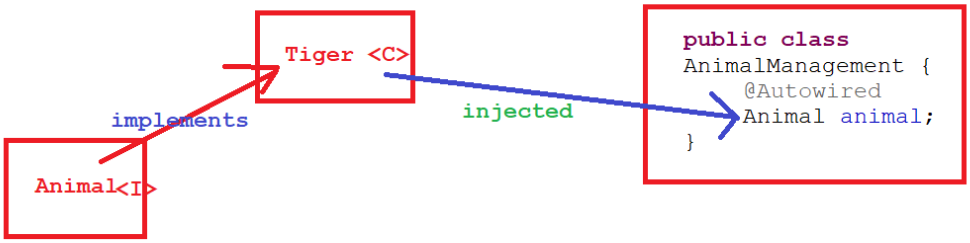
Now Test, Animal type property injected with what type of Object.

```
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;
```

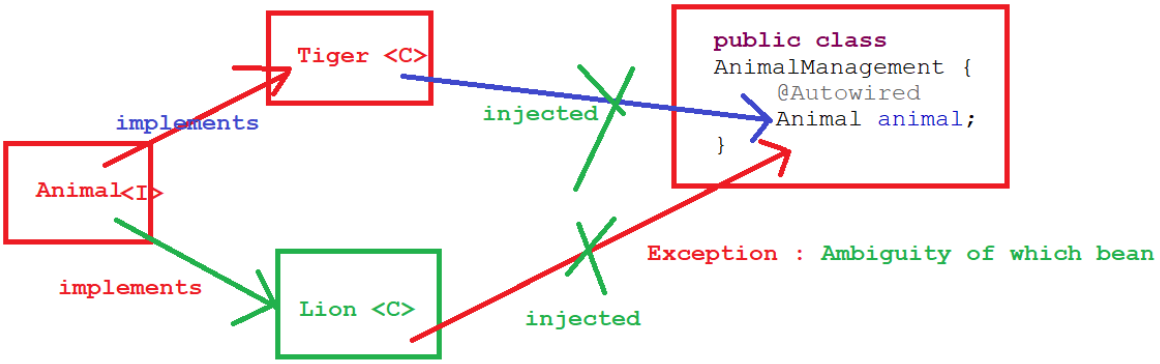
```
@Component("lion")
public class Lion implements Animal {
	@Override
	public void printNameOfAnimal() {
		System.out.println("I am a Lion ");
	}
}
```

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.UnsatisfiedDependencyExcepti
on: 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 again this ambiguity between multiple implementation of single interface, again we can use **@Qualifier** with Bean Id or Component Id.

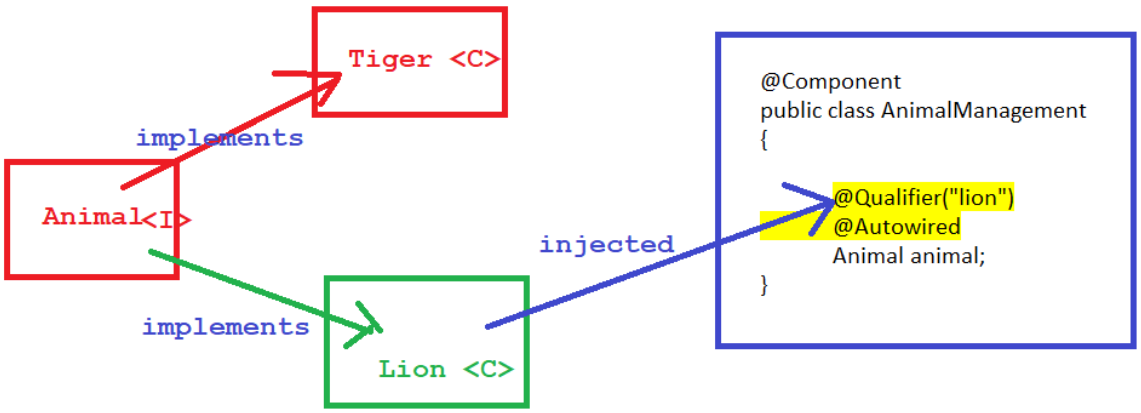
```
package com.dilip.auto.wiring;

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 if 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 filed.

Requirement : Address id Dependency of Employee class.

Address.java : Created 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 etter 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
```

Question: In Java, private properties can't access outside of the class, then How Spring Injecting Dependency Object of with private variable of Address type externally in Employee Object?

Internally , Spring Uses Reflection API and loads Bean class Details and Injects from backward process.

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 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:

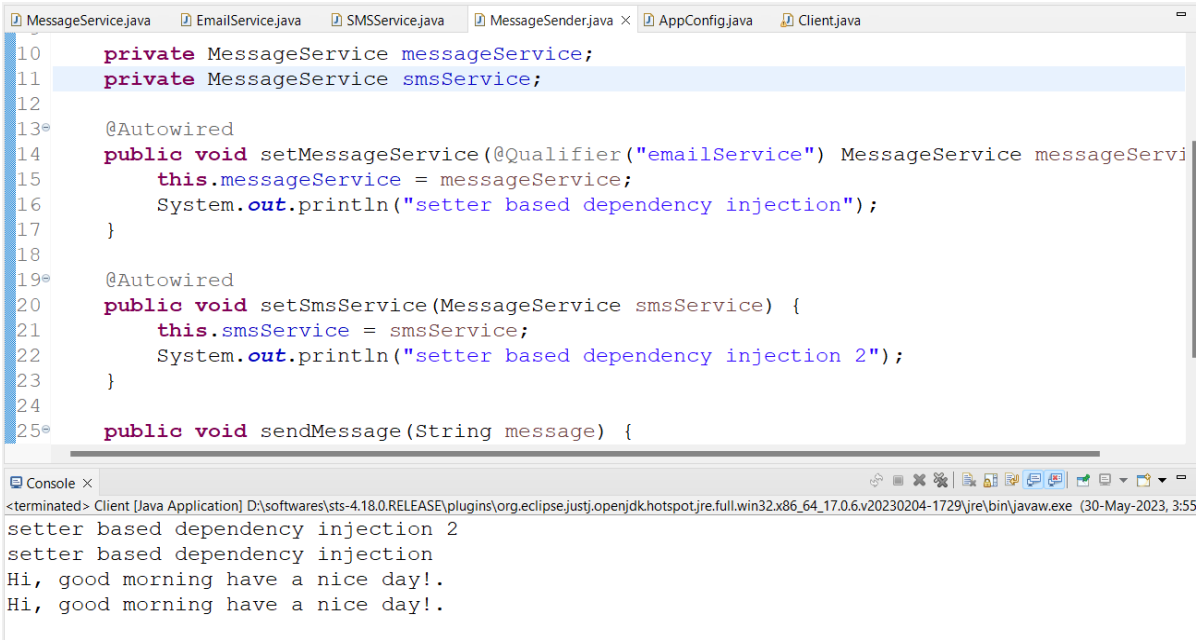
```
package com.dilip.setter.injection;

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!.
```



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!.
```

Question: 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 **@Qualifeir** then, Spring injecting Dependency Object in side composite Object specific to Bean ID.

constructor : when we are using **@Autowired** and **@Qulaifier** 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.

@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 processed by Spring Container.

This can be important for cases where the order of execution matters. Here's how you can use **@Order** in Spring Boot:

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 {
        // ...
}
```

In this example, **MyFirstComponent** will be processed 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'.

```
@Component
@Order(-1)
public class MyHighPriorityComponent {

}
```

If you have multiple beans with the same order value, the initialization order among them is not guaranteed.

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.

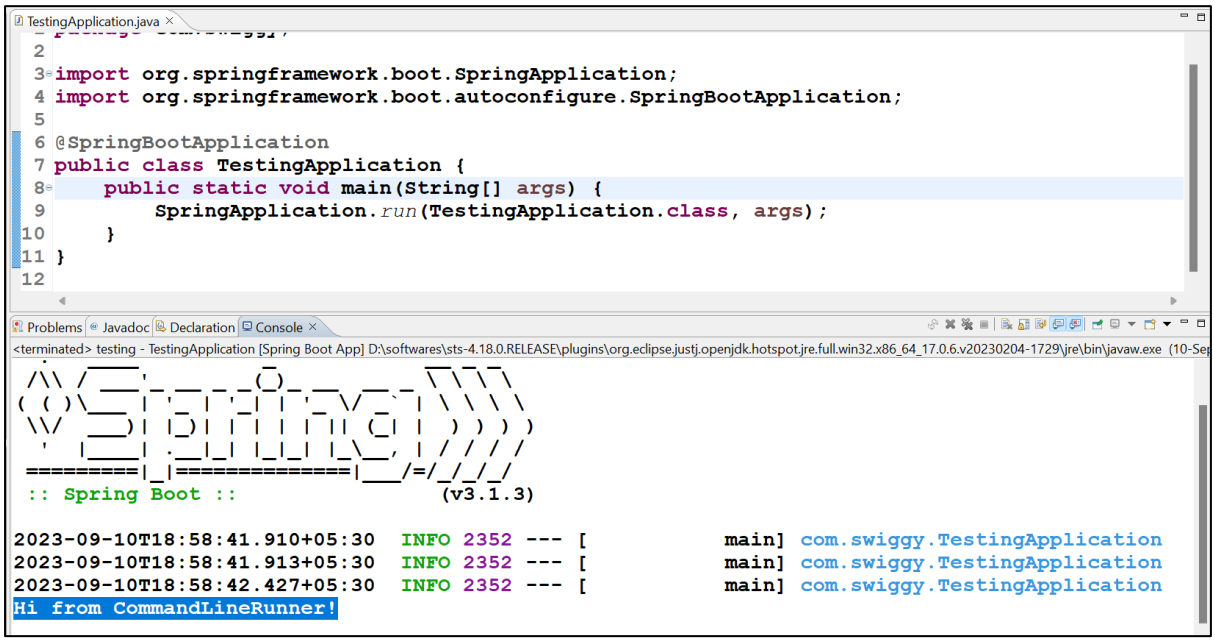
3. We can have multiple **CommandLineRunner** implementations, and they will be executed in the order specified by the **@Order** annotation or the **Ordered** interface if you want to control the execution order.

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.



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.

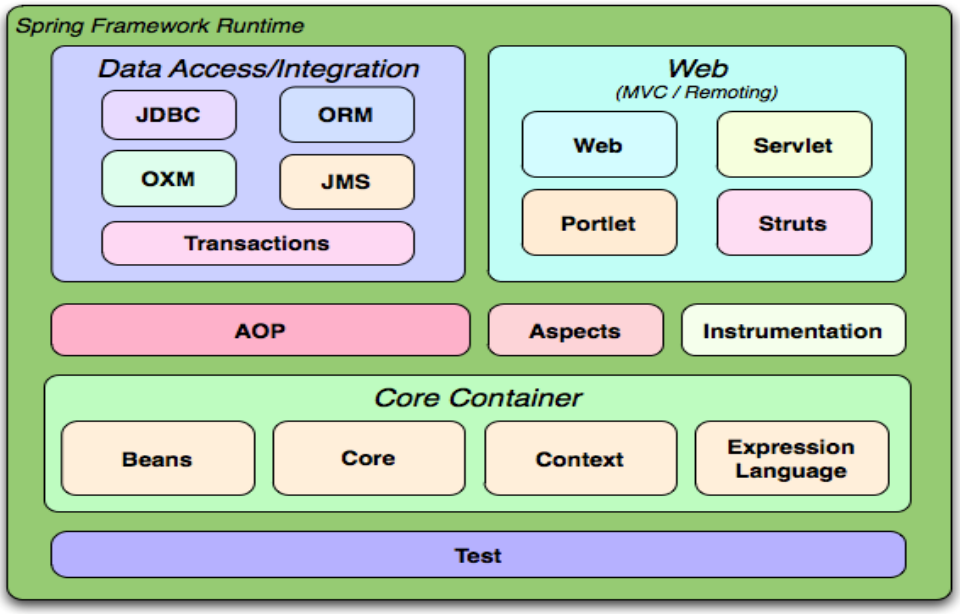
Your choice between **CommandLineRunner** and **ApplicationRunner** depends on your specific requirements and the complexity of command-line argument processing in your Spring Boot application. If you need advanced features like option parsing and validation, **ApplicationRunner** is the more suitable choice. Otherwise, **CommandLineRunner** provides a simpler, more straightforward approach.

SpringBoot
Web,
REST API,
JPA,
Security 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.

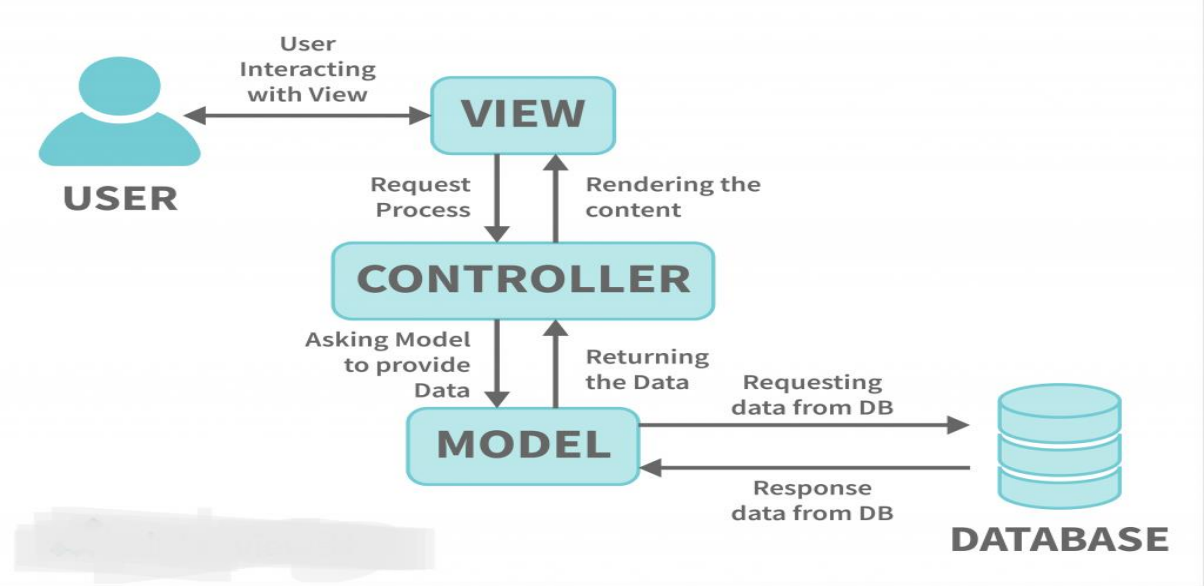


What is MVC?

MVC (Model, View, Controller) Architecture is the design pattern that is used to build the applications. This architectural pattern was mostly used for Web Applications.

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 data 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.

Advantages of Spring MVC Framework

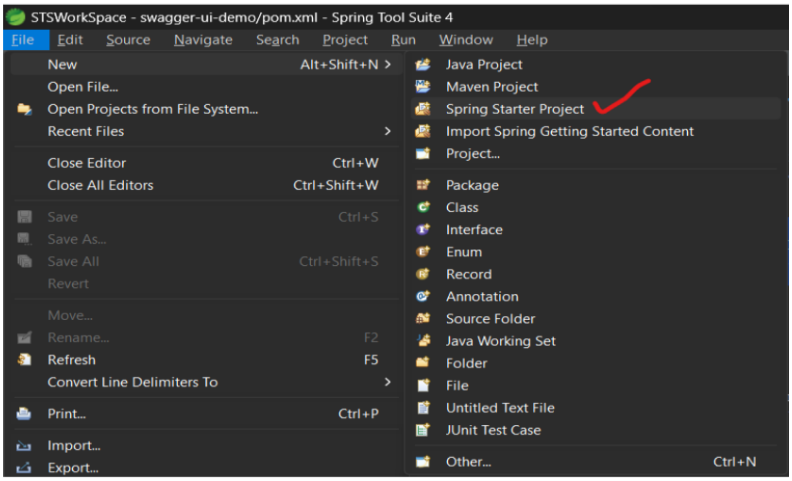
Let's see some of the advantages of Spring MVC Framework:

- **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.

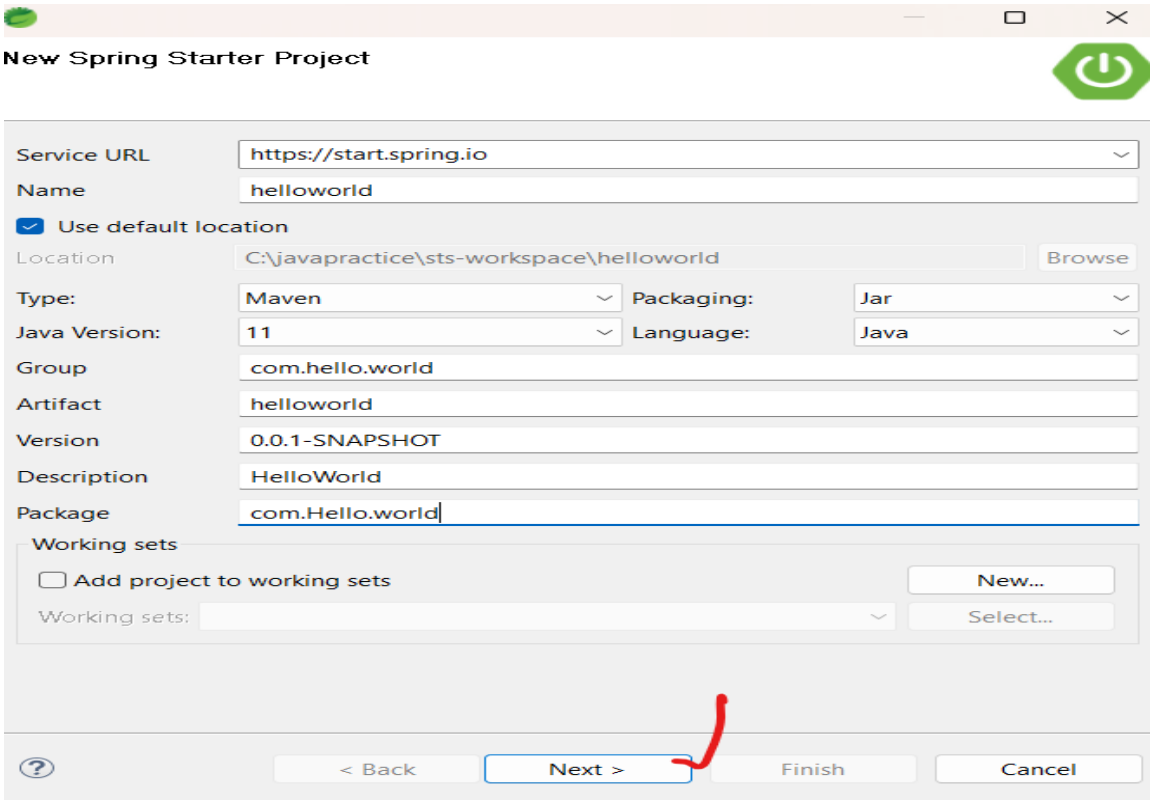
- **Rapid development** - The Spring MVC facilitates fast and parallel development.
- **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
- **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
- **Flexible Mapping** - It provides the specific annotations that easily redirect the page.

Creating SpringBoot Web Application:

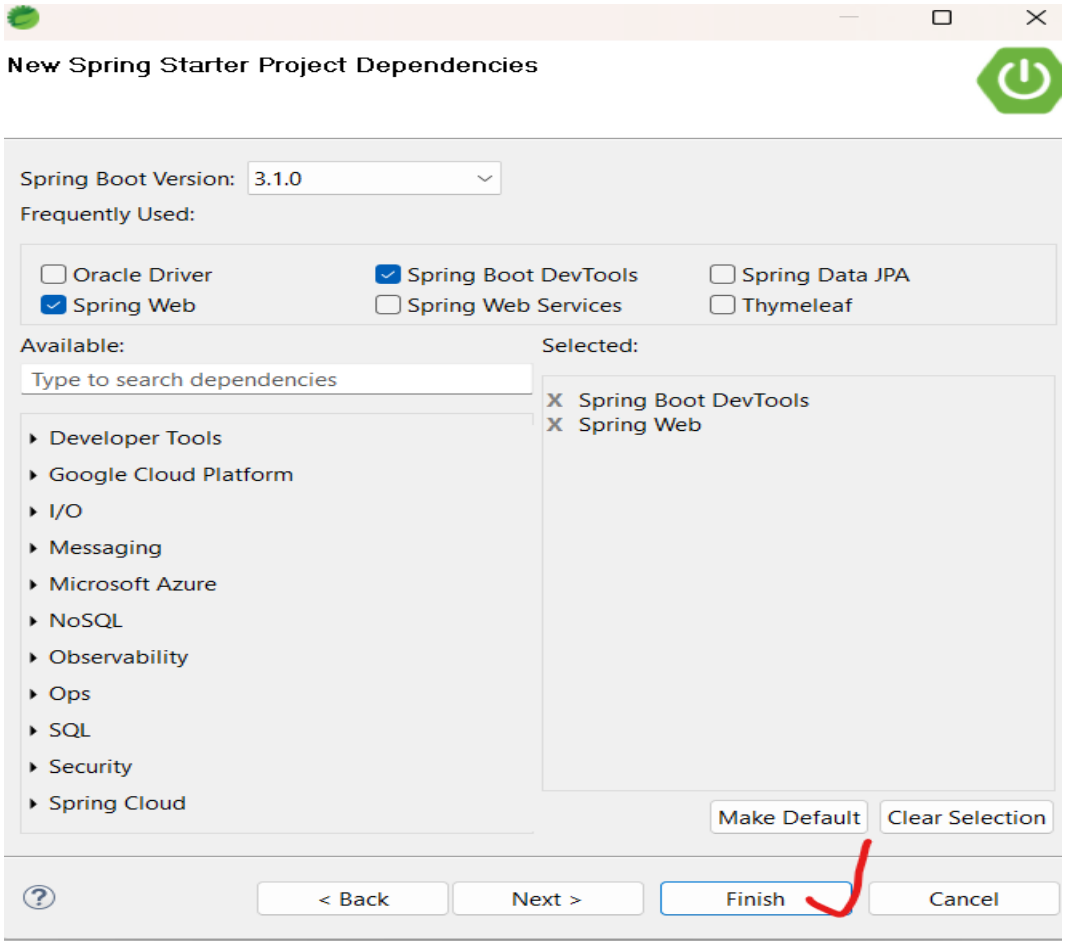
1. Open STS
2. File-> New > Spring Starter Project



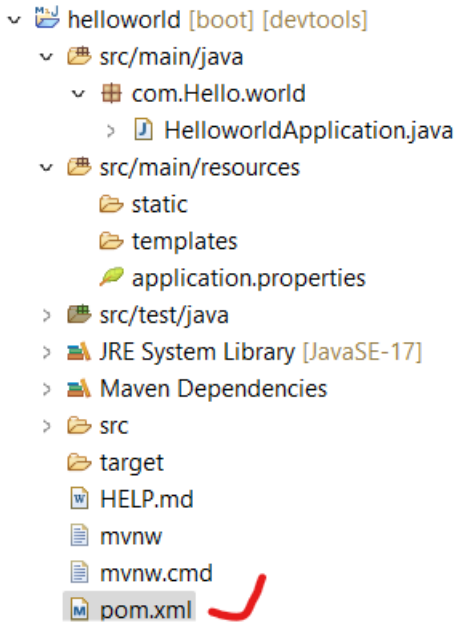
3. Fill All Project details as shown below and click on Next.



4. In Next Page, Add Spring Boot Modules/Starters as shown below and click on finish.
NOTE: Spring Web is mandatory



5. 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.

Integrated Server Started with Default Port : 8080 with context path ' '. i.e., if we won't give any port number the port number will be 8080 by default. If you want to change the port number then, we should add a property in **application.properties**.

Now open **application. Properties** file and add below properties and save it.

server.port=8899
server.servlet.context-path= /hello

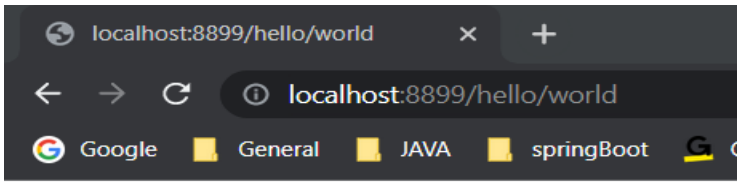
Restart our application again, application started on port(s): 8899 (http) with context path '/hello'

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 printHelloWrold() {
		return "Hello world! Welcome to Spring Boot MVC";
	}
}
```

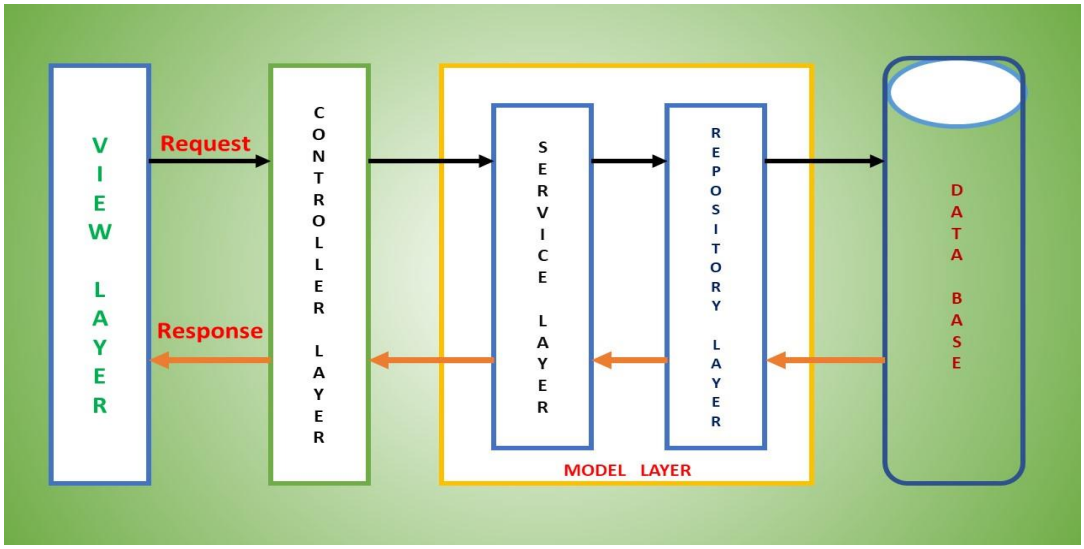
Output:



Hello world! Welcome to Spring Boot MVC

Spring MVC Application Workflow:

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 application. The request and response processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the diagram.

Front Controller:

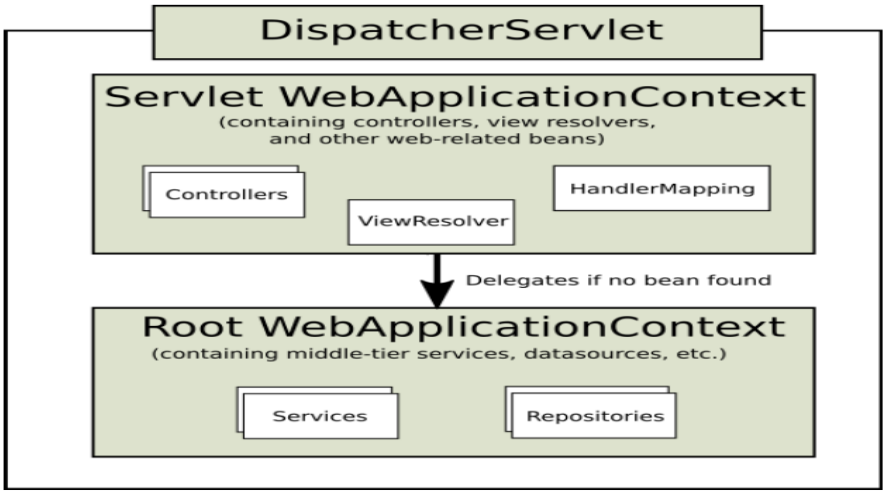
A front controller is defined as a controller that handles all requests for a Web Application. DispatcherServlet servlet is the front controller in Spring MVC that intercepts every request and then dispatches requests to an appropriate controller. The DispatcherServlet is a Front Controller and one of 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 Dispatcher Servlet will with the help of Handler Mapping understand the Controller class name associated with the received request. Once the Dispatcher Servlet 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:

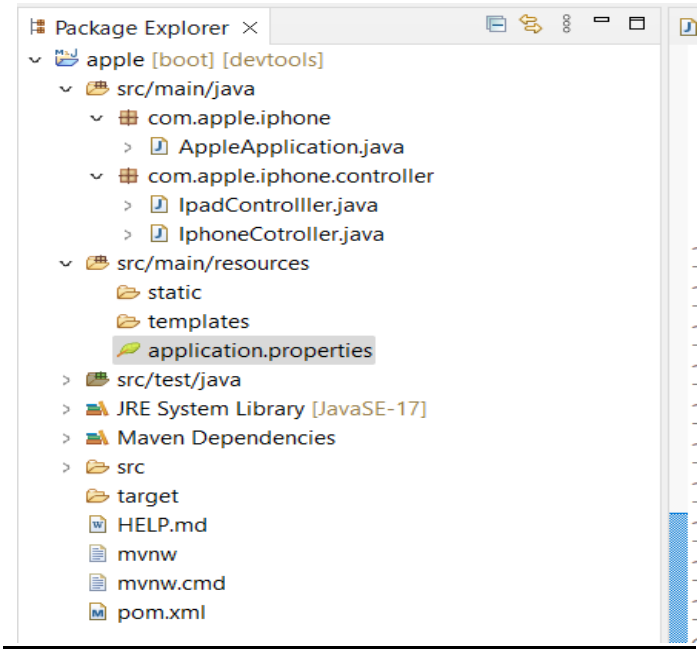
- **Create a controller class : IphoneCotroller.java**

```
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 IphoneCotroller {

        @GetMapping("/message")
```

```
	@ResponseBody
	public String printIphoneMessage() {
		//Logic of Method
		return " Welcome to Ihpne World.";
	}
	@GetMapping("/cost")
	@ResponseBody
	public String printIphone14Cost() {
		return " Price is INR : 150000";
	}
}
```

- **Create another Controller class.**

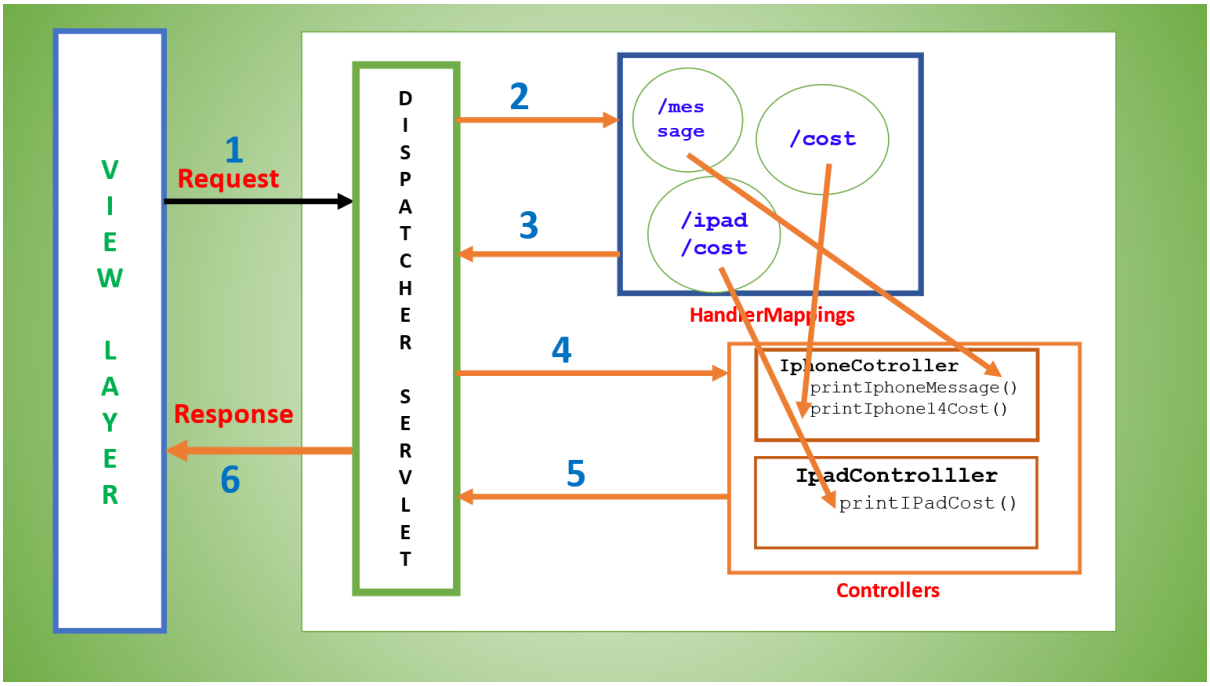
```
package com.apple.iphone.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class IpadControlller {
	@GetMapping("/ipad/cost")
	@ResponseBody
	public String printIPadCost() {
		return " Ipad Price is INR : 200000";
	}
}
```



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 WebApplcationContext 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 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 *HandlerMapping* to call the appropriate *Controller and its associated* method of endpoint URL.
- 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 REST API requests, preparing a model, and returning the view to be rendered as a response. 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: 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 *HttpResponse*. This indicates a method return value should be bound to the web response i.e. *HttpResponse* 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 HttpResponse object.

@RequestMapping: 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.
		
		`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.
		
		`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: 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 IphoneCotroller {
	@RequestMapping("/message")
	@ResponseBody
	public String printIphoneMessage() {
		return " Welcome to Ihpne World.";
	}
}
```

@RequestMapping("/message"):

1. If we are not defined in HTTP 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 to configure URL path of handler method. We can use either of them. **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",
```

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 call. 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",
```

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:

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 MAC2 World.";
}
```

@RequestMapping with Class:

We can use it with class definition to create the base URI of that specific controller. 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 IpadControllller {
	@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.

http://localhost:6655/apple/ipad/model
http://localhost:6655/apple/ipad/cost

@GetMapping: **org.springframework.web.bind.annotation.GetMapping**

Annotation 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 expression.

Similar to this annotation, we have other Composed Annotations to handle different HTTP methods.

@PostMapping:

Annotation 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:

Annotation 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:

Annotation for mapping HTTP DELETE requests onto specific handler methods. @DeleteMapping is a composed annotation that acts as a shortcut for **@RequestMapping(method = RequestMethod.DELETE).**

@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.

```
@Controller
public class MAcBookController {
	@GetMapping(path = "/mac/details")
	@ResponseBody
	public String getMacBookDetail() {
		return "MAC Book Details : Price 200000. Model 2022";
	}
}
```

- Used **@RestController** with controller class so removed @**ResponseBody** at method.

```
@RestController
public class MAcBookController {
	@GetMapping(path = "/mac/details")
	public String getMacBookDetail() {
		return "MAC Book Details : Price 200000. Model 2022";
	}
```

}

JSON:

JSON stands for **J**ava**S**cript **O**bject **N**otation. JSON is a **text format** for storing and transporting data. JSON is "self-describing" and easy to understand.

This example is a JSON string: `{"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

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

@RequestBody Annotation:

Simply put, the **@RequestBody** annotation maps the **HttpRequest** body to a transfer or domain object, enabling automatic deserialization of the inbound HttpRequest body onto a Java object. This annotation indicating a method parameter should be bound to the body of the request body data. The body of the request is passed through an HttpMessageConverter to resolve the method argument depending on the content type of the request.

First, let's have a look at a Spring controller method.

```
@RestController
public class UserController {

	@RequestMapping(path = "/register", method = RequestMethod.POST)
	public String registerUser(@RequestBody UserRegisterRequest request) {
		return "User Registerd Successfully";
	}
}
```

i.e. Creating an endpoint **/register** with HTTP method **POST**. This endpoint contains Request Body of either XML or JSON format. Now Spring will convert JSON or XML Request payload body converted to Java Object form i.e. UserRegisterRequest object.

Assume like For User Registration, we are receiving data of below properties.

1. firstName
2. lastName
3. emailId
4. mobile
5. password

i.e. JSON Request example payload:

```
{
   "firstName": "Dilip",
   "lastName": "Singh",
   "emailId": "dilipsingh1306@gmail.com",
   "mobile": "8125262702",
   "password": "Abc@123"
}
```

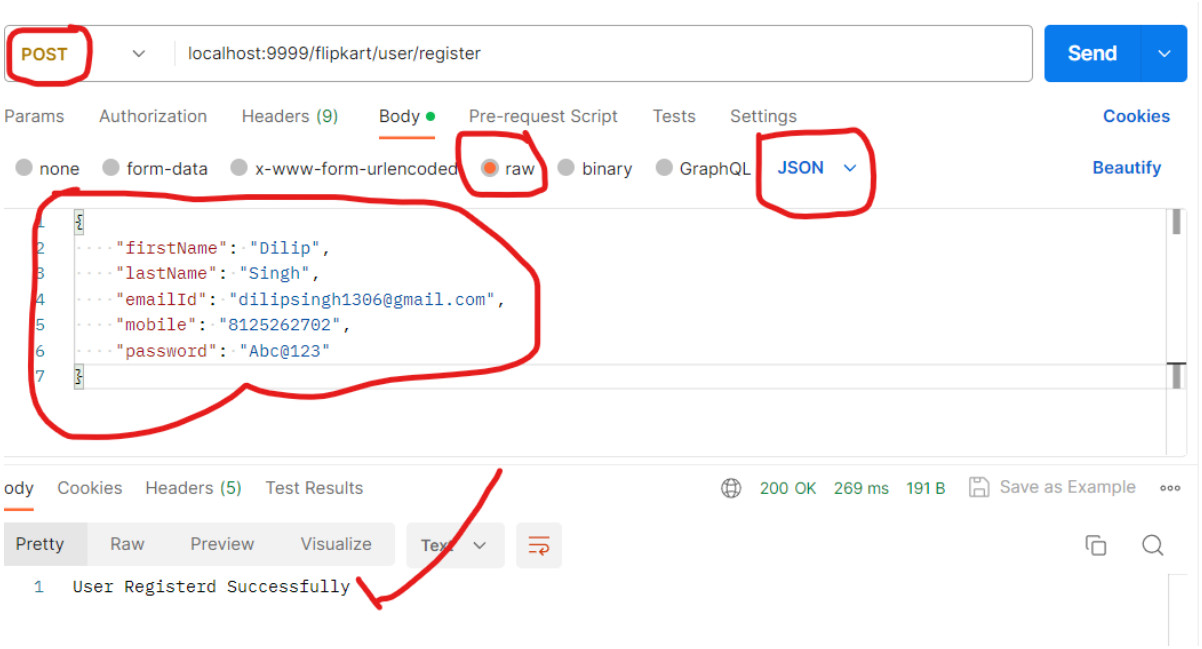
Spring automatically deserializes the JSON into a Java type, assuming an appropriate one is specified. By default, **the type we annotate with the** *@RequestBody* **annotation must correspond to the JSON sent from our client to controller.**

So now we should create a POJO class of **UserRegisterRequest** binding with an annotation **@RequestBody** in our request handling method.

```
package com.flipkart.user.request;

public class UserRegisterRequest {

	private String firstName;
	private String lastName;
	private String emailId;
	private String mobile;
	private String password;

	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 getMobile() {
		return mobile;
	}
	public void setMobile(String mobile) {
		this.mobile = mobile;
```

```
        }
        public String getPassword() {
                return password;
        }
        public void setPassword(String password) {
                this.password = password;
        }
}
```

- **Now we will trigger endpoint from Postman tool as shown in below.**
- Choose Http Method as : POST
- Enter URL of endpoint : localhost:9999/flipkart/user/register
- In Request Body, select raw and chose content type as JSON as shown highlighted.
- Now Pass JSON request payload what we prepared.
- Click on Send.
- We will Get response now.



Here, Spring will convert our JSON data to JAVA object with help of Jackson API internally, when are used @RequestBody annotation with method param of Java Type.

NOTE: In Spring REST APIs, Spring uses 'application/json' as a default media type. That is why, a REST controller can consume or produce JSON format payloads without having to specify the media types explicitly.

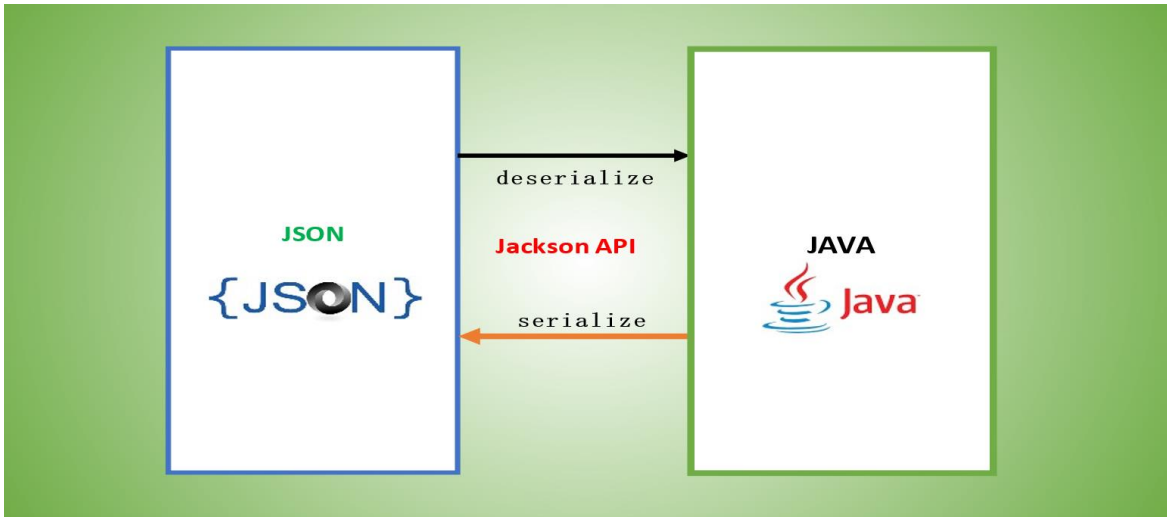
Make sure JSON property names and Java POJO class Property names are same with case sensitive. If Difference exists then we should use @JsonProperty annotation on JAVA property level.

JSON to JAVA Conversion:

Spring boot comes with Jackson API which will take care of un-marshalling JSON request body to Java objects. Jackson dependency is implicit and we do not have to define explicitly. You 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.

And as you have annotated with @RestController there is no need to do explicit json conversion. Just return a POJO and jackson serializer will take care of converting to json. It is equivalent to using @ResponseBody when used with @Controller. Rather than placing @ResponseBody on every controller method we place @RestController instead of vanilla @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 Lsit<String> with JSON key name

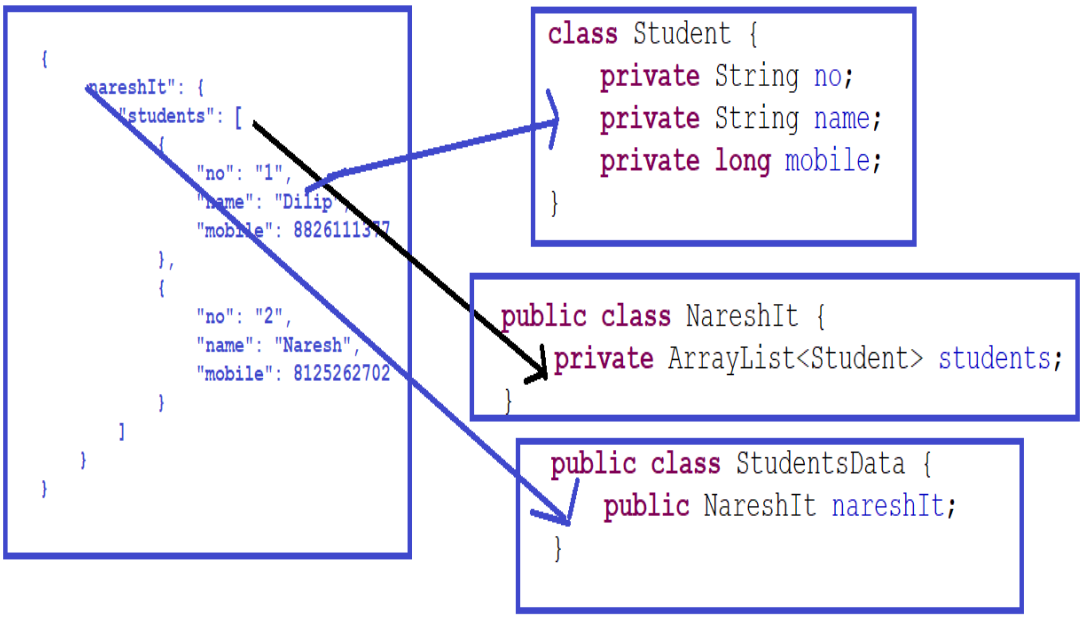
```
import java.util.List;

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 value**

```
{
   "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 JOSN data to Java Object with **@RequestBody** annotation.

JAVA Object to JSON Payload Conversion i.e. JSON Response of endpoint:

When we returns a user defined type from endpoint method, it will be converted as JSON format. Let's take a class as below.
Java Class : **ProductDetails.java**

```
public class ProductDetails {

	private String name;
	private double price;
	private String companyName;
	private String contactEmail;
	private long contactNumber;

	public ProductDetails(String name, double price, String companyName, String contactEmail,
                                                                    long contactNumber) {
		super();
		this.name = name;
		this.price = price;
		this.companyName = companyName;
		this.contactEmail = contactEmail;
		this.contactNumber = contactNumber;
	}
	public ProductDetails() {
		super();
	}
	//Setters and Getters
}
```

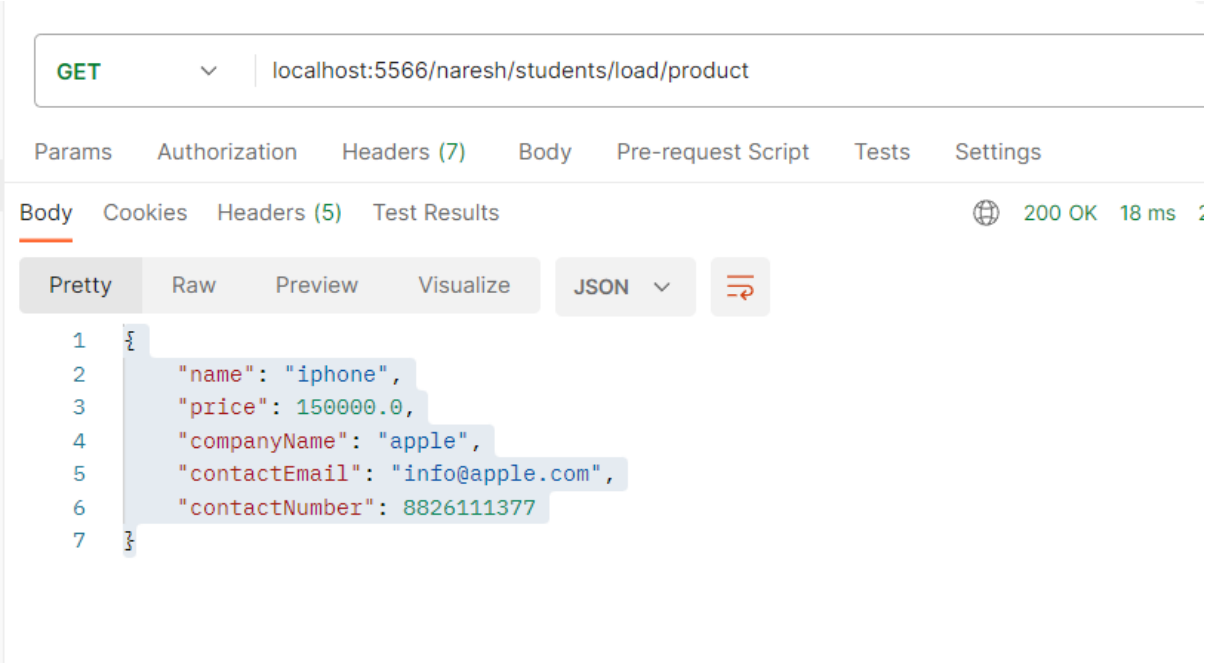
- **Return above Java lass with endpoint method.**

```
	@RequestMapping(path = "/load/product", method = RequestMethod.GET)
	public ProductDetails loadProductDetails() {
		ProductDetails details = new ProductDetails("iphone", 150000.00, "apple",
		"info@apple.com", 8826111377l);
		return details;
	}
```

Now See, By default Spring converted your Java Object Data to Json Response Payload format internally by using Jackson API.

Response:


```
{
  "name": "iphone",
  "price": 150000.0,
  "companyName": "apple",
  "contactEmail": "info@apple.com",
  "contactNumber": 8826111377
}
```

@RequestMapping Consumes and Produces:

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.

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..

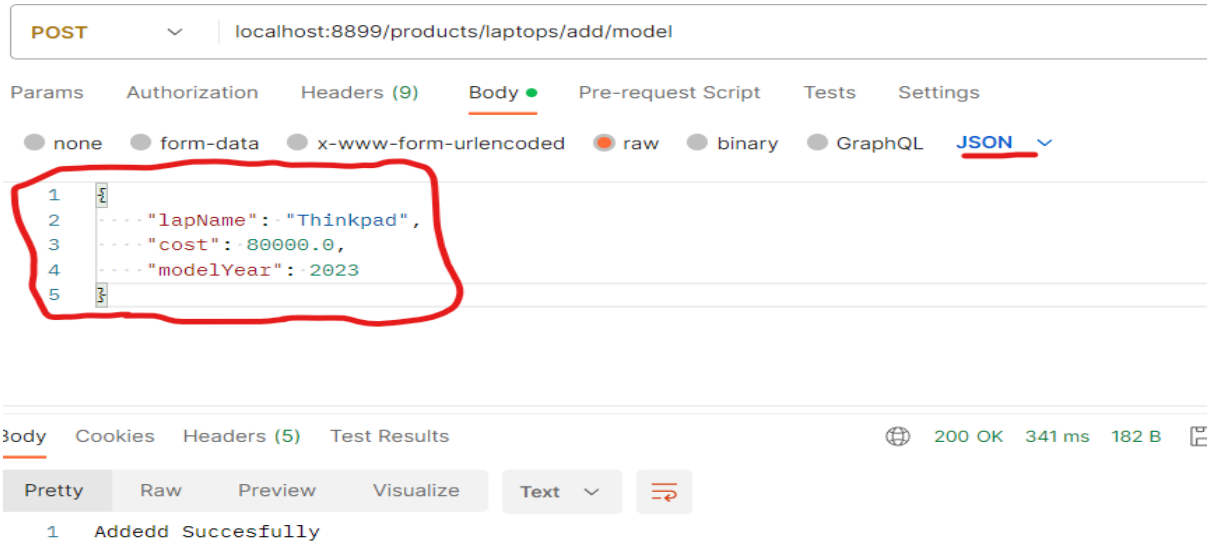
Now let's have an example, as below shown. Created an endpoint method, which accepts only JSON data Request by providing consumes ="application/json".

```
@RequestMapping(path = "/add/model", consumes ="application/json", method =
RequestMethod.POST)
public String addLaptopDetails(@RequestBody LaptopDetails details) {

        return "Addedd Succesfully";
}
```

LaptopDetails.java : To Bind Request Body of JSON

```
public class LaptopDetails {
        private String lapName;
        private double cost;
        private int modelYear;

        //Setters and Getters
}
```

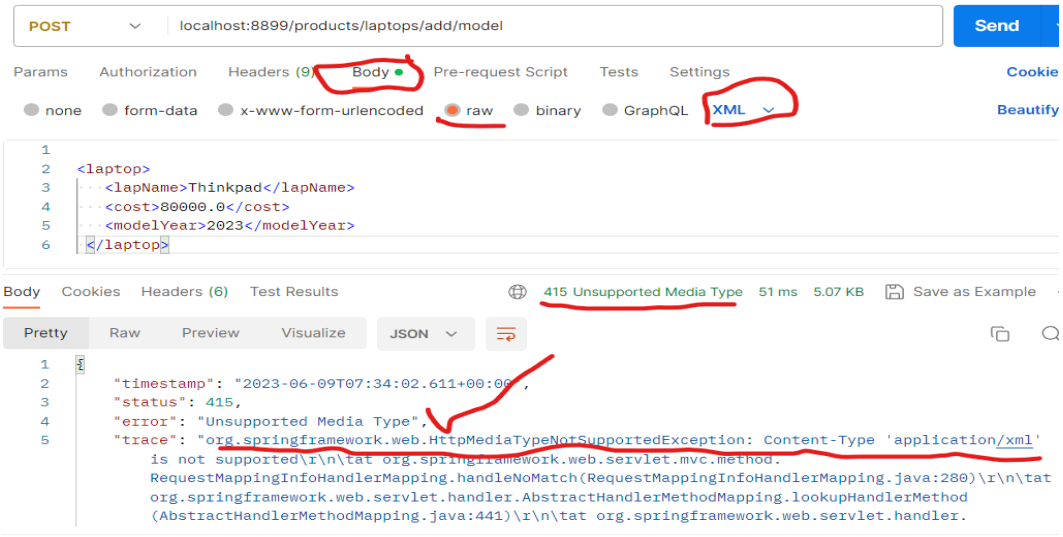
- Now Trigger Endpoint with JSON data in Request Body.



- **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.HttpMediaTypeNotSupportedExc
eption: 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.

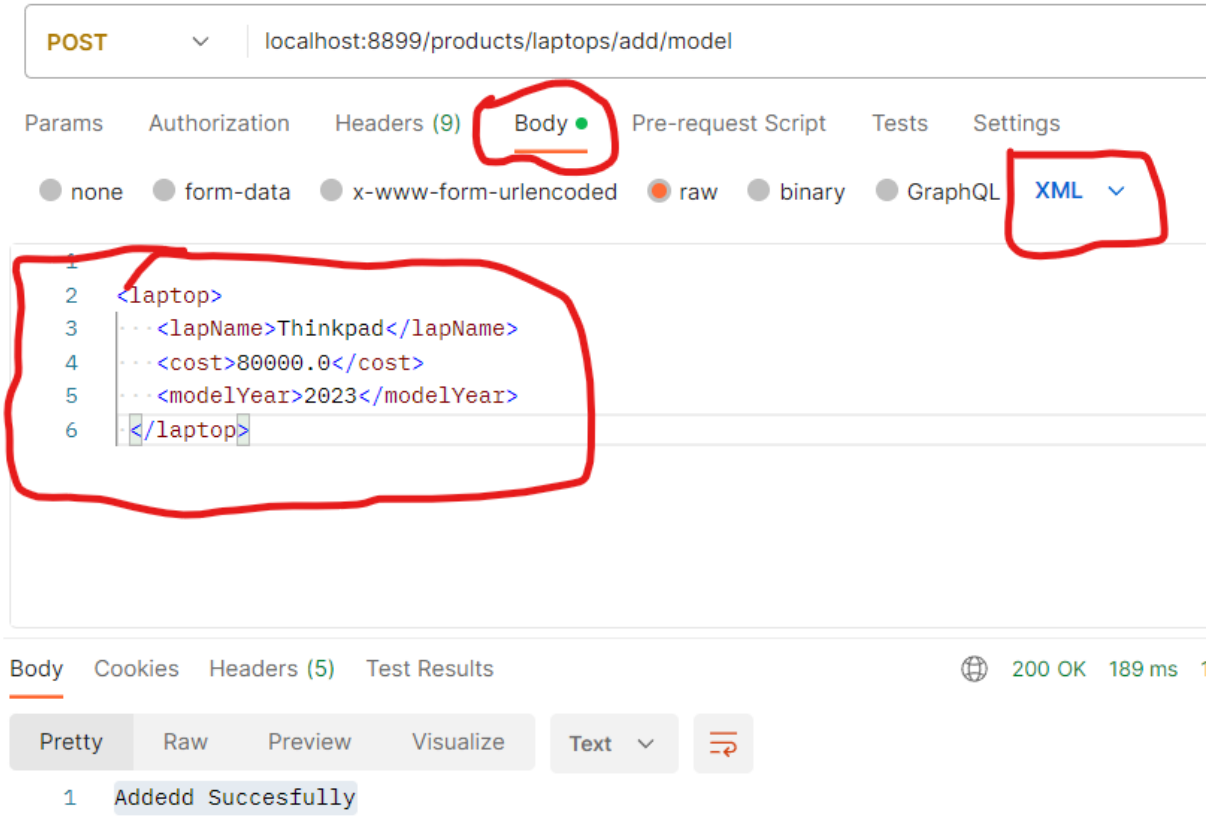
```
<dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
<dependency>
        <groupId>org.glassfish.jaxb</groupId>
        <artifactId>jaxb-runtime</artifactId>
</dependency>
```

Now we can access an API with the request header "Accept: application/xml" and then API will respond with XML response.

Below endpoint accepts only XML request body.

```
@RequestMapping(path = "/add/model", method = RequestMethod.POST,
                        consumes ="application/xml")
public String addLaptopDetails(@RequestBody LaptopDetails details) {
                return "Addedd Succesfully";
}
```

- Now Trigger Endpoint with XML Request data in Body.



- **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 "Addedd 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}

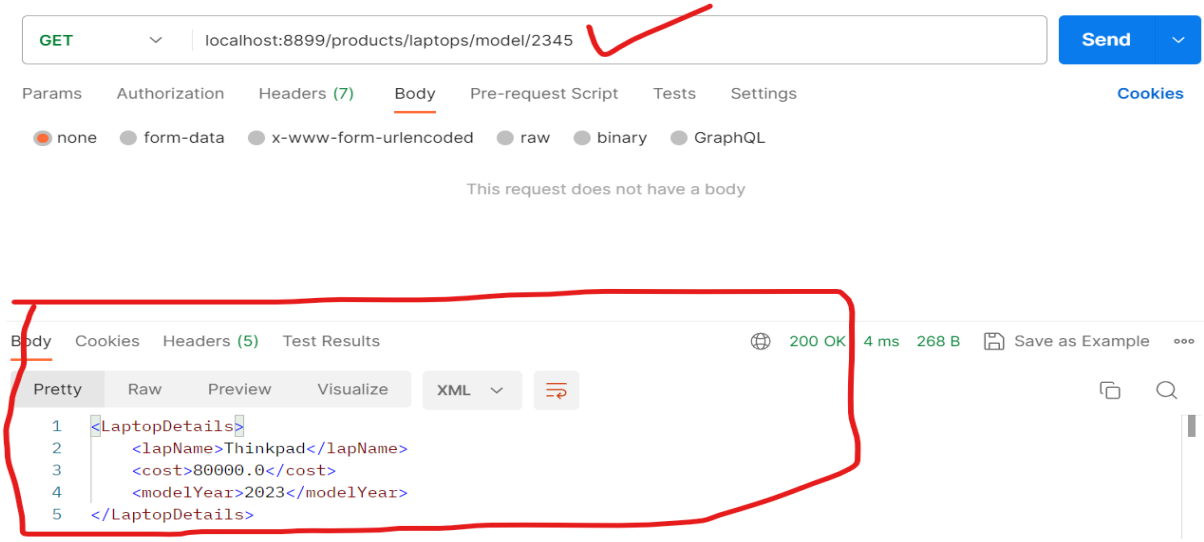
produces: with **produces** attributes, we can configure which type of Response data should be generated from Response object.

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.

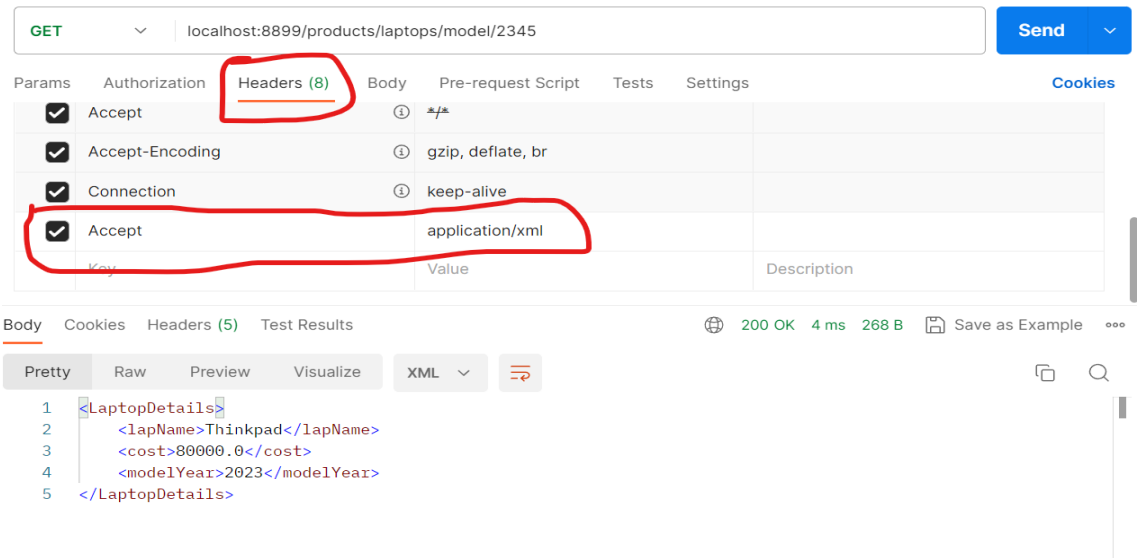


Creating an Endpoint Producing both JSON and XML response.

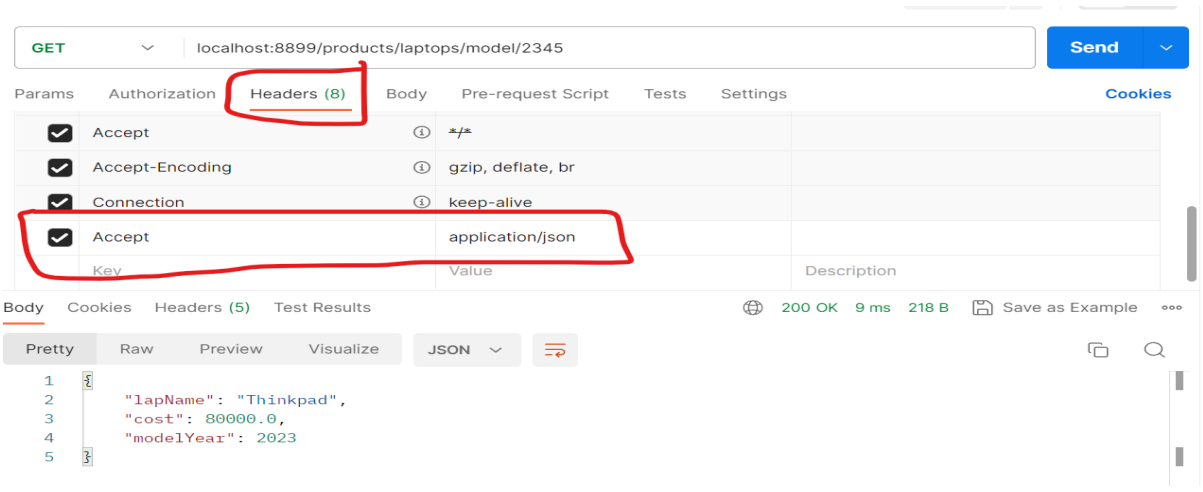
Configure Request mapping with **produces** attribute supporting both Media Types values i.e. array of values. So that now this endpoint generates either XML or JSON response depends on header **Accept** and its value. The HTTP **Accept** header is a request type header. The Accept header is used to 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.



Request for JSON response: Add Header **Accept** and value as **application/json** as shown.



@SpringBootApplication Annotation:

Spring Boot @SpringBootApplication annotation is used to mark a configuration class that declares one or more @Bean methods and also triggers auto-configuration and component scanning. It's same as declaring a class with @Configuration, @EnableAutoConfiguration and @ComponentScan annotations.

@EnableAutoConfiguration – This enables Spring Boot's autoconfiguration mechanism. Auto-configuration refers to creating beans automatically by scanning the classpath.

@Configuration – Designates the class as a configuration class for Java configuration. In addition to beans configured via component scanning, an application may desire to configure some additional beans via the @Bean annotation as demonstrated here. Thus, the return value of methods having the @Bean annotation in this class are registered as beans.

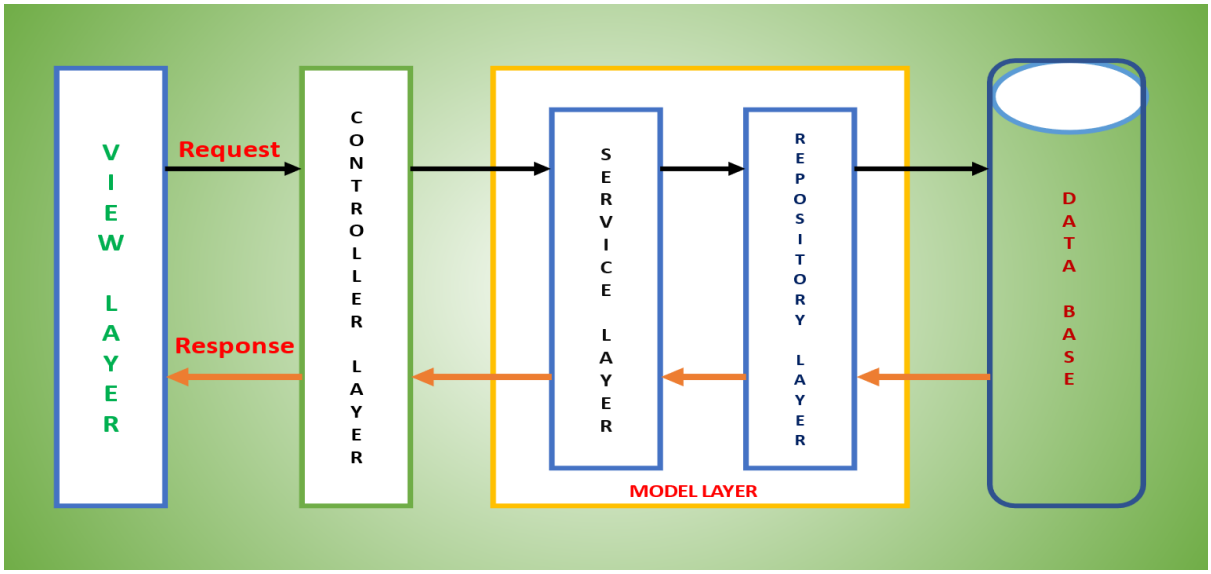
@ComponentScan – Typically, in a Spring application, annotations like @Component, @Controller, @Service, @Repository are specified on classes to mark them as Spring beans. The @ComponentScan annotation basically tells Spring Boot to scan the current package and its sub-packages in order to identify annotated classes and configure them as Spring beans. Thus, it designates the current package as the root package for component scanning.

SpringApplication class:

SpringApplication used to bootstrap and launch a Spring application from a Java main method. This class automatically creates the ApplicationContext from the classpath, scan the configuration classes and launch the application. This class is very helpful in launching Spring MVC or Spring REST application using Spring Boot.

- The Main class has the @SpringBootApplication annotation
- It simply invokes the SpringApplication.run method. This starts the Spring application as a standalone application, runs the embedded servers and loads the beans.

Service Layer: A service layer is a layer in an application that facilitates communication between the controller and the persistence layer. Additionally, business logic is stored in the 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.



@Service is annotated on class to say spring boot, this is my Service Layer.

Create An Example with Service Layer:

- Create Spring Boot MVC Application.
- Create Controller and Service classes.

Controller Class:

```
@RestController
@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.

```
@RestController
@RequestMapping("/admission")
public class UniversityAdmissionsController {

        //Injecting Service Class Object
        @Autowired
        UniversityAdmissionsService service;
        //Logic
}
```

Requirement: Now create endpoints inside controller to add admission details and as well as loading admission details.

POJO class of Admission:

Admission.java

```
public class Admission {

	private String name;
	private double cgpa;
	private long contact;

	// Setters and Getters
}
```

UniversityAdmissionsController.java

```
@RestController
@RequestMapping("/admission")
public class UniversityAdmissionsController {

	@Autowired
	UniversityAdmissionsService service;
	//Creating Admission
	@PostMapping("/create")
	public String createAdmission(@RequestBody Admission admission) {
		String result = service.createAdmission(admission);
		return result;
	}
	//Fetching Admission Details on ID
	@GetMapping("/1123353")
	public Admission getAdmissionDetails() {
		Admission = service.getAdmissionDetails("1123353");
		return admission;
	}
}
```

From above, We are passing information from controller to service layer. Now inside Service class, we are writing Business Logic and data pass to persistence layer for endpoints.

```
@Service
public class UniversityAdmissionsService {

	public String createAdmission(Admission admission) {
		boolean isCreated = false;
		String result = null;
		// Forwarding Data to Repository layer
		// Depends on DB operation, Generating return value returns to controller
```

```
                if (isCreated) {
                        result = "Admission Created Succesdfully";
                } else {
                        result = "Admission Not Crated Succesdfully. Try Again";
                }
                return result;
        }

        public Admission getAdmissionDetails(String admissionId) {
                // From DB, you got Admission Records
                Admission ad = new Admission();
                ad.setCgpa(88.99);
                ad.setContact(88888);
                ad.setName("Dilip Singh");
                return ad;
        }
}
```

Now returning values of service methods are passed to Controller level. This is how we are using service layer with controller layer. Now we should integrate Service layer with Data 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 in a Spring Boot Application. Spring Data is considered a Spring based programming model for accessing data. A huge amount of code is required for working with the databases, which can be easily reduced by Spring Data. It consists of multiple modules. There are many Spring applications that use JPA technology, so these development procedures can be easily simplified by Spring Data JPA.

Spring Data JPA:

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.

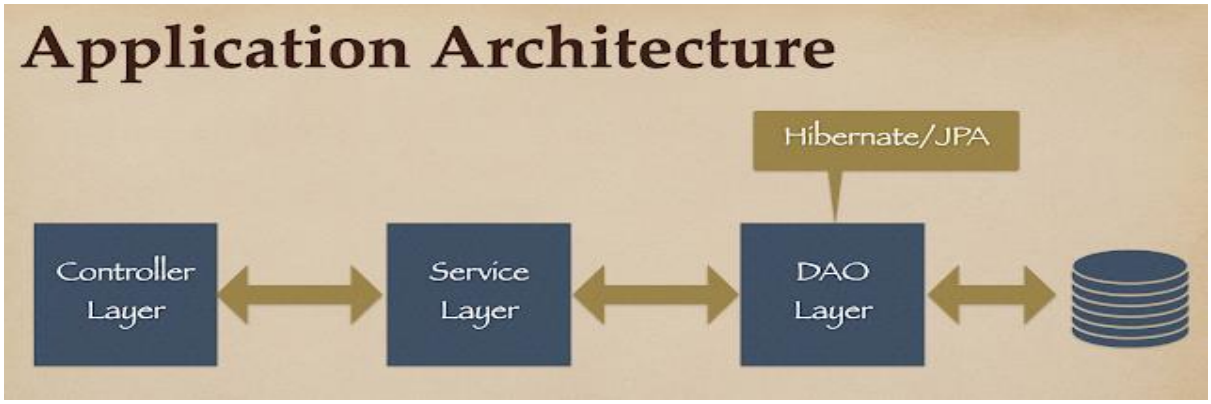
- Spring Data JPA is not a JPA provider. It is a library/framework that adds an extra layer of abstraction on top of our JPA provider (like Hibernate).
- Spring Data JPA uses Hibernate as a default JPA provider.

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.



ORM(Object-Relational Mapping))

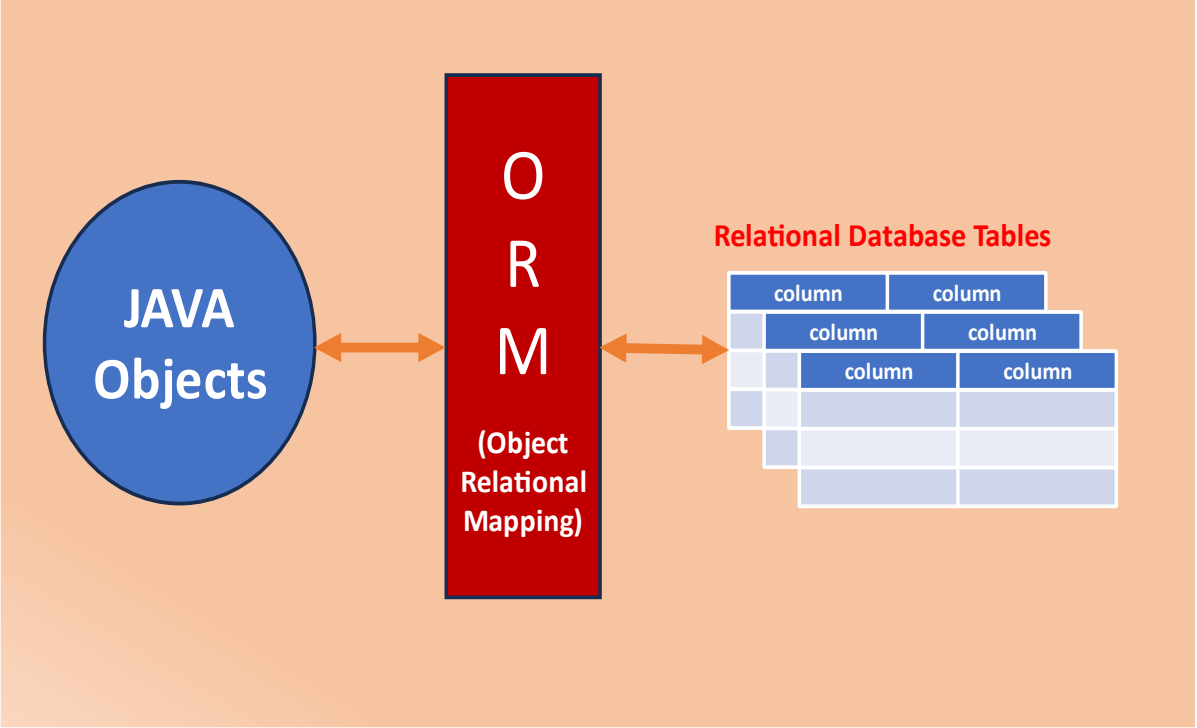
ORM(Object-Relational Mapping) is the method of querying and manipulating data from a database using an object-oriented paradigm/programming language. By using this method, we are able to interact with a relational database without having to use SQL. Object Relational Mapping (ORM) is a functionality 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.

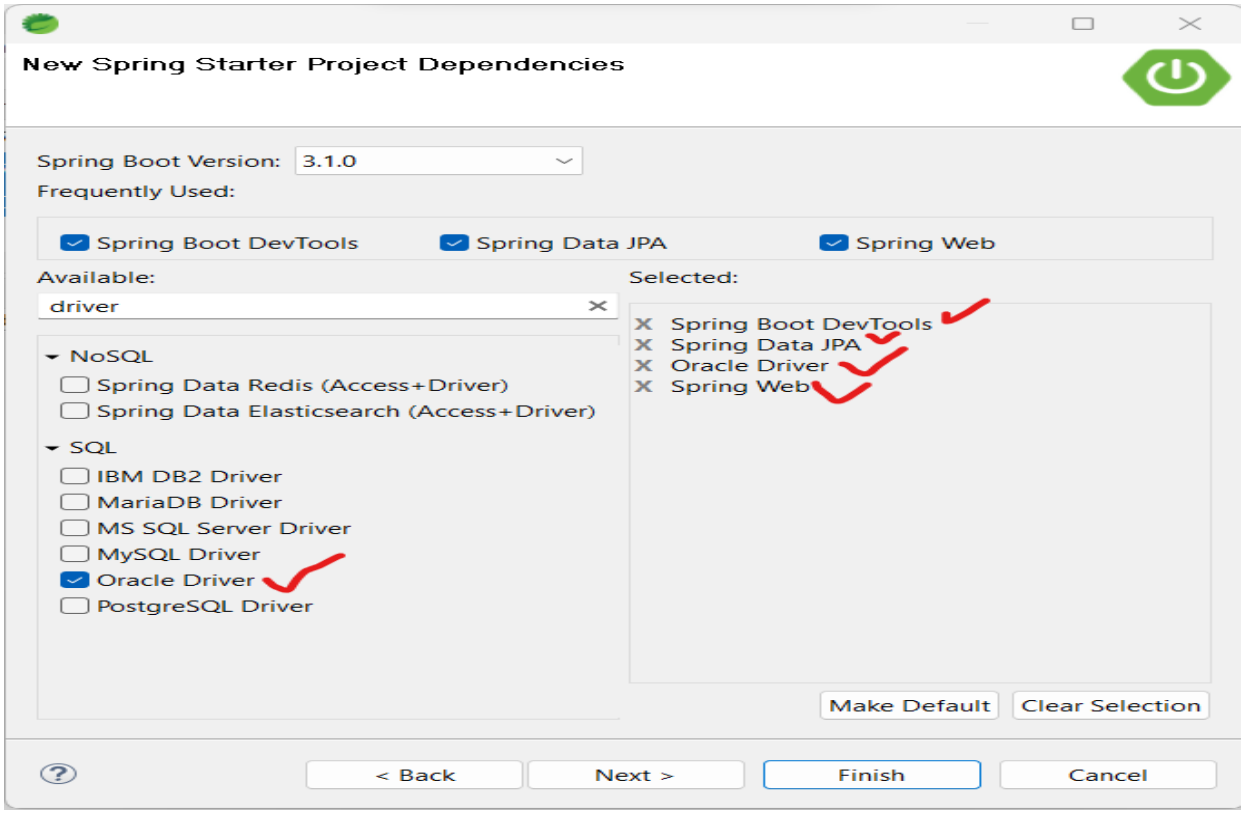


Creating Web and JPA Module Project:

Note: Please Make Sure DB installed in your computer. Here I am taking examples and training with Oracle Database.

Step 1: Create Spring Boot application by Selecting Web and JPA Modules along with Database Driver.

Note: Please choose Driver depends on which database you were integrating with application.



Step 2: After Project creation, we should configure Database Properties like URL, User Name and Password of Database inside **application.properties** file. The Properties value are similar however what we used in JDBC API programming level.

application.properties

```
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
```

Adding other properties for port and context path, please start your application.

application.properties

```
server.port=6677
server.servlet.context-path=/appolo

#add Db properties
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
```

```
Problems  Javadoc  Declaration  Console  History
pharmacy - PharmacyApplication [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  (14-Jun-2
          : HHH035001: Using dialect: org.hibernate.dialect.OracleDialect, version
or        : HHH000021: Bytecode provider name : bytebuddy
tor       : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.tra
ctoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
iguration : spring.jpa.open-in-view is enabled by default. Therefore, database que
ver       : LiveReload server is running on port 35729
ebServer  : Tomcat started on port(s): 6677 (http) with context path '/appolo'
          : Started PharmacyApplication in 0.619 seconds (process running for 152.
```

i.e. Our Basic Project setup completed with DB integration, once we can see Server started log in console. If we have anything wrong in configuration, we will get exceptions when we start application.

Details of 3 Properties:

Name	Description
spring.datasource.url	JDBC URL of the database.
spring.datasource.username	Login username of the database.
spring.datasource.username	Login password of the database.

Step 3: Now we can Add Controller and Service Layer classes.

PharmacyController.java

```
@RestController
public class PharmacyController {
        @Autowired
        PharmacyService pharmacyService;

}
```

PharmacyService.java

```
@Service
public class PharmacyService {

}
```

Step 4: Now create a entity class. Before creating entity class we should have Database table details with us i.e. POJO to Table mapping.

For example In our requirement, I would like to store information of pharmacy shops details like Location Name, Contact Number and Pincode in side DB. Created Table in DB as below.

Table Name: pharmacy_location
Column Names:

1. location_name
2. contact_number
3. pincode

SQL script :

```
create table pharmacy_location(location_name varchar2(50),contact_number varchar2(14),pincode number(6));
```

Now create a POJO class aligned to DB table name and columns of data types.

1. We should mark class with @Entity annotation, to make sure POJO as entity class
2. We should mark class with @Table and pass DB table name value for **name** attribute
3. We should mark POJO properties with @Column annotation and Table column name

Annotations:

@Entity: Specifies that the class is an entity. This annotation is applied to the entity class.
@Table: Specifies the primary table for the annotated entity.
@Column: Specifies the mapped column for a persistent/POJO property or field.
@Id: The field or property to which the Id annotation is applied should be one of the following types: any Java primitive type; any primitive wrapper type; String; java.util.Date; java.sql.Date; java.math.BigDecimal; java.math.BigInteger. The mapped column for the primary key of the entity is assumed to be the primary key of the primary table.

PharmacyLocation.java

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "pharmacy_location")
public class PharmacyLocation {

	@Id
	@Column(name="contact_number")
	private String conatcNumber;

	@Column(name = "location_name")
	private String locationName;

	@Column(name="pincode")
	private int pincode;

	public String getLocationName() {
		return locationName;
	}
	public void setLocationName(String locationName) {
		this.locationName = locationName;
	}
	public String getConatcNumber() {
		return conatcNumber;
	}
	public void setConatcNumber(String conatcNumber) {
		this.conatcNumber = conatcNumber;
	}
	public int getPincode() {
		return pincode;
	}
	public void setPincode(int pincode) {
```

```
            this.pincode = pincode;
        }
    }
```

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. If both are different then we should use @Column annotation along with value.

For Example : In Above Entity class, we written as

```
@Column(name="pincode")
private int pincode;
```

In this case we can define only property name i.e. internally Spring JPA considers as pincode is aligned with **pincode** column in table

```
private int pincode;
```

Step 5: Now Add Repository Layer for DB operations.

Defining Repository Interfaces:

In Spring/Spring Boot, we should define Interfaces instead of classes for Repository Layer. We will use an annotation called as **@Repository. @Repository** Annotation is a specialization of **@Component** annotation which is used to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects. Though it is a specialization of **@Component** annotation, so Spring Repository classes are autodetected by spring framework through classpath scanning.

In order to start leveraging the Spring Data programming model with JPA, a DAO/Repository interface needs to extend the JPA specific Repository interfaces like **JpaRepository**, **CrudRepository**. This will enable Spring Data to find this interface and automatically create an implementation for it. By extending the interface, we get the most relevant CRUD methods for standard data access available in a standard DAO.

- Create an interface by extending Spring provided JPA repositories with **CrudRepository**.

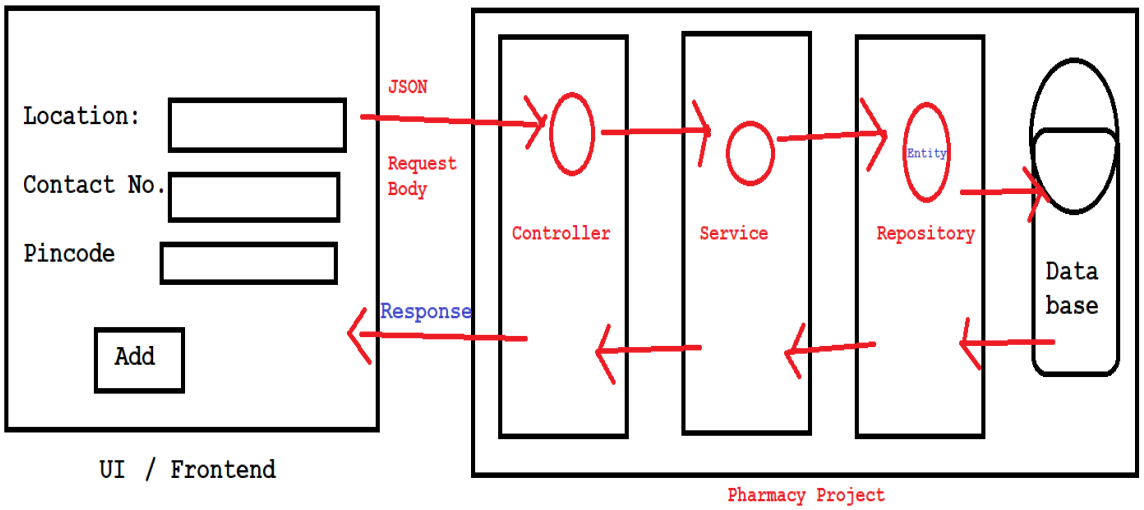
```
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.appolo.pharmacy.entity.PharmacyLocation;

@Repository
public interface PharmacyRepository extends CrudRepository<PharmacyLocation, String>{

}
```

In above as part of **CrudRepository** Generic values, we should pass Entity Class name followed by Data Type of @Id annotated POJO property of same entity class. So Now this repository by defaults works always with DB table configured inside entity class i.e. pharmacy_location.

Now we can integrate Repository layer with Service layer. i.e. Auto wiring.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class PharmacyService {
        @Autowired
        PharmacyRepository repository;
}
```

Now as per Requirement, we have to store Pharmacy Location Details in Table i.e. we are going to get Data as part of Request.



➢ **So let's create an point inside controller class.**

```
@PostMapping("/add/store/location")
public String addPharmacyStoreDetails(@RequestBody PharmacyLocationRequest request) {
        String result = pharmacyService.addPharmacyStoreDetails(request);
        return result;
}
```

Create Request POJO class to bind incoming JSON Request with @RequestBody.
PharmacyLocationRequest.java

```
public class PharmacyLocationRequest {
        private String locationName;
```

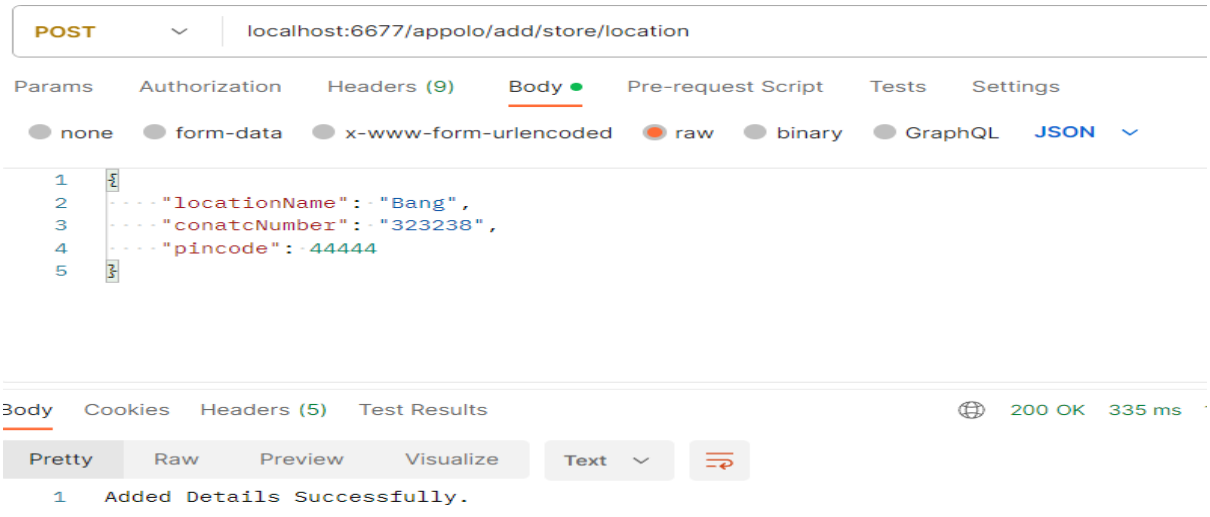
```
        private String conatcNumber;
        private int pincode;
        //Setters & Getters
    }
```

Now create new method in Service layer class to pass request body data from controller.

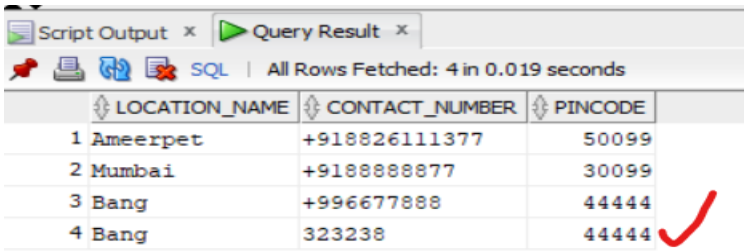
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.appolo.pharmacy.entity.PharmacyLocation;
import com.appolo.pharmacy.repository.PharmacyRepository;
import com.appolo.pharmacy.requests.PharmacyLocationRequest;

@Service
public class PharmacyService {
    @Autowired
    PharmacyRepository repository;

    public String addPharmacyStoreDetails(PharmacyLocationRequest request) {
        // Object of  Entity class
        PharmacyLocation location = new PharmacyLocation();
        location.setConatcNumber(request.getConatcNumber());
        location.setLocationName(request.getLocationName());
        location.setPincode(request.getPincode());
        repository.save(location);
        return "Added Details Successfully.";
    }
}
```

Now we no need to write any logic inside Repository to utilize predefined functionalities of Spring JPA module.

Now trigger request from postman and check data persisted or not in DB table.



Check in Data base Table now.



	LOCATION_NAME	CONTACT_NUMBER	PINCODE
1	Ameerpet	+918826111377	50099
2	Mumbai	+9188888877	30099
3	Bang	+996677888	44444
4	Bang	323238	44444

Internally, Insert SQL Query will be generated by Hibernate as shown below when executing predefined save() method on repository interface which inherited method of CrudRepository.

Hibernate: insert into pharmacy_location (location_name,pincode,contact_number) values (?,?,?)

This is how we can reuse JPA functionalities for all CRUD operations of DB instead of Defining SQL Queries.

Let's have Other scenarios or Requirements.

Requirement: Please get Pharmacy Location Details with contact number

i/p : contact Number

o/p : o or 1 (Because of Primary key Column)

Location Details:

pin code, location name and contact number

Define an endpoint inside controller for passing contact Number and get the response of Location Details.

```
@GetMapping("/load/contact")
public PharmacyLocationResponse getLocationDetailsByContactNumber(@RequestBody
ContactNumberRequest request) {

PharmacyLocationResponse result = pharmacyService.getLocationDetailsByContactNumber(request);
        return result;
}
```

- **Creating Request class with contact number property.**

ContactNumberRequest.java

```
public class ContactNumberRequest {
        private String conatcNumber;
        //Setter & Getters
}
```

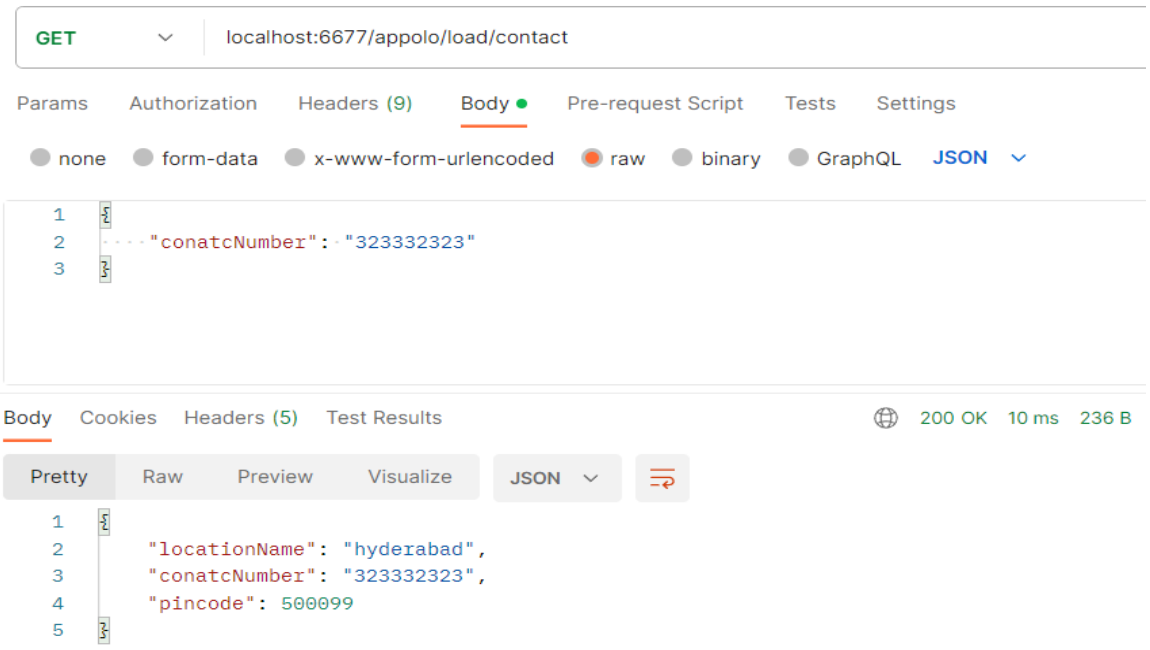
➢ **Now create method at service layer.**

```
public PharmacyLocationResponse
getLocationDetailsByContactNumber(ContactNumberRequest request) {

Optional<PharmacyLocation> data = repository.findById(request.getConatcNumber());
	PharmacyLocationResponse response = null;
	if (data.isPresent()) {
		PharmacyLocation location = data.get();
		// entity Object to Response Object OJO
		response = new PharmacyLocationResponse();
		response.setConatcNumber(location.getConatcNumber());
		response.setLocationName(location.getLocationName());
		response.setPincode(location.getPincode());
	}
	return response;
}
```

Create Response class : **PharmacyLocationResponse.java**

```
public class PharmacyLocationResponse {

	private String locationName;
	private String conatcNumber;
	private int pincode;

	public PharmacyLocationResponse() {
		super();
	}

public PharmacyLocationResponse(String locationName, String conatcNumber, int pincode) {
		super();
		this.locationName = locationName;
		this.conatcNumber = conatcNumber;
		this.pincode = pincode;
	}

	// Setters and Getters
}
```

➢ **Now Test our endpoint**



Requirement: Please get Pharmacy Location Details with location name

I/P : location name
O/P : 0 or more records of Location Details pin code, location name and contact

i.e. We will get more records with one location value because it's not a unique constraint, We will get List of Records.

Derived Query Methods in Spring Data JPA Repositories:

For simple queries, it's easy to derive what the query should be just by looking at the corresponding method name in our repository layer.

Structure of Derived Query Methods in Spring:

Derived method names have two main parts separated by the first By keyword.

The first part — such as **find** — is the introducer, and the rest — such as **ByName** — is the criteria. Spring Data JPA supports find, read, query, count and get.

For example, Assume we have a table in database called as user and aligned JPA class **User.** Now when we are trying retrieve information for columns where we will get more than one records i.e. NO unique Constraint columns. In Such case we will get more records in result set. Now JPA will convert every record as an entity class object and returns as List of Entity class objects.

primarykey/unique column = 0 or 1 Record i.e. Single Object of User Entity Class.
Non-Primary key/Unique Coloumn = 0 or more records i.e. list of User Objects List<User>

List<User> findByName(String name);

And we can add **Is** or **Equals** for readability: **List<User> findByNameIs(String name);**
List<User> findByNameEquals(String name);
This extra readability comes in handy when we need to express inequality instead:
List<User> findByNameIsNot(String name);

These Derived Query methods will be created in Repository layer interface as an Abstract methods and those will be called from service layer.

Create endpoint method in controller.

```
@GetMapping("/load/location")
public List<PharmacyLocationResponse> getLocationDetailsBylocationName(@RequestBody
LocationNameRequest request) {

		List<PharmacyLocationResponse> response =
		pharmacyService.getLocationDetailsBylocationName(request);
		return response;
}
```

Request Object class: LocationNameRequest.java

```
	public class LocationNameRequest {
		private String locationName;
		//Setter and Getter
	}
```

Create method with Derived Query Method inside Repositor.

```
import java.util.List;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.appolo.pharmacy.entity.PharmacyLocation;

@Repository
public interface PharmacyRepository extends CrudRepository<PharmacyLocation, String> {
	List<PharmacyLocation> findByLocationName(String locationName);
}
```

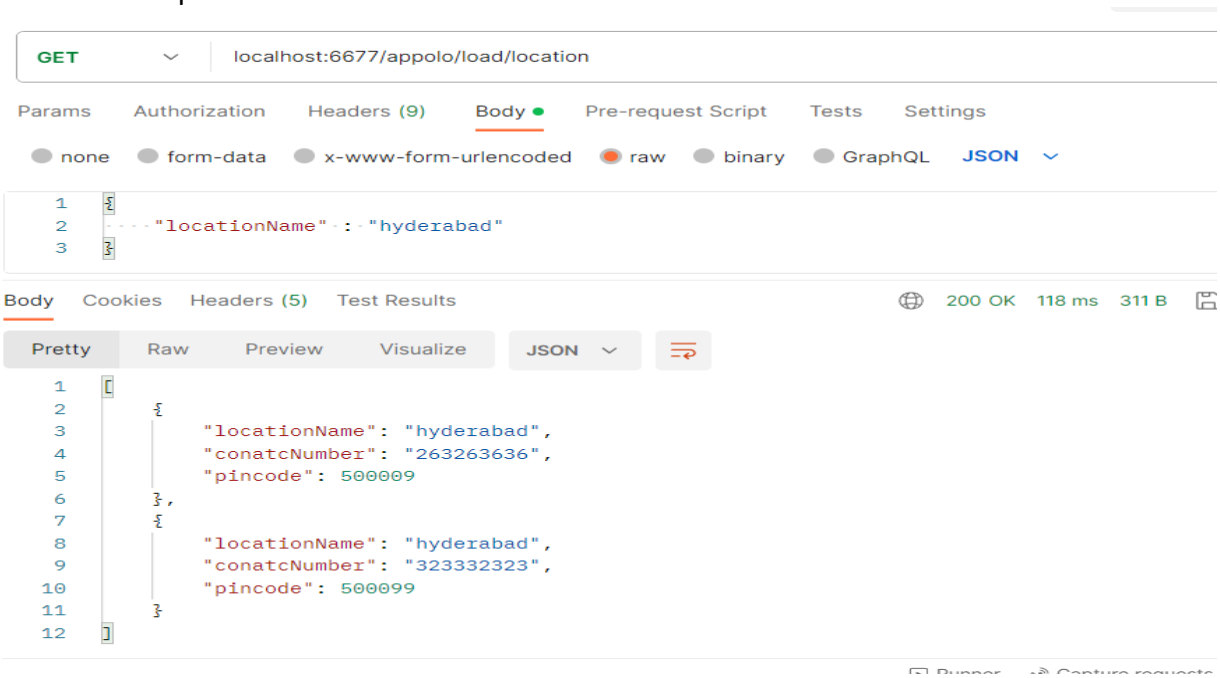
Create method in Service Layer Method :

```
public List<PharmacyLocationResponse>
getLocationDetailsBylocationName(LocationNameRequest request) {

String locationName = request.getLocationName();
List<PharmacyLocation> data = repository.findByLocationName(locationName);

//Converting List of Entity Objects to List of response class POJO Objects
List<PharmacyLocationResponse> listOfLocations = data.stream().map(
                                                    l -> new PharmacyLocationResponse(
                                                            l.getLocationName(),
                                                            l.getConatcNumber(),
                                                            l.getPincode()
                                                    )
                                                ).collect(Collectors.toList());

                return listOfLocations;
}
```

In above service, method we are converted List of Entity class PharmacyLocation objects to equivalent Response POJO class Objects PharmacyLocationResponse because Controller layer expecting same List of Objects instead of Entity Objects directly. It's gudlines or recommendation as per real time projects like we should use entity classes at controller layer level.

i.e. we have to maintain separate classes for Request, Response and Entity layers level.

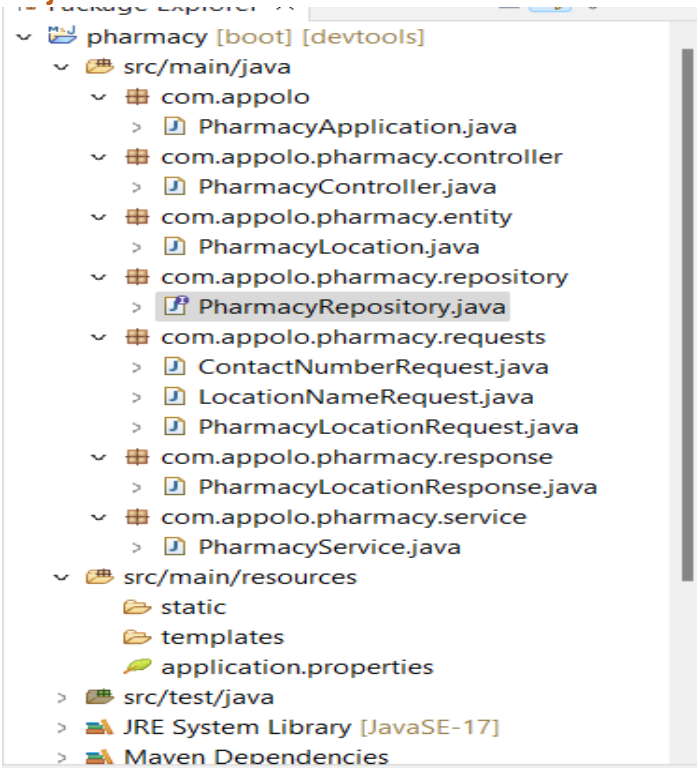
Now Test endpoint.



We got response as List of Values i.e. JSON array of Objects. Internally JPA dynamically created a SQL query as below, when Derived Query Method executed.

Hibernate: select p1_0.contact_number,p1_0.location_name,p1_0.pincode from pharmacy_location p1_0 where p1_0.location_name=?

Project Structure:



So we can derive multiple query methods for different columns and criteria's as per JPA guidelines. We will have more examples further.

Path Variables in Controller Endpoint Method Mappings :

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;

}
```

Requirement : Please get Pharmacy Location Details with location name.
In this case, we are passing single value of location name to find out all Pharmacy Location Details including contact, pincode details. Now we can take **Path variable** here to fulfil this requirement.

URI: /location**/{locationName}**

- **Create an endpoint method in controller with above URI contains path variable.**

```
@GetMapping("/location/{locationName}")
public List<PharmacyLocationResponse> loadLocationByLocationName(
        @PathVariable("locationName") String locationName) {

        //Passing Path variable value to service method
        List<PharmacyLocationResponse> response =
        pharmacyService.loadLocationByLocationName(locationName);
        return response;
}
```

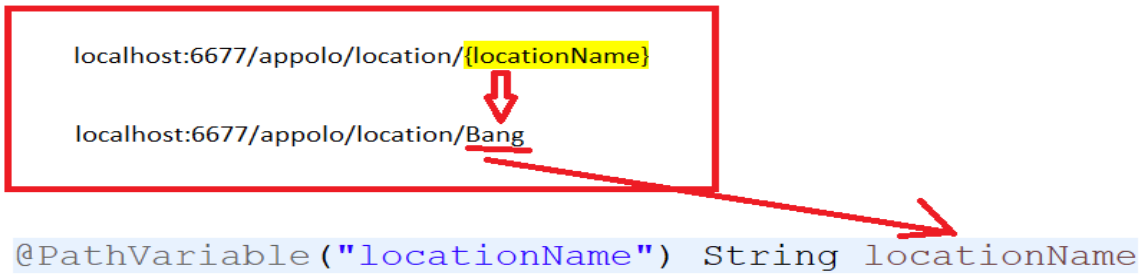
- **Now create a method in Service layer.**

```
public List<PharmacyLocationResponse> loadLocationByLocationName(String locationName) {

// List of entity class Objects from repository
List<PharmacyLocation> data = repository.findByLocationName(locationName);

// Mapping Entity Objects Data to Response Objects Data to transfer Controller layer
List<PharmacyLocationResponse> listOfLocations = data.stream()
                                        .map(
                                                l -> new PharmacyLocationResponse(
                                                        l.getLocationName(),
                                                        l.getConatcNumber(),
                                                        l.getPincode()
                                                )
                                        ).collect(Collectors.toList());

  return listOfLocations;
}
```

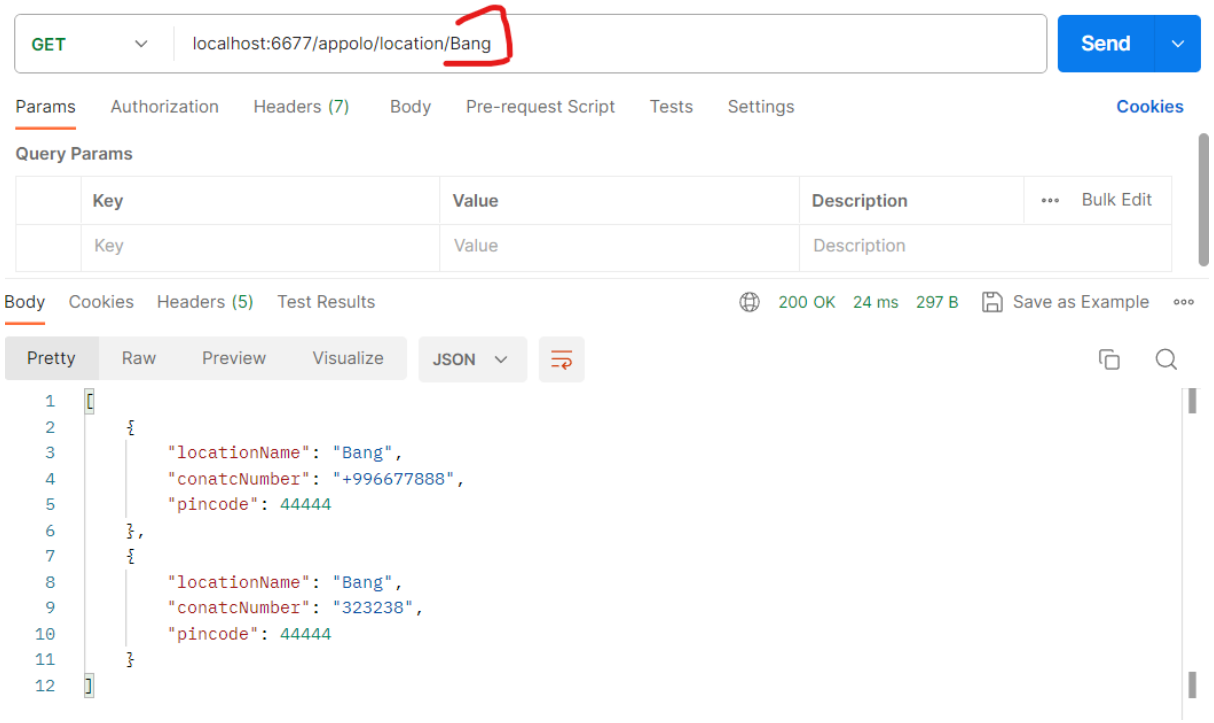
➢ **Now create a Derived Query method in Repository layer.** PharmacyRepository.java

List<PharmacyLocation> findByLocationName(String locationName);

Test end point: URL formation, replacing Path variable place with real value of variable



From Postman : URL with Path variable value.



Multiple Path Variable as part of URI:

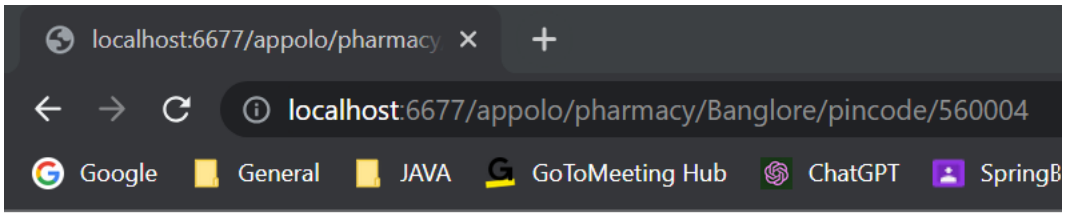
We can define more than one path variables as part of URI, then equal number of method parameters with @PathVaribale annotation defined in handler mapping method.

NOTE: We no need to define value inside @PathVariable when we are taking method parameter name as it is URI template/Path variable.

Example:

```
@GetMapping("/pharmacy/{location}/pincode/{pincode}")
public String getPharmacyByLocationAndPincode(@PathVariable String location,
                                              @PathVariable  String pincode) {
        return "Location Name : " + location + ", Pincode: " + pincode;
}
```

Test Service/Endpoint:

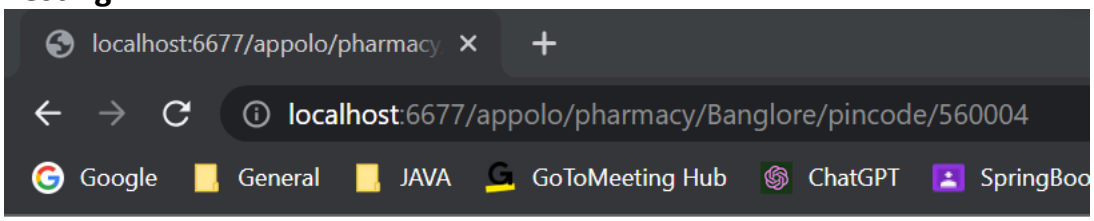


Location Name : Banglore, Pincode: 560004

We can also handle more than one @PathVariable parameter 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");

        if (location != null && pincode != null) {
            return "Location Name : " + location + ", Pin code: " + pincode;
        } else {
            return "Missing Parameters";
        }
}
```

Testing:



Location Name : Banglore, Pin code: 560004

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;
```

```
}
```

URI with Query Params: **details?email=<value>&mobileNumber=<>**

Add Method in Service class: UsersService.java

```
public List<Users> getUsersByEmailOrMobile(String email, String mobileNumber) {

	List<Users> users = repository.findByEmailOrMobileNumber(email, mobileNumber);
	// TODO : Convert List of Entity Objects to List of Response/DTO Objects
	return users;
}
```

Add Derived Query Method in Repository:

```
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.flipkart.entity.Users;

@Repository
public interface UsersRepository extends JpaRepository<Users, String>{
	List<Users> findByEmailOrMobileNumber(String email, String mobileNumber);
}
```

Create Entity Class as per below Table:

```
Create table flipkart_users(first_name varchar2(30), last_name varchar2(30), mobile_number varchar2(15), email varchar2(50) primary key, password varchar2(30), city varchar2(30), pincode number(6));
```

Entity Class: Users.java

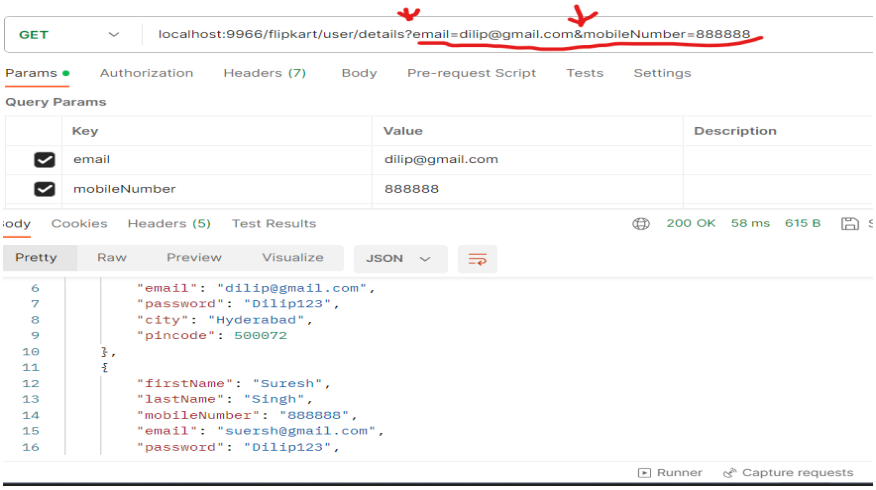
```
	import jakarta.persistence.Column;
	import jakarta.persistence.Entity;
	import jakarta.persistence.Id;
	import jakarta.persistence.Table;

	@Entity
	@Table(name = "flipkart_users")
	public class Users {

		@Column(name = "first_name")
		private String firstName;
		@Column(name = "last_name")
		private String lastName;
		@Column(name = "mobile_number")
		private String mobileNumber;
		@Id
		private String email;
```

```
        // No Need to Define Column Annotation when both DB table column name and
        // Entity POJO Property name are same
        private String password;
        private String city;
        private int pincode;

        //Setters and Getters
}
```

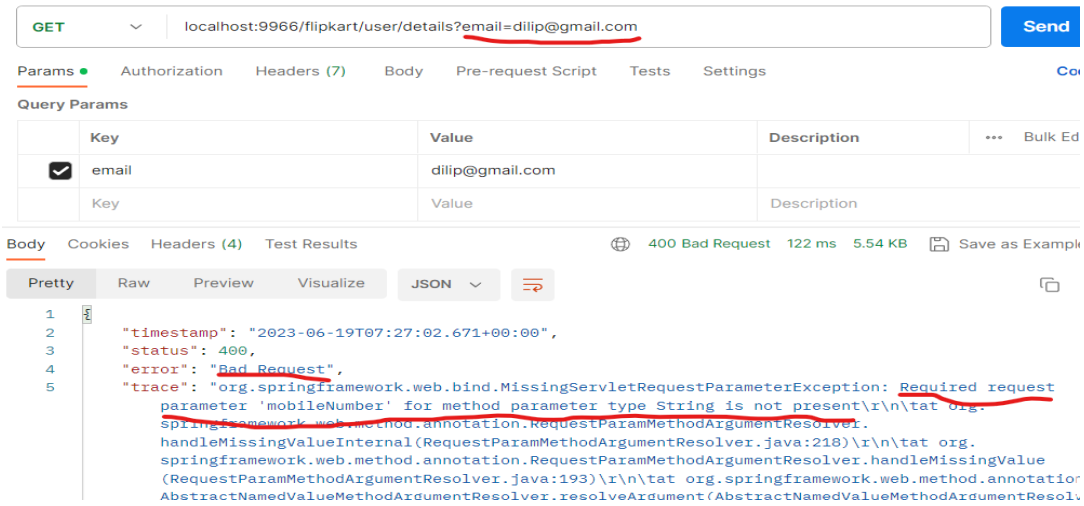
Now execute our endpoint by passing Query Parameters of Query String.
URL: **localhost:9966/flipkart/user/details?email=dilip@gmail.com&mobileNumber=888888**

Query String : ?email=dilip@gmail.com&mobileNumber=888888

Query Parameter 1 : email=dilip@gmail.com
Query Parameter 2 : mobileNumber=888888



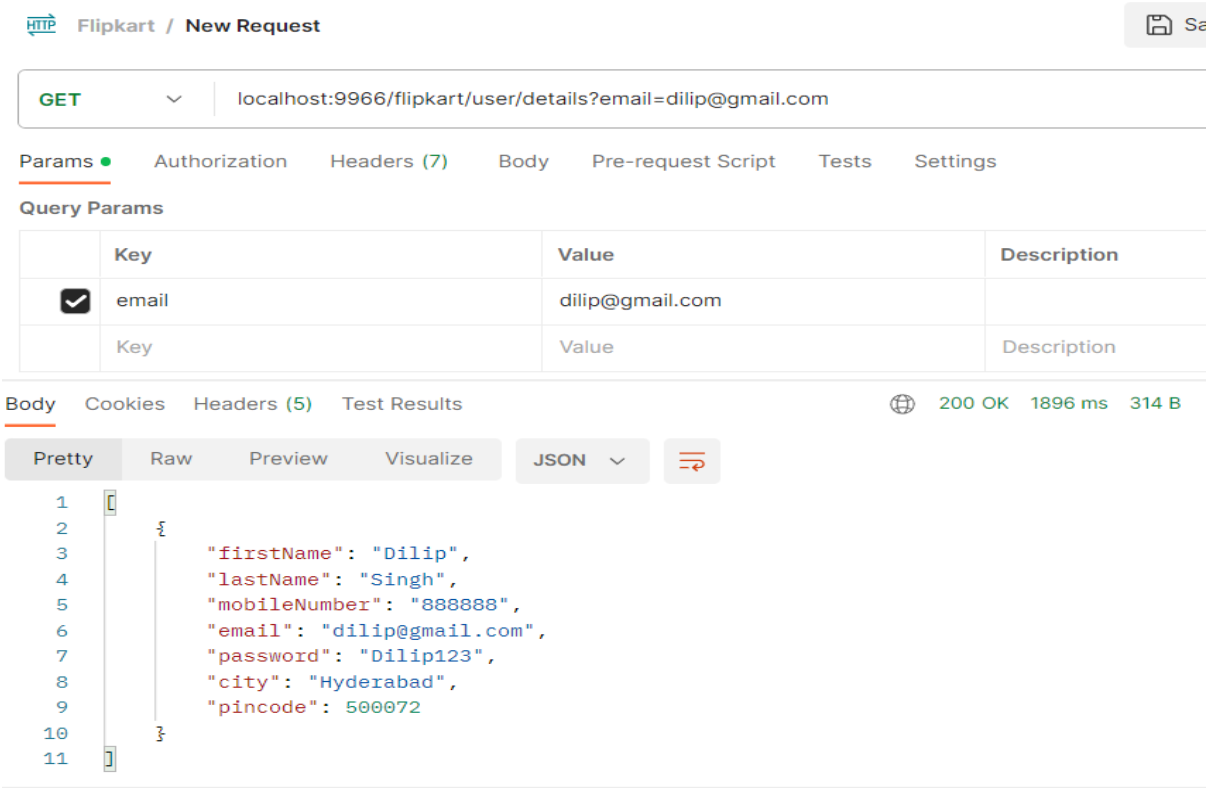
By Default every Request Parameter variable is Required i.e. we should pass Query Parameter and its value as part of URL. If we are missed any parameter, then we will get bad request.



If we want to make sure an optional of any request parameter, then we have to use attribute **required=false** in side @RequestParam annotation. Now lets' 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 is missing in URI.



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
```

Native 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. The repository is responsible for persistence, so it's a better place to store these definitions.

JPQL Query:

The JPQL (Java Persistence Query Language) is an object-oriented query language which is used to perform database operations on persistent entities. Instead of database table, JPQL uses entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks. JPQL is developed based on SQL syntax, but it won't affect the database directly. JPQL can retrieve information or data using SELECT clause, can do bulk updates using UPDATE clause and DELETE clause.

By default, the query definition uses JPQL in Spring JPA. Let's look at a simple repository method that returns Users entities based on city value from the database:

```
// JPQL Query in Repository Layer
@Query(value = "Select u from Users u where city=?1 ")
List<Users> getUsersByCityName(String city);
```

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:

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 users.

```
@Query(value = "select * from flipkart_users", nativeQuery = true)
List<Users> getUsers();
```

For passing values to 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:

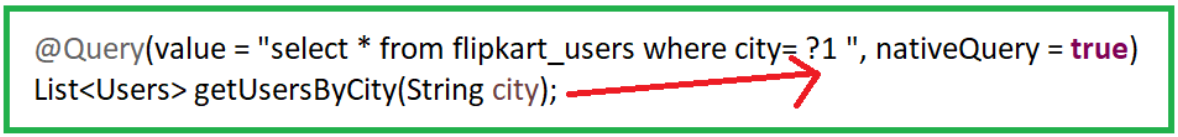
If SQL query contains positional parameters and we have to pass values to those, we should use Indexed Params i.e. index count of parameters. For indexed parameters, Spring JPA Data will pass method parameter values to the query in the same order they appear in the method declaration.

Example:

```
@Query(value = "select * from flipkart_users ", nativeQuery = true)
List<Users> getUsersByCity();
```

Now below method declaration in repository will return List of Entity Objects with city parameter.

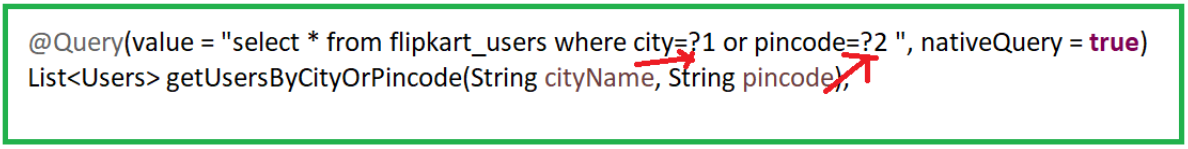
```
@Query(value = "select * from flipkart_users where city= ?1 ", nativeQuery = true)
List<Users> getUsersByCity(String city);
```



Another example with more indexed parameters:

Get Data of users from either city or pincode matches.

```
@Query(value = "select * from flipkart_users where city=?1 or pincode=?2 ", nativeQuery = true)
List<Users> getUsersByCityOrPincode(String cityName, String pincode);
```



By using Named Query Parameters:

We can also pass 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);
```

IN JPQL also we can use index and named Query parameters.

Spring Boot JPA - Named Queries:

We can use **@NamedQuery** annotation to specify a named query with in an entity class and then declare that with method in repository. We will add custom methods in Repository. Now let's add another method using **@NamedQuery** and test it.

@NamedQuery is nothing but providing name to a query and will be defined in Entity class. We will use same name in repository method level as part of **@Query** annotation with attribute **name.**

Entity Class: Named Query will be defined in entity class at class level. We can define multiple Named Queries as well.

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.NamedQuery;
import jakarta.persistence.Table;

@Entity
@Table(name = "flipkart_users")
@NamedQuery(name="mobileQuery", query= "Select u from Users u where mobileNum =:mobile ")
public class Users {

	@Column(name = "first_name")
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Column(name = "mobile_number")
	private String mobileNum;

	@Id
	private String email;

	private String password;
	private String city;
	private int pincode;

	//Setters and Getters

}
```

Now Define method in side Repository layer and configure named Query of mobileQuery.

```
// Named  Query
@Query(name = "mobileQuery")
List<Users> getUsersByMobile(@Param("mobile") String mobileNumber);
```

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:

1. **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.

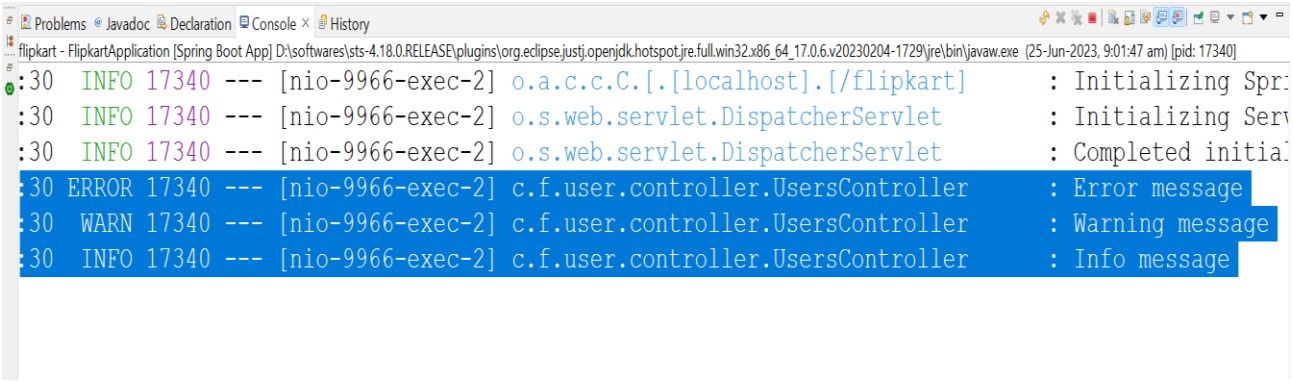
Ex : Logger = LoggerFactory.*getLogger*(UsersController.class);

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.



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 UsersService {

		Logger logger = LoggerFactory.getLogger(UsersService.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

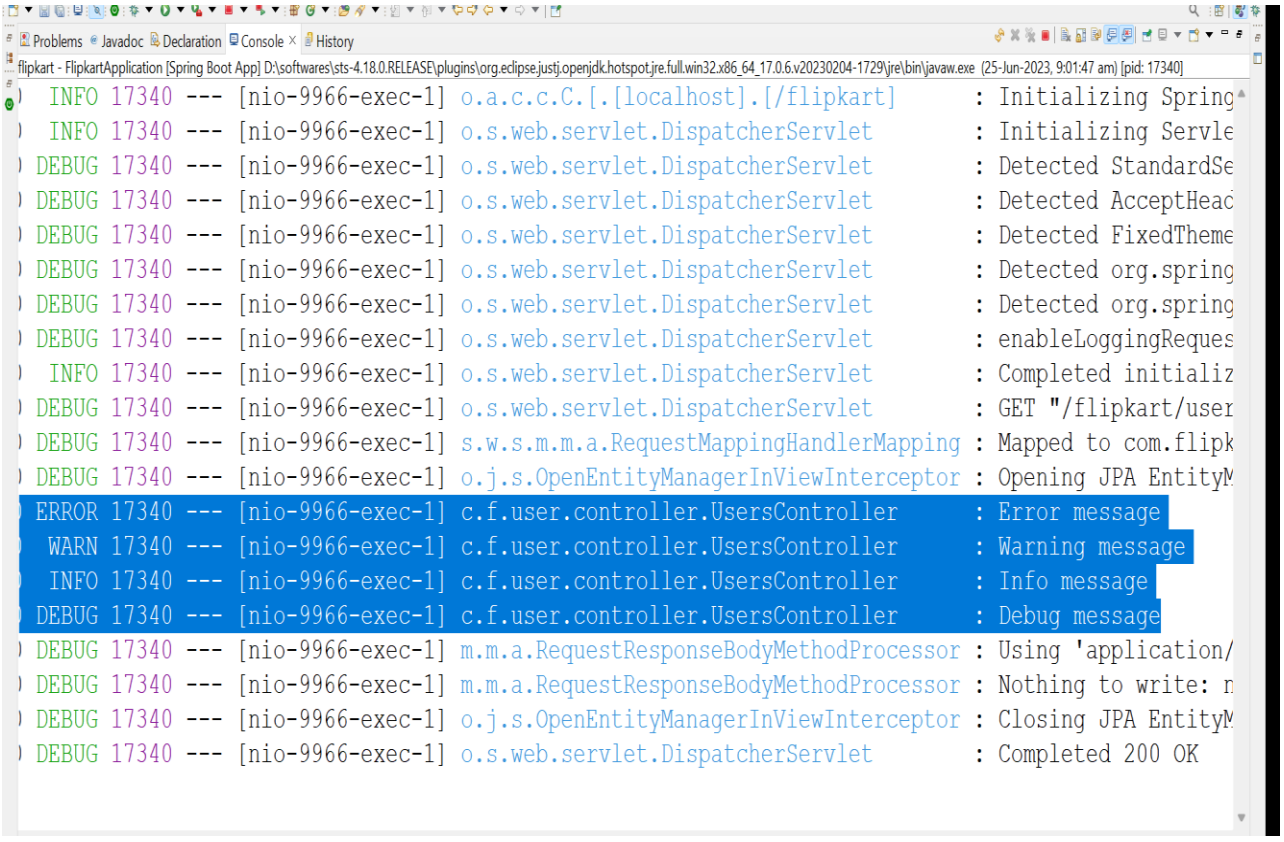
of the application can be configured with different log levels, which is especially useful for code isolation and debugging. To specify a log level for all classes that don't have their own log level settings, the root logger can be set using with property **logging.level.root**.

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
```



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.

REST API / Services:

REST API:

REST stands for **Representational State Transfer**. 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 via HTTP Protocol. JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text. 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

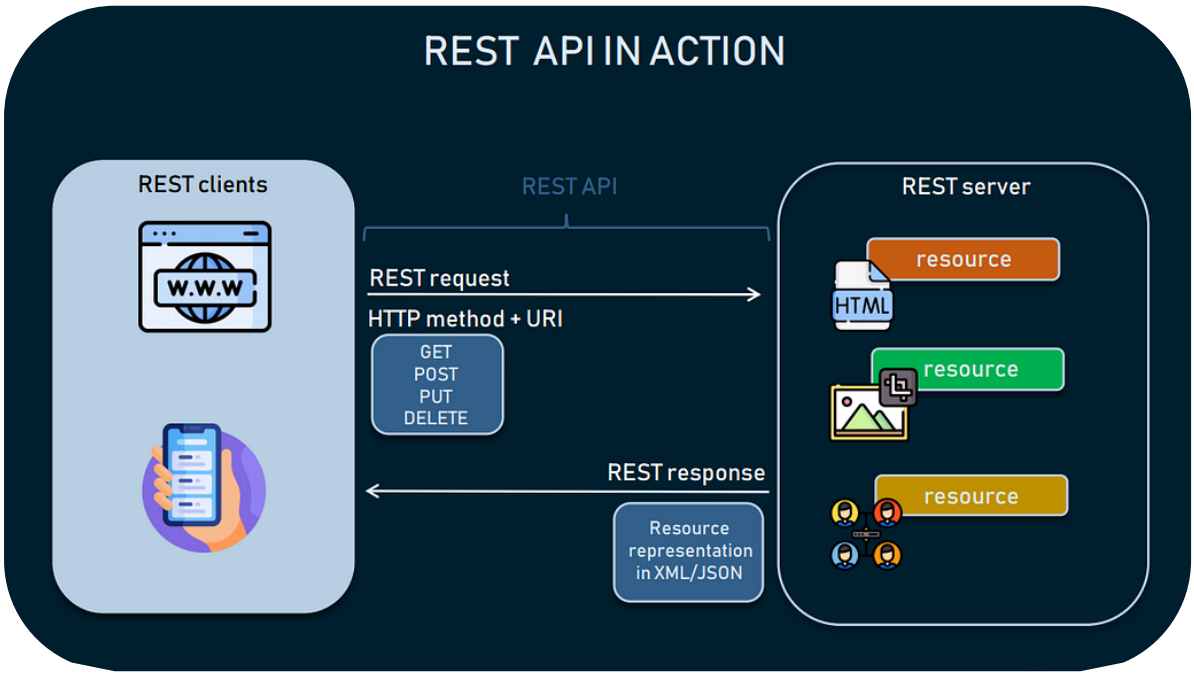
used on the server side. You can write both client and server applications in various programming languages without affecting the API design. We can also change the technology on both sides without affecting the communication.

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.

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

NAME	DESCRIPTION	SQL EQUIVALENT
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

So far we are created many POST and GET method endpoints aligned to REST guidelines in previous topics or classes. We will see few more in upcoming topics as well.

REST Service with HTTP method PUT:

PUT is used to send data to a server to update/create a resource. The difference between POST and PUT is that **PUT requests are idempotent**. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly have side effects of creating the same resource multiple times.

- Generally, when we want to update Details of existing resource, then we will define an endpoint with HTTP method PUT in controller class.

- In some other cases also we will go for PUT method endpoint creation, like when we have to update data when Resource found in DB, if resource not found then we have to create Resource i.e. inserting data in same API call.

```
If(Resource Available){
    Update Data/Query;
}else{
     Insert Data/Query;
}
```

We can perform update operation in Database by selecting either Native/SQL Queries or JPA predefined functionalities.

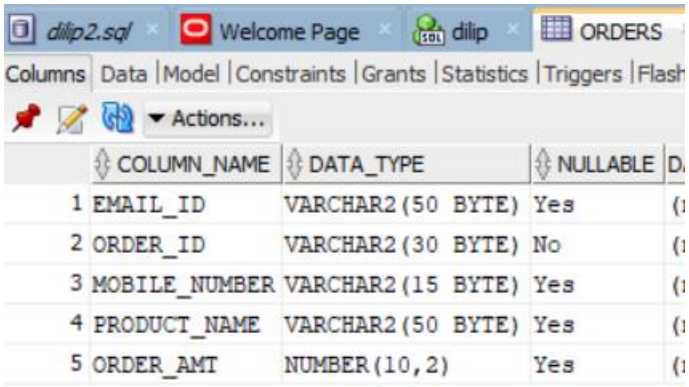
Requirement: Update Order Details Based on Order ID.

Use Case 1:

Create REST API Service, for Adding Order Details
Create REST API Service, for Updating Product Name by passing Order ID.

Now Let's Identify our DB Table and Create Entity class for same.
Table SQL Script :

Create table orders(email_id varchar2(50), order_id varchar2(30) primary key, mobile_number varchar2(15), product_name varchar2(50), order_amt number(10,2));


dilip2.sql | Welcome Page | dilip | ORDERS

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flash

Actions...

	COLUMN_NAME	DATA_TYPE	NULLABLE	D
1	EMAIL_ID	VARCHAR2(50 BYTE)	Yes	(
2	ORDER_ID	VARCHAR2(30 BYTE)	No	(
3	MOBILE_NUMBER	VARCHAR2(15 BYTE)	Yes	(
4	PRODUCT_NAME	VARCHAR2(50 BYTE)	Yes	(
5	ORDER_AMT	NUMBER(10,2)	Yes	(

- Now create an Entity class aligned to Above DB table: **OrderDetails.java**

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "orders")
public class OrderDetails {

	@Id
	@Column(name = "order_id")
	private String orderID;

	@Column(name = "email_id")
	private String emailId;

	@Column(name = "product_name")
	private String productName;
```

```
	@Column(name = "mobile_number")
	private String mobileNumber;

	@Column(name = "order_amt")
	private double orderAmount;

	//setters and getters methods

}
```

- **Now create an end point method in side controller class:** OrdersController.java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.flipkart.orders.request.OrderDetailsRequest;
import com.flipkart.orders.request.OrderDetailsUpdateRequest;
import com.flipkart.orders.service.OrderService;



@RestController
@RequestMapping("/order")
public class OrdersController {

	// Interface Name
	@Autowired
	OrderService orderService;

	Logger log = LoggerFactory.getLogger(OrdersController.class);

	// Adding Orders
	@PostMapping("/add")
	public String addOrdersOfUSer(@RequestBody OrderDetailsRequest request) {
		log.info("Order details Recived  " + request);
		return  orderService.addOrdersOfUSer(request);
	}

	//Updating Orders : Resource Id is Order ID as path Variable.
	// Updated Information Taken as Request Body class
	@PutMapping("/update/product/{orderId}")
	public String updateOrdersOfUSer(@PathVariable String orderId, @RequestBody
OrderDetailsUpdateRequest request) {
```

```
            log.info("Order details Received for Update  " + request);
            return orderService.updateOrdersOfUSer(orderId, request);
        }
}
```

In above controller, created 2 endpoints for adding and updating details.

For Adding Order, create POST API call.

Request Body Class: **OrderDetailsRequest.java**

```
public class OrderDetailsRequest {
        private String orderID;
        private String emailId;
        private String productName;
        private String mobileNumber;
        private double orderAmount;

        // Setters and Getters
}
```

For Updating Order Details, Created PUT method call.

Request Body class: **OrderDetailsUpdateRequest.java**

As per Requirement Updating only Product Name via this REST Call.

```
public class OrderDetailsUpdateRequest {
        private String productName;
        //Setter and Getters
}
```

Now Create Service layer. **We are creating Service layer interface and Implemented classes.**

Interface : OrderService.java

```
import com.flipkart.orders.request.OrderDetailsRequest;
import com.flipkart.orders.request.OrderDetailsUpdateRequest;

// Service Layer:  Interface
public interface OrderService {
        String addOrdersOfUSer(OrderDetailsRequest request);
        String updateOrdersOfUSer(String orderId, OrderDetailsUpdateRequest request);
}
```

Service Layer Implementation Class: OrdersServiceImpl.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.flipkart.entity.OrderDetails;
```

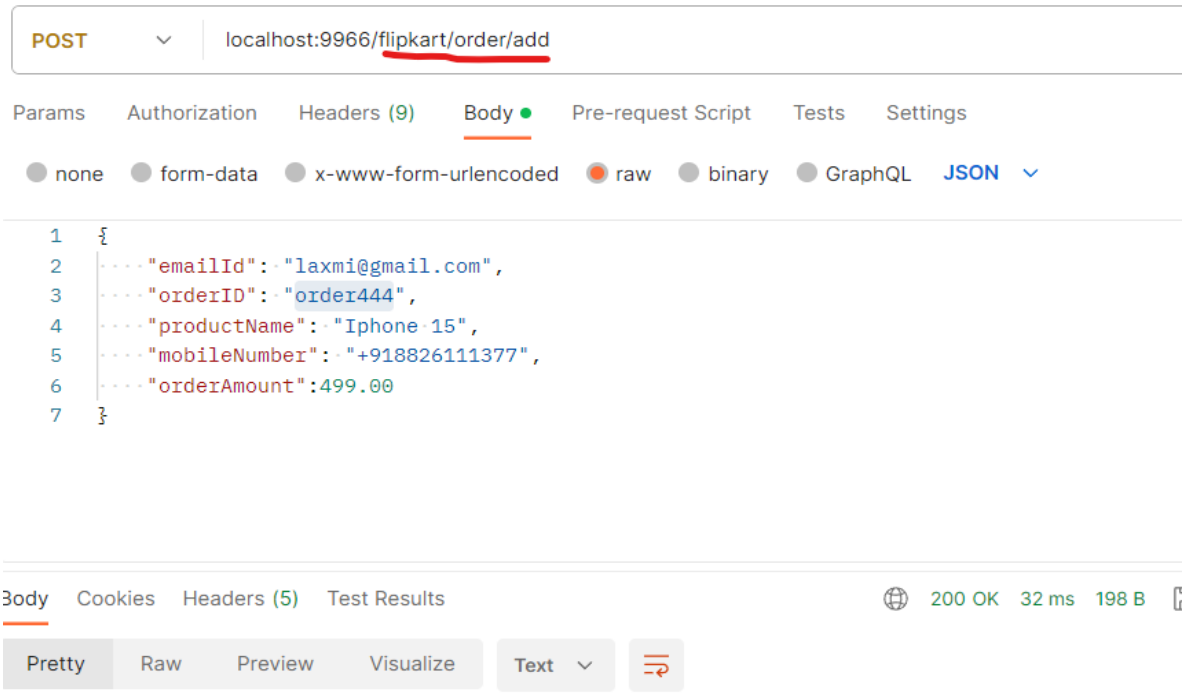
```
import com.flipkart.orders.request.OrderDetailsRequest;
import com.flipkart.orders.request.OrderDetailsUpdateRequest;
import com.flipkart.repository.OrdersRepostiory;

@Service
@Transactional
public class OrdersServiceImpl implements OrderService {

	@Autowired
	OrdersRepostiory repository;
	@Override
	public String addOrdersOfUSer(OrderDetailsRequest request) {
		// Onverting to Entity Object
		OrderDetails order = new OrderDetails();
		order.setEmailId(request.getEmailId());
		order.setMobileNumber(request.getMobileNumber());
		order.setOrderID(request.getOrderID());
		order.setProductName(request.getProductName());
		order.setOrderAmount(request.getOrderAmount());
		repository.save(order);
		return "Order Drtails Addedd Successfuly. ";
	}

	@Override
	public String updateOrdersOfUSer(String orderId, OrderDetailsUpdateRequest request) {
		repository.updateProductName(request.getProductName(), orderId);
		return "Order Drtails Updated Successfuly. ";
	}

}
```

Now Create Repository Layer with User Derived Method for Update Native Query.
Repository Class : OrdersRepostiory.java

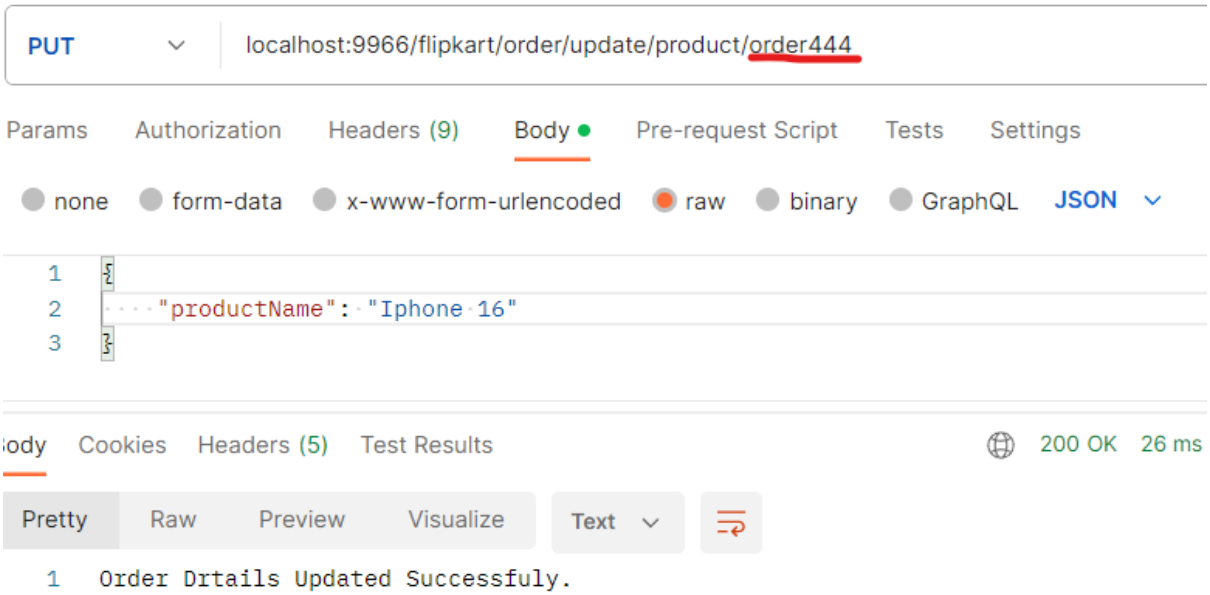
```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;
import com.flipkart.entity.OrderDetails;

@Repository
public interface OrdersRepostiory extends JpaRepository<OrderDetails, String>{

	// Named Query Parameters used along with Native SQL Query
	@Modifying
	@Query(value = "update orders set product_name=:pname where order_id=:orderID",
	nativeQuery = true)
	public void updateProductName(@Param("pname") String pname, @Param("orderID")
	String orderID);
}
```

Now Start our application and Perform Execution from POST man.

Testing: Add Order : Create JOSN payload as per Request Body Class, as shown in below snap.



Testing: Update Order : Create JOSN payload as per Request Body Class and Path variable value i.e. Order Id as part of URL as shown in below snap.



Check in data base against order id record whether data updated or not.

In Update Query implementation we used 2 new annotations **@Transactional** and **@Modifying** at service and repository layers.

@Transactional :

In Spring Boot, `@Transactional` is an annotation that is used to mark a method or class as transactional. Transactions ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties of database operations, where a group of operations are treated as a single unit of work.

When `@Transactional` is applied to a method or class, it enables the management of transactions for that particular method or all the methods within the class.

Here's how you can use `@Transactional` in Spring Boot:

1. Method-level Transactional Annotation:

```
@Transactional
public void someTransactionalMethod() {
        // Method implementation
}
```

Applying `@Transactional` at the method level ensures that the entire method is executed within a transaction. If an exception occurs, the transaction is rolled back, and any changes made within the method are undone.

2. Class-level Transactional Annotation:

```
@Transactional
public class SomeService {
        // Class methods
}
```

When `@Transactional` is applied at the class level, it applies transactional behaviour to all public methods of that class.

It's important to note that `@Transactional` works when used with a transaction manager. Spring Boot provides default transaction management through its integrated support for the Java Persistence API (JPA) or you can configure other transaction managers, such as Hibernate or JDBC, based on your database technology.

By using `@Transactional` appropriately, you can ensure the consistency and integrity of your data by managing database transactions effectively within your Spring Boot application.

@Modifying:

In Spring Boot, `@Modifying` is an annotation that works in addition with `@Query` to indicate that a query method modifies the data in the database. It is commonly used when executing update, delete, or insert operations using Spring Data JPA or Spring Data JDBC.

Here's how you can use `@Modifying` in Spring Boot:

1. **Define a repository interface:**

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;

public interface UserRepository extends JpaRepository<User, Long> {

   @Modifying
   @Query("UPDATE User u SET u.name = ?1 WHERE u.id = ?2")
   void updateUserName(String newName, Long userId);
}
```

In this example, the `UserRepository` interface extends `JpaRepository` and defines a custom query method `updateUserName`. The `@Modifying` annotation indicates that this method modifies the database. The `@Query` annotation defines the custom update query.

The `@Modifying` annotation is crucial because it informs Spring Data JPA or Spring Data JDBC that the query is intended to modify the database state. Without this annotation, the query will be treated as a read-only operation, and attempting to execute an update, delete, or insert operation will result in an exception.

Update Query with save() method:

We can update data on database when we are updating with primary key column in condition and trying to modify other columns data or values.

The Approach is **Find and Load then update**:

For above same update operation, we are choosing this approach.

In Controller we are writing logic as shown in below i.e. we are not deriving any method in repository. JPA interfaces are not providing any method update operation directly.

OrdersServiceImpl.java

```
@Override
public String updateOrdersOfUSer(String orderId, OrderDetailsUpdateRequest request) {
        // Find and load and then update

        // Find and Load Data
        Optional<OrderDetails> order = repository.findById(orderId); // Select Query : I
```

```
        if (order.isPresent()) {

                // Updating data of entity Object
                // extract entity object from optional
                OrderDetails orderDetails = order.get();

                // Update entity object data what should be modified
                orderDetails.setProductName(request.getProductName());

                // call save method
                repository.save(orderDetails);
        } else {
                return "Order ID Not found. Please Check order ID ";
        }
        return "Order Details Updated Successfully. ";
}
```

Now we can test our end point, and notice SQL queries which are generated.

Hibernate: select
o1_0.order_id,o1_0.email_id,o1_0.mobile_number,o1_0.order_amt,o1_0.product_name from orders o1_0 where o1_0.order_id=?

Hibernate: update orders set email_id=?,mobile_number=?,order_amt=?,product_name=? where order_id=?

HTTP DELETE Method and REST Services:

For Deleting Data/Resources we have to define Delete Mapping API calls i.e. HTPP method is DELETE we should choose for REST API call. In Spring Boot, the @DeleteMapping annotation is used to handle HTTP DELETE requests.

For example I want to delete on order by passing order ID.

In Side Controller class:

```
@DeleteMapping("/delete/{orderId}")
public String deleteOrder(@PathVariable String orderId) {
        return orderService.deleteOrder(orderId);
}
```

In Service class:

```
@Override
public String deleteOrder(String orderId) {
        repository.deleteById(orderId);  // Using Predefined JPA method
        return "Order Drtails Deleted Successfuly.";
}
```

Now test our service, So specific record will be deleted from database **Orders** table.

Similarly we have few other JPA methods to perform delete operations on database.
For example : Deletes All from table

```
@Override
public String deleteAllOrders() {
        repository.deleteAll();
        return "All Order Drtails Deleted Successfuly.";
}
```

We can define native SQL and JPQL queries for delete operations in side repository layer, however we did for update and insert and select operations in previous examples.

Swagger UI With SpringBoot Applications

"Swagger is a set of rules, specifications and tools that help us document our APIs."

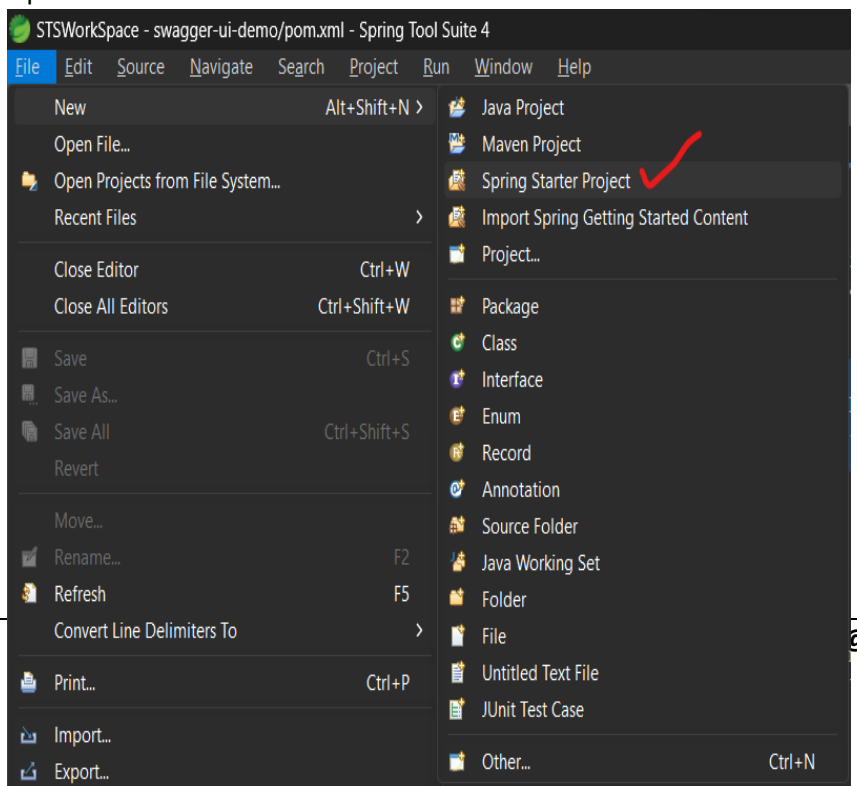
Swagger UI allows anyone — be it your development team or your end consumers — to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from your OpenAPI (formerly known as Swagger) Specification, with the visual documentation 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 browseable 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.

"By using Swagger UI to expose our API's documentation, we can save significant time."

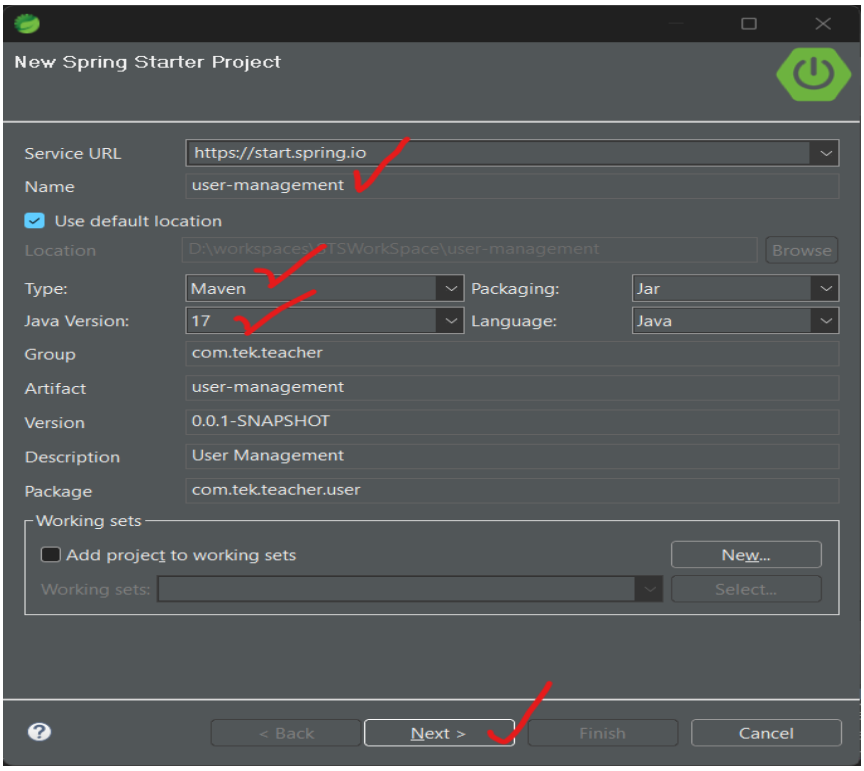
Create Spring Boot Web Application.

1. Open STS

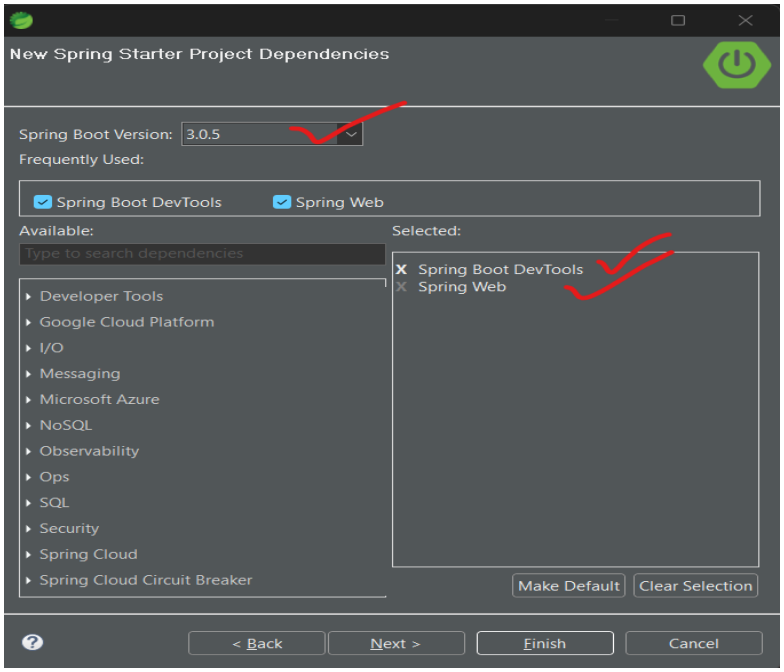


2. File-> New > Spring Starter Project

3. Fill All Project details as shown below and click on Next.

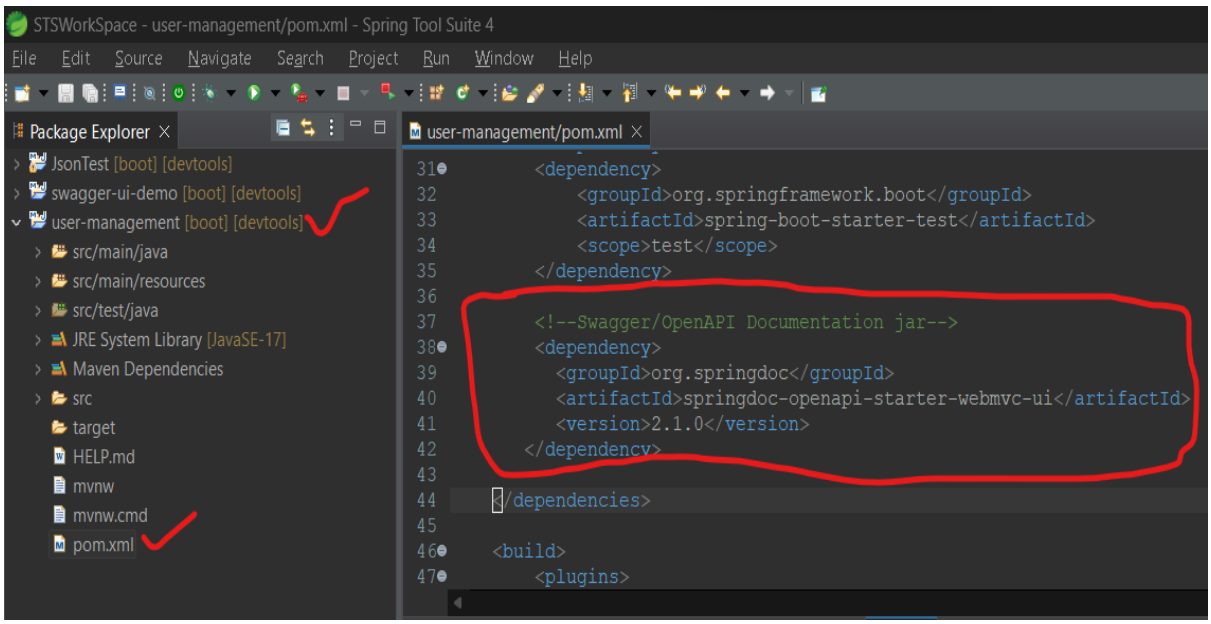


4. In Next Page, Add Spring Boot Modules/Starters as shown below and click on finish.
   NOTE: Spring Web is mandatory



5. After Project Creation, Open **pox.xml** file and add below dependency in side dependencies section and save.

```
<!--Swagger/OpenAPI Documentation jar-->
    <dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>2.1.0</version>
    </dependency>
```

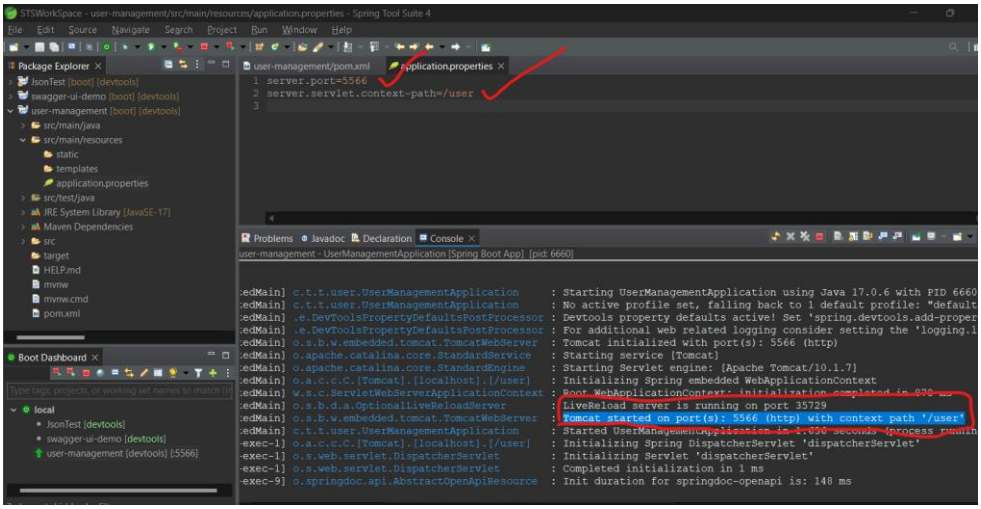


6. Now open **application.properties** file and add below two properties and save. With these properties application started on port : 5566 with context path '/user'.

   server.port=5566

   server.servlet.context-path=/user

7. Now start your spring Boot application



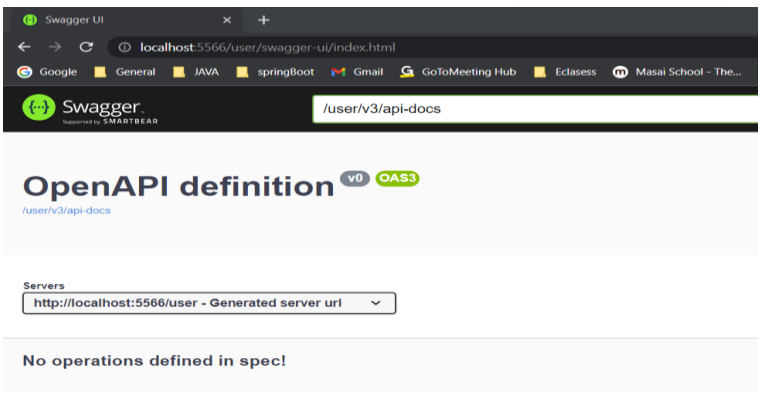
8. Enter URL in Browser for OpenAPI Swagger Documentation of Web services. Then you can Swagger UI page with empty Services List. Because Our application not contained any web services.

   NOTE :

   The Swagger UI page will then 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 or IP, **port:** The server port, **context-path:** The context path of the application

   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

```
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;
	}

}
```

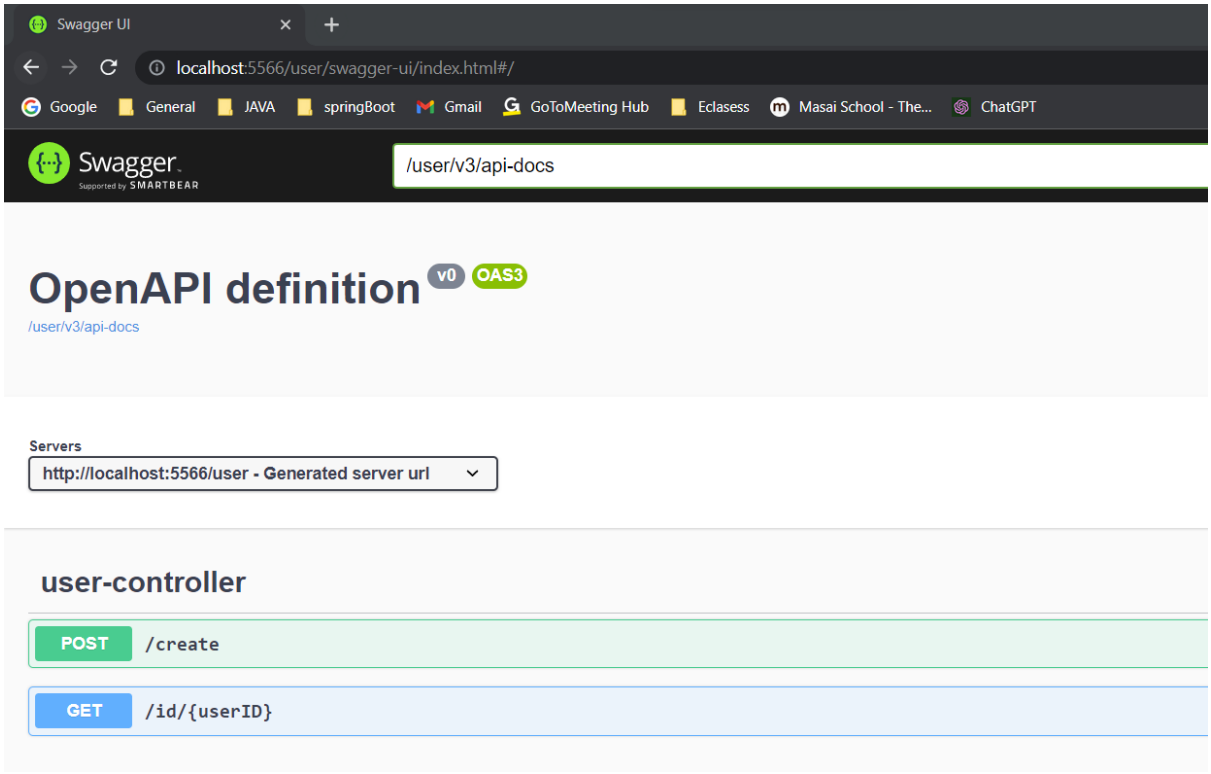
CreateUserResponse.java

```
package com.tek.teacher.user.controller;
```

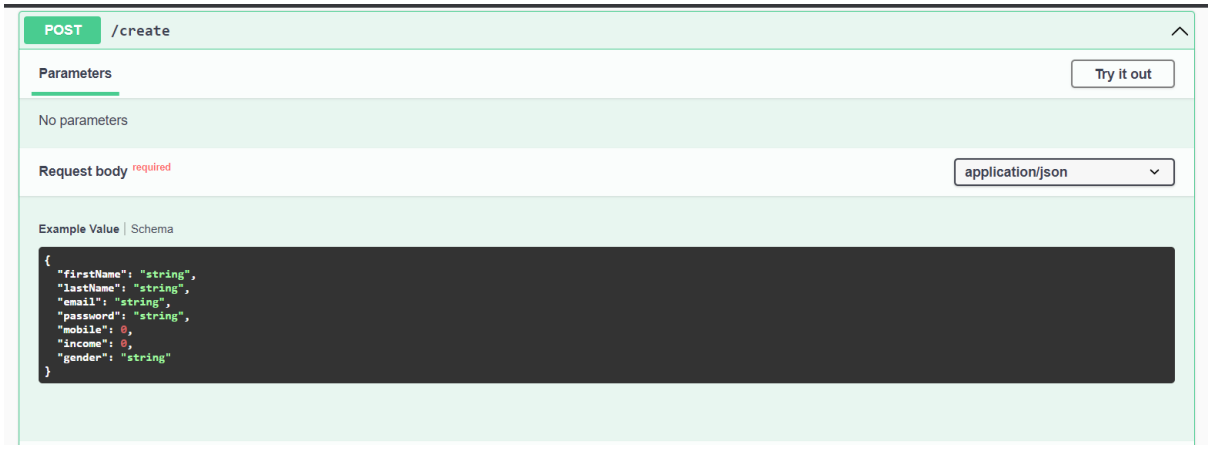
```java
public class CreateUserResponse {

	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;
	}

}
```

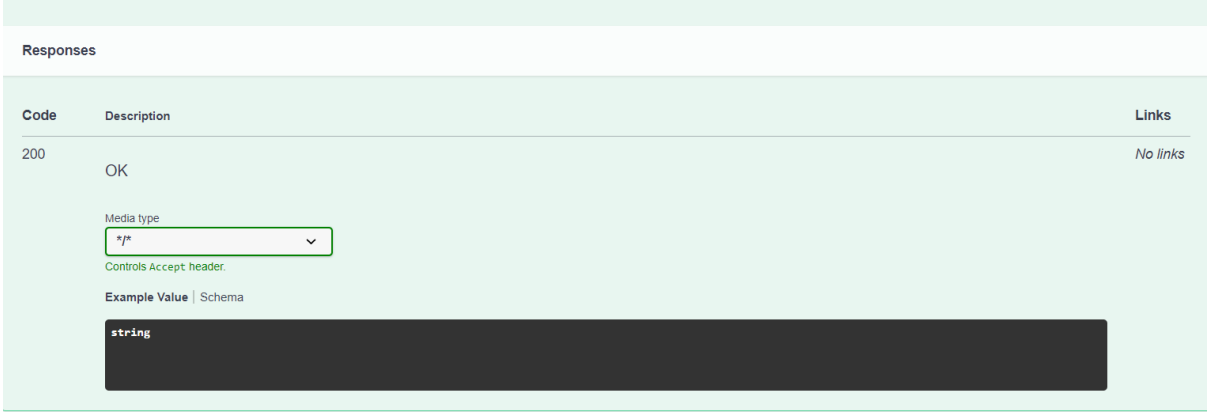
Now Start Your Spring Boot App. After application started, Now please enter swagger URL in browser. You can see all endpoints/services API request and response format Data.

http://localhost:5566/user/swagger-ui/index.html

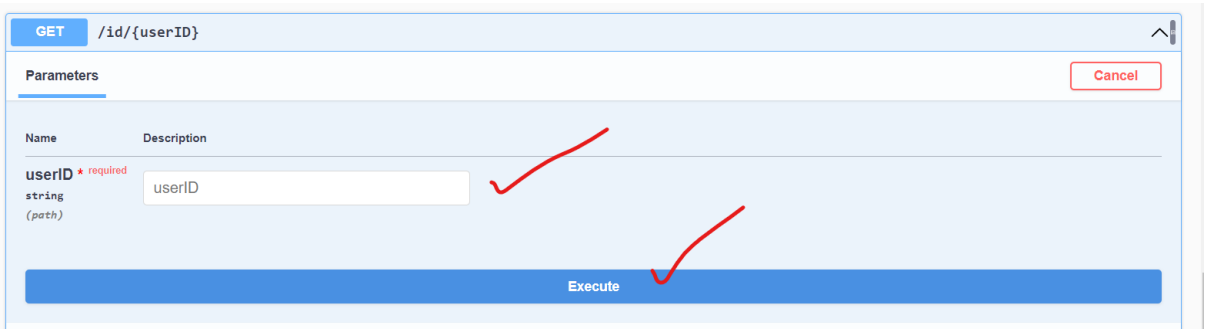


➢ **You can expand above both endpoints and look for payload details.**

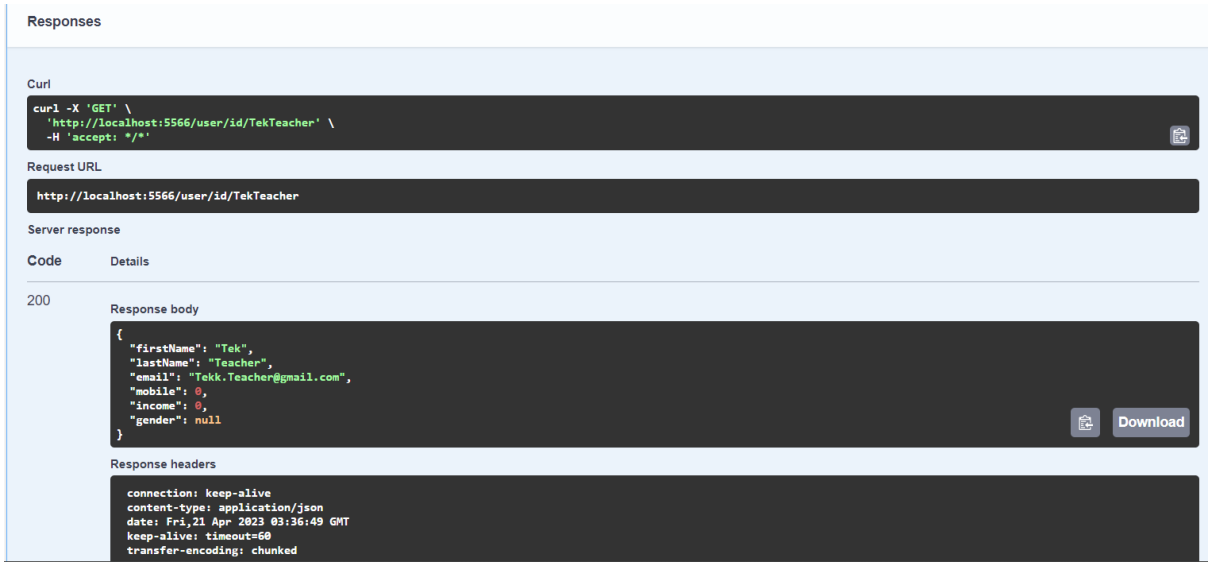




We can Test API calls from Swagger UI, Please click on **Try it Out** button then it will you pass values to parameters/properties. Below I am passing **userId** value in GET API service and then click on Execute.



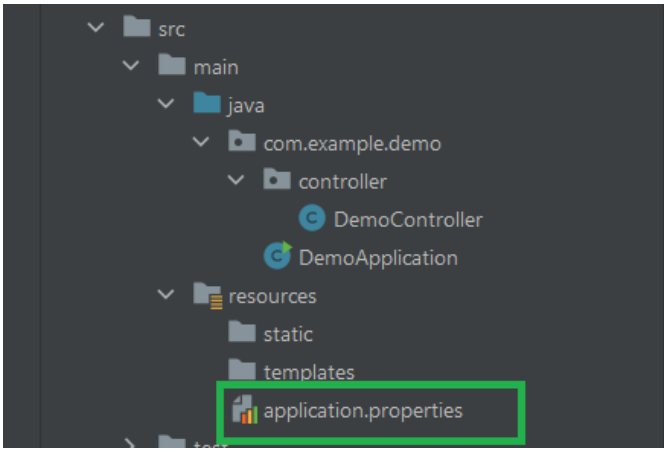
After clicking on Execute You will get Response in response Section.



This is how we can integrate and use swagger UI with our applications to share all webservices 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.

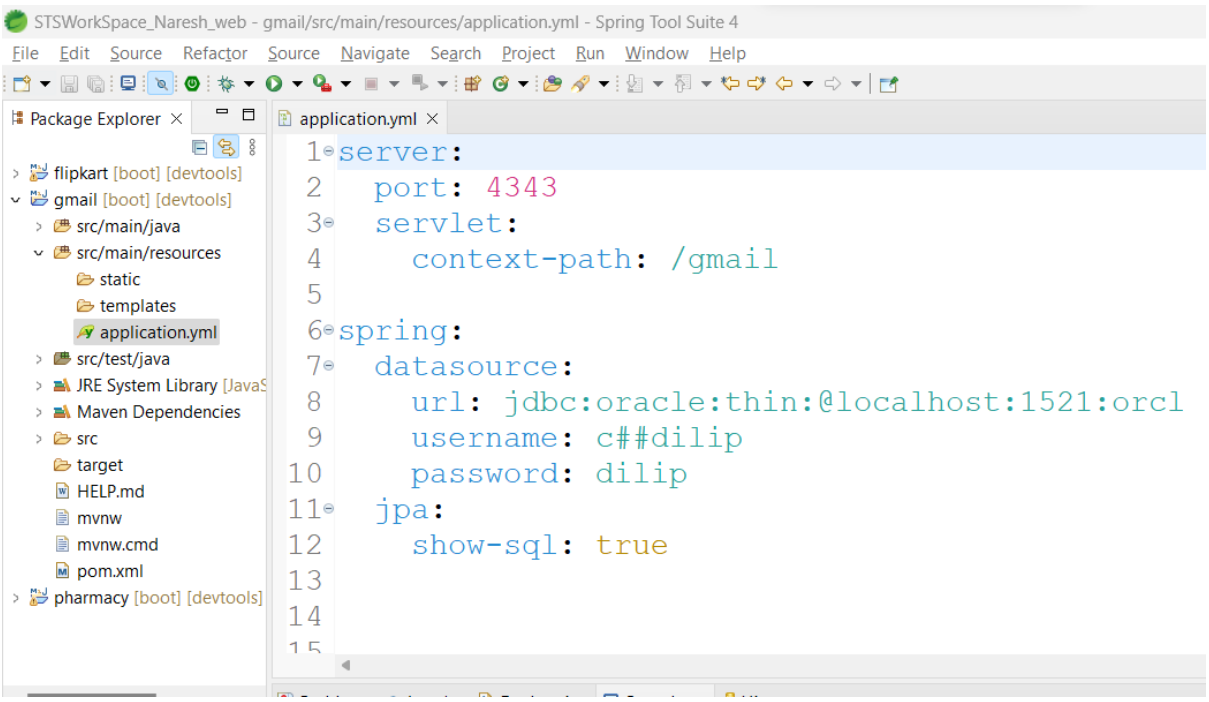
- Comments can be identified with a pound or hash symbol (#). YAML does not support multi-line comment, each line needs to be suffixed with the pound character.
- YAML files use a .yml or .yaml extension, and follow specific syntax rules.

Now let's see some examples for better understanding :

If it is **application.properties** file **:**

```
server.port=9966
server.servlet.context-path=/flipkart
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password,=dilip
spring.jpa.show-sql=true
```

application.yml:



Now can start our application as usual and test.

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.

1xx: Informational – Communicates transfer protocol-level information.
2xx: Success – Indicates that the client's request was accepted successfully.
3xx: Redirection **–** Indicates that the client 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 Error

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 Error

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 Specific HTTP Status Codes:

Generally we will use below scenarios and respective status code in REST API services.

POST - Create Resource : **201 Created** : Successfully Request Completed.

PUT - Update Resource : **200 Ok** : Successfully Updated Data

If not i.e. Resource Not Found Data
404 Not Found : Successfully Processed but Data Not available

GET - Read Resource : **200 Ok** : Successfully Retrieved Data

If not i.e. Resource Not Found Data
404 Not Found : Successfully Processed but Data Not available

DELETE - Deleting Resource : **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.

For example In RestTemplate, this class is returned by getForEntity() and exchange():

```
ResponseEntity<String> entity = template.getForEntity("https://example.com", String.class);
```

This can also be used in Spring MVC as the return value from an Controller method:

```
@RequestMapping("/handle")
public ResponseEntity<String> handle() {
                // Logic
  return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);
}
```

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 Netbanking 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 OrderUpdateRequest 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);
	}
}
```

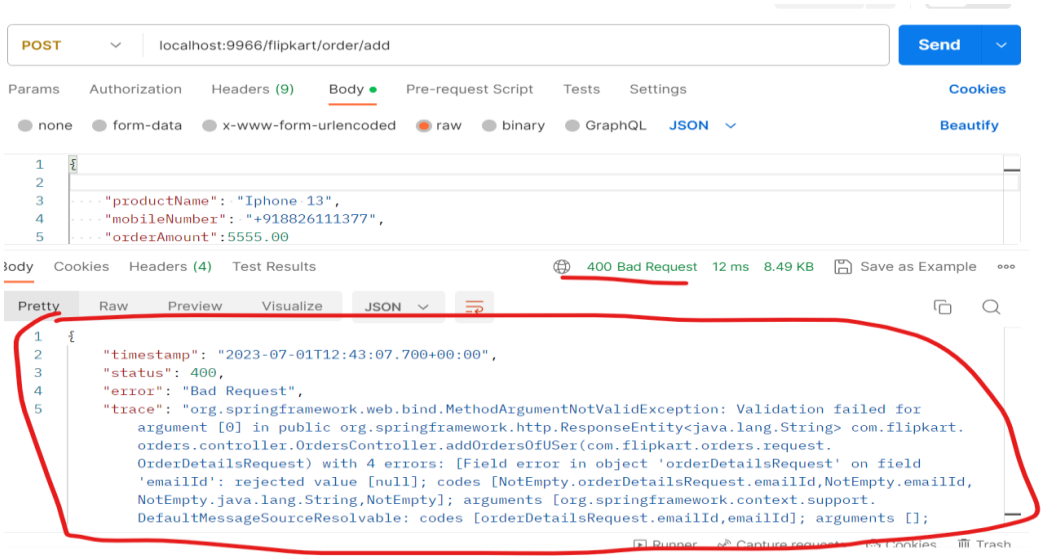
This is how can write any Response Status code in REST API Service implementation. Please refer RET API Guidelines for more information at what time which HTTP status code should be returned to client.

Exception Handling in Controllers:

What I have to do with Errors or Exceptions ?

Spring brings support for a global @ExceptionHandler with the @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.



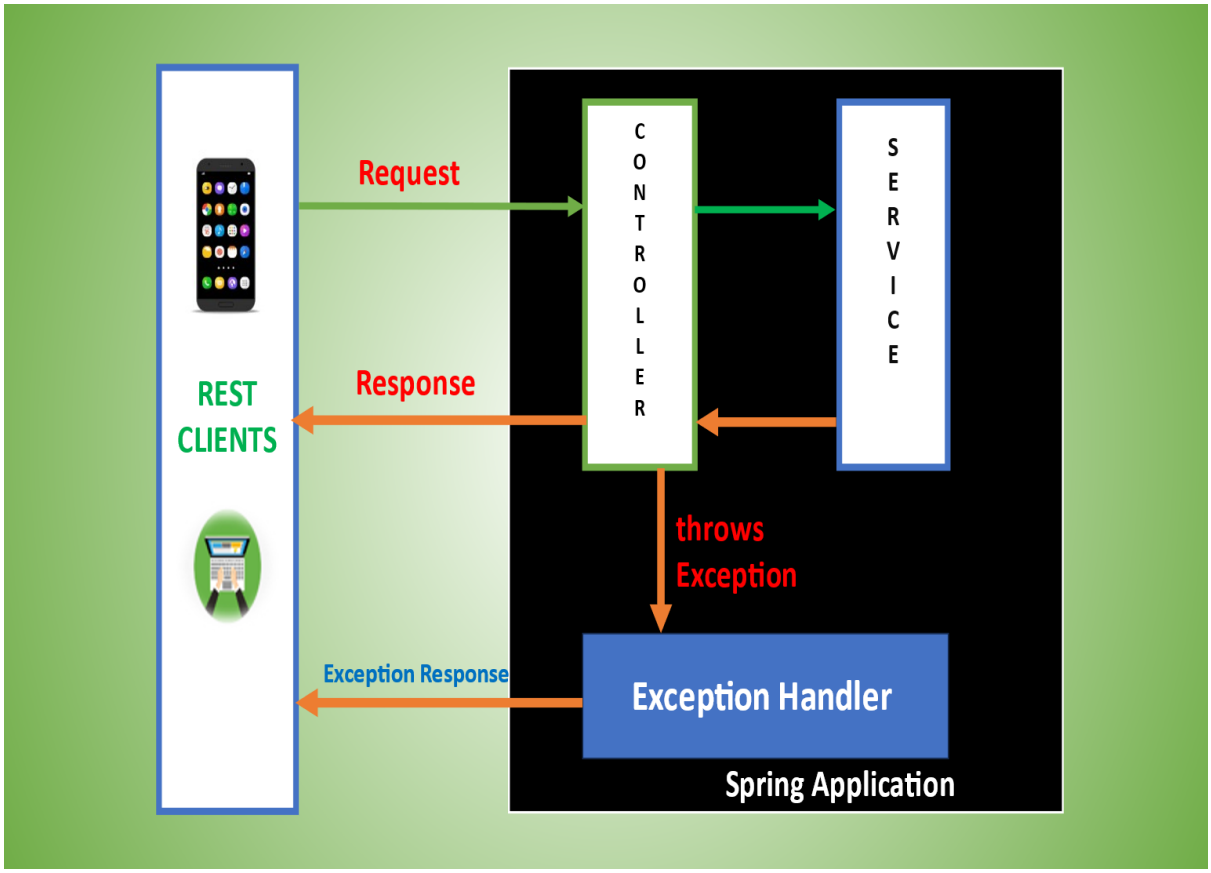
Spring provided other ways as well to handle exceptions but controller advice and Exception handler will provide better way exception handling.

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 ExceptionHandlers 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 ResposeEntity response.

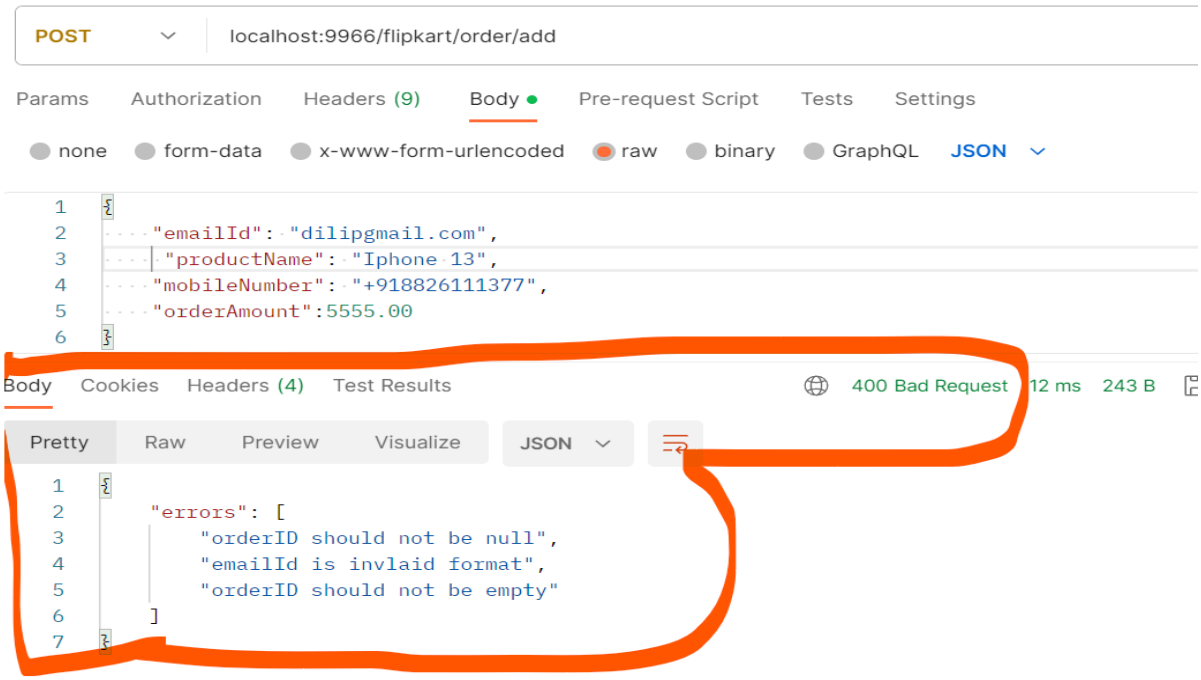
One thing to keep in our mind here is to match the exceptions declared with @ExceptionHandler to the exception used as the argument of the method.

Examples of Handler class : Controller Advice class 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;

// Exception handler class

@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());

		Map<String, List<String>> errors = new HashMap<>();
		errors.put("errors", messages);

		return new ResponseEntity<>(errors, 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(ArithmeticException.class)
	public ResponseEntity<?> handleArithmeticException(ArithmeticException 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.

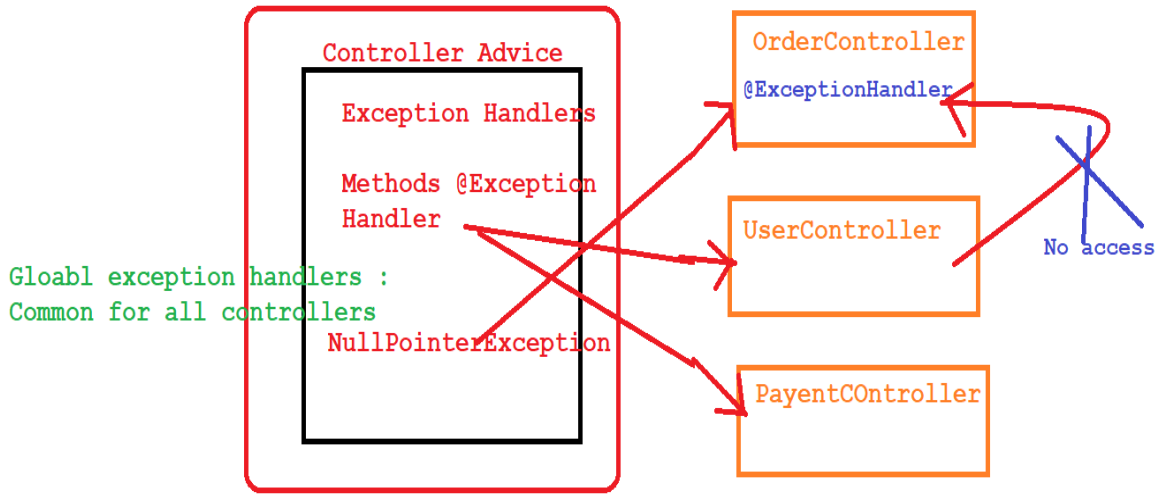


How it is working?

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 an 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.



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 other application REST API services from our application logic.

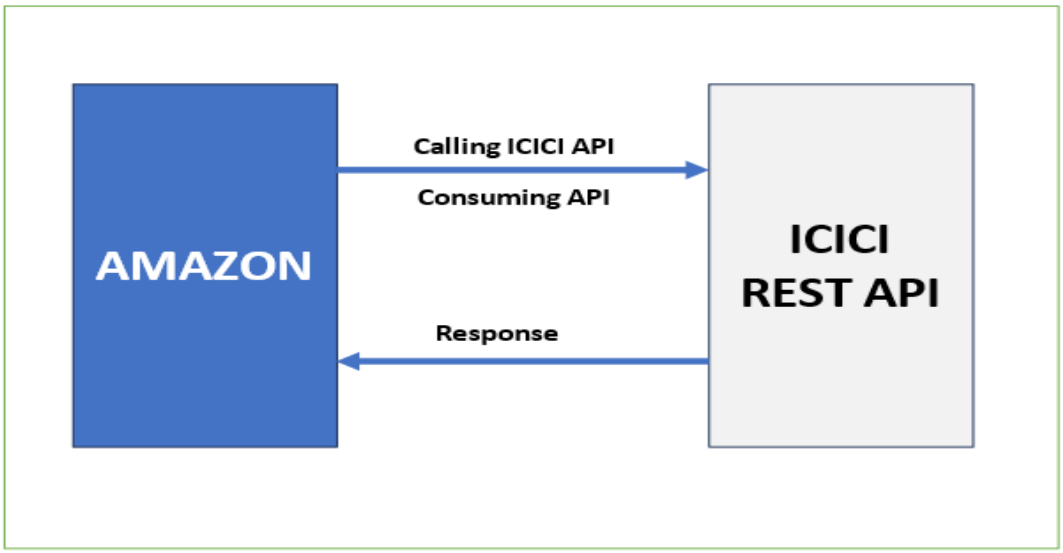
For Example,

ICICI bank will produce API services to enable banking functionalities. Now Amazon application integrated with ICICI REST services for performing Payment Options.

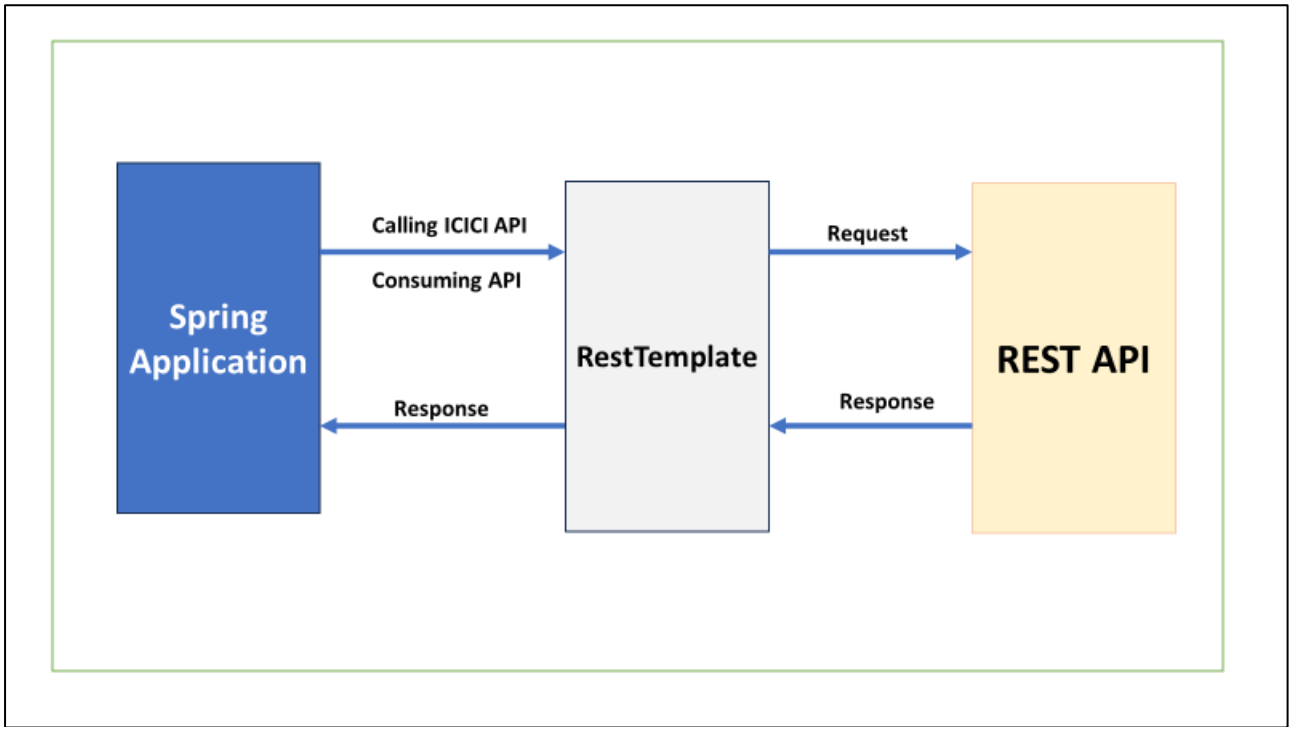
In This case:

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.



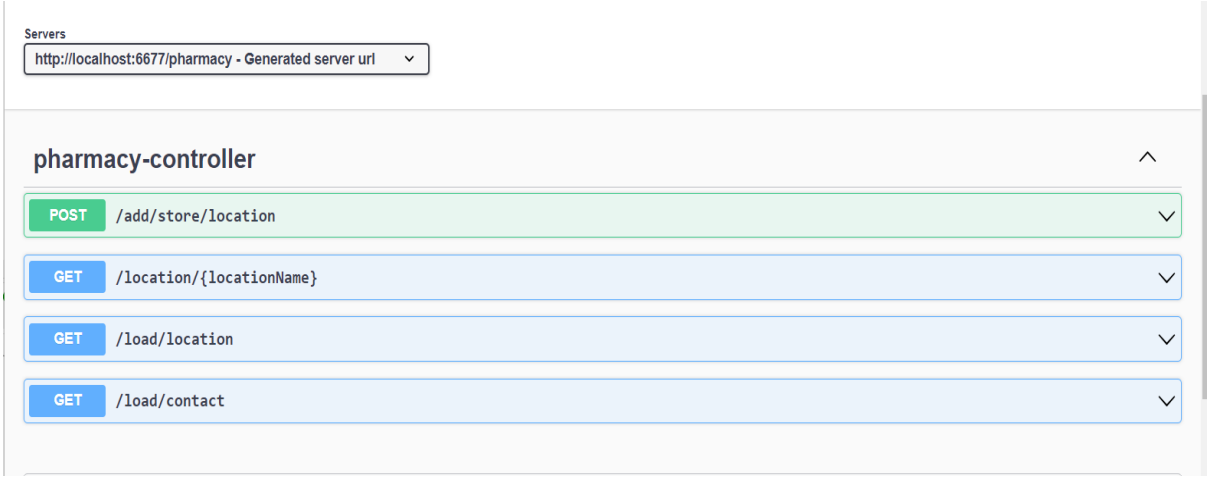
RestTemplate is used to create applications that consume RESTful Web Services. You can use the **exchange()** or specific http methods to consume the web services for all HTTP methods.

Now we are trying to call Pharmacy Application API from our Spring Boot Application Flipkart i.e. **Flipkart consuming Pharmacy Application REST API.**

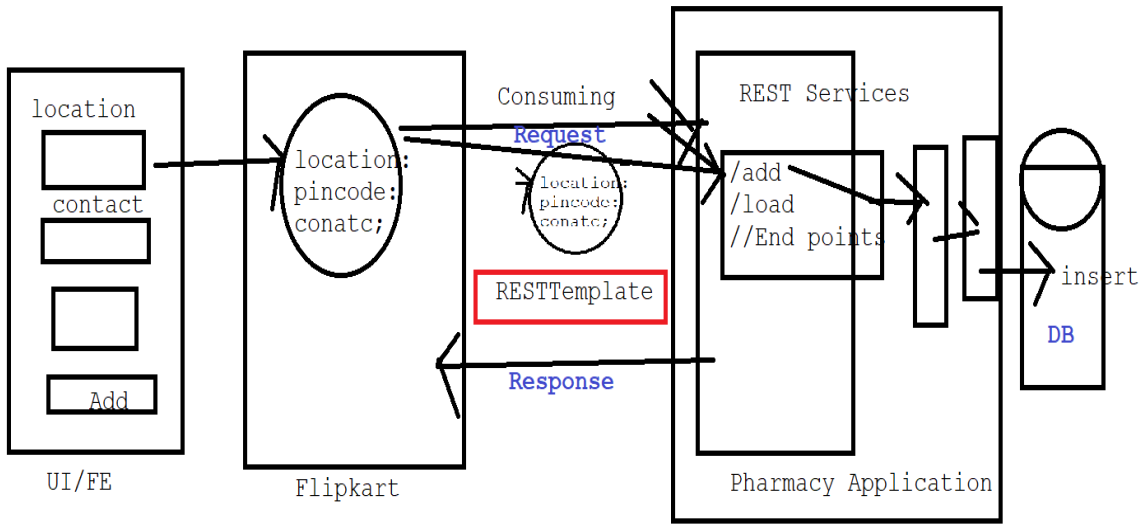
Now I am giving only API details of Pharmacy Application as swagger documentation. Because in Realtime Projects, swagger documentation or Postman collection data will be shared to Developers team, but not source code. So we will try to consume by seeing Swagger API documentation of tother application. 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.

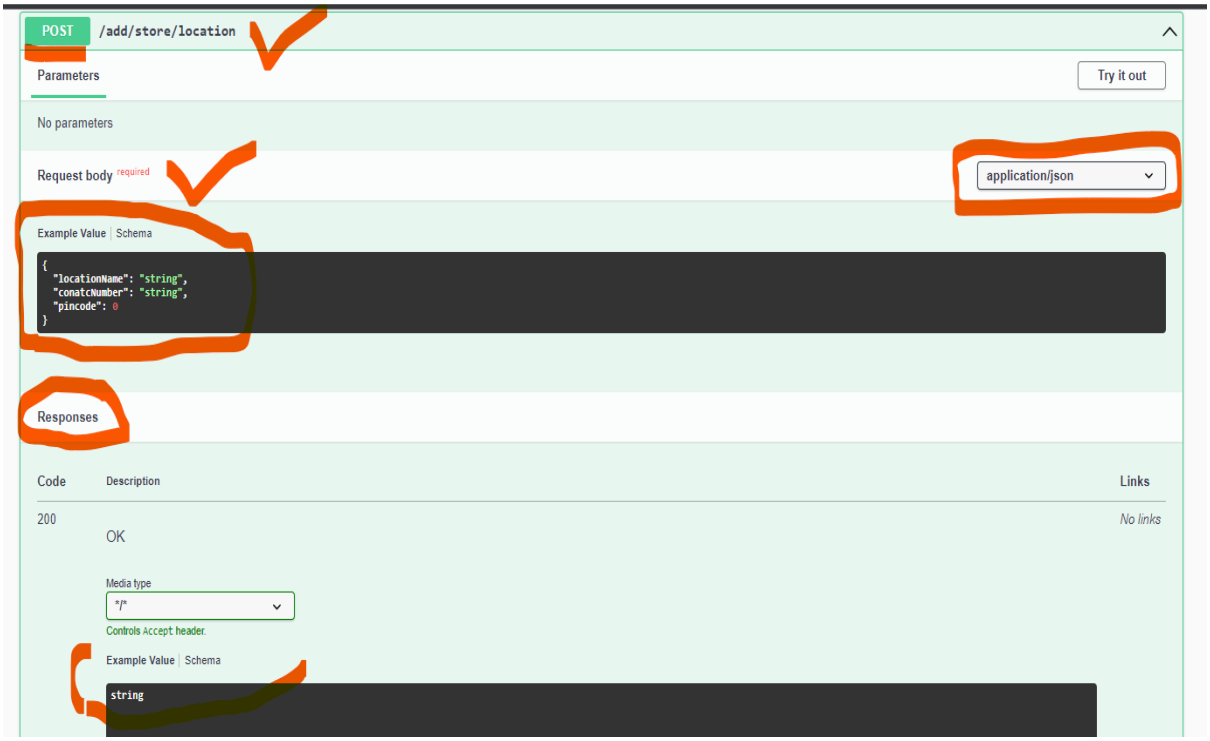
Below snap shows what are all services available in side pharmacy application.



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.



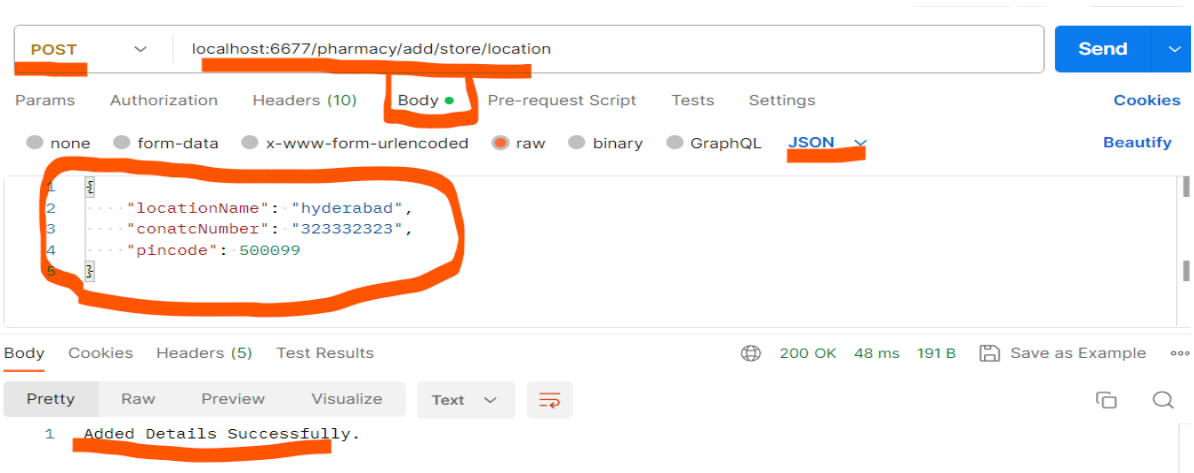
From above swagger snap, we should understand below points for that API call.

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.



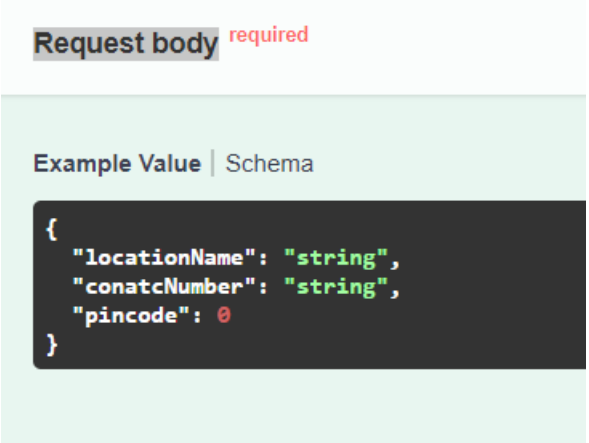
Now based on above data, we are going to write logic of RestTemplate to consume in our application flipkart.

Now will receive 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.
- Crate a POJO class which is equal to JOSN Request payload of Pharmacy API call.



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
}
```

- 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();

}

}
```

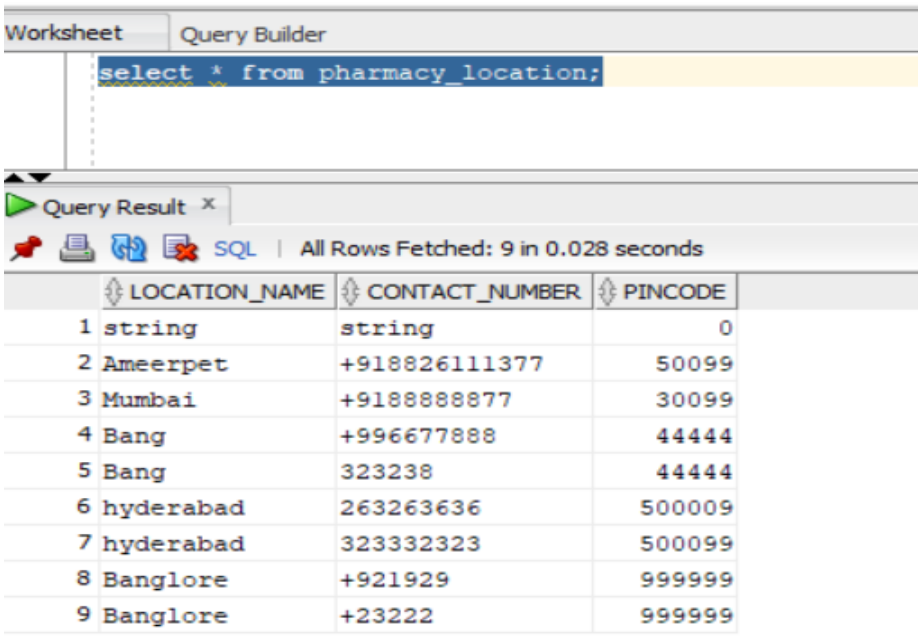
Now Test from postman and check pharmacy API call triggered or not i.e. check data is inserted in DB or not from pharmacy application.

flipkart URL : **localhost:9966/flipkart/pharmacy/add/location**

Noe create body as per our controller request body class.

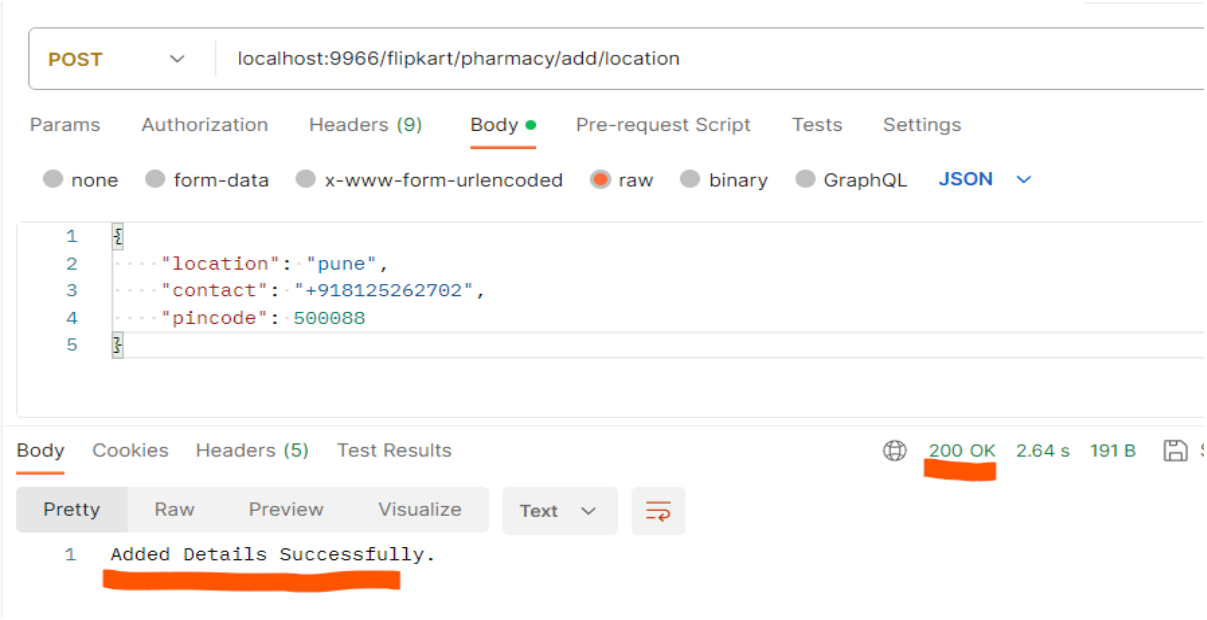
```
{
        "location": "pune",
        "contact": "+918125262702",
        "pincode": 500088
}
```

Before executing from post man, please check DB data. In my table I have below data right now.



	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.**



Request executed successfully and you got response from Pharmacy API of post call what we integrated. Verify In Database record inserted or not. It's inserted.

Worksheet | Query Builder

```
select * from pharmacy_location;
```

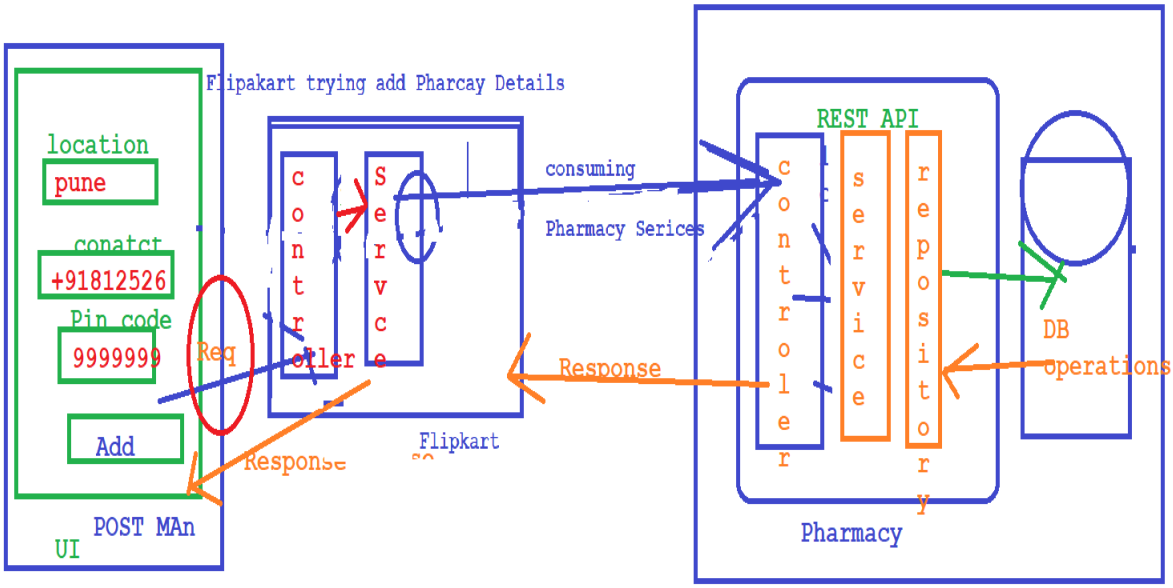
Query Result

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	+9188888877	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

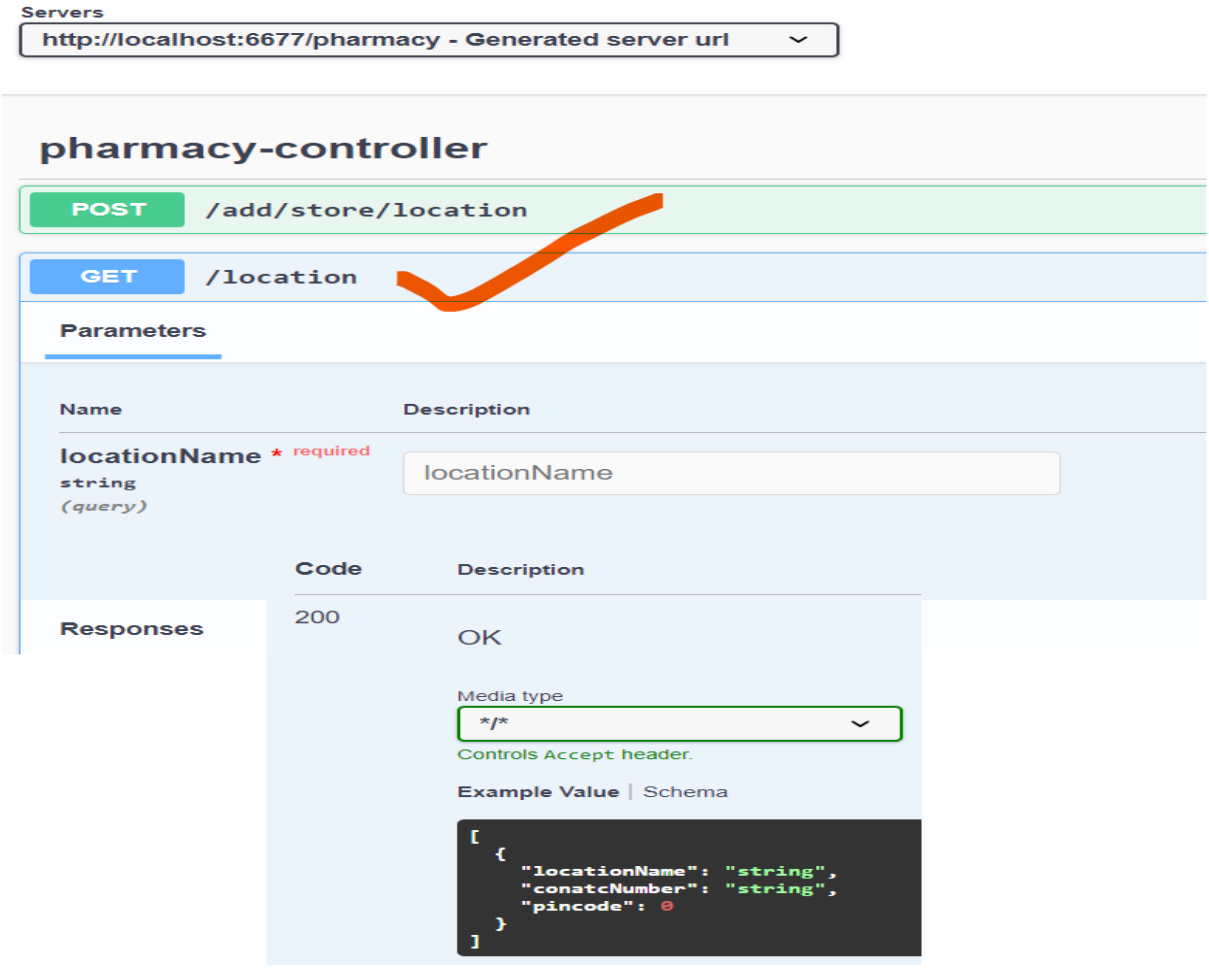
Internal Execution Workflow:

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:

Example1 : Consume below Service which contains Query String i.e. Query Parameters.



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

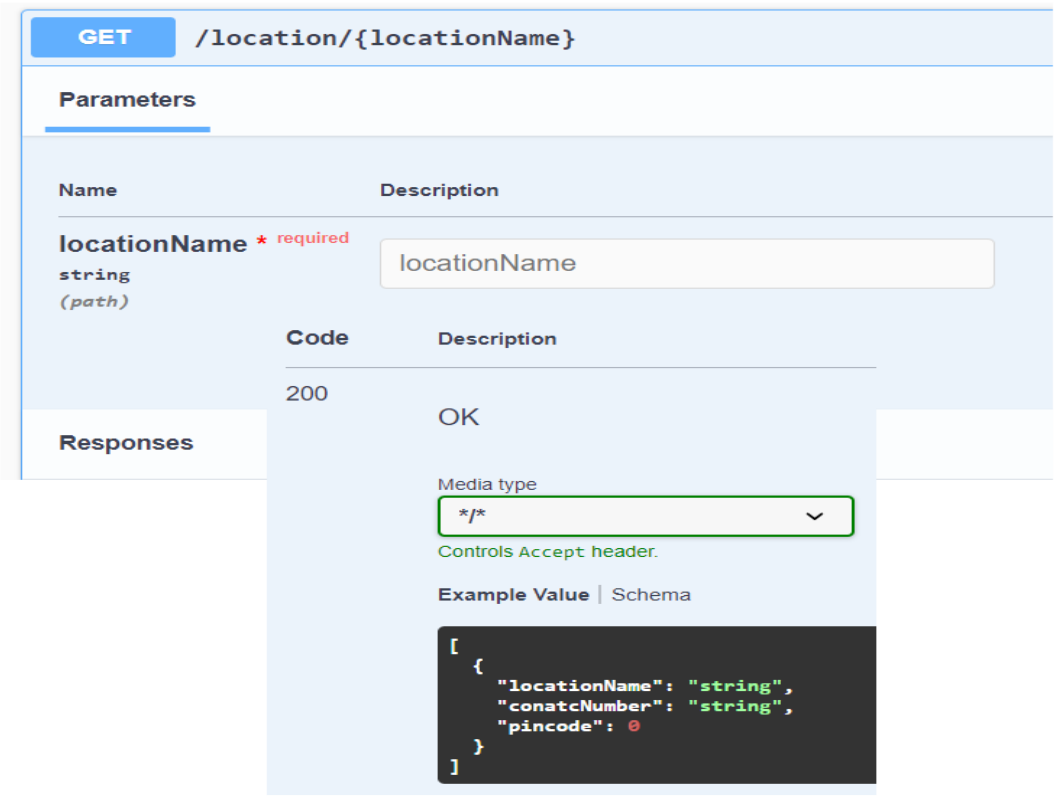
```
public class PharmacyResponse {

	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}";

// Creating Http Header for setting Headers Information
HttpHeaders headrs = new HttpHeaders();
headrs.setAccept(List.of(MediaType.APPLICATION_JSON));
HttpEntity<String> entity = new HttpEntity<String>(headrs);

// For Query parameters
Map<String, String> values = new HashMap<>();
values.put("locationName", location);   // passing value with location

RestTemplate restTemplate = new RestTemplate();
List<PharmacyResponse> respone = restTemplate.exchange(url, HttpMethod.GET,    entity, List.class,
values).getBody();

return respone;
}
```

Example2 : Consume below Service which contains Path Variable.



Consuming GET API Service with Path Parameter:

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.

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.**

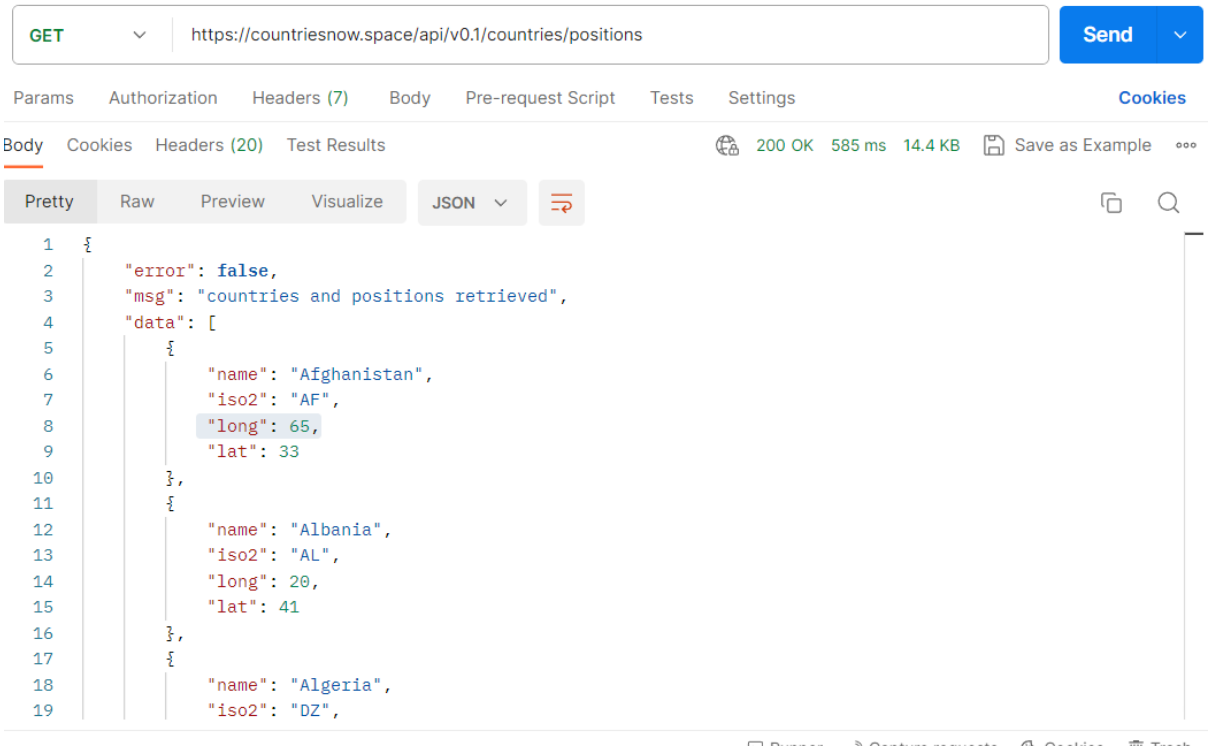
```
public List<PharmacyResponse> loadByLocationName(String location) {

String url = "http://localhost:6677/pharmacy/location/{locationName}";

// Creating Http Header
HttpHeaders headrs = new HttpHeaders();
headrs.setAccept(List.of(MediaType.APPLICATION_JSON));
HttpEntity<String> entity = new HttpEntity<String>(headrs);

Map<String, String> values = new HashMap<>();
values.put("locationName", location);

RestTemplate restTemplate = new RestTemplate();
List<PharmacyResponse> respone = restTemplate.exchange(url, HttpMethod.GET, entity, List.class,
values).getBody();

return respone;

}
```

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.

Example 3: We are Integrating one Real time API service from Online.

REST API GET URL: https://countriesnow.space/api/v0.1/countries/positions

Producing JSON Response, as shown in below Postman.



Based on 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
}
```

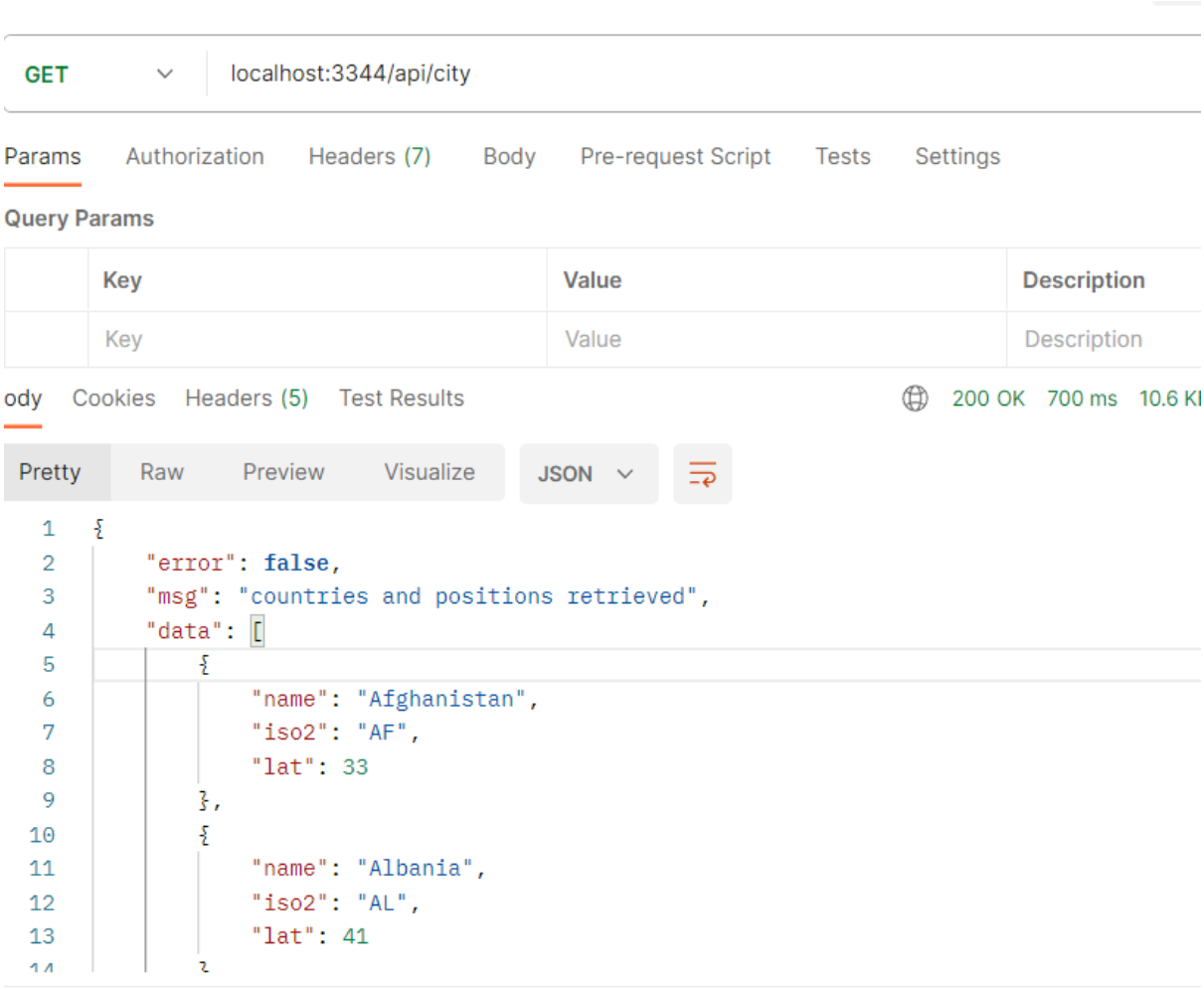
CountriesResponse.java

```
public class CountriesResponse {

        private boolean error;
        private String msg;
        List<Country> data;

        //Setters and Getters

}
```

Consuming Logic:

```
public CountriesResponse loadCities() {

	String url = "https://countriesnow.space/api/v0.1/countries/positions";
	// Creating Http Header
	HttpHeaders headrs = new HttpHeaders();
	headrs.setAccept(List.of(MediaType.APPLICATION_JSON));
	HttpEntity<String> entity = new HttpEntity<String>(headrs);

	RestTemplate restTemplate = new RestTemplate();
	CountriesResponse respone = restTemplate.exchange(url, HttpMethod.GET, entity,
	CountriesResponse.class).getBody();

	System.out.println(respone);

	return respone;
}
```

Testing from our Application:



@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:

```
server.port=8899
server.servlet.context-path=/citi
bank.name=CITI BANK
bank.main.location=USA
bank.total.emplyees=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 Controller class.

```
@RestController
public class CitiController {

	@Value("${citi.db.userName}")
	String dbName;

	@Value("${bank.total.emplyees}")
	int totalEmplyeCount;

	@Value("${server.port}")
	int portNumber;

	@GetMapping("/values")
	public String testMethod() {
		return "DB User Name: " + dbName;
	}

}
```

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: application.properties

```
trainingCourses = java,python,angular
```

Inside Bean 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: application.properties

```
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**.

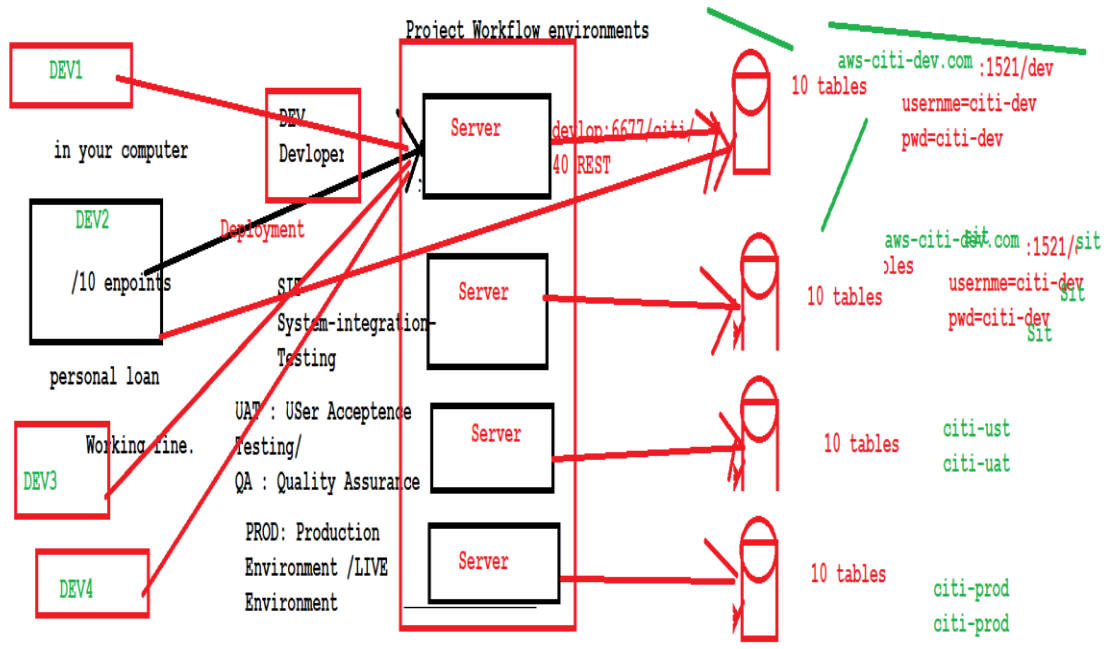
If we want different values for different arguments then we can use @Value annotation directly with the argument.

```
@Value("Test")
public void printValues(String value1, @Value("Data") String value2){

}
// value1=Test, value2=Data
```

Similarly we can use @Value along with Constructor arguments.

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.

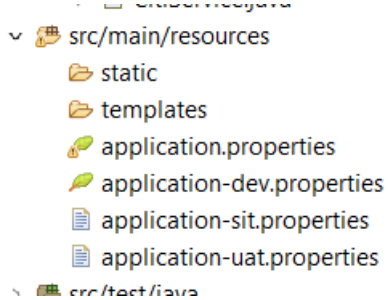


Each environment requires a setting that is specific to them. For example, in DEV, we do not need to constantly check database consistency. Whereas in UAT and PROD, we need to.

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:

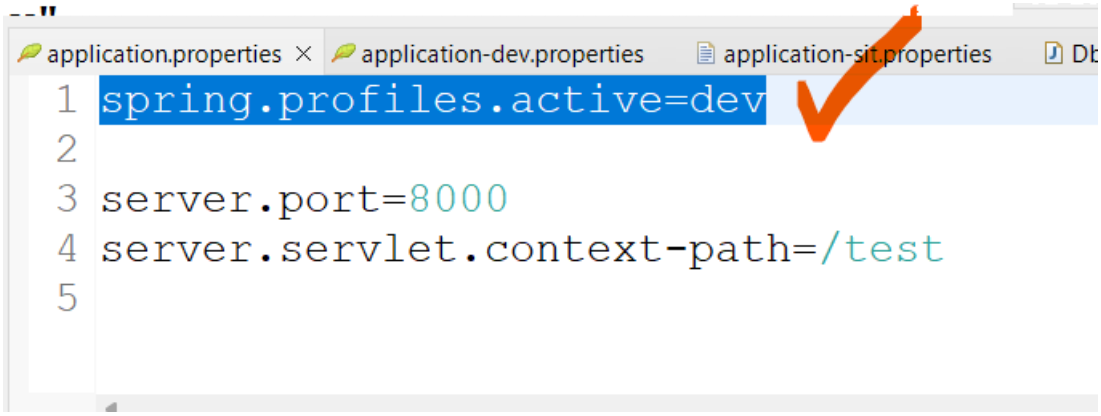
- application-dev.properties
- application-sit.properties
- application-uat.properties

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.propertie file.

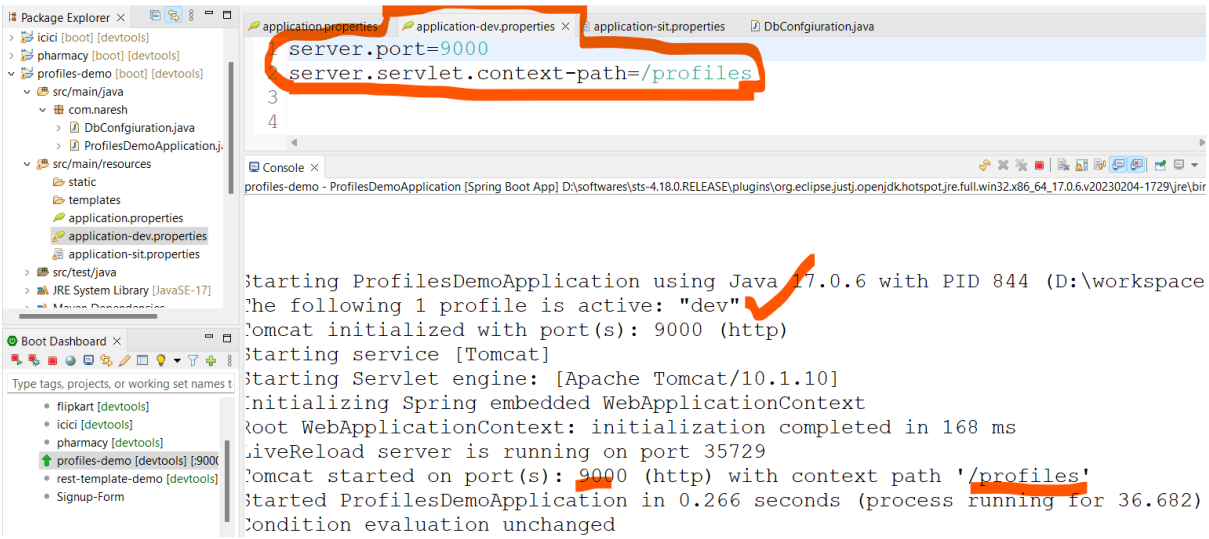
How to run Application with Specific Profile:

In side **application.properties** file, we have to add a property called **spring.profiles.active** with profile value.



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.



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.

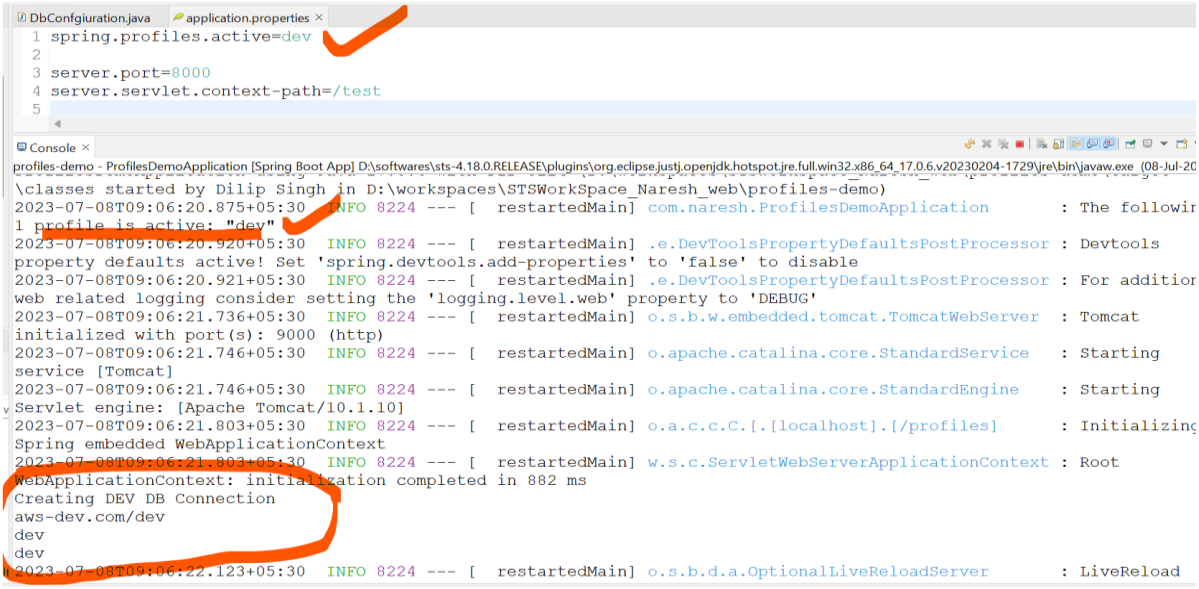
For Example, Database Connection should be created for specific profile or environment from configuration class.

```
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 DbConfgiuration {

	@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 Sccusessfu.";
	}

	@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 Sccusessfu.";
	}
}
```

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.

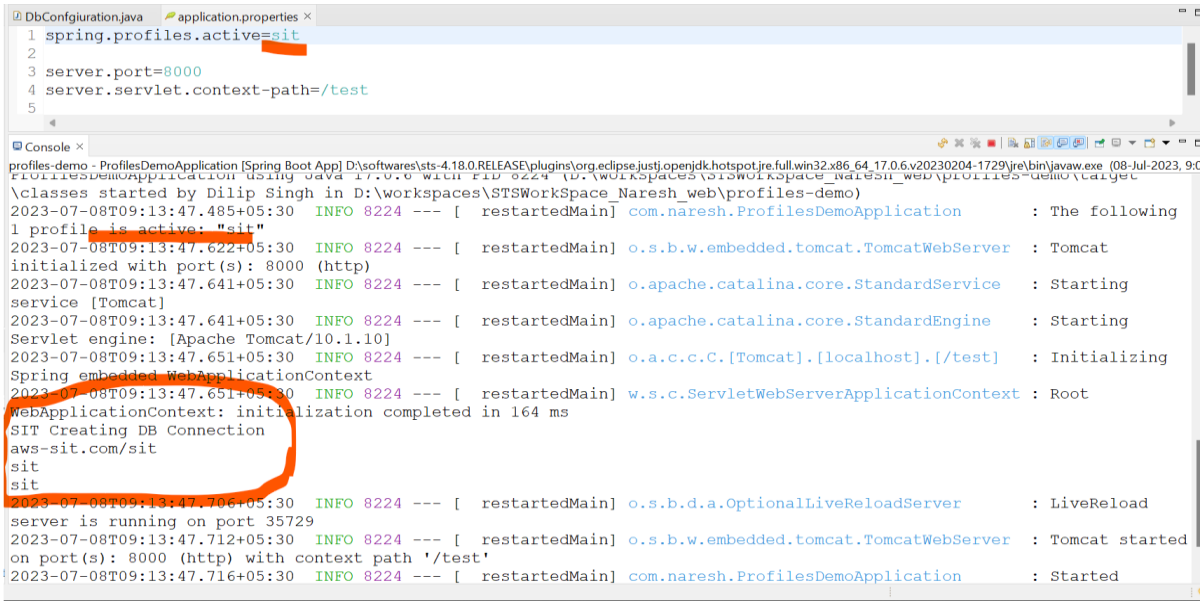
How application knows that this is DEV or SIT profile? how do we do this?
We will use **application.properties** with property **spring.profiles.active**=<profile>

From here, Spring Boot will know which profile to pick. Let's run the application now!



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.



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.

SpringBoot
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.

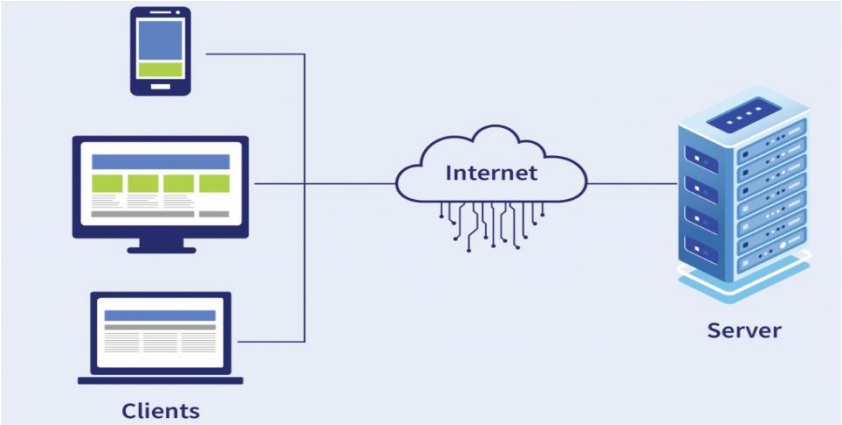
Security Interceptors: Spring Security uses security interceptors to enforce authorization rules. These interceptors are responsible for intercepting requests and checking whether the user has the required permissions to access the requested resource. They can be configured to protect URLs, methods, 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.

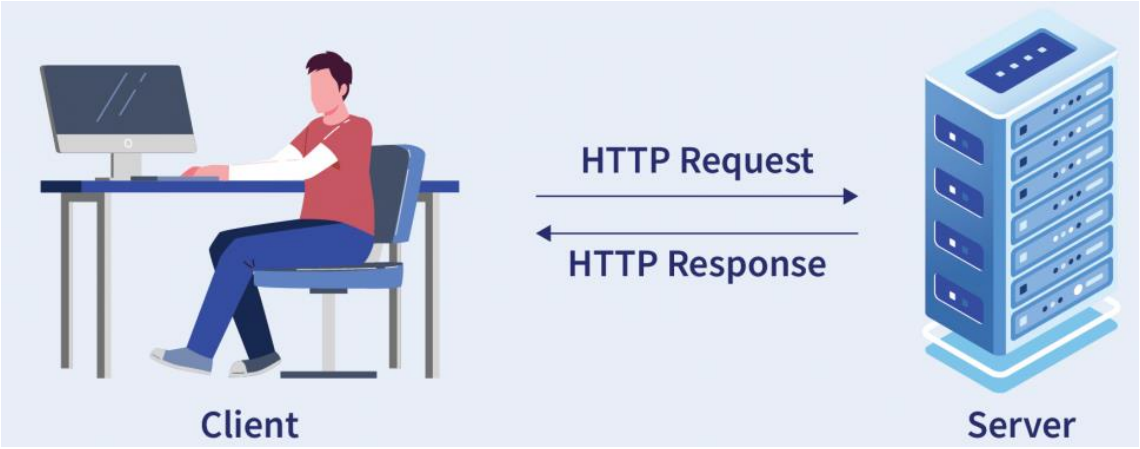
Stateless and Stateful Protocols:

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.

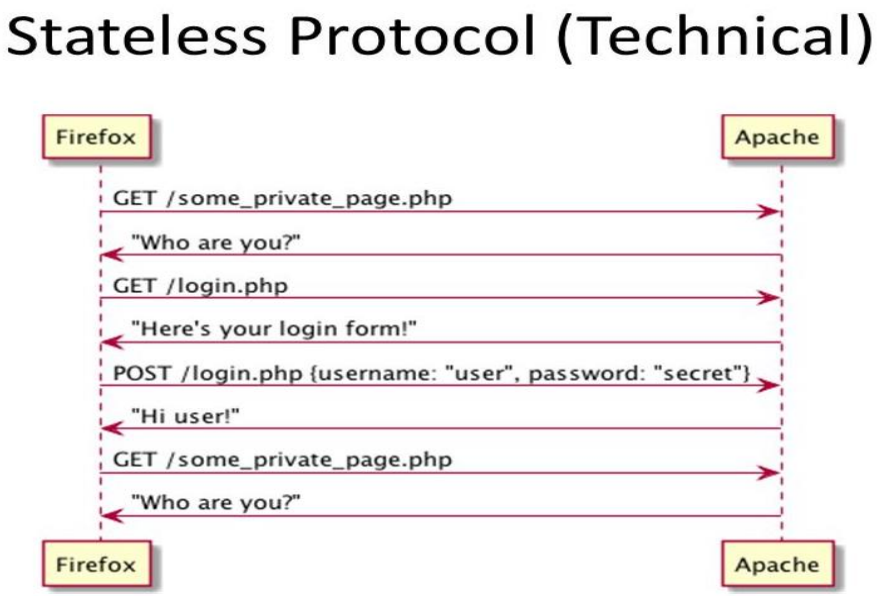


Stateless protocols have the following characteristics:

- 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

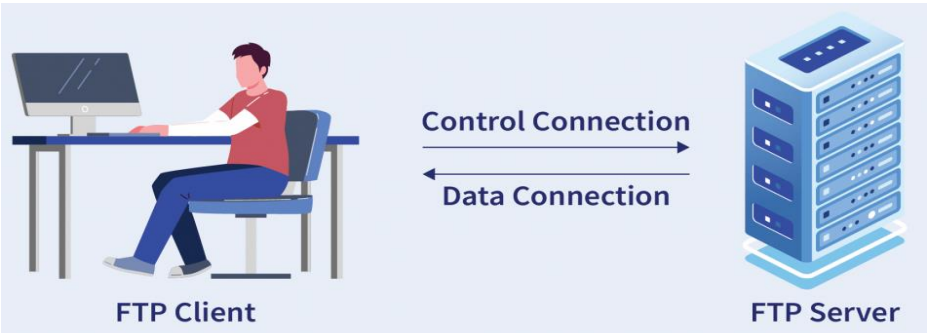
simplifies the server's implementation and scalability but presents challenges for handling user sessions and maintaining continuity between multiple requests.



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:

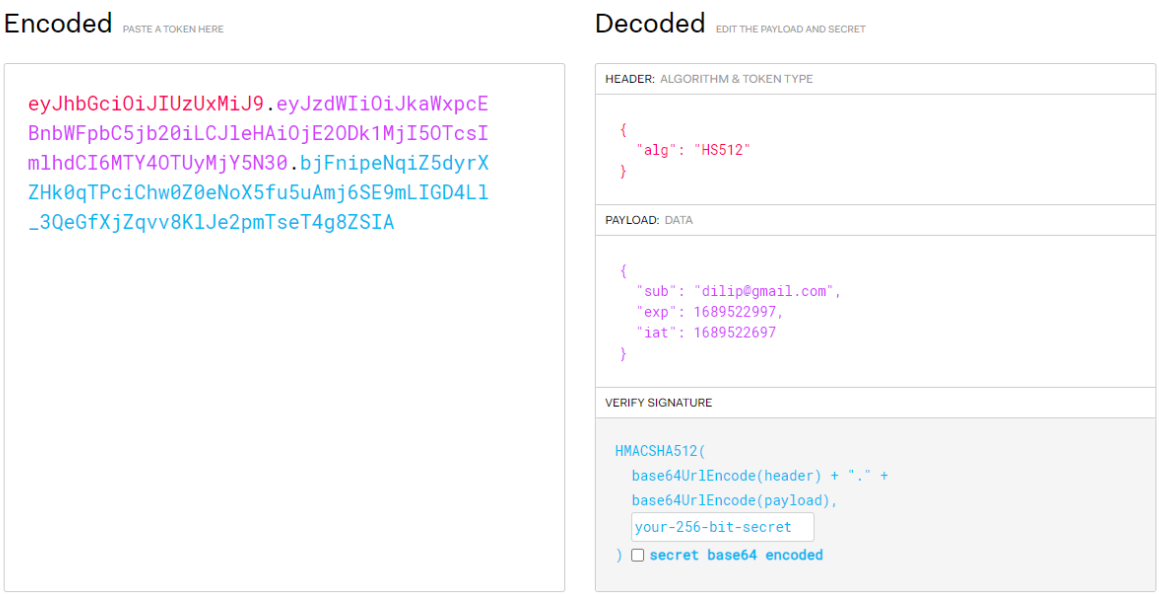
- **Header**: Consists of two parts:
  - The signing algorithm that's being used.
  - The type of token, which, in this case, is mostly "JWT".
- **Payload**: The payload contains the claims or the JSON object of clients.
- **Signature**: A string that is generated via a cryptographic algorithm that can 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.eyJzdWIiOiJkaWxpcEBnbWFpbC5jb20iLCJleHAiOjE2ODk
1MjI5OTcsImlhdCI6MTY4OTUyMjY5N30.bjFnipeNqiZ5dyrXZHk0qTPciChw0Z0eNoX
5fu5uAmj6SE9mLIGD4Ll_3QeGfXjZqvv8KlJe2pmTseT4g8ZSIA
```

Following image showing details of Encoded Token.



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.

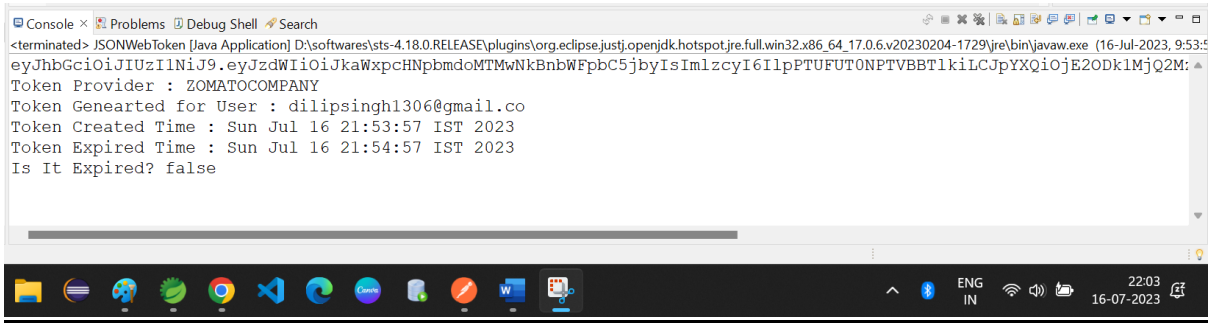
```
<dependencies>
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt</artifactId>
			<version>0.9.1</version>
		</dependency>
		<dependency>
			<groupId>javax.xml.bind</groupId>
			<artifactId>jaxb-api</artifactId>
			<version>2.3.0</version>
		</dependency>
</dependencies>
```

- **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.Jwts;
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 = (Claims)
		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 Genearted 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:



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/tek-teacher/javaJWTToken

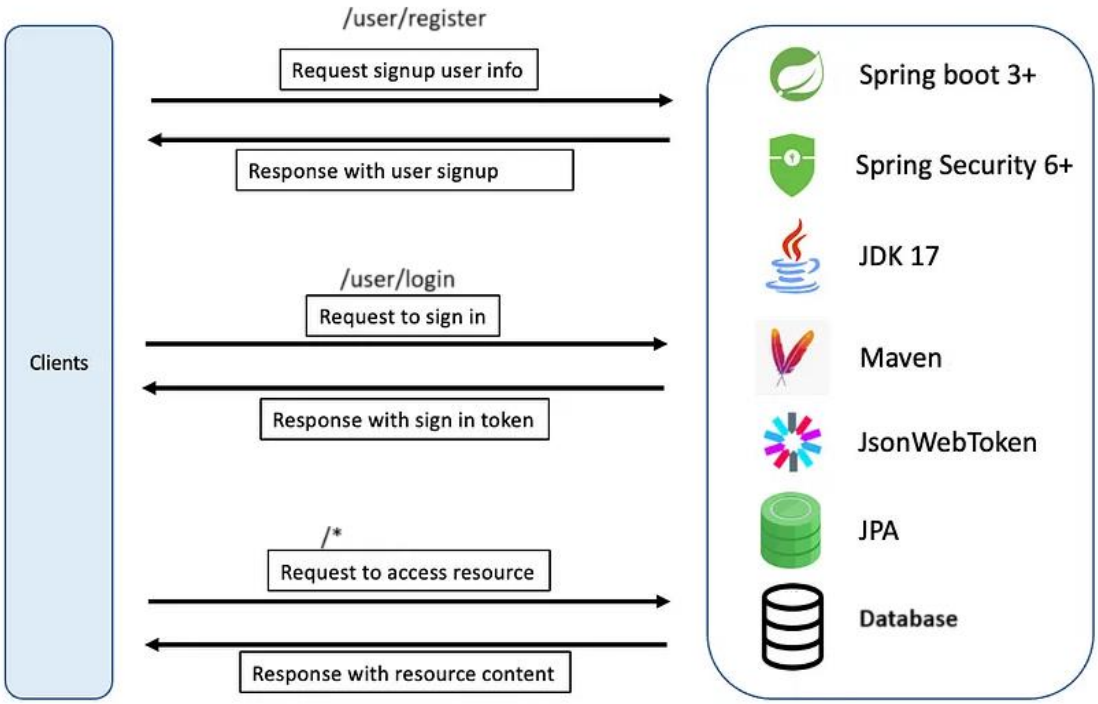
SpringBoot Security + (JSON WEB TOKEN):

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.

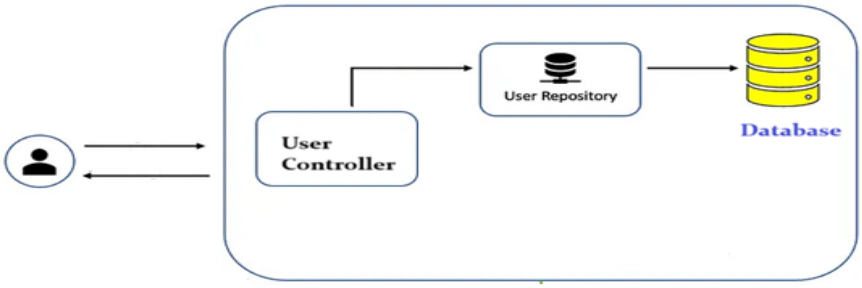
Application Architecture: Scenarios:

- 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.



Signup Flow

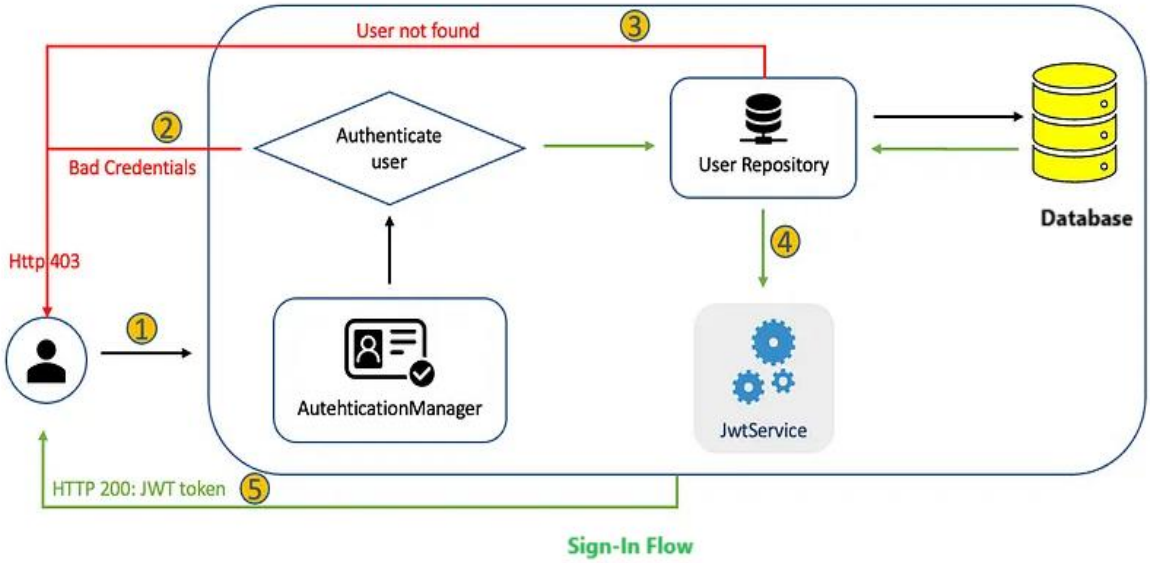
- The process starts when a user submits a request to our service. A user object is then generated from the request data, and we should encode password before storing inside Database. The password being encoded by using Spring provided Password Encoders.

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:



Sign-In Flow

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 defne 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.

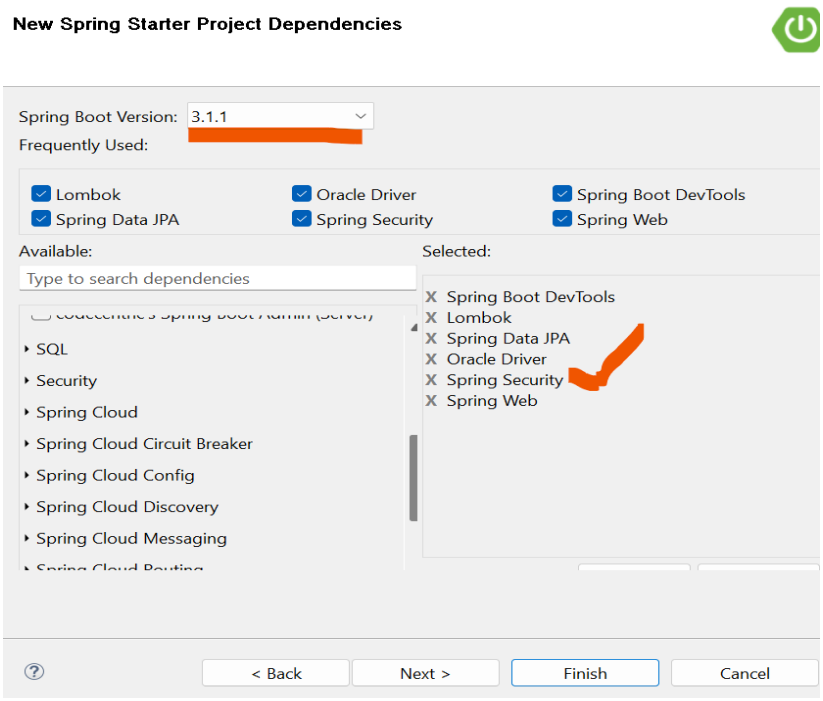
SecurityContextHolder: The **SecurityContextHolder** class is responsible for managing the **SecurityContext** object, which holds the security-related information. The **SecurityContext** contains the Authentication object representing the current user's authentication details, such as their principal (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.

<u>Security Logic Implementation:</u> Now Create Spring Boot Application with Security API:



- After Successsful Project creation, we should add JWT librarys dependecis inside Maven pom.xml file because by default SpringBoot not provding 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:orcl
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 annoatations.

- 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
@Getter
@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 syaHello() {
		return "Welcome to Security";
	}

	// User Sing-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 Credentils");

	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 "Registerded 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. Implantation **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");

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 = fasfafacasdasfasxASFACASDFACASDFASFASFDAFASFASDAADSCSDFADCVSGCFVADX";

   //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;
```

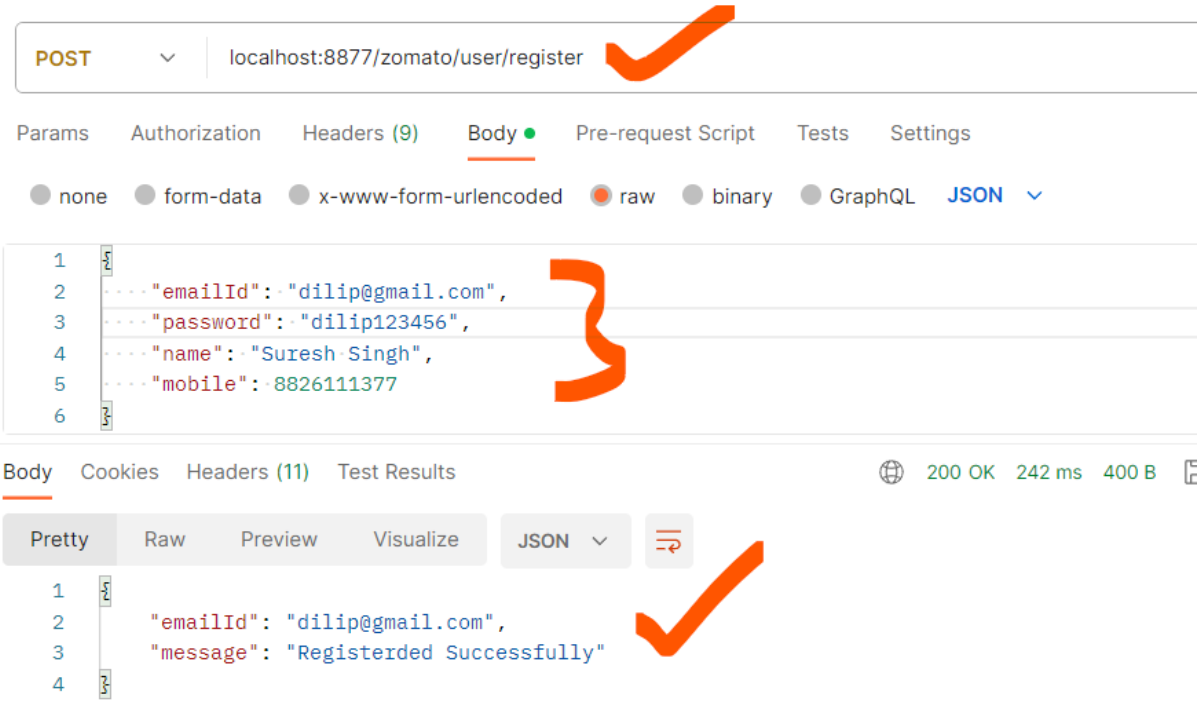
```
	@Bean
	AuthenticationManager	getAuthenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
		logger.info("Initilizing Bean AuthenticationManager");
		return authenticationConfiguration.getAuthenticationManager();
	}

	@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.

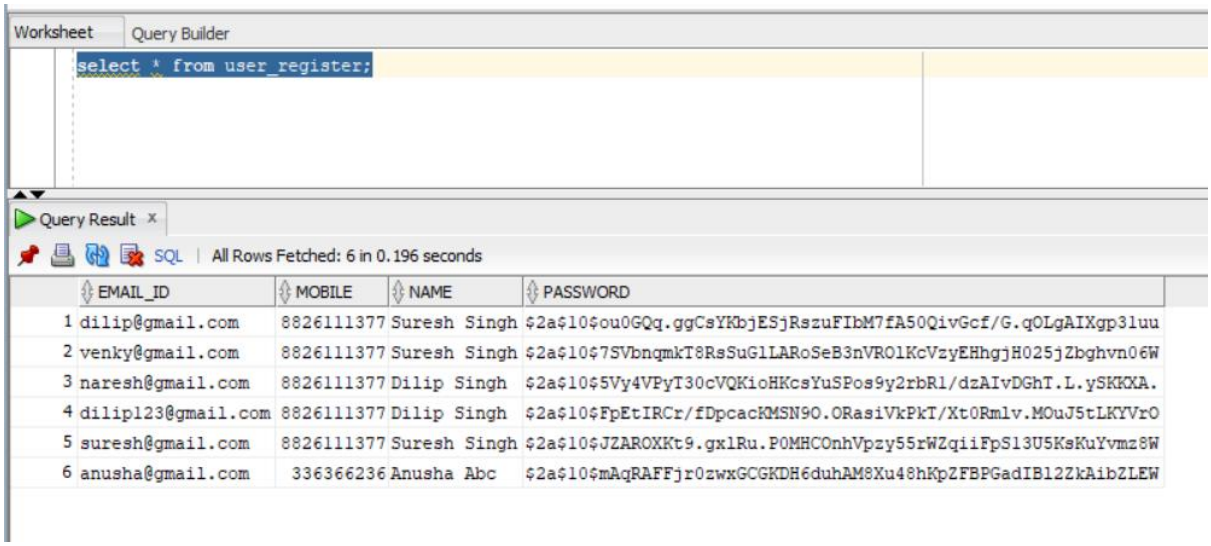
Testing : Let's Test our application as per our requirement and security configuration.

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.



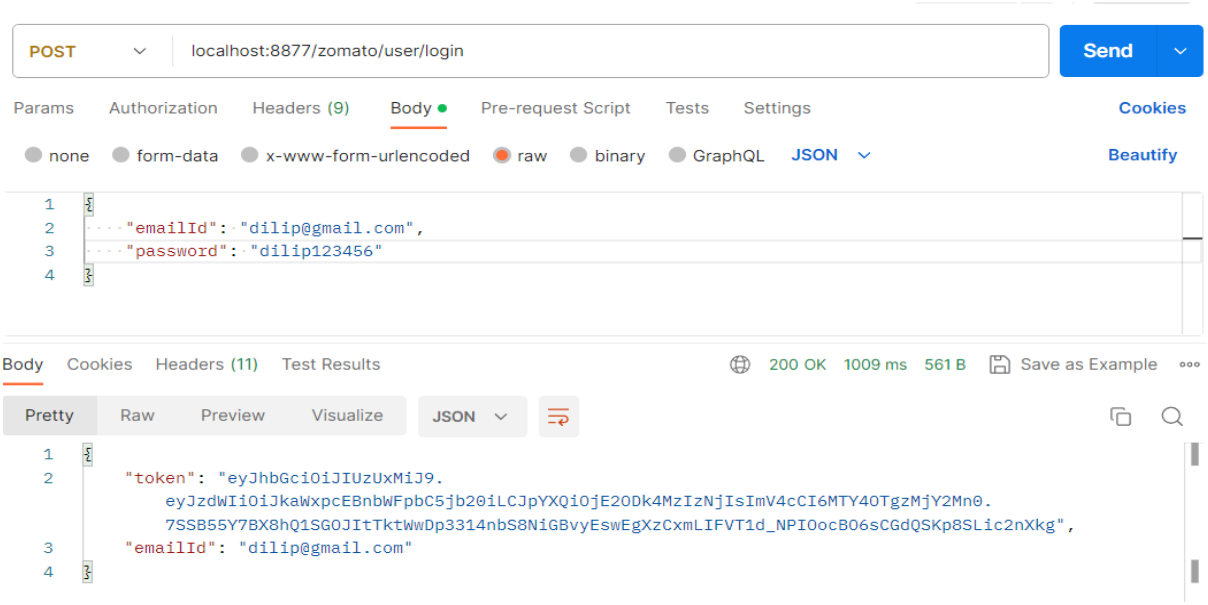
Now verify password value in database how it is stored because we encoded it.



	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$7SVbnqmkT8RsSuGlLARoSeB3nVROlKcVzyEHhgjH025jZbghvn06W
3	naresh@gmail.com	8826111377	Dilip Singh	$2a$10$5Vy4VPyT30cVQKioHKcsYuSPos9y2rbRl/dzAIvDGhT.L.ySKKXA.
4	dilip123@gmail.com	8826111377	Dilip Singh	$2a$10$FpEtIRCr/fDpcacKMSN9O.ORasiVkPkT/Xt0Rmlv.MOuJ5tLKYVrO
5	suresh@gmail.com	8826111377	Suresh Singh	$2a$10$JZAROXKt9.gxlRu.P0MHCOnhVpzy55rWZqiiFpS13U5KsKuYvmz8W
6	anusha@gmail.com	336366236	Anusha Abc	$2a$10$mAqRAFFjr0zwxGCGKDH6duhAM8Xu48hKpZFBPGadIBl2ZkAibZLEW

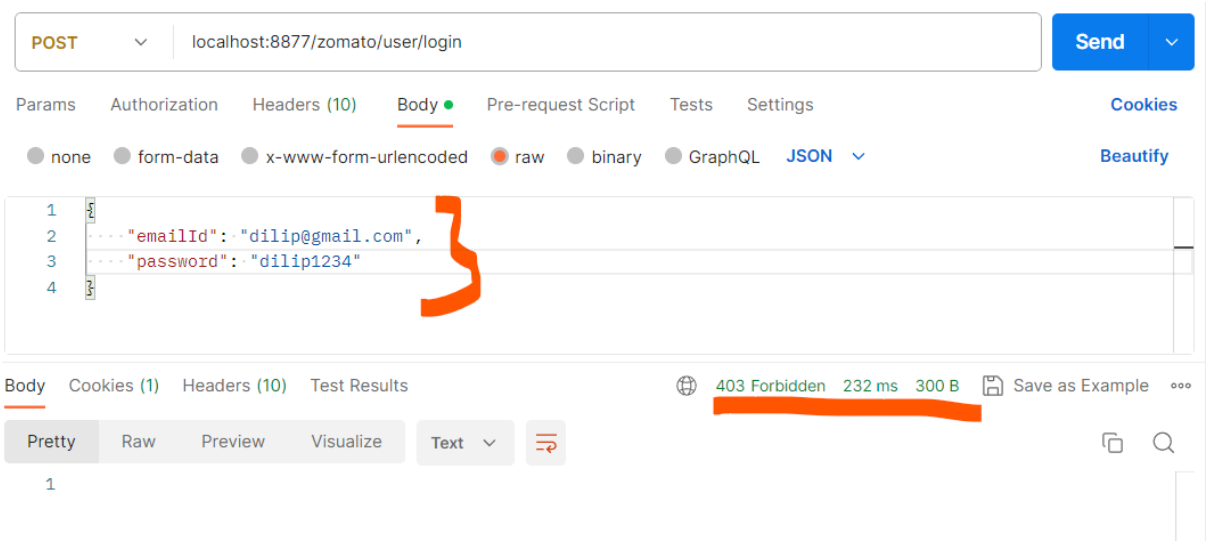
Passwords are stored as encoded format. Now Spring also takes care of decoded while authentication because we are created Bean of Password Encryptor.

Case 2: Now try to login with User Details. localhost:8877/zomato/user/login



On Successful validation, We received JSON Web token.

If we provide Wrong Credentials: Entered Wrong Password.

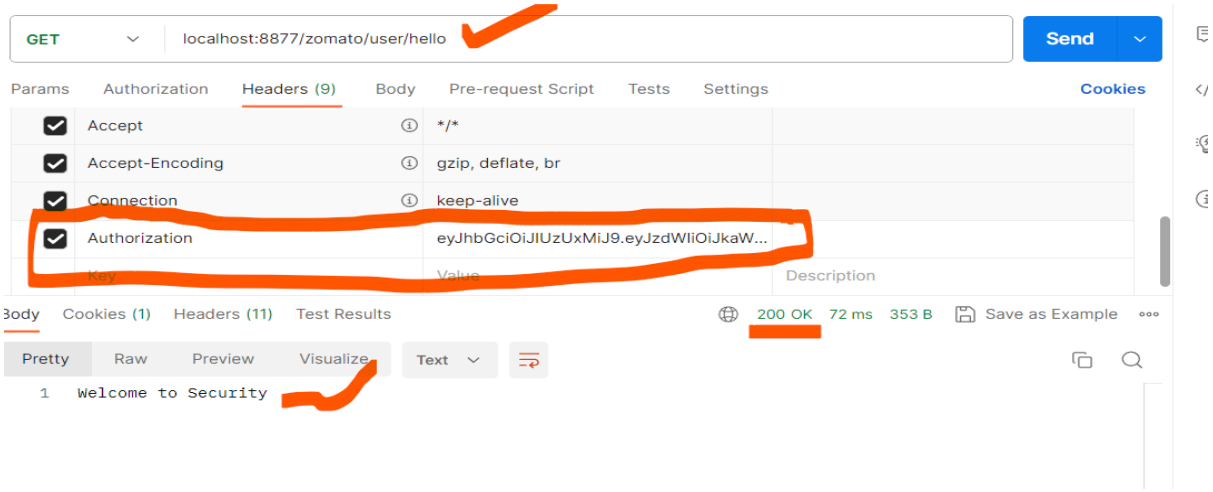


Now we got expected and default Response as Forbidden with status code 403:

Forbidden meaning is : **not allowed; banned.**

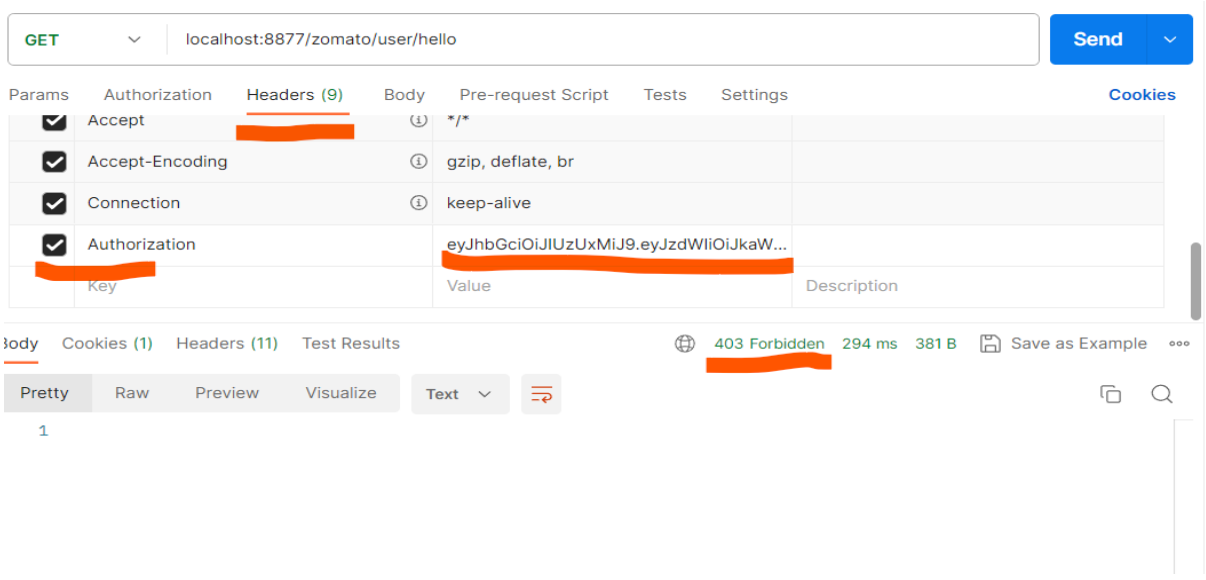
Case 3: Access Other URI or Services with JWT: localhost:8877/zomato/user/hello

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.



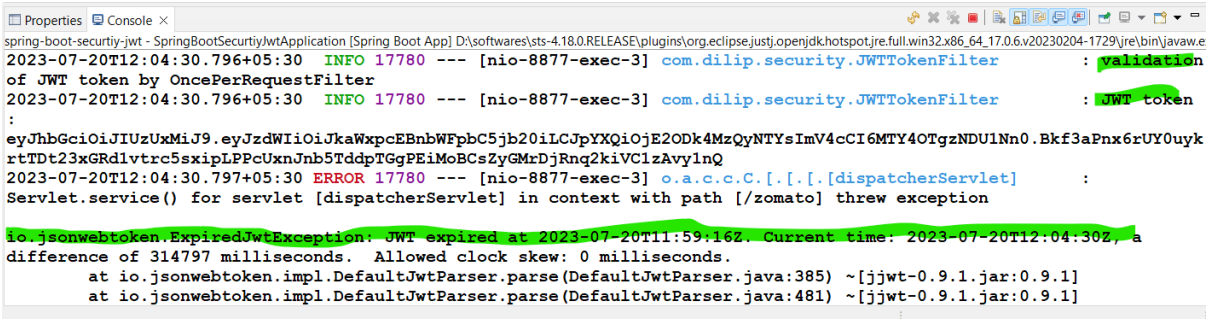
Now received respected Response value and Status code as 200 OK. i.e. Internally JWT is validated with user and database as per our logic implemented. We can see in Logs of application.

Case 4: Access Other URI or Services with Invalid JWT: localhost:8877/zomato/user/hello i.e. Provided Expired Token.



Token validated and found as Expired, so Server application returns Response as 403 Forbidden at client level.

Check Server Level logs:



This is how we are applying security layer to our SpringBoot Web Application with JWT exchanging.

I have uploaded this entire working copy project in GitHub.

GitHub Repository Link : https://github.com/tek-teacher/spring-boot-3-securtiy-jwt.git

SpringBoot 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:

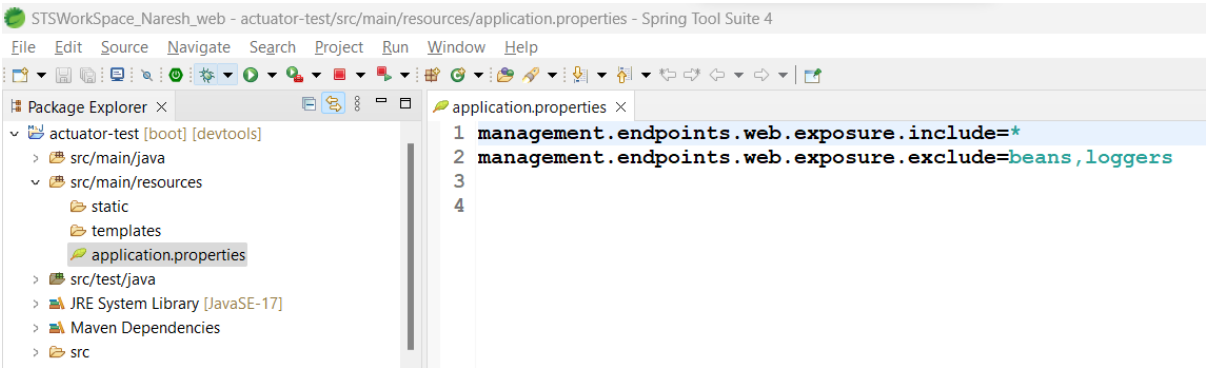
Properties:

```
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=env,beans
```

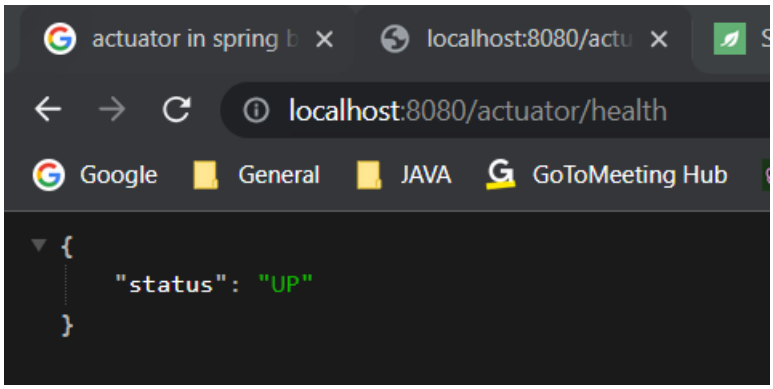
For security purposes, only the **/health** endpoint is exposed over HTTP by default. You can use the **management.endpoints.web.exposure.include** property to configure the endpoints that are exposed.

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.

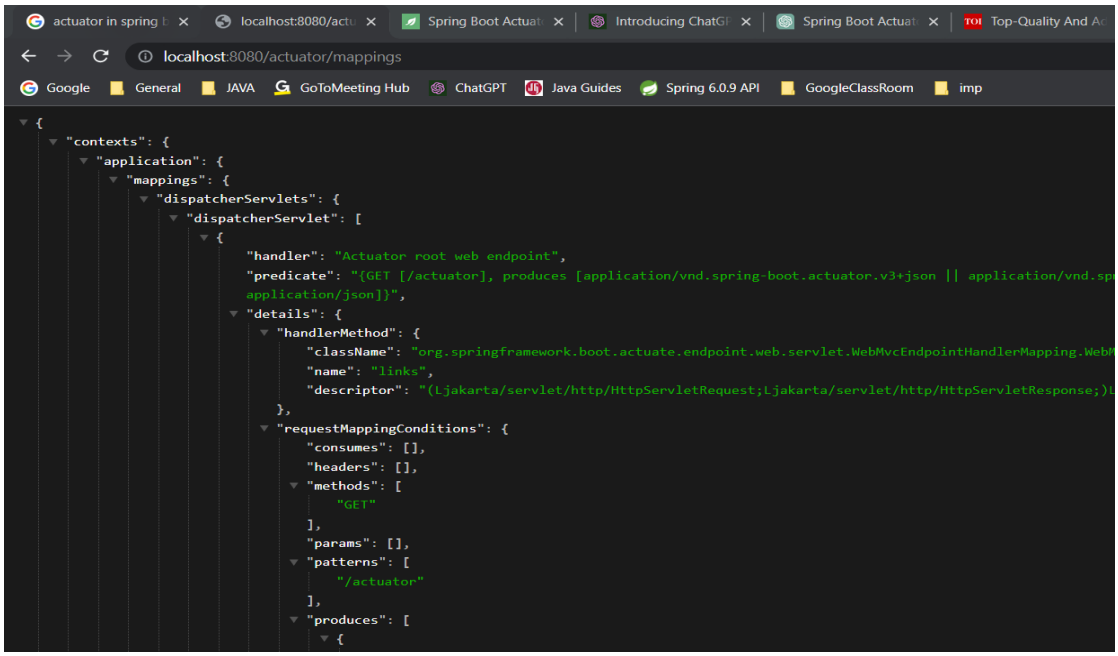
Configure Properties in **application.properties**:



Accessing Health Endpoint : Getting status as UP i.e. Application Started and Deployed Successfully.



Similarly, http://localhost:8080/**actuator/mappings**



Similar to above actuator endpoints, we can enable and access regards to their specifications.

Thank you
Dilip Singh
dilipsingh1306@gmail.com