

Spring Framework

By –

Dilip Singh

 dilipsingh1306@gmail.com

Follow Here for Updates



@DilipItAcademy

+91 8125262702

Spring Core Module

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

What is a programming language?

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

do we have only one language in our project?

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

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

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

What is a Framework?

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

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

What kind of tasks does a framework solve?

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

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

Do I need a framework?

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

Conclusion:

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

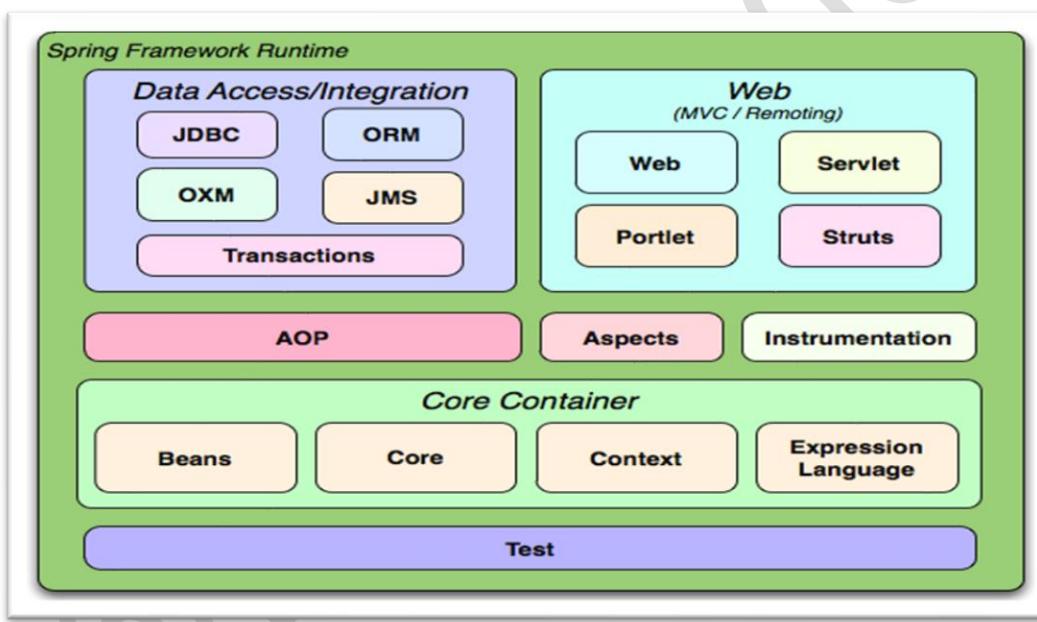
Overall, Spring Framework has become one of the most popular Java frameworks due to its ease of use, modularity, and extensive features. It is widely used in enterprise applications, web applications, and other types of Java-based projects.

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:

- 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.
- Spring Bean:** This module provides an implementation for the factory design pattern through BeanFactory.
- Spring Context:** This module is built on the solid base provided by the Core and the Beans modules and is a medium to access any object defined and configured.

Spring Bean:

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

Dependency Injection in Spring:

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

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

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

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

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

Employee.java

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

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

Address.java

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

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

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

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

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

```
}
```

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

Spring Container / IOC Container:

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

Inversion of Control:

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

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

The main tasks performed by IoC container are:

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

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

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

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

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

Spring provides two types of Container Implementations namely as follows:

1. BeanFactory Container
2. ApplicationContext Container

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

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

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

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

Spring – BeanFactory:

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

Spring ApplicationContext:

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

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

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

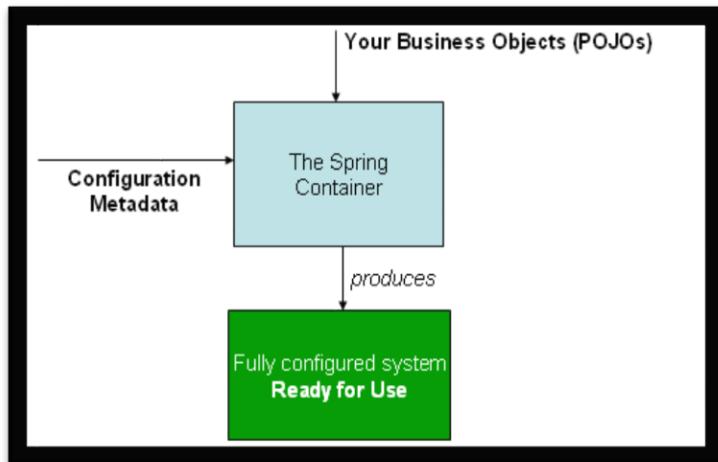


Figure :The Spring IoC container

Configuration Metadata:

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

Instantiating a Container:

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

`ApplicationContext` interface: `ClassPathXmlApplicationContext` and `FileSystemXmlApplicationContext` for standalone applications, and `WebApplicationContext` for web applications.

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

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

Difference Between BeanFactory Vs ApplicationContext:

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

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

Create First Spring Core module Application:

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

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

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

Creation of Project with Downloaded Spring Jar Files :

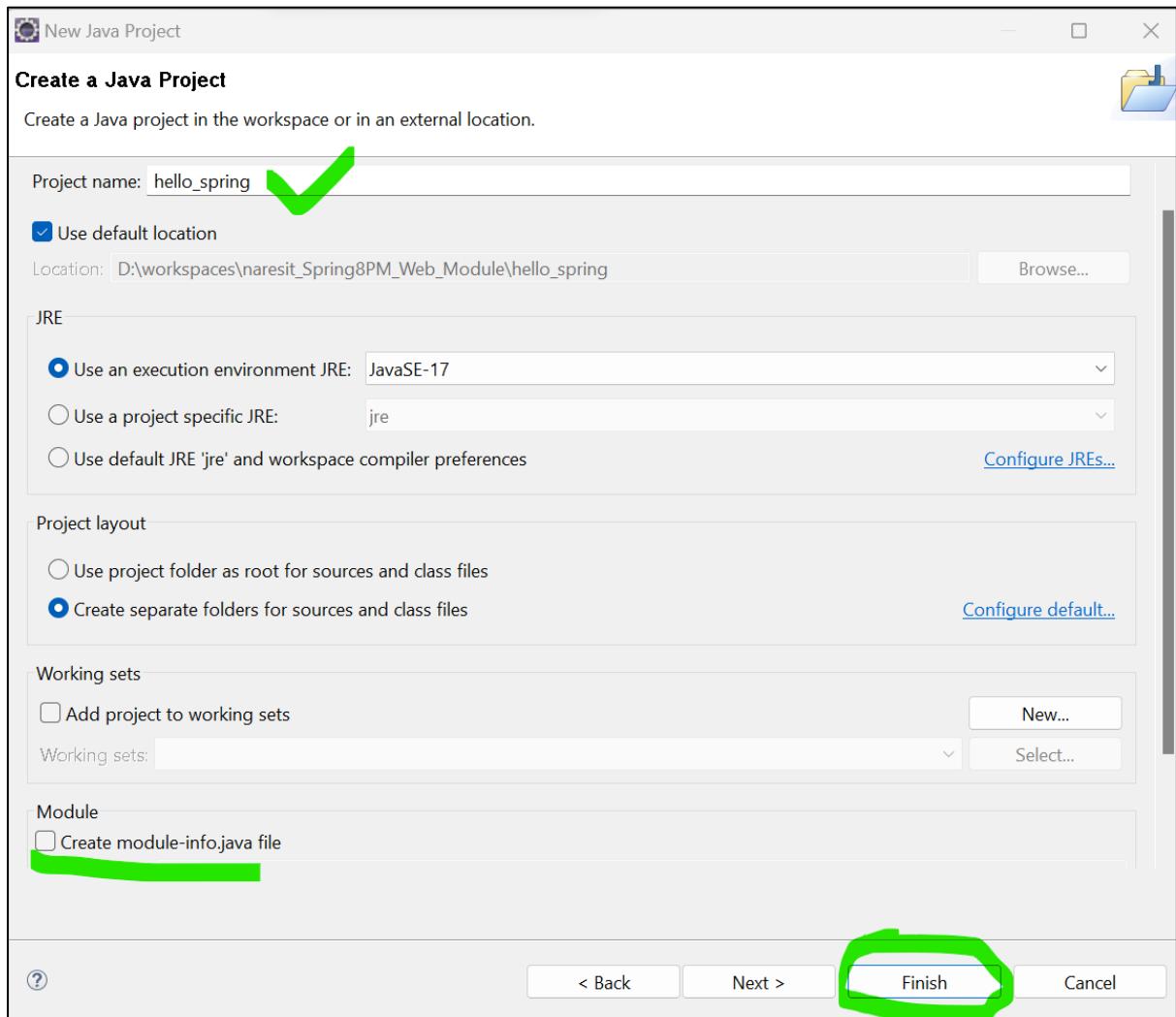
1. Open Eclipse

File -> new -> Project -> Java Project

Enter Project Name

Un-Select Create Module-Info

Click Finish.



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

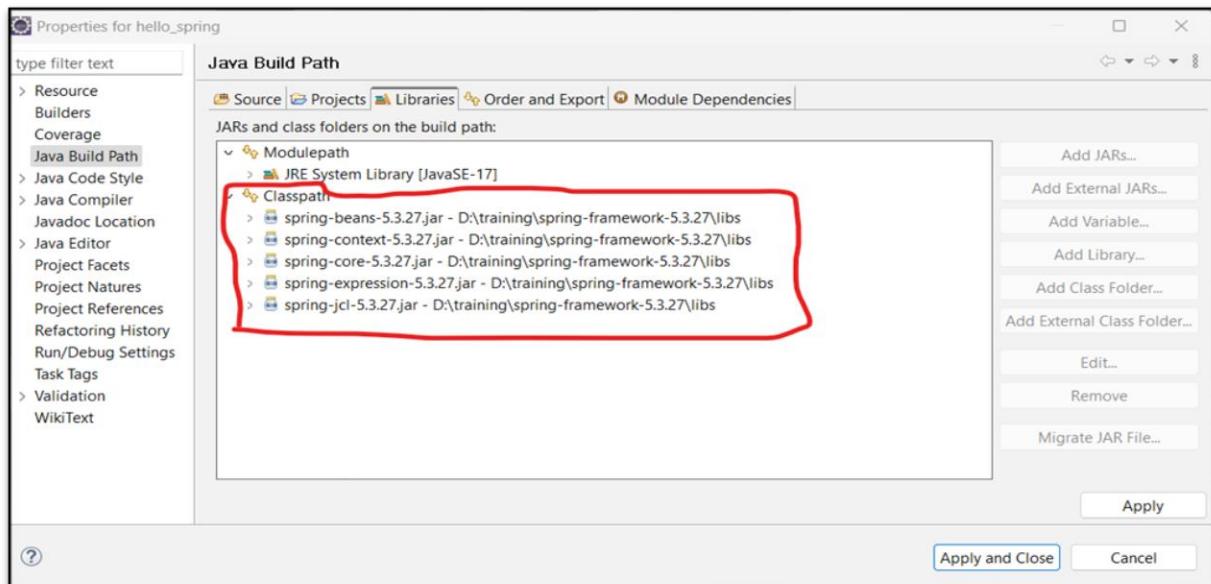
I have uploaded copy of all Spring JAR files uploaded in Google Drive. You can download from below link directly.

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

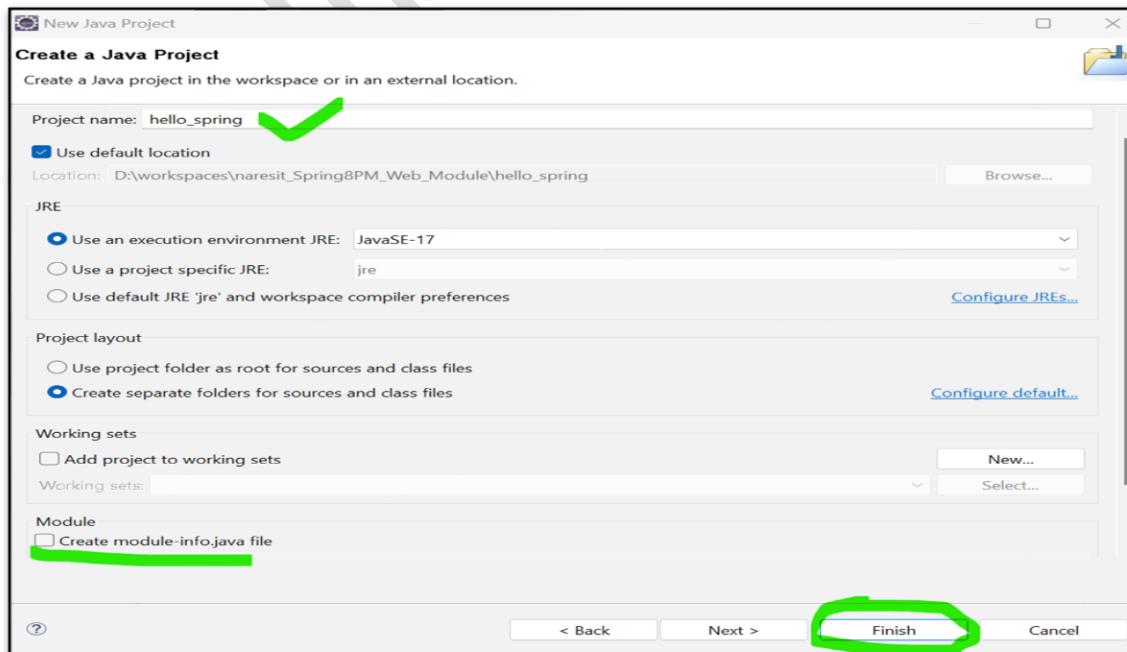


With This Our Java Project is Supporting Spring Core Module Functionalities. We can Continue with Spring Core Module Functionalities.

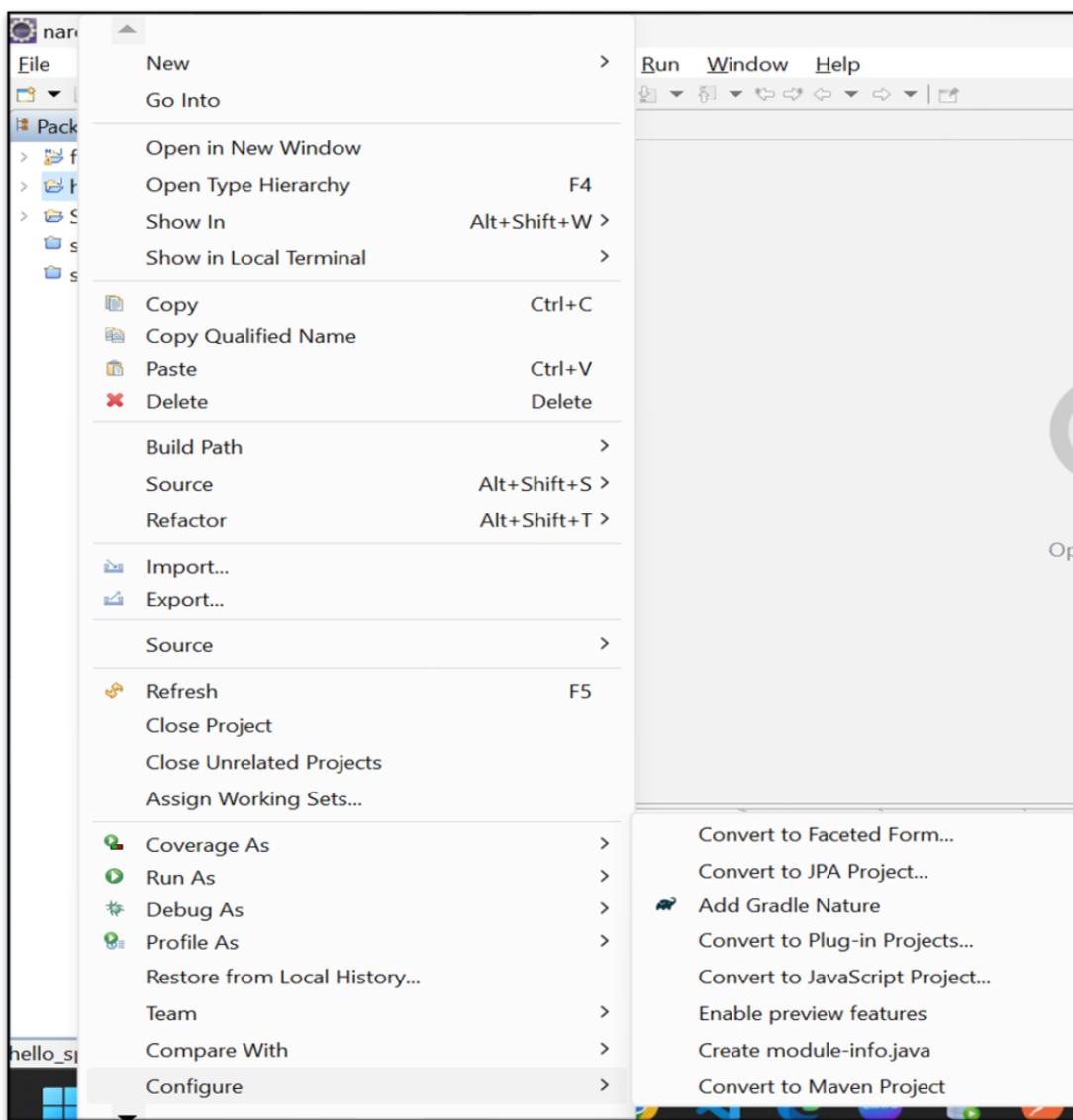
Creating Spring Core Project with Maven Configuration:

1. Create Java Project.

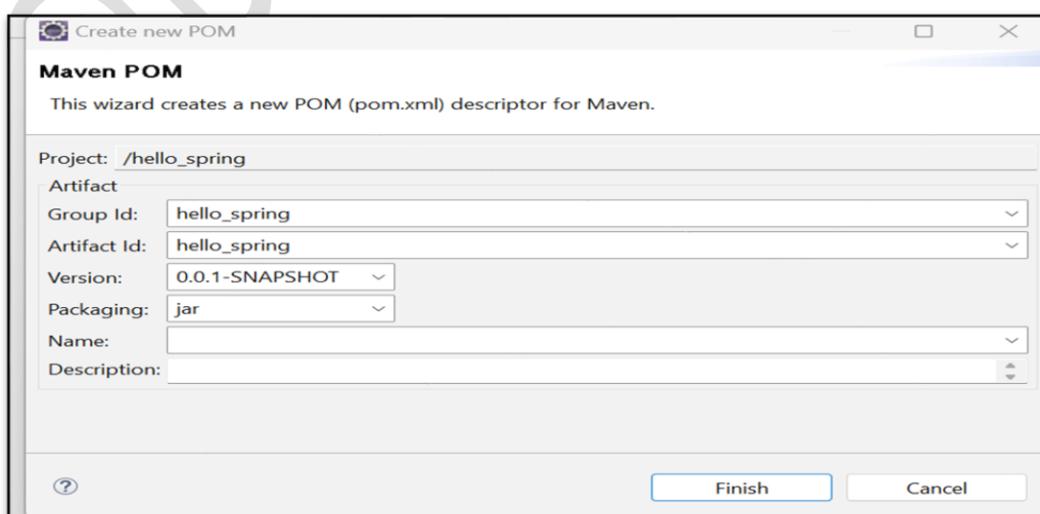
Open Eclipse -> File -> new -> Project -> Java Project -> Enter Project Name -> Un-Select Create Module-Info -> Click Finish.



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.

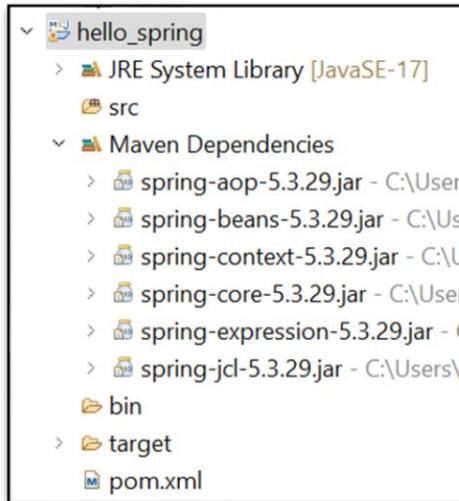


Now Open pom.xml file, add Spring Core JAR Dependencies to project and save it.

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

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

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



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

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

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

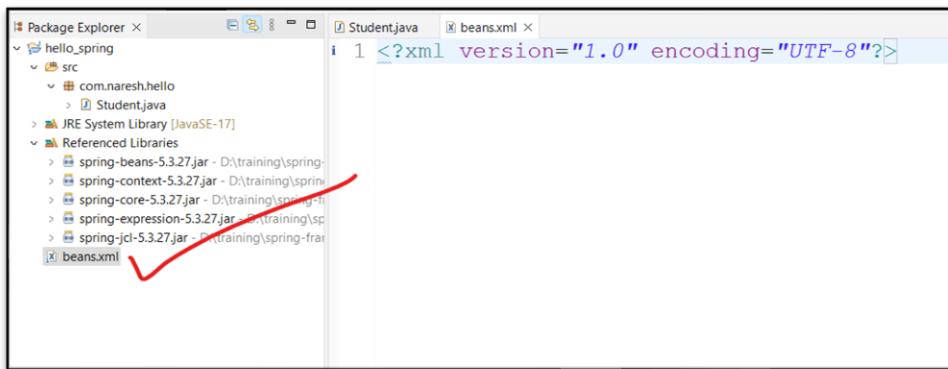
```
package com.naresh.hello;

public class Student {
    private String studnetName;

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

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

Ex: **beans.xml**



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

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

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

We can get above content from below link as well.

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

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

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

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

From above Configuration, Points to be Noted:

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

8. Now create a main method class for testing.

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

```
package com.naresh.hello;

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

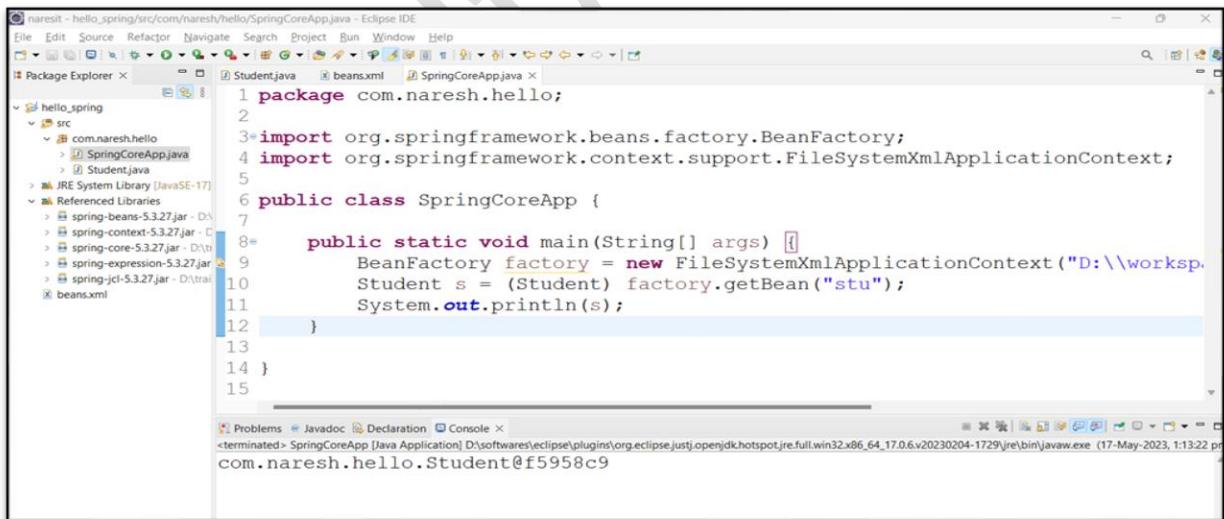
public class SpringCoreApp {
    public static void main(String[] args) {

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

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

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

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



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

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

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

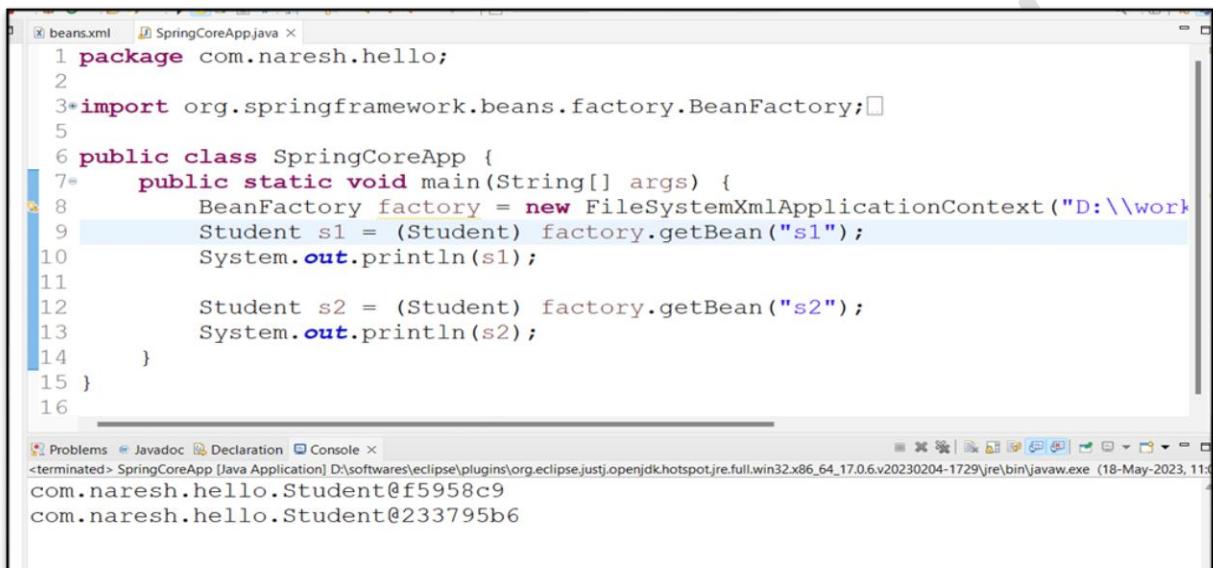
```

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

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

```

- Now In Main Application class, Get Second Object.



The screenshot shows the Eclipse IDE interface with two files open:

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

```
- SpringCoreApp.java:**

```

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

```

The Eclipse interface includes tabs for Problems, Javadoc, Declaration, and Console. The Console tab shows the output of the application's execution:

```

com.naresh.hello.Student@f5958c9
com.naresh.hello.Student@233795b6

```

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

Bean Overview:

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

Bean Naming Conventions:

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

Instantiating Beans:

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

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

Types of Dependency Injection:

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

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

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

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

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

Setter Injection:

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

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

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

Now Let's take example for setter injection.

1. Create a class.

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

public class Student {

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

2. Configure bean in **beans.xml** file :

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

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

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

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

<property> tag contains some attributes.

- **name:** Name of the Property i.e. variable name of Bean class
- **value:** Real/Actual value of Variable for injecting/storing

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

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

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

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

        BeanFactory factory =
            new
        FileSystemXmlApplicationContext("D:\\workspaces\\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
```

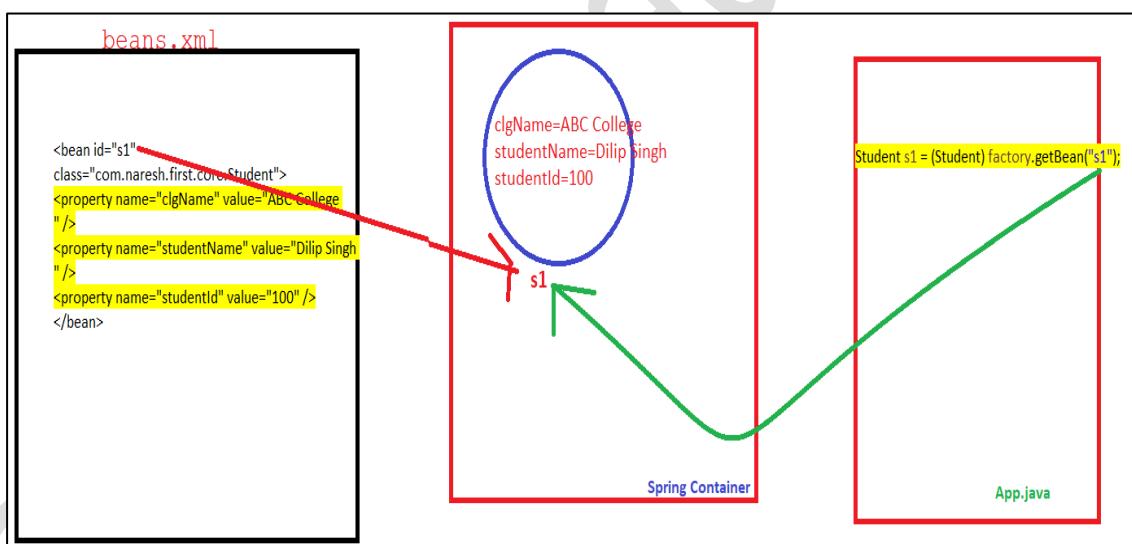
Internal Workflow/Execution of Above Program.

- 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\\nareshit\\spring_first\\beans.xml");
```

- Now with below code, we are getting bean object of **Student** configured with bean id : s1

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



- Now we can use **s1** reference and we can call methods of **Student** 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");
        // get Student instance from container
        Student s1 = (Student) context.getBean("studentOne");
        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 **<set>** 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.

```

package com.naresh.hello;

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

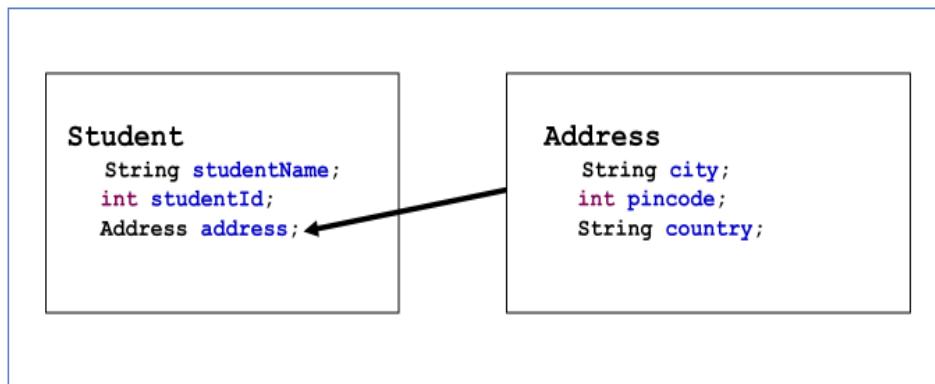
public class SpringCoreApp {
    public static void main(String[] args) {
        ApplicationContext context =
            new FileSystemXmlApplicationContext("D:\\\\workspaces\\\\naresit\\\\spring_notes\\\\beans.xml");
        // get student instance from container
        Student s1 = (Student) context.getBean("studentOne");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

Output:

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

Injecting Dependency's of Other Bean Objects:

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



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

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

public class Address {

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

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

```

    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
}

```

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

```

package com.naresh.training.spring.core;

public class Student {

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

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

```

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

ref attribute / <ref> element:

The **ref** element is the element inside a **<property>** or **<constructor-arg>** element. Here, you set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container. Sometimes we can use **ref** attribute as part of **<property>** or **<constructor-arg>**. We will provide bean Id for **ref** element which should be injected into target Object. Please refer below, how to inject Bean Objects via **ref** element or attribute.

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

```

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

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

```

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

```

package com.naresh.training.spring.core;

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

```

```

public class SpringSetterInjectionDemo {

    public static void main(String[] args) {

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

        System.out.println("***** Student1 Data *****");
        Student stu1 = (Student) factory.getBean("student1");
        System.out.println(stu1.getStudentId());
        System.out.println(stu1.getStudentName());
        Address stu1Address = stu1.getAddress();
        System.out.println(stu1Address.getCity());
        System.out.println(stu1Address.getCountry());
        System.out.println(stu1Address.getPincode());

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

```

Output:

```

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

```

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

Constructor Injection:

Constructor injection is a form of dependency injection where dependencies are provided to the object through its constructor. It is a way to ensure that all required dependencies are supplied when creating an object. In constructor injection, the class that requires dependencies has one or more parameters in its constructor that represent the dependencies. When an instance of the class is created, the dependencies are passed as arguments to the constructor. Constructor injection is often considered a best practice in Spring because it helps ensure that the dependencies required for an object to function are provided at the time of its creation. This can lead to more maintainable and testable code.

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

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

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

public class Product {

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

    // All Params Constructor
    public Product(int productId, String productName, double price) {
        super();
        System.out.println("Product All Param's Constructor Executed");
        this.productId = productId;
        this.productName = productName;
        this.price = price;
    }

    public int getProductId() {
        return productId;
    }

    public String getProductName() {
        return productName;
    }

    public double getPrice() {
        return price;
    }

    public void setProductId(int productId) {
        System.out.println("setProductId is called");
        this.productId = productId;
    }

    public void setProductName(String productName) {
        System.out.println("setProductName is called");
    }
}
```

```

        this.productName = productName;
    }
    public void setPrice(double price) {
        System.out.println("setPrice is called");
        this.price = price;
    }
}

```

- Now Configure Bean Data With Constructor Injection.

```

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

```

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

Testing:

```

package com.naresh.spring.di.ci;

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

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

```

Output:

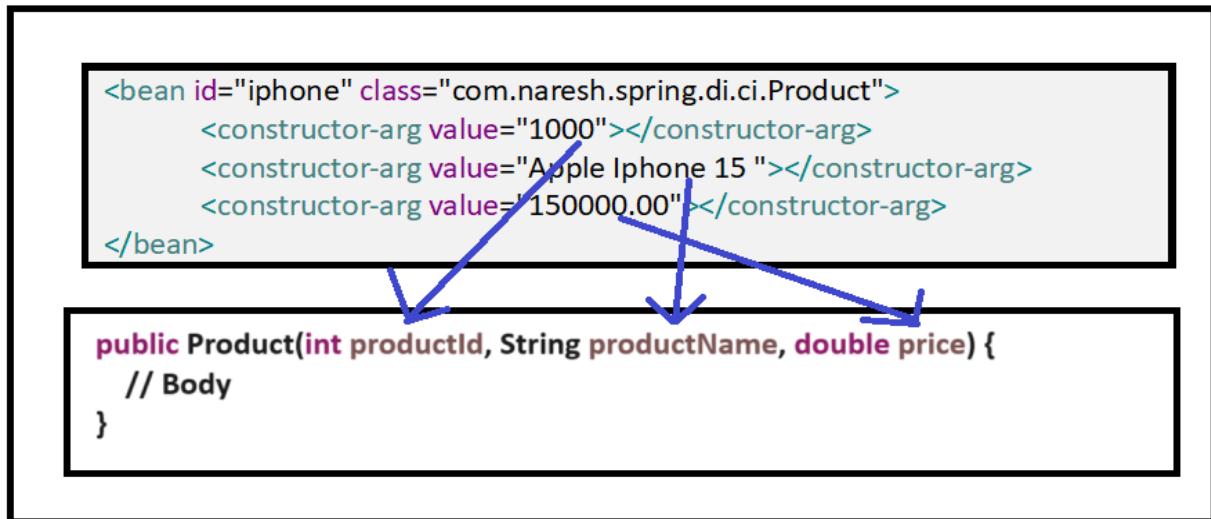
```

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

```

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

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



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

Exception in thread "main"

org.springframework.beans.factory.UnsatisfiedDependencyException:

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

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

```

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

```

```

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

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

```

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

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

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

Exception in thread "main"

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

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

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

- Define AccountDetails Bean class.

```

package com.naresh.hello;

import java.util.Set;

public class AccountDetails {

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

```

```

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

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

- Define Dependency Class Address.

```

package com.naresh.hello;

public class Address {

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

```

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

```

- Define Bean Configuration

```

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

    <bean id="addr" class="com.naresh.hello.Address">
        <property name="flatNo" value="333"/>
        <property name="houseName" value="Lotus Homes"/>
        <property name="mobile" value="9182222222"/>
    </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>+91-8888888888</value>
                <value>+232388888888</value>
            </set>
        </constructor-arg>
        <constructor-arg name="customerAddress" ref="addr" />
    </bean>

```

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

- Now Test Constructor Injection Beans and Configuration.

```
package com.naresh.hello;

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

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

        AccountDetails details = (AccountDetails) context.getBean("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:

- 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.
- Timing of Injection:** Setter injection can be performed after the object is created, allowing for the possibility of injecting dependencies at a later stage. Constructor injection, on the other hand, requires all dependencies to be provided at the time of object creation.

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

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

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

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

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

Bean Wiring in Spring:

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

```

package com.naresh.hello;

public class AreaDeatils {

    private String street;
    private String pincode;

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

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

- Create Another Bean : **Address**

```
package com.naresh.hello;

public class Address {

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

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

- Create Another Bean : **AccountDetails**

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

```
import java.util.Set;

public class AccountDetails {

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

    public AccountDetails() {
    }

    public Address getCustomerAddress() {
        return customerAddress;
    }

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

    public String getName() {
        return name;
    }

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

    public double getBalance() {
        return balance;
    }

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

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

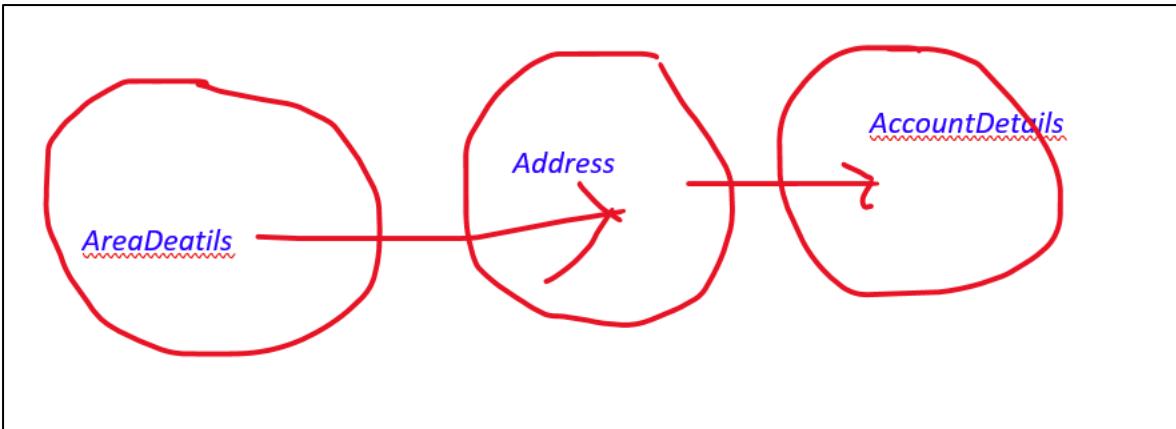
    public void setMobiles(Set<String> mobiles) {
        this.mobiles = mobiles;
    }
}
```

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

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

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

    <bean id="areaDetails" class="com.naresh.hello.AreaDeatils">
        <property name="street" value="Naresh It road"></property>
        <property name="pincode" value="323232"></property>
    </bean>
    <bean id="addr" class="com.naresh.hello.Address">
        <property name="flatNo" value="333"></property>
        <property name="houseName" value="Lotus Homes"></property>
        <property name="mobile" value="9182222222"></property>
        <property name="area" ref="areaDetails"></property>
    </bean>
    <bean id="accountDeatils" class="com.naresh.hello.AccountDetails">
        <constructor-arg name="name" value="Dilip" />
        <constructor-arg name="balance" value="500.00" />
        <constructor-arg name="customerAddress" ref="addr" />
        <constructor-arg name="mobiles">
            <set>
                <value>8826111377</value>
                <value>8826111377</value>
                <value>+91-88888888</value>
                <value>+232388888888</value>
            </set>
        </constructor-arg>
    </bean>
</beans>
```



Testing of Bean Configuration:

```

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

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

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

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

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

```

Output:

```

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

```

AutoWiring in Spring:

Autowiring feature of spring framework enables you to inject the objects dependency implicitly. It internally uses setter or constructor injection. In Spring framework, the “**autowire**” attribute is used in XML <bean> configuration files to enable automatic

dependency injection. It allows Spring to automatically wire dependencies between beans without explicitly specifying them in the XML file.

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

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

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

autowire=no:

For example, Define Product Class.

```
public class Product {  
  
    private String productId;  
    private String productName;  
  
    public String getProductId() {  
        return productId;  
    }  
    public void setProductId(String productId) {  
        this.productId = productId;  
    }  
    public String getProductName() {  
        return productName;  
    }  
}
```

```

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

```

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

```

Package com.flipkart.orders;

import com.flipkart.product.Product;

public class Order {

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

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

```

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

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

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

```
package com.flipkart.main;

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

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

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

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

- Now we got an exception, as shown below.

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

Means, Product and Order Object created by Spring but not injected product bean object automatically in side Order Object by IOC Container with setter injection internally. Because

we used **autowire** mode as **no**. By Default autowire value is “**no**” i.e. Even if we are not given autowire attribute internally Spring Considers it as **autowire=“no”**.

autowire=“byName”:

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

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

Product class Bean ID = Property Name of Order class

```

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

public class Order

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

    // setters , getters, constructors
}

```

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

```

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

```

- Now test our application.

```

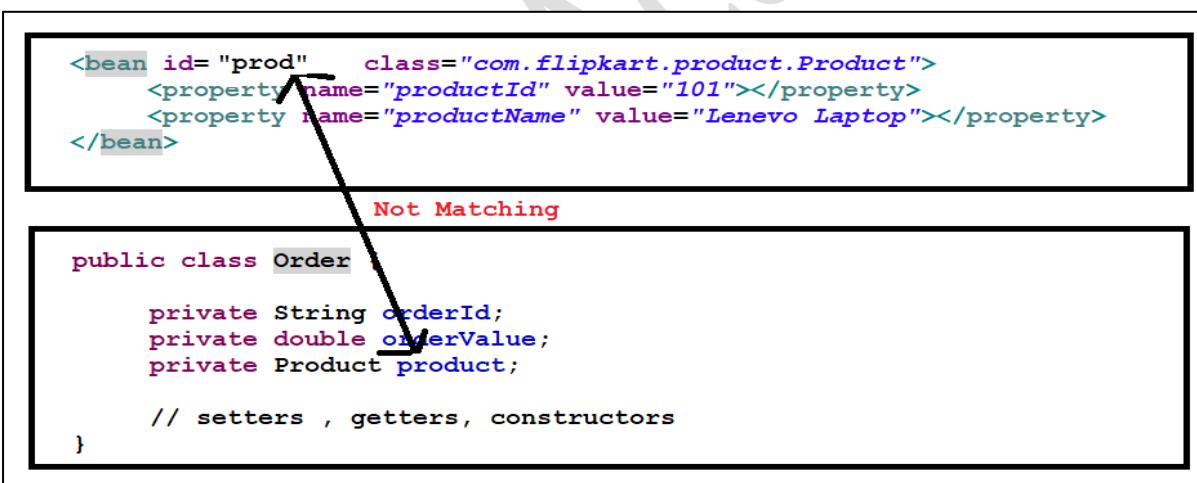
1 package com.flipkart.orders;
2
3 public class OrdersManagement {
4
5     private int noOfOrders;
6     private double totalAmount;
7     private Order order;
8
9     public OrdersManagement(int noOfOrders, double totalAmount, Order order)
10        super();
11        this.noOfOrders = noOfOrders;
12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

```

Console <terminated> AutowiringDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\re\bin\javaw.exe (11-Jul-2023, 1:16:00)
Product Object Created by IOC
Order Object Created by IOC
101

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

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



```

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

```

```

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

    // setters , getters, constructors
}

```

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

```

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

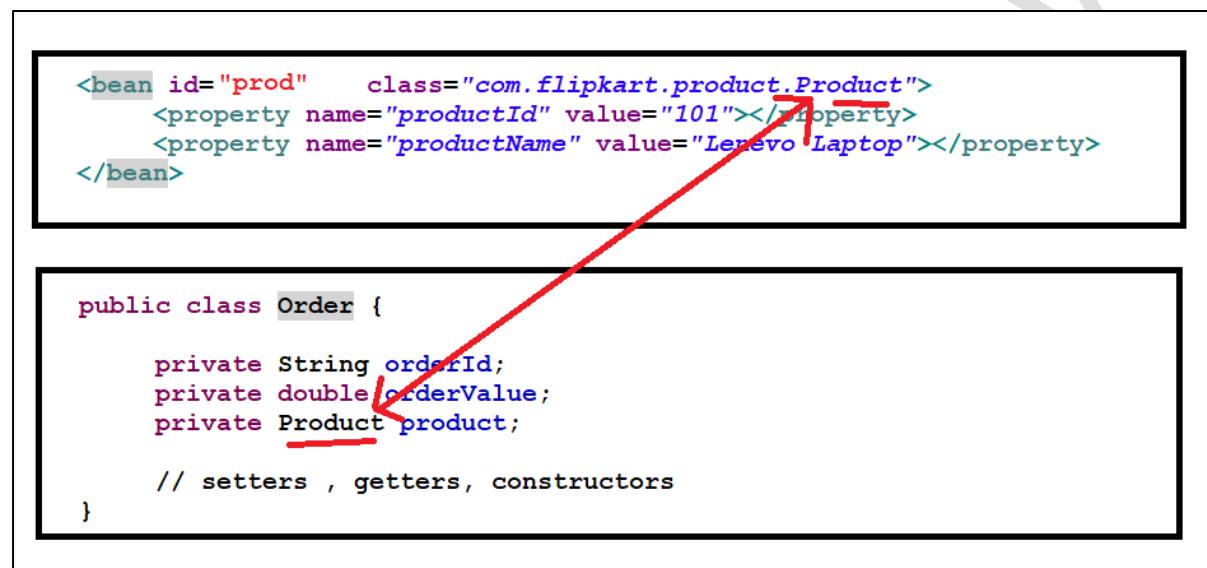
```

autowire="byType":

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

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

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



```

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

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

    // setters , getters, constructors
}

```

- **Beans Configuration:**

```

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

```

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

```

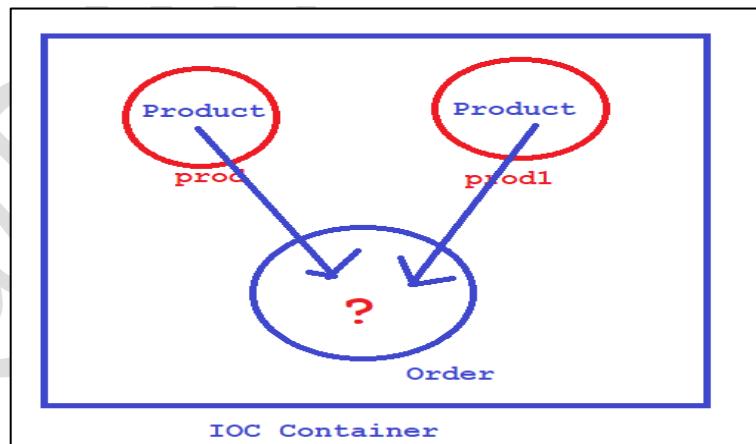
3* import org.springframework.context.ApplicationContext;
4
5
6 public class AutowiringDemo {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new FileSystemXmlApplicationContext(
10             "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");
11
12         Order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

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

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

```

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



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

```

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

```

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

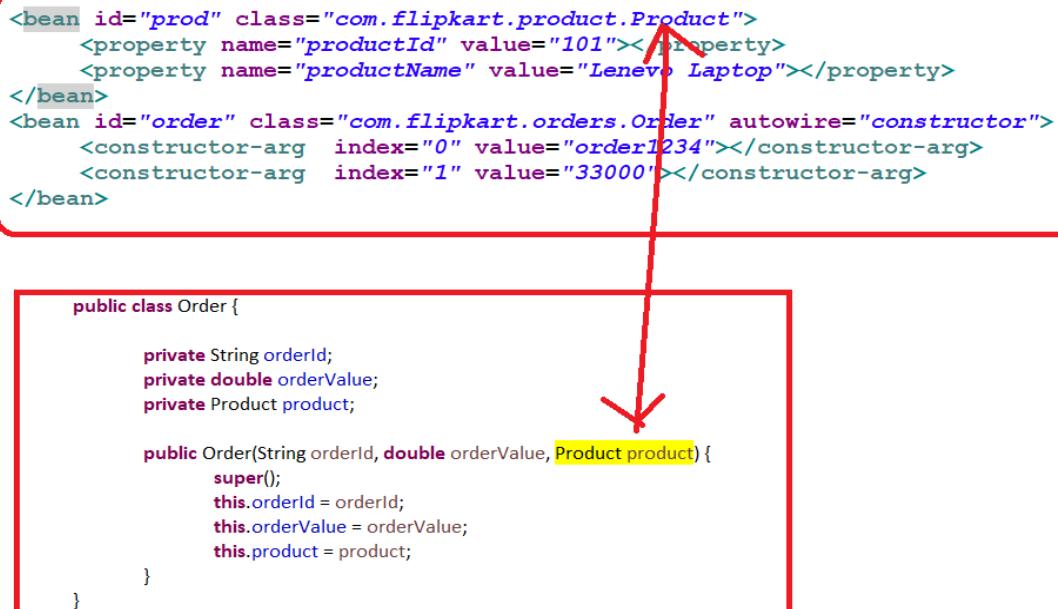
```

autowire="constructor":

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

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

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



Beans Configuration:

```

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

```

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

```

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

```

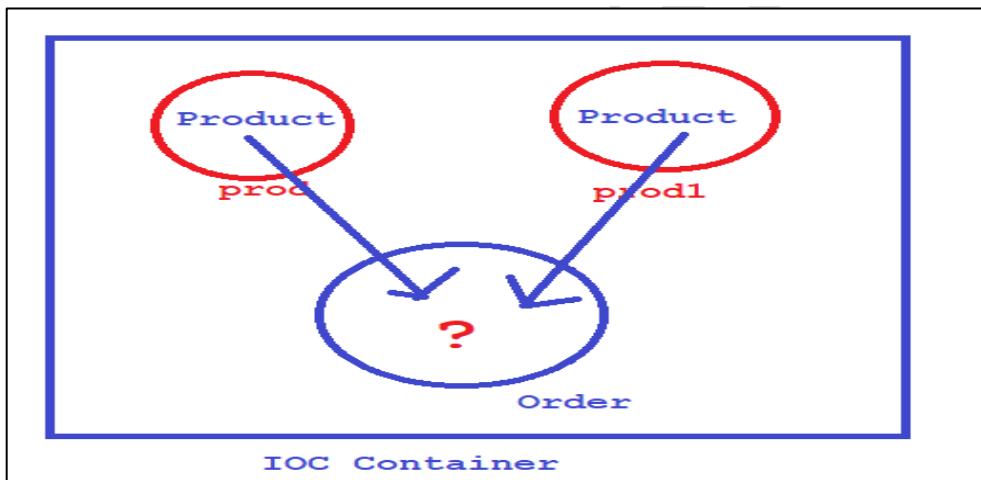
Console ×
 <terminated> AutowiringDemo [Java Application] D:\softwares\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230404-1729\bin\javaw.exe (11-Jul-20)
 Product Object Created by IOC
 Order Object Created: With Params Constructor
 101

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

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

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

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



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

```
Product Object Created by IOC
Product Object Created by IOC
Jul 11, 2023 1:56:23 PM org.springframework.context.support.AbstractApplicationContext
refresh
WARNING: Exception encountered during context initialization - cancelling refresh
attempt: org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'order' defined in file [D:\workspaces\naresit\bean-
wiring\beans.xml]: Unsatisfied dependency expressed through bean property 'product';
nested
exception
is
org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying
bean of type 'com.flipkart.product.Product' available: expected single matching bean but
```

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

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

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

- **Test Our Application :**

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

Console X |      

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

Advantage of Autowiring:

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

Disadvantages of Autowiring:

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

Bean Scopes in Spring:

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

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

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

Defining Scope of beans syntax:

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

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

singleton:

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

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

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

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

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

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

Product.java

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

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

    //setters and getters methods
}
```

- Now configure above class in side beans xml file.

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

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

```
public class BeanScopesDemo {

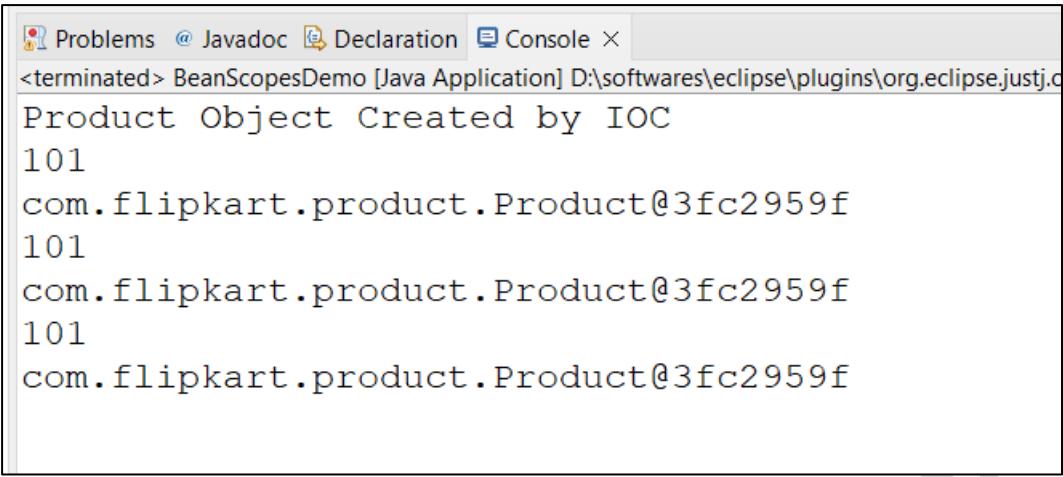
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
            "D:\\beans.xml");

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

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

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

Output:



```

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

```

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

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

Create one more bean configuration of Product.

```

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

```

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

```

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

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

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

```

```

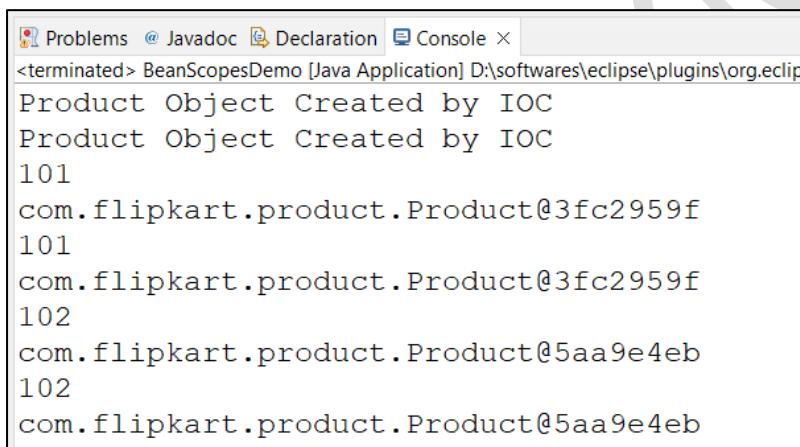
// 2. Requesting/Calling IOC Container for Product Object
Product p2 = (Product) context.getBean("product");
System.out.println(p2.getProductId());
System.out.println(p2);

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

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

}

```

Output:


```

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

```

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

prototype:

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

Bean class: Product.java

```

public class Product {

    private String productId;
    private String productName;

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

```

```

    }

    //setters and getters
}

```

XML Bean configuration:

```

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

```

Testing :

```

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

```

Output:

```

Problems @ Javadoc Declaration Console x
<terminated> BeanScopesDemo [Java Application] D:\\softwares\\eclipse\\plugins\\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\\jre\\bin\\javaw.exe (12)
Product Object Created by IOC
com.flipkart.product.Product@5542c4ed
101
Product Object Created by IOC
com.flipkart.product.Product@1573f9fc
101

```

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

request:

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

session:

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

application:

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

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

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

websocket:

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

Java/Annotation Based Beans Configuration:

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

@Configuration & @Bean Annotations:

@Configuration:

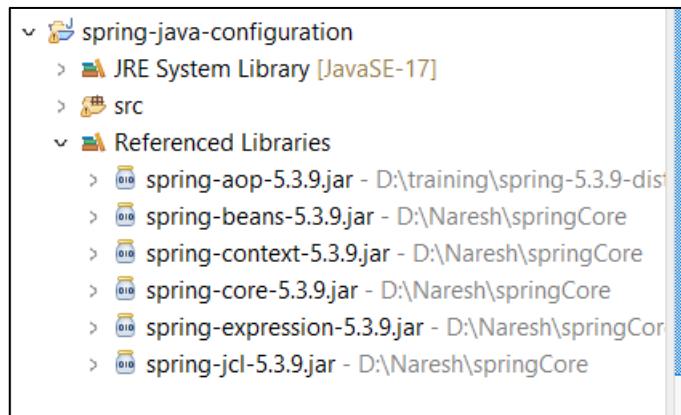
In Java Spring, the **@Configuration** annotation is used to indicate that this class is a configuration class of Beans. A configuration class is responsible for defining beans and their dependencies in the Spring application context. Beans are objects that are managed by the Spring IOC container. Annotating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

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

@Bean:

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

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



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

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

```
package com.amazon.users;

public class UserDetails {

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

    //setters and getters
}
```

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

Configuration class would be as follows:

```
package com.amazon.config;

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

@Configuration
public class BeansConfiguration {
    @Bean("userDetails")
```

```
UserDetails getUserDetails() {
    return new UserDetails();
}
```

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

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

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

```
package com.amazon;

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

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

        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(BeansConfiguration.class);

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

Output: Prints hash code of Object

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

Understand From above code :

AnnotationConfigApplicationContext:

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

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

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

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

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

Multiple Configuration Classes and Beans:

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

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

```
package com.amazon.products;

public class ProductDetails {

    private String pname;
    private double price;

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

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

```
package com.amazon.config;

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

public class BeansConfigurationTwo {

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

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

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

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context
            = new AnnotationConfigApplicationContext();

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

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

Now Crate multiple Bean Configurations for same Bean class.

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

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

public class BeansConfigurationTwo {

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

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

- Now get Both bean Objects of ProductDeatils.

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

public class SpringBeanMainApp {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new
        AnnotationConfigApplicationContext();

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

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

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

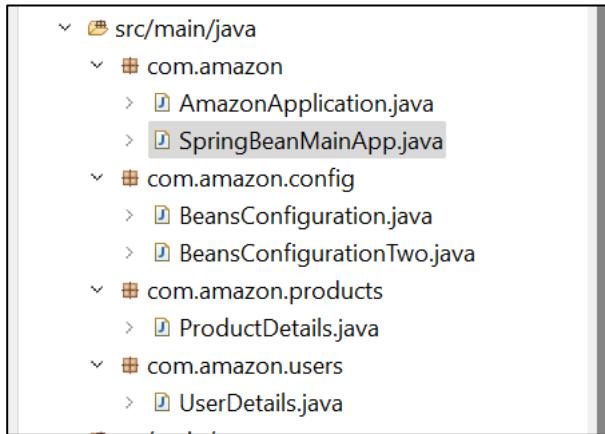
        ProductDetails productTwo = (ProductDetails) context.getBean("productDetailsTwo");
        System.out.println(productTwo);
        context.close();
    }
}
```

{}

Output:

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

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

Above Project Files Structure:**@Component Annotation :**

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

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

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

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

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

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

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

Define Spring Components :

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

```
package com.tek.teacher;

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

@Component
public class Product {

    private String pname;
    private double price;

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

- Now Test in Main Class.

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

public class SpringComponentDemo {
```

```

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

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

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

```

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

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

@ComponentScan:

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

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

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

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

```

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

```

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

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

Test our component class:

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

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

}

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

```
package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {

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

- Now Run your Main class application.

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

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

Now Add One More @Component class:

```
package com.tek.teacher;

import org.springframework.stereotype.Component;

@Component
public class UserDetails {
```

```

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

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

```

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

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

```

```

public class SpringComponentDemo {

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

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

```

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

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

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

Can we Pass Bean Scope to @Component Classes?

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

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

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

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

```

```

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

```

Output:

```

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

```

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

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

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

Inside Configuration Class:

```

package com.tek.teacher;

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

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

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

```

➤ Testing from Main Class:

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

```

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

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

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

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

Inside Configuration Class:

```
package com.tek.teacher;

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

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

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

Main App:

```
package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

        // UserDetails Component
    }
}
```

```

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

```

Output:

dilip@gmail.com
8826111377

Defining Scope of beans with Annotations:

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

```

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

```

Annotation Based Scope Configuration:

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

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

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

```

package com.amazon.products;

public class ProductDetails {
    private String pname;
    private double price;
    public String getPname() {
        return pname;
    }
    public void setPname(String pname) {
        this.pname = pname;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public void printProductDetails() {
    }
}

```

```
        System.out.println("Product Details Are : .....");
    }
}
```

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

Singleton Scope:

A single Bean object instance created and returns same Bean instance for each Spring IoC container call i.e. **getBean()**. 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 **ProductDetails** Object is singleton or not. Request multiple times **ProductDetails** Object from Spring Container by passing bean id **productDetails**.

```
package com.amazon;

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

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

```

        System.out.println(productTwo);
        context.close();
    }
}

```

Output:

```

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

```

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

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

```

package com.amazon.config;

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

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

```

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

```

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

public class SpringBeanScopeTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfigurationThree.class);
    }
}

```

```

        context.refresh();
        //Prototype Beans:
        ProductDetails productThree =
            (ProductDetails) context.getBean("productTwoDetails");
        System.out.println(productThree);
        ProductDetails productFour =
            (ProductDetails) context.getBean("productTwoDetails");
        System.out.println(productFour);
        context.close();
    }
}

```

Output:

```

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

```

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

Scope of @Component classes:

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

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

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

```

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

```

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

```

package com.tek.teacher.products;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

```

```

AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext();
context.register(BeansConfiguration.class);
context.scan("com.tek.*");
context.refresh();

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

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

```

Output:

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

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

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

Auto Wiring In Spring

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

Auto wiring will be achieved in multiple ways/modes.

Auto Wiring Modes:

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

- constructor

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

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

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

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

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

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

@Autowired on Properties

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

First, Let's Define a bean : Address.

```
package com.dilip.account;

import org.springframework.stereotype.Component;

@Component
public class Address {

    private String streetName;
    private int pincode;

    public String getStreetName() {
        return streetName;
    }
}
```

```

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

```

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

```

package com.dilip.account;

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

@Component
public class Account {

    private String name;
    private long accNumber;

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

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

```

```

        return addr;
    }
    public void setAddr(Address addr) {
        this.addr = addr;
    }
}

```

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

```

package com.dilip.account;

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

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

```

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

```

package com.dilip.account;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

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

        System.out.println(address);
    }
}

```

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

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

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

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

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

import org.springframework.stereotype.Component;

@Component("home")
public class Addresss {

    private String streetName;
    private int pincode;

    public String getStreetName() {
        return streetName;
    }

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

    public int getPincode() {
        return pincode;
    }

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

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

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

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

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

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

    @Bean("hyd")
    Addresss createAddress() {
        Addresss a = new Addresss();
        a.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 getAddress() {
        return add;
    }

    public void setAddress(Addresss add) {
```

```
    this.add = add;
}
}
```

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

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

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AutowiringTestMainApp {

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

We got an exception now as follows,

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

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

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

@Qualifier:

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

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

Now add **@Qualifier** on **Address** field, 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 empemployee = (Employee) context.getBean("emp");
    Addresss empAdd = empemployee.getAdd();
    System.out.println(empAdd.getPincode());
    System.out.println(empAdd.getStreetName());
}
}

```

Output: 500067
Gachibowli

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

@Primary:

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

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

```

package com.hello.spring.boot.employees;

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

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

```

```
Addresss bangloreAddress() {
    Addresss a = new Addresss();
    a.setPincode(560043);
    a.setStreetName("Banglore");
    return a;
}
```

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

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

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

@Component("emp")
public class Employee {

    private String lastName;
    private long mobile;

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

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

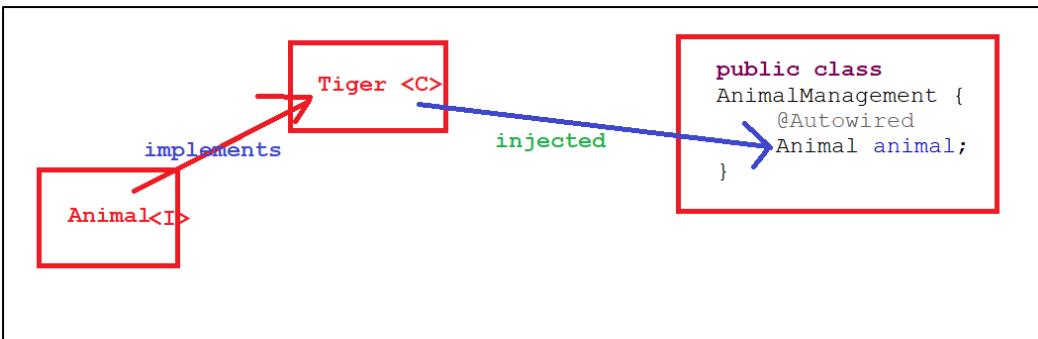
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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;

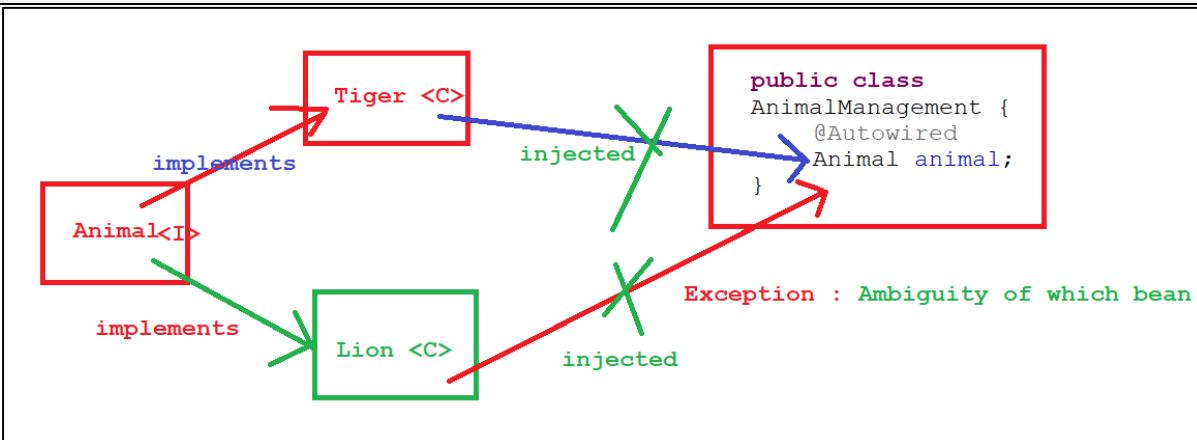
public class AutoWringDemo {
    public static void main(String[] args) {

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

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

We got an Exception as,

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



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

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

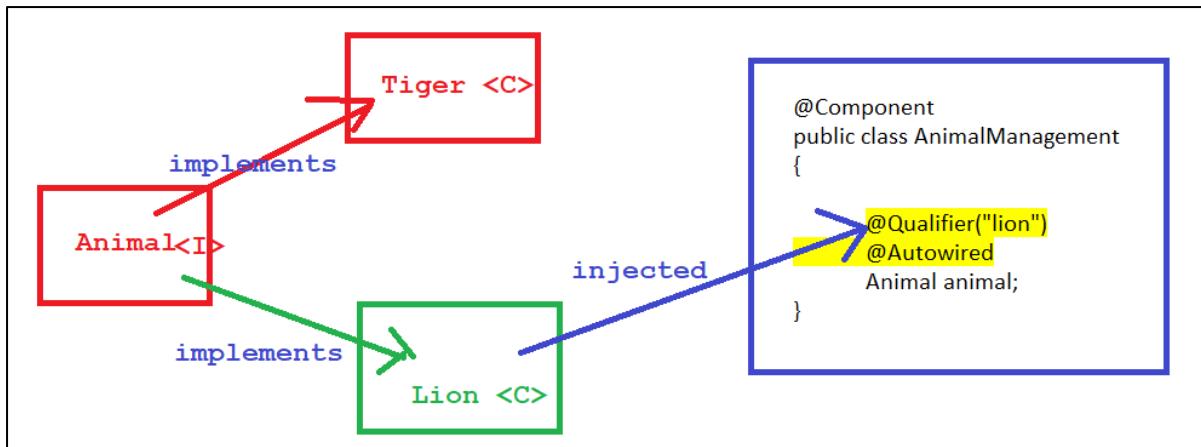
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

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

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

Run again now **AutoWringDemo.java**

Output : I am a Lion.



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

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

```

package com.dilip.auto.wiring;

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

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

Run again now **AutoWringDemo.java**

Output : I am a Tiger

Types of Dependency Injection with Annotations :

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

We can inject dependent objects in three ways, using:

Spring Framework supporting 3 types of Dependency Injection .

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

Filed Injection:

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

Requirement : Address is Dependency of Employee class.

Address.java : Create as Component class

```
package com.dilip.spring;

import org.springframework.stereotype.Component;

@Component
public class Address {

    private String city;
    private int pincode;

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

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

```
package com.dilip.spring;

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

@Component
```

```

public class Employee {

    private String empName;
    private double salary;

    //Field Injection
    @Autowired
    private Address address;

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

    public String getEmpName() {
        return empName;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        System.out.println("This is 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

Setter Injection Overview:

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

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

Interface : **MessageService.java**

```

package com.dilip.setter.injection;

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

```

Impl. Class : **EmailService.java**

```

package com.dilip.setter.injection;

import org.springframework.stereotype.Component;

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

```

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

Impl. Class : **SMSService.java**

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

import org.springframework.stereotype.Component;

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

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

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

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

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

@Component
public class MessageSender {

    private MessageService messageService;

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

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

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

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

```

package com.dilip.setter.injection;

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

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

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

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

```

Output:

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

Injecting Multiple Dependencies using Setter Injection:

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

```

package com.dilip.setter.injection;

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

@Component
public class MessageSender {

    private MessageService messageService;
    private MessageService smsService;

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

```

```

    }

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

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

```

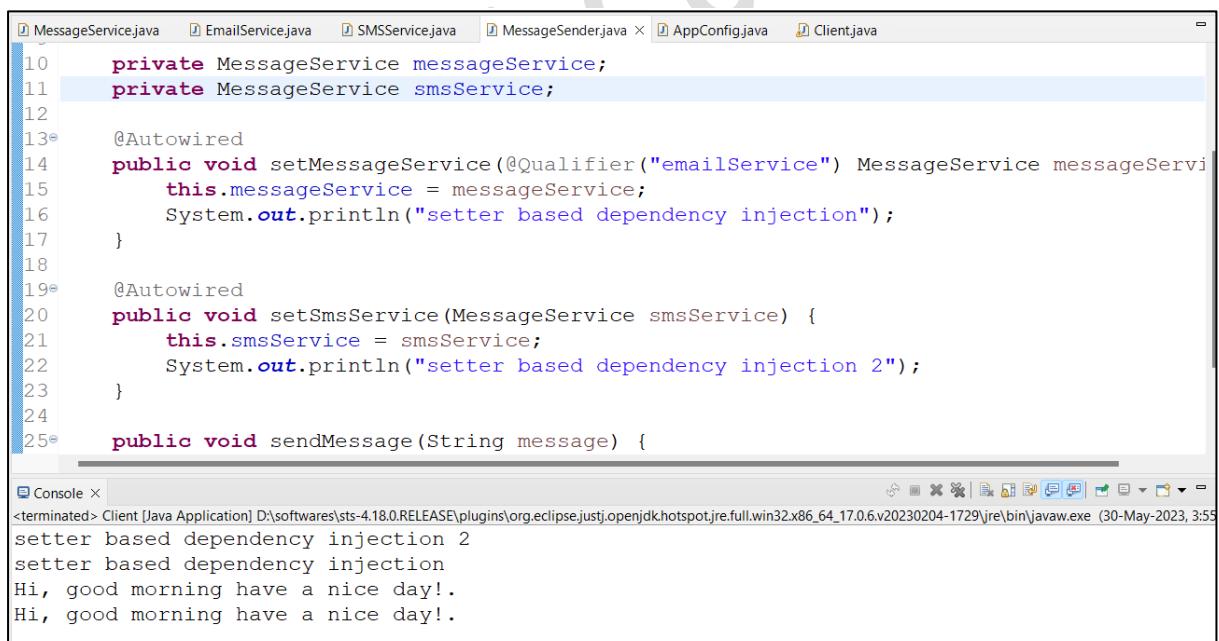
➤ Now Run **Client.java**, One more time to see both Bean Objects injected or not.

Output:

```

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

```



The screenshot shows the Eclipse IDE interface with several Java files listed in the top tab bar: MessageService.java, EmailService.java, SMSService.java, MessageSender.java, AppConfig.java, and Client.java. The Client.java file is open and contains the following code:

```

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

```

In the bottom right corner of the IDE, there is a 'Console' tab. The console output window shows the following text:

```

<terminated> Client [Java Application] 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 (30-May-2023, 3:55
setter based dependency injection 2
setter based dependency injection
Hi, good morning have a nice day!.
Hi, good morning have a nice day!.

```

Constructor Injection:

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

➤ **MessageService.java**

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

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

➤ **EmailService.java**

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

import org.springframework.stereotype.Component;

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

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

➤ **SMSService.java**

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

import org.springframework.stereotype.Component;

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

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

➤ **MessageSender.java**

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

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

```

import org.springframework.stereotype.Component;

@Component
public class MessageSender {

    private MessageService messageService;

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

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

```

➤ Now create a Configuration class: AppConfig.java

```

package com.dilip.setter.injection;

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

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

```

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

```

package com.dilip.setter.injection;

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

public class Client {

    public static void main(String[] args) {

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

        ApplicationContext applicationContext =
            new AnnotationConfigApplicationContext(AppConfig.class);
    }
}

```

```

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

```

Output:

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

How to Declare Types of Autowiring with Annotations?

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

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

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

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

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

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

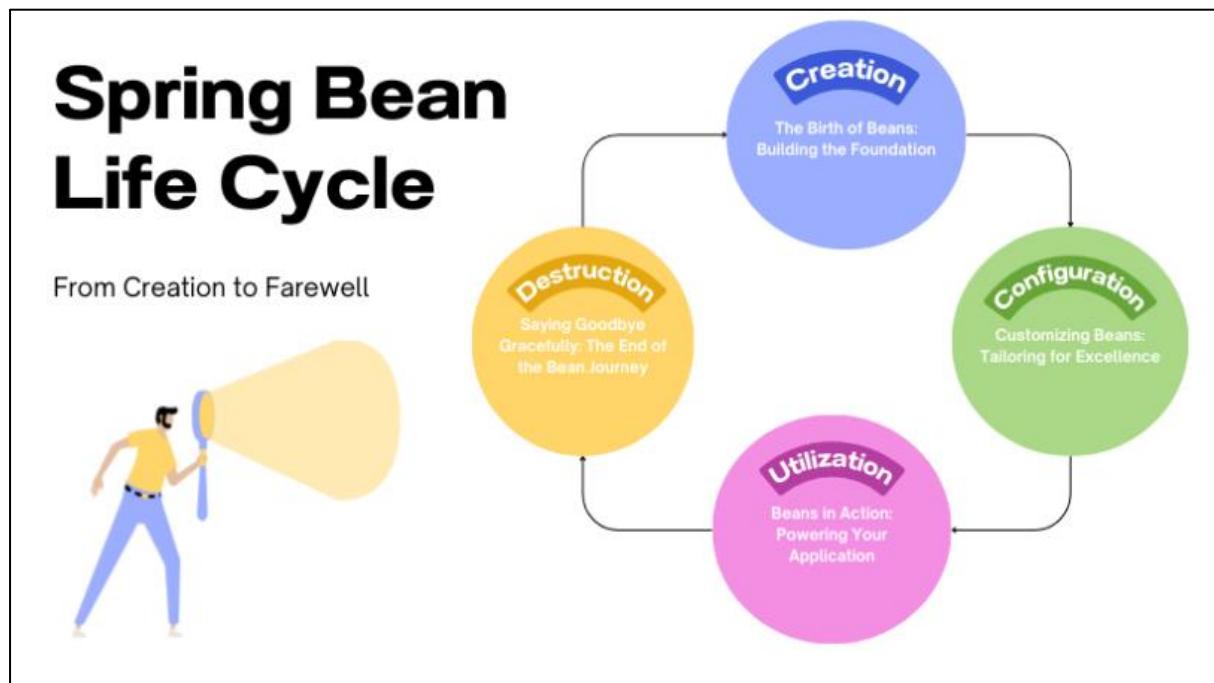
byName: If we define **@Autowired** on field/setter/constructor level along with **@Qualifeir** then, Spring injecting Dependency Object in side composite Object specific to Bean ID.

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.

Bean life cycle in Java Spring:

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



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

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

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

Benefits of Exploring the Spring Bean Life Cycle:

- **Resource Management:** As you traverse the life cycle stages of Bean Object, you're in control of resources. This translates to efficient memory utilization and prevents resource leaks, ensuring your application runs like a well configured machine.
- **Customization:** By walking through the life cycle stages, you can inject custom logic at strategic points. This customization allows your beans to adapt to specific requirements, setting the stage for a flexible and responsive application.

- **Dependency Injection:** Understanding the stages of bean initialization also resolves the magic of dependency injection. You'll learn how beans communicate, collaborate, and share information, building a cohesive application architecture.
- **Debugging:** With a firm grasp of the life cycle, troubleshooting becomes very easy. By tracing a bean's journey through each stage, you can pinpoint issues and enhance the overall stability of your application.

Defining Bean Life Cycle Methods:

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

InitializingBean:

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

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

DisposableBean:

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

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

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

Defining Bean Life Cycle Methods with Beans:

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

```
package com.dilip;

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

@Component
```

```

public class Customer implements InitializingBean, DisposableBean {

    private int id;
    private String name;

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

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

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

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

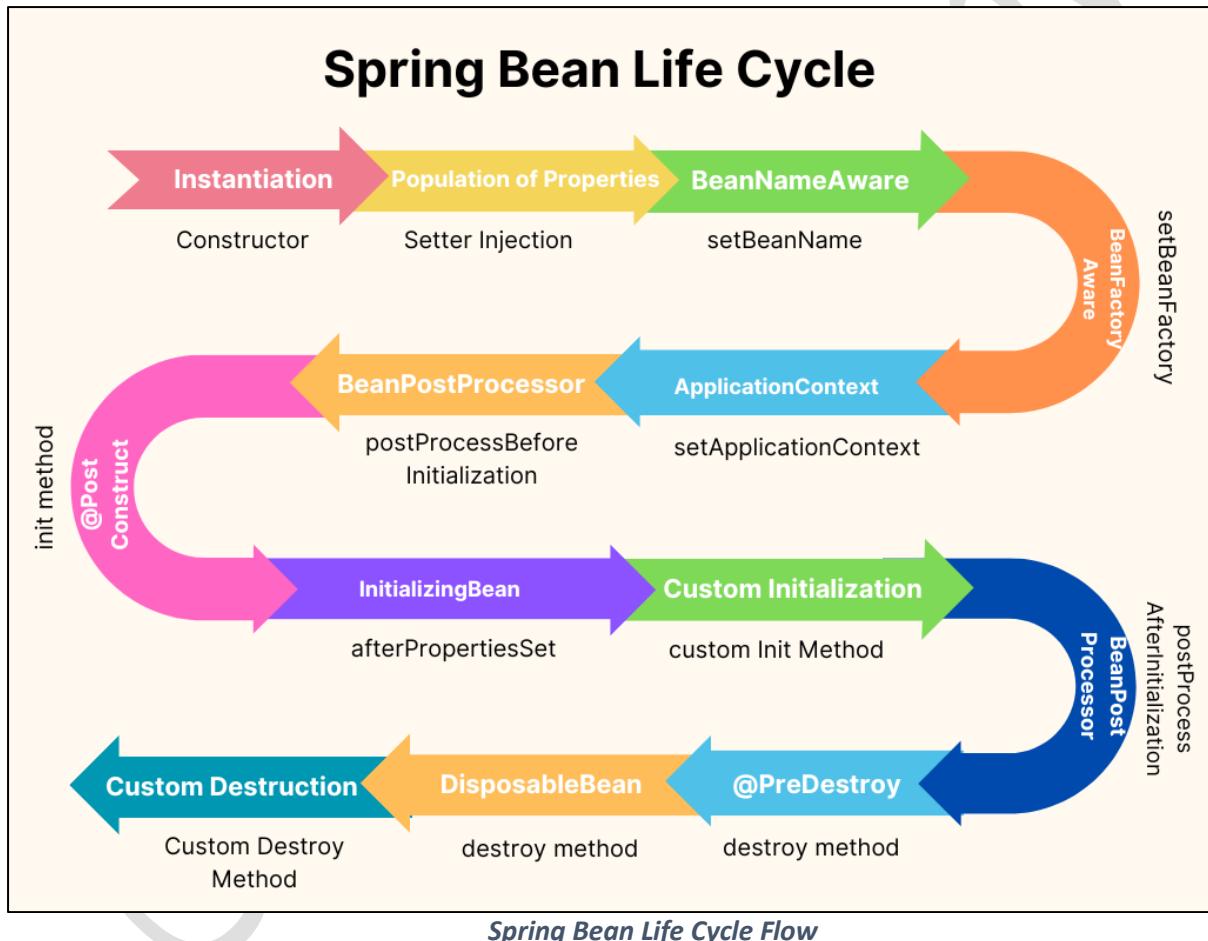
```

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

- Now get this Bean Object from IOC container.
- Once we created or initialized Spring Container, Container starts the process of Instantiating Bean Objects.
- Bean Object will be created/instantiated
- After Bean Object creation , Properties Values will be injected if any available.
- All Dependencies will be identified and injected

- Now Bean Initialization method i.e. **afterPropertiesSet()** method logic will be executed by IOC container one time i.e. this method will be executed once anew Object is created always by container.
- Now Bean Object is ready with all configuration values of properties and initialization values.
- We will always get current object always when we get it from container always.
- After utilization of Bean Object, when the Spring container is shutting down or removing the bean, it will call the **destroy()** method automatically for every Bean Object, allowing you to perform any necessary cleanup tasks written as part of the method.
- After executing all bean Objects **destroy()** methods then finally container got closed.

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



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

```

package com.dilip;

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

```

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

- Now Pass above Configuration class to container.

```
package com.dilip;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

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

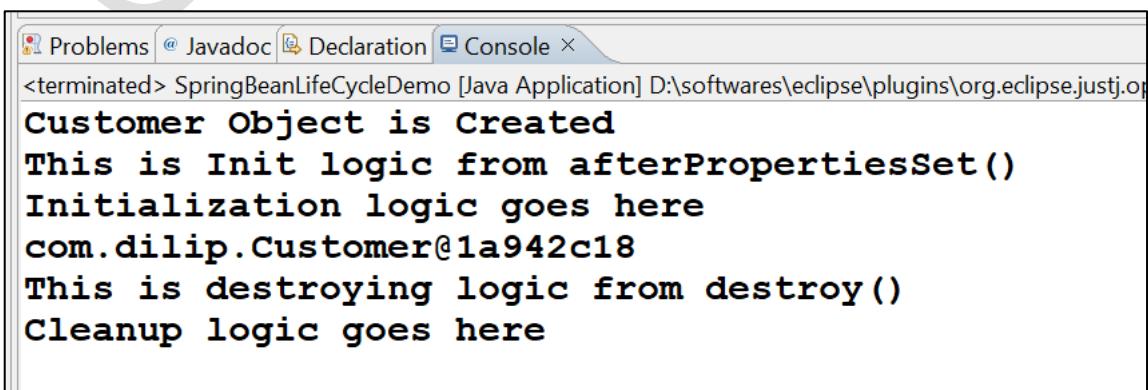
        // Providing Bean Classes information to Container
        context.register(BeansConfiguration.class);

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

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

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

Output:



The screenshot shows the Eclipse IDE's Console tab with the following output:

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

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

BeansConfiguration.java

```
package com.dilip;

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

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

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

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

```
package com.dilip;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

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

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

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

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

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

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

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

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

Define these annotations on custom methods of Bean instantiation and destroying logic with out using any Predefined Interfaces from Spring FW.

@PostConstruct:

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

@PreDestroy:

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

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

Example: Bean Class: Customer.java

```
package com.dilip;

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

@Component
public class Customer {

    private int id;
    private String name;

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

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

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

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

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

```
package com.dilip;

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

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

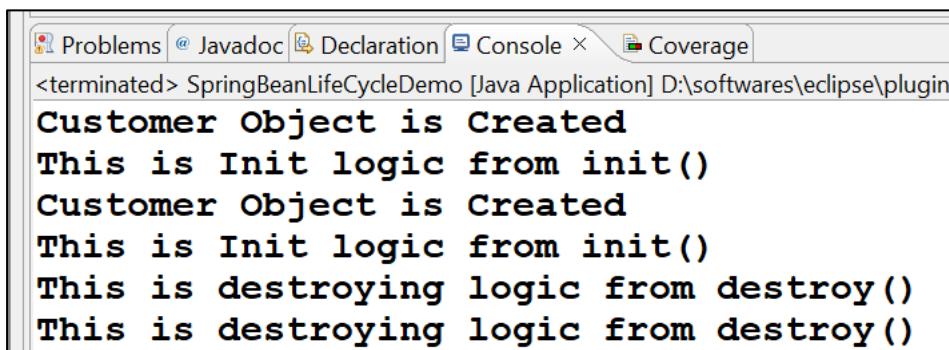
- Now Execute container creation and closing Lofigc again and observe initialization and destroy methods executed 2 times for 2 Customer Bean Objects creation.

```
package com.dilip;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

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



```
Customer Object is Created
This is Init logic from init()
Customer Object is Created
This is Init logic from init()
This is destroying logic from destroy()
This is destroying logic from destroy()
```

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

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

Bean class: Student.java

```
package com.dilip;

public class Student {

    private int sid;

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

In XML Configuration :

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

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

<beans>
    <bean id="student" class="com.dilip.Student"
```

```
        init-method="beanInitialization"  
        destroy-method="beanDestruction">  
  
    </bean>  
</beans>
```

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

In Annotation based Configuration :

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

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

Question: Why Understand the Spring Bean Life Cycle?

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

Spring JDBC Module

@DilipItAcademy

Spring JDBC:

Spring JDBC is a part of the Spring Framework, which is a popular Java-based application framework used for building enterprise-level applications. Spring JDBC is a framework that provides an abstraction layer on top of JDBC. This makes it easier to write JDBC code and reduces the amount of boilerplate code that needs to be written. Spring JDBC also provides a number of features that make it easier to manage database connections, handle transactions, and execute queries.

Advantages of using Spring JDBC:

Reduced boilerplate code: Spring JDBC provides a number of classes and interfaces that can be used to simplify JDBC code. This can save a lot of time and effort, especially for complex queries.

Simplified transaction management: Spring JDBC provides a simple API for managing transactions. This makes it easier to ensure that database operations are performed atomically.

Improved exception handling: Spring JDBC converts JDBC exceptions into Spring's own Runtime Exceptions. This makes it easier to handle exceptions in a consistent way.

Here are some key features and concepts of Spring JDBC:

DataSource: DataSource is a JDBC object that represents a connection to a database. Spring provides a number of data source implementations, such as DriverManagerDataSource.

JdbcTemplate: The `org.springframework.jdbc.core.JdbcTemplate` is a central class in Spring JDBC that simplifies database operations. It encapsulates the common JDBC operations like executing queries, updates, and stored procedures. It handles resource management, exception handling, and result set processing.

RowMapper: A RowMapper is an interface used to map rows from a database result set to Java objects. It defines a method to convert a row into an object of a specific class.

Transaction management: Spring provides built-in transaction management capabilities through declarative or programmatic approaches. You can easily define transaction boundaries and have fine-grained control over transaction behaviour with Spring JDBC.

Overall, Spring JDBC is a powerful framework that can make it easier to write JDBC code. However, it is important to be aware of the limitations of Spring JDBC before using it. Using Spring JDBC, you can perform typical CRUD (Create, Read, Update, Delete) operations on databases without dealing with the boilerplate code typically required in JDBC programming.

Here are some of the basic steps involved in using Spring JDBC:

- Create a DataSource.
- Create a JdbcTemplate.
- Execute a JDBC query.
- Handle exceptions.

Let's see few methods of spring JdbcTemplate class.

int update(String query)	is used to insert, update and delete records.
int update(String query, Object... args)	is used to insert, update and delete records using PreparedStatement using given arguments.
void execute(String query)	is used to execute DDL query.
List query(String query, RowMapper rm)	is used to fetch records using RowMapper.

Similarly Spring JDBC module provided many predefined class and methods to perform all database operations whatever we can do with JDBC API.

Create a Project with Spring JDBC module Functionalities: Perform below Operations on Student Table.

- Insert Data
- Update Data
- Select Data
- Delete Data

NOTE: Please be ready with Database table before writing logic.

Table Creation : **create table student(sid number(10), name varchar2(50), age number(3));**

Please add Specific jar files which are required for JDBC module, as followed.

Project: Create Java Maven Project with Dependencies:

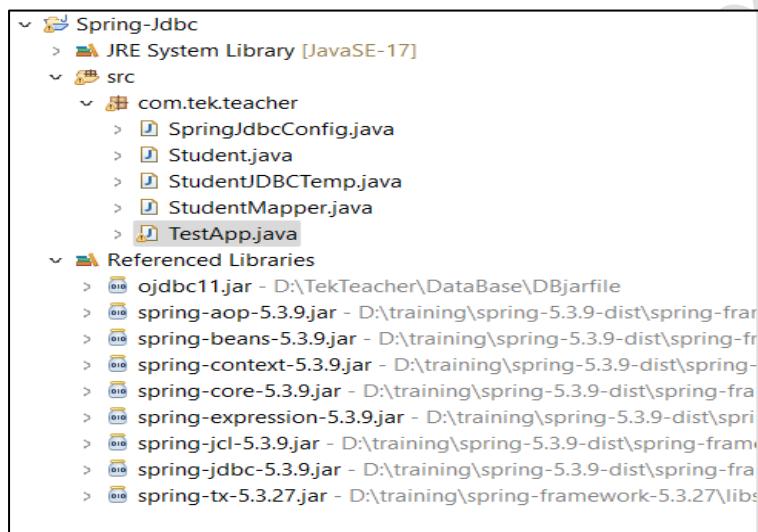
```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>6.0.13</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.0.13</version>
    </dependency>
    <dependency>
```

```

<groupId>org.springframework</groupId>
<artifactId>spring-jdbc</artifactId>
<version>6.0.13</version>
</dependency>
<dependency>
<groupId>com.oracle.database.jdbc</groupId>
<artifactId>ojdbc8</artifactId>
<version>21.9.0.0</version>
</dependency>
</dependencies>

```

Please find Project Structure for List of Jar files Required:



- Please Create Configuration class for Configuring JdbcTemplate Bean Object with DataSource Properties. So Let's start with some simple configuration of the data source.
- We are using Oracle database:
- The DriverManagerDataSource is used to contain the information about the database such as driver class name, connection URL, username and password.

SpringJdbcConfig.java

```

package com.tek.teacher;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
@ComponentScan("com.tek.teacher")
public class SpringJdbcConfig {

```

```

    @Bean
    public JdbcTemplate getJdbcTemplate() {

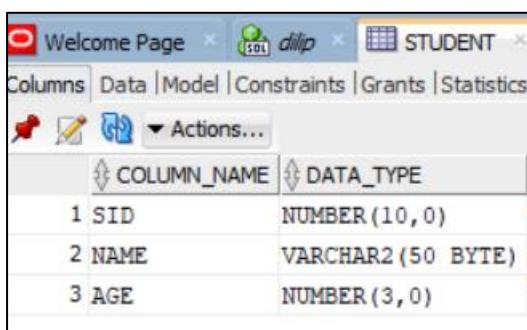
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
        dataSource.setUsername("c##dilip");
        dataSource.setPassword("dilip");

        return new JdbcTemplate(dataSource);
    }
}

```

- Create a POJO class of Student which should be aligned to database table columns and data types.

Table : Student



COLUMN_NAME	DATA_TYPE
1 SID	NUMBER (10, 0)
2 NAME	VARCHAR2 (50 BYTE)
3 AGE	NUMBER (3, 0)

- Student.java POJO class:

```

package com.tek.teacher;

public class Student {

    // Class data members
    private Integer age;
    private String name;
    private Integer sid;

    // Setters and Getters
    public void setAge(Integer age) {
        this.age = age;
    }
    public Integer getAge() {
        return age;
    }
    public void setName(String name) {

```

```

        this.name = name;
    }
    public String getName() {
        return name;
    }
    public Integer getSid() {
        return sid;
    }
    public void setSid(Integer sid) {
        this.sid = sid;
    }
    @Override
    public String toString() {
        return "Student [age=" + age + ", name=" + name + ", sid=" + sid + "]";
    }
}

```

Mapping Query Results to Java Object:

Another very useful feature is the ability to map query results to Java objects by implementing the **RowMapper** interface i.e. when we execute select query's, we will get ResultSet Object with many records of database table. So if we want to convert every row as a single Object then this row mapper will be used. For every row returned by the query, Spring uses the row mapper to populate the java bean object.

A **RowMapper** is an interface in Spring JDBC that is used to map a row from a **ResultSet** to an object. The **RowMapper** interface has a single method, **mapRow()**, which takes a **ResultSet** and a row number as input and returns an object.

The **mapRow()** method is called for each row in the **ResultSet**. The RowMapper implementation is responsible for extracting the data from the **ResultSet** and creating the corresponding object. The object can be any type of object, but it is typically a POJO (Plain Old Java Object).

StudentMapper.java

```

package com.tek.teacher;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student>{

    @Override
    public Student mapRow(ResultSet rs, int arg1) throws SQLException {
        Student student = new Student();
        student.setSid(rs.getInt("sid"));
        student.setName(rs.getString("name"));
    }
}

```

```

        student.setAge(rs.getInt("age"));
        return student;
    }
}

```

- Write a class to perform all DB operation i.e. execution of Database Queries based on our requirement. As part of this class we will use Spring JdbcTemplate Object, and methods to execute database queries.

➤ StudentJDBCTemp.java

```

package com.tek.teacher;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class StudentJDBCTemp {

    @Autowired
    private JdbcTemplate jdbcTemplateObject;

    public List<Student> listStudents() {
        // Custom SQL query
        String query = "select * from student";
        List<Student> students = jdbcTemplateObject.query(query, new
        StudentMapper());

        return students;
    }

    //@Override
    public int addStudent(Student student) {
        String query =
            "insert into student
             values("+student.getId()+","+student.getName()+","+student.getAge()+")";

        System.out.println(query);
        return jdbcTemplateObject.update(query);
    }
}

```

- Write class for testing all our functionalities.

TestApp.java

```
package com.tek.teacher;

import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class TestApp {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new
        AnnotationConfigApplicationContext();
        context.scan("com.tek.*");
        context.refresh();

        StudentJDBCTemp template = context.getBean(StudentJDBCTemp.class);

        //Insertion of Data
        Student s = new Student();
        s.setAge(30);
        s.setName("tek");
        s.setSid(2);

        int count = template.addStudent(s);
        System.out.println(count);

        // Load all Students
        List<Student> students = template.listStudents();
        students.stream().forEach(System.out::println);
    }
}
```

Spring JPA Module

Spring Framework JPA is a powerful abstraction layer that makes it easy to interact with relational databases in Java applications. It provides a simple and consistent API for performing CRUD operations, as well as more advanced features such as pagination, auditing, and querying.

Spring Framework JPA is built on top of the Java Persistence API (JPA), which is a standard API for accessing relational databases. This means that Spring Framework JPA is compatible with a wide range of databases, including MySQL, PostgreSQL, Oracle, and SQL Server.

What is JPA?

JPA(Jakarta Persistence API) is just a specification that facilitates object-relational mapping to manage relational data in Java applications. It provides a platform to work directly with objects instead of using SQL statements.

Jakarta Persistence API (JPA; formerly Java Persistence API) is a Jakarta EE application programming interface specification that describes the management of relational data in enterprise Java applications. The API itself, defined in the **jakarta.persistence** package (**javax.persistence for Jakarta EE 8 and below**). The Jakarta Persistence Query Language (JPQL; formerly Java Persistence Query Language)

History:

- JPA 1.0 specification was 11 May 2006 as part of Java Community.
- JPA 2.0 specification was released 10 December 2009
- JPA 2.1 specification was released 22 April 2013
- JPA 2.2 specification was released in the summer of 2017.
- JPA 3.1 specification, the latest version, was released in the spring of 2022 **as part of Jakarta EE 10**

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.

If you are looking for a powerful and easy-to-use way to interact with relational databases in Java applications, then Spring Framework JPA is a great option.

Here are some of the benefits of using Spring Framework JPA:

Simple and consistent API: Spring Framework JPA provides a simple and consistent API for performing CRUD operations, as well as more advanced features such as pagination, auditing, and querying.

Compatibility with a wide range of databases: Spring Framework JPA is compatible with a wide range of databases, including MySQL, PostgreSQL, Oracle, and SQL Server.

Ease of use: Spring Framework JPA is easy to use, even for developers who are not familiar with JPA.

Performance: Spring Framework JPA is designed for performance, and it can provide significant performance improvements over traditional JDBC code.

If you are looking for a powerful and easy-to-use way to interact with relational databases in Java applications, then Spring Framework JPA is a great option.

Here are the steps involved in using Spring JPA in our application:

- Create entity classes that represent the data that you want to store in the database.
- Annotate the entity classes with JPA annotations.
- Create a Spring Data repository interface that extends one of the Spring Data repository interfaces, such as JpaRepository or CrudRepository.
- Inject the repository interface into your application code.
- Use the repository interface to access the data in the database.

Spring JPA Module Project Example:

Requirement: CURD Operations with Order table

NOTE: We are integrating Hibernate ORM framework implantation of JPA in our Spring JPA module.

Project Setup:

- Create Maven Project
- Add required Dependencies for Spring JPA Modules in **pom.xml** file.

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.flipkart</groupId>
  <artifactId>flipkart</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>6.0.11</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>6.0.11</version>
    </dependency>
    <dependency>
      <groupId>com.oracle.database.jdbc</groupId>
      <artifactId>ojdbc8</artifactId>
      <version>21.9.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>6.0.11</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.2.6.Final</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-jpa</artifactId>
      <version>3.1.2</version>
    </dependency>
  </dependencies>
</project>

```

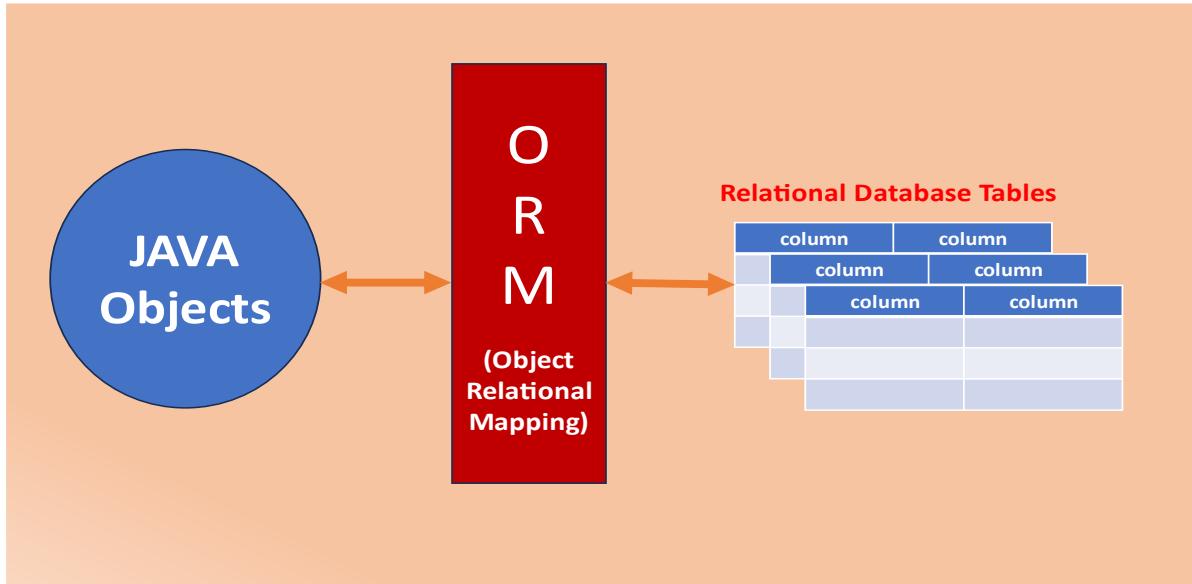
- Define an Entity class for Orders table.

What is Entity Class?

An entity class is a Java class that represents a row in a database table. It is used to store data in the database and to interact with the database. Entity classes are annotated with the `@Entity` annotation, which tells the Java Persistence API (JPA) that the class is an entity. Entities in JPA are nothing but POJOs representing data that can be persisted 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.



Few Annotations of JPA Entity class:

@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.

How to Create an Entity class:

First we should have Database table details with us, Based on Table Details we are creating a POJO class i.e. configuring Table and column details along with POJO class Properties. I have a table in my database as following. Then I will create POJO class by creating Properties aligned to DB table datatypes and column names with help of JPA annotations.

Table Name : flipkart_orders

Table : FLIPKART_ORDERS		JAVA Entity Class : FlipkartOrder	
ORDERID	NUMBER(10)	orderID	long
PRODUCTNAME	VARCHAR2(50)	productName	String
TOTALAMOUNT	NUMBER(10,2)	totalAmount	float

Entity Class: FlipkartOrder.java

```
package flipkart.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "FLIPKART_ORDERS")
public class FlipkartOrder {

    @Column(name = "ORDERID")
    private long orderID;

    @Column(name = "PRODUCTNAME")
    private String productName;

    @Column(name = "TOTALAMOUNT")
    private float totalAmount;

    public long getOrderID() {
        return orderID;
    }

    public void setOrderID(long orderID) {
        this.orderID = orderID;
    }

    public String getProductName() {
        return productName;
    }

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

    public float getTotalAmount() {
        return totalAmount;
    }

    public void setTotalAmount(float totalAmount) {
        this.totalAmount = totalAmount;
    }
}
```

@Table and **@Column** Annotations are used in **@Entity** class to represent database table details, **name** is an attribute. Inside **@Table**, **name** value should be **Database table name**. Inside **@Column**, **name** value should be **table column name**.

Note: When Database **Table column name** and **Entity class property name** are equal, it's not mandatory to use **@Column** annotation i.e. It's an Optional in such case. If both are different then we should use **@Column** annotation along with value.

For Example : Assume, we written as below in a Entity class.

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

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

```
private int pincode;
```

Spring JPA Repositories:

In Spring Data JPA, a repository is an abstraction that provides an interface to interact with a database using Java Persistence API (JPA). Spring Data JPA repositories offer a set of common methods for performing CRUD (Create, Read, Update, Delete) operations on database entities without requiring you to write boilerplate code. These repositories also allow you to define custom queries using method names, saving you from writing complex SQL queries manually.

Spring JPA Provided 2 Types of Repositories

- **JpaRepository**
- **CrudRepository**

Repositories work in Spring JPA by extending the **JpaRepository/CrudRepository** interface. These interfaces provides a number of default methods for performing CRUD operations, such as **save**, **findById**, and **delete** etc.. we can also extend the JpaRepository interface to add your own custom methods.

When we create a repository, Spring Data JPA will automatically create an implementation for it. This implementation will use the JPA provider that you have configured in your Spring application.

Here is an example of a simple repository:

```
public interface OrderRepository extends JpaRepository< FlipkartOrder, Long > {
}
```

This repository is for a **FlipkartOrder** entity. The **Long** parameter in the extends JpaRepository statement specifies the type of the entity's identifier representing primary key column in Table.

The **OrderRepository** interface provides a number of default methods for performing **CRUD operations** on **FlipkartOrder** entities. For example, the **save()** method

can be used to save a new **FlipkartOrder** entity, and the **findById()** method can be used to find a **FlipkartOrder** entity by its identifier.

We can also extend the **OrderRepository** interface to add your own custom methods. For example, you could add a method to find all **FlipkartOrder** entities that based on a Product name.

Benefits of using Spring JPA Repositories:

Reduced boilerplate code: Repositories provide a number of default methods for performing CRUD operations, so you don't have to write as much code and SQL Queries.

Enhanced flexibility: Repositories allow you to add your own custom methods, so you can tailor your data access code to your specific requirements.

If We are using JPA in your Spring application, highly recommended using Spring JPA Repositories. They will make your code simpler, more consistent, and more flexible.

Here's how repositories work in Spring Data JPA:

Define an Entity Class: An entity class is a Java class that represents a database table. It is annotated with `@Entity` and contains fields that map to table columns.

Create a Repository Interface: Create an interface that extends the `JpaRepository` interface provided by Spring Data JPA. This interface will be used to perform database operations on the associated entity. You can also extend other repository interfaces such as `PagingAndSortingRepository`, `CrudRepository`, etc., based on your needs.

Method Naming Conventions: Spring Data JPA automatically generates queries based on the method names defined in the repository interface. For example, a method named `findByFirstName` will generate a query to retrieve records based on the first name.

Custom Queries: You can define custom query methods by using specific keywords in the method name, such as `find...By...`, `read...By...`, `query...By...`, or `get...By...`. Additionally, you can use `@Query` annotations to write JPQL (Java Persistence Query Language) or native SQL queries.

Dependency Injection: Inject the repository interface into your service or controller classes using Spring's dependency injection.

Use Repository Methods: You can now use the methods defined in the repository interface to perform database operations. Spring Data JPA handles the underlying database interactions, such as generating SQL queries, executing them, and mapping the results back to Java objects.

Spring JPA Configuration:

- For using Spring Data, first of all we have to configure **DataSource** bean. Then we need to configure **LocalContainerEntityManagerFactoryBean** bean.
- The next step is to configure bean for transaction management. In our example it's **JpaTransactionManager**.
- @EnableTransactionManagement**: This annotation allows users to use transaction management in application.
- @EnableJpaRepositories("flipkart.*")**: indicates where the repositories classes are present.

Configuring the DataSource Bean:

- Configure the database connection. We need to set the name of the JDBC url, the username of database user, and the password of the database user.

Configuring the Entity Manager Factory Bean:

We can configure the entity manager factory bean by following steps:

- Create a new **LocalContainerEntityManagerFactoryBean** object. We need to create this object because it creates the JPA **EntityManagerFactory**.
- Configure the Created **DataSource** in Previous Step.
- Configure the Hibernate specific implementation of the **HibernateJpaVendorAdapter**. It will initialize our configuration with the default settings that are compatible with Hibernate.
- Configure the packages that are scanned for entity classes.
- Configure the JPA/Hibernate properties that are used to provide additional configuration to the used JPA provider.

Configuring the Transaction Manager Bean:

Because we are using JPA, we have to create a transaction manager bean that integrates the JPA provider with the Spring transaction mechanism. We can do this by using the **JpaTransactionManager** class as the transaction manager of our application.

We can configure the transaction manager bean by following steps:

- Create a new **JpaTransactionManager** object.
- Configure the entity manager factory whose transactions are managed by the created **JpaTransactionManager** object.

```
package flipkart.entity;

import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
```

```

import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;

@Configuration
@EnableJpaRepositories("flipkart.*")
public class SpringJpaConfiguration {

    //DB Details
    @Bean
    public DataSource getDataSource() {
        DriverManagerDataSource dataSource =
            new DriverManagerDataSource();
        dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
        dataSource.setUsername("c##dilip");
        dataSource.setPassword("dilip");
        return dataSource;
    }

    @Bean("entityManagerFactory")
    LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {
        LocalContainerEntityManagerFactoryBean factory
            = new LocalContainerEntityManagerFactoryBean();

        // 1. Setting Datasource Object // DB details
        factory.setDataSource(getDataSource());
        // 2. Provide package information of entity classes
        factory.setPackagesToScan("flipkart.*");
        // 3. Providing Hibernate Properties to EM
        factory.setJpaProperties(hibernateProperties());
        // 4. Passing Predefined Hiberante Adaptor Object EM
        HibernateJpaVendorAdapter adapter =
            new HibernateJpaVendorAdapter();
        factory.setJpaVendorAdapter(adapter);

        return factory;
    }

    @Bean("transactionManager")
    public PlatformTransactionManager createTransactionManager() {
        JpaTransactionManager transactionManager = new
        JpaTransactionManager();

        transactionManager.setEntityManagerFactory(createEntityManagerFactory()
            .getObject());
        return transactionManager;
    }
}

```

```

}

// these are all from hibernate FW , Predefined properties : Keys
Properties hibernateProperties() {

    Properties hibernateProperties = new Properties();
    hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "create");

    //This is for printing internally generated SQL Queries
    hibernateProperties.setProperty("hibernate.show_sql", "true");
    return hibernateProperties;
}
}

```

- Now create another Component class For Performing DB operations as per our Requirement:

```

package flipkart.entity;

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

// To Execute/Perform DB operations
@Component
public class OrderDbOperations {

    @Autowired
    FlipkartOrderRepository flipkartOrderRepository;

    public void addOrderDetails(FlipkartOrder order) {
        flipkartOrderRepository.save(order);
    }
}

```

- Now Create a Main method class for creating Spring Application Context for loading all Configurations and Component classes.

```

package flipkart.entity;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context =
            new

```

```

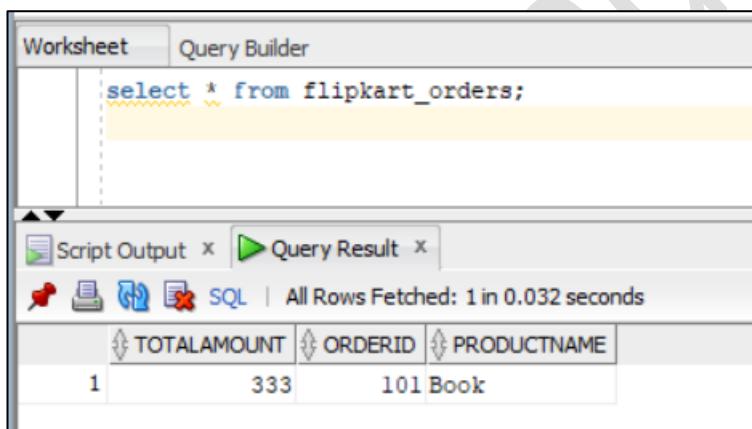
AnnotationConfigApplicationContext();
    context.scan("flipkart.*");
    context.refresh();

    // Created Entity Object
    FlipkartOrder order = new FlipkartOrder();
    order.setOrderID(9988);
    order.setProductName("Book");
    order.setTotalAmount(333.00);

    // Pass Entity Object to Repository MEthod
    OrderDbOperations dbOperation = 
context.getBean(OrderDbOperations.class);
    dbOperation.addOrderDetails(order);
}
}

```

- Execute The Programme Now. Verify In Database Now.



In Eclipse Console Logs, Printed internally generated SQL Queries to perform insert operation.

NOTE: In our example, we are nowhere written any SQL query to do Database operation.

Internally, Based on Repository method **save()** of **flipkartOrderRepository.save(order)**, JPA internally generates implementation code, SQL queries and will be executed internally.

Similarly, we have many predefined methods of Spring repository to do CRUD operations.

We learned to configure the persistence layer of a Spring application that uses Spring Data JPA and Hibernate. Let's create few more examples to do CRUD operations on Db table.

From Spring JPA Configuration class, we have used two properties related to Hibernate FW.

hibernate.hbm2ddl.auto:

The `hibernate.hbm2ddl.auto` property is used to configure the automatic schema generation and management behaviour of Hibernate. This property allows you to control how Hibernate handles the database schema based on your entity classes.

Here are the possible values for the **hibernate.hbm2ddl.auto** property:

1. **none**: No action is performed. The schema will not be generated.
2. **validate**: The database schema will be validated using the entity mappings. This means that Hibernate will check to see if the database schema matches the entity mappings. If there are any differences, Hibernate will throw an exception.
3. **update**: The database schema will be updated by comparing the existing database schema with the entity mappings. This means that Hibernate will create or modify tables in the database as needed to match the entity mappings.
4. **create**: The database schema will be created. This means that Hibernate will create all of the tables needed for the entity mappings.
5. **create-drop**: The database schema will be created and then dropped when the SessionFactory is closed. This means that Hibernate will create all of the tables needed for the entity mappings, and then drop them when the SessionFactory is closed.

hibernate.show_sql:

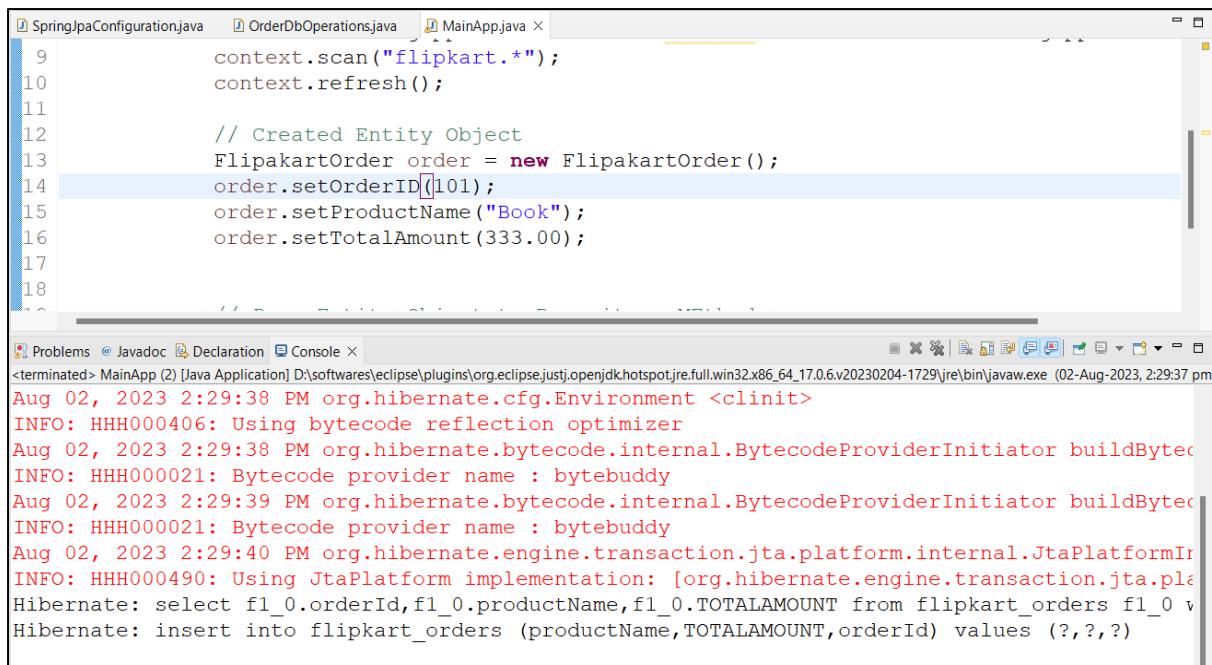
The **hibernate.show_sql** property is a Hibernate configuration property that controls whether or not Hibernate will log the SQL statements that it generates. The possible values for this property are:

true: Hibernate will log all SQL statements to the console.

false: Hibernate will not log any SQL statements.

The default value for this property is **false**. This means that Hibernate will not log any SQL statements by default. If you want to see the SQL statements that Hibernate is generating, you will need to set this property to **true**.

Logging SQL statements can be useful for debugging purposes. If you are having problems with your application, you can enable logging and see what SQL statements Hibernate is generating. This can help you to identify the source of the problem.



The screenshot shows the Eclipse IDE interface. In the top editor tab, there are three tabs: 'SpringJpaConfiguration.java', 'OrderDbOperations.java', and 'MainApp.java'. The 'MainApp.java' tab is active, displaying Java code for creating a Spring configuration and a FlipkartOrder object. Below the editor, the 'Console' tab is selected, showing the command-line output of the application's execution. The log output includes standard Hibernate startup messages and a SQL query and insert statement for interacting with a database table named 'flipkart_orders'.

```

9         context.scan("flipkart.*");
10        context.refresh();
11
12        // Created Entity Object
13        FlipkartOrder order = new FlipkartOrder();
14        order.setOrderID(101);
15        order.setProductName("Book");
16        order.setTotalAmount(333.00);
17
18
Aug 02, 2023 2:29:38 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000406: Using bytecode reflection optimizer
Aug 02, 2023 2:29:38 PM org.hibernate.bytecode.internal.BytecodeProviderInitiator buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : bytebuddy
Aug 02, 2023 2:29:39 PM org.hibernate.bytecode.internal.BytecodeProviderInitiator buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : bytebuddy
Aug 02, 2023 2:29:40 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformImpl
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformImpl]
Hibernate: select f1_0.orderId,f1_0.productName,f1_0.TOTALAMOUNT from flipkart_orders f1_0
Hibernate: insert into flipkart_orders (productName,TOTALAMOUNT,orderId) values (?,?,?)

```

Creation of New Spring JPA Project:

Requirement : Patient Information

- Name
- Age
- Gender
- Contact Number
- Email Id

1. Add Single Patient Details
2. Add More Than One Patient Details
3. Update Patient Details
4. Select Single Patient Details
5. Select More Patient Details
6. Delete Patient Details

1. Create Simple Maven Project and Add Required Dependencies

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.dilip</groupId>
  <artifactId>spring-jpa-two</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>

```

```

<artifactId>spring-core</artifactId>
<version>6.0.11</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.0.11</version>
</dependency>
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc8</artifactId>
    <version>21.9.0.0</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>6.0.11</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.2.6.Final</version>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>3.1.2</version>
</dependency>
</dependencies>
</project>

```

2. Now Create Spring JPA Configuration

```

package com.dilip;

import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;

@Configuration
@EnableJpaRepositories("com.*")
public class SpringJpaConfiguration {

    //DB Details
}

```

```
@Bean
public DataSource getDataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
    dataSource.setUsername("c##dilip");
    dataSource.setPassword("dilip");
    return dataSource;
}

@Bean("entityManagerFactory")
LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {

    LocalContainerEntityManagerFactoryBean factory = new
    LocalContainerEntityManagerFactoryBean();

    // 1. Setting Datasource Object // DB details
    factory.setDataSource(getDataSource());

    // 2. Provide package information of entity classes
    factory.setPackagesToScan("com.*");

    // 3. Providing Hibernate Properties to EM
    factory.setJpaProperties(hibernateProperties());

    // 4. Passing Predefined Hiberante Adaptor Object EM
    HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
    factory.setJpaVendorAdapter(adapter);

    return factory;
}

//Spring JPA: configuring data based on your project req.
@Bean("transactionManager")
public PlatformTransactionManager createTransactionManager() {
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(createEntityManagerFactory())
        .getObject();
    return transactionManager;
}

// these are all from hibernate FW , Predefined properties : Keys
Properties hibernateProperties() {
    Properties hibernateProperties = new Properties();
    hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");
    //This is for printing internally genearted SQL queries
    hibernateProperties.setProperty("hibernate.show_sql", "true");
    return hibernateProperties;
}
```

3. Create Entity Class

NOTE : Configured `hibernate.hbm2ddl.auto` value as `update`. So Table Creation will be done by JPA internally.

```
package com.dilip.entity;

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

@Entity
@Table
public class Patient {

    @Id
    @Column
    private String emailld;

    @Column
    private String name;

    @Column
    private int age;

    @Column
    private String gender;

    @Column
    private String contact;

    public Patient() {
        super();
        // TODO Auto-generated constructor stub
    }

    public Patient(String name, int age, String gender, String contact, String emailld) {
        super();
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.contact = contact;
        this.emaiilld = emaiilld;
    }

    public String getName() {
        return name;
    }
}
```

```

    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
    public String getContact() {
        return contact;
    }
    public void setContact(String contact) {
        this.contact = contact;
    }
    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }

    @Override
    public String toString() {
        return "Patient [name=" + name + ", age=" + age + ", gender=" + gender + ",
contact=" + contact + ", emailId=" +
                + emailId + "]";
    }
}

```

4. Create A Repository Now

```

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {
}

```

5. Create a class for DB operations

```

@Component
public class PatientOperations {
    @Autowired
    PatientRepository repository;
}

```

{}

Spring JPA Repositories Provided many predefined abstract methods for all DB CURD operations. We should recognize those as per our DB operation.

Requirement : Add Single Patient Details

Here, we are adding Patient details means at Database level this is insert Query Operation.

save() : Used for insertion of Details. We should pass Entity Object.

- **Add Below Method in PatientOperations.java:**

```
public void addPatient(Patient p) {
    repository.save(p);
}
```

- **Now Test it** : Create Main method class

```
package com.dilip.operations;

import java.util.ArrayList;
import java.util.List;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.dilip.entity.Patient;

public class PatientApplication {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();

        PatientOperations ops = context.getBean(PatientOperations.class);

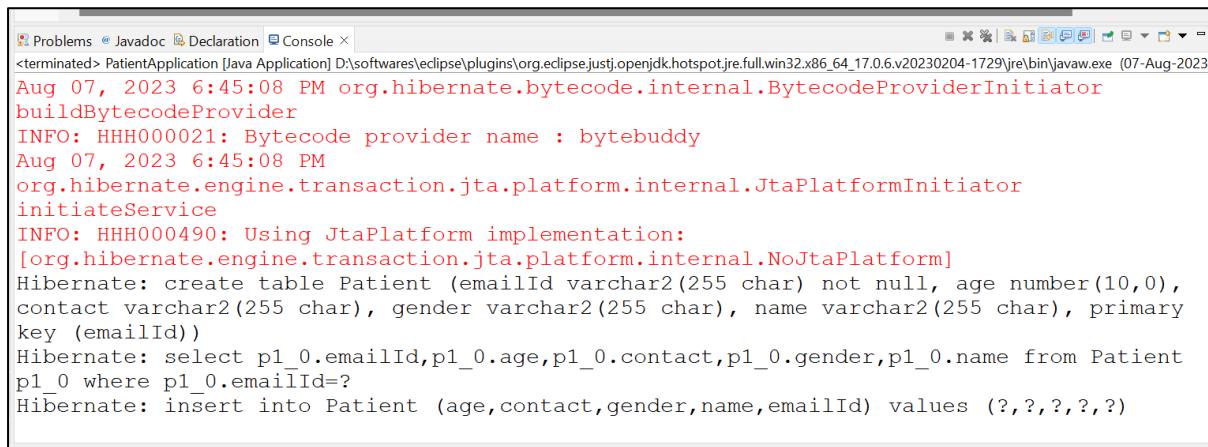
        // Add Single Patient
        Patient p = new Patient();
        p.setEmailId("one@gmail.com");
        p.setName("One Singh");
        p.setContact("+918826111377");
        p.setAge(30);
        p.setGender("MALE");
    }
}
```

```

        ops.addPatient(p);
    }
}

```

- Now Execute It. Table also created by JPA module and One Record is inserted.



The screenshot shows the Eclipse IDE's Console tab with the following log output:

```

Problems Declaration Console <terminated> PatientApplication [Java Application] D:\softwares\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (07-Aug-2023)
Aug 07, 2023 6:45:08 PM org.hibernate.bytecode.internal.BytecodeProviderInitiator
buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : bytebuddy
Aug 07, 2023 6:45:08 PM
org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator
initiateService
INFO: HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Hibernate: create table Patient (emailId varchar2(255 char) not null, age number(10,0),
contact varchar2(255 char), gender varchar2(255 char), name varchar2(255 char), primary
key (emailId))
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient
p1_0 where p1_0.emailId=?
Hibernate: insert into Patient (age,contact,gender,name,emailId) values (?, ?, ?, ?, ?)

```

Requirement: Add multiple Patient Details at a time

Here, we are adding Multiple Patient details means at Database level this is also insert Query Operation.

saveAll() : This is for adding List of Objects at a time. We should pass List Object of Patient Type.

Add Below Method in PatientOperations.java:

```

public void addMorePatients(List<Patient> patients) {
    repository.saveAll(patients);
}

```

Now Test it : Inside Main method class, add Logic below.

```

package com.dilip.operations;

import java.util.ArrayList;
import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new
            AnnotationConfigApplicationContext();
        context.scan("com.*");
    }
}

```

```

        context.refresh();

        PatientOperations ops = context.getBean(PatientOperations.class);

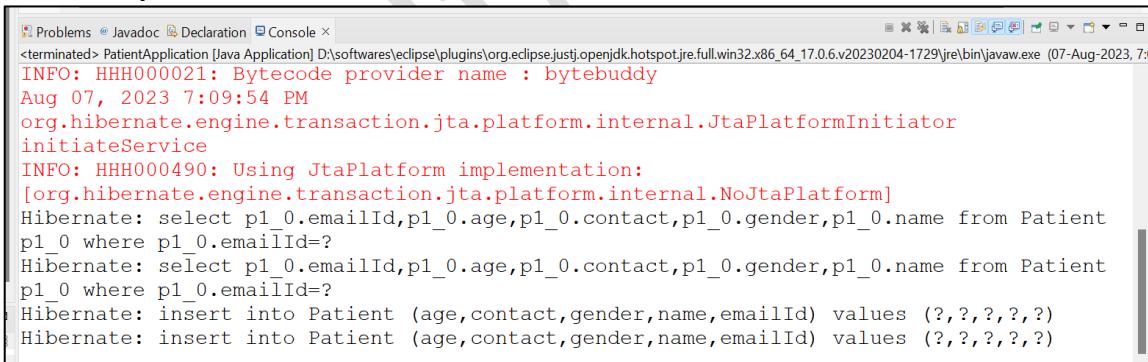
        // Adding More Patients
        Patient p1 = new Patient();
        p1.setEmailId("two@gmail.com");
        p1.setName("Two Singh");
        p1.setContact("+828388");
        p1.setAge(30);
        p1.setGender("MALE");

        Patient p2 = new Patient();
        p2.setEmailId("three@gmail.com");
        p2.setName("Xyz Singh");
        p2.setContact("+44343423");
        p2.setAge(36);
        p2.setGender("FEMALE");

        List<Patient> allPatients = new ArrayList<>();
        allPatients.add(p1);
        allPatients.add(p2);
        ops.addMorePatients(allPatients);
    }
}

```

Execution Output:



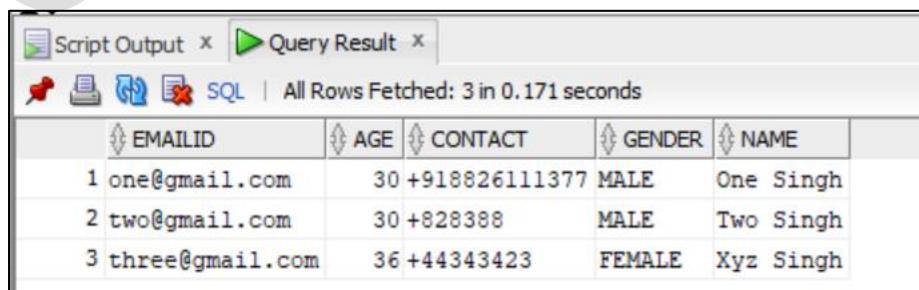
The screenshot shows the Eclipse IDE's Console view with the following log output:

```

Problems Javadoc Declaration Console x
<terminated> PatientApplication [Java Application] D:\softwares\eclipse\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (07-Aug-2023, 7:09:54 PM)
INFO: HHH000021: Bytecode provider name : bytewuddy
INFO: HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.emailId=?
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.emailId=?
Hibernate: insert into Patient (age,contact,gender,name,emailId) values (?, ?, ?, ?, ?)
Hibernate: insert into Patient (age,contact,gender,name,emailId) values (?, ?, ?, ?, ?)

```

Verify In DB Table:



The screenshot shows the MySQL Workbench interface with a table named 'Patient'. The table has columns: EMAILID, AGE, CONTACT, GENDER, and NAME. The data is as follows:

EMAILID	AGE	CONTACT	GENDER	NAME
1 one@gmail.com	30	+918826111377	MALE	One Singh
2 two@gmail.com	30	+828388	MALE	Two Singh
3 three@gmail.com	36	+44343423	FEMALE	Xyz Singh

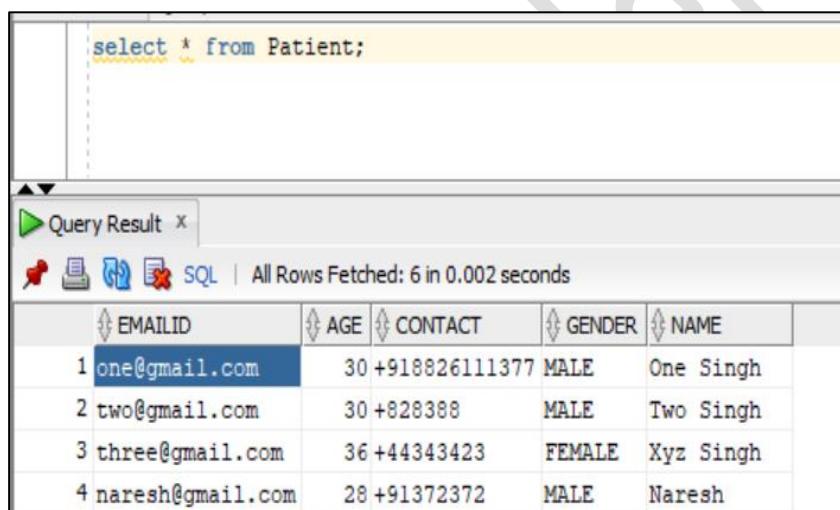
Requirement : Update Patient Details

In Spring Data JPA, the **save()** method is commonly used for both **insert** and **update** operations. When you call the **save()** method on a repository, Spring Data JPA checks whether the entity you're trying to save already exists in the database. If it does, it updates the existing entity; otherwise, it inserts a new entity.

So that is the reason we are seeing a select query execution before inserting data in previous example. After select query execution with primary key column JPA checks row count and if it is 1, then JPA will convert entity as insert operation. If count is 0 , then Spring JPA will convert entity as update operation specific to Primary key.

Using the **save()** method for updates is a common and convenient approach, especially when we want to leverage Spring Data JPA's automatic change tracking and transaction management.

Requirement: Please update name as **Dilip Singh** for email id: **one@gmail.com**



The screenshot shows a MySQL Workbench interface. In the top pane, there is a SQL editor window containing the query: `select * from Patient;`. Below it, a results window titled "Query Result" displays the following data:

EMAILID	AGE	CONTACT	GENDER	NAME
1 one@gmail.com	30	+918826111377	MALE	One Singh
2 two@gmail.com	30	+828388	MALE	Two Singh
3 three@gmail.com	36	+44343423	FEMALE	Xyz Singh
4 naresh@gmail.com	28	+91372372	MALE	Naresh

The first row (one@gmail.com) is highlighted in blue, indicating it is selected for update.

➤ Add Below Method in PatientOperations.java:

```
public void updatePateinData(Patient p) {
    repository.save(p);
}
```

Now Test it from Main class: In below if we observe, first select query executed by JPA as per our entity Object, JPA found data so JPA decided for update Query execution.

```

PatientApplication.java
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;
public class PatientApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();
        PatientOperations ops = context.getBean(PatientOperations.class);
        //Update Patient Details
        Patient abc = new Patient();
        abc.setEmailId("one@gmail.com");
        abc.setName("Dilip Singh");
        abc.setAge(30);
        abc.setGender("MALE");
        abc.setContact("+918826111377");
        ops.updatePatientData(abc);
    }
}

Java Console Output:
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.emailId=?
Hibernate: update Patient set age=?,contact=?,gender=?,name=? where emailId=?

System Bar Icons: Writable, Smart Insert, 11:1:368, ENG IN, 10:00, 08-08-2023

```

Verify In DB :

EMAILID	AGE	CONTACT	GENDER	NAME
one@gmail.com	30	+918826111377	MALE	Dilip Singh

Requirement: Delete Patient Details

Deleting Patient Details based on Email ID.

Spring JPA provided a predefined method **deleteById()** for primary key columns delete operations.

deleteById():

The **deleteById()** method in Spring Data JPA is used to remove an entity from the database based on its primary key (ID). It's provided by the **JpaRepository** interface and allows you to delete a single entity by its unique identifier.

Here's how you can use the **deleteById()** method in a Spring Data JPA repository:

Add Below Method in PatientOperations.java:

```
public void deletePatient(String email) {
    repository.deleteById(email);
}
```

Testing from Main Class:

```
package com.dilip.operations;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class PatientApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();
        PatientOperations ops = context.getBean(PatientOperations.class);
        //Delete Patient Details
        ops.deletePatientWithEmailID("two@gmail.com");
    }
}
```

Before Execution/Deletion:

EMAILID	AGE	CONTACT	GENDER	NAME
1 one@gmail.com	30	+918826111377	MALE	Dilip Singh
2 two@gmail.com	30	+828388	MALE	Two Singh
3 three@gmail.com	36	+44343423	FEMALE	Xyz Singh
4 naresh@gmail.com	28	+91372372	MALE	Naresh
5 dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
6 suresh@gmail.com	28	+91372372	MALE	Suresh
7 laxmi@gmail.com	28	+91372372	MALE	Suresh
8 vijay@gmail.com	28	+91372372	MALE	Suresh

After Execution/Deletion:

EMAILID	AGE	CONTACT	GENDER	NAME
1 one@gmail.com	30	+918826111377	MALE	Dilip Singh
2 three@gmail.com	36	+44343423	FEMALE	Xyz Singh
3 naresh@gmail.com	28	+91372372	MALE	Naresh
4 dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
5 suresh@gmail.com	28	+91372372	MALE	Suresh
6 laxmi@gmail.com	28	+91372372	MALE	Suresh
7 vijay@gmail.com	28	+91372372	MALE	Suresh

Requirement: Get Patient Details Based on Email Id.

Here Email Id is Primary key Column. Finding Details based on Primary key column name Spring JPA provide a method **findById()**.

- Add Below Method in PatientOperations.java:

```
public Patient fetchByEmailId(String emailId) {
    return repository.findById(emailId).get();
}
```

➤ **Testing from Main Class:**

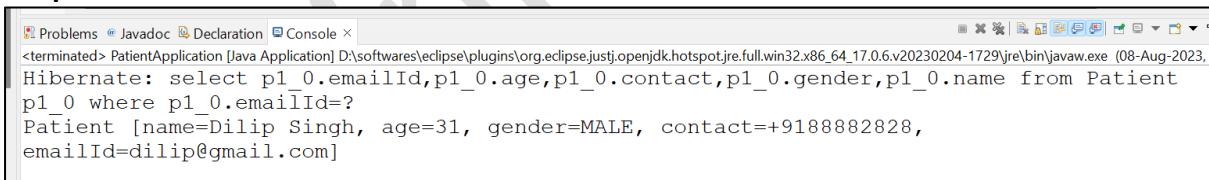
```
package com.dilip.operations;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();

        PatientOperations ops = context.getBean(PatientOperations.class);
        //Fetch Patient Details By Email ID
        Patient patient = ops.fetchByEmailId("dilip@gmail.com");
        System.out.println(patient);
    }
}
```

Output:



```
Problems Javadoc Declaration Console ×
<terminated> PatientApplication [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe [08-Aug-2023]
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient
p1_0 where p1_0.emailId=?
Patient [name=Dilip Singh, age=31, gender=MALE, contact=+9188882828,
emailId=dilip@gmail.com]
```

Similarly Spring JPA provided many useful and predefined methods inside JPA repositories to perform CRUD Operations.

For example :

findAll() : for retrieve all Records From Table
deleteAll(): for deleting all Records From Table
etc..

For Non Primary Key columns of Table or Entity, Spring JPA provided Custom Query Repository Methods. Let's Explore.

Spring Data JPA Custom Query Repository Methods:

Spring Data JPA allows you to define custom repository methods by simply declaring method signature with **entity class property Name** which is aligned with Database column. The method name must start with **findBy**, **getBy**, **queryBy**, **countBy**, or **readBy**. The **findBy** is mostly used by the developer.

For Example: Below query methods are valid and gives same result like Patient name matching data from Database.

```
public List<Patient> findByName(String name);
public List<Patient> getByName(String name);
public List<Patient> queryByName(String name);
public List<Patient> countByName(String name);
public List<Patient> readByName(String name);
```

- **Patient:** Name of Entity class.
- **Name:** Property name of Entity.

Rule: After **findBy**, The first character of Entity class field name should Upper case letter. Although if we write the first character of the field in lower case then it will work but we should use **camelCase** for the method names. Equal Number of Method Parameters should be defined in abstract method.

Requirement: Get Details of Patients by Age i.e. Single Column.

Result we will get More than One record i.e. List of Entity Objects. So return type is **List<Patient>**

- **Step 1:** Create a Custom method inside Repository

```
package com.dilip.repository;

import java.util.List;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {
    List<Patient> findByAge(int age);
}
```

- **Step 2:** Now call Above Method inside Db operations to pass Age value.

```

package com.dilip.operations;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

    // Non- primary Key column
    public List<Patient> fetchByAge(int age) {
        return repository.findByAge(age);
    }
}

```

➤ **Step 3: Now Test It from Main Class.**

```

package com.dilip.operations;

import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();
        PatientOperations ops = context.getBean(PatientOperations.class);
        //Fetch Patient Details By Age
        List<Patient> patients = ops.fetchByAge(31);
        System.out.println(patients);
    }
}

```

Output: In Below, Query generated by JPA and Executed. Got Two Entity Objects In Side List

```
[log] [INFO] [main] 2023-07-20 11:11:37,772 [main] INFO o.s.d.r.h.HibernateTemplate - Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.age=?  
[Patient [name=Dilip Singh, age=31, gender=MALE, contact=+918826111377,  
emailId=dilip@gmail.com], Patient [name=Dilip Singh, age=31, gender=MALE, contact=  
+9188882828, emailId=dilip@gmail.com]]
```

Similar to **Age**, we can fetch data with other columns as well by defining custom Query methods.

Fetching Data with Multiple Columns:

Rule: We can write the query method using multiple fields using predefined keywords(eg. **And**, **Or** etc) but these keywords are case sensitive. **We must use “And” instead of “and”.**

Requirement: Fetch Data with Age and Gender Columns.

- Age is 28
- Gender is Female

Step 1: Create a Custom findBy method inside Repository.

Method should have 2 parameters **age** and **gender** in this case because we are getting data with 2 properties.

```
package com.dilip.repository;  
  
import java.util.List;  
import org.springframework.data.repository.CrudRepository;  
import org.springframework.stereotype.Component;  
import com.dilip.entity.Patient;  
  
@Component  
public interface PatientRepository extends CrudRepository<Patient, String>  
{  
    List<Patient> findByAgeAndGender(int age, String gender);  
}
```

Step 2: Now call Above Method inside Db operations to pass Age and gender values.

```
package com.dilip.operations;  
  
import java.util.List;  
import org.springframework.beans.factory.annotation.Autowired;
```

```

import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {
    @Autowired
    PatientRepository repository;

    // based on Age + Gender
    public List<Patient> getPatientsWithAgeAndGender(int age, String gender) {
        return repository.findByAgeAndGender(age, gender);
    }
}

```

Step 3: Now Test It from Main Class.

```

package com.dilip.operations;

import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();
        PatientOperations ops = context.getBean(PatientOperations.class);
        //Fetch Patient Details By Age and Gender
        List<Patient> patients = ops.getPatientsWithAgeAndGender(28, "FEMALE");
        System.out.println(patients);
    }
}

```

Table Data:

EMAILID	AGE	CONTACT	GENDER	NAME
1 one@gmail.com	31	+918826111377	MALE	Dilip Singh
2 three@gmail.com	36	+44343423	FEMALE	Xyz Singh
3 naresh@gmail.com	28	+91372372	MALE	Naresh
4 vijay45@gmail.com	28	+91372372	MALE	Suresh
5 dilip@gmail.com	31	+9188882828	MALE	Dilip Singh
6 suresh@gmail.com	28	+91372372	MALE	Suresh
7 laxmi@gmail.com	28	+91372372	FEMALE	Laxmi
8 vijay@gmail.com	28	+91372372	MALE	Suresh

Expected Output:

```
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient
p1_0 where p1_0.age=? and p1_0.gender=?
[Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com] ]
```

WritableDatabase Smart Insert 13 : 27 : 394

We can write the query method if we want to restrict the number of records by directly providing the number as the digit in method name. We need to add the First or the Top keywords before the By and after find.

```
public List<Student> findFirst3ByName(String name);
public List<Student> findTop3ByName(String name);
```

Both query methods will return only first 3 records.

Similar to these examples and operations we can perform multiple Database operations however we will do in SQL operations.

List of keywords used to write custom repository methods:

And, Or, Is, Equals, Between, LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, After, Before, IsNull, IsNotNull, NotNull, Like, NotLike, StartingWith, EndingWith, Containing, OrderBy, Not, In, NotIn, True, False, IgnoreCase.

Some of the examples for Method Names formations:

```
public List<Student> findFirst3ByName(String name);
public List<Student> findByNames(String name);
public List<Student> findByNameEquals(String name);
public List<Student> findByRollNumber(String rollNumber);
public List<Student> findByUniversity(String university);
public List<Student> findByNameAndRollNumber(String name, String rollNumber);
public List<Student> findByRollNumberIn(List<String> rollNumbers);
public List<Student> findByRollNumberNotIn(List<String> rollNumbers);
public List<Student> findByRollNumberBetween(String start, String end);
public List<Student> findByNameNot(String name);
public List<Student> findByNameContainingIgnoreCase(String name);
public List<Student> findByNameLike(String name);
public List<Student> findByRollNumberGreaterThan(String rollnumber);
public List<Student> findByRollNumberLessThan(String rollnumber);
```

Native Queries & JPQL Queries with Spring JPA:

Native Query is Custom SQL query. In order to define SQL Query to execute for a Spring Data repository method, we have to annotate the method with the **@Query** annotation. This annotation value attribute contains the SQL or JPQL to execute in Database. We will define **@Query** above the method inside the repository.

Spring Data JPA allows you to execute native SQL queries by using the **@Query** annotation with the **nativeQuery** attribute set to **true**. For example, the following method uses the **@Query** annotation to execute a native SQL query that selects all customers from the database:

```
@Query(value = "SELECT * FROM customer", nativeQuery = true)
public List<Customer> findAllCustomers();
```

The **@Query** annotation allows you to specify the SQL query that will be executed. The **nativeQuery** attribute tells Spring Data JPA to execute the query as a native SQL query, rather than considering it to JPQL.

JPQL Query:

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

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

```
// JPQL Query in Repository Layer
@Query(value = "Select u from Users u")
List<Users> getUsers();
```

JPQL can perform:

- It is a platform-independent query language.
- It can be used with any type of database such as MySQL, Oracle.
- join operations
- update and delete data in a bulk.
- It can perform aggregate function with sorting and grouping clauses.
- Single and multiple value result types.

Native SQL Querys:

We can use **@Query** to define our Native Database SQL query. All we have to do is set the value of the **nativeQuery** attribute to **true** and define the native SQL query in the **value** attribute of the annotation.

Example, Below Repository Method representing Native SQL Query to get all records.

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

For passing values to Positional parameters of SQL Query from method parameters, JPA provides 2 possible ways.

1. Indexed Query Parameters
2. Named Query Parameters

By using Indexed Query Parameters:

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

Example: Get All Records Of Table

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

Example: Get All Records Of Table where city is matching

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

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

Example with more indexed parameters: users from either **city** or **pincode** matches.

Example: Get All Records Of Table where city or pincode is matching

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

Examples:

Requirement:

1. Get All Patient Details
2. Get All Patient with Email Id
3. Get All Patients with Age and Gender

Step 1: Define Methods Inside Repository with Native Queries:

```
package com.dilip.repository;
```

```

import java.util.List;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import com.dilip.entity.Patient;

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {

    //Get All Patients
    @Query(value = "select * from patient", nativeQuery = true)
    public List<Patient> getAllPatients();

    //Get All Patient with EmailId
    @Query(value = "select * from patient where emailid=?1", nativeQuery = true)
    public Patient getDetailsByEmail(String email);

    //Get All Patients with Age and Gender
    @Query(value = "select * from patient where age=?1 and gender=?2", nativeQuery =
true)
    public List<Patient> getPatientDetailsByAgeAndGender(int age, String gender);
}

```

➤ Step 2: Call Above Methods from DB Operations Class

```

package com.dilip.operations;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

    //Select all patients
}

```

```

public List<Patient> getPatientDetails() {
    return repository.getAllPatients();
}
//by email Id
public Patient getPatientDetailsbyEmailId(String email) {
    return repository.getDetailsByEmail(email);
}
//age and gender
public List<Patient> getPatientDetailsbyAgeAndGender(int age, String gender) {
    return repository.getPatientDetailsByAgeAndGender(age, gender);
}
}

```

➤ Step 3: Testing From Main Method class

```

package com.dilip.operations;

import java.util.List;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.entity.Patient;

public class PatientApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();
        PatientOperations ops = context.getBean(PatientOperations.class);

        //All Patients
        List<Patient> allPatients = ops.getPatientDetails();
        System.out.println(allPatients);

        //By email Id
        System.out.println("***** with email Id *****");
        Patient patient = ops.getPatientDetailsbyEmailId("laxmi@gmail.com");
        System.out.println(patient);

        //By Age and Gender
        System.out.println("***** PAteints with Age and gender*****");
        List<Patient> patients = ops.getPatientDetailsbyAgeAndGender(31,"MALE");
        System.out.println(patients);
    }
}

```

Output:

```
Hibernate: select * from patient
[Patient [name=Naresh, age=28, gender=MALE, contact=+91372372, emailId=naresh@gmail.com], Patient [name=Rakhi, age=44, gender=MALE, contact=+91372372, emailId=Rakhi@gmail.com], Patient [name=Dilip Singh, age=31, gender=MALE, contact=+9188882828, emailId=dilip@gmail.com], Patient [name=Suresh, age=28, gender=MALE, contact=+91372372, emailId=suresh@gmail.com], Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com], Patient [name=Anusha, age=31, gender=FEMALE, contact=+9188882828, emailId=anusha@gmail.com]]
***** with email Id *****
Hibernate: select * from patient where emailid=?
Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com]
***** Patients with Age and gender*****
Hibernate: select * from patient where age=? and gender=?
[Patient [name=Dilip Singh, age=31, gender=MALE, contact=+9188882828, emailId=dilip@gmail.com]]
```

By using Named Query Parameters:

We can also pass values of method parameters to the query using named parameters i.e. we are providing these using the **@Param** annotation inside our repository method declaration. Each parameter annotated with **@Param** must have a value string matching the corresponding **JPQL or SQL** query parameter name. A query with named parameters is easier to read and is less error-prone in case the query needs to be refactored.

```
@Query(value = "select * from flipkart_users where city=:cityName and pincode=:pincode", nativeQuery = true)
List<Users> getUsersByCityAndPincode(@Param("cityName") String city, @Param("pincode") String pincode);
```

NOTE: In JPQL also, we can use index and named Query parameters.

Requirement: Insert Patient Data

Step 1: Define Method Inside Repository with Native Query:

```
package com.dilip.repository;

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

```

import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import com.dilip.entity.Patient;

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {

    //adding Patient Details
    @Transactional
    @Modifying
    @Query(value = "INSERT INTO patient VALUES(:emailId,:age,:contact,:gender,:name)",
    nativeQuery = true)
    public void addPAatient( @Param("name") String name,
                            @Param("emailId") String email,
                            @Param("age") int age,
                            @Param("contact") String mobile,
                            @Param("gender") String gender );
}

```

Step 2: Call Above Method from DB Operations Class

```

package com.dilip.operations;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

    //add Pateint
    public void addPAatient(String name, String email, int age, String mobile, String gender) {
        repository.addPAatient(name, email, age, mobile, gender);
    }
}

```

Step 3: Test it From Main Method class.

```

package com.dilip.operations;

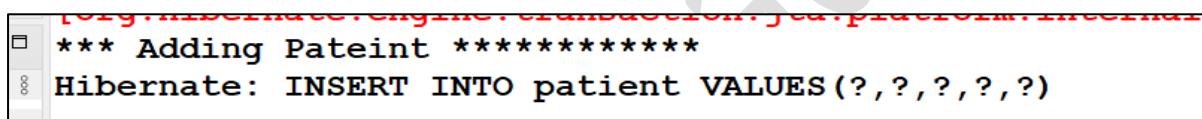
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class PatientApplication {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();
        PatientOperations ops = context.getBean(PatientOperations.class);
        //add Patient
        System.out.println("**** Adding Patient *****");
        ops.addPatient("Rakhi", "Rakhi@gmail.com", 44, "+91372372", "MALE");
    }
}

```

Output:



```

*** Adding Patient *****
Hibernate: INSERT INTO patient VALUES (?, ?, ?, ?, ?)

```

In above We executed DML Query, So it means some Modification will happen finally in Database Tables data. In Spring JPA, for DML Queries like `insert`, `update` and `delete` provided mandatory annotations `@Transactional` and `@Modifying`. We should declare these annotations while executing DML Queries.

@Transactional:

Package : `org.springframework.transaction.annotation.Transactional`

In Spring Framework, the `@Transactional` annotation is used to indicate that a method, or all methods within a class, should be executed within a transaction context. Transactions are used to ensure data integrity and consistency in applications that involve database operations. Specifically, when used with Spring Data JPA, the `@Transactional` annotation plays a significant role in managing transactions around JPA (Java Persistence API) operations.

Describes a transaction attribute on an individual method or on a class. When this annotation is declared at the class level, it applies as a default to all methods of the declaring class and its subclasses. If no custom rollback rules are configured in this annotation, the transaction will roll back on `RuntimeException` and `Error` but not on checked exceptions.

@Modifying:

The **@Modifying** annotation in Spring JPA is used to indicate that a method is a modifying query, which means that it will update, delete, or insert data in the database. This annotation is used in conjunction with the **@Query** annotation to specify the query that the method will execute.

The **@Modifying** annotation is a powerful tool that can be used to update, delete, and insert data in the database. It is often used in conjunction with the **@Transactional** annotation to ensure that the data is updated or deleted in a safe and consistent manner.

Here are some of the benefits of using the **@Modifying** annotation:

- It makes it easy to update, delete, and insert data in the database.
- It can be used in conjunction with the **@Transactional** annotation to ensure that the data is updated or deleted in a safe and consistent manner.
- It can be used to optimize performance by batching updates and deletes.

If you are developing an application that needs to update, delete, or insert data in the database, I highly recommend using the **@Modifying** annotation. It is a powerful tool that can help you to improve the performance and reliability of your application.

JPQL Queries Execution:

Examples for executing JPQL Query's. Here We will not use **nativeQuery** attribute means by default **false** value. Then Spring JPA considers **@Query** Value as JPQL Query.

Requirement:

- Fetch All Patients
- Fetch All Patients Names
- Fetch All Male Patients Names

Step1: Define Repository Methods along with JPQL Queries.

```

package com.dilip.repository;

import java.util.List;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;

@Component
public interface PatientRepository extends CrudRepository<Patient, String> {

    //JPQL Queries
    //Fetch All Patients
}

```

```

@Query(value="Select p from Patient p")
public List<Patient> getAllPatients();

//Fetch All Patients Names
@Query(value="Select p.name from Patient p")
public List<String> getAllPatientsNames();

//Fetch All Male Patients Names
@Query(value="Select p from Patient p where gender=?1")
public List<Patient> getPatientsByGender(String gender);
}

```

Step 2: Call Above Methods From DB Operations class.

```

package com.dilip.operations;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.dilip.entity.Patient;
import com.dilip.repository.PatientRepository;

@Component
public class PatientOperations {

    @Autowired
    PatientRepository repository;

    // JPQL
    // All Patients
    public List<Patient> getAllpatients() {
        return repository.getAllPatients();
    }

    // All Patients Names
    public List<String> getAllpatientsNames() {
        return repository.getAllPatientsNames();
    }

    // All Patients Names
    public List<Patient> getAllpatientsByGender(String gender) {
        return repository.getPatientsByGender(gender);
    }
}

```

Step 3: Test it From Main Method class.

```

package com.dilip.operations;

```

```

import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class PatientApplication {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.scan("com.*");
        context.refresh();
        PatientOperations ops = context.getBean(PatientOperations.class);
        System.out.println("===== > All Patients Details ");
        System.out.println(ops.getAllpatients());
        System.out.println("===== > All Patients Names ");
        System.out.println(ops.getAllpatientsNames());
        System.out.println("===== > All MALE Patients Details ");
        System.out.println(ops.getAllpatientsByGender("MALE"));
    }
}

```

Output:

```

<terminated> PatientApplication (1) [Java Application] D:\softwares\clipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (10-Aug-2023, 9:43:55 am - 9:44
===== > All Patients Details
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0
[Patient [name=Naresh, age=28, gender=MALE, contact=+91372372, emailId=naresh@gmail.com], Patient [name=Rakhi, age=44, gender=MALE, contact=+91372372, emailId=Rakhi@gmail.com], Patient [name=Dilip Singh, age=31, gender=MALE, contact=+9188882828, emailId=dilip@gmail.com], Patient [name=Suresh, age=28, gender=MALE, contact=+91372372, emailId=suresh@gmail.com], Patient [name=Laxmi, age=28, gender=FEMALE, contact=+91372372, emailId=laxmi@gmail.com], Patient [name=Anusha, age=31, gender=FEMALE, contact=+9188882828, emailId=anusha@gmail.com]]
===== > All Patients Names
Hibernate: select p1_0.name from Patient p1_0
[Naresh, Rakhi, Dilip Singh, Suresh, Laxmi, Anusha]
===== > All MALE Patients Details
Hibernate: select p1_0.emailId,p1_0.age,p1_0.contact,p1_0.gender,p1_0.name from Patient p1_0 where p1_0.gender=?
[Patient [name=Naresh, age=28, gender=MALE, contact=+91372372, emailId=naresh@gmail.com], Patient [name=Rakhi, age=44, gender=MALE, contact=+91372372, emailId=Rakhi@gmail.com], Patient [name=Dilip Singh, age=31, gender=MALE, contact=+9188882828, emailId=dilip@gmail.com], Patient [name=Suresh, age=28, gender=MALE, contact=+91372372, emailId=suresh@gmail.com]]

```

Internally JPA Translates JPQL queries to Actual Database SQL Queries and finally those queries will be executed. We can see those queries in Console Messages.

JPQL Query Guidelines

JPQL queries follow a set of rules that define how they are parsed and executed. These rules are defined in the JPA specification. Here are some of the key rules of JPQL:

- The SELECT clause: The SELECT clause specifies the entities that will be returned by the query.
- The FROM clause: The FROM clause specifies the entities that the query will be executed against.
- The WHERE clause: The WHERE clause specifies the conditions that the entities must meet in order to be included in the results of the query.
- The GROUP BY clause: The GROUP BY clause specifies the columns that the results of the query will be grouped by.
- The HAVING clause: The HAVING clause specifies the conditions that the groups must meet in order to be included in the results of the query.
- The ORDER BY clause: The ORDER BY clause specifies the order in which the results of the query will be returned.

Here are some additional things to keep in mind when writing JPQL queries:

- JPQL queries are case-insensitive. This means that you can use the names of entities and columns in either upper or lower case.
- JPQL queries can use parameters. Parameters are variables that can be used in the query to represent values that are not known at the time the query is written.

@GeneratedValue Annotation:

In **Java Persistence API (JPA)**, the **@GeneratedValue** annotation is used to specify how primary key values for database entities should be generated. This annotation is typically used in conjunction with the **@Id** annotation, which marks a field or property as the primary key of an entity class. The **@GeneratedValue** annotation provides options for automatically generating primary key values when inserting records into a database table.

When you annotate a field with **@GeneratedValue**, you're telling Spring Boot to automatically generate unique values for that field.

Here are some of the key attributes of the **@GeneratedValue** annotation:

strategy:

This attribute specifies the generation strategy for primary key values. This is used to specify how to auto-generate the field values. There are five possible values for the strategy element on the **GeneratedValue** annotation: **IDENTITY**, **AUTO**, **TABLE**, **SEQUENCE** and **UUID**. These five values are available in the enum, **GeneratorType**.

1. **GenerationType.AUTO:** This is the default strategy. The JPA provider selects the most appropriate strategy based on the database and its capabilities. Assign the field a generated value, leaving the details to the JPA vendor. Tells JPA to pick the strategy that is preferred by the used database platform.

The preferred strategies are **IDENTITY** for MySQL, SQLite and MsSQL and **SEQUENCE** for Oracle and PostgreSQL. This strategy provides full portability.

2. **GenerationType.IDENTITY:** The primary key value is generated by the database system itself (e.g., auto-increment in MySQL or identity columns in SQL Server, SERIAL in PostgreSQL).
3. **GenerationType.SEQUENCE:** The primary key value is generated using a database sequence. Tells JPA to use a database sequence for ID generation. This strategy does currently not provide full portability. Sequences are supported by Oracle and PostgreSQL. When this value is used then **generator** filed is mandatory to specify the generator.
4. **GenerationType.TABLE:** The primary key value is generated using a database table to maintain unique key values. Tells JPA to use a separate table for ID generation. This strategy provides full portability. When this value is used then **generator** filed is mandatory to specify the generator.
5. **GenerationType.UUID:** Jakarta EE 10 now adds the GenerationType for a **UUID**, so that we can use Universally Unique Identifiers (UUIDs) as the primary key values. Using the GenerationType.UUID strategy, This is the easiest way to generate **UUID** values. Simply annotate the primary key field with the **@GeneratedValue** annotation and set the strategy attribute to **GenerationType.UUID**. The persistence provider will automatically generate a UUID value for the primary key column.

NOTE: Here We are working with Oracle Database. Sometimes Different Databases will exhibit different functionalities w.r.to different Generated Strategies.

Examples For All Strategies:

GenerationType.AUTO:

In JPA, the **GenerationType.AUTO** strategy is the default strategy for generating primary key values. It instructs the persistence provider to choose the most appropriate strategy for generating primary key values based on the underlying database and configuration. This typically maps to either **GenerationType.IDENTITY** or **GenerationType.SEQUENCE**, depending on database capabilities.

When to Use GenerationType.AUTO?

The **GenerationType.AUTO** strategy is a convenient choice for most applications because it eliminates the need to explicitly specify a generation strategy of primary key values. It is particularly useful when you are using a database that supports both **GenerationType.IDENTITY** and **GenerationType.SEQUENCE**, as the persistence provider will automatically select the most efficient strategy for your database.

However, there are some cases where we may want to explicitly specify a generation strategy. For example, if you need to ensure that primary key values are generated sequentially, you should use the **GenerationType.SEQUENCE** strategy. Or, if you need to use

a custom generator, you should specify the name of the generator using the generator attribute of the **@GeneratedValue** annotation.

Benefits of GenerationType.AUTO

- **Convenience:** It eliminates the need to explicitly specify a generation strategy.
- **Automatic selection:** It selects the most appropriate strategy for the underlying database.
- **Compatibility:** It is compatible with a wide range of databases.

Limitations of GenerationType.AUTO

- **Lack of control:** It may not be the most efficient strategy for all databases.
- **Potential for performance issues:** If the persistence provider selects the wrong strategy, it could lead to performance issues.

Overall, the **GenerationType.AUTO** strategy is a good default choice for generating primary key values in JPA applications. However, you should be aware of its limitations and consider explicitly specifying a generation strategy if you have specific requirements.

Overall, the **GenerationType.AUTO** strategy is a good default choice for generating primary key values in JPA applications. However, you should be aware of its limitations and consider explicitly specifying a generation strategy if you have specific requirements.

- **Create Spring Data JPA project**
- **Now Create An Entity class with @GeneratedValue column : Patient.java**

```
package com.tek.teacher.data;

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

@Entity
@Table(name = "patient_details")
public class Patient {

    @Id
    @Column(name = "patient_id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long pateintId;

    @Column(name = "patient_name")
}
```

```

private String name;

@Column(name = "patient_age")
private int age;

public long getPateintId() {
    return pateintId;
}

public void setPateintId(long pateintId) {
    this.pateintId = pateintId;
}

public String getName() {
    return name;
}

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

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}
}

```

- Now Try to Persist Data with above entity class. Create a Repository.

```

package com.tek.teacher.data;

import org.springframework.data.jpa.repository.JpaRepository;

public interface PatientRepository extends JpaRepository<Patient, Long> {
}

```

- Now Create Entity Object and try to execute. Here we are not passing pateintId value to Entity Object.

```

package com.tek.teacher.data;

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

@Component
public class PatientOperations {
}

```

```

@Autowired
PatientRepository repository;

public void addPatient() {

    // patientId : Not Passing value as part of Entity Object
    Patient patient = new Patient();
    patient.setAge(44);
    patient.setName("naresh Singh");
    repository.save(patient);
}
}

```

- Call/Execute above method for persisting data.

```

package com.tek.teacher.data;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringJpaGeneartedvalueApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(JPAConfiguration.class);
        context.refresh();
        PatientOperations ops = context.getBean(PatientOperations.class);
        ops.addPatient();
    }
}

```

Result : Please Observe in Console Logs, How Spring JPA created values of Primary Key Column of Patient table.

```

Hibernate: create table patient_details (patient_id number(19,0) not null, patient_age number(10,0),
patient_name varchar2(255 char), primary key (patient_id))
Hibernate: create sequence patient_details_seq start with 1 increment by 50
2023-11-09T18:58:22.530+05:30  INFO 20572 --- [           main] j.LocalContainerEntityManagerFactoryBean :
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-09T18:58:22.843+05:30  INFO 20572 --- [           main] t.SpringBootJpaGeneartedvalueApplication :
Started SpringBootJpaGeneartedvalueApplication in 98.978 seconds (process running for 99.614)
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)
2023-11-09T18:58:23.112+05:30  INFO 20572 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean :

```

i.e. Spring JPA created by a new **sequence** for the column **PATIENT_ID** values.

Verify Data Inside Table now.

```
select * from patient_details;
```

Query Result x

SQL | All Rows Fetched: 1 in 0.004 seconds

PATIENT_ID	PATIENT_NAME	PATIENT_NAME
1	44	Dilip Singh

- Now Whenever we are persisting data in **patient_details**, **patient_id** column values will be inserted by executing sequence automatically.
- Execute Logic one more time.

```
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)
2023-11-09T19:24:08.977+05:30 INFO 4812 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean :
```

Table Result :

```
select * from patient_details;
```

Query Result x

SQL | All Rows Fetched: 2 in 0.002 seconds

PATIENT_ID	PATIENT_NAME	PATIENT_NAME
1	44	Dilip Singh
2	44	Dilip Singh

GenerationType.IDENTITY:

This strategy will help us to generate the primary key value by the database itself using the auto-increment or identity of column option. It relies on the database's native support for generating unique values.

- Entity class with IDENTITY Type:

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

    @Id
    @Column(name = "patient_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long pateintId;

    @Column(name = "patient_name")
    private String name;

    @Column(name = "patient_age")
    private int age;

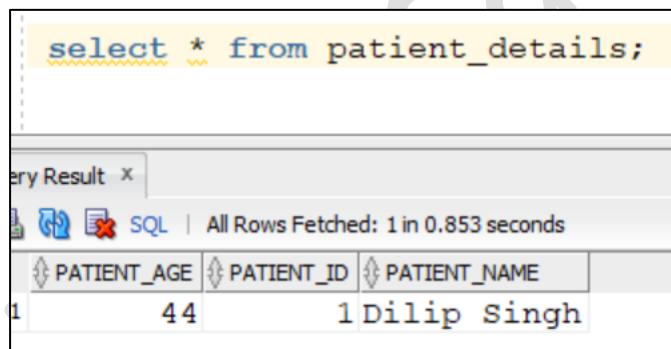
    public long getPateintId() {
        return pateintId;
    }
    public void setPateintId(long pateintId) {
        this.pateintId = pateintId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

- Now Execute Logic again to Persist Data in table.
- If we observe console logs JPA created table as follows

```
Hibernate: create table patient_details (patient_age number(10,0), patient_id number(19,0) generated as identity, patient_name varchar2(255 char), primary key (patient_id))
2023-11-09T19:51:16.768+05:30 INFO 15408 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-09T19:51:17.240+05:30 INFO 15408 --- [           main] t.SpringBootJpaGeneartedvalueApplication : Started SpringBootJpaGeneartedvalueApplication in 12.662 seconds (process running for 13.861)
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,default)
2023-11-09T19:51:19.387+05:30 INFO 15408 --- [onShutdownHook] i.LocalContainerEntityManagerFactoryBean :
```

- For Column **patient_id** of table **patient_details**, JPA created **IDENTITY** column.
- Oracle introduced a way that allows you to define an identity column for a table, which is similar to the **AUTO_INCREMENT** column in MySQL or **IDENTITY** column in SQL Server.
- i.e. If we connected to MySQL and used **GenerationType.IDENTITY** in JPA, then JPA will create **AUTO_INCREMENT** column to generate Primary Key Values. Similarly If it is SQL Server , then JPA will create **IDENTITY** column for same scenario.
- The **identity** column is very useful for the surrogate primary key column. When you insert a new row into the identity column, Oracle auto-generates and insert a sequential value into the column.

Table Data :



The screenshot shows the Oracle SQL Developer interface. A SQL statement is entered in the SQL worksheet:

```
select * from patient_details;
```

The results are displayed in the Result window:

	PATIENT_ID	PATIENT_NAME
1	44	Dilip Singh

SQL | All Rows Fetched: 1 in 0.853 seconds

Execute Again :

```
2023-11-09T19:59:52.976+05:30 INFO 20280 --- [           main] t.SpringBootJpaGeneartedvalueApplication : Started SpringBootJpaGeneartedvalueApplication in 12.662 seconds (process running for 13.861)
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,default)
2023-11-09T19:59:53.424+05:30 INFO 20280 --- [onShutdownHook] i.LocalContainerEntityManagerFactoryBean :
```

Table Result:

Result		
PATIENT_AGE	PATIENT_ID	PATIENT_NAME
44	1	Dilip Singh
44	2	Dilip Singh

GenerationType.SEQUENCE:

GenerationType.SEQUENCE is used to specify that a database sequence should be used for generating the primary key value of the entity.

➤ Entity class with **IDENTITY** Type:

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

    @Id
    @Column(name = "patient_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long pateintId;

    @Column(name = "patient_name")
    private String name;

    @Column(name = "patient_age")
    private int age;

    public long getPateintId() {
        return pateintId;
    }

    public void setPateintId(long pateintId) {
        this.pateintId = pateintId;
    }

    public String getName() {
```

```

        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}

```

- Now Execute Logic to Persist Data in table.
- If we observe console logs JPA created table as follows

```

Hibernate: create sequence patient_details_seq start with 1 increment by 50
Hibernate: create table patient_details (patient_age number(10,0), patient_id number(19,0) not null, patient_name
varchar2(255 char), primary key (patient_id))
2023-11-10T18:26:17.446+05:30  INFO 14180 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized
JPA EntityManagerFactory for persistence unit 'default'
2023-11-10T18:26:17.733+05:30  INFO 14180 --- [           main] t.SpringBootJpaGeneartedvalueApplication : Started
SpringBootJpaGeneartedvalueApplication in 93.094 seconds (process running for 94.269)
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?, ?, ?)
2023-11-10T18:26:17.733+05:30  INFO 14180 --- [           main] o.h.t.l.i.TableGeneratorListener : Cleaning up

```

- JPA created a Sequence to generate unique values. By executing this sequence, values are inserted into Patient table for primary key column.
- Now when we are persisting data inside Patient table by Entity Object, always same sequence will be used for next value.

Table Data :

select * from patient_details		
Result		
PATIENT_ID	PATIENT_NAME	PATIENT_AGE
1	Dilip Singh	44

- Execute Logic for saving data again inside Patient table.
- Primary Key column value is generated from sequence and same supplied to Entity Level.

```
2023-11-10T10:49:12.000+05:30 INFO 4404 [main] t.SpringBootApp.main() - main() -> SpringBootApp.main()
Hibernate: select patient_details_seq.nextval from dual
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?, ?, ?)
2023-11-10T10:49:12.000+05:30 INFO 4404 [main] t.SpringBootApp.main() - main() -> SpringBootApp.main()
```

Table Data:

The screenshot shows the Oracle SQL Developer interface. A SQL query window contains the command: `select * from patient_details`. Below it, a 'Query Result' window displays the output of the query. The table has three columns: PATIENT_ID, PATIENT_NAME, and PATIENT_AGE. The data shows two rows: one for Dilip Singh (Patient ID 1, Age 44) and one for Suresh Singh (Patient ID 2, Age 33).

PATIENT_ID	PATIENT_NAME	PATIENT_AGE
1	Dilip Singh	44
2	Suresh Singh	33

This is how JPA will create a sequence when we defined `@GeneratedValue(strategy = GenerationType.IDENTITY)` with `@Id` column of Entity class.

GenerationType.TABLE:

When we use **GenerationType.TABLE**, the persistence provider uses a separate database table to manage the primary key values. A table is created in the database specifically for tracking and generating primary key values for each entity.

This strategy is less common than some others (like **GenerationType.IDENTITY** or **GenerationType.SEQUENCE**) but can be useful in certain scenarios, especially when dealing with databases that don't support identity columns or sequences.

Example With GenerationType.TABLE:

- Create a Entity class and it's ID property should be aligned with **GenerationType.TABLE**

```
package com.tek.teacher.data;
import jakarta.persistence.Column;
```

```

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private long pateintId;

    @Column(name = "patient_name")
    private String name;

    @Column(name = "patient_age")
    private int age;

    public long getPateintId() {
        return pateintId;
    }

    public void setPateintId(long pateintId) {
        this.pateintId = pateintId;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

- In this example, the **@GeneratedValue** annotation is used with the **GenerationType.TABLE** strategy to indicate that the **id** field of **Entity** should have its values generated using a separate table.
- Now Try to insert data inside table **patient_details** from JPA Repository.

```
package com.tek.teacher.data;
```

```

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

@Component
public class PatientOperations {
    @Autowired
    PatientRepository repository;

    public void addPatient() {
        Patient patient = new Patient();
        patient.setAge(33);
        patient.setName("Dilip Singh");
        repository.save(patient);
    }
}

```

- Now execute Logic and try to monitor in application console logs, how JPA working with **GenerationType.TABLE** strategy of **GeneratedValue**.

```

Hibernate: create table hibernate_sequences (next_val number(19,0), sequence_name varchar2(255 char) not null,
primary key (sequence_name))
Hibernate: insert into hibernate_sequences(sequence_name, next_val) values ('default',0)
Hibernate: create table patient_details (patient_age number(10,0), patient_id number(19,0) not null,
patient_name varchar2(255 char), primary key (patient_id))
2023-11-15T17:11:56.341+05:30  INFO 22708 --- [           main] j.LocalContainerEntityManagerFactoryBean :
Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-15T17:11:56.603+05:30  INFO 22708 --- [           main] t.SpringBootJpaGenearatedvalueApplication :
Started SpringBootJpaGenearatedvalueApplication in 4.255 seconds (process running for 4.855)
Hibernate: select tbl.next_val from hibernate_sequences tbl where tbl.sequence_name=? for update
Hibernate: update hibernate_sequences set next_val=? where next_val=? and sequence_name=?
Hibernate: insert into patient_details (patient_age,patient_name,patient_id) values (?,?,?)

```

- Now, JPA created a separate database table to manage primary key values of Entity Table as follows.

```

create table hibernate_sequences (next_val number(19,0),
sequence_name varchar2(255 char) not null, primary key
sequence_name)

```

- This Table will be used for generating next Primary key values of our Table **patient_details**

Table Data:

```
select * from patient_details;
```

Query Result x

SQL | All Rows Fetched: 1 in 0.809 seconds

PATEINT_ID	PATIENT_AGE	PATIENT_NAME
1	33	Dilip Singh

Primary Key Table: **hibernate_sequences**

HIBERNATE_SEQUENCES	
Data Model Constraints Grants Statistics Triggers	
NEXT_VAL	SEQUENCE_NAME
50	default

- Execute Same Logic Again and Again With New Patients Data:

Table Data : Primary key Values are Generated by default with help of table.

```
select * from patient_details;
```

Query Result x

SQL | All Rows Fetched: 6 in 0.003 seconds

PATEINT_ID	PATIENT_AGE	PATIENT_NAME
1	33	Dilip Singh
2	25	tek teacher
3	52	tek teacher
4	102	tek teacher
5	152	tek teacher
6	202	tek teacher

Note:

Keep in mind that the choice of the generation strategy depends on the database you are using and its capabilities. Some databases support identity columns (**GenerationType.IDENTITY**), sequences (**GenerationType.SEQUENCE**), or a combination of strategies. The **GenerationType.TABLE** strategy is generally used when other strategies are not suitable for the underlying database.

GenerationType.UUID:

In Spring Data JPA, **UUIDs** can be used as the primary key type for entities. Indicates that the persistence provider must assign primary keys for the entity by generating Universally Unique Identifiers. These are non-numerical values like alphanumeric type.

What is UUID?

A **UUID**, or Universally Unique Identifier, is a 128-bit identifier standardized by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE). It is also known as a GUID (Globally Unique Identifier). A UUID is typically expressed as a string of 32 hexadecimal digits, displayed in five groups separated by hyphens, in the form 8-4-4-4-12 for a total of 36 characters (32 alphanumeric characters and 4 hyphens).

For example: **a3335f0a-82ef-47ae-a7e1-1d5c5c3bc4e4**

UUIDs are widely used in various computing systems and scenarios where unique identification is crucial. They are commonly used in databases, distributed systems, and scenarios where it's important to generate unique identifiers without centralized coordination.

In the context of databases and Spring JPA, using UUIDs as primary keys for entities is a way to generate unique identifiers that can be more suitable for distributed systems compared to traditional auto-incremented numeric keys.

Note: we have a pre-defined class in JAVA, `java.util.UUID` for dealing with UUID values. We can consider as String value as well.

Create Entity Class with UUID Generator Strategy:

```
package com.tek.teacher.data;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "patient_details")
public class Patient {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private String patientId;

    @Column(name = "patient_name")
    private String name;

    @Column(name = "patient_age")
}
```

```

private int age;

public String getPateintId() {
    return pateintId;
}

public void setPateintId(String pateintId) {
    this.pateintId = pateintId;
}

public String getName() {
    return name;
}

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

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}
}

```

- Now execute Logic and try to monitor in application console logs, how JPA working with **GenerationType.UUID** strategy of **GeneratedValue**.
- **Execute Same Logic Again and Again With New Patients Data:**

Table Data: Primary key **UUID** type Values are Generated and persisted in table.

SELECT * FROM PATIENT_DETAILS		
ts 1 ×		
PATEINTID	PATIENT_AGE	PATIENT_NAME
a3335f0a-82ef-47ae-a7e1-1d5c5c3bc4e4	44	naresh Singh
003e48fd-c1cd-40a1-9aae-1b828efc1397	33	Dilip Singh

Sorting and Pagination in JPA:

Sorting: Sorting is a fundamental operation in data processing that involves arranging a collection of items or data elements in a specific order. The primary purpose of sorting is to make it easier to search for, retrieve, and work with data. Sorting can be done in ascending (from smallest to largest) or descending (from largest to smallest) order, depending on the requirements.

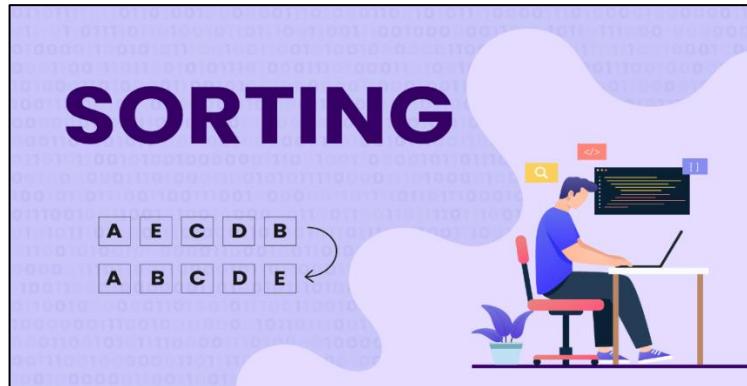


Table and Data:

Screenshot of Oracle SQL Developer showing the 'AMAZON_ORDERS' table.

AMAZON_ORDERS

ORDER_ID	AMOUNT	CITY	EMAIL	GENDER	NO_OF_ITEMS	PINCODE
1	3323	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
2	3232	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
3	9988	Chennai	ramesh@gmail.com	MALE	3	600088
4	3344	Hyderbad	dilip@gmail.com	MALE	4	500072
5	1234	4000 Hyderbad	naresh@gmail.com	MALE	2	500072
6	5566	6000 Hyderbad	naresh@gmail.com	MALE	8	500070
7	8888	10000 Chennai	suresh@gmail.com	MALE	10	600099
8	3636	44444 Chennai	ramesh@gmail.com	MALE	3	600088

⊕ **Requirement:** Get Details by Email Id with Sorting

- **Create Spring JPA Project**
- **Create Entity Class as Per Database Table.**

```
package com.dilip.dao;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "amazon_orders")
public class AmazonOrders {

    @Id
    @Column(name="order_id")
    private int orderId;
```

```

    @Column(name = "no_of_items")
    private int noOfItems;

    @Column(name = "amount")
    private double amount;

    @Column(name="email" )
    private String email;

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

    @Column(name="city")
    private String city;

    @Column(name="gender")
    private String gender;

    //Setter & Getter Methods
}

```

➤ Now Create A Repository.

```

package com.dilip.dao;

import org.springframework.data.jpa.repository.JpaRepository;

public interface AmazonOrderRepository extends JpaRepository<AmazonOrders, Integer> {
}

```

➤ Now Create a Component Class for Database Operations and Add a Method for Sorting Data

To achieve this requirement, Spring Boot JPA provided few methods in side **JpaRepository**. Inside **JpaRepository**, JPA provided a method **findAll(...)** with different Arguments.

For Sorting Data : **findAll(Sort sort)**

Sort: In Spring Data JPA, you can use the **Sort** class to specify sorting criteria for your query results. The **Sort** class allows you to define sorting orders for one or more attributes of our entity class. we can use it when working with repository methods to sort query results.

Here's how you can use the **Sort** class in Spring Data JPA:

```

package com.dilip.dao;

```

```

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

    // getting order Details in ascending order
    public void loadDataByEmailIdWithSorting() {
        List<AmazonOrders> allOrders = repository.findAll(Sort.by("email"));
        System.out.println(allOrders);
    }
}

```

Note: we have to pass Entity class Property name as part of **by(..)** method, which is related to database table column.

➤ Now Execute above Logic

```

package com.dilip;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.dao.OrdersOperations;

public class SpringJpaSortingPaginationApplication {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(JPAConfiguration.class);
        context.refresh();
        OrdersOperations ops = context.getBean(OrdersOperations.class);
        ops.loadDataByEmailIdWithSorting();
    }
}

```

Output: Table Records are Sorted by email ID and got List of Entity Objects

[

AmazonOrders(orderId=3344, noOfItems=4, amount=3000.0, email=dilip@gmail.com,
pincode=500072, city=Hyderabad, gender=MALE),

AmazonOrders(orderId=3232, noOfItems=4, amount=63643.0, email=laxmi@gmail.com,
pincode=500070, city=Hyderabad, gender=FEMALE),

AmazonOrders(orderId=3323, noOfItems=4, amount=63643.0, email=laxmi@gmail.com,
pincode=500070, city=Hyderabad, gender=FEMALE),

AmazonOrders(orderId=1234, noOfItems=2, amount=4000.0, email=naresh@gmail.com,
pincode=500072, city=Hyderabad, gender=MALE),

AmazonOrders(orderId=5566, noOfItems=8, amount=6000.0, email=naresh@gmail.com,
pincode=500070, city=Hyderabad, gender=MALE),

AmazonOrders(orderId=9988, noOfItems=3, amount=44444.0, email=ramesh@gmail.com,
pincode=600088, city=Chennai, gender=MALE),

AmazonOrders(orderId=3636, noOfItems=3, amount=44444.0, email=ramesh@gmail.com,
pincode=600088, city=Chennai, gender=MALE),

AmazonOrders(orderId=8888, noOfItems=10, amount=10000.0, email=suresh@gmail.com,
pincode=600099, city=Chennai, gender=MALE)

]

Requirement: Get Data by sorting with property **noOfItems** of Descending Order.

In Spring Data JPA, you can specify the direction (**ascending** or **descending**) for sorting when using the **Sort** class. The **Sort** class allows you to create sorting orders for one or more attributes of our entity class. To specify the direction, you can use the **Direction enum**.

Here's how you can use the **Direction** enum in Spring Data JPA:

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.data.domain.Sort.Direction;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

    // getting order Details in ascending order of email
}
```

```

public void loadDataByEmailIdWithSorting() {
    List<AmazonOrders> allOrders = repository.findAll(Sort.by("email"));
    System.out.println(allOrders);
}

// Get Data by sorting with property noOfItems of Descending Order
public void loadDataByNoOfItemsWithDescOrder() {
    List<AmazonOrders> allOrders =
        repository.findAll(Sort.by(Direction.DESC, "noOfItems"));
    System.out.println(allOrders);
}

```

Output: We got Entity Objects, by following **noOfItems** property in Descending Order.

```
[
    AmazonOrders(orderId=8888,           noOfItems=10,           amount=10000.0,
                 email=suresh@gmail.com, pincode=600099, city=Chennai, gender=MALE),
    AmazonOrders(orderId=5566,           noOfItems=8,            amount=6000.0,
                 email=naresh@gmail.com, pincode=500070, city=Hyderabad, gender=MALE),
    AmazonOrders(orderId=3232,           noOfItems=4,            amount=63643.0,
                 email=laxmi@gmail.com, pincode=500070, city=Hyderabad, gender=FEMALE),
    AmazonOrders(orderId=3323,           noOfItems=4,            amount=63643.0,
                 email=laxmi@gmail.com, pincode=500070, city=Hyderabad, gender=FEMALE),
    AmazonOrders(orderId=3344,           noOfItems=4,            amount=3000.0,
                 email=dilip@gmail.com, pincode=500072, city=Hyderabad, gender=MALE),
    AmazonOrders(orderId=3636,           noOfItems=3,            amount=44444.0,
                 email=ramesh@gmail.com, pincode=600088, city=Chennai, gender=MALE),
    AmazonOrders(orderId=9988,           noOfItems=3,            amount=44444.0,
                 email=ramesh@gmail.com, pincode=600088, city=Chennai, gender=MALE),
    AmazonOrders(orderId=1234,           noOfItems=2,            amount=4000.0,
                 email=naresh@gmail.com, pincode=500072, city=Hyderabad, gender=MALE)
]
```

Similarly we can get table Data with Sorting Order based on any table column by using Spring Boot JPA. We can sort data with multiple columns as well.

Example: `repository.findAll(Sort.by("email", "noOfItems"));`

Pagination:

Pagination is a technique used in software applications to divide a large set of data or content into smaller, manageable segments called "pages." Each page typically contains a fixed number of items, such as records from a database, search results, or content items in a user interface. Pagination allows users to navigate through the data or content one page at a time, making it easier to browse, consume, and interact with large datasets or content collections.



Key features and concepts related to pagination include:

Page Size: The number of items or records displayed on each page is referred to as the "page size" or "items per page." Common page sizes might be 10, 20, 50, or 100 items per page. The choice of page size depends on usability considerations and the nature of the data.

Page Number: Pagination is typically associated with a page number, starting from 1 and incrementing as users navigate through the data. Users can move forward or backward to view different segments of the dataset.

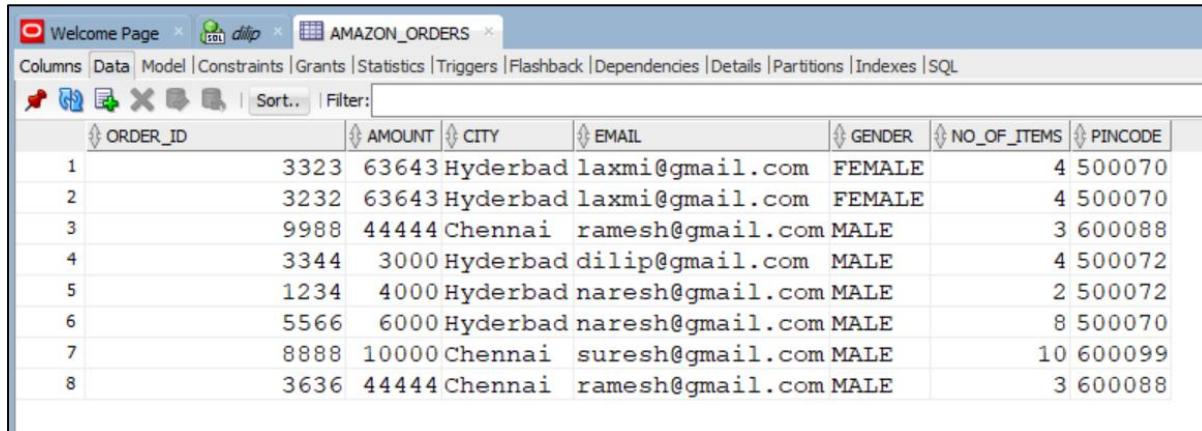
Navigation Controls: Pagination is usually accompanied by navigation controls, such as "Previous" and "Next" buttons or links. These controls allow users to move between pages easily.

Total Number of Pages: The total number of pages in the dataset is determined by dividing the total number of items by the page size. For example, if there are 100 items and the page size is 10, there will be 10 pages.

- Assume a scenario, where we have 200 Records. Each page should get 25 Records, then page number and records are divided as shown below.

Total Records	200
Page 1	1 - 25
Page 2	26 - 50
Page 3	51 - 75
Page 4	76 - 100
Page 5	101 - 125
Page 6	126 - 150
Page 7	151 - 175
Page 8	176 - 200

 **Requirement:** Get first set of Records by default with some size from below Table data.



	ORDER_ID	AMOUNT	CITY	EMAIL	GENDER	NO_OF_ITEMS	PINCODE
1	3323	63643	Hyderabad	laxmi@gmail.com	FEMALE	4	500070
2	3232	63643	Hyderabad	laxmi@gmail.com	FEMALE	4	500070
3	9988	44444	Chennai	ramesh@gmail.com	MALE	3	600088
4	3344	3000	Hyderabad	dilip@gmail.com	MALE	4	500072
5	1234	4000	Hyderabad	naresh@gmail.com	MALE	2	500072
6	5566	6000	Hyderabad	naresh@gmail.com	MALE	8	500070
7	8888	10000	Chennai	suresh@gmail.com	MALE	10	600099
8	3636	44444	Chennai	ramesh@gmail.com	MALE	3	600088

In Spring Data JPA, **Pageable** is an interface that allows you to paginate query results easily. It provides a way to specify the page number, the number of items per page (page size), and optional sorting criteria for your query results. This is particularly useful when you need to retrieve a large set of data from a database and want to split it into smaller pages.

Here's how you can use **Pageable** in Spring JPA:

Pageable.ofSize(int size)) : size is, number of records to be loaded.

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

    public void getFirstPageData() {
        List<AmazonOrders> orders =
            repository.findAll(Pageable.ofSize(2)).getContent();
        System.out.println(orders);
    }
}
```

- Now execute above logic

```
package com.dilip;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dilip.dao.OrdersOperations;

public class SpringJpaSortingPaginationApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext();
        context.register(JPAConfiguration.class);
        context.refresh();
        OrdersOperations ops = context.getBean(OrdersOperations.class);
        ops.getFirstPageData();
    }
}
```

Output: We got first 2 records of table.

```
[AmazonOrders(orderId=3323, noOfItems=4, amount=63643.0, email=laxmi@gmail.com, pincode=500070, city=Hyderbad, gender=FEMALE), AmazonOrders(orderId=3232, noOfItems=4, amount=63643.0, email=laxmi@gmail.com, pincode=500070, city=Hyderbad, gender=FEMALE),]
```

➡ **Requirement:** Get 2nd page of Records with some size of records i.e. **3 Records**.

ORDER_ID	AMOUNT	CITY	EMAIL	GENDER	NO_OF_ITEMS	PINCODE
3323	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
3232	63643	Hyderbad	laxmi@gmail.com	FEMALE	4	500070
9988	44444	Chennai	ramesh@gmail.com	MALE	3	600088
3344	3000	Hyderbad	dilip@gmail.com	MALE	4	500072
1234	4000	Hyderbad	naresh@gmail.com	MALE	2	500072
5566	6000	Hyderbad	naresh@gmail.com	MALE	8	500070
8888	10000	Chennai	suresh@gmail.com	MALE	10	600099
3636	44444	Chennai	ramesh@gmail.com	MALE	3	600088

Here we will use **PageRequest** class which provides pre-defined methods, where we can provide page Numbers and number of records.

Method: **PageRequest.of(int page, int size);**

Note: In JPA, Page Index always Starts with **0** i.e. Page number 2 representing 1 index.

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

    public void getRecordsByPageIdAndNoOfRecords(int pageId, int noOfRecords) {
        Pageable pageable = PageRequest.of(pageId, noOfRecords);
        List<AmazonOrders> allOrders =
            repository.findAll(pageable).getContent();
        System.out.println(allOrders);
    }
}
```

➤ Now execute above logic

```
package com.dilip;

import com.dilip.dao.OrdersOperations;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringJpaSortingPaginationApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(JPAConfiguration.class);
        context.refresh();
        OrdersOperations ops = context.getBean(OrdersOperations.class);
        ops.getRecordsByPageIdAndNoOfRecords(1,3);
    }
}
```

Output: From our Table data, we got **4-6 Records** which is representing **2nd Page** of Data.

[

```
AmazonOrders(orderId=3344, noOfItems=4, amount=3000.0, email=dilip@gmail.com,
pincode=500072, city=Hyderabad, gender=MALE),

AmazonOrders(orderId=1234, noOfItems=2, amount=4000.0,
email=naresh@gmail.com, pincode=500072, city=Hyderabad, gender=MALE),

AmazonOrders(orderId=5566, noOfItems=8, amount=6000.0,
email=naresh@gmail.com, pincode=500070, city=Hyderabad, gender=MALE)
]
```

➤ **Requirement:** Pagination with Sorting:

Get **2nd page** of Records with some size of records i.e. **3 Records** along with Sorting by **noOfItems** column in Descending Order.

```
package com.dilip.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.data.domain.Sort.Direction;
import org.springframework.stereotype.Component;

@Component
public class OrdersOperations {

    @Autowired
    AmazonOrderRepository repository;

    public void getDataByPaginationAndSorting(int pagId, int noOfRecords) {

        List<AmazonOrders> allOrders =
            repository.findAll(PageRequest.of(pagId, noOfRecords,
                Sort.by(Direction.DESC, "noOfItems"))).getContent();

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

➤ **Execute Above Logic**

```
package com.dilip;

import com.dilip.dao.OrdersOperations;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

```
public class SpringJpaSortingPaginationApplication {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(JPAConfiguration.class);
        context.refresh();
        OrdersOperations ops = context.getBean(OrdersOperations.class);
        ops.getDataByPaginationAndSorting(1,3);
    }
}
```

Output: We got Entity Objects with Sorting by `noOfItems` column, and we got 2nd page set of records.

```
[
AmazonOrders(orderId=3344, noOfItems=4, amount=3000.0,
email=dilip@gmail.com, pincode=500072, city=Hyderabad, gender=MALE),

AmazonOrders(orderId=3232, noOfItems=4, amount=63643.0,
email=laxmi@gmail.com, pincode=500070, city=Hyderabad, gender=FEMALE),

AmazonOrders(orderId=9988, noOfItems=3, amount=44444.0,
email=ramesh@gmail.com, pincode=600088, city=Chennai, gender=MALE)
]
```

Relationship Mapping with JPA:

In Java Persistence API (JPA), relationship mappings refer to the way in which entities are connected or associated with each other in a relational database. JPA provides annotations that allow you to define these relationships in your Java code, and these annotations influence how the corresponding tables and foreign key constraints are created in the underlying database.

Here are JPA relationship mappings:

1. One-to-One (1:1) Relationship:

- An entity A is associated with only one entity B, and vice versa.

Example: An Employee has one Address.

2. One-to-Many (1:N) Relationship:

- An entity A is associated with many instances of entity B, but each instance of entity B is associated with only one instance of entity A.

Example: An Employee has more than one Address.

3. Many-to-One (N:1) Relationship:

- The reverse of a One-to-Many relationship. Many instances of entity A can be associated with one instance of entity B.

Example: Many Employees belong to one Role.

4. Many-to-Many (N:N) Relationship:

- Many instances of entity A are associated with many instances of entity B, and vice versa.

Example: An Employee can enrol in many Roles, and a Role can have many Employees.

5. Unidirectional and Bidirectional Relationships:

- In a unidirectional relationship, one entity knows about the other, but the other entity is not aware of the relationship. In a bidirectional relationship, both entities are aware of the relationship.

Example: A Post has many Comments (unidirectional) vs. A Comment belongs to a Post, and a Post has many Comments (bidirectional).

To map these relationships, JPA uses annotations such as `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`, and additional annotations like `@JoinColumn` and `@JoinTable` to define the details of the database schema.

Cascading and Cascade Types:

What Is Cascading?

Entity relationships often depend on the existence of another entity, for example the **Employee–Address relationship**. Without the **Employee**, the **Address** entity doesn't have any meaning of its own. When we delete the **Employee** entity, our **Address** entity should also get deleted.

When we perform some action on the target entity, the same action will be applied to the associated entity. Cascading is the way to achieve this. To enable this behaviour, we had used the `CascadeType` attribute with mappings. To establish a dependency between related entities, JPA provides `jakarta.persistence.CascadeType` enumerated types that define the cascade operations. These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many.

JPA Cascade Types:

JPA allows us to propagate the state transition from a parent entity to the associated child entity. For this purpose, JPA defines various cascade types under `CascadeType` Enum.

-  **PERSIST**
-  **MERGE**
-  **REMOVE**
-  **REFRESH**
-  **DETACH**
-  **ALL**

Type	Description
PERSIST	if the parent entity is persisted then all its related entity will also be persisted.
MERGE	if the parent entity is merged then all its related entity will also be merged.
DETACH	if the parent entity is detached then all its related entity will also be detached.
REFRESH	if the parent entity is refreshed then all its related entity will also be refreshed.
REMOVE	if the parent entity is removed then all its related entity will also be removed.
ALL	In this case, all the above cascade operations can be applied to the entities related to parent entity. The value is equivalent to cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}

CascadeType.PERSIST:

CascadeType.PERSIST is a cascading type in JPA that specifies that the create (or persist) operation should be cascaded from the parent entity to the child entities.

When **CascadeType.PERSIST** is used, any new child entities associated with a parent entity will be automatically persisted when the parent entity is persisted. However, updates or deletions made to the parent entity will not be cascaded to the child entities.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.PERSIST**, any new Order entities associated with a Customer entity will be persisted when the Customer entity is persisted. However, if you update or delete a Customer entity, any associated Order entities will not be automatically updated or deleted.

CascadeType.MERGE:

CascadeType.MERGE is a cascading type in JPA that specifies that the update (or merge) operation should be cascaded from the parent entity to the child entities.

When **CascadeType.MERGE** is used, any changes made to a detached parent entity (i.e., an entity that is not currently managed by the persistence context) will be automatically merged with its associated child entities when the parent entity is merged back into the persistence context. However, new child entities that are not already associated with the parent entity will not be automatically persisted.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.MERGE**, any changes made to a detached Customer entity (such as changes made in a different session or transaction) will be automatically merged with its associated Order entities when the Customer entity is merged back into the persistence context.

CascadeType.REMOVE:

CascadeType.REMOVE is a cascading type in JPA that specifies that the delete operation should be cascaded from the parent entity to the child entities.

When **CascadeType.REMOVE** is used, any child entities associated with a parent entity will be automatically deleted when the parent entity is deleted. However, updates or modifications made to the parent entity will not be cascaded to the child entities.

For example, consider a scenario where we have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.REMOVE**, any Order entities associated with a Customer entity will be automatically deleted when the Customer entity is deleted.

CascadeType.REFRESH:

CascadeType.REFRESH is a cascading type in JPA that specifies that the refresh operation should be cascaded from the parent entity to the child entities.

When **CascadeType.REFRESH** is used, any child entities associated with a parent entity will be automatically refreshed when the parent entity is refreshed. This means that the latest state of the child entities will be loaded from the database and any changes made to the child entities will be discarded.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.REFRESH**, any associated Order entities will be automatically refreshed when the Customer entity is refreshed.

CascadeType.DETACH:

CascadeType.DETACH is a cascading type in JPA that specifies that the detach operation should be cascaded from the parent entity to the child entities.

When **CascadeType.DETACH** is used, any child entities associated with a parent entity will be automatically detached when the parent entity is detached. This means that the child entities will become detached from the persistence context and their state will no longer be managed.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.DETACH**, any associated Order entities will be automatically detached when the Customer entity is detached.

CascadeType.ALL:

CascadeType.ALL is a cascading type in JPA that specifies that all state transitions (create, update, delete, and refresh) should be cascaded from the parent entity to the child entities.

When **CascadeType.ALL** is used, and any operation performed on the parent entity will be automatically propagated to all child entities. This means that if you persist, update, or delete a parent entity, all child entities associated with it will also be persisted, updated, or deleted accordingly.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using **CascadeType.ALL**, any operation performed on the Customer entity (such as persist, merge, remove, or refresh) will also be propagated to all associated Order entities.

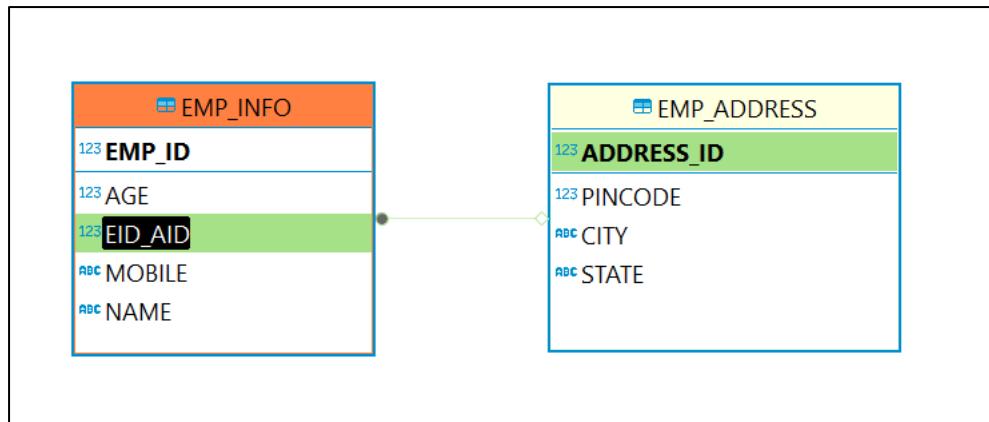
One-to-One (1:1) Relationship:

A one-to-one relationship is a relationship where a record in one table is associated with exactly one record in another table i.e. a record in one entity (table) is associated with exactly one record in another entity (table).

@OneToOne annotation is used to define a one-to-one relationship between two entities. This means that one instance of an entity is associated with exactly one instance of another entity, and vice versa. In database terms, this often translates to a shared primary key or a unique constraint on one of the tables. One way, we can implement a one-to-one relationship in a database is to add a new column and make it as foreign key.

Here's a example of how to use **@OneToOne** in JPA:

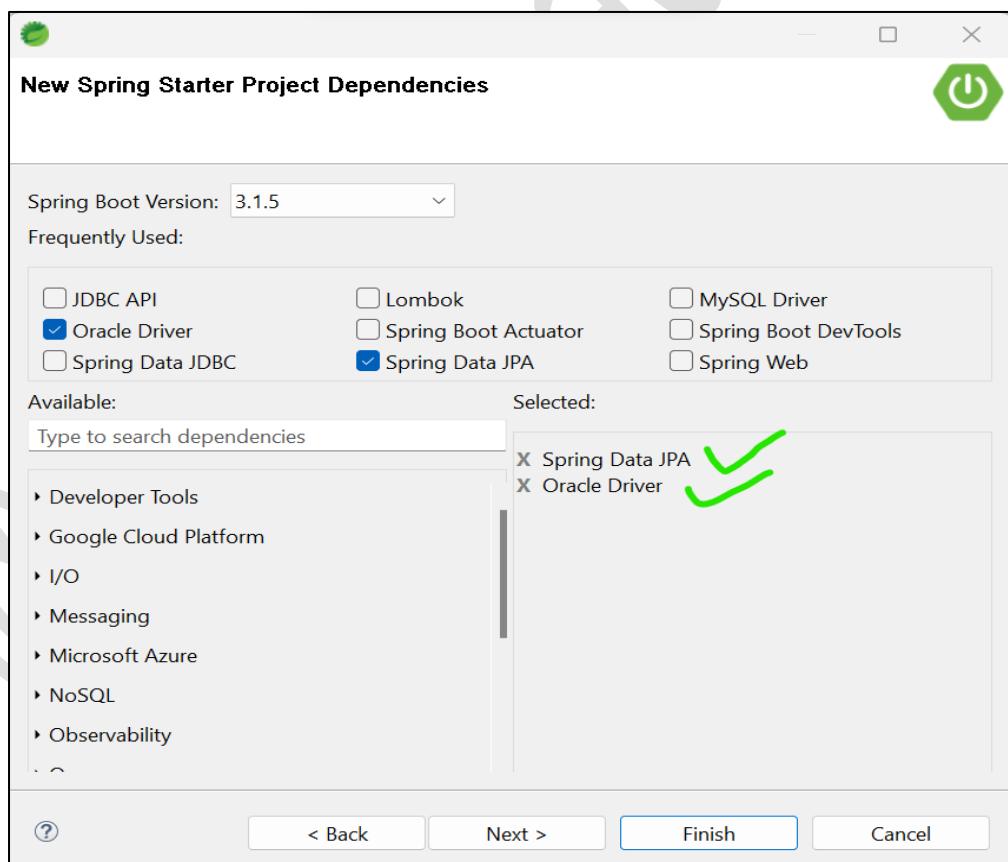
Requirement: Add Employee and Address Details with One to One Relationship as shown in below Entity Relationship Diagram.



- From Above ER diagram, **EID_AID** column of **EMP_INFO** table representing a Foreign Key relationship with another table **EMP_ADDRESS** primary key column **ADDRESS_ID**.

Implementation:

- Now Create a Spring Boot JPA Project.



- After Project Creation, Please add database details inside **application.properties** file

```

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

```
spring.datasource.password=dilip  
  
spring.jpa.show-sql=true  
spring.jpa.hibernate.ddl-auto=update
```

- Now Create Entity Classes as details shown ER diagram.

Entity Class : Address.java

```
package com.dilip.entity;  
  
import jakarta.persistence.Column;  
import jakarta.persistence.Entity;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.Id;  
import jakarta.persistence.Table;  
  
@Entity  
@Table(name = "emp_address")  
public class Address {  
  
    @Id  
    @Column  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private int addressId;  
  
    @Column  
    private String city;  
  
    @Column  
    private int pincode;  
  
    @Column  
    private String state;  
  
    public int getAddressId() {  
        return addressId;  
    }  
    public void setAddressId(int addressId) {  
        this.addressId = addressId;  
    }  
    public String getCity() {  
        return city;  
    }  
    public void setCity(String city) {  
        this.city = city;  
    }
```

```
    }
    public int getPincode() {
        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
}
```

Entity Class : Employye.java

```
package com.dilip.entity;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_info")
public class Employye {

    @Id
    @Column
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long empId;

    @Column
    private String name;

    @Column
    private int age;

    @Column
    private String mobile;
```

```

@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "eid_aid")
Address address;

public Long getEmpId() {
    return empId;
}
public void setEmpId(Long empId) {
    this.empId = empId;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}
public String getMobile() {
    return mobile;
}
public void setMobile(String mobile) {
    this.mobile = mobile;
}
public Address getAddress() {
    return address;
}
public void setAddress(Address address) {
    this.address = address;
}
}

```

@JoinColumn:

In JPA, the **@JoinColumn** annotation is used to define the column that will be used to join two entities in an association. It is typically used in one-to-many, many-to-one, and many-to-many relationships.

The **@JoinColumn** annotation has several attributes that can be used to customize the join column mapping. Some of the most common attributes are:

- **name:** Specifies the name of the join column in the owning entity's table.

- **referencedColumnName**: Specifies the name of the column in the referenced entity's table.
- **insertable**: Whether the join column should be included in INSERT statements.
- **updatable**: Whether the join column should be included in UPDATE statements.
- **nullable**: Whether the join column can be null.
- **unique**: Whether the join column is a unique key.

From above example, the **@JoinColumn** annotation is used to define a foreign key column named **eid_aid** in the **emp_info** entity's table. This column will reference the primary key column (**addressId**) in the **emp_address** entity's table.

- Create JPA Repository's for Employye Entity Classes.

Repository Interface : EmployeeRepository.java

```
package com.dilip.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.dilip.entity.Employee;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long>{

}
```

- Define a Service Class to perform Database Operations with Entity.

EmployeeService.java

```
package com.dilip.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.dilip.entity.Address;
import com.dilip.entity.Employee;
import com.dilip.repository.EmployeeRepository;

@Component
public class EmployeeService {

    @Autowired
    EmployeeRepository repository;

    public String addNewEmployee() {

        //Create Address Entity Object
    }
}
```

```

        Address address = new Address();
        address.setCity("Hyderabad");
        address.setState("Telangana");
        address.setPincode(500072);

        //Create Employye Entity Object
        Employye employye = new Employye();
        employye.setName("Dilip Singh");
        employye.setAge(30);
        employye.setMobile("+91-8826111377");

        // Setting Address Entity Object inside Employye
        employye.setAddress(address);

        Employye resultEntity = repository.save(employye);
        return "Employye Data Submitted Successfully.
                Please Find Your Employye Id "+resultEntity.getEmpld();
    }
}

```

- Execute above Service class methods to do Database Operations.

```

package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.service.EmployyeService;

@SpringBootApplication
public class SpringBootJpaMappingsApplication {

    public static void main(String[] args) {
        ApplicationContext context =
            SpringApplication.run(SpringBootJpaMappingsApplication.class, args);

        EmployyeService service = context.getBean(EmployyeService.class);
        String result = service.addNewEmployye();
        System.out.println(result);
    }
}

```

Results:

If we observe application logs, JPA created both Tables with Foreign Key Relationship.

```

Hibernate: create table emp_address (address_id number(10,0) generated as identity, pincode
number(10,0), city varchar2(255 char), state varchar2(255 char), primary key (address_id))
Hibernate: create table emp_info (age number(10,0), eid_aid number(10,0) unique, emp_id
number(19,0) generated as identity, mobile varchar2(255 char), name varchar2(255 char),
primary key (emp_id))
Hibernate: alter table emp_info add constraint FK6kyk4gkrokuu1we20ha2pnmkn foreign key
(eid_aid) references emp_address
2023-11-20T19:38:39.243+05:30  INFO 20036 --- [           main]
j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for
persistence unit 'default'
2023-11-20T19:38:39.601+05:30  INFO 20036 --- [           main]
c.d.SpringBootJpaMappingsApplication : Started SpringBootJpaMappingsApplication in 4.05
seconds (process running for 4.689)
Hibernate: insert into emp_address (city,pincode,state,address_id) values (?, ?, ?, default)
Hibernate: insert into emp_info (eid_aid,age,mobile,name,emp_id) values (?, ?, ?, ?, default)
Employee Data Submitted Successfully. Please Find Your Employee Id 1

```

Data in Tables:

SELECT * FROM EMP_INFO					
	EMP_ID	NAME	MOBILE	AGE	EID_AID
1	1	Dilip Singh	+91-8826111377	30	1

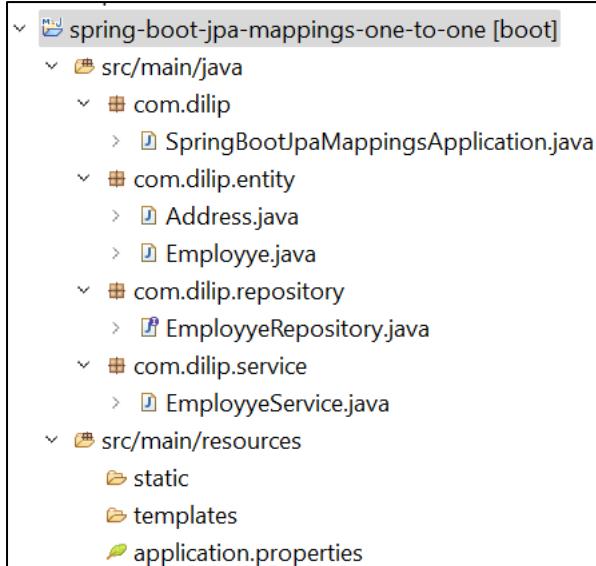
SELECT * FROM EMP_ADDRESS				
	ADDRESS_ID	PINCODE	CITY	STATE
1	1	500,072	Hyderabad	Telangana

- Similarly Try to Insert Few More Records.

SELECT * FROM EMP_INFO ei					
	EMP_ID	NAME	AGE	EID_AID	MOBILE
1	1	Dilip Singh	30	1	+91-8826111377
2	2	Naresh	44	2	+91-8125262702
3	3	Anusha	22	3	+1772727727

SELECT * FROM EMP_ADDRESS ea				
	ADDRESS_ID	PINCODE	CITY	STATE
1	2	400,000	Banglore	Karnatka
2	3	300,555	texas	USA
3	1	500,072	Hyderabad	Telangana

Project Directory Structure:



Delete Owner Entity Records:

We are enabled Cascade Propagation level as ALL i.e. `@OneToOne(cascade = CascadeType.ALL)`, in Entity Relationship between Employee and Address Tables. So in such case, Whatever DB operations we are performing at Owner Entity **Employee** side, same action should be performed on target entity Address level.

Delete One Employee Details From Employee Table:

- Add a method in our Service class : **EmployyeService.java**

```
package com.dilip.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.dilip.repository.EmployyeRepository;

@Component
public class EmployyeService {

    @Autowired
    EmployyeRepository repository;

    //Passing only Employee ID
    public void deleteEmployye(Long emplId) {
        repository.deleteById(emplId);
    }
}
```

- Execute Above Method From Main Method:

```
package com.dilip;
```

```

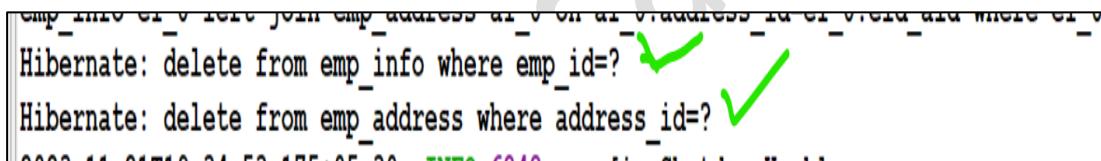
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.service.EmployeeService;

@SpringBootApplication
public class SpringBootJpaMappingsApplication {

    public static void main(String[] args) {
        ApplicationContext context =
            SpringApplication.run(SpringBootJpaMappingsApplication.class, args);
        EmployeeService service = context.getBean(EmployeeService.class);
        service.deleteEmployee(Long.valueOf(2));
    }
}

```

Results: From followed Console Logs, if we observe Employee details deleted and as well as associated Address Details also deleted from both tables. So Delete Operation cascaded from source to target table.



Hibernate: delete from emp_info where emp_id=? ✓
 Hibernate: delete from emp_address where address_id=? ✓

Data in Tables : Employee 2 details deleted from both table.

SELECT * FROM EMP_INFO ei					SELECT * FROM EMP_ADDRESS				
1	X	*	FROM EMP_INFO ei	Enter a SQL expression to filter results (use Ctrl+F)	X	*	FROM EMP_ADDRESS	Enter a SQL expression to filter results (use Ctrl+F)	
EMP_ID ↑	ABC NAME ↓	ABC MOBILE ↓	123 AGE ↓	123 EID_AID ↓	ADDRESS_ID ↑	123 PINCODE ↓	ABC CITY ↓	ABC STATE ↓	
1	Dilip Singh	+91-8826111377	30	1	1	500,072	Hyderabad	Telangana	
3	Anusha	+1772727727	22	3	3	300,555	texas	USA	

This is how we can enable One to One relationship between tables by using Spring JPA.

One-to-Many (1:N) Relationship:

One-to-Many relationship represents a situation where one entity is associated with multiple instances of another entity. This is a common scenario in database design, where one record in a table is related to multiple records in another table. In JPA, we can model One-to-Many relationships using annotations.

One-to-many relationships are very common in databases and are often used to model real-world relationships. For example, a Employee can have many Addresses, but each Address only have one Employee. Or, a customer can have many orders, but each order can only belong to one customer. **One-to-many** relationships are often implemented in databases using foreign keys. A foreign key is a column in a table that references the primary key of another table.

Requirement: Define One to Many Relationship between Employee and Addresses.

- Define Entity Classes with One to Many Relationships.

Entity class : Employee.java

```
package com.dilip.entity;

import java.util.List;
import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_info")
public class Employee {

    @Id
    @Column
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long empld;

    @Column
    private String name;

    @Column
    private int age;

    @Column
    private String mobile;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "eid_aid")
    List<Address> address;
```

```

public Long getEmpld() {
    return empld;
}
public void setEmpld(Long empld) {
    this.empld = empld;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}
public String getMobile() {
    return mobile;
}
public void setMobile(String mobile) {
    this.mobile = mobile;
}
public List<Address> getAddress() {
    return address;
}
public void setAddress(List<Address> address) {
    this.address = address;
}
}

```

- **@OneToMany** is used to define a One-to-Many relationship. The **cascade** attribute is used to specify operations that should be cascaded to the target of the association (e.g., if you delete an **Employee**, delete all associated **Address** entities).

Entity class : Address.java

```

package com.dilip.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;

```

```
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_address")
public class Address {

    @Id
    @Column
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int addressId;

    @Column
    private String city;
    @Column
    private int pincode;
    @Column
    private String state;

    public int getAddressId() {
        return addressId;
    }
    public void setAddressId(int addressId) {
        this.addressId = addressId;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public int getPincode() {
        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
}
```

- Now Add More Addresses with One Employee Object.

```
package com.dilip.service;

import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.dilip.entity.Address;
import com.dilip.entity.Employee;
import com.dilip.repository.EmployeeRepository;

@Service
public class EmployeeService {

    @Autowired
    EmployeeRepository repository;

    public String addNewEmployee() {

        List<Address> empAddresses = new ArrayList<>();
        Address home = new Address();
        home.setCity("HYDERBAD");
        home.setPincode(500072);
        home.setState("TELANGANA");
        empAddresses.add(home);

        Address office = new Address();
        office.setCity("BANGLORE");
        office.setPincode(400072);
        office.setState("KARNATAKA");
        empAddresses.add(office);

        Employee employee = new Employee();
        employee.setName("Dilip Singh");
        employee.setMobile("+918826111377");
        employee.setAge(30);
        employee.setAddress(empAddresses);

        employee = repository.save(employee);

        return "Employee Data Submitted Successfully. Please Find Employee Id :" + employee.getEmplId();
    }

    public void deleteEmployee(String emplId) {
        repository.deleteById(Long.valueOf(emplId));
    }
}
```

- In Above, we are adding 2 Address instance with one Employee Entity Instance.
- Execute the logic of **addNewEmployee()**

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.service.EmployeeService;

@SpringBootApplication
public class SbootJpaOneToManyMapping {

    public static void main(String[] args) {
        ApplicationContext context =
            SpringApplication.run(SbootJpaOneToManyMapping.class, args);

        EmployeeService employeeService = context.getBean(EmployeeService.class);
        employeeService.addNewEmployee();

    }
}
```

- If we observe Application console logs, both Employee and Address tables are created with Foreign Key relationship.
- One Record inside Employee and Two Records inside Address Tables are inserted.

* FROM EMP_INFO ei <input type="text"/> Enter a SQL expression to filter results				
AGE	EMP_ID	MOBILE	NAME	
30	1	+918826111377	Dilip Singh	

* FROM EMP_ADDRESS ea <input type="text"/> Enter a SQL expression to filter results (use Ctrl+Space)				
ADDRESS_ID	PINCODE	EID_AID	CITY	STATE
1	500,072	1	HYDERBAD	TELANGANA
2	400,072	1	BANGLORE	KARNATAKA

Many-to-One (N:1) Relationship:

The **@ManyToOne** mapping is used to represent a many-to-one relationship between entities in JPA Hibernate. It is used when multiple instances of one entity are

associated with a single instance of another entity. Let's explain this mapping with a clear explanation and a code example.

Consider two entities: **Employee** and **Department**. Each employee belongs to a single department, but a department can have multiple employees. This scenario represents a many-to-one relationship, where **multiple employees (many)** can be associated with a **single department (one)**.

Here's how you can implement the **@ManyToOne** mapping:

In the **Employee** entity, we have defined a department field with the **@ManyToOne** annotation. This indicates that **multiple employees can be associated with a single department**. The **@JoinColumn** annotation is used to specify the foreign key column in the **employees** table that references the **departments** table. In this case, the foreign key column is **dept_id**.

To understand the usage, let's consider an example:

Creating Employee Entity Class: Employee.java

```
package com.dilip.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "emp_info")
public class Employee {

    @Id
    @Column
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long empId;

    @Column
    private String name;

    @Column
}
```

```

private int age;

@Column
private String mobile;

@ManyToOne
@JoinColumn(name = "dept_id")
Department department;

public Long getEmpld() {
    return empld;
}
public void setEmpld(Long empld) {
    this.empld = empld;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}
public String getMobile() {
    return mobile;
}
public void setMobile(String mobile) {
    this.mobile = mobile;
}
public Department getDepartment() {
    return department;
}
public void setDepartment(Department department) {
    this.department = department;
}
}

```

Creating Department Entity Class: Department.java

```

package com.dilip.entity;

import jakarta.persistence.Entity;

```

```

import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "departments")
public class Department {

    @Id
    private Long id;

    private String name;

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

```

- Now Create JPA Repositories for both Employee and Department Entity's .

```

package com.dilip.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.dilip.entity.Employee;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
}

```

```

package com.dilip.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.dilip.entity.Department;

@Repository

```

```
public interface DepartmentRepository extends JpaRepository<Department, Long> {  
}
```

- Now Create Data like More Employees of same Department.

```
package com.dilip.service;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import com.dilip.entity.Department;  
import com.dilip.entity.Employee;  
import com.dilip.repository.DepartmentRepository;  
import com.dilip.repository.EmployeeRepository;  
  
@Service  
public class EmployeeService {  
  
    @Autowired  
    EmployeeRepository employeeRepository;  
  
    @Autowired  
    DepartmentRepository departmentRepository;  
  
    public void addNewEmployee() {  
  
        Department department = new Department();  
        department.setId(1L);  
        department.setName("HR");  
  
        Employee employee1 = new Employee();  
        employee1.setMobile("+918826111377");  
        employee1.setName("Dilip Singh");  
        employee1.setAge(30);  
        employee1.setDepartment(department);  
  
        Employee employee2 = new Employee();  
        employee2.setName("Naresh");  
        employee2.setMobile("+918125262702");  
        employee2.setAge(31);  
        employee2.setDepartment(department);  
  
        departmentRepository.save(department);  
        employeeRepository.save(employee1);  
        employeeRepository.save(employee2);  
    }  
}
```

{}

In this example, we create a **Department** object representing the HR department. Then, we create two **Employee** objects and associate them with the HR department using the **setDepartment()** method. When you persist these entities, it will automatically handle the foreign key relationship between the **employees** and **departments** tables. The **dept_id** column in the employees table will store the appropriate department ID.

This way, you can establish a many-to-one relationship between entities using the **@ManyToOne** mapping in JPA Hibernate.

- Now Execute Above Logic.

```
package com.dilip;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import com.dilip.service.EmployeeService;

@SpringBootApplication
public class SpringJpaApplication {
    public static void main(String[] args) {
        ApplicationContext context =
            SpringApplication.run(SpringJpaApplication.class, args);
        EmployeeService employeeService =
            context.getBean(EmployeeService.class);
        employeeService.addNewEmployee();
    }
}
```

- Now Observe Console logs and see how tables created and data getting inserted.

```
Hibernate: create table departments (id number(19,0) not null, name varchar2(255 char), primary key (id))
Hibernate: create table emp_info (age number(10,0), dept_id number(19,0), emp_id number(19,0) generated as identity,
mobile varchar2(255 char), name varchar2(255 char), primary key (emp_id))
Hibernate: alter table emp_info add constraint FKci9jim8lqk0nt9dkaisvik4mv foreign key (dept_id) references departments
2023-12-29T18:54:50.860+05:30  INFO 19896 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized
JPA EntityManagerFactory for persistence unit 'default'
2023-12-29T18:54:51.187+05:30  INFO 19896 --- [           main] com.dilip.SpringJpaNotesApplication      : Started
SpringJpaNotesApplication in 4.103 seconds (process running for 4.714)
Hibernate: select d1_0.id,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into departments (name,id) values (?,?)✓
Hibernate: select null,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into emp_info (age,dept_id,mobile,name,emp_id) values (?,?,?,?,?,default)✓
Hibernate: select null,d1_0.name from departments d1_0 where d1_0.id=?
Hibernate: insert into emp_info (age,dept_id,mobile,name,emp_id) values (?,?,?,?,?,default)✓
```

- Now Try to insert data of employees with invalid department id i.e. department id is not existed in Department table. Then we will get an exception and data will not be inserted in employee table i.e. when Department Id is existed then only employee data will be inserted because of foreign key relationship.

Many-to-Many (N:N) Relationship:

A many-to-many relationship in the context of databases refers to a relationship between two entities where each record in one entity can be related to multiple records in another entity, and vice versa. This type of relationship is common in relational database design and is typically implemented using an intermediary table, often called a junction or linking table. Here's a simple example to illustrate a many-to-many relationship:

Let's consider two entities: "**Employee**" and "**Role**"

Employee
Id (Primary Key)
name
email

Role:
Id (Primary Key)
name

In a many-to-many relationship, a **Employee** can have multiple **Roles**, and a **Role** can have multiple **Employees**. To represent this relationship, you would introduce a third table, often referred to as a junction table or linking table.

Junction Table: `employee_roles`

employee_id (Foreign Key referencing Students)
role_id (Foreign Key referencing Courses)

Each record in the **employee_roles** table represents a connection between a **Employee** and a **Role**. The combination of Employee ID and Role ID in the Enrolment table creates a unique identifier for each mapping, preventing duplicate entries for the same **Employee-Role** pair.

This setup allows for flexibility and efficiency in querying the database. You can easily find all Roles of an Employee is enrolled or all Employs enrolled in a particular Role by querying the : **employee_roles** table.

In summary, many-to-many relationships are handled by introducing a linking table to manage the associations between entities. This linking table resolves the complexity of directly connecting entities with a many-to-many relationship in a relational database.

- Define **Employee** and **Role** Entity classes as per above solution.

Employee Entity: Employee.java

```
package com.tek.teacher.employye;

import java.util.List;
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.Table;

@Entity
@Table(name = "employees")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long id;

    String name;
    String email;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name = "employee_roles",
               joinColumns = @JoinColumn(name = "employee_id"),
               inverseJoinColumns = @JoinColumn(name = "role_id"))
    List<Role> roles;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
}
```

```

public void setEmail(String email) {
    this.email = email;
}
public List<Role> getRoles() {
    return roles;
}
public void setRoles(List<Role> roles) {
    this.roles = roles;
}
}

```

Role Entity: Role.java

```

package com.tek.teacher.employee;

import java.util.List;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.Table;

@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long id;

    String name;

    @ManyToMany
    List<Employee> employees;

    public List<Employee> getEmployees() {
        return employees;
    }
    public void setEmployees(List<Employee> employees) {
        this.employees = employees;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
}

```

```

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

```

➤ **Create JPA Repository : EmployeeRepository.java**

```

package com.tek.teacher.employee;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long>{
}

```

➤ Now Create a class and Persist Data inside tables.

```

package com.tek.teacher.employee;

import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class EmployeeOperations {

    @Autowired
    EmployeeRepository repository;

    public void addEmployee() {
        Employee e = new Employee();
        e.setEmail("Naresh@gmail.com");
        e.setName("Naresh Singh");

        Role r1 = new Role();
        r1.setName("DEVLOPER");
        Role r2 = new Role();
        r2.setName("USER");

        List<Role> roles = new ArrayList<Role>();

```

```

        roles.add(r1);
        roles.add(r2);
        e.setRoles(roles);
        repository.save(e);
    }
}

```

➤ Now Execute Above Logic

```

Hibernate: create table employee_roles (employee_id number(19,0) not null, role_id number(19,0) not null)
Hibernate: create table employees (id number(19,0) generated as identity, email varchar2(255 char), name
varchar2(255 char), primary key (id))
Hibernate: create table roles (id number(19,0) generated as identity, name varchar2(255 char), primary
key (id))
Hibernate: alter table employee_roles add constraint FK398vvu81xw154mvv3g2eytscn foreign key (role_id)
references roles
Hibernate: alter table employee_roles add constraint FK3uwwaxeiuvcfixgd45etkjmgm foreign key
(employee_id) references employees
2024-01-02T18:52:17.969+05:30  INFO 17312 --- [           main] j.LocalContainerEntityManagerFactoryBean
: Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-01-02T18:52:18.378+05:30  INFO 17312 --- [           main] t.SpringBootJpaGeneartedvalueApplication
: Started SpringBootJpaGeneartedvalueApplication in 4.375 seconds (process running for 4.819)
Hibernate: insert into employees (email,name,id) values (?,?,default)
Hibernate: insert into roles (name,id) values (?,default)
Hibernate: insert into roles (name,id) values (?,default)
Hibernate: insert into employee_roles (employee_id,role_id) values (?,?)
Hibernate: insert into employee_roles (employee_id,role_id) values (?,?)

```

➤ From the above Console Logs, if we observe a joining Table is created along with Employee and Roles. Data is inserted in total 3 tables and **employee_roles** table maintaining the relation between **Employees** and **Roles** of Many to Many Association.

Database Data of 3 Tables:

SELECT * FROM EMPLOYEES e			
	ID	EMAIL	NAME
1	1	Naresh@gmail.com	Naresh Singh
2	2	Dilip@gmail.com	Dilip Singh

SELECT * FROM ROLES r		
	ID	NAME
1	1	DEVLOPER
2	2	USER
3	3	DEVLOPER
4	4	USER
5	5	MANAGER

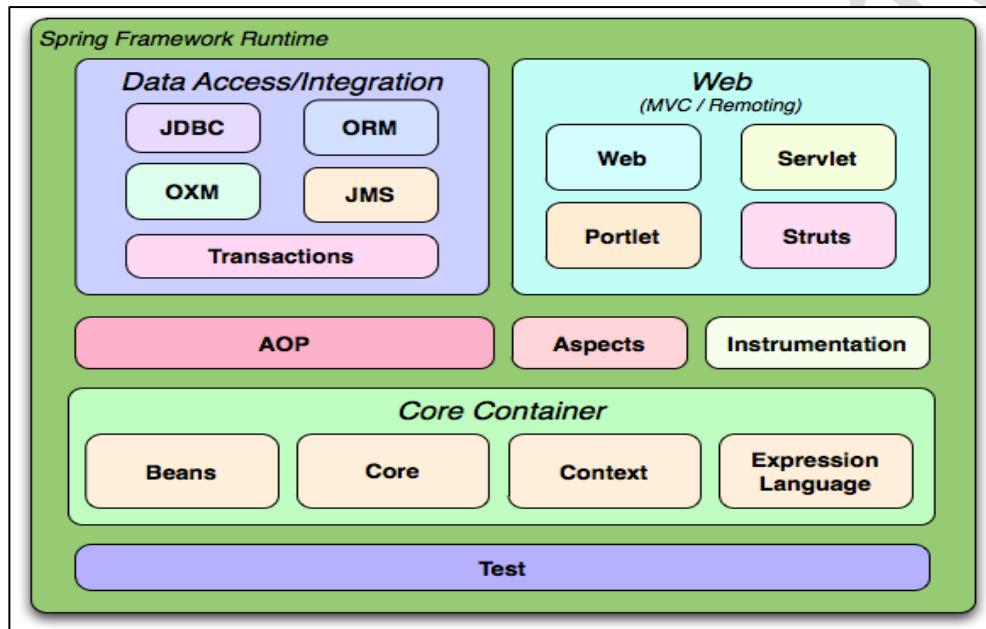
	EMPLOYEE_ID	ROLE_ID
1	1	1
2	1	2
3	2	3
4	2	4
5	2	5

Spring Web/MVC Module

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 is well suited for web application development.

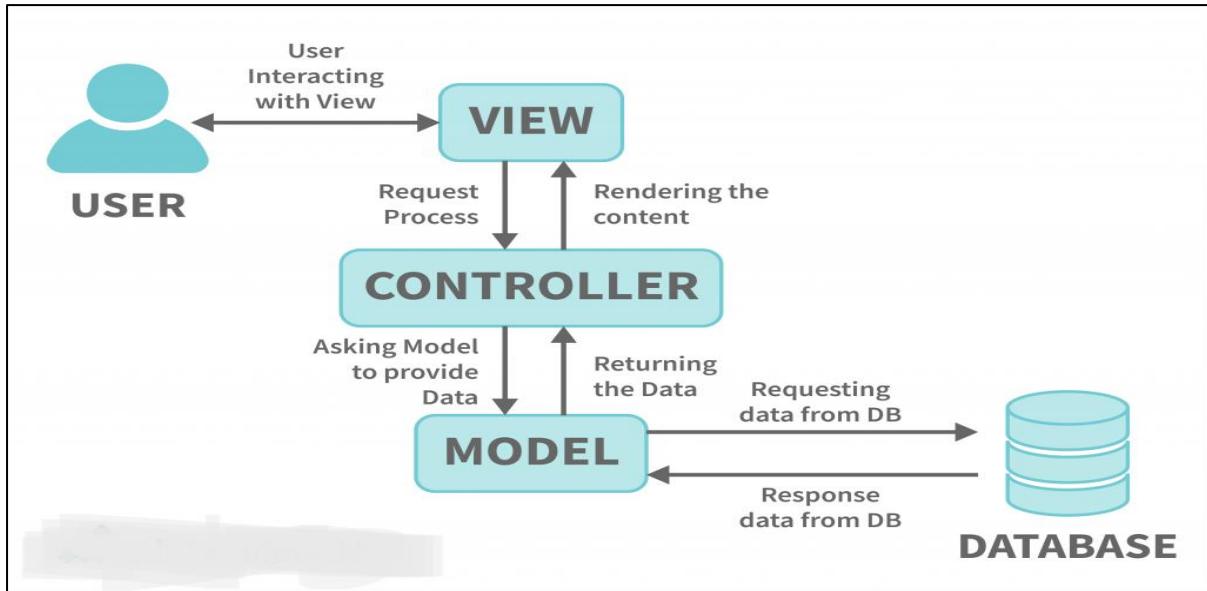


What is MVC?

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:

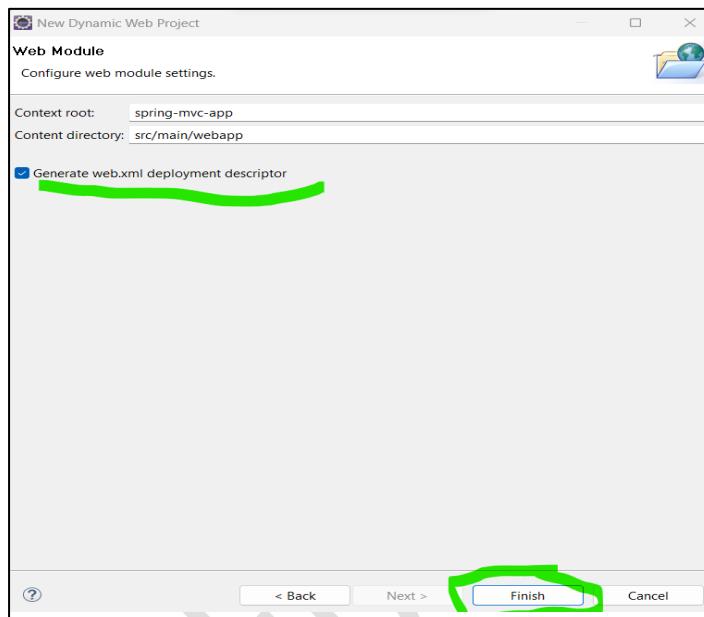
- **Separate roles** - The Spring MVC separates each role, where the model object, controller, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
- **Light-weight** - It uses light-weight servlet container to develop and deploy your application.
- **Powerful Configuration** - 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.

Steps for Creating Spring Web Application:

1. Create Dynamic Web Project in Eclipse

Note: Select Generate web.xml



2. Convert to Maven Project

Right Click On project -> Configure -> Convert to Maven Project

3. Add Dependencies in POM file

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>spring-mvc-app</groupId>
    <artifactId>spring-mvc-app</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
        
```

```
<version>5.3.29</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
</dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.2.3</version>
        </plugin>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <release>17</release>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

4. Create MVC Config Class

```
package com.naresh.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Configuration
@ComponentScan("com.*")
public class MVCConfiguration extends WebMvcConfigurationSupport{

}
```

5. Create Dispatcher Servlet Initialization

```
package com.naresh.config;

import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class SpringWebInitialization
        extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] {MVCConfiguration.class};
    }
    @Override
    protected String[] getServletMappings() {
        String[] allowedURLMapping = {"/*"};
        return allowedURLMapping;
    }
}
```

➤ With this basic spring Web Application Setup is completed.

Now add A Controller for testing our setup : **TestController.java**

```
package com.naresh.controller;

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

@Controller
public class TestController {

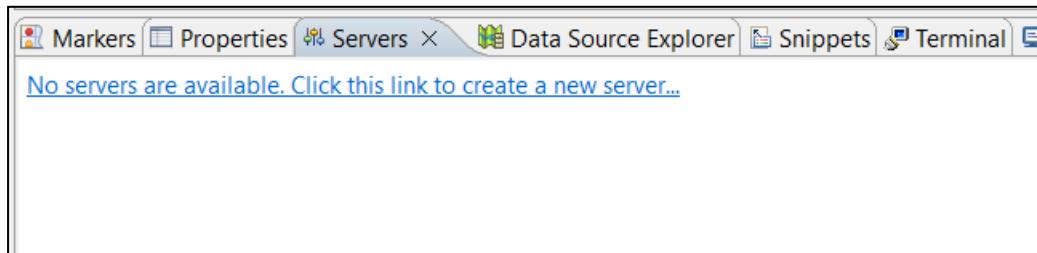
    @RequestMapping("/welcome")
    @ResponseBody
    public String sayHelloToall() {
        return "Welcome to Spring MVC World.";
    }
}
```

For Testing above URI Mapping, We required Web Server because It's web application.

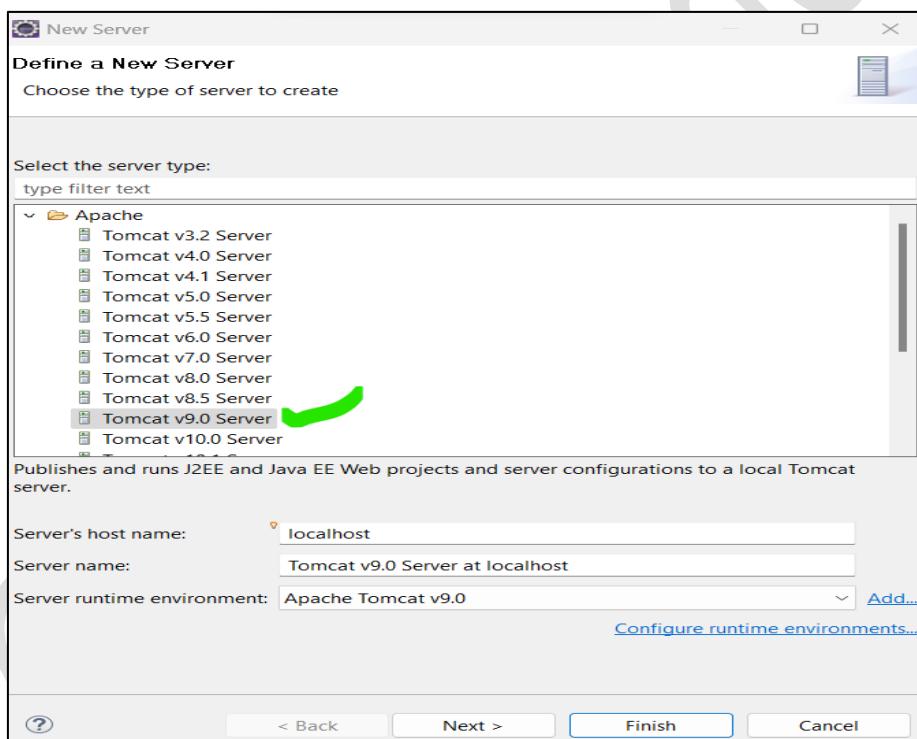
Add Tomcat Server to Eclipse: Download Tomcat and Extract it to some location.

<https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.78/bin/apache-tomcat-9.0.78-windows-x64.zip>

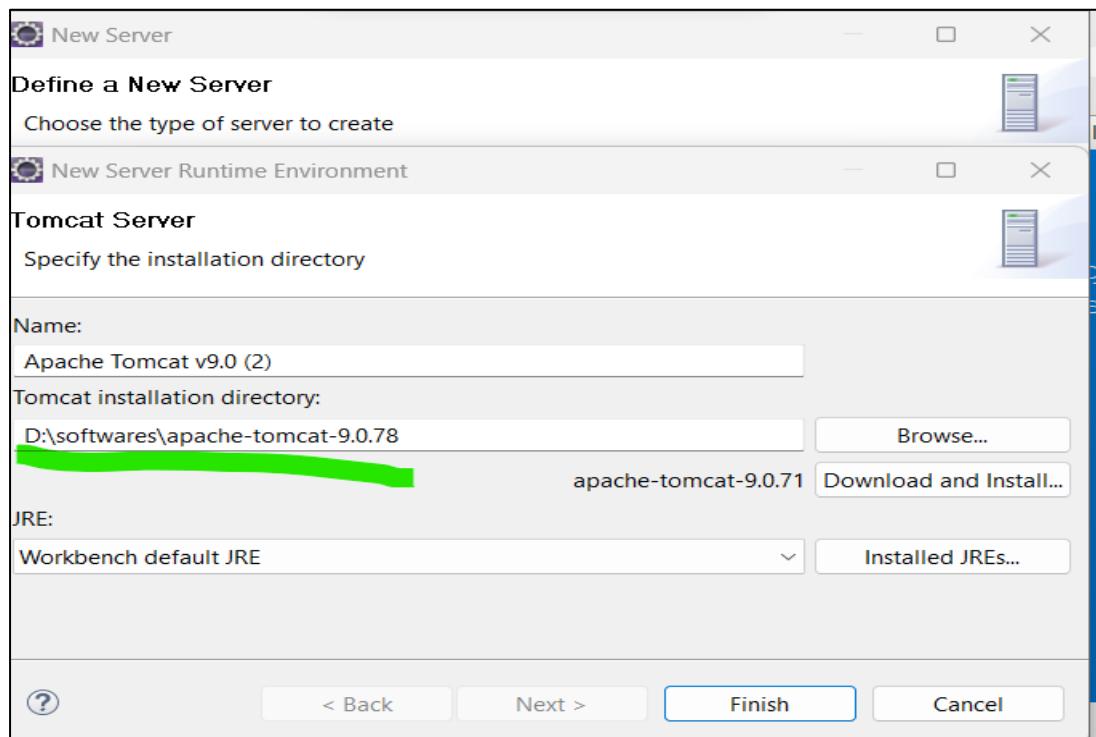
Now From Eclipse Servers Console:



Click on Create New Server



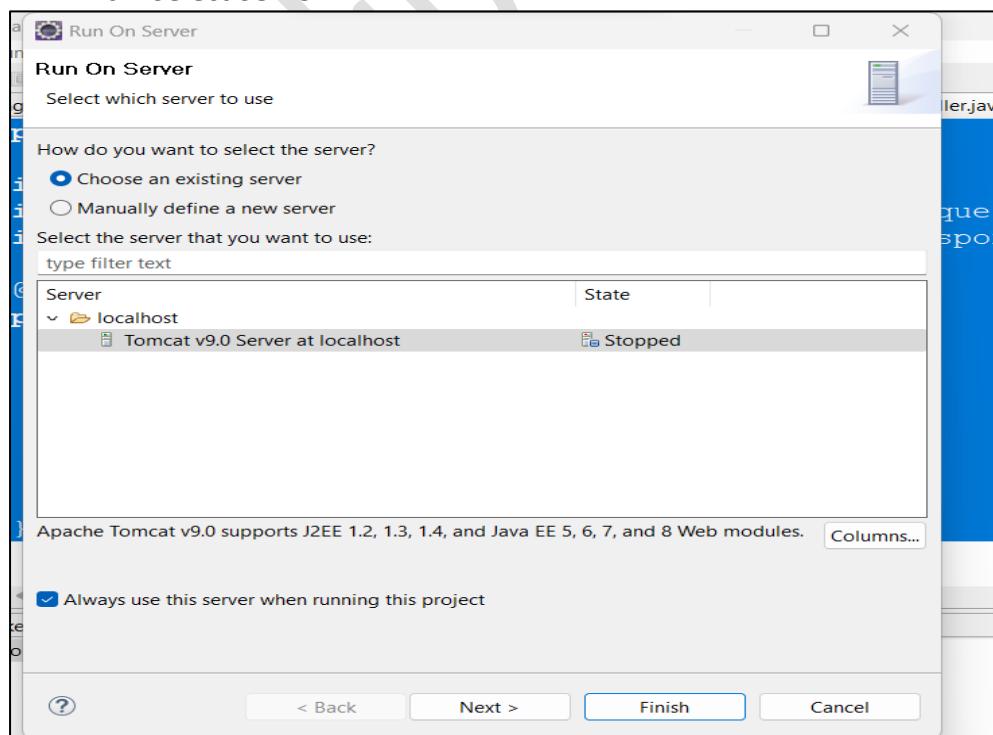
➤ **Select Tomcat folder Location**



➤ **Click on Finish, Server Added to eclipse.**

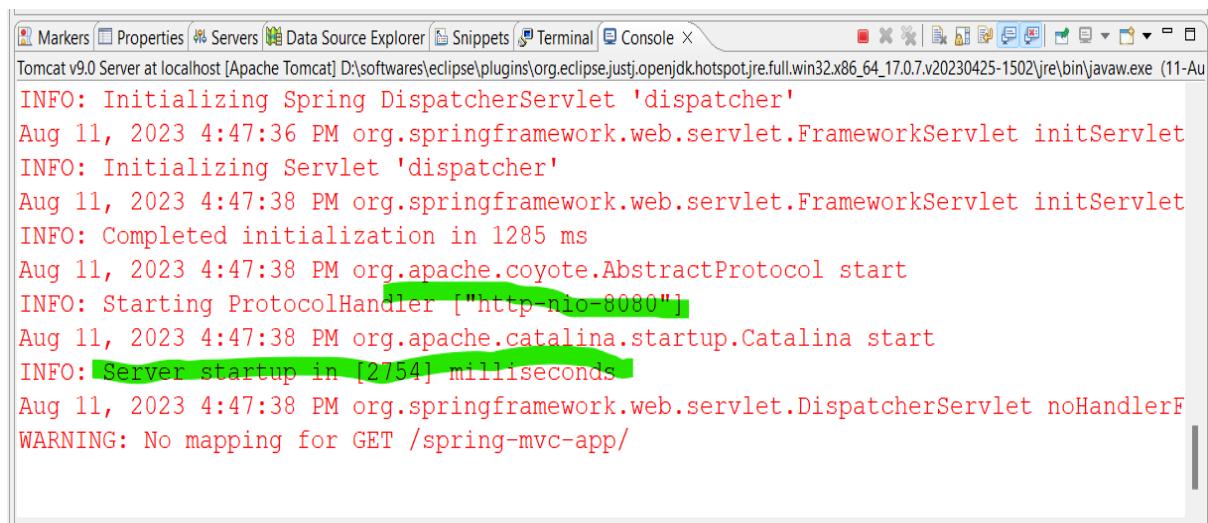
Project Deployment Process :

- Now Right Click On Project.
- Click on Run as -> Run On Server.
- Select Server



Click On Finish

Now Start Server, then Our Application will be Deployed and Shows message in console as Started.



The screenshot shows the Eclipse IDE's Console view with the following log output:

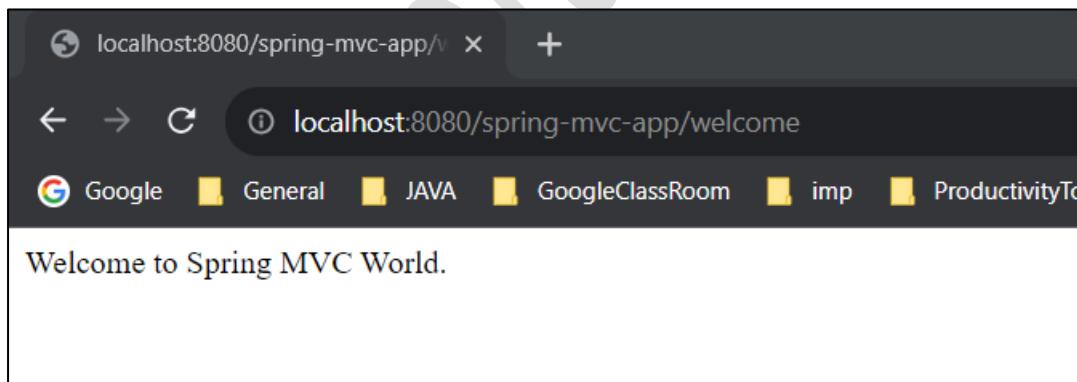
```
Tomcat v9.0 Server at localhost [Apache Tomcat] D:\softwares\apache-tomcat-9.0.54\bin\java.exe (11-Aug-2023) INFO: Initializing Spring DispatcherServlet 'dispatcher'  
Aug 11, 2023 4:47:36 PM org.springframework.web.servlet.FrameworkServlet initServlet  
INFO: Initializing Servlet 'dispatcher'  
Aug 11, 2023 4:47:38 PM org.springframework.web.servlet.FrameworkServlet initServlet  
INFO: Completed initialization in 1285 ms  
Aug 11, 2023 4:47:38 PM org.apache.coyote.AbstractProtocol start  
INFO: Starting ProtocolHandler ["http-nio-8080"]  
Aug 11, 2023 4:47:38 PM org.apache.catalina.startup.Catalina start  
INFO: Server startup in [2754] milliseconds  
Aug 11, 2023 4:47:38 PM org.springframework.web.servlet.DispatcherServlet noHandlerFound  
WARNING: No mapping for GET /spring-mvc-app/
```

We have Created One URI with **/welcome**. Now Test this URI by framing URL

Syntax : <http://localhost:8080/<project-name>/<URI>>

URL : <http://localhost:8080/spring-mvc-app/welcome>

Enter From Browser.

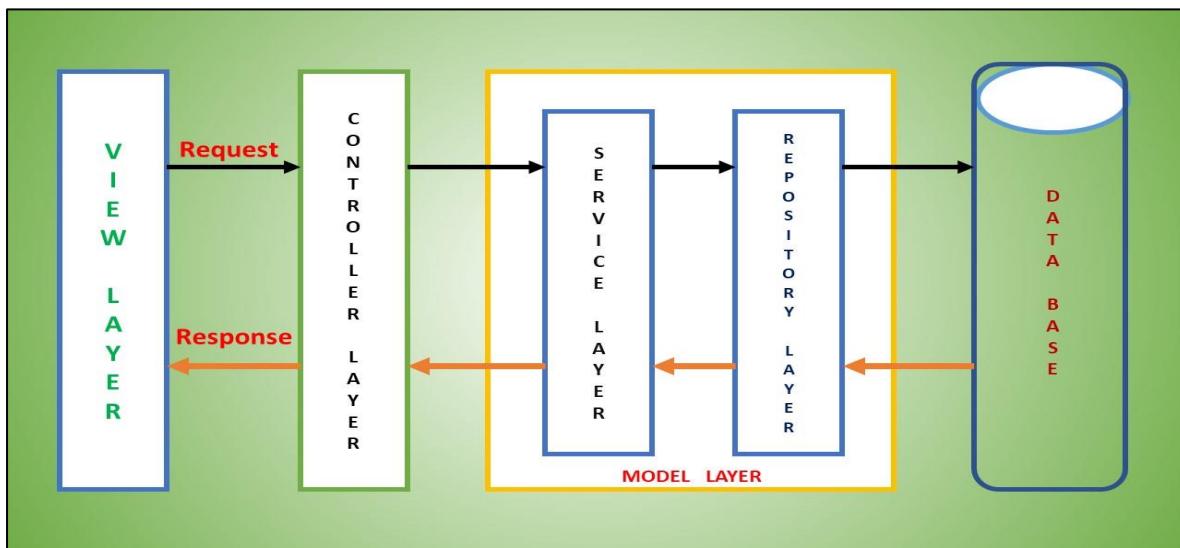


Now We got Response, So Project Setup is Completed.

Let's Focus on Internal Workflow of MVC Module.

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.

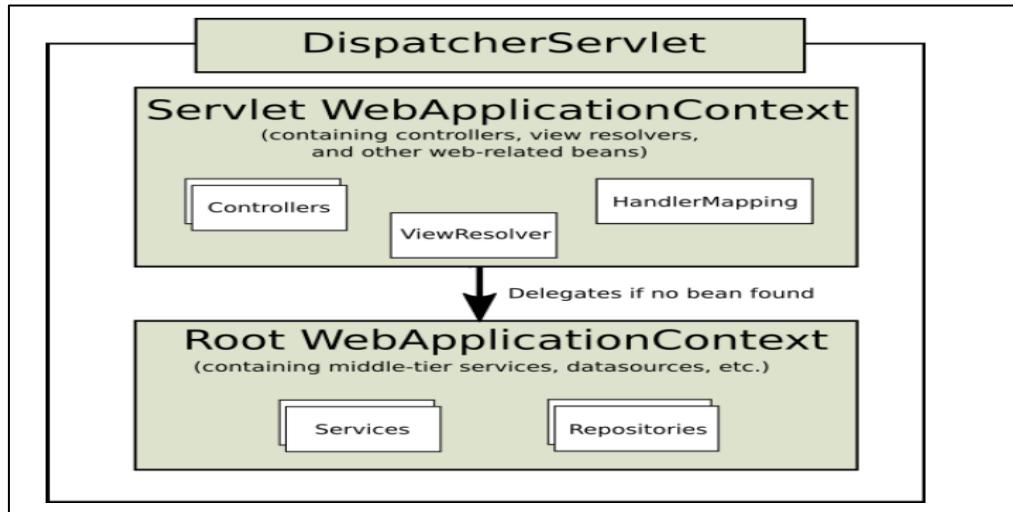
Front Controller:

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 MVC application:

- Create a controller class : IphoneController.java

```

package com.apple.iphone.controller;

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

```

@Controller
public class IphoneController {

    @GetMapping("/message")
    @ResponseBody
    public String printIphoneMessage() {
        //Logic of Method
        return " Welcome to Iphone World.";
    }

    @GetMapping("/cost")
    @ResponseBody
    public String printIphone14Cost() {
        return " Price is INR : 150000";
    }
}

```

➤ **Create another Controller class: IpadController.java**

```

package com.apple.iphone.controller;

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

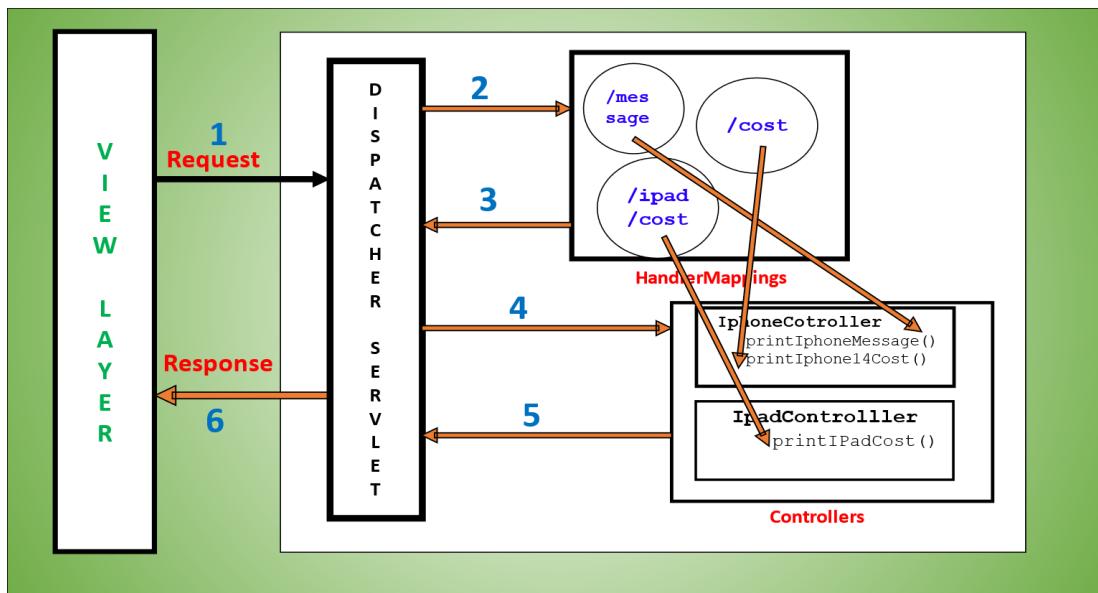
@Controller
public class IpadController {
    @GetMapping("/ipad/cost")
    @ResponseBody
    public String printIPadCost() {
        return " Ipad Price is INR : 200000";
    }
}

```

Now when we start our project as Spring Application, Internally Project deployed to tomcat server and below steps will be executed.

- When we are started/deployed out application, Spring MVC internally creates **WebApplicationContext** i.e. Spring Container to instantiate and manage all Spring Beans associated to our project.
- Spring instantiates Pre Defined Front Controller class called as **DispatcherServlet** as well as **WebApplicationContext** scans all our packages for **@Component**, **@Controller** etc.. and other Bean Configurations.
- Spring MVC **WebApplicationContext** will scan all our Controller classes which are marked with **@Controller** and starts creating Handler Mappings of all URL patterns defined in side controller classes with Controller and endpoint method names mappings.

In our App level, we created 2 controller classes with total 3 endpoints/URL-patterns.



After Starting/Deploying our Spring 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:8080/spring-mvc-app/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, 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	<code>String</code>	Assign a name to this mapping.
value	<code>String[]</code>	The primary mapping expressed by this annotation.
method	<code>RequestMethod[]</code>	The HTTP request methods to map to, narrowing the primary mapping: GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE.
headers	<code>String[]</code>	The headers of the mapped request, narrowing the primary mapping.
path	<code>String[]</code>	The path mapping URIs (e.g. "/profile").
consumes	<code>String[]</code>	media types that can be consumed by the mapped handler. Consists of one or more media types one of which must match to the request Content-Type header. <code>consumes = "text/plain"</code> <code>consumes = {"text/plain", "application/*"}</code> <code>consumes = MediaType.TEXT_PLAIN_VALUE</code>

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 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 at class level:

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 IpadController {

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

    @GetMapping("/model")
    @ResponseBody
```

```
public String printIPadModel() {
    return " Ipad Model is 2023 Mode";
}
```

From above example, class level Request mapping value ("/**ipad**") will be base URI for all handler method URI values. Means All URIs starts with **/ipad** of the controller.

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 value.

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)**

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

View Resolver:

In Spring MVC, a **ViewResolver** is a component responsible for resolving the logical view names returned by controller methods into actual view implementations that can render the response content. It acts as an intermediary between your controller methods and the actual view templates, helping to decouple the view resolution process from the controllers themselves.

When a controller method returns a view name, the **ViewResolver** is consulted to determine which view implementation should be used to render the response. The view resolver interprets the logical view name and maps it to the appropriate view template, which could be supported by the framework.

InternalResourceViewResolver:

This view resolver is commonly used with JSP views. It resolves view names to JSP files located within the web application's `WEB-INF` directory. You can configure the prefix and suffix to determine the exact location of your JSP files.

Here's a simple example of configuring an **InternalResourceViewResolver** in Spring configuration. In this example, the `InternalResourceViewResolver` is configured to resolve JSP views located in the `/WEB-INF/views/` directory with a `.jsp` extension.

Step 1: Add below Dependencies to **pom.xml** file, to support JSP and JSTL tags.

```

<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>spring-mvc-app</groupId>
  <artifactId>spring-mvc-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.3.29</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.0.1</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.1</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.2.3</version>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <release>17</release>
        </configuration>
      </plugin>
    </plugins>
  </build>

```

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

Step 2: We should Configure **InternalResourceViewResolver**. Here We are Configuring where our view files i.e. JSP files located in which path.

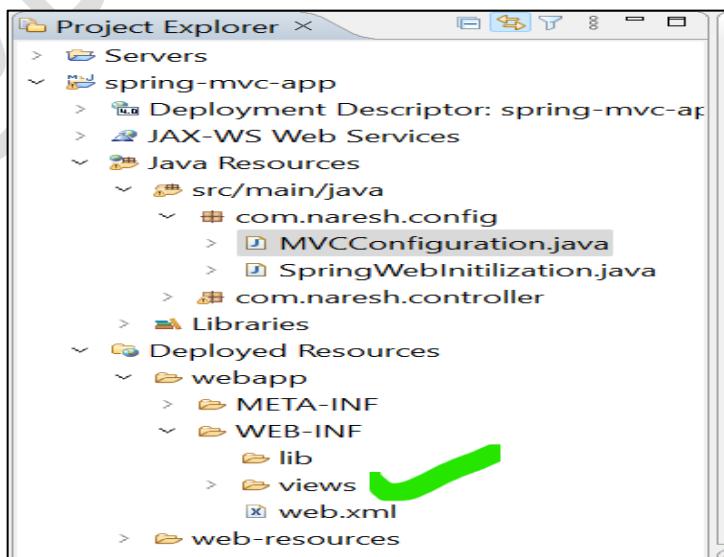
```
package com.naresh.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@ComponentScan("com.*")
@EnableWebMvc
public class MVCConfiguration extends WebMvcConfigurationSupport {

    @Bean
    public InternalResourceViewResolver internalResourceViewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

Step 2: Now create a folder **views inside WEB-INF, where we have to place all our JSP files.**



Step 3: Create JSP file inside views folder.

```
<%@ page contentType = "text/html; charset = UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<title>Spring MVC</title>
</head>
<body>
    <h1>${message}</h1>
</body>
</html>
```

Step 4: Now Inside Controller, Define a method with an URI and finally which should return view name.

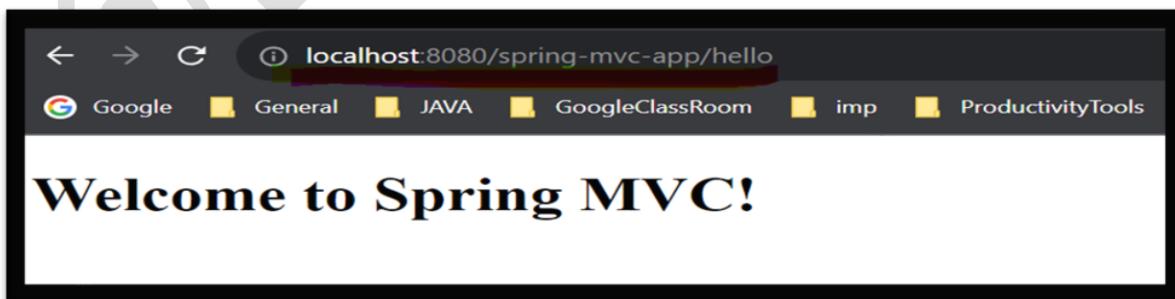
```
package com.naresh.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {

    @GetMapping("/hello")
    public String home(Model model) {
        model.addAttribute("message", "Welcome to Spring MVC!");
        return "home"; //This corresponds to a view named "home.jsp"
    }
}
```

Step 5: Now Test above URI from Browser.



In Above, JSP file rendered with an Output, what we set as part of Model Object in Controller.

What is Model?

In Spring MVC, the "**Model**" refers to a data structure that holds the data that needs to be displayed or processed by the **view**. It serves as a container for passing data between the controller and the view in a clean and organized manner. The Model allows you to separate concerns by keeping the data separate from the view rendering logic.

The Model interface itself is quite simple, and it's generally implemented using the Model class or a subclass of it. The main purpose of the Model is to store attributes (key-value pairs), where the keys are strings (attribute names) and the values are the actual data objects. These attributes can be accessed within the view to populate the content dynamically.

Rendering: When the controller method is executed and returns the logical view name, Spring MVC consults the configured **ViewResolver** to find the appropriate view template. The view template is then populated with the data from the Model, and the rendered output is sent as the response to the client.

The Model serves as an abstraction that helps you maintain a clear separation between your application's logic, data, and presentation. It promotes the MVC (Model-View-Controller) architectural pattern, allowing you to modify the data and presentation independently.

Remember that the Model is distinct from the request parameters and attributes. While request parameters are passed directly to the controller method as method parameters, the Model is used to provide data specifically for rendering views.

➤ **Similarly let's create other JSP file for login: login.jsp**

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>
<head><title>Login</title></head>
<body>

<form:form action="loginCheck" method="post">
    Enter User Name:<input type="text" name="name"> <br/>
    Enter Password :<input type="password" name="pwd"/><br/>
    <input type="submit">
</form:form>

</body>
</html>
```

➤ **Create another JSP which should display Validation of User : success.jsp**

```
<%@ page contentType = "text/html; charset = UTF-8" %>
<html>
<head><title>Login Success</title></head>
<body>
    Login Message : ${message}
```

```
</body>
</html>
```

- Define Controller Methods for login page and validation of User Credentials.

```
package com.naresh.controller;

import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HomeController {

    @GetMapping("/hello")
    public String home(Model model) {
        model.addAttribute("message", "Welcome to Spring MVC!");
        return "home"; // This corresponds to a view named "home.jsp"
    }

    @GetMapping("/")
    public String userLogin() {
        return "login"; // This corresponds to a view named "login.jsp"
    }

    @RequestMapping(value="/loginCheck", method = RequestMethod.POST)
    public String userCheck(ModelMap model, HttpServletRequest request) {

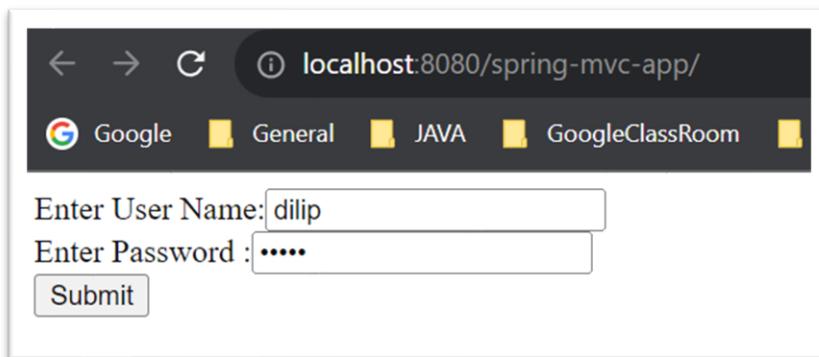
        String name=request.getParameter("name");
        String pwd=request.getParameter("pwd");

        if("dilip".equalsIgnoreCase(name)&&"dilip".equalsIgnoreCase(pwd)){
            model.addAttribute("message", "Successfully logged in.");
        }else{
            model.addAttribute("message", "Username or password is wrong.");
        }
        return "success"; // success.jsp
    }
}
```

Now Test our Login Page and Validation of Credentials.

Login Page URL : <http://localhost:8080/spring-mvc-app/>

- Enter User Name and password



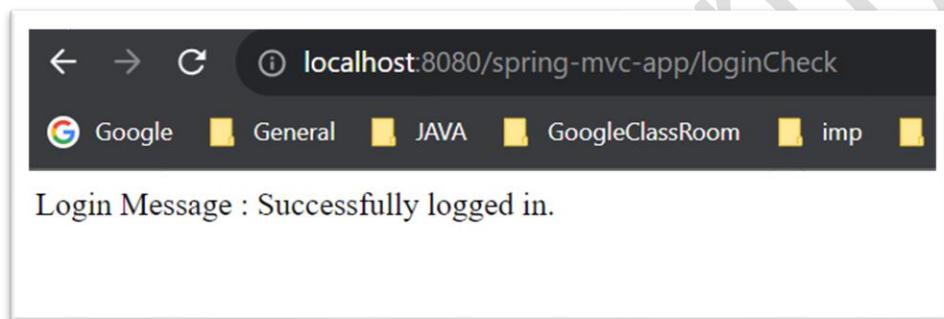
localhost:8080/spring-mvc-app/

Enter User Name: dilip

Enter Password : *****

Submit

After Submit,



localhost:8080/spring-mvc-app/loginCheck

Login Message : Successfully logged in.

Similarly We can add other View Layer **JSP** files and should integrate with Spring **MVC** Layer.

Assignment:

Create Spring MVC application with JSP files Integration.

1. Add User

User Name
Email
Mobile
Gender

2. Delete User

3. Update User Details

Webservices:

Web services are a standardized way for different software applications to communicate and exchange data over the internet. They enable interoperability between various systems, regardless of the programming languages or platforms they are built on. Web services use a set of protocols and technologies to enable communication and data exchange between different applications, making it possible for them to work together seamlessly.

Web services are used to integrate different applications and systems, regardless of their platform or programming language. They can be used to provide a variety of services, such as:

- Information retrieval
- Transaction processing
- Data exchange
- Business process automation

There are two main types of web services:

1. SOAP (Simple Object Access Protocol) Web Services:

SOAP is a protocol for exchanging structured information using XML. It provides a way for applications to communicate by sending messages in a predefined format. SOAP web services offer a well-defined contract for communication and are often used in enterprise-level applications due to their security features and support for more complex scenarios.

2. REST (Representational State Transfer) Web Services:

REST is an architectural style that uses HTTP methods (GET, POST, PUT, DELETE) to interact with resources in a stateless manner. RESTful services are simple, lightweight, and widely used due to their compatibility with the HTTP protocol. They are commonly used for building APIs that can be consumed by various clients, such as web and mobile applications.

The choice of web service type depends on factors such as the nature of the application, the level of security required, the complexity of communication, and the preferred data format.

REST Services Implementation in Spring MVC:

Spring MVC is a popular framework for creating web applications in Java. Implementing RESTful web services in Spring MVC involves using the Spring framework to create endpoints that follow the principles of the REST architectural style. It can be used to create RESTful web services, which are web services that use the REST architectural style.

RESTful services allow different software systems to communicate over the internet using standard HTTP methods, like GET, POST, PUT, and DELETE. These services are based on a set of principles that emphasize simplicity, scalability, and statelessness.

Here's how REST services work and the key principles they follow:

Resources: REST services revolve around resources, which can be any piece of information that can be named. Resources are identified by unique URLs, known as URIs (Uniform Resource Identifiers).

HTTP Methods: RESTful services use the standard HTTP methods to perform operations on resources:

- **GET:** Retrieve data from a resource.
- **POST:** Create a new resource.
- **PUT:** Update an existing resource or create if not exists.
- **DELETE:** Remove a resource.

Stateless Communication: One of the fundamental principles of REST is statelessness. Each request from a client to a server must contain all the information necessary for the server to understand and fulfil the request. This means that the server doesn't store any session information about the client between requests, making the system more scalable and easier to manage.

Representation: Resources are represented data using various formats such as **JSON**, **XML**, **HTML**, or **plain text**. The client and server communicate using these representations. Clients can request specific representations, and servers respond with the requested data format.

Cache ability: Responses from RESTful services can be cacheable. This reduces the need for repeated requests to the server and improves performance.

In REST Services implementation, Data will be represented as JSON/XML type most of the times. Now a days JSON is most popular data representational format to create and produce REST Services.

So, we should know about JSON.

JSON:

JSON stands for **JavaScript Object Notation**. JSON is a **text format** for storing and transporting data. JSON is "self-describing" and easy to understand.

This example is a JSON string: `{"name": "John", "age": 30, "car": null}`

JSON is a lightweight data-interchange format. JSON is plain text written in JavaScript object notation. JSON is used to exchange data between multiple applications/services. JSON is language independent.

JSON Syntax Rules

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

Example: "name": "John"

In JSON, values must be one of the following data types:

- a string
- a number
- an object
- an array
- a boolean
- null

JSON vs XML:

Both JSON and XML can be used to receive data from a web server. The following JSON and XML examples both define an employee's object, with an array of 3 employees:

JSON Example

```
{  
  "employees": [  
    {  
      "firstName": "John",  
      "lastName": "Doe"  
    },  
    {  
      "firstName": "Anna",  
      "lastName": "Smith"  
    },  
    {  
      "firstName": "Peter",  
      "lastName": "Jones"  
    }  
  ]  
}
```

XML Example:

```

<employees>
    <employee>
        <firstName>John</firstName>
        <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName>
        <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName>
        <lastName>Jones</lastName>
    </employee>
</employees>

```

JSON is Like XML Because

- Both JSON and XML are "self-describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages

When A Request Body Contains JSON/XML data Format, then how Spring MVC/JAVA language handling Request data?

Here, We should **Convert JSON/XML data to JAVA Object** while Request landing on Controller method, after that we are using JAVA Objects in further process. Similarly, Sometimes we have to send Response back as either JSON or XML format i.e. JAVA Objects to JSON/XML Format.

For these conversions, we have few pre-defined solutions in Market like Jackson API, GSON API, JAXB etc..

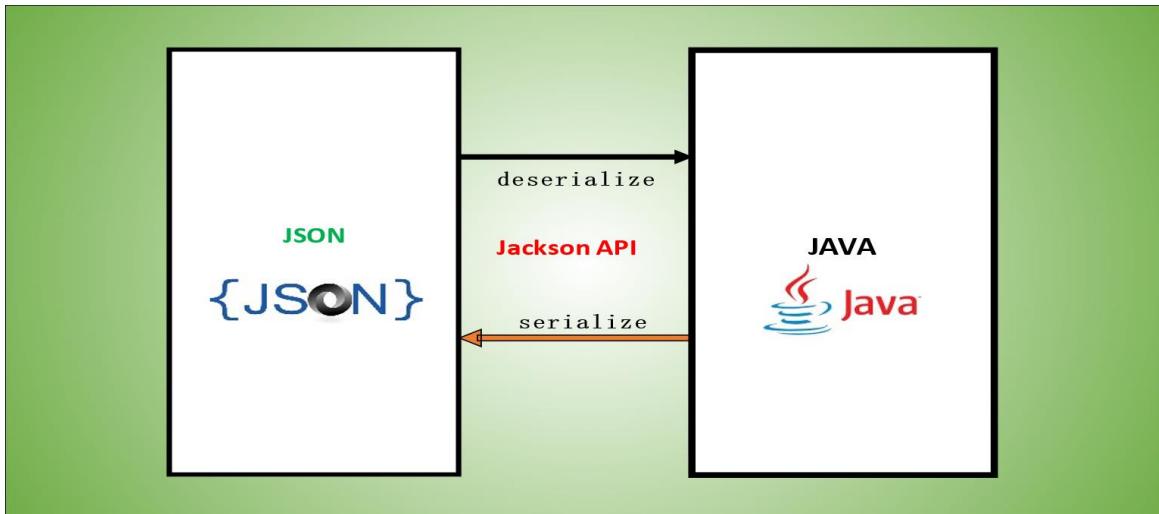
JSON to JAVA Conversion:

Spring MNC supports Jackson API which will take care of un-marshalling JSON request body to Java objects. We should add Jackson API dependencies explicitly. We can use **@RequestBody** Spring MVC annotation to deserialize/un-marshall JSON string to Java object. Similarly, java method return data will be converted to JSON format i.e. Response of Endpoint by using an annotation **@ResponseBody**.

And as you have annotated with **@ResponseBody** of endpoint method, we no need to do explicitly JAVA to JSON conversion. Just return a POJO and Jackson serializer will take care of converting to Json format. It is equivalent to using **@ResponseBody** when used with **@Controller**. Rather than placing **@ResponseBody** on every controller method we place

`@RestController` instead of `@Controller` and `@ResponseBody` by default is applied on all resources in that controller.

Note: we should create 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 taken as `List<String>` with JSON key name.

```

import java.util.List;

public class StudentDetails {

    private List<String> student;

    public List<String> getStudent() {
        return student;
    }

    public void setStudent(List<String> student) {
        this.student = student;
    }
}
  
```

```

    }
    @Override
    public String toString() {
        return "StudentDetails [student=" + student + "]";
    }
}

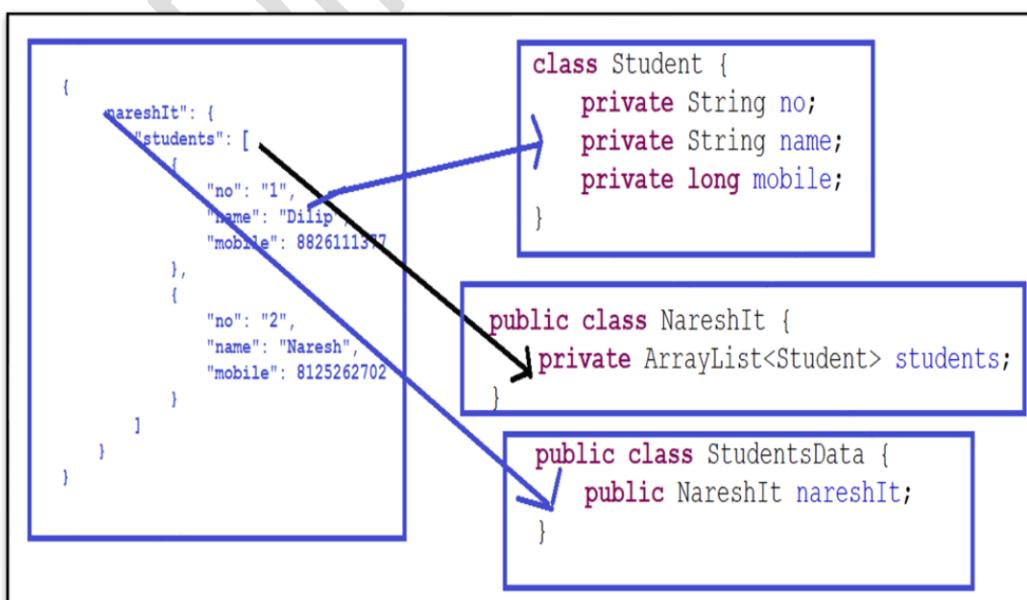
```

JSON payload with Array of Student Object Values:

Below JSON payload contains array of Student values.

```
{
  "nareshIt": {
    "students": [
      {
        "no": "1",
        "name": "Dilip",
        "mobile": 8826111377
      },
      {
        "no": "2",
        "name": "Naresh",
        "mobile": 8125262702
      }
    ]
  }
}
```

Below picture showing how are creating JAVA classes from above payload.



Created Three java files to wrap above JSON payload structure:

Student.java

```
public class Student {
    private String no;
    private String name;
    private long mobile;

    //Setters and Getters
}
```

Add Student as List in class NareshIt.java

```
import java.util.ArrayList;
public class NareshIt {
    private ArrayList<Student> students;

    //Setters and Getters

}
```

StudentsData.java

```
public class StudentsData {
    public NareshIt nareshIt;

    //Setters and Getters
}
```

Now we will use **StudentsData** class to bind our JSON Payload.

- [Let's Take another Example of JSON to JAVA POJO class:](#)

JSON PAYLOAD: Json with Array of Student Objects

```
{
    "student": [
        {
            "firstName": "Dilip",
            "lastName": "Singh",
            "mobile": 88888,
            "pwd": "Dilip",
            "emailID": "Dilip@Gmail.com"
        },
    ]}
```

```
{
    "firstName": "Naresh",
    "lastName": "It",
    "mobile": 232323,
    "pwd": "Naresh",
    "emailID": "Naresh@Gmail.com"
}
]
```

For the above Payload, JAVA POJO'S are:

```
import com.fasterxml.jackson.annotation.JsonProperty;

public class StudentInfo {

    private String firstName;
    private String lastName;
    private long mobile;
    private String pwd;
    @JsonProperty("emailID")
    private String email;

    //Setters and Getters
}
```

Another class To Wrap above class Object as List with property name student as per JSON.

```
import java.util.List;

public class Students {

    List<StudentInfo> student;

    public List<StudentInfo> getStudent() {
        return student;
    }
    public void setStudent(List<StudentInfo> student) {
        this.student = student;
    }
}
```

From the above JSON payload and JAVA POJO class, we can see a difference for one JSON property called as **emailID** i.e. in JAVA POJO class property name we taken as **email** instead of emailID. In Such case to map JSON to JAVA properties with different names, we use an annotation called as `@JsonProperty("jsonPropertyName")`.

@JsonProperty:

The **@JsonProperty** annotation is used to specify the property name in a JSON object when serializing or deserializing a Java object using the Jackson API library. It is often used when the JSON property name is different from the field name in the Java object, or when the JSON property name is not in camelCase.

If you want to serialize this object to JSON and specify that the JSON property names should be "first_name", "last_name", and "age", you can use the **@JsonProperty** annotation like this:

```
public class Person {
    @JsonProperty("first_name")
    private String firstName;
    @JsonProperty("last_name")
    private String lastName;
    @JsonProperty
    private int age;

    // getters and setters go here
}
```

As a developer, we should always create POJO classes aligned to JSON payload to bind JSON data to Java Object with **@RequestBody** annotation.

To implement REST services in Spring MVC, you can use the **@RestController** annotation. This annotation marks a class as a controller that returns data to the client in a RESTful way.

@RestController:

Spring introduced the **@RestController** annotation in order to simplify the creation of RESTful web services. **@RestController** is a specialized version of the controller. It's a convenient annotation that combines **@Controller** and **@ResponseBody**, which eliminates the need to annotate every request handling method of the controller class with the **@ResponseBody** annotation.

Package: org.springframework.web.bind.annotation.RestController;

For example, When we mark class with **@Controller** and we will use **@ResponseBody** at request mapping method level.

```
@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";
    }
}
```

Project Setup:

1. Create Spring MVC Project by following earlier Steps.
2. Add Jackson API Dependencies in pom.xml file, to Support JSON Data Representation i.e. JSON to JAVA and vice versa conversion.

```
<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>swiggy</groupId>
    <artifactId>swiggy</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.3.29</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
            <scope>provided</scope>
        </dependency>
    <!--JSON : Jackson API jars :-->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.15.2</version>
    </dependency>

```

```
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.2</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.15.2</version>
</dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.2.3</version>
        </plugin>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <release>17</release>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

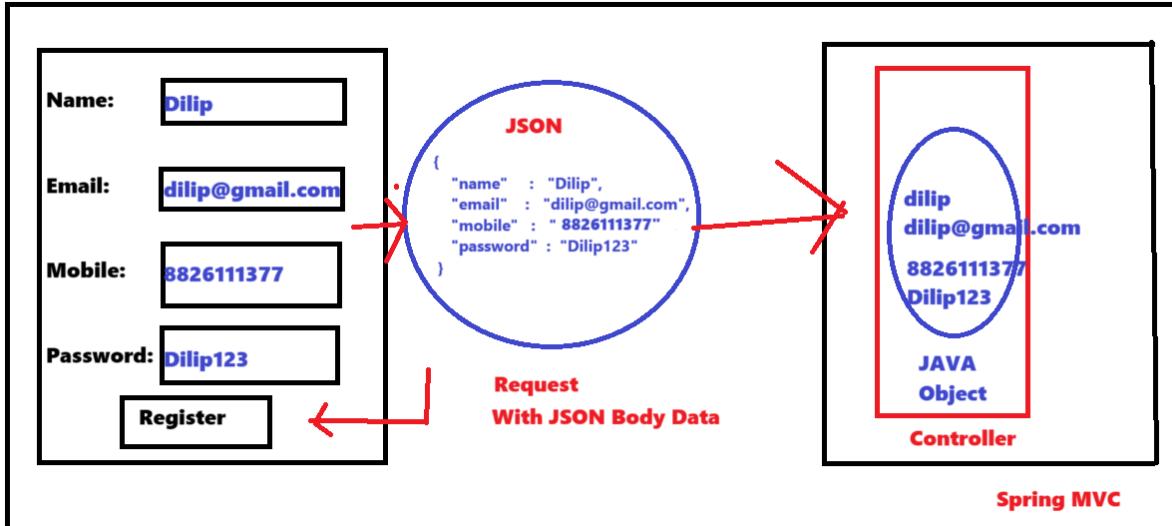
With These steps, Project Setup completed in support of JSON Data Format.

3. Now Create an endpoint or REST Service for below requirement.

Requirement: Write a Rest Service for User Registration.

User Details Should Be :

- User Name
- Email Id
- Mobile
- Password



4. Create a JSON Mapping for above Requirement with dummy values.

```
{
  "name" : "Dilip",
  "email" : "dilip@gmail.com",
  "mobile" : "+91 73777373",
  "password" : "Dilip123"
}
```

5. Before Creating Controller class, we should Create JAVA POJO class which is compatible with JSON Request Data. So create a JAVA class, as discussed previously. Which is Responsible for holding Request Data of JSON.

```
package com.swiggy.user.request;

public class UserRegisterRequest {

    private String name;
    private String email;
    private String mobile;
    private String password;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
```

```

        this.email = email;
    }
    public String getMobile() {
        return mobile;
    }
    public void setMobile(String mobile) {
        this.mobile = mobile;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}

```

6. Now Create A controller and inside an endpoint for User Register Request Handling.

```

package com.swiggy.user.controller;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.swiggy.user.request.UserRegisterRequest;

@RestController
@RequestMapping("/user")
public class UserController {

    @PostMapping("/register")
    public String getUserDetails(@RequestBody UserRegisterRequest request){

        System.out.println(request.getEmail());
        System.out.println(request.getName());
        System.out.println(request.getPassword());

        return "User Created Sucessfully";
    }
}

```

In Above, We are used **@RequestBody** for binding/mapping incoming JSON request to JAVA Object at method parameter layer level. Means, Spring MVC internally maps JSON to JAVA with help of Jackson API jar files.

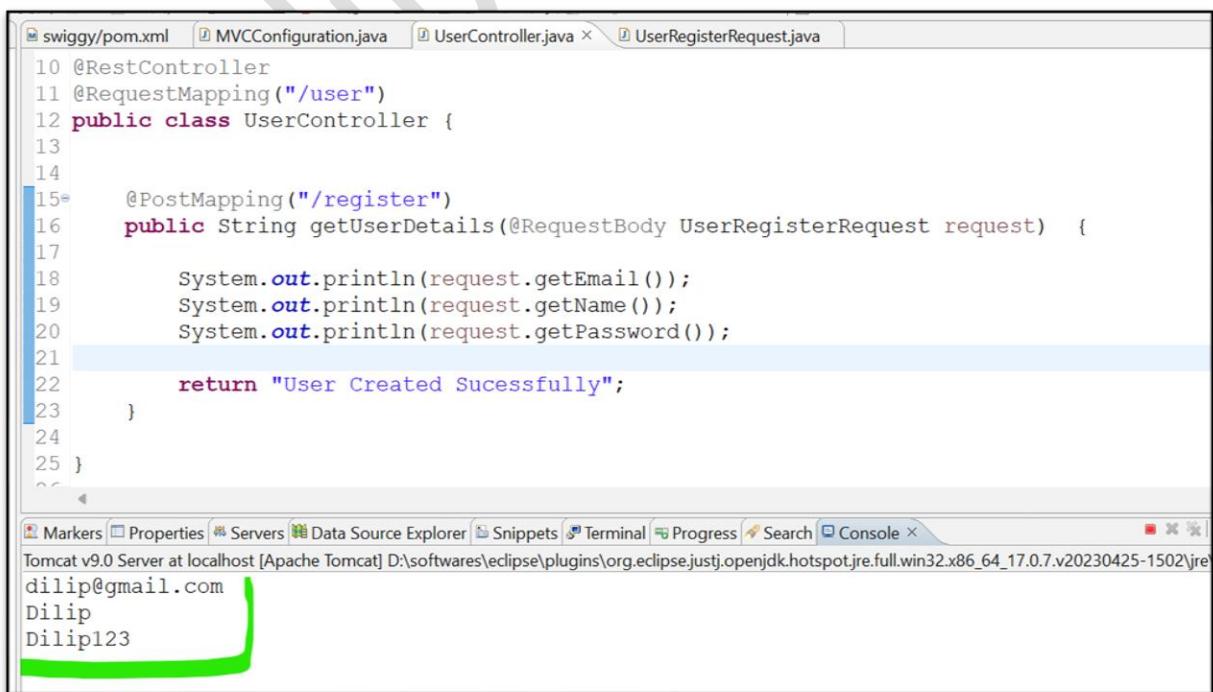
7. Deploy Your Application in Server Now. After Deployment we have to Test now weather it Service is working or not.

8. Now We are Taking Help of **Postman** to do API/Services Testing.

- Open Postman
- Now Click on Add request
- Select Your Service HTTP method
- And Enter URL of Service
- Select Body
- Select raw
- Select JSON
- Enter JSON Body as shown in Below.



After Clicking on **Send** Button, Summited Request to Spring MVC REST Service Endpoint method and we got Response back with **200 Ok** status Code. We can See in Server Console, Request Data printed what we Received from Client level as JSON data.



```

10 @RestController
11 @RequestMapping("/user")
12 public class UserController {
13
14
15@PostMapping("/register")
16 public String getUserDetails(@RequestBody UserRegisterRequest request) {
17
18     System.out.println(request.getEmail());
19     System.out.println(request.getName());
20     System.out.println(request.getPassword());
21
22     return "User Created Sucessfully";
23 }
24
25 }
  
```

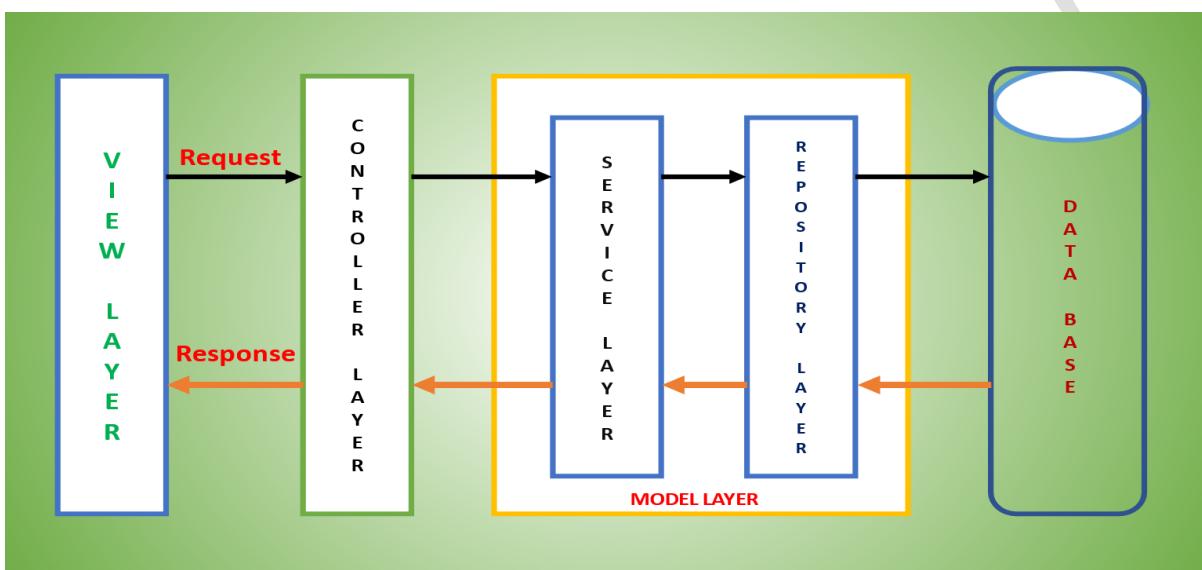
The Tomcat server console output shows the received JSON data:

```

dilip@gmail.com
Dilip
Dilip123
  
```

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, this is my Service Layer.

Create An Example with Service Layer:

- Create Controller and Service classes in MVC Application

Example:

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 Integrate Service Layer with Controller Layer in our previous application.

Means, forward request data to Service layer from controller.

➤ Create User Service : UserService.java

```

package com.swiggy.user.service;

import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.swiggy.user.entity.SwiggyUsers;
import com.swiggy.user.request.UserRegisterRequest;

@Service
public class UserService {
    public String registerUserDetails(UserRegisterRequest request) {
        System.out.println(request.getEmail());
        System.out.println(request.getName());
        System.out.println(request.getPassword());
        return "User Registered Successfully";
    }
}

```

➤ Inside Controller Class, Add Service Layer: UserController.java

```

package com.swiggy.user.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;

```

```

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;
import com.swiggy.user.entity.SwiggyUsers;
import com.swiggy.user.request.UserRegisterRequest;
import com.swiggy.user.response.UserRegisterResponse;
import com.swiggy.user.service.UserService;

@RestController
@RequestMapping("/user")
public class UserController {

    @Autowired
    UserService userService;

    @PostMapping("/register")
    public String registerUserDetails(@RequestBody UserRegisterRequest request) {

        // Controller -> Service
        // From Service receiving Result
        String response = userService.registerUserDetails(request);

        return response;
    }
}

```

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

Now return 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 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.

As we discussed earlier in JPA functionalities, Now we have to integrate JPA Module to our existing application.

JPA Module Integration:

Add Below Jar Dependencies to existing **pom.xml** file:

```
<!--Spring ORM -->
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc8</artifactId>
    <version>21.9.0.0</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.9.Final</version>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>2.1.0.RELEASE</version>
</dependency>
```

Spring JPA Config Class: SpringJpaConfiguration.java

```
package com.swiggy;

import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories("com.*")
public class SpringJpaConfiguration {

    // DB Details
    @Bean
    public DataSource getDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
```

```

        dataSource.setUsername("c##dilip");
        dataSource.setPassword("dilip");
        dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        return dataSource;
    }

    @Bean("entityManagerFactory")
    LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {
        LocalContainerEntityManagerFactoryBean factory =
            new LocalContainerEntityManagerFactoryBean();

        // 1. Setting Datasource Object // DB details
        factory.setDataSource(getDataSource());

        // 2. Provide package information of entity classes
        factory.setPackagesToScan("com.*");

        // 3. Providing Hibernate Properties to EM
        factory.setJpaProperties(hibernateProperties());

        // 4. Passing Predefined Hiberante Adaptor Object EM
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        factory.setJpaVendorAdapter(adapter);

        return factory;
    }

    // Spring JPA: configuring data based on your project req.
    @Bean("transactionManager")
    public PlatformTransactionManager createTransactionManager() {
        JpaTransactionManager transactionManager =
            new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(createEntityManagerFactory().getObject());
        return transactionManager;
    }

    // these are all from hibernate FW , Predefined properties : Keys
    Properties hibernateProperties() {
        Properties hibernateProperties = new Properties();
        hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");
        hibernateProperties.setProperty("hibernate.dialect",
            "org.hibernate.dialect.Oracle10gDialect");
        hibernateProperties.setProperty("hibernate.show_sql", "true");
        return hibernateProperties;
    }
}

```

Now Our Spring MVC Application is Ready to support JPA functionalities. We should Follow JPA logic like entity classes and Repositories.

➤ **Create Entity class for storing User Registration. Table Name is : swiggy_users**

```
package com.swiggy.user.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="swiggy_users")
public class SwiggyUsers {

    @Id
    @Column
    private String email;
    @Column
    private String name;
    @Column
    private String mobile;
    @Column
    private String password;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }

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

```

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

```

➤ Now Create Spring JPA Repository.

```

package com.swiggy.user.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.swiggy.user.entity.SwiggyUsers;

@Repository
public interface UserRepository extends JpaRepository<SwiggyUsers, String>{
}

```

@Repository:

The **@Repository** annotation is part of the Spring Data Access Layer, which provides support for data access operations and database interactions. When you apply the **@Repository** annotation to a class, you're indicating to Spring that the class is a repository or a data access object (DAO). Repositories are responsible for managing data storage and retrieval, often interacting with databases using technologies like Java Persistence API (JPA) or Hibernate.

By using **@Repository**, you allow Spring to manage the instantiation, configuration, and handling of the repository beans, and you can benefit from features such as exception translation (converting database-specific exceptions into Spring's `DataAccessException` hierarchy) and declarative transaction management.

Remember that **@Repository** is just one of the many annotations provided by Spring to help manage different aspects of your application. Other annotations, like **@Service** and **@Controller**, have specific roles in the Spring framework and are used to mark classes for different responsibilities, like service layer logic and web request handling, respectively.

Now We should Integrate Repository Layer with Service Layer. So Inside Service class, we have to autowire Repository Interface and then We will call Spring JPA Repositories methods.

```

package com.swiggy.user.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.swiggy.user.entity.SwiggyUsers;

```

```

import com.swiggy.user.repository.UserRepository;
import com.swiggy.user.request.UserRegisterRequest;
import com.swiggy.user.response.UserRegisterResponse;

@Service
public class UserService {

    @Autowired
    UserRepository repository;

    public String registerUserDetails(UserRegisterRequest registerRequest) {
        // Mapping Request data to Entity Object because Repository will use Entity
        Object
        SwiggyUsers user = new SwiggyUsers();
        user.setEmail(registerRequest.getEmail());
        user.setMobile(registerRequest.getMobile());
        user.setName(registerRequest.getName());
        user.setPassword(registerRequest.getPassword());
        repository.save(user);
        return "User Registered Successfully";
    }
}

```

With This Step, REST service for User Register is Completed. Let's Verify Data is getting inserted or not. **Deploy Our Application in Tomcat Server.**

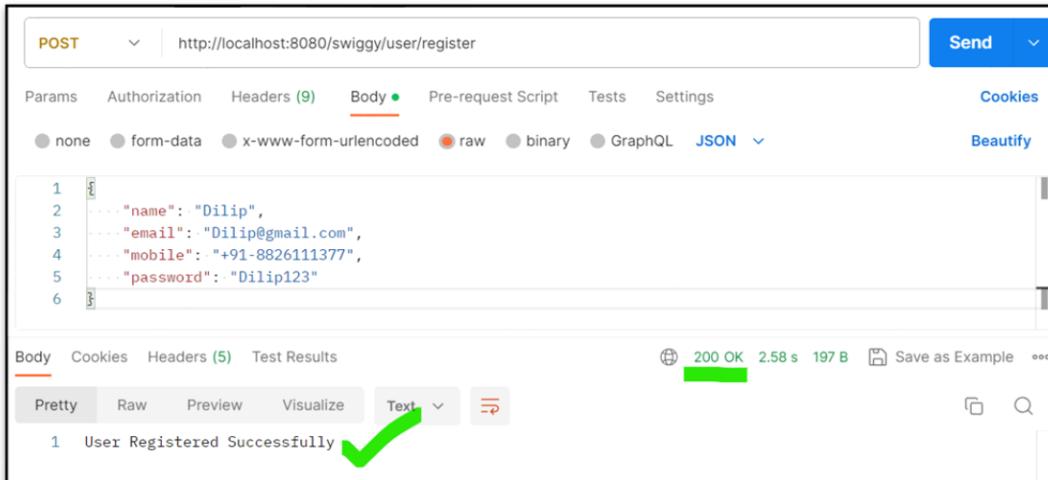
NOTE: we have used "hibernate.hbm2ddl.auto" value as "update". So, If Table Not available in DB, Table will be created while Deployment, or uses existing table for DB operations.

```

Hibernate: create table swiggy_users (email varchar2(255 char) not null, mobile varchar2(255 char), name varchar2(255 ch
Aug 23, 2023 2:45:30 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform
Aug 23, 2023 2:45:32 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-nio-8080"]
Aug 23, 2023 2:45:32 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in [74110] milliseconds

```

Testing: From Postman, Trigger REST Service.



POST http://localhost:8080/swiggy/user/register

Body (JSON)

```

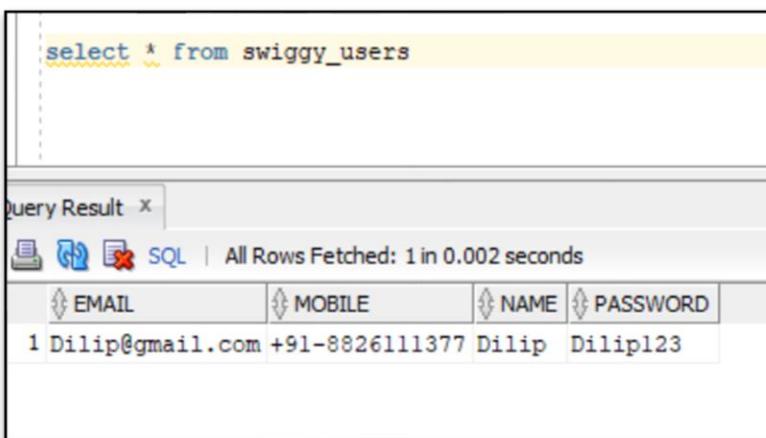
1
2 ... "name": "Dilip",
3 ... "email": "Dilip@gmail.com",
4 ... "mobile": "+91-8826111377",
5 ... "password": "Dilip123"
6

```

200 OK 2.58 s 197 B Save as Example

User Registered Successfully

- Verify In Database. Data Inserted.



```
select * from swiggy_users
```

EMAIL	MOBILE	NAME	PASSWORD
Dilip@gmail.com	+91-8826111377	Dilip	Dilip123

Similarly, we can create other REST services now as per our requirement with Database Operations.

- Create REST Service for Fetching USER Details based on email ID.
- Create REST API call for fetching all Users information.

User Details :

- User Name
- Mobile Number
- Email Id

- User Details Response DTO class for returning Response. **UserRegisterResponse.java**

```
package com.swiggy.user.response;

public class UserRegisterResponse {

    private String name;
    private String email;
    private String mobile;
```

```

public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public String getMobile() {
    return mobile;
}
public void setMobile(String mobile) {
    this.mobile = mobile;
}
}

```

- Now Create Endpoint methods/Service Methods in Controller, Service and Repository Layers respectively.

UserController.java

```

package com.swiggy.user.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.swiggy.user.entity.SwiggyUsers;
import com.swiggy.user.request.UserRegisterRequest;
import com.swiggy.user.response.UserRegisterResponse;
import com.swiggy.user.service.UserService;

@RestController
@RequestMapping("/user")
public class UserController {

    @Autowired
    UserService userService;
}

```

```

    @PostMapping("/register")
    public String registerUserDetails(@RequestBody UserRegisterRequest request) {
        // Controller -> Service and From Service receiving Result
        String response = userService.registerUserDetails(request);
        return response;
    }

    // Fetch user Details of one Person by email ID
    @GetMapping("/fetch")
    public UserRegisterResponse getUserDetails() {
        String email = "aaa@gmail.com";
        UserRegisterResponse response = userService.getUserDetails(email);
        return response;
    }

    // Loading all User Details
    @GetMapping("/fetch/all")
    public List<SwiggyUsers> getAllUserDetails() {
        return userService.getAllUserDetails();
    }
}

```

UserService.java

```

package com.swiggy.user.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.swiggy.user.entity.SwiggyUsers;
import com.swiggy.user.repository.UserRepository;
import com.swiggy.user.request.UserRegisterRequest;
import com.swiggy.user.response.UserRegisterResponse;

@Service
public class UserService {

    @Autowired
    UserRepository repository;

    public String registerUserDetails(UserRegisterRequest registerRequest) {
        // Mapping data to Entity Object
        SwiggyUsers user = new SwiggyUsers();
        user.setEmail(registerRequest.getEmail());
        user.setMobile(registerRequest.getMobile());
        user.setName(registerRequest.getName());
    }
}

```

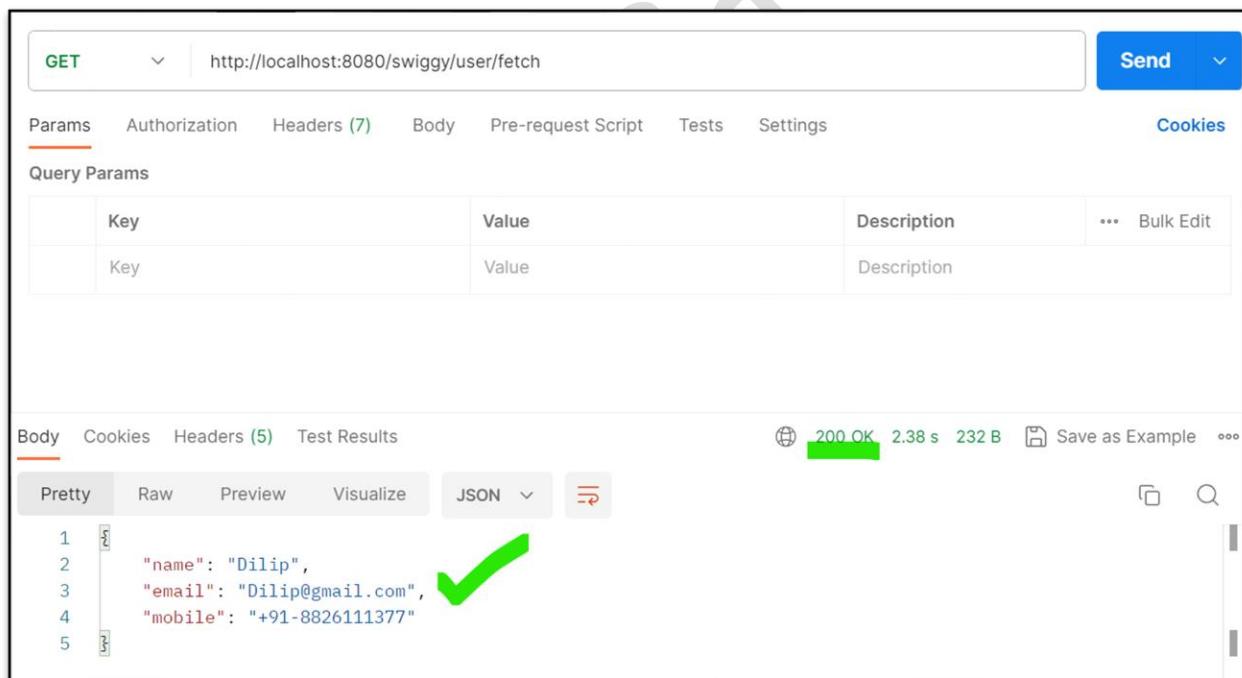
```

        user.setPassword(registerRequest.getPassword());
        repository.save(user);
        return "User Registered Successfully";
    }

    public UserRegisterResponse getUserDetails(String email) {
        SwiggyUsers user = repository.findById(email).get();
        UserRegisterResponse response = new UserRegisterResponse();
        response.setEmail(user.getEmail());
        response.setMobile(user.getMobile());
        response.setName(user.getName());
        return response;
    }
    public List<SwiggyUsers> getAllUserDetails() {
        return repository.findAll();
    }
}

```

Testing: Calling API services.

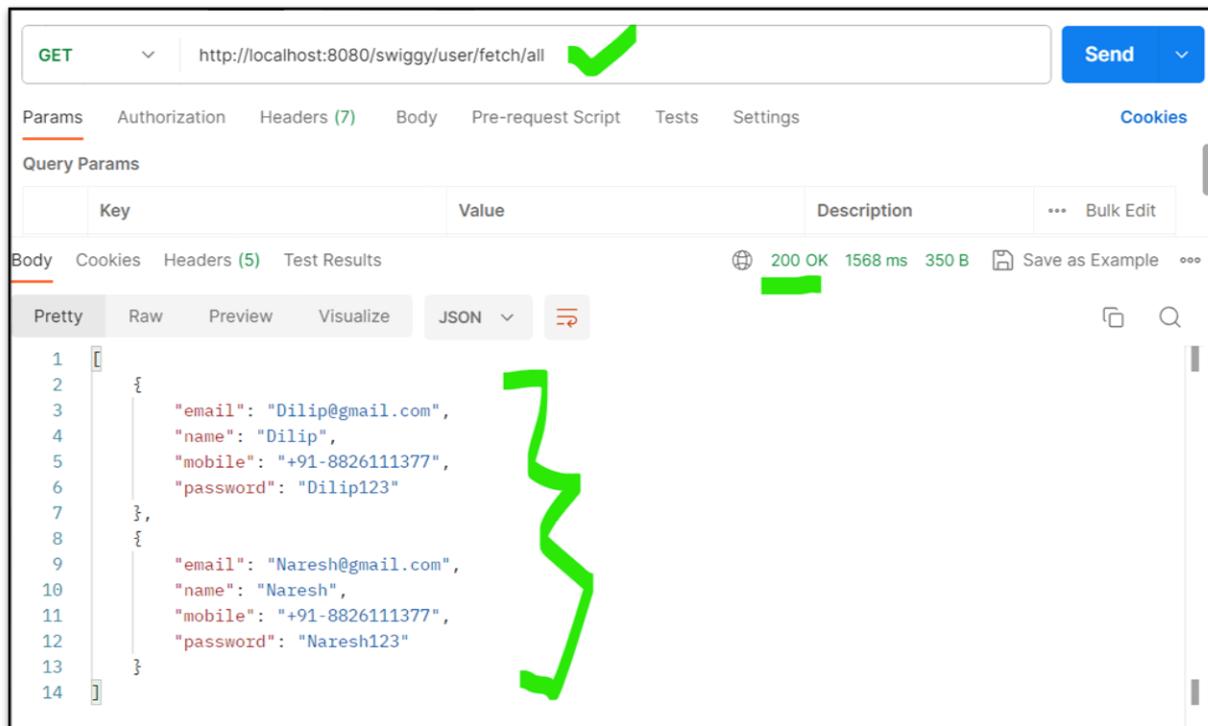


The screenshot shows a Postman request configuration for a GET request to `http://localhost:8080/swiggy/user/fetch`. The 'Params' tab is selected. In the 'Body' section, the 'Pretty' option is chosen, and the response is displayed as:

```

1  {
2   "name": "Dilip",
3   "email": "Dilip@gmail.com",
4   "mobile": "+91-8826111377"
5 }
```

The status bar at the bottom indicates a successful response: `200 OK`, `2.38 s`, `232 B`.



GET http://localhost:8080/swiggy/user/fetch/all

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description	... Bulk Edit

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

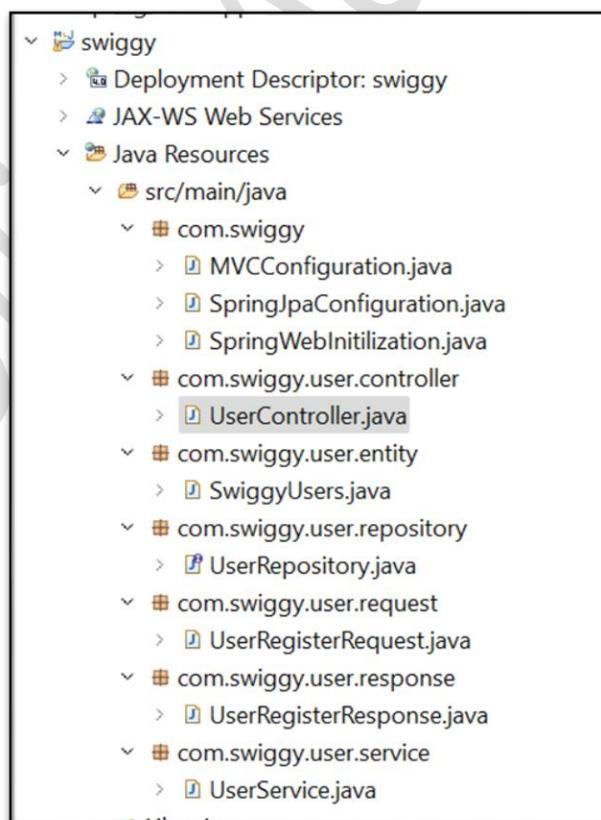
```

1 [
2   {
3     "email": "Dilip@gmail.com",
4     "name": "Dilip",
5     "mobile": "+91-8826111377",
6     "password": "Dilip123"
7   },
8   {
9     "email": "Naresh@gmail.com",
10    "name": "Naresh",
11    "mobile": "+91-8826111377",
12    "password": "Naresh123"
13 }
14 ]

```

200 OK 1568 ms 350 B Save as Example ...

Project Structure:



Project on GitHub: https://github.com/dilipsingh1306/spring_mvc_jpa_oracle_example

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;

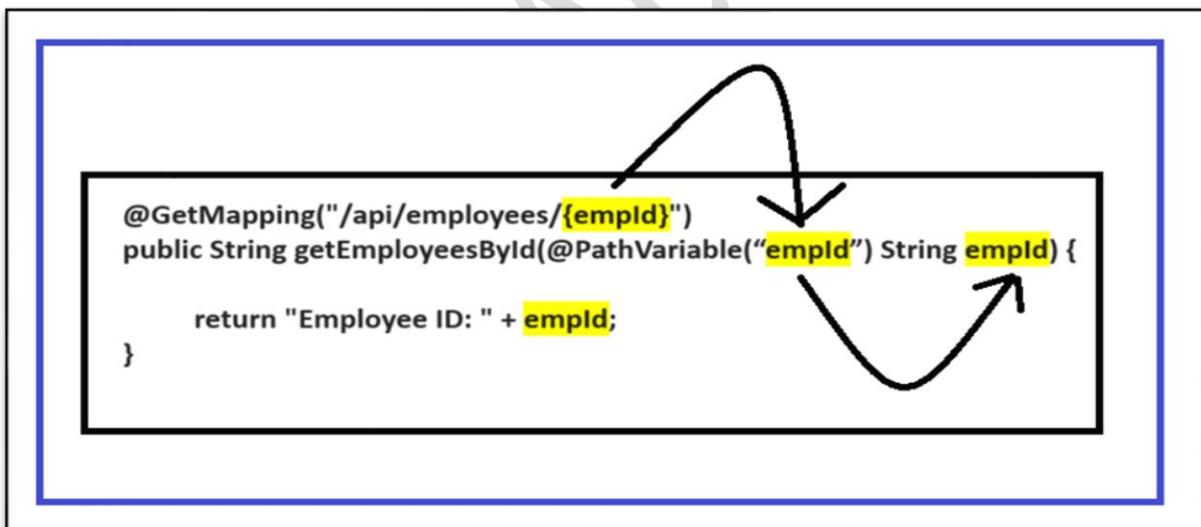
Examples:

URI with Template Path variables : /location/{locationName}/pincode/{pincode}

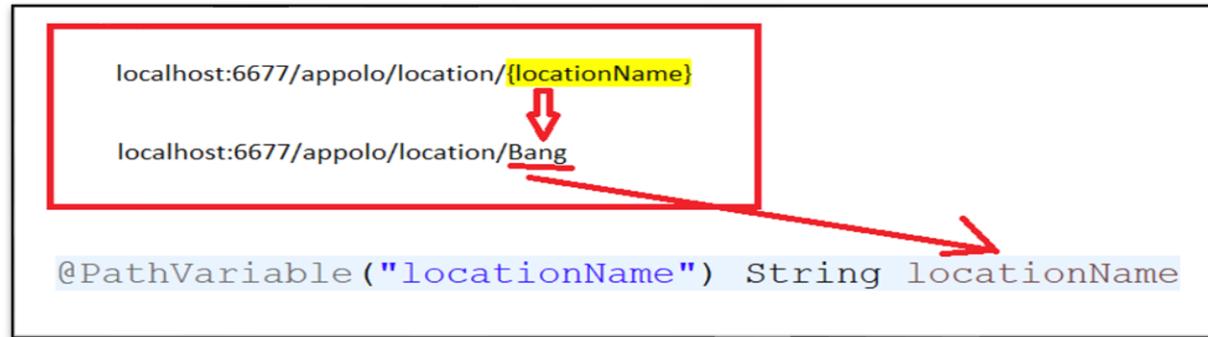
URI with Data replaced : /location/**Hyderabad**/pincode/**500072**

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;
}
```



Example 2:



Requirement : Get User Details with User Email Id.

In these kind of requirements, like getting Data with Resource ID's. We can Use Path Variable as part of URI instead of JSON mapping and equivalent Request Body.

So Create a REST endpoint with Path Variable of Email ID.

UserController.java : Add Below Logic In existing User Controller.

```
@RequestMapping(value = "/get/{emailId}", method = RequestMethod.GET)
public UserRegisterResponse getUserByEmailId(@PathVariable("emailId") String email) {
    return userService.getUserDetails(email);
}
```

Now Add Method in Service Class for interacting with Repository Layer.

Method inside Service Class : UserService.java

```
public UserRegisterResponse getUserDetails(String email) {
    SwiggyUsers user = repository.findById(email).get();
    UserRegisterResponse response = new UserRegisterResponse();
    response.setEmail(user.getEmail());
    response.setMobile(user.getMobile());
    response.setName(user.getName());
    return response;
}
```

Testing: Pass Email Value in place of PATH variable

GET http://localhost:8080/swiggy/user/get/Dilip@gmail.com

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (6) Test Results

200 OK 1645 ms 276 B Save as Example ...

Pretty Raw Preview Visualize JSON

```

1  {
2   "name": "Dilip",
3   "email": "Dilip@gmail.com",
4   "mobile": "+91-8826111377"
5 }
```

Multiple Path Variable as part of URI:

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

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

Example: /pharmacy/{location}/pincode/{pincode}

Requirement : Add Order Details as shown in below.

- Order ID
- Order status
- Amount
- Email Id
- City

After adding Orders, Please Get Order Details based on Email Id and Order Status.

In this case, we are passing values of Email ID and Order Status to find out Order Details. Now we can take **Path variables** here to fulfil this requirement.

- Create an endpoints for adding Order Details and Getting Order Details with Email ID and Order Status.

Create Request, Response and Entity Classes.

- **OrderRequest.java**

```

package com.swiggy.order.request;

public class OrderRequest {
```

```

private String orderID;
private String orderstatus;
private double amount;
private String emailId;
private String city;

public String getOrderID() {
    return orderID;
}
public void setOrderID(String orderID) {
    this.orderID = orderID;
}
public String getOrderstatus() {
    return orderstatus;
}
public void setOrderstatus(String orderstatus) {
    this.orderstatus = orderstatus;
}
public double getAmount() {
    return amount;
}
public void setAmount(double amount) {
    this.amount = amount;
}
public String getEmailId() {
    return emailId;
}
public void setEmailId(String emailId) {
    this.emailId = emailId;
}
public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
}

```

➤ OrderResponse.java

```

package com.swiggy.order.response;

public class OrderResponse {

    private String orderID;
    private String orderstatus;
    private double amount;
}

```

```
private String emailId;
private String city;

public OrderResponse() {
}

public OrderResponse(String orderId, String orderstatus, double amount,
                     String emailId, String city) {
    this.orderId = orderId;
    this.orderstatus = orderstatus;
    this.amount = amount;
    this.emailId = emailId;
    this.city = city;
}

public String getOrderId() {
    return orderId;
}

public void setOrderId(String orderId) {
    this.orderId = orderId;
}

public String getOrderstatus() {
    return orderstatus;
}

public void setOrderstatus(String orderstatus) {
    this.orderstatus = orderstatus;
}

public double getAmount() {
    return amount;
}

public void setAmount(double amount) {
    this.amount = amount;
}

public String getEmailId() {
    return emailId;
}

public void setEmailId(String emailId) {
    this.emailId = emailId;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}
```

- Entity Class : [SwiggyOrders.java](#)

```
package com.swiggy.order.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "swiggy_orders")
public class SwiggyOrders {

    @Id
    @Column
    private String orderID;
    @Column
    private String orderstatus;
    @Column
    private double amount;
    @Column
    private String emailId;
    @Column
    private String city;

    public String getOrderID() {
        return orderID;
    }
    public void setOrderID(String orderID) {
        this.orderID = orderID;
    }
    public String getOrderstatus() {
        return orderstatus;
    }
    public void setOrderstatus(String orderstatus) {
        this.orderstatus = orderstatus;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
}
```

```

public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
}

```

➤ OrderController.java

```

package com.swiggy.order.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.swiggy.order.request.OrderRequest;
import com.swiggy.order.response.OrderResponse;
import com.swiggy.order.service.OrderService;

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

    @Autowired
    OrderService orderService;

    @PostMapping(value = "/create")
    public String createOrder(@RequestBody OrderRequest request) {
        return orderService.createOrder(request);
    }

    @GetMapping("/email/{emailId}/status/{status}")
    public List<OrderResponse> getOrdersByemailIDAndStaus(@PathVariable String
emailId,
                                                                @PathVariable("status") String orderStaus){
        List<OrderResponse> orders =
            orderService.getOrdersByemailIDAndStaus(emailId, orderStaus);
        return orders;
    }
}

```

➤ Now create methods in Service layer.

```

package com.swiggy.order.service;

```

```

import java.util.List;
import java.util.stream.Collectors;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.swiggy.order.entity.SwiggyOrders;
import com.swiggy.order.repository.OrderRepository;
import com.swiggy.order.request.OrderRequest;
import com.swiggy.order.response.OrderResponse;

@Service
public class OrderService {

    @Autowired
    OrderRepository orderRepository;

    public String createOrder(OrderRequest request) {
        SwiggyOrders order = new SwiggyOrders();
        order.setAmount(request.getAmount());
        order.setCity(request.getCity());
        order.setEmailId(request.getEmailId());
        order.setOrderID(request.getOrderID());
        order.setOrderstatus(request.getOrderstatus());
        orderRepository.save(order);
        return "Order Created Successfully";
    }

    public List<OrderResponse> getOrdersByemailIDAndStatus(String emailId,
                                                          String orderStatus) {
        List<SwiggyOrders> orders =
            orderRepository.findByEmailIdAndOrderstatus(emailId, orderStatus);

        List<OrderResponse> allOrders = orders.stream().map(
            v -> new OrderResponse(
                v.getOrderID(),
                v.getOrderstatus(),
                v.getAmount(),
                v.getEmailId(),
                v.getCity()
            )).collect(Collectors.toList());

        return allOrders;
    }
}

```

➤ Create Repository : OrderRepository.java

Add JPA Derived Query `findBy()` Method for Email Id and Order Status.

```

package com.swiggy.order.repository;

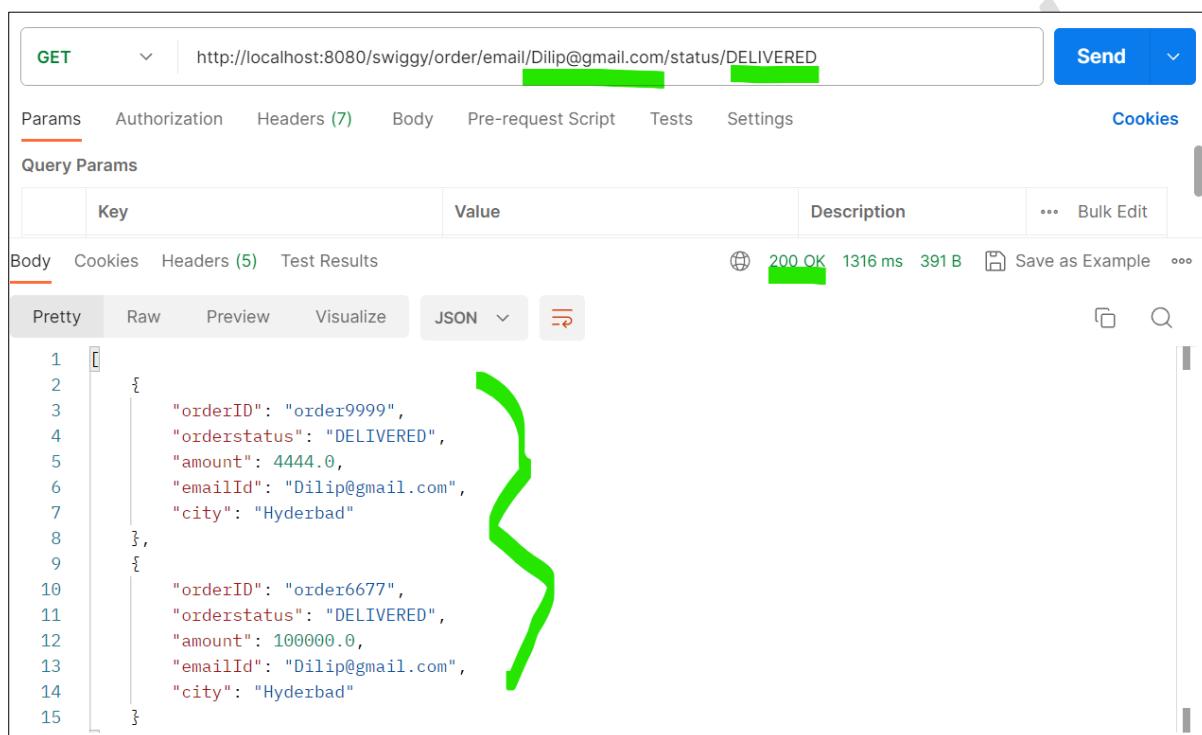
import java.util.List;

```

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.swiggy.order.entity.SwiggyOrders;

@Repository
public interface OrderRepository extends JpaRepository<SwiggyOrders, String>{
    List<SwiggyOrders> findByEmailIdAndOrderstatus(String emailId, String orderStatus);
}
```

From Postman Test end point: URL formation, replacing Path variables with real values



The screenshot shows a Postman test endpoint. The URL is `http://localhost:8080/swiggy/order/email/Dilip@gmail.com/status/DELIVERED`. The response status is 200 OK, with a response time of 1316 ms and a size of 391 B. The response body is a JSON array:

```

1 [
2   {
3     "orderID": "order9999",
4     "orderstatus": "DELIVERED",
5     "amount": 4444.0,
6     "emailId": "Dilip@gmail.com",
7     "city": "Hyderabad"
8   },
9   {
10    "orderID": "order6677",
11    "orderstatus": "DELIVERED",
12    "amount": 100000.0,
13    "emailId": "Dilip@gmail.com",
14    "city": "Hyderabad"
15  }
]

```

- We can also handle more than one Path Variables of URI by using a method parameter of type `java.util.Map<String, String>`.

```
@GetMapping("/pharmacy/{location}/pincode/{pincode}")
public String getPharmacyByLocationAndPincode(@PathVariable Map<String, String> values) {
    String location = values.get("location"); // Key is Path variable
    String pincode = values.get("pincode");

    return "Location Name : " + location + ", Pin code: " + pincode;
}
```

Query String and Query Parameters:

Query string is a part of a uniform resource locator (URL) that assigns values to specified parameters. A query string commonly includes fields added to a base URL by a Web browser or other client application. Let's understand this statement in a simple way by an example. Suppose we have filled out a form on websites and if we have noticed the URL something like as shown below as follows:

`http://internet.org/process-homepage?number1=23&number2=12`

So in the above URL, the query string is whatever follows the question mark sign ("?") i.e (number1=23&number2=12) this part. And "number1=23", "number2=12" are Query Parameters which are joined by a connector "&".

Let us consider another URL something like as follows:

`http://internet.org?title=Query_string&action=edit`

So in the above URL, the query string is "`title=Query_string&action=edit`" this part. And "`title=Query_string`", "`action=edit`" are Query Parameters which are joined by a connector "&".

Now we are discussing the concept of the query string and query parameter from the Spring MVC point of view. Developing Spring MVC application and will understand how query strings and query parameters are generated.

@RequestParam:

In Spring, we use `@RequestParam` annotation to extract the id of query parameters. Assume we have Users Data, and we should get data based on email Id.

Example : URL : `/details?email=<value-of-email>`

```
@GetMapping("/details")
public String getUserDetails(@RequestParam String email) {
    //Now we can pass Email Id to service layer to fetch user details
    return "Email Id of User : " + email;
}
```

Example with More Query Parameters :

Requirement: Please Get User Details by using either email or mobile number

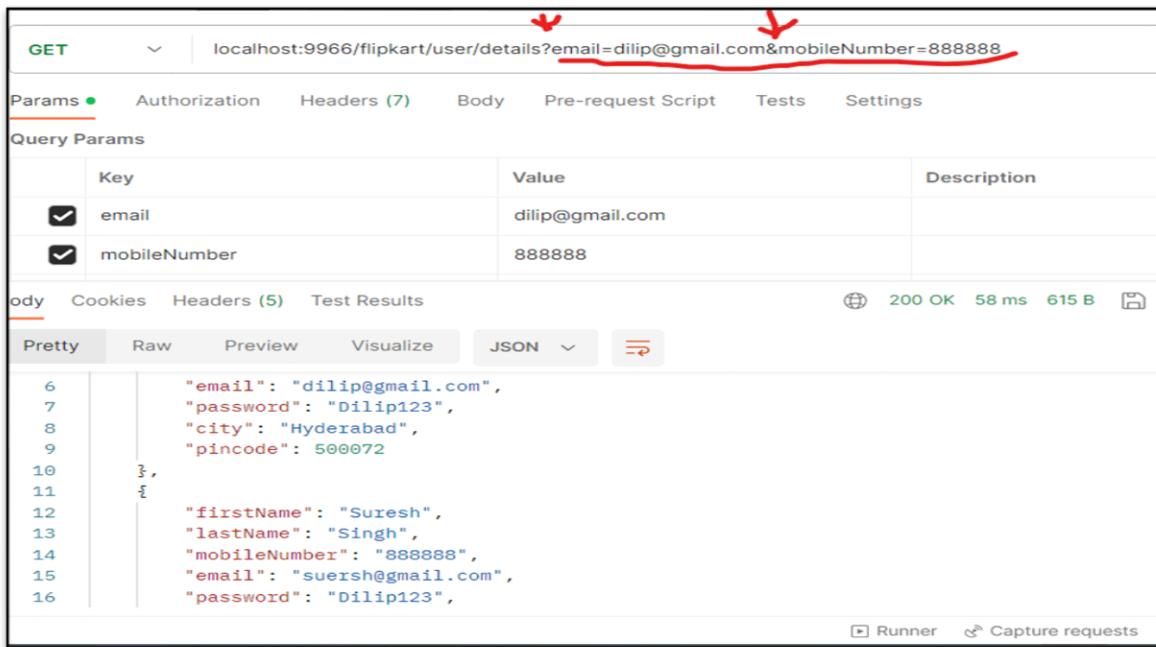
Method in controller:

```
@GetMapping("/details")
public List<Users> getUsersByEmailOrMobile(@RequestParam String email,
                                              @RequestParam String mobileNumber) {
    //Now we can pass Email Id and Mobile Number to service layer to fetch user
    //details
}
```

```
List<Users> response = service.getUsersByEmailOrMobile(email, mobileNumber);
return response;
}
```

NOTE: Add Service, Repository layers.

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



The screenshot shows a POSTMAN interface with a GET request. The URL is `localhost:9966/flipkart/user/details?email=dilip@gmail.com&mobileNumber=888888`. The 'Params' tab is selected, showing two query parameters: 'email' with value `dilip@gmail.com` and 'mobileNumber' with value `888888`. The 'Body' tab shows a JSON response:

```

{
  "email": "dilip@gmail.com",
  "password": "Dilip123",
  "city": "Hyderabad",
  "pincode": 500072
},
{
  "firstName": "Suresh",
  "lastName": "Singh",
  "mobileNumber": "888888",
  "email": "suresh@gmail.com",
  "password": "Dilip123",
}

```

The status bar at the bottom indicates `200 OK 58 ms 615 B`.

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**.

ResponseType:

ResponseType 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. **ResponseType** 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.

Defined in Spring MVC as the return value from an Controller method:

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

Points to be noted:

1. We should Define **ResponseType<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 **ResponseType** class, and then we returning that **ResponseType** Object to HTTP Client.

Few Examples of Controller methods with ResponseEntity:

```
@RestController
public class NetBankingController {

    @PostMapping("/create")
    @ResponseStatus(value = HttpStatus.CREATED) //Using ResponseStatus Annotation
    public String createAccount(@RequestBody AccountDetails accountDetails) {
```

```

        return "Created Netbanking Account. Please Login.";
    }

    @PostMapping("/create/loan") //Using ResponseEntity Class
    public ResponseEntity<String> createLoan(@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.

Headers in Spring MVC:

HTTP headers are part of the Hypertext Transfer Protocol (HTTP), which is the foundation of data communication on the World Wide Web. They are metadata or key-value pairs that provide additional information about an HTTP request or response. Headers are used to convey various aspects of the communication between a client (typically a web browser) and a server.

HTTP headers can be classified into two main categories: request headers and response headers.

Request Headers:

Request headers are included in an HTTP request sent by a client to a server. They provide information about the client's preferences, the type of data being sent, authentication credentials, and more. Some common request headers include:

- **User-Agent:** Contains information about the user agent (usually a web browser) making the request.
- **Accept:** Specifies the media types (content types) that the client can process.
- **Authorization:** Provides authentication information for accessing protected resources.
- **Cookie:** Sends previously stored cookies back to the server.
- **Content-Type:** Specifies the format of the data being sent in the request body.

Response Headers:

Response headers are included in an HTTP response sent by the server to the client. They convey information about the server's response, the content being sent, caching directives, and more. Some common response headers include:

- **Content-Type:** Specifies the format of the data in the response body.
- **Content-Length:** Specifies the size of the response body in bytes.
- **Set-Cookie:** Sets a cookie in the client's browser for managing state.

HTTP headers are important for various purposes, including negotiating content types, enabling authentication, handling caching, managing sessions, and more. They allow both clients and servers to exchange additional information beyond the basic request and response data. Proper understanding and usage of HTTP headers are essential for building efficient and secure web applications.

Spring MVC provides mechanisms to work with HTTP headers both in requests and responses.

Here's how you can work with HTTP headers in Spring MVC.

Handling Request Headers:

Accessing Request Headers: Spring provides the **@RequestHeader** annotation that allows you to access specific request headers in your controller methods. You can use this annotation as a method parameter to extract header values.

In Spring Framework's MVC module, **@RequestHeader** is an annotation used to extract values from HTTP request headers and bind them to method parameters in your controller methods. This annotation is part of Spring's web framework and is commonly used to access and work with the values of specific request headers.

```
@GetMapping("/endpoint")
public ResponseEntity<String> handleRequest(@RequestHeader("Header-Name") String
```

```

        headerValue) {

    // Do something with the header and other values
}

```

Example: Defined a header `user-name` inside request:

Header and its Value should come from Client while they are triggering this endpoint.

```

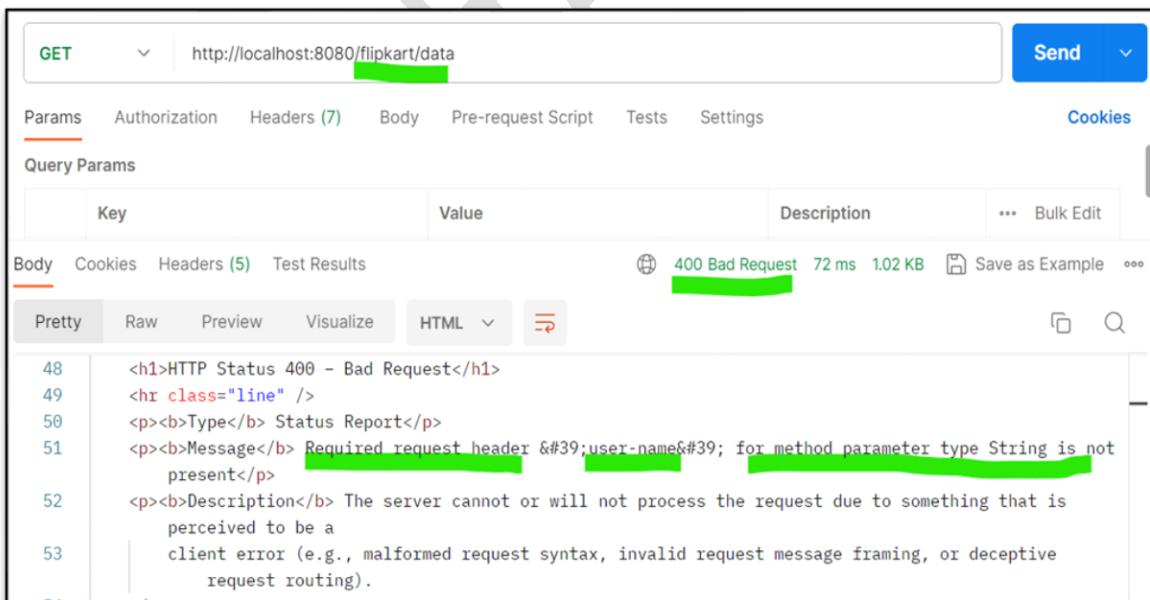
package com.flipkart.controller;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;

@RestController
public class OrderController {
    //Header is Part of Request, i.e. Should be Come from Client Side.
    @GetMapping("/data")
    public String testHeaders(@RequestHeader("user-name") String userName) {
        return "Connected User : " + userName;
    }
}

```

Testing: Without Sending Header and Value from Client, Sending Request to Service.

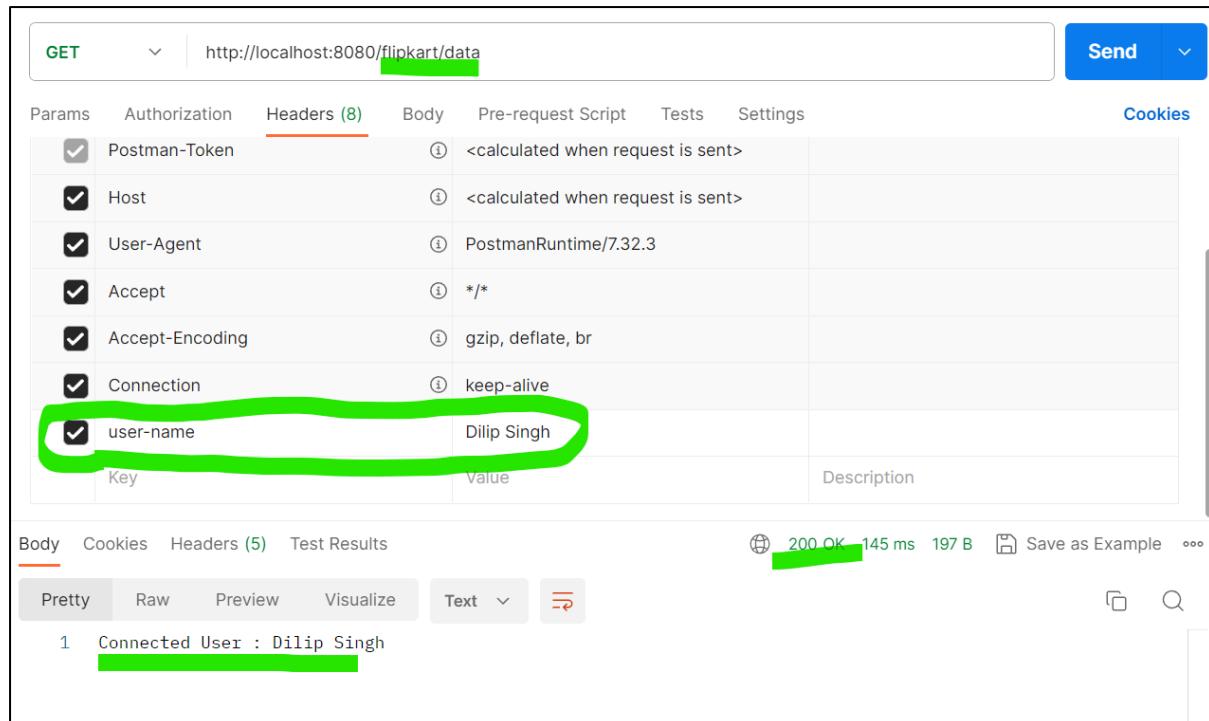


The screenshot shows a Postman interface with a failed API call. The URL is `http://localhost:8080/flipkart/data`. The Headers tab shows 7 items, including the required `user-name`. The response status is 400 Bad Request, with a detailed error message: "Required request header 'user-name' for method parameter type String is not present".

Result : We Got Bad request like Header is Missing i.e. Header is Mandatory by default if we defined in Controller method.

Setting Header in Client: i.e. In Our Case From Postman:

In Postman, Add header `user-name` and its value under Headers Section. Now request is executed Successfully.



GET Send

Params	Authorization	Headers (8)	Body	Pre-request Script	Tests	Settings	Cookies
<input checked="" type="checkbox"/>	Postman-Token	<small>(i) <calculated when request is sent></small>					
<input checked="" type="checkbox"/>	Host	<small>(i) <calculated when request is sent></small>					
<input checked="" type="checkbox"/>	User-Agent	<small>(i) PostmanRuntime/7.32.3</small>					
<input checked="" type="checkbox"/>	Accept	<small>(i) */*</small>					
<input checked="" type="checkbox"/>	Accept-Encoding	<small>(i) gzip, deflate, br</small>					
<input checked="" type="checkbox"/>	Connection	<small>(i) keep-alive</small>					
<input checked="" type="checkbox"/>	user-name	Dilip Singh					

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text ...

```
1 Connected User : Dilip Singh
```

Optional Header:

If we want to make Header as an Optional i.e. non mandatory. we have to add an attribute of **required** and Its value as **false**.

```
package com.flipkart.controller;

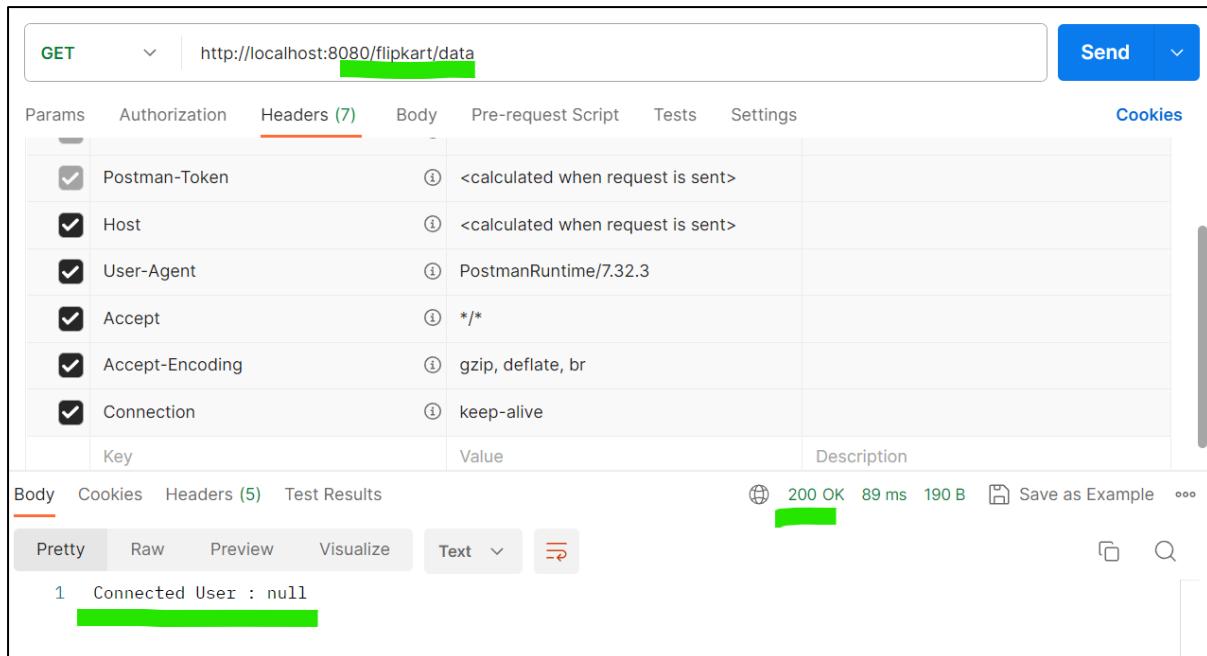
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    //Header is Part of Request, i.e. Should be Come from Client Side.
    @GetMapping("/data")
    public String testHeaders(@RequestHeader(name = "user-name",
                                             required = false) String userName) {
        return "Connected User : " + userName;
    }
}
```

Testing:

- No header Added, So Header value is null.



The screenshot shows a Postman request for a GET method to the URL `http://localhost:8080/flipkart/data`. The Headers section is active, displaying the following headers:

Key	Description
Postman-Token	<calculated when request is sent>
Host	<calculated when request is sent>
User-Agent	PostmanRuntime/7.32.3
Accept	*/*
Accept-Encoding	gzip, deflate, br
Connection	keep-alive

The Body tab shows the response details: 200 OK, 89 ms, 190 B. The response body is displayed as:

```
1 Connected User : null
```

Default Value Of Header:

- We can Set Header Default Value also in case if we are not getting it from Client. Add an attribute **defaultValue** and its value.

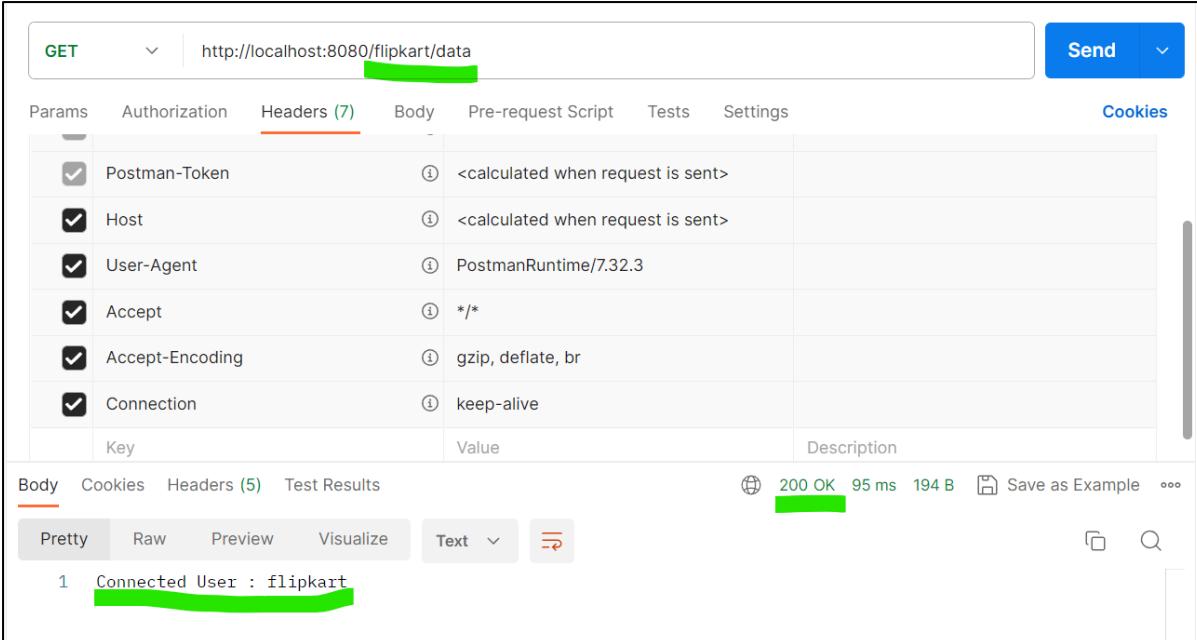
```
package com.flipkart.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    //Header is Part of Request, i.e. Should be Come from Client Side.
    @GetMapping("/data")
    public String testHeaders(@RequestHeader(name = "user-name", required = false,
                                             defaultValue = "flipkart") String userName) {
        return "Connected User : " + userName;
    }
}
```

Testing: Without adding Header and its value, triggering Service. Default Value of Header **flipkart** is considered by Server as per implementation.



The screenshot shows a Postman interface with a GET request to `http://localhost:8080/flipkart/data`. The Headers section contains the following entries:

Key	Description
Postman-Token	<calculated when request is sent>
Host	<calculated when request is sent>
User-Agent	PostmanRuntime/7.32.3
Accept	*/*
Accept-Encoding	gzip, deflate, br
Connection	keep-alive

The response status is 200 OK with a duration of 95 ms and a size of 194 B. The response body is "Connected User : flipkart".

Setting Response Headers:

In Spring MVC, response headers can be set using the **HttpServletResponse** object or the **ResponseEntity** class.

Here are some of the commonly used response headers in Spring MVC:

Content-Type: The MIME type of the response body.

Expires: The date and time after which the response should no longer be cached.

Last-Modified: The date and time when the resource was last modified.

The **HttpServletResponse** object is the standard way to set headers in a servlet-based application. To set a header using the **HttpServletResponse** object, you can use the **addHeader()** method.

For example:

```
HttpServletResponse response = request.getServletResponse();
response.addHeader("Content-Type", "application/json");
```

The **ResponseEntity** class is a more recent addition to Spring MVC. It provides a more concise way to set headers, as well as other features such as status codes and body content. To set a header using the **ResponseEntity** class, you can use the **headers()** method.

For example:

```
ResponseEntity<String> response = new ResponseEntity<>("Hello, world!", HttpStatus.OK);
response.headers().add("Content-Type", "application/json");
```

In another approach, We can create **HttpHeaders** instance and we can add multiple Headers and their values. After that, we can pass **HttpHeaders** instance to **ResponseEntity** Object.

HttpHeaders:

In Spring MVC, the **HttpHeaders** class is provided by the framework as a convenient way to manage HTTP headers in both request and response contexts. HttpHeaders is part of the **org.springframework.http** package, and it provides methods to add, retrieve, and manipulate HTTP headers. Here's how you can use the **HttpHeaders** class in Spring MVC:

In a Response:

You can use **HttpHeaders** to set custom headers in the HTTP response. This is often done when you want to include specific headers in the response to provide additional information to the client.

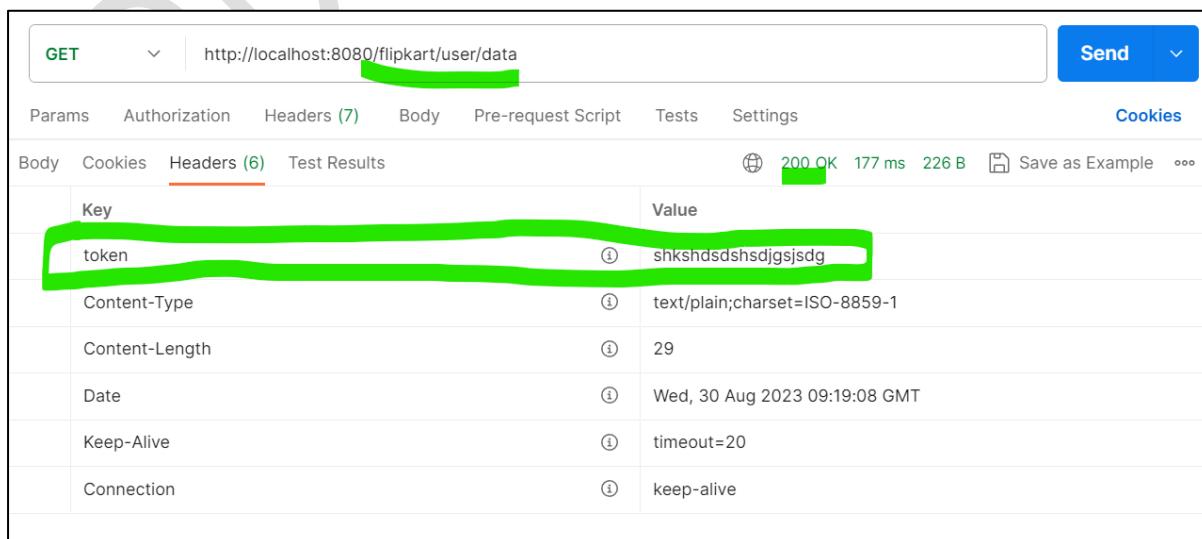
Example: Sending a Header and its value as part of response Body.

```
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    // Header is Part of Response, i.e. Should be set from Server Side.
    @GetMapping("/user/data")
    public ResponseEntity<String> testResponseHeaders() {
        HttpHeaders headers = new HttpHeaders();
        headers.set("token", "shkshdsdshsdjgsjsdg");
        return new ResponseEntity<String>("Sending Response with Headers", headers,
                                         HttpStatus.OK);
    }
}
```

Testing: Trigger endpoint from Client: Got Token and its value from Service in Headers.



Key	Value
token	shkshdsdshsdjgsjsdg
Content-Type	text/plain; charset=ISO-8859-1
Content-Length	29
Date	Wed, 30 Aug 2023 09:19:08 GMT
Keep-Alive	timeout=20
Connection	keep-alive

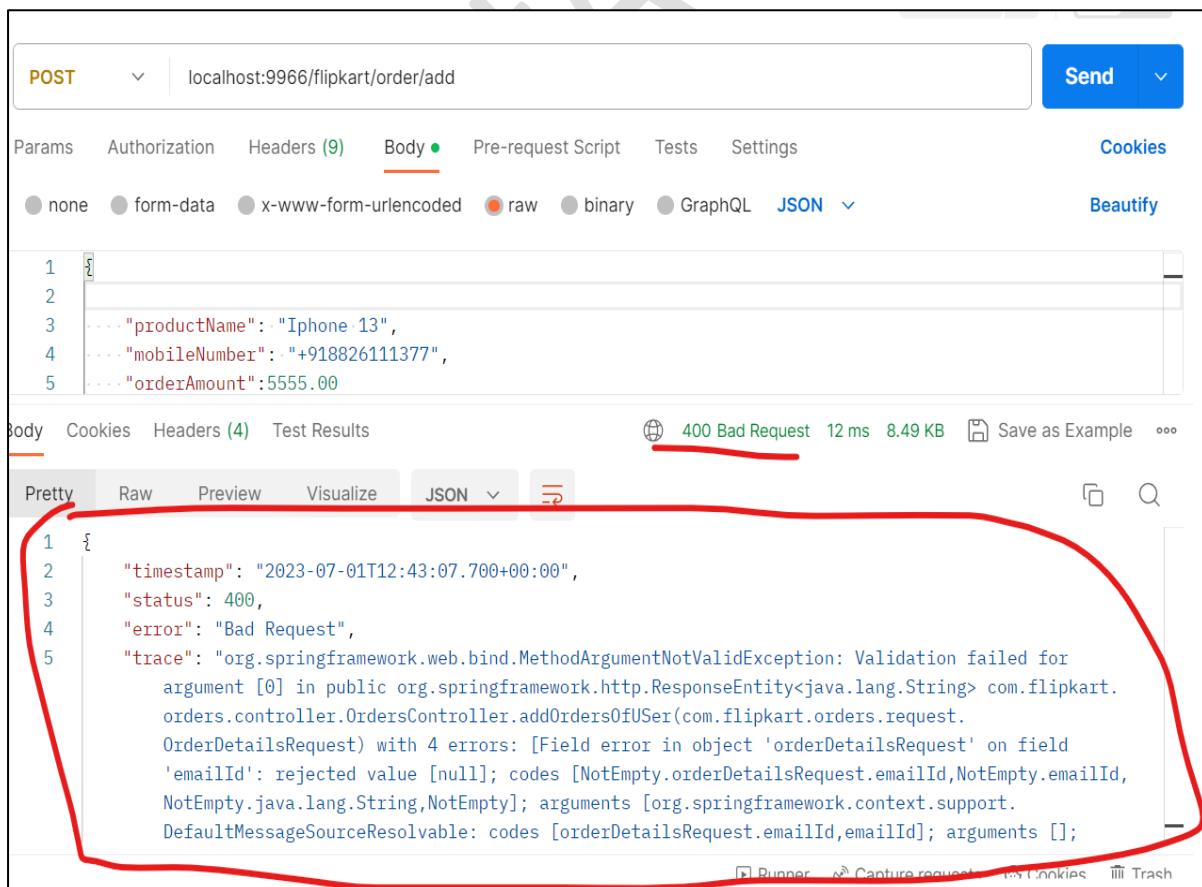
Exception Handling in Spring MVC Controllers:

What I have to do with Errors or Exceptions ?



Spring brought **@ExceptionHandler** & **@ControllerAdvice** annotations for handling Exceptions thrown at controller layer. So we can handle exceptions and will be forwarded meaning full Error response messages with response status code to HTTP clients.

If we are not handled exceptions then we will see Exception stack trace as shown in below at HTTP client level as a response. As a Best Practice we should show meaningful Error Response messages.



POST localhost:9966/flipkart/order/add

Params Authorization Headers (9) Body **JSON** Pre-request Script Tests Settings Cookies Beautify

```

1 {
2
3   "productName": "Iphone 13",
4   "mobileNumber": "+918826111377",
5   "orderAmount":5555.00

```

Body Cookies Headers (4) Test Results 400 Bad Request 12 ms 8.49 KB Save as Example

Pretty Raw Preview Visualize JSON

```

1 {
2   "timestamp": "2023-07-01T12:43:07.700+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "trace": "org.springframework.web.bind.MethodArgumentNotValidException: Validation failed for argument [0] in public org.springframework.http.ResponseEntity<java.lang.String> com.flipkart.orders.controller.OrdersController.addOrdersOfUser(com.flipkart.orders.request.OrderDetailsRequest) with 4 errors: [Field error in object 'orderDetailsRequest' on field 'emailId': rejected value [null]; codes [NotEmpty.orderDetailsRequest.emailId,NotEmpty.emailId,NotEmpty.java.lang.String,NotEmpty]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [orderDetailsRequest.emailId,emailId]; arguments []];"

```

Runner Capture request Cookies Trash

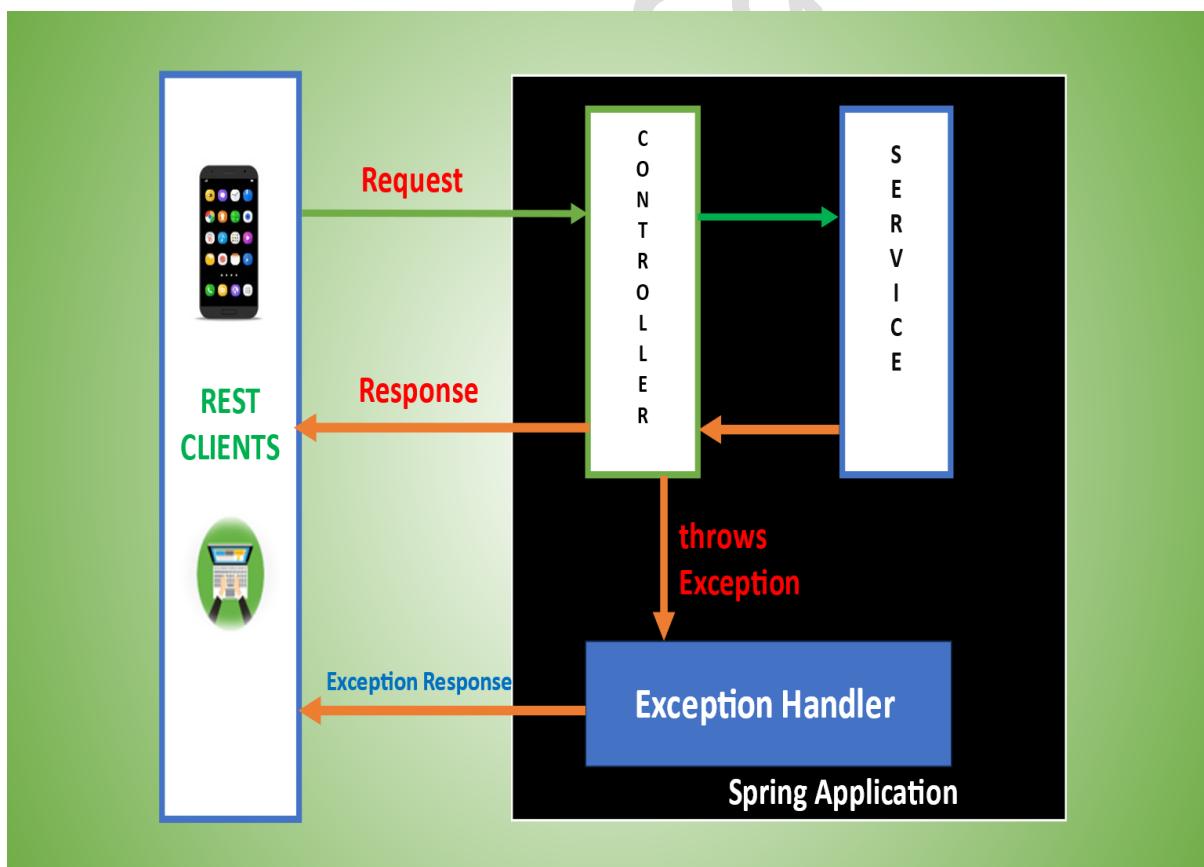
Note: Spring provided other ways as well to handle exceptions but controller advice and Exception handler will provide better way of exception handling.

@ExceptionHandler is a Spring annotation that provides a mechanism to treat exceptions thrown during execution of handlers (controller operations). This annotation, if used on methods of controller classes, will serve as the entry point for handling exceptions thrown within this controller only.

Altogether, the most common implementation is to use **@ExceptionHandler** on methods of **@ControllerAdvice** classes so that the Spring Boot exception handling will be applied globally or to a subset of controllers.

ControllerAdvice is an annotation in Spring and, as the name suggests, is “advice” for all controllers. It enables the application of a single **ExceptionHandler** to multiple controllers. With this annotation, we can define how to treat an exception in a single place, and the system will call this exception handler method for thrown exceptions on classes covered by this **ControllerAdvice**.

By using **@ExceptionHandler** and **@ControllerAdvice**, we’ll be able to define a central point for treating exceptions and wrapping them in an Error object with the default Spring Boot error-handling mechanism.



Solution 1: Controller-Level @ExceptionHandler:

The first solution works at the **@Controller** level. We will define a method to handle exceptions and annotate that with **@ExceptionHandler** i.e. We can define Exception Handler Methods in side controller classes. This approach has a major drawback: The **@ExceptionHandler** annotated method is only active for that particular Controller, not globally for the entire application. But better practice is writing a separate controller advice classes dedicatedly handle different exception at one place.

```
@RestController
public class FooController{

    // Endpoint Methods

    @ExceptionHandler({ ExceptionName.class, ExceptionName.class })
    public void handleException() {
        //
    }
}
```

Solution 2: @ControllerAdvice:

The **@ControllerAdvice** annotation allows us to consolidate multiple Exception Types with 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 **ResponseEntity** response.

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

Example of Controller Advice class : Controller Advice With Exception Handler methods

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import jakarta.servlet.http.HttpServletRequest;
```

```
@ControllerAdvice
```

```

public class OrderControllerExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?>
handleMethodArgumentException(MethodArgumentNotValidException ex,
                               HttpServletRequest rq) {

    List<String> messages = ex.getFieldErrors().stream().map(e ->
        e.getDefaultMessage()).collect(Collectors.toList());
    return new ResponseEntity<>( messages, HttpStatus.BAD_REQUEST);
}

    @ExceptionHandler(NullPointerException.class)
    public ResponseEntity<?> handleNullpointerException(NullPointerException ex,
HttpServletRequest request) {

    return new ResponseEntity<>("Please Check data, getting as null values",
HttpStatus.INTERNAL_SERVER_ERROR);
}

    @ExceptionHandler(ArithmaticException.class)
    public ResponseEntity<?> handleArithmaticException(ArithmaticException 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.

POST localhost:9966/flipkart/order/add

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Body Cookies Headers (4) Test Results

400 Bad Request 12 ms 243 B

Body JSON

```

1
2     "emailId": "dilip@gmail.com",
3     "productName": "Iphone 13",
4     "mobileNumber": "+918826111377",
5     "orderAmount": 5555.00
6

```

Pretty Raw Preview Visualize JSON

```

1
2     "errors": [
3         "orderID should not be null",
4         "emailId is invalid format",
5         "orderID should not be empty"
6     ]
7

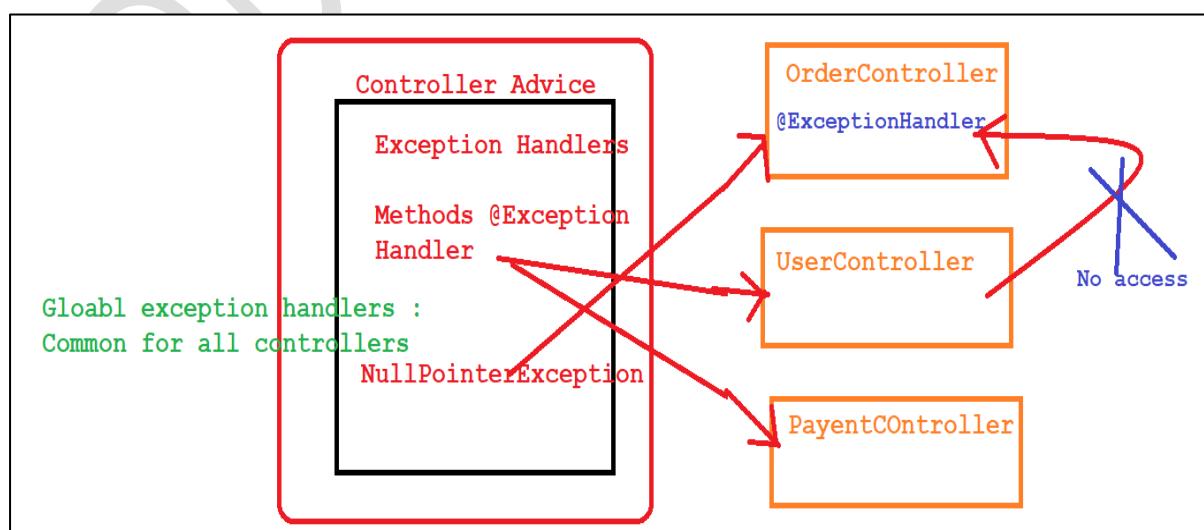
```

How it is working?

Whenever an exception occurred at controller layer due to any reason, immediately controller will check for relevant exceptions handled as part of Exception Handler or not. If handled, then that specific exception handler method will be executed and response will be forwarded to clients. If not handled, then entire exception error stack trace will be forwarded to client as it's not suggestable.

Which Exception takes Priority if we defined Child and Parent Exceptions Handlers?

From above example, if **NullPointerException** occurred then **handleNullpointerException()** method will be executed even though we have logic for parent **Exception** handling i.e. Priority given to child exception if we handled and that will be returned as response data. Similarly we can define multiple controller advice classes with different types of Exceptions along with relevant Http Response Status Codes.



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:

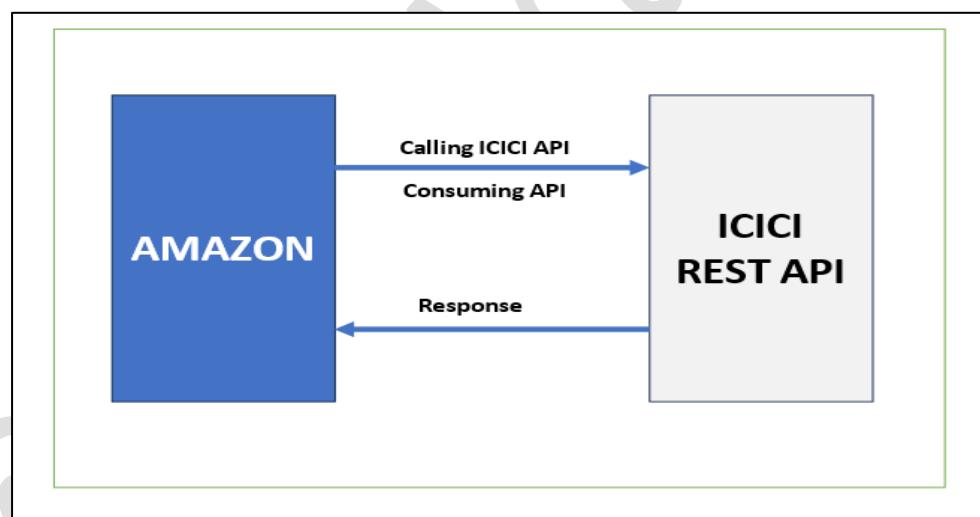
Consuming REST services is nothing but integrating other application REST API services from our application logic.

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:

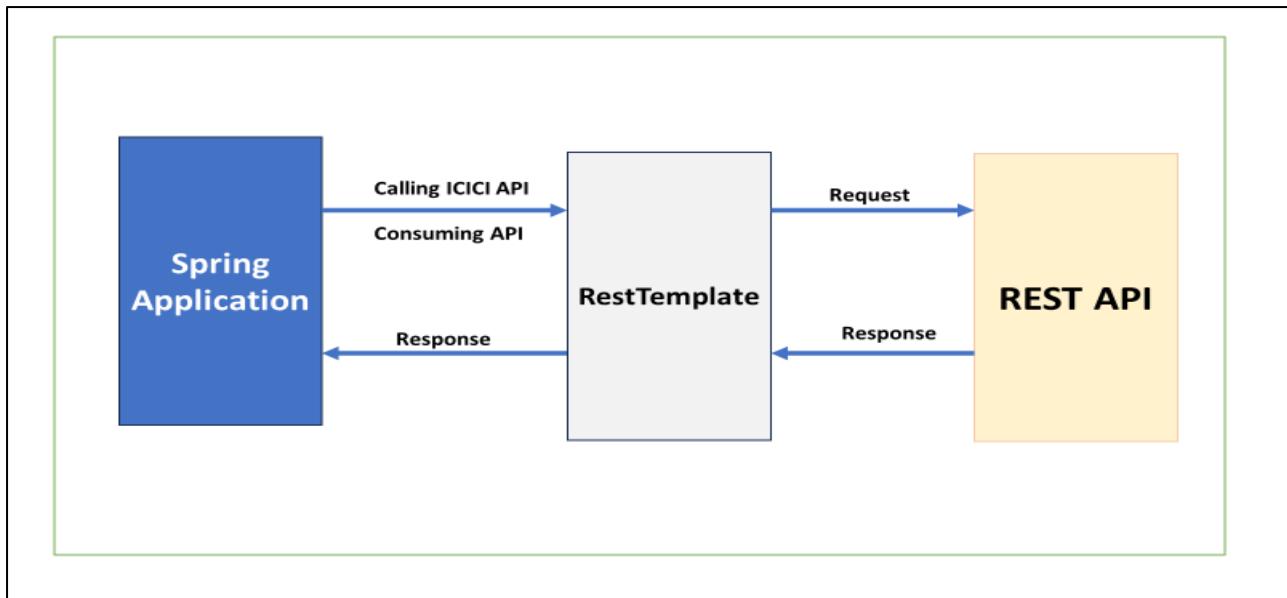
Producer is : ICICI

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 a class provided by Spring Framework that simplifies the process of making HTTP requests to external RESTful APIs or web services. It abstracts away the low-level details of creating and managing HTTP connections, sending requests, and processing responses. **RestTemplate** provides a higher-level API for working with RESTful resources.



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**.

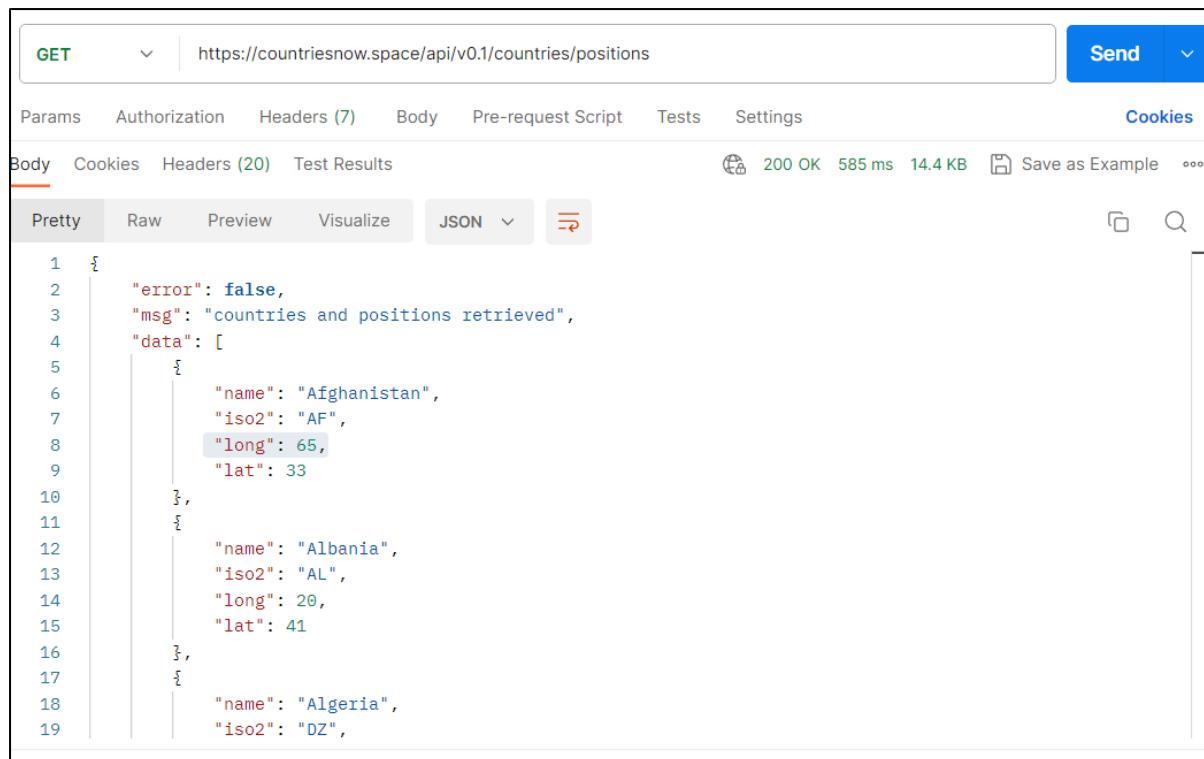
Now I am giving only API details of Pharmacy Application as swagger documentation. Because in Realtime Projects, swagger documentation or Postman collection data will be shared to Developers team, but not source code. So we will try to consume by seeing Swagger API documentation of tother application.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

Example : 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.



The screenshot shows a POSTMAN API request for `https://countriesnow.space/api/v0.1/countries/positions`. The response status is 200 OK with a time of 585 ms and a size of 14.4 KB. The response body is a JSON object:

```

1  {
2    "error": false,
3    "msg": "countries and positions retrieved",
4    "data": [
5      {
6        "name": "Afghanistan",
7        "iso2": "AF",
8        "long": 65,
9        "lat": 33
10       },
11       {
12         "name": "Albania",
13         "iso2": "AL",
14         "long": 20,
15         "lat": 41
16       },
17       {
18         "name": "Algeria",
19         "iso2": "DZ",

```

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);

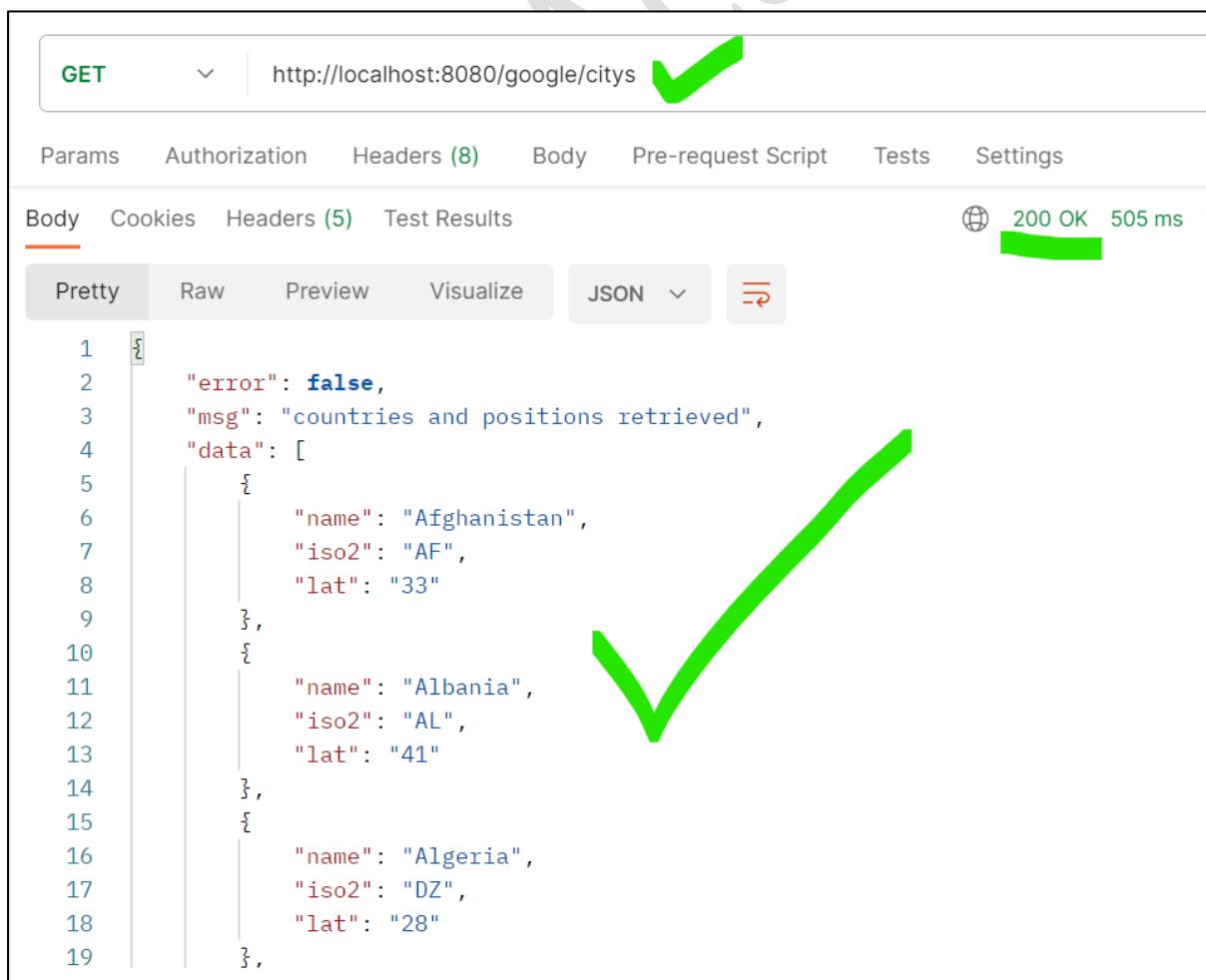
    RestTemplate restTemplate = new RestTemplate();
    CountriesResponse respose = restTemplate.exchange(url, HttpMethod.GET,
            entity,
            CountriesResponse.class).getBody();

    System.out.println(respose);

    return respose;
}

```

Testing from our Application:



GET http://localhost:8080/google/citys

Params	Authorization	Headers (8)	Body	Pre-request Script	Tests	Settings
Body	Cookies	Headers (5)	Test Results		200 OK	505 ms

Pretty Raw Preview Visualize JSON ↻

```

1  {
2      "error": false,
3      "msg": "countries and positions retrieved",
4      "data": [
5          {
6              "name": "Afghanistan",
7              "iso2": "AF",
8              "lat": "33"
9          },
10         {
11             "name": "Albania",
12             "iso2": "AL",
13             "lat": "41"
14         },
15         {
16             "name": "Algeria",
17             "iso2": "DZ",
18             "lat": "28"
19         }
]

```

XML Request and XML Response in REST API:

As of Now, By Default we are used JSON Data Format for Request and Responses in REST API implementation. Now we are going to see, how to support XML Data Formats in Request and Responses of REST Services along with JSON Data Format.

In Spring MVC, for formatting data from JSON to JAVA and vice versa We used Jackson API dependencies. Similarly for XML data format also, we have to add dependencies for XML to Java and Java to XML data conversions. Jackson API provided support for XML data format as well. So add below dependency in existing project pom.xml file. So that, now our project will support both JSON and XML data formats in REST services.

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.15.2</version>
</dependency>
```

Now Create REST Endpoints/Services with Request and Responses of either JSON or XML data formats.

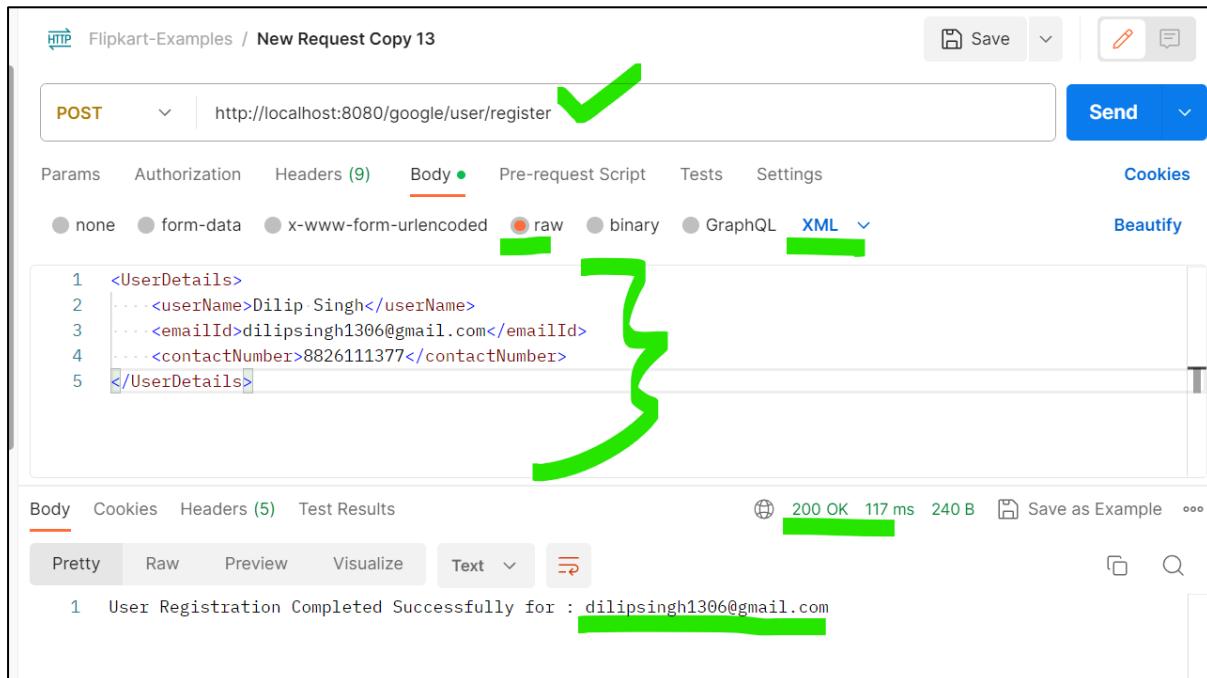
Create Rest Service accepting Request Data either XML or JSON Data Format:

➤ Inside Controller class: Create a New Service with Request Body:

Below Service will accept either JSON or XML data formats as Request Payload. We have to send an Header **Content-Type** and its associated values **application/xml** or **application/json** depends on our input data format. With This Header and its value our Spring MVC converts data to POJO or vice versa. In case, if we missed **Content-Type** header, Spring MVC will throw Exception with status code “**415 Unsupported Media Type**”.

```
@PostMapping("/user/register")
public String RegisterUserInfo(@RequestBody UserDetails userDetails) {
    //Transfer Request Data to Service and Service to Repository
    System.out.println("Registering User Data for :" + userDetails.getEmailId());
    return "User Registration Completed Successfully for :" + userDetails.getEmailId();
}
```

Now, Test Above REST API Service by passing either JSON or XML Request payload Data.

XML Payload: Postman: Please Select Data Type as XML in Body-> raw -> XML


HTTP Flipkart-Examples / New Request Copy 13

POST http://localhost:8080/google/user/register

Params Authorization Headers (9) **Body** • Pre-request Script Tests Settings Cookies Beautify

Body Type: XML

```

1 <UserDetails>
2   ...<userName>Dilip Singh</userName>
3   ...<emailId>dilipsingh1306@gmail.com</emailId>
4   ...<contactNumber>8826111377</contactNumber>
5 </UserDetails>

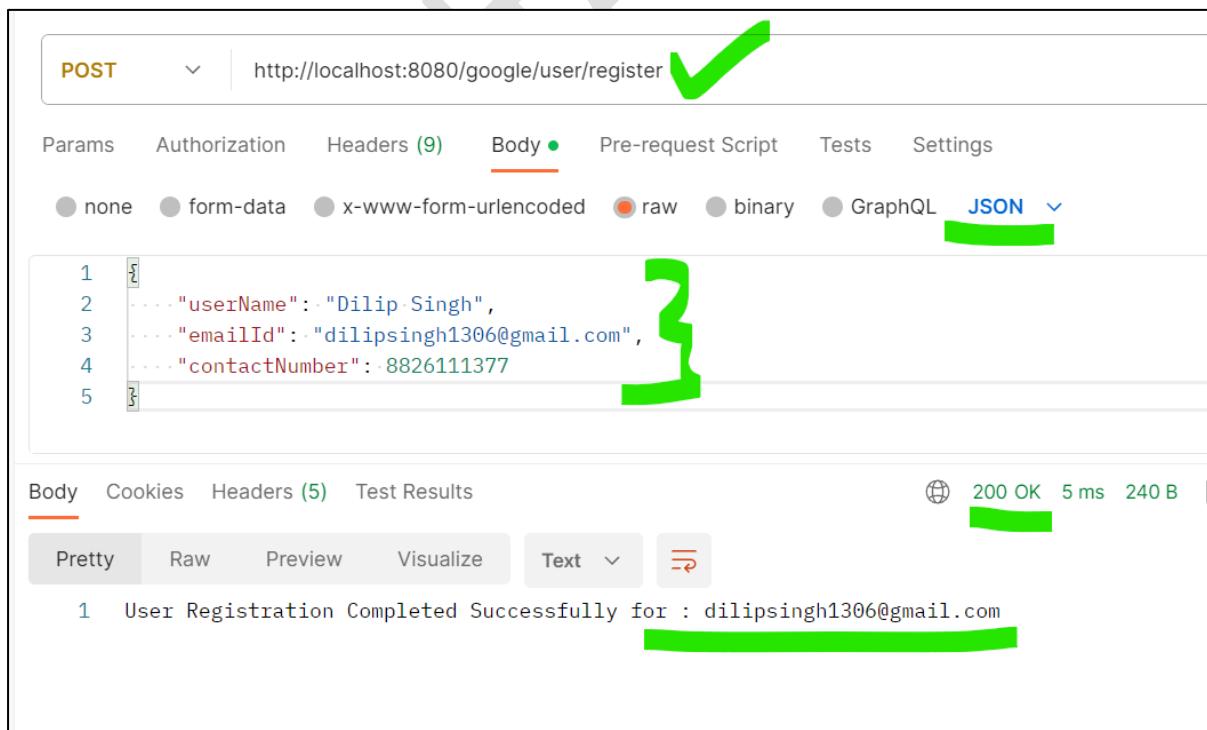
```

Body Cookies Headers (5) Test Results

200 OK 117 ms 240 B Save as Example

Pretty Raw Preview Visualize Text

User Registration Completed Successfully for : dilipsingh1306@gmail.com

JSON Payload: Postman : Please Select Data Type as JSON in Body-> raw -> JSON


POST http://localhost:8080/google/user/register

Params Authorization Headers (9) **Body** • Pre-request Script Tests Settings

Body Type: JSON

```

1 {
2   ... "userName": "Dilip Singh",
3   ... "emailId": "dilipsingh1306@gmail.com",
4   ... "contactNumber": 8826111377
5 }

```

Body Cookies Headers (5) Test Results

200 OK 5 ms 240 B

Pretty Raw Preview Visualize Text

User Registration Completed Successfully for : dilipsingh1306@gmail.com

Creating REST Service which Returns Response wither XML or JSON Data Format.

```

@GetMapping("/user/data")
public UserDetails getUserInfo() {
    //Assume Data Received from Service Layer
    UserDetails userData = new UserDetails();
    userData.setUserName("Dilip Singh");
    userData.setEmailId("dilipsingh1306@gmail.com");
    userData.setContactNumber(8826111377);
    return userData;
}

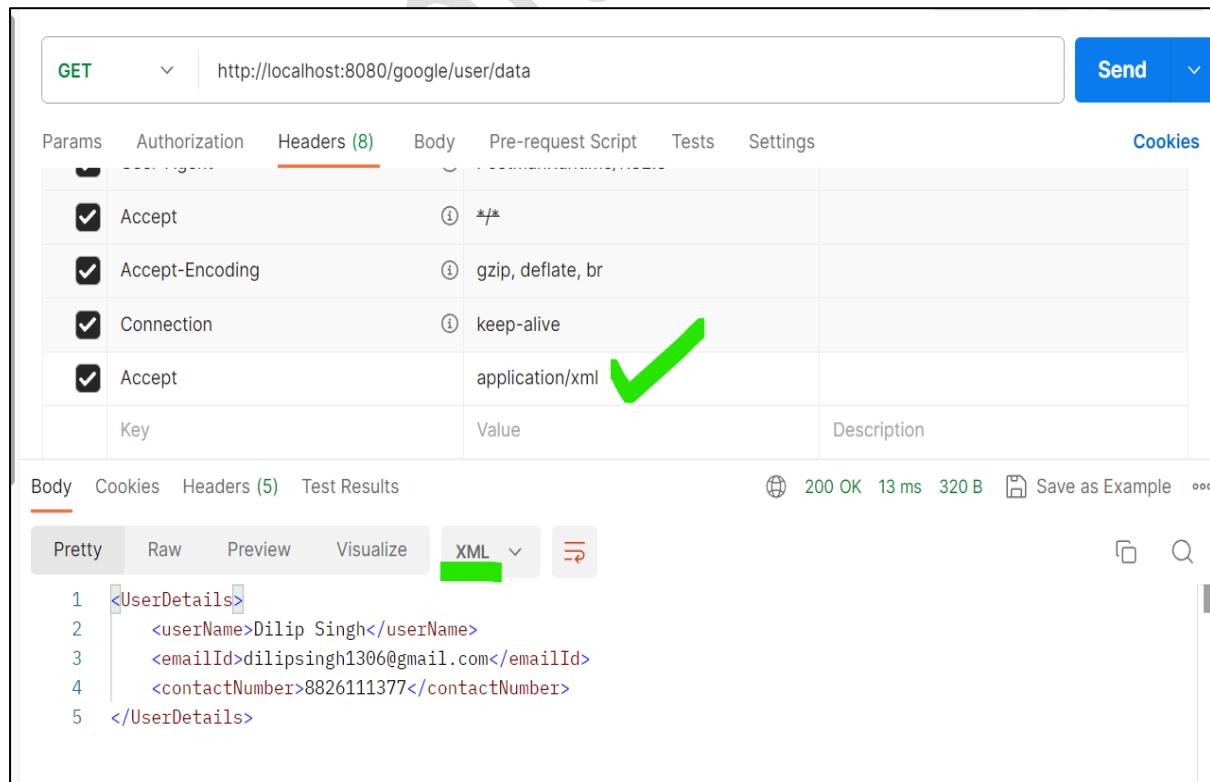
```

Now Test From Postman :

1. If we want to get Response as XML format , then add a Request header “accept : application/xml” and send it to Server as part off Request.
2. If we want to get Response as JSON format , then add a Request header “accept : application/json” and send it to Server as part off Request.

Based on header data Spring application processing Response Data Format and return back to Client level.

XML Response:



The screenshot shows a Postman request for `http://localhost:8080/google/user/data`. The Headers tab is selected, showing the following configuration:

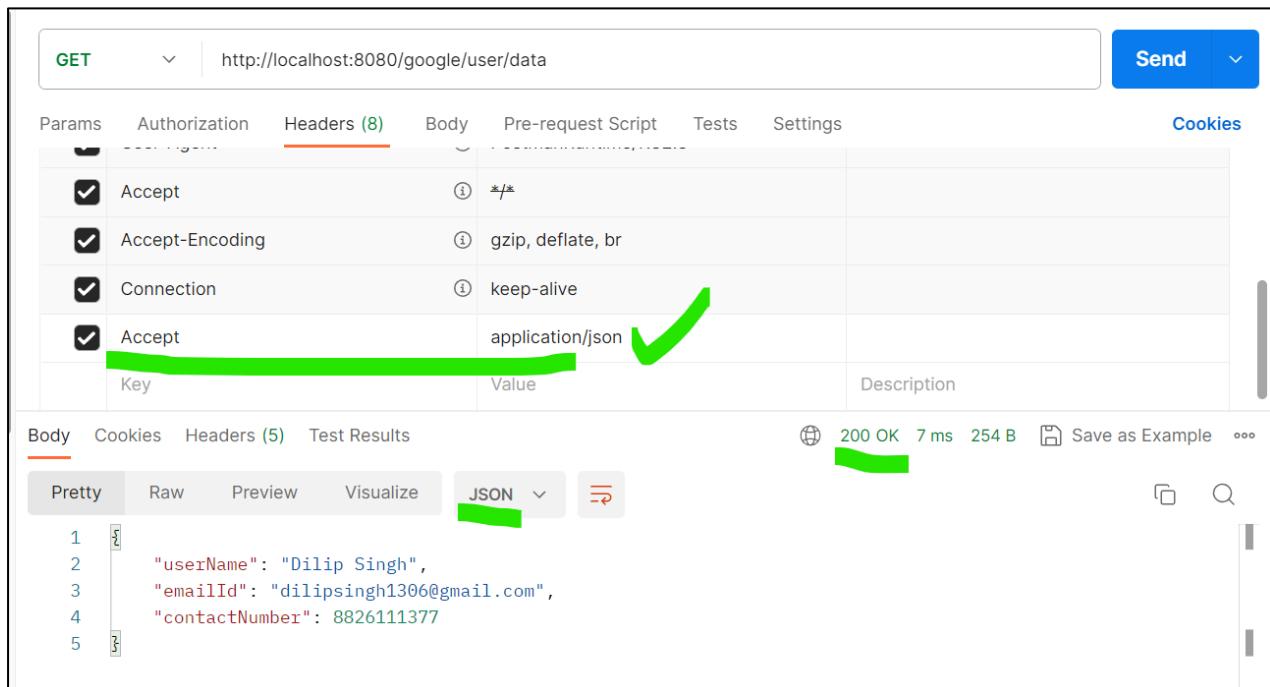
Key	Value
Accept	application/xml

The 'Pretty' option is selected in the Body tab, displaying the XML response:

```

1 <UserDetails>
2   <userName>Dilip Singh</userName>
3   <emailId>dilipsingh1306@gmail.com</emailId>
4   <contactNumber>8826111377</contactNumber>
5 </UserDetails>

```

JSON Response:

The screenshot shows a Postman request for a GET operation on the URL `http://localhost:8080/google/user/data`. The Headers tab is selected, displaying the following configuration:

Key	Value
Accept	*
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
Accept	application/json

The 'Accept' header entry has a green checkmark next to it. Below the headers, the Body tab is selected, showing a JSON response:

```
1 {  
2   "userName": "Dilip Singh",  
3   "emailId": "dilipsingh1306@gmail.com",  
4   "contactNumber": 8826111377  
5 }
```

The response status is 200 OK with 7 ms latency and 254 B size. A green checkmark is also present near the response body.

Spring

Security Module

Spring Security:

Spring Security is a powerful and highly customizable authentication and access-control framework. Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements. It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application. The main goal of Spring Security is to make it easy to add security features to your applications. It follows a modular design, allowing you to choose and configure various components according to your specific requirements. Some of the key features of Spring Security include:

Authentication: Spring Security supports multiple authentication mechanisms, such as form-based, HTTP Basic, HTTP Digest, and more. It integrates seamlessly with various authentication providers, including in-memory, JDBC, LDAP, and OAuth.

Authorization: Spring Security enables fine-grained authorization control based on roles, permissions, and other attributes. It provides declarative and programmatic approaches for securing application resources, such as URLs, methods, and domain objects.

Session Management: Spring Security offers session management capabilities, including session fixation protection, session concurrency control, and session timeout handling. It allows you to configure session-related properties and customize session management behaviour.

Security Filters: Spring Security leverages servlet filters to intercept and process incoming requests. It includes a set of predefined filters for common security tasks, such as authentication, authorization, and request/response manipulation. You can easily configure and extend these filters to meet your specific needs.

Integration with Spring Framework: Spring Security seamlessly integrates with the Spring ecosystem. It can leverage dependency injection and aspect-oriented programming features provided by the Spring Framework to enhance security functionality.

Customization and Extension: Spring Security is highly customizable, allowing you to override default configurations, implement custom authentication/authorization logic, and integrate with third-party libraries or existing security infrastructure.

Overall, Spring Security simplifies the process of implementing robust security features in Java applications. It provides a flexible and modular framework that addresses common security concerns and allows developers to focus on building secure applications.

This module targets two major areas of application are **authentication** and **authorization**.

What is Authentication?

Authentication in Spring refers to the process of verifying the identity of a user or client accessing a system or application. It is a crucial aspect of building secure applications to ensure that only authorized individuals can access protected resources.

In the context of Spring Security, authentication involves validating the credentials provided by the user and establishing their identity. Spring Security offers various authentication mechanisms and supports integration with different authentication providers.

Here's a high-level overview of how authentication works in Spring Security:

User provides credentials: The user typically provides credentials, such as a username and password, in order to authenticate themselves.

Authentication request: The application receives the user's credentials and creates an authentication request object.

Authentication manager: The authentication request is passed to the authentication manager, which is responsible for validating the credentials and performing the authentication process.

Authentication provider: The authentication manager delegates the actual authentication process to one or more authentication providers. An authentication provider is responsible for verifying the user's credentials against a specific authentication mechanism, such as a user database, LDAP server, or OAuth provider.

Authentication result: The authentication provider returns an authentication result, indicating whether the user's credentials were successfully authenticated or not. If successful, the result typically contains the authenticated user details, such as the username and granted authorities.

Security context: If the authentication is successful, Spring Security establishes a security context for the authenticated user. The security context holds the user's authentication details and is associated with the current thread.

Access control: With the user authenticated, Spring Security can enforce access control policies based on the user's granted authorities or other attributes. This allows the application to restrict access to certain resources or operations based on the user's role or permissions.

Spring Security provides several authentication mechanisms out-of-the-box, including form-based authentication, HTTP Basic/Digest authentication, JWT token, OAuth-based authentication. Spring also supports customization and extension, allowing you to integrate

with your own authentication providers or implement custom authentication logic to meet your specific requirements.

By integrating Spring Security's authentication capabilities into your application, you can ensure that only authenticated and authorized users have access to your protected resources, helping to safeguard your application against unauthorized access.

What is Authorization?

Authorization, also known as access control, is the process of determining what actions or resources a user or client is allowed to access within a system or application. It involves enforcing permissions and restrictions based on the user's identity, role, or other attributes. Once a user is authenticated, authorization is used to control their access to different parts of the application and its resources.

Here are the key concepts related to authorization in Spring Security:

Roles: Roles represent a set of permissions or privileges granted to a user. They define the user's high-level responsibilities or functional areas within the application. For example, an application may have roles such as "admin," "user," or "manager."

Permissions: Permissions are specific actions or operations that a user is allowed to perform. They define the granular level of access control within the application. For example, a user with the "admin" role may have permissions to create, update, and delete resources, while a user with the "user" role may only have read permissions.

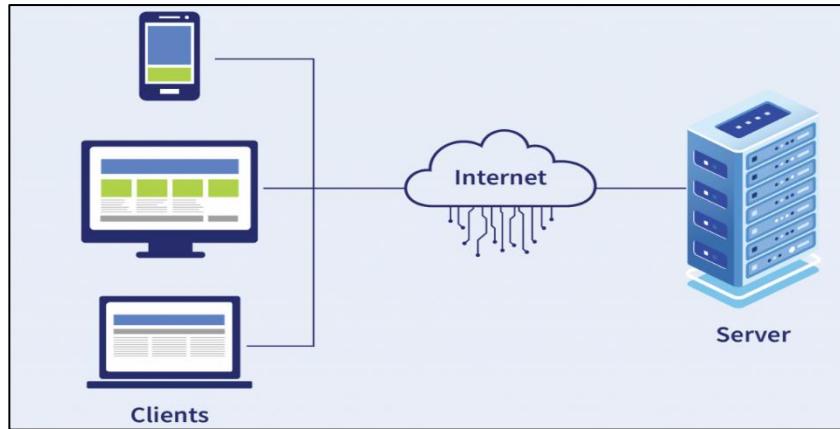
Security Interceptors: Spring Security uses security interceptors to enforce authorization rules. These interceptors are responsible for intercepting requests and checking whether the user has the required permissions to access the requested resource. They can be configured to protect URLs, methods,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

Role-Based Access Control (RBAC): RBAC is a common authorization model in which access control is based on roles. Users are assigned roles, and permissions are associated with those roles. Spring Security supports RBAC by allowing you to define roles and assign them to users.

By implementing authorization in your Spring application using Spring Security, you can ensure that users have appropriate access privileges based on their roles and permissions. This helps protect sensitive resources and data from unauthorized access and maintain the overall security and integrity of your application.

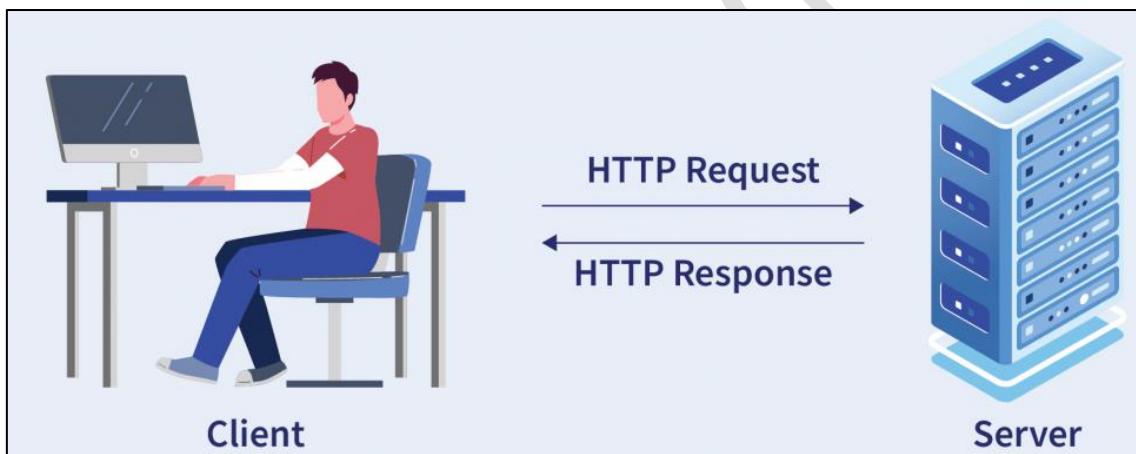
Stateless and Stateful Protocols:

In the context of the protocol, "stateless" and "stateful" refer to different approaches in handling client-server interactions and maintaining session information. Let's explore each concept:



Stateless:

In a stateless protocol, such as HTTP, the server does not maintain any information about the client's previous interactions or session state. Each request from the client to the server is considered independent and self-contained. The server treats each request as if it is the first request from the client.

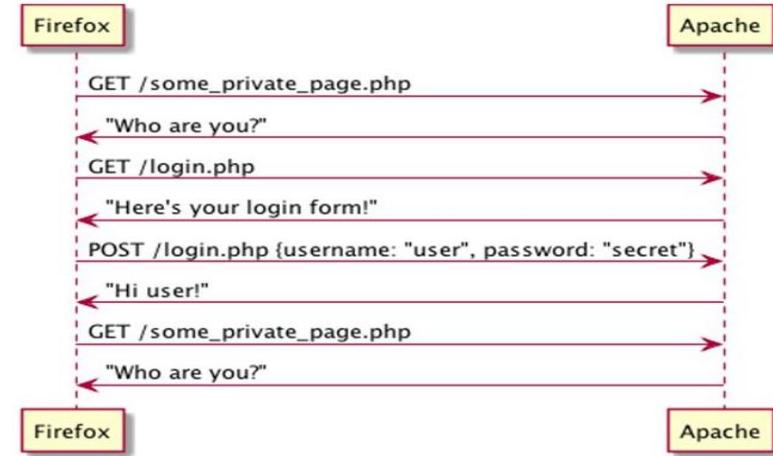


Stateless protocols have the following characteristics:

- No client session information is stored on the server.
- Each request from the client must contain all the necessary information for the server to process the request.
- The server responds to each request independently, without relying on any prior request context.

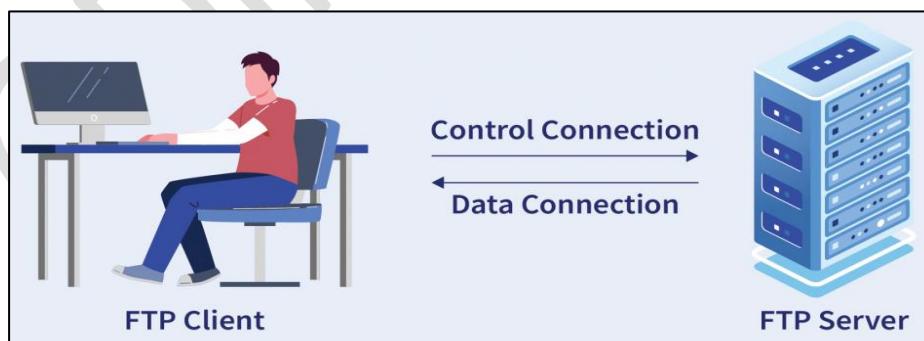
HTTP is primarily designed as a stateless protocol. When a client makes an HTTP request, the server processes the request and sends back a response. However, the server does not maintain any information about the client after the response is sent. This approach simplifies the server's implementation and scalability but presents challenges for handling user sessions and maintaining continuity between multiple requests.

Stateless Protocol (Technical)



Stateful: In contrast, a stateful protocol maintains information about the client's interactions and session state between requests. The server stores client-specific information and uses it to provide personalized responses and maintain continuity across multiple requests.

However, the major feature of stateful is that it maintains the state of all its sessions, be it an authentication session, or a client's request for information. Stateful are those that may be used repeatedly, such as online banking or email. They're carried out in the context of prior transactions in which the states are stored, and what happened in previous transactions may have an impact on the current transaction. Because of this, stateful apps use the same servers every time they perform a user request. An example of stateful is FTP (File Transfer Protocol) i.e. File transferring between servers. For FTP session, which often includes many data transfers, the client establishes a Control Connection. After this, the data transfer takes place.



Stateful protocols have the following characteristics:

- The server keeps track of client session information, typically using a session identifier
- The session information is stored on the server.
- The server uses the session information to maintain context between requests and responses.

While HTTP itself is stateless, developers often implement mechanisms to introduce statefulness. For example, web applications often use cookies or tokens to maintain session state. These cookies or tokens contain session identifiers that the server can use to retrieve or store client-specific data.

By introducing statefulness, web applications can provide a more personalized and interactive experience for users. However, it adds complexity to the server-side implementation and may require additional considerations for scalability and session management.

It's important to note that even when stateful mechanisms are introduced, each individual HTTP request-response cycle is still stateless in nature. The statefulness is achieved by maintaining session information outside the core HTTP protocol, typically through additional mechanisms like cookies, tokens, or server-side session stores.

Q&A:

What is the difference between stateful and stateless?

The major difference between stateful and stateless is whether or not they store data regarding their sessions, and how they respond to requests. Stateful services keep track of sessions or transactions and respond to the same inputs in different ways depending on their history. Clients maintain sessions for stateless services, which are focused on activities that manipulate resources rather than the state.

Is stateless better than stateful?

In most cases, stateless is a better option when compared with stateful. However, in the end, it all comes down to your requirements. If you only require information in a transient, rapid, and temporary manner, stateless is the way to go. Stateful, on the other hand, might be the way to go if your app requires more memory of what happens from one session to the next.

Is HTTP stateful or stateless?

HTTP is stateless because it doesn't keep track of any state information. In HTTP, each order or request is carried out in its own right, with no awareness of the demands that came before it.

Is REST API stateless or stateful?

REST APIs are stateless because, rather than relying on the server remembering previous requests, REST applications require each request to contain all of the information necessary for the server to understand it. Storing session state on the server violates the REST architecture's stateless requirement. As a result, the client must handle the complete session state.

Security Implementation:

Stateless Security and **Stateful Security** are two approaches to handling security in systems, particularly in the context of web applications. Let's explore the differences between these two approaches:

Stateless Security:

Stateless security refers to a security approach where the server does not maintain any session state or client-specific information between requests. It is often associated with stateless protocols, such as HTTP, where each request is independent and self-contained. Stateless security is designed to provide security measures without relying on server-side session state.

In the context of web applications and APIs, stateless security is commonly implemented using mechanisms such as JSON Web Tokens (JWT) or OAuth 2.0 authentication scheme. These mechanisms allow authentication and authorization to be performed without the need for server-side session storage.

Here are the key characteristics and advantages of stateless security:

- **No server-side session storage:** With stateless security, the server does not need to maintain any session-specific information for each client. This eliminates the need for server-side session storage, reducing the overall complexity and resource requirements on the server side.
- **Scalability:** Stateless security simplifies server-side scaling as there is no need to replicate session state across multiple instances of application deployed to multiple servers. Each server can process any request independently, which makes it easier to distribute the load and scale horizontally.
- **Decentralized authentication:** Stateless security allows for decentralized authentication, where the client sends authentication credentials (such as a JWT token) with each request. The server can then validate the token's authenticity and extract necessary information to authorize the request.
- **Improved performance:** Without the need to perform expensive operations like session lookups or database queries for session data, stateless security can lead to improved performance. Each request carries the necessary authentication and authorization information, reducing the need for additional server-side operations.

It's important to note that while stateless security simplifies server-side architecture and offers advantages in terms of scalability and performance, it also places additional responsibilities on the client-side. The client must securely store and transmit the authentication token and include it in each request.

Stateless security is widely adopted in modern web application development, especially in distributed systems and microservices architectures, where scalability, performance, and decentralized authentication are important considerations.

In stateless security:

- **Authentication:** The client provides credentials (e.g., username and password or a token) with each request to prove its identity. The server verifies the credentials and grants access based on the provided information.
- **Authorization:** The server evaluates each request independently, checking if the user has the necessary permissions to access the requested resource.

Cons of Stateless Security:

- **Increased overhead:** The client needs to send authentication information with each request, which can increase network overhead, especially when the authentication mechanism involves expensive cryptographic operations.

Stateful Security:

Stateful security involves maintaining session state on the server. Once the client is authenticated, the server stores session information and associates it with the client. The server refers to the session state to validate subsequent requests and provide appropriate authorization.

In stateful security:

Authentication: The client typically authenticates itself once using its credentials (e.g., username and password or token). After successful authentication, the server generates a session identifier or token and stores it on the server.

Session Management: The server maintains session-specific data, such as user roles, permissions, and other contextual information. The session state is referenced for subsequent requests to determine the user's authorization level.

Pros of Stateful Security:

- **Enhanced session management:** Session state allows the server to maintain user context, which can be beneficial for handling complex interactions and personalized experiences.
- **Reduced overhead:** Since the client doesn't need to send authentication information with each request, there is a reduction in network overhead.

Cons of Stateful Security:

- **Scalability challenges:** The server needs to manage session state, which can be a scalability bottleneck. Sharing session state across multiple servers or implementing session replication techniques becomes necessary.
- **Complexity:** Implementing stateful security requires additional effort to manage session state and ensure consistency across requests.

The choice between stateless security and stateful security depends on various factors, including the specific requirements of the application, performance considerations, and the desired level of session management and personalization. Stateless security is often

preferred for its simplicity and scalability advantages, while stateful security is suitable for scenarios requiring more advanced session management capabilities.

JWT Authentication & Authorization:

JWTs or JSON Web Tokens are most commonly used to identify an authenticated user. They are issued by an authentication server and are consumed by the client-server (to secure its APIs).

What is a JWT?

JSON Web Token is an open industry standard used to share information between two entities, usually a client (like your app's frontend) and a server (your app's backend). They contain JSON objects which have the information that needs to be shared. Each JWT is also signed using cryptography (hashing) to ensure that the JSON contents (also known as JWT claims) cannot be altered by the client or a malicious party.

A token is a string that contains some information that can be verified securely. It could be a random set of alphanumeric characters which point to an ID in the database, or it could be an encoded JSON that can be self-verified by the client (known as JWTs).

Structure of a JWT:

A JWT contains three parts:

- **Header:** the signing algorithm that's being used.
- **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 generate a token with JWT, the token is generated in the format of `<header>. <payload>. <signature>` inside JWT.

Example:

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWliOiJkaWxpEBnbWFpbC5jb20iLCJleHAiOiE2ODk1MjI5OTcsImlhdCI6MTY4OTUyMjY5N30.bjFnipeNqiZ5dyrXZHk0qTPciChw0Z0eNoX5fu5uAmj6SE9mLIGD4LI_3QeGfxjZqvv8KIJe2pmTseT4g8ZSIA
```

Following image showing details of Encoded Token.

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpcE
BnbWFpbC5jb20iLCJleHaiOjE2ODk1MjI5OTcsI
mlhdCI6MTY40TUyMjY5N30.bjFnipeNqiZ5dyrX
ZHk0qTPciChw0Z0eNoX5fu5uAmj6SE9mLIGD4L1
_3QeGfxjZqvv8K1Je2pmTseT4g8ZSIA
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE
{ "alg": "HS512" }
PAYLOAD: DATA
{ "sub": "dilip@gmail.com", "exp": 1689522997, "iat": 1689522697 }
VERIFY SIGNATURE
HMACSHA512(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret <input type="checkbox"/> secret base64 encoded)

JWT Token Creation and Validation:

We are using Java JWT API for creation and validation of Tokens.

- Create A Maven Project
- Add Below both dependencies, required for java JWT API.

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

- Now Write a Program for creating, claiming and validating JWT tokens : **JSONWebToken.java**

```
import java.util.Date;
import java.util.concurrent.TimeUnit;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtBuilder;
import io.jsonwebtoken.SignatureAlgorithm;
```

```

//JWT Token Generation
public class JSONWebToken {

    static String key = "ZOMATO";

    public static void main(String ar[]) {

        // Creating/Producing Tokens
        String token = Jwts.builder()
            .setSubject("dilipsingh1306@gmail.co") // User ID
            .setIssuer("ZOMATOCOMPANY")
            .setIssuedAt(new Date(System.currentTimeMillis()))
.setExpiration(new Date(System.currentTimeMillis() + TimeUnit.MINUTES.toMillis(1)))
            .signWith(SignatureAlgorithm.HS256, key.getBytes())
            .compact();

        System.out.println(token);

        // Reading/Parsing Token Details
        claimToken(token);

        //Checking Expired or not.
        boolean isExpired = isTokenExpired(token);
        System.out.println("Is It Expired? " + isExpired);
    }

    public static void claimToken(String token) {
        // Claims : Reading details from generated token by passing secret
        Claims claim = Jwts.parser().setSigningKey(key.getBytes()).parse(token).getBody();

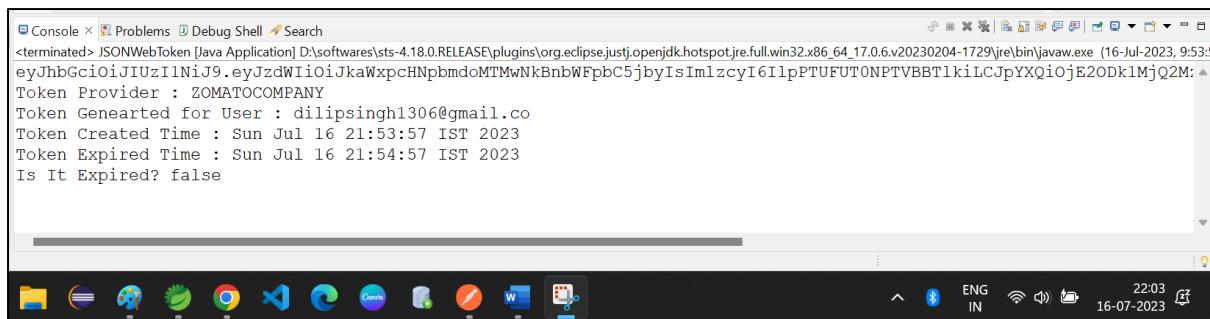
        Date createdDateTime = claim.getIssuedAt();
        Date expDateTime = claim.getExpiration();
        String issuer = claim.getIssuer();
        String subject = claim.getSubject();

        System.out.println("Token Provider : " + issuer);
        System.out.println("Token Generated for User : " + subject);
        System.out.println("Token Created Time : " + createdDateTime);
        System.out.println("Token Expired Time : " + expDateTime);
    }

    public static boolean isTokenExpired(String token) {
        Claims claim = Jwts.parser().setSigningKey(key.getBytes()).parse(token).getBody();
        Date expDateTime = claim.getExpiration();
        return expDateTime.before(new Date(System.currentTimeMillis()));
    }
}

```

```
}
```

Output:


```
Console x Problems Debug Shell Search
<terminated> JSONWebToken [Java Application] 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 (16-Jul-2023, 9:53:54)
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJkaWxpCHNpbmdoMTMwNkBnbWFpbC5jb21lczc1I6IlpPTUFUT0NPVBBTlk1LCJpYXQiOjE2ODk1MjQ2M: Token Provider : ZOMATO COMPANY
Token Generated for User : dilipsingh1306@gmail.co
Token Created Time : Sun Jul 16 21:53:57 IST 2023
Token Expired Time : Sun Jul 16 21:54:57 IST 2023
Is It Expired? false

22:03
ENG IN 16-07-2023
```

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 Spring Security Implementation. Let's start Spring Security with JWT.

GitHub Repository Link : <https://github.com/dilipsingh1306/javJWTToken>

Spring Security + (JSON WEB TOKEN):

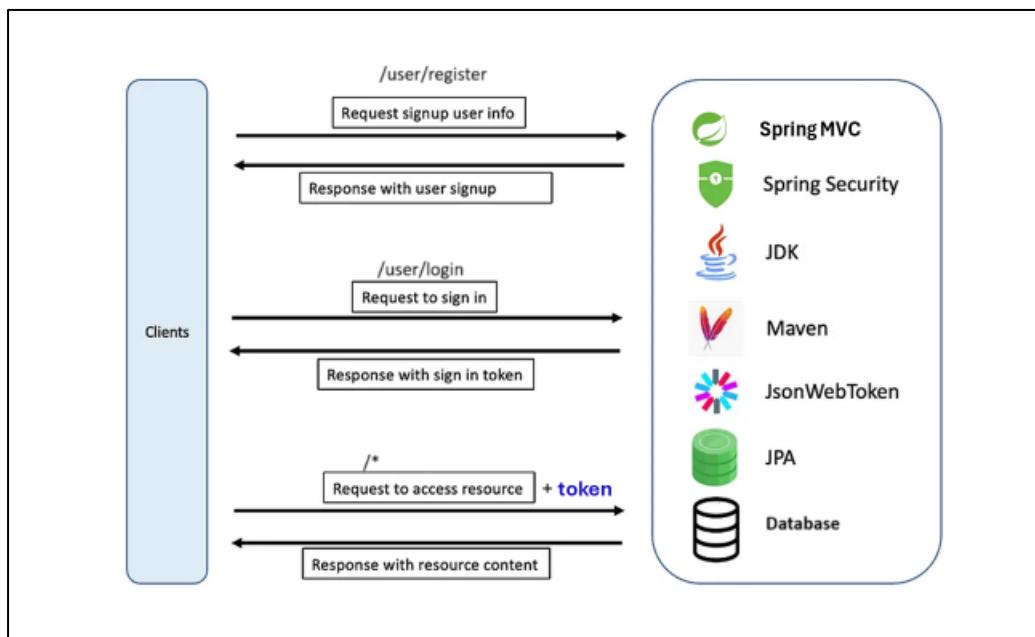
The application we are going to develop will handle user authentication and authorization with JWT's for securing an exposed REST API Services.



Application Architecture: Scenarios:

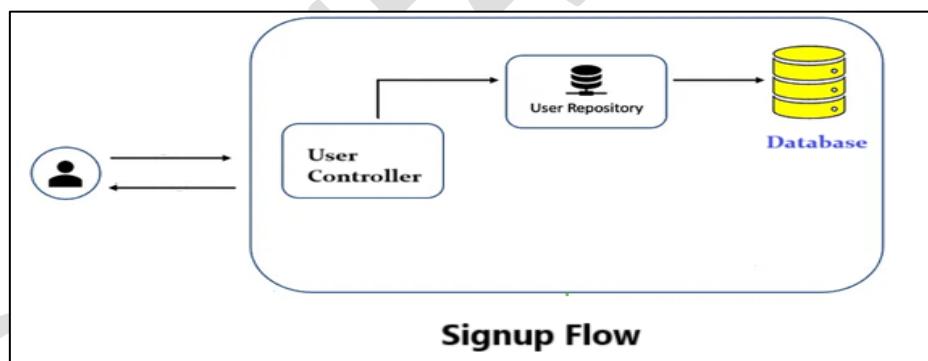
- User makes a request to the service, for create an account.
- User submits login request to the service to authenticate their account.

- An authenticated user sends a request to access resources/services.



Sign Up Process:

Step 1: Implement Logic for User Sign Up Process. The Sign-up process is very simple. Please understand following Signup Flow Diagram.



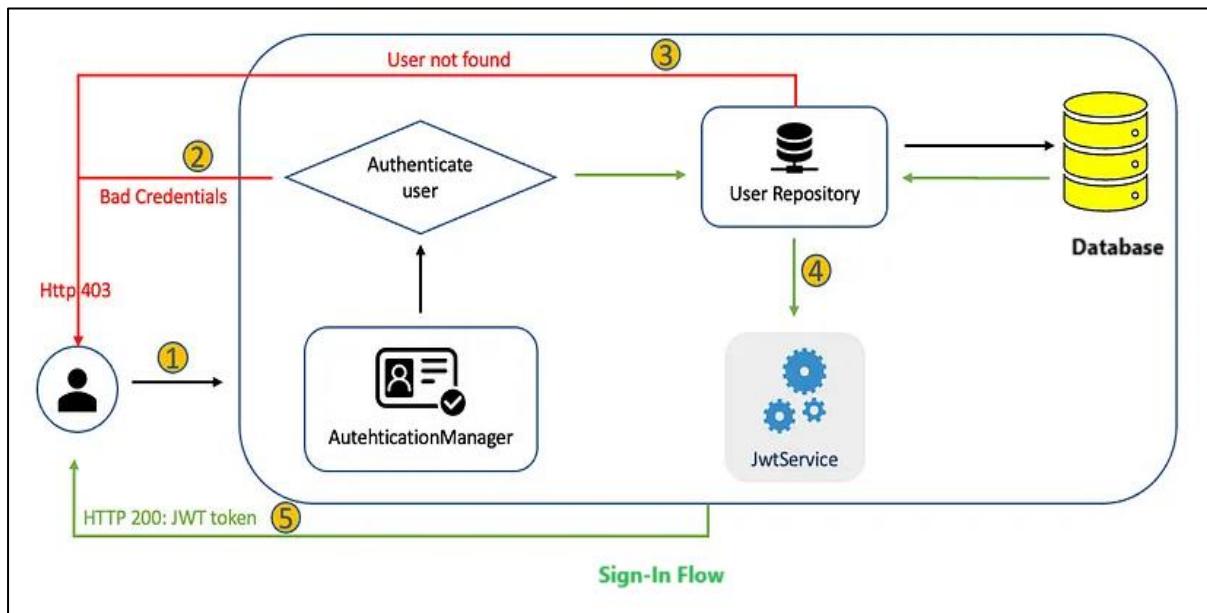
- The process starts when a user submits a request to our service. A user object is then generated from the request data, and we should encode password before storing inside Database. The password being encoded by using Spring provided Password Encoders.

It is important that we must inform Spring about the specific password encoder utilized in the application, In this case, we are using **BCryptPasswordEncoder**. This information is necessary for Spring to properly authenticate users by decoding their passwords. We will have more information about Password Encoder further.

In our application requirement is, For User Sign-up provide details of email ID, Password, Name and Mobile Number. **Email ID and Password are inputs for Sign-In operation.**

Sign-In Activity:

Internal Process and Logic Implementation:



1. The process begins when a user sends a sign-in request to the Service. An Authentication object called **UsernamePasswordAuthenticationToken** is then generated, using the provided username and password.
2. The **AuthenticationManager** is responsible for authenticating the Authentication object, handling all necessary tasks. If the **username or password is incorrect**, an exception is thrown as Bad Credentials, and a response with HTTP Status 403 is returned to the user.
3. After successful authentication, Once we have the user information, we call the JWT Service 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 the authentication object and will do all authentication jobs for us.

Resource/Services Accessibility:

When User tries to access any other resources/REST services of application, then we will apply security rules and after success authentication and authorization of user, we will allow to access/execute services. If Authentication failed, then we will send Specific Error Response codes usually 403 Forbidden.

Internally how we are going to enabling Security with JSON web token:

This process is secured by Spring Security, Let's define its flow as follows.

1. When the Client sends a request to the Service, The request is first intercepted by **JWT Token Filter**, 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, **JWT Token Filter** 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 **User Details Service** 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, user details are encapsulated in **UsernamePasswordAuthenticationToken** object and stored inside the **SecurityContextHolder**.
8. The Spring Security Authorization process is automatically invoked.
9. The request is dispatched to the controller, and a successful JSON response is returned to the user.

This process is a little bit tricky because involving some new concepts. Let's have some information about all new items.

SecurityFilterChain: In Spring Security, the **SecurityFilterChain** is responsible for managing a chain of security filters that process and enforce security rules for incoming requests in order to decide whether rules applies to that request or not. It plays a crucial role in handling authentication, authorization, and other security-related tasks within a Spring Security-enabled application. The **SecurityFilterChain** interface represents a single filter chain configuration. If we want, we can define multiple **SecurityFilterChain** instances to handle different sets of URLs or request patterns, allowing over security rules based on specific requirements.

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

such as their principal (typically a user object) and their granted authorities. You can access the **SecurityContext** using the static methods of **SecurityContextHolder**.

UserDetailsService: In Spring Security, the **UserDetailsService** interface is used to retrieve user-related data during the authentication process. It provides a mechanism for Spring Security to load user details (such as username, password, and authorities) from database or any other data source. The **UserDetailsService** interface defines a single method:

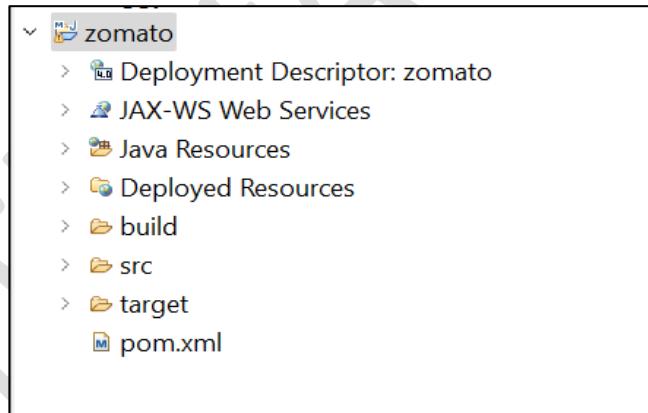
```
UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException;
```

The **loadUserByUsername()** method is responsible for retrieving the user details for a given username. It returns an implementation of the **UserDetails** interface, which represents the user's security-related data.

Security Logic Implementation: Now Create Spring Application with Security API:

- Creating Spring MVC+JPA project.
- Create Dynamic Web Project
- Convert Project to Maven project

Right Click on Project -> Configure -> Convert Maven project.



- Now Add Dependencies to **pom.xml** file to support Spring MVC + JPA

```
<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>zomato</groupId>
    <artifactId>zomato</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.29</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
        <scope>provided</scope>
    </dependency>

    <!--JSON : Jackson API jars :-->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.15.2</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.15.2</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>2.15.2</version>
    </dependency>

    <!--Spring JPA jar files -->
    <dependency>
        <groupId>com.oracle.database.jdbc</groupId>
        <artifactId>ojdbc8</artifactId>
        <version>21.9.0.0</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.9.Final</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
        <version>2.1.0.RELEASE</version>
    </dependency>
</dependencies>
```

```

<build>
  <plugins>
    <plugin>
      <artifactId>maven-war-plugin</artifactId>
      <version>3.2.3</version>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <release>17</release>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

➤ Now Add MVC Module Configuration to work with Annotation Based Configuration.

MVCConfiguration.java

```

package com.zomato;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Configuration
@ComponentScan("com.*")
@EnableWebMvc
public class MVCConfiguration extends WebMvcConfigurationSupport {

}

```

SpringWebInitialization.java

```

package com.zomato;

import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class SpringWebInitialization extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }
}

```

```

@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[] {MVCCConfiguration.class};
}

@Override
protected String[] getServletMappings() {
    String[] allowedURLMapping = {"/*"};
    return allowedURLMapping;
}
}

```

➤ Now Add JPA Configuration Class.

```

package com.zomato;

import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EnableTransactionManagement
@EnableJpaRepositories("com.*")
public class SpringJpaConfiguration {

    // DB Details
    @Bean
    public DataSource getDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");
        dataSource.setUsername("c##dilip");
        dataSource.setPassword("dilip");
        dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        return dataSource;
    }

    @Bean("entityManagerFactory")
    LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {
        LocalContainerEntityManagerFactoryBean factory =
            new LocalContainerEntityManagerFactoryBean();

```

```

// 1. Setting Datasource Object // DB details
factory.setDataSource(getDataSource());

// 2. Provide package information of entity classes
factory.setPackagesToScan("com.*");

// 3. Providing Hibernate Properties to EM
factory.setJpaProperties(hibernateProperties());

// 4. Passing Predefined Hiberante Adaptor Object EM
HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
factory.setJpaVendorAdapter(adapter);
return factory;
}

// PSrring JPA: configuring data based on your project req.
@Bean("transactionManager")
public PlatformTransactionManager createTransactionManager() {
    JpaTransactionManager transactionManager =
        new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(createEntityManagerFactory().getObject());
    return transactionManager;
}

Properties hibernateProperties() {
    Properties hibernateProperties = new Properties();
    hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");
    hibernateProperties.setProperty("hibernate.dialect",
        "org.hibernate.dialect.Oracle10gDialect");
    hibernateProperties.setProperty("hibernate.show_sql", "true");
    return hibernateProperties;
}
}

```

With These Steps and Configuration classes, Our Project supports MVC and JPA Concepts.

Let's Define REST Services.

- **Add User Registration REST Service.**

Create Request Body Class : UserRegister.java

```

package com.zomato.dto;

public class UserRegister {
    private String emailId;
}

```

```

private String password;
private String fullName;
private String conatctNumber;

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 String getFullName() {
    return fullName;
}
public void setFullName(String fullName) {
    this.fullName = fullName;
}
public String getConatctNumber() {
    return conatctNumber;
}
public void setConatctNumber(String conatctNumber) {
    this.conatctNumber = conatctNumber;
}
}

```

➤ Now Create REST Service for User Registration In Controller : ZomatoController.java

```

package com.zomato.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.zomato.dto.UserLogin;
import com.zomato.dto.UserRegister;
import com.zomato.service.ZomatoUserService;

@RestController
public class ZomatoController {

    @Autowired
    ZomatoUserService service;
}

```

```

    @PostMapping("/create/user")
    public String registerUser(@RequestBody UserRegister request) {
        return service.registerUser(request);
    }
}

```

➤ **Service Layer Class: ZomatoUserService.java**

```

package com.zomato.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.zomato.dto.UserLogin;
import com.zomato.dto.UserRegister;
import com.zomato.entity.UserEntity;
import com.zomato.repository.ZomatoUserRepository;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Service
public class ZomatoUserService {

    @Autowired
    ZomatoUserRepository repository;

    @Autowired
    BCryptPasswordEncoder passwordEncoder;

    public String registerUser(UserRegister request) {
        //Convert to Entity Object
        UserEntity entity = new UserEntity();
        entity.setEmailId(request.getEmailId());
        entity.setPassword(passwordEncoder.encode(request.getPassword()));
        entity.setFullName(request.getFullName());
        entity.setContactNumber(request.getContactNumber());
        repository.save(entity);
        return "User Created Successfully. Please Login Now.";
    }
}

```

➤ **Create Entity Class: UserEntity.java**

```

package com.zomato.entity;

import javax.persistence.Column;
import javax.persistence.Entity;

```

```
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "zomato_users")
public class UserEntity {

    @Id
    @Column
    private String emailId;

    @Column
    private String password;

    @Column
    private String fullName;

    @Column
    private String conatctNumber;

    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 String getFullName() {
        return fullName;
    }

    public void setFullName(String fullName) {
        this.fullName = fullName;
    }

    public String getConatctNumber() {
        return conatctNumber;
    }

    public void setConatctNumber(String conatctNumber) {
        this.conatctNumber = conatctNumber;
    }
}
```

- **Repository Layer:** ZomatoUserRepository.java

```
package com.zomato.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.zomato.entity.UserEntity;

@Repository
public interface ZomatoUserRepository extends JpaRepository<UserEntity, String>{

}
```

Now we are ready with User registration Service.

As per Requirement, User Login REST Service should be integrated with Security Layer and as well as JWT token when User is Valid.

Now Add JWT functionality our application, for generating and validating token based on User Name.

- We should add JWT library dependencies inside Maven **pom.xml** file, because by default Spring not providing support of JWT.

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.2</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.2</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.2</version>
</dependency>
```

- **Now Create JWT Token Creator and Validation Utility class.**

```
package com.zomato.service;
```

```

import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import org.springframework.stereotype.Component;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

@Component
public class JWTTokenUtil {

    //5 Mins
    public static final long JWT_TOKEN_VALIDITY_MILLIS = 5 * 60000;
    private String secret =
"ASFACASDFACASDFASFASFDASFASDAADSCSDFADCSFDAFASFASDFACASDFASFASFDAFASFASDAADSCSDFADCSFDAFAS";

    // 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 token for user
    public String generateToken(String userName) {
        Map<String, Object> claims = new HashMap<>();
        return doGenerateToken(claims, userName);
    }

    // while creating the token -
    // Sign the JWT using the HS512 algorithm and secret key.
    private String doGenerateToken(Map<String, Object> claims,
                                   String subject) {

        return Jwts.builder()
            .setClaims(claims)
            .setSubject(subject)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis()
                + JWT_TOKEN_VALIDITY_MILLIS))
            .signWith(SignatureAlgorithm.HS512, secret).compact();
    }
}

```

```

    }

    // validate token
    public Boolean validateToken(String token, String userName) {
        final String username = getUsernameFromToken(token);
        return (username.equals(userName)
                && !isTokenExpired(token));
    }
}

```

- Now Create Rest Service for Login User which should be integrated via Security Layer. Once user is valid, we should send a response back to Client with Token value.

In Spring, We should add below Dependencies in **pom.xml** file for Spring Security Support.

```

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.7.10</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>5.7.10</version>
</dependency>

```

Security Layer Setup in Spring MVC:

AbstractSecurityWebApplicationInitializer with Spring MVC:

If we were using Spring MVC in our application we probably already had a **WebApplicationInitializer** that is loading our Spring MVC Configuration. We should register Spring Security with the existing **ApplicationContext**. For example, if we are using Spring MVC our **SecurityWebApplicationInitializer** would look something like the following:

- So Add Below class in our Configuration classes.

```

package com.zomato;

import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer {
}

```

This would simply only register the **springSecurityFilterChain** Filter for every URL in your application. After that we would ensure that **WebSecurityConfig** was loaded in our existing ApplicationInitializer.

Now We have to Define User Validation Logic with Security layer Integration. To achieve these functionalities , Spring Security provided few pre-defined interfaces and Classes.

Step 1: Customize our Entity Class by Implementing **UserDetails** Interface of Spring Security API. So that we can directly Store Repository/Database Data of User Credentials and roles inside **UserDetails** entity. Now same Entity will be utilized by Spring Authentication and Authorization Modules internally.

Here This interface contains few pre-defined abstract methods, those should be overridden with our logic.

```
package com.zomato.entity;

import java.util.Collection;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

@Entity
@Table(name = "zomato_users")
public class UserEntity implements UserDetails {

    @Id
    @Column
    private String emailId;

    @Column
    private String password;

    @Column
    private String fullName;

    @Column
    private String contactNumber;

    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 String getFullName() {
        return fullName;
    }
    public void setFullName(String fullName) {
        this.fullName = fullName;
    }
    public String getConatctNumber() {
        return conatctNumber;
    }
    public void setConatctNumber(String conatctNumber) {
        this.conatctNumber = conatctNumber;
    }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return null;
    }
    @Override
    public String getUsername() {
        return this.emailld;
    }
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

Step 2: Now we have to Implement User Service layer with pre-defined Interface provided by Spring Security Layer.

Interface UserDetailsService:

This Core interface which loads user-specific data. It is used throughout the framework as a user DAO and is the strategy used by the **DaoAuthenticationProvider**. An **AuthenticationProvider** implementation that retrieves user details from a **UserDetailsService**.

So we have to implement **UserDetailsService** interface from our User Service layer class.

- Implantation of Interface from **ZomatoUserService.java**

```
package com.zomato.service;

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.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;
import com.zomato.dto.UserLogin;
import com.zomato.dto.UserRegister;
import com.zomato.entity.UserEntity;
import com.zomato.repository.ZomatoUserRepository;

@Service
public class ZomatoUserService implements UserDetailsService{

    @Autowired
    ZomatoUserRepository repository;

    //for encrypt passwords
    @Autowired
    BCryptPasswordEncoder passwordEncoder;

    public String registerUser(UserRegister request) {
        //Convert to Entity Object
        UserEntity entity = new UserEntity();
        entity.setEmailId(request.getEmailId());
        entity.setPassword(passwordEncoder.encode(request.getPassword()));
        entity.setFullName(request.getFullName());
        entity.setConatctNumber(request.getConatctNumber());
        repository.save(entity);
        return "User Created Successfully. Please Login Now.";
    }
}
```

```

@Override
public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {

    UserEntity user = repository.findByEmailId(username).orElseThrow(() 
        -> new RuntimeException("User Not Found"));

    return user;
}
}

```

- Now we have to Add **findByEmailId()** method inside Repository:

ZomatoUserRepository.java

```

package com.zomato.repository;

import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.zomato.entity.UserEntity;

@Repository
public interface ZomatoUserRepository extends JpaRepository<UserEntity, String>{

    Optional<UserEntity> findByEmailId(String username);
}

```

- Now, we have to create a Filter which is responsible for reading JWT token and validation of token for every incoming Request before it reaching to our Controller Services or endpoint methods. For this we have extend a pre-defined filter OncePerRequestFilter of Spring Security Module.
- As part of this class, we have to override **doFilterInternal()** method.

```

package com.zomato.security;

import java.io.IOException;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
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.stereotype.Component;

```

```
import org.springframework.web.filter.OncePerRequestFilter;

import com.zomato.service.JWTTokenUtil;
import com.zomato.service.ZomatoUserService;

@Component
public class JWTTokenAuthenticationFilter extends OncePerRequestFilter {

    @Autowired
    ZomatoUserService userService;

    @Autowired
    JWTTokenUtil jwtTokenUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        // Authorization
        String token = request.getHeader("Authorization");
        System.out.println(" Header : Authorization : " + token);

        String username = null;
        if (token != null) {
            username = this.jwtTokenUtil.getUsernameFromToken(token.trim());
        } else {
            System.out.println("Invalid Header Value !! ");
        }

        if (username != null
                && SecurityContextHolder.getContext().getAuthentication() == null) {

            // fetch user detail from username
            UserDetails userDetails =
                    this.userService.loadUserByUsername(username);
            Boolean validateToken =
                    this.jwtTokenUtil.validateToken(token, userDetails.getUsername());
            if (validateToken) {

                // set the authentication
                UsernamePasswordAuthenticationToken authentication =
                        new UsernamePasswordAuthenticationToken(
                                userDetails, null, userDetails.getAuthorities());

                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        }
    }
}
```

```
    } else {
        System.out.println("Invalid Token... !!");
    }
}
filterChain.doFilter(request, response);
}
}
```

What is UsernamePasswordAuthenticationToken ?

`org.springframework.security.authentication.UsernamePasswordAuthenticationToken`

UsernamePasswordAuthenticationToken is a class in Spring Security, which is a popular framework for implementing authentication and authorization in Java applications. This class is typically used for representing an authentication request where a user provides their username and password for authentication.

Here's a brief explanation of how **UsernamePasswordAuthenticationToken** works:

- **User Provides Credentials:** When a user submits their username and password for authentication, these credentials are typically collected through a login form in a web application.
- **Creation of UsernamePasswordAuthenticationToken:** The application creates a UsernamePasswordAuthenticationToken object to encapsulate the user's credentials. This token is used to represent the authentication request.
- **Authentication Process:** The UsernamePasswordAuthenticationToken is then passed to the Spring Security framework, which initiates the authentication process.
- **Authentication Provider:** Spring Security uses an authentication provider, typically a DaoAuthenticationProvider, to verify the user's credentials. This provider checks the provided username and password against the stored user credentials (usually in a database) to determine if they are valid.
- **Authentication Success or Failure:** Depending on whether the credentials are valid, the authentication process results in either success or failure. If successful, the user is considered authenticated and gains access to the protected resources. If the authentication fails, an exception is thrown or an error response is generated.
- **Security Context:** If authentication is successful, Spring Security stores the authenticated user's details in the security context. This allows the application to access information about the authenticated user during their session.
- **Authorization:** After successful authentication, Spring Security can also handle authorization, determining what actions or resources the authenticated user is allowed to access.

SecurityContextHolder:

SecurityContextHolder is a central component in Spring Security that manages the security context of an application. It allows you to access and manipulate information about the currently authenticated user and their security details throughout the lifecycle of a user's interaction with your application. This class provides a way to store and retrieve the Authentication object, which represents the currently authenticated principal (user) and their associated authorities.

- Now finally, we have to create Spring Security Context Configuration, where we should define like what kind of incoming requests should be authenticated and authorized. To make sure this Spring provided an Interface **SecurityFilterChain**.

SecurityFilterChain:

SecurityFilterChain is a core concept in Spring Security that represents a chain of filters responsible for processing HTTP requests and implementing various security features, including authentication and authorization. Each filter in the chain performs a specific security-related task, such as verifying credentials, checking access permissions, or handling session management. SecurityFilterChain is associated with a set of request patterns, such as URL patterns or antMatchers, that specify which requests should be processed by the filters within that chain. This allows you to apply different security configurations to different parts of your application.

Create A Security Configuration class: SecurityConfig.java

```
package com.zomato.security;

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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import com.zomato.service.ZomatoUserService;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
```

```

    @Autowired
    JWTTokenAuthenticationFilter jwtTokenAuthenticationFilter;

    @Autowired
    ZomatoUserService userService;

    @Bean
    AuthenticationManager authenticationManager(AuthenticationConfiguration
                                                builder) throws Exception {
        return builder.getAuthenticationManager();
    }

    @Bean
    BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/create/user", "/login/user")
            .permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilterBefore(this.jwtTokenAuthenticationFilter,
                            UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}

```

➤ Now Define Login REST Service in Controller with Spring Security layer with Authentication, as followed.

➤ In User Login, we will have User Name and Password.

- User Name : emailId
- Password : password

After User Validation, as a Response we will send emailId and token Value.

Login Response Class: UserLoginResponse.java

```

package com.zomato.dto;

public class UserLoginResponse {

    private String emailId;
    private String token;
}

```

```

public UserLoginResponse(String emailId, String token) {
    super();
    this.emailId = emailId;
    this.token = token;
}
public UserLoginResponse() {
    super();
}
public String getEmailId() {
    return emailId;
}
public void setEmailId(String emailId) {
    this.emailId = emailId;
}
public String getToken() {
    return token;
}
public void setToken(String token) {
    this.token = token;
}
}

```

- **Add Login Service in Controller:** ZomatoController.java

```

package com.zomato.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
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.RestController;
import com.zomato.dto.UserLogin;
import com.zomato.dto.UserLoginResponse;
import com.zomato.dto.UserRegister;
import com.zomato.service.JWTTokenUtil;
import com.zomato.service.ZomatoUserService;

```

```
@RestController
public class ZomatoController {

    @Autowired
    ZomatoUserService service;

    @Autowired
    JWTTOKENUtil jwtTokenUtil;

    @Autowired
    AuthenticationManager authenticationManager;

    @PostMapping("/create/user")
    public String registerUser(@RequestBody UserRegister request) {
        return service.registerUser(request);
    }

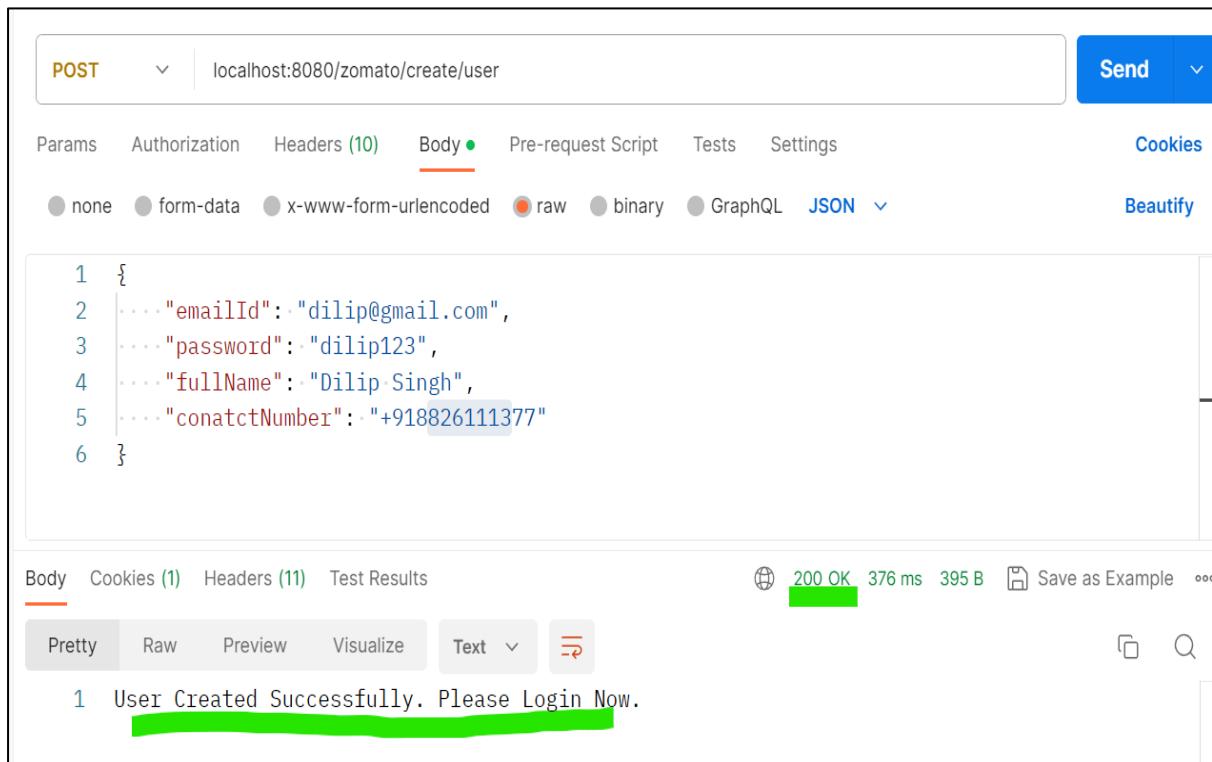
    @PostMapping("/login/user")
    public UserLoginResponse loginUser(@RequestBody UserLogin request) {
        this.doAuthenticate(request.getEmailId(), request.getPassword());
        String token = this.jwtTokenUtil.generateToken(request.getEmailId());
        return new UserLoginResponse(request.getEmailId(), token);
    }

    private void doAuthenticate(String emailId, String password) {
        UsernamePasswordAuthenticationToken authentication =
            new UsernamePasswordAuthenticationToken(emailId, password);
        try {
            authenticationManager.authenticate(authentication);
        } catch (BadCredentialsException e) {
            throw new RuntimeException("Invalid UserName and Password");
        }
    }
}
```

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

Test User Registration :

Register User: Security not applicable for this endpoint i.e. to execute below endpoint, no need to provide JWT.



The screenshot shows a Postman request to `localhost:8080/zomato/create/user` via a POST method. The request body is a JSON object containing user details:

```

1 {
2   "emailId": "dilip@gmail.com",
3   "password": "dilip123",
4   "fullName": "Dilip Singh",
5   "conatctNumber": "+918826111377"
6 }

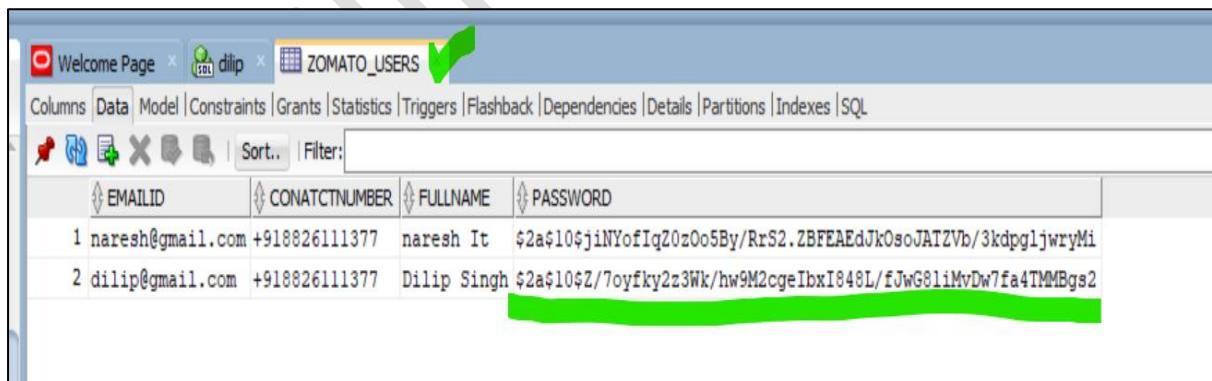
```

The response status is 200 OK, with a response time of 376 ms and a size of 395 B. The response body is:

- User Created Successfully. Please Login Now.

Now Go and Verify in Database Table, How User Details are inserted:

Because we are used Password Encryptor as per security Layer. Passwords are encrypted.

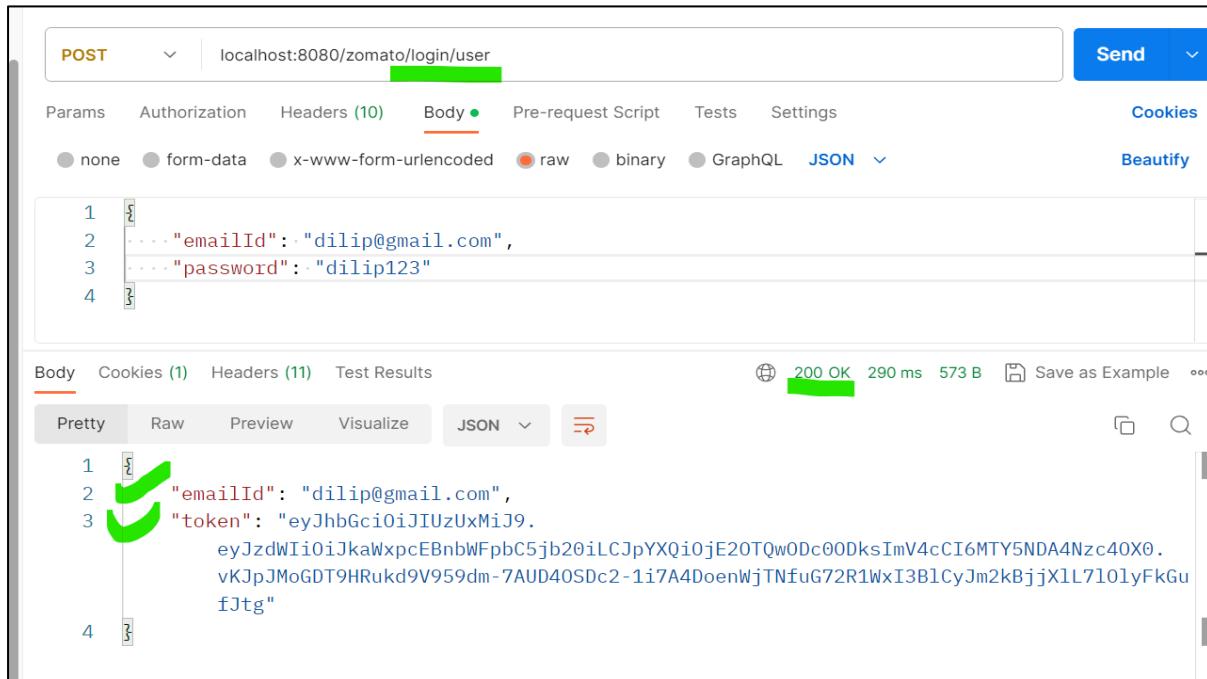


The screenshot shows the `ZOMATO_USERS` table in Oracle SQL Developer. The table has four columns: `EMAILID`, `CONATCTNUMBER`, `FULLNAME`, and `PASSWORD`. There are two rows of data:

	EMAILID	CONATCTNUMBER	FULLNAME	PASSWORD
1	naresh@gmail.com	+918826111377	naresh It	\$2a\$10\$jiNYofIqZ0z0o5By/RrS2.2BFEAEEdJk0soJATZVb/3kdpgljwryMi
2	dilip@gmail.com	+918826111377	Dilip Singh	\$2a\$10\$Z/7oyfky2z3Wk/hw9M2cgeIbxI848L/fJwG8liMvDw7fa4TMMBg2

Login User:

Security not applicable for this. i.e. to execute we no need to provide JWT. After User Validation, JWT Token comes as part of response.



The screenshot shows a Postman request to `localhost:8080/zomato/login/user`. The request method is POST. The request body is a JSON object with two fields: `"emailId": "dilip@gmail.com"` and `"password": "dilip123"`. The response status is 200 OK, and the response body is a JSON object with `"emailId": "dilip@gmail.com"` and a long string for `"token"`.

Now Access Other REST services of Application.

Adding 2 Endpoints in Controller: `/get/profile & /get/orders`

```
package com.zomato.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
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.RestController;
import com.zomato.dto.UserLogin;
import com.zomato.dto.UserLoginResponse;
import com.zomato.dto.UserRegister;
import com.zomato.service.JWTTokenUtil;
import com.zomato.service.ZomatoUserService;
```

```
@RestController
public class ZomatoController {

    @Autowired
    ZomatoUserService service;

    @Autowired
    JWTTOKENUtil jwtTokenUtil;

    @Autowired
    AuthenticationManager authenticationManager;

    @PostMapping("/create/user")
    public String registerUser(@RequestBody UserRegister request) {
        return service.registerUser(request);
    }

    @PostMapping("/login/user")
    public UserLoginResponse loginUser(@RequestBody UserLogin request) {
        this.doAuthenticate(request.getEmailId(), request.getPassword());
        String token = this.jwtTokenUtil.generateToken(request.getEmailId());
        return new UserLoginResponse(request.getEmailId(), token);
    }

    private void doAuthenticate(String emailId, String password) {
        UsernamePasswordAuthenticationToken authentication =
            new UsernamePasswordAuthenticationToken(emailId, password);
        try {
            authenticationManager.authenticate(authentication);
        } catch (BadCredentialsException e) {
            throw new RuntimeException("Invalid UserName and Password");
        }
    }

    @GetMapping("/get/profile")
    public String getProfile() {

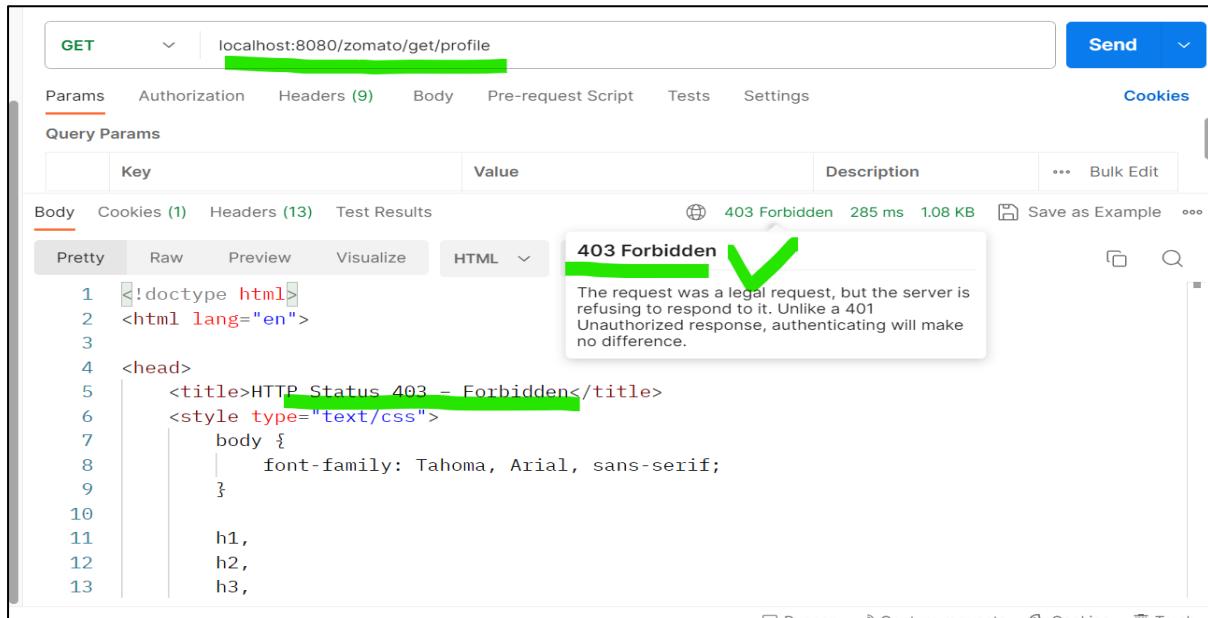
        return "Welcome Dilip , Please find Your Profile Details";
    }

    @GetMapping("/get/orders")
    public String getOrders(@RequestHeader String token) {

        return "Welcome Dilip , Please find Your Order Details";
    }
}
```

Now We have to access /get/profile and /get/orders. Let's Do it.

Testing: /get/profile



The screenshot shows a POST request to `localhost:8080/zomato/get/profile`. The 'Headers' tab is selected, showing the following headers:

- User-Agent: PostmanRuntime/7.32.3
- Accept: */*
- Accept-Encoding: gzip, deflate, br
- Connection: keep-alive
- Authorization: (empty)

The 'Body' tab is selected. The response status is **403 Forbidden**. The response body is:

```

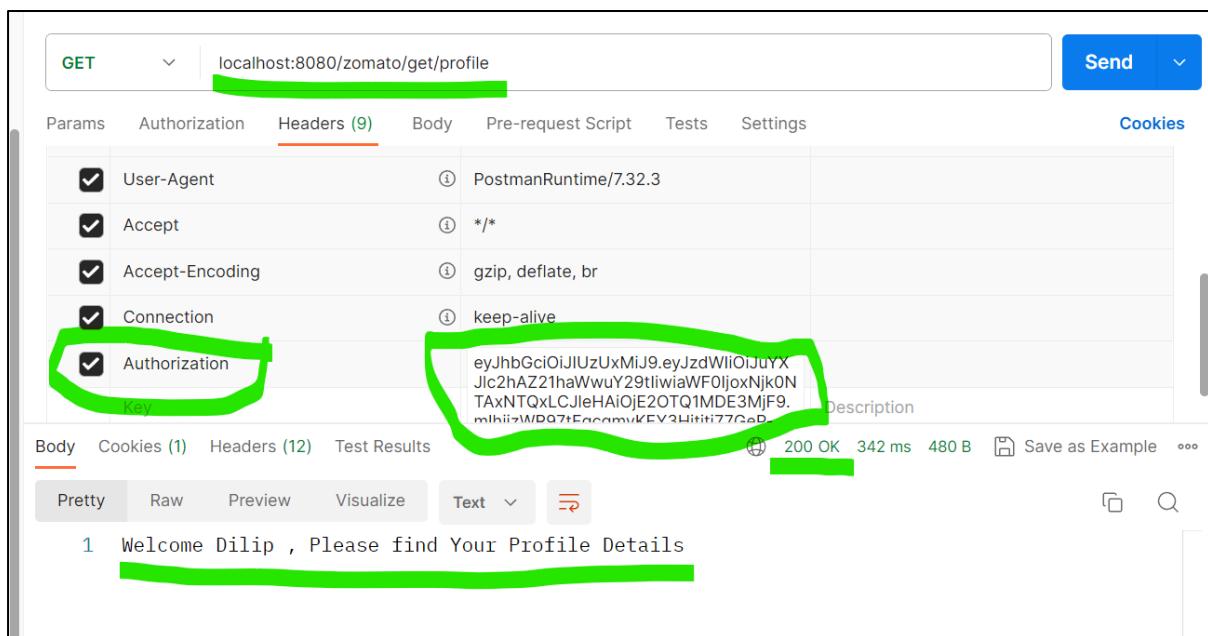
1 <!doctype html>
2 <html lang="en">
3
4 <head>
5   <title>HTTP Status 403 – Forbidden</title>
6   <style type="text/css">
7     body {
8       font-family: Tahoma, Arial, sans-serif;
9     }
10
11   h1,
12   h2,
13   h3,

```

The response description states: "The request was a legal request, but the server is refusing to respond to it. Unlike a 401 Unauthorized response, authenticating will make no difference."

We got forbidden Access, means Unauthorized. Because our application security is allowing only **/create/user** and **/login/user** without **JWT token**. Other than those two, any URI should be authenticated with JWT token. So Let's Trigger same request again with JWT token.

Add token in Headers : **Authorization <token>**



The screenshot shows a POST request to `localhost:8080/zomato/get/profile`. The 'Headers' tab is selected, showing the following headers:

- User-Agent: PostmanRuntime/7.32.3
- Accept: */*
- Accept-Encoding: gzip, deflate, br
- Connection: keep-alive
- Authorization: eyJhbGciOiJIUzI1n2Mj9.eyJzdWIiOiJuYXJlc2hAZ21haWwUy29tliwiWF0ijoNjk0NTAxNTQxLCJleHAiOjE2OTQ1MDE3MjF9.mhii7MPQ7tEcammvKEV3Hiitit77GeP

The 'Body' tab is selected. The response status is **200 OK**. The response body is:

```

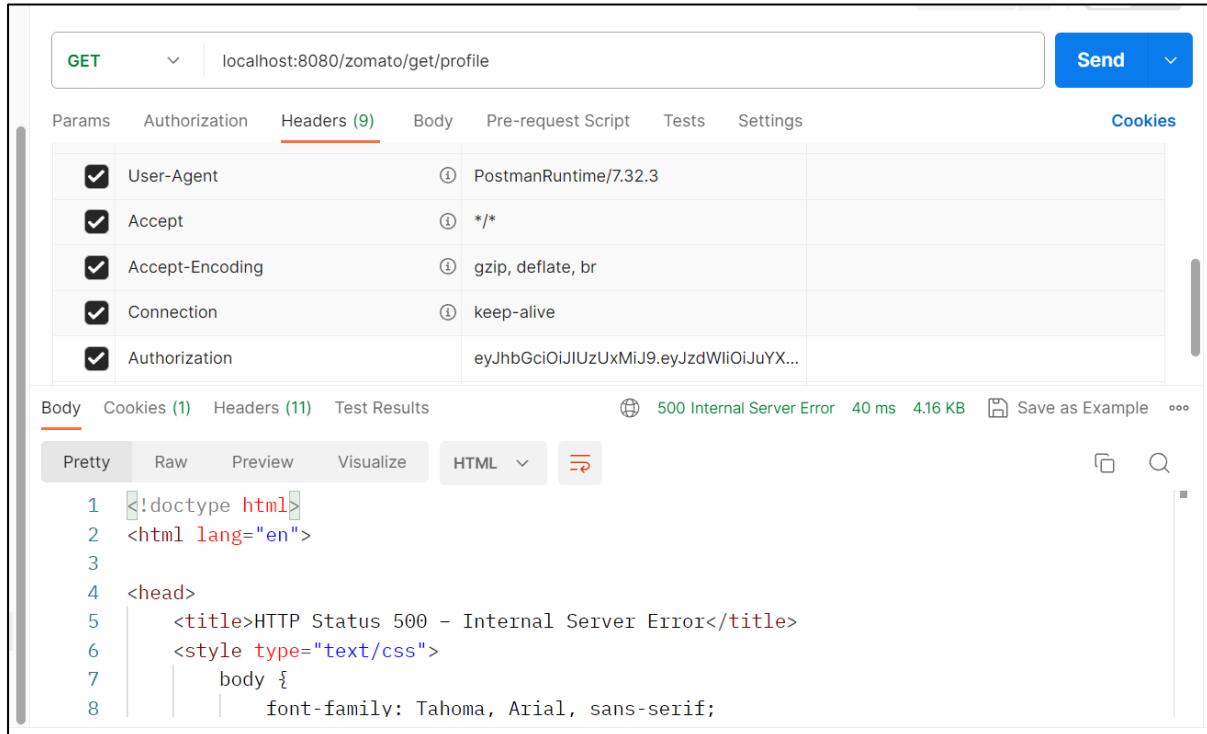
1 Welcome Dilip , Please find Your Profile Details

```

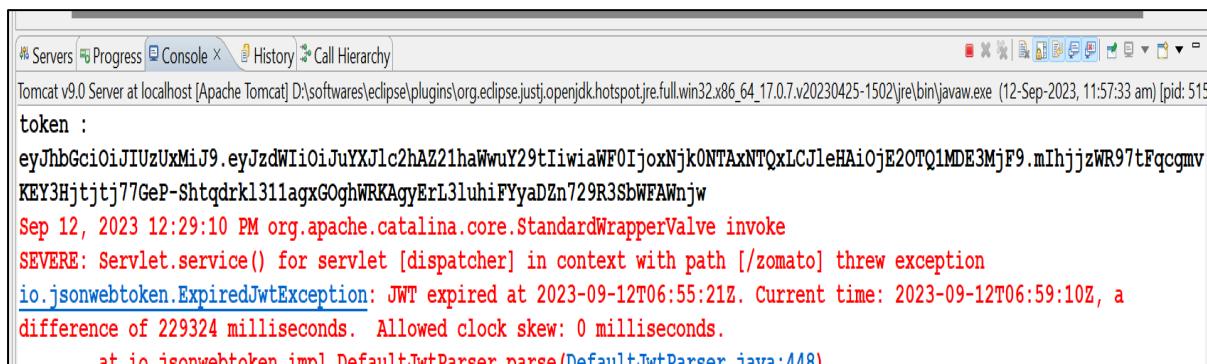
Now Token Validated and then Security Layer allowed to access out endpoint method and got expected Response.

In case If we pass an expired token, then our application will not allow to execute Endpoint logic, send response as Invalid token.

Send Invalid/expired Token: We got Error Response, See Server Side Console Logs.



Server Logs: We got Exception as `ExpiredJwtException` while validating token by JWT framework. So we should handle these exceptions as well to process meaningful messages to client.



GitHub Link for Project: <https://github.com/DilipItAcademy/spring-mvc-security-jwt-jpa>

You can refer GitHub repository, for all code base and steps.