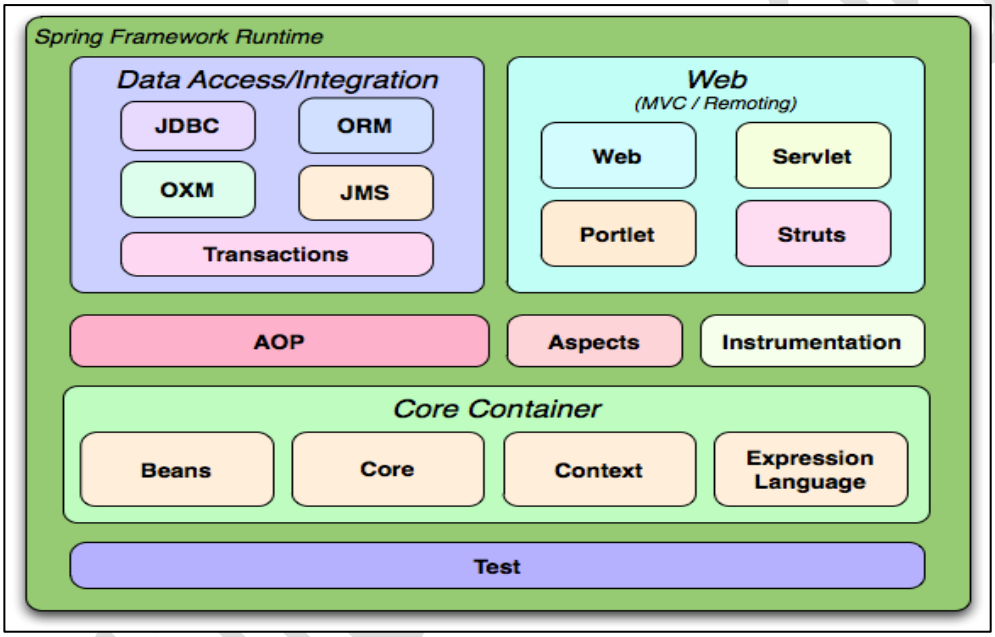
Spring Web/MVC Module

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.

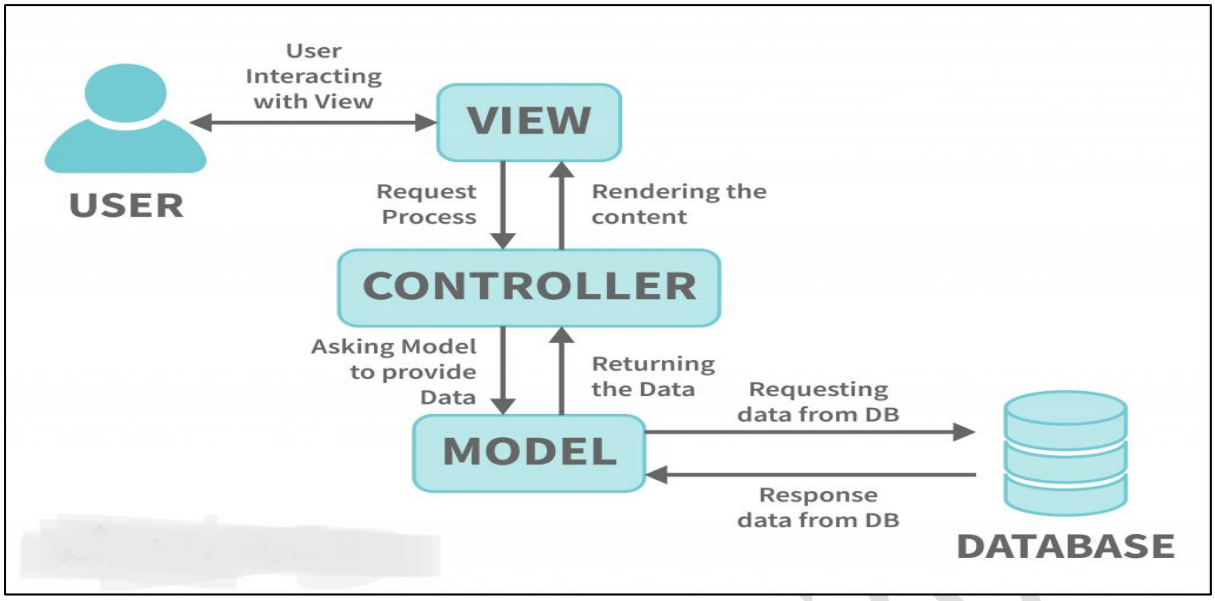


What is MVC?

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

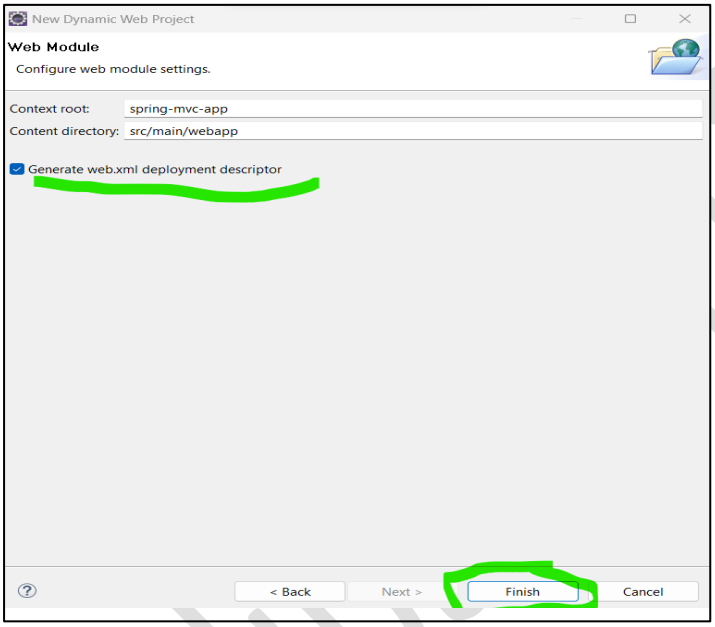
Let's see some of the advantages of Spring MVC Framework:

- **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 SpringWebInitilization
                        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**

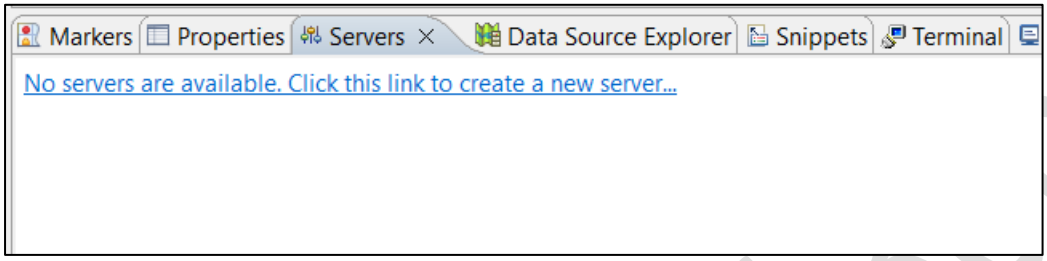
```
package com.naresh.controller;

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.

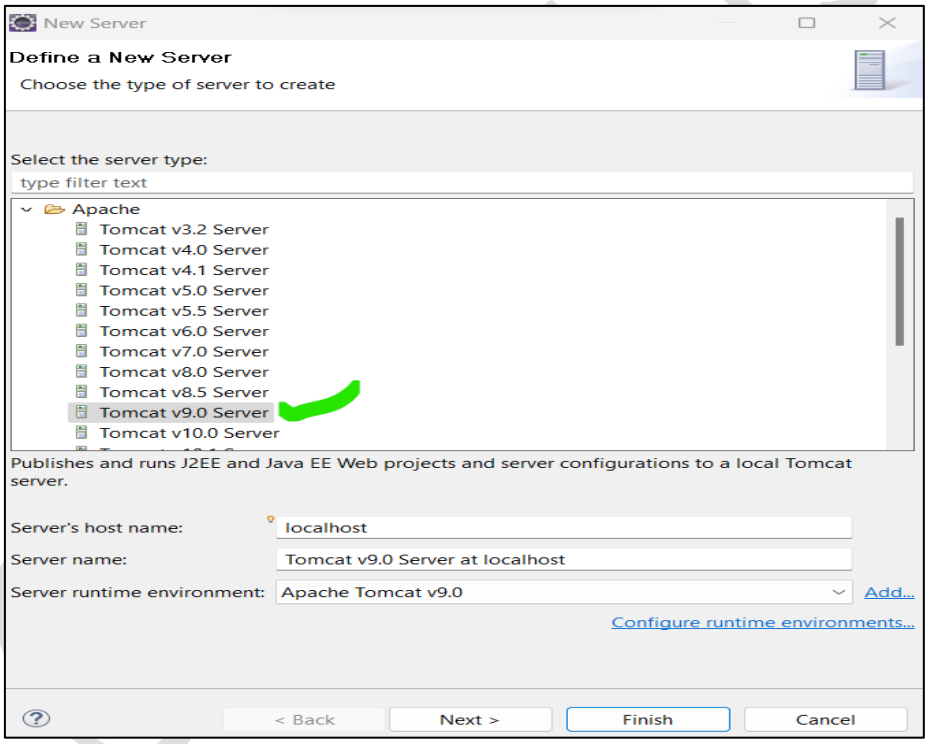
Add Tomcat Server to Eclipse: Download Tomcat and Extract it to some location.

https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.78/bin/apache-tomcat-9.0.78-windows-x64.zip

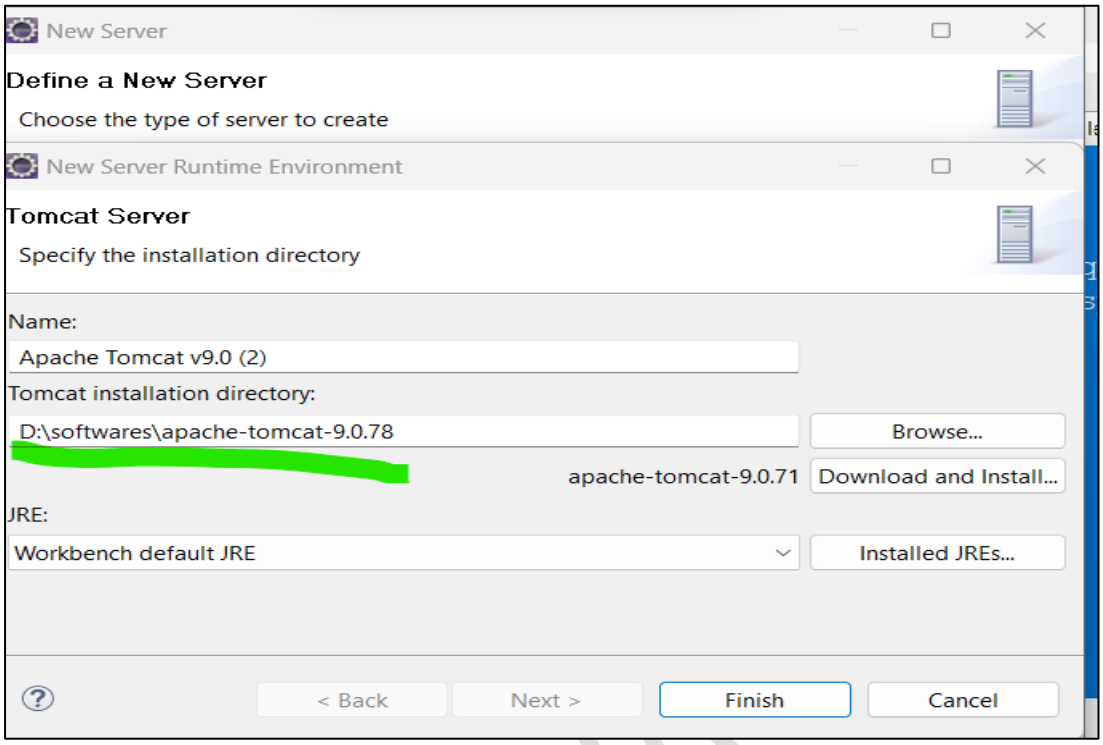
Now From Eclipse Servers Console:



Click on Create New Server



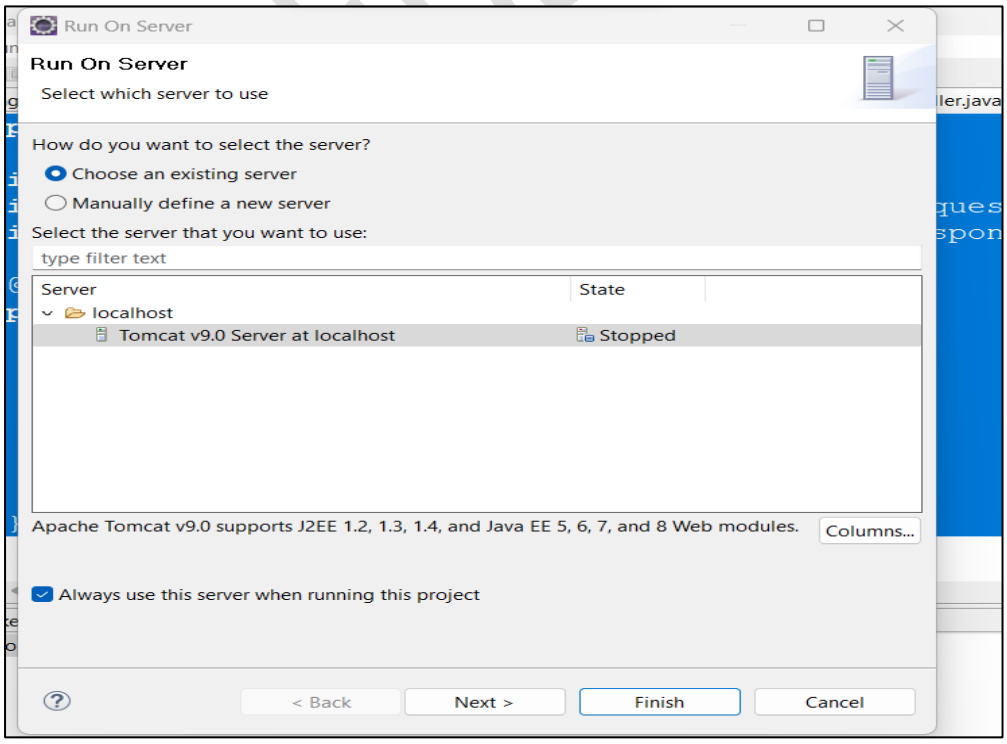
➢ **Select Tomcat folder Location**



➢ **Click on Finish, Server Added to eclipse.**

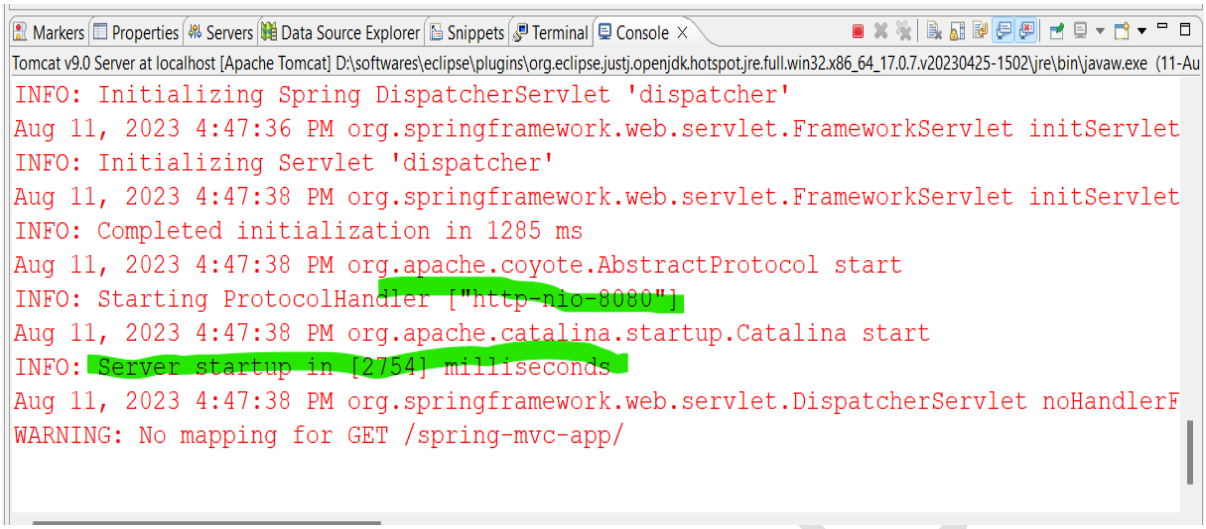
Project Deployment Process :

- **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.

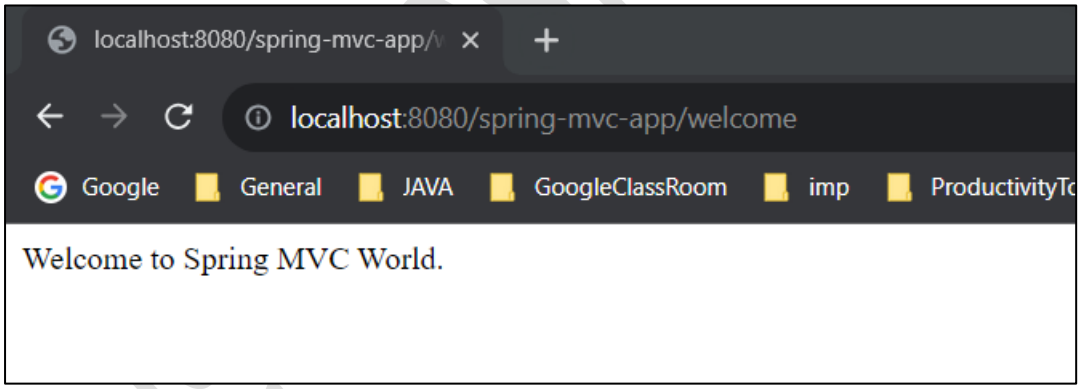


We have Created One URI with /welcome. Now Test this URI by framing URL

Syntax : **http://localhost:8080/<project-name>/<URI>**

URL : **http://localhost:8080/spring-mvc-app/welcome**

Enter From Browser.

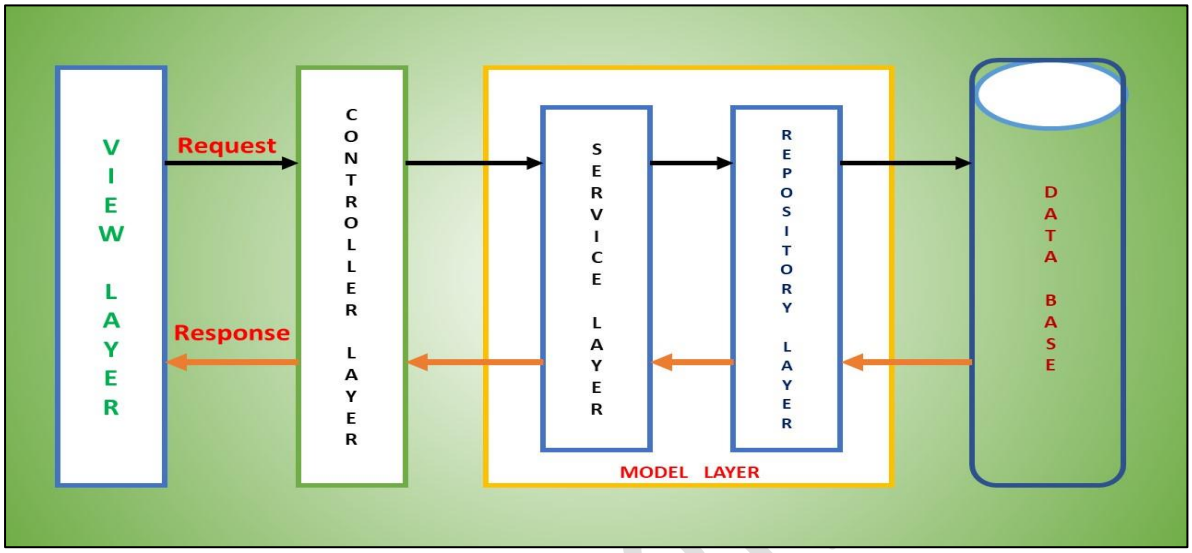


Now We got Response, So Project Setup is Completed.

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.

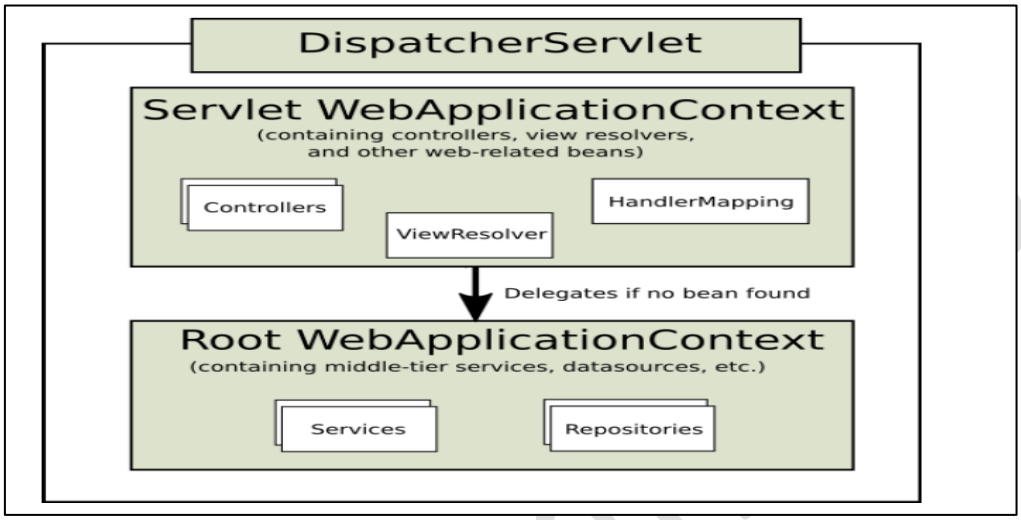
Front Controller:

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 : IphoneCotroller.java**

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

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

```
@Controller
public class IphoneCotroller {

	@GetMapping("/message")
	@ResponseBody
	public String printIphoneMessage() {
		//Logic of Method
		return " Welcome to Ihpne World.";
	}

	@GetMapping("/cost")
	@ResponseBody
	public String printIphone14Cost() {
		return " Price is INR : 150000";
	}
}
```

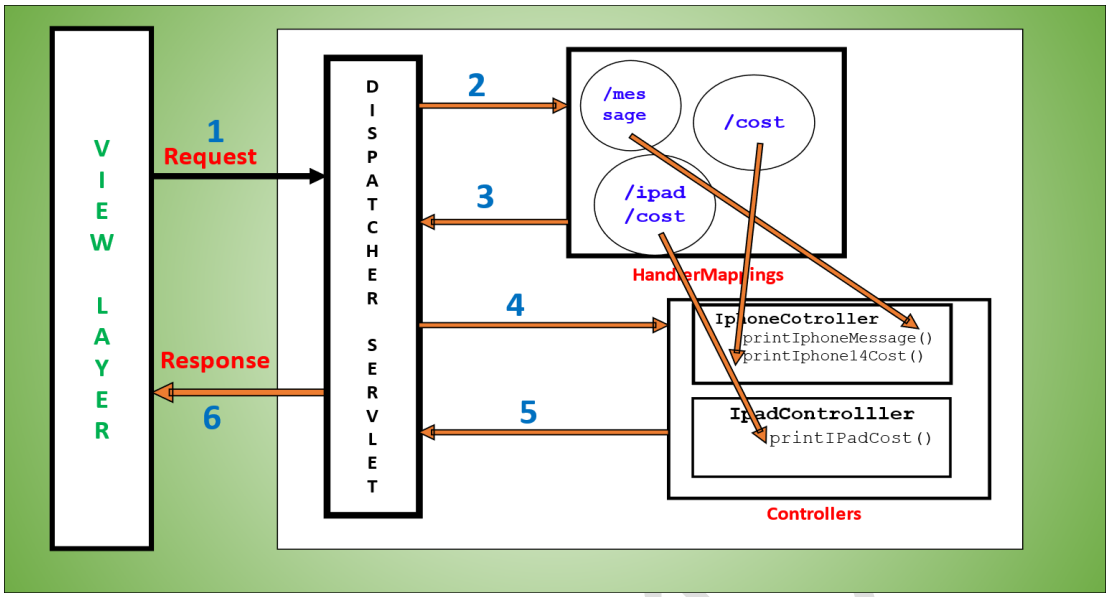
- **Create another Controller class:** IpadControlller.java

```
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 IpadControlller {
	@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 **WebApplcationContext** i.e. Spring Container to instantiate and manage all Spring Beans associated to our project.
- Spring instantiates Pre Defined Front Controller class called as **DispatcherServlet** as well as **WebApplicationContext** scans all our packages for **@Component, @Controller** etc.. and other Bean Configurations.
- Spring MVC **WebApplicationContext** will scan all our Controller classes which are marked with **@Controller** and starts creating Handler Mappings of all URL patterns defined in side controller classes with Controller and endpoint method names mappings.

In our App level, we created 2 controller classes with total 3 endpoints/URL-patterns.



After Starting/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	String	Assign a name to this mapping.
value	String[]	The primary mapping expressed by this annotation.
method	RequestMethod[]	The HTTP request methods to map to, narrowing the primary mapping: GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE.
headers	String[]	The headers of the mapped request, narrowing the primary mapping.
path	String[]	The path mapping URIs (e.g. "/profile").

consumes	String[]	media types that can be consumed by the mapped handler. Consists of one or more media types one of which must match to the request `Content-Type` header.
		
		`consumes = "text/plain"`
		`consumes = {"text/plain", "application/*"}`
		`consumes = MediaType.TEXT_PLAIN_VALUE`

produces	String[]	mapping by media types that can be produced by the mapped handler. Consists of one or more media types one of which must be chosen via content negotiation against the "acceptable" media types of the request.
		
		`produces = "text/plain"`
		`produces = {"text/plain", "application/*"}`
		`produces = MediaType.TEXT_PLAIN_VALUE`
		`produces = "text/plain;charset=UTF-8"`

params	String[]	The parameters of the mapped request, narrowing the primary mapping. Same format for any environment: a sequence of "myParam=myValue" style expressions, with a request only mapped if each such parameter is found to have the given value.

Note: This annotation can be used both at the class and at the method level. In most cases, at the method level applications will prefer to use one of the HTTP method specific variants **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**.

Example: without any attributes with method level

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

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

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

@RequestMapping("/message"):

1. If we are not defined in HTTP method type attribute and value, then same handler method will be executed for all HTTP methods along with endpoint.
2. @RequestMapping("/message") is equivalent to @RequestMapping(value="/message") or @RequestMapping(path="/message")

   i.e. **value** and **path** are same to configure URL path of handler method. We can use either of them. **value** is an alias for **path**.

Example : With method attribute and 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 IpadControlller {

	@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.

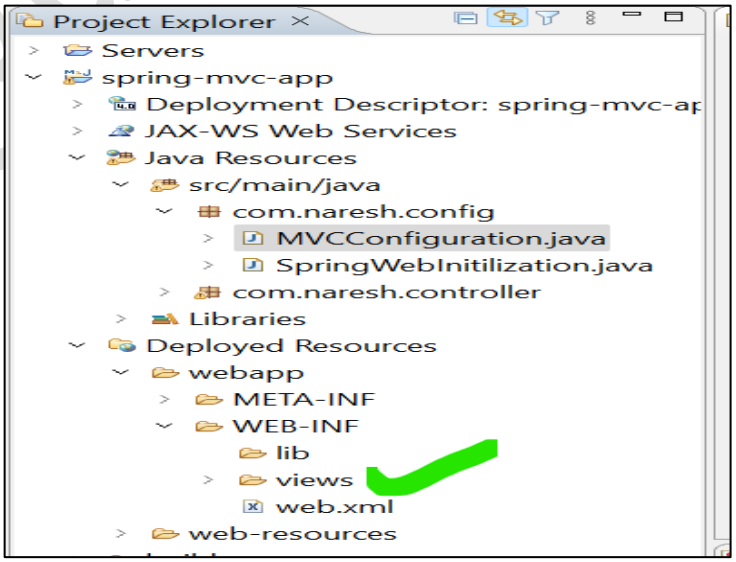
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>spring-mvc-app</groupId>
 <artifactId>spring-mvc-app</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>
</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 crate a folder views inside WEB-INF, where we have to place all our JSP files.



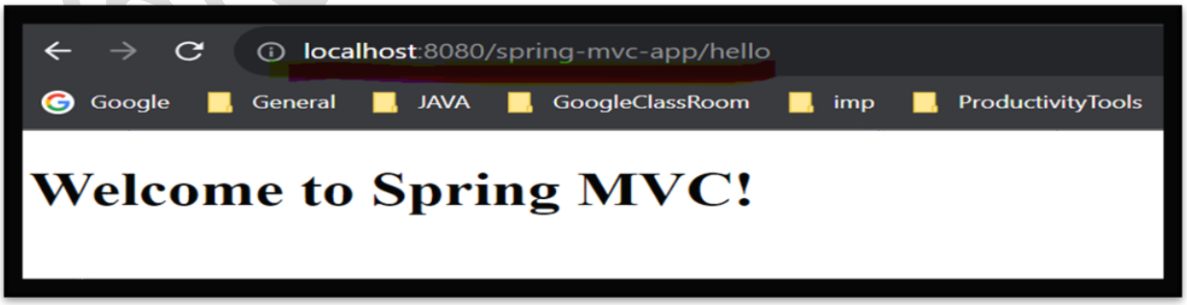
Step 3: Create JSP file inside views folder.

```
<%@ 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>
```

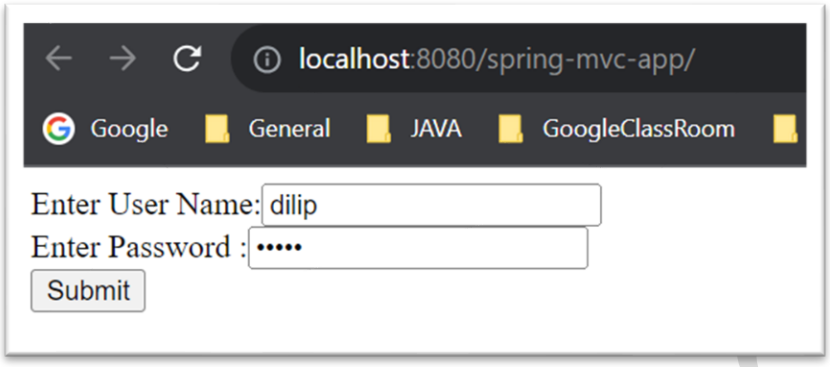
- **Define Controller Methods for login page and validation of User Credentials.**

```
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
    }
}
```

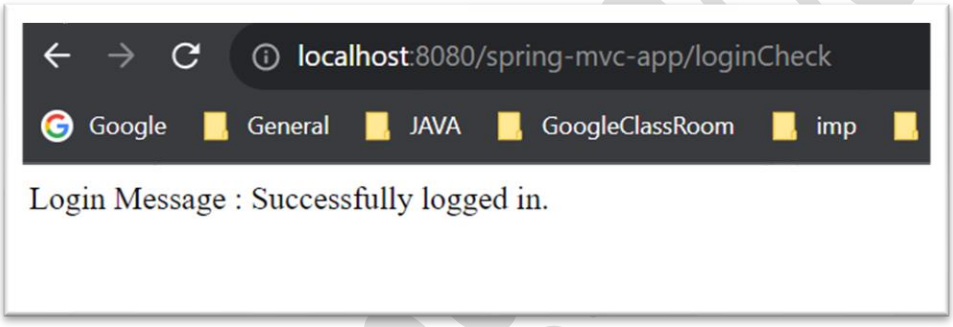
Now Test our Login Page and Validation of Credentials.

Login Page URL : **http://localhost:8080/spring-mvc-app/**

- **Enter User Name and password**



After Submit,



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

This example is a JSON string: `{"name":"John", "age":30, "car":null}`

JSON is a lightweight data-interchange format. JSON is plain text written in JavaScript object notation. JSON is used to exchange data between multiple applications/services. JSON is language independent.

JSON Syntax Rules

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

Example: "name":"John"

In JSON, values must be one of the following data types:

- a string
- a number
- an object
- an array
- a boolean
- null

JSON vs XML:

Both JSON and XML can be used to receive data from a web server. The following JSON and XML examples both define an employee's object, with an array of 3 employees:

JSON Example

```
{
  "employees": [
    {
      "firstName": "John",
      "lastName": "Doe"
    },
    {
      "firstName": "Anna",
      "lastName": "Smith"
    },
    {
      "firstName": "Peter",
      "lastName": "Jones"
    }
  ]
}
```

XML Example:

```
<employees>
      <employee>
            <firstName>John</firstName>
            <lastName>Doe</lastName>
       </employee>
       <employee>
            <firstName>Anna</firstName>
            <lastName>Smith</lastName>
       </employee>
       <employee>
            <firstName>Peter</firstName>
            <lastName>Jones</lastName>
       </employee>
</employees>
```

JSON is Like XML Because

- Both JSON and XML are "self-describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages

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 Obje**ct 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..

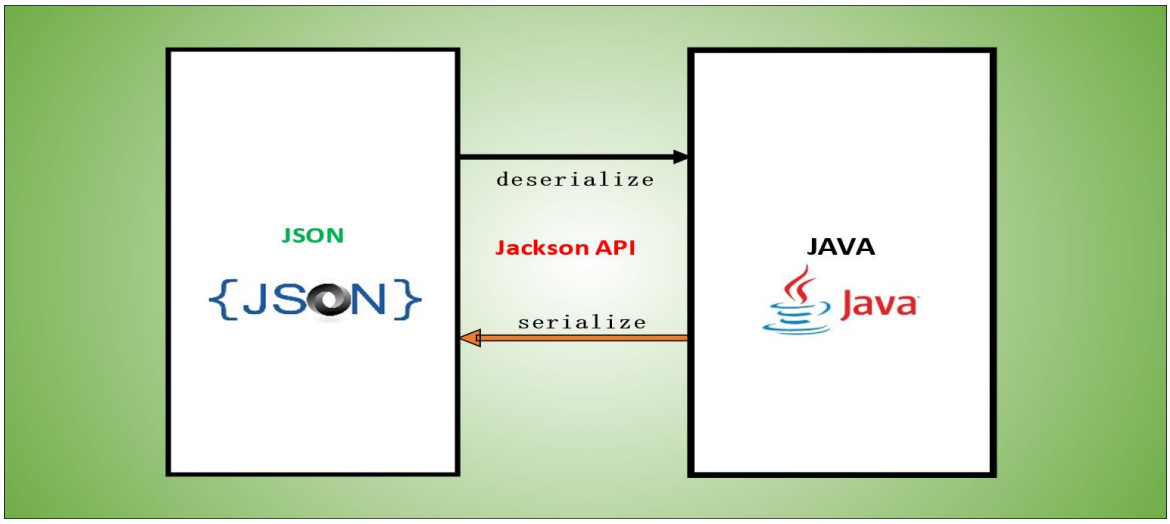
JSON to JAVA Conversion:

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-marshal JSON string to Java object. Similarly, java method return data will be converted to JSON format i.e. Response of Endpoint by using an annotation **@ResponseBody.**

And as you have annotated with **@ResponseBody** of endpoint method, we no need to do explicitly JAVA to JSON conversion. Just return a POJO and Jackson serializer will take care of converting to Json format. It is equivalent to using **@ResponseBody** when used with **@Controller**. Rather than placing **@ResponseBody** on every controller method we place

@RestController instead of **@Controller** and **@ResponseBody** by default is applied on all resources in that controller.

Note: we should carte Java POJO classes specific to JSON payload structure, to enable auto conversion between JAVA and JSON.



JSON with Array of String values:

JSON Payload: Below Json contains ARRY of String Data Type values

```
{
   "student":[
         "Dilip",
         "Naresh",
         "Mohan",
         "Laxmi"
   ]
}
```

Java Class: JSON Array of String will be takes as **List<String>** with JSON key name.

```
import java.util.List;

public class StudentDetails {

        private List<String> student;

        public List<String> getStudent() {
                return student;
        }
        public void setStudent(List<String> student) {
                this.student = student;
```

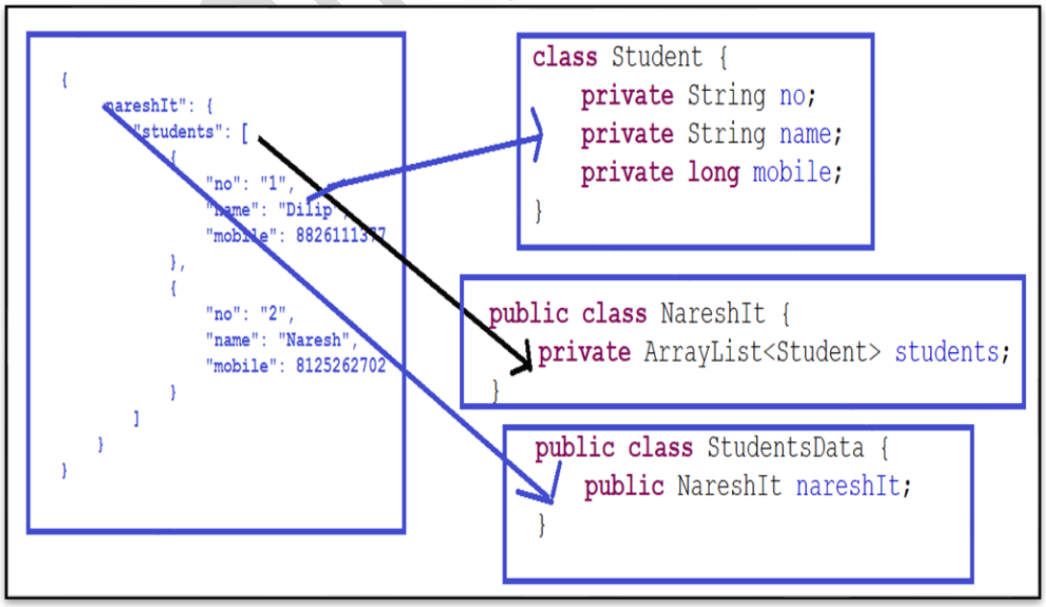
```
        }
        @Override
        public String toString() {
                return "StudentDetails [student=" + student + "]";
        }
}
```

JSON payload with Array of Student Object Values:

Below JSON payload contains array of Student values.

```
{
 "nareshIt": {
  "students": [
   {
    "no": "1",
    "name": "Dilip",
    "mobile": 8826111377
   },
   {
    "no": "2",
    "name": "Naresh",
    "mobile": 8125262702
   }
  ]
 }
}
```

Below picture showing how are creating JAVA classes from above payload.



Created Three java files to wrap above JSON payload structure:

Student.java

```
public class Student {
	private String no;
	private String name;
	private long mobile;

	//Setters and Getters
}
```

Add Student as List in class **NareshIt.java**

```
import java.util.ArrayList;
public class NareshIt {
	private ArrayList<Student> students;

	//Setters and Getters

}
```

StudentsData.java

```
public class StudentsData {
	public NareshIt nareshIt;

	//Setters and Getters
}
```

Now we will use **StudentsData** class to bind our JSON Payload.

- **Let's Take another Example of JSON to JAVA POJO class:**

JSON PAYLOAD: Json with Array of Student Objects

```
{
  "student": [
    {
      "firstName": "Dilip",
      "lastName": "Singh",
      "mobile": 88888,
      "pwd": "Dilip",
      "emailID": "Dilip@Gmail.com"
    },
```

```
                        {
                          "firstName": "Naresh",
                          "lastName": "It",
                          "mobile": 232323,
                          "pwd": "Naresh",
                          "emailID": "Naresh@Gmail.com"
                        }
                       ]
                      }
```

For the above Payload, JAVA POJO'S are:

```
import com.fasterxml.jackson.annotation.JsonProperty;

public class StudentInfo {

        private String firstName;
        private String lastName;
        private long mobile;
        private String pwd;
        @JsonProperty("emailID")
        private String email;

        //Setters and Getters
}
```

Another class To Wrap above class Object as List with property name student as per JSON.

```
import java.util.List;

public class Students {

        List<StudentInfo> student;

        public List<StudentInfo> getStudent() {
                return student;
        }
        public void setStudent(List<StudentInfo> student) {
                this.student = student;
        }
}
```

From the above JSON payload and JAVA POJO class, we can see a difference for one JSON property called as **emailID** i.e. in JAVA POJO class property name we taken as **email** instead of emailID. In Such case to map JSON to JAVA properties with different names, we use an annotation called as @JsonProperty("jsonPropertyName").

@JsonProperty:

The **@JsonProperty** annotation is used to specify the property name in a JSON object when serializing or deserializing a Java object using the Jackson API library. It is often used when the JSON property name is different from the field name in the Java object, or when the JSON property name is not in camelCase.

If you want to serialize this object to JSON and specify that the JSON property names should be "**first_name**", "**last_name**", and "age", you can use the @JsonProperty annotation like this:

```
public class Person {
   @JsonProperty("first_name")
   private String firstName;
   @JsonProperty("last_name")
   private String lastName;
   @JsonProperty
   private int age;

   // getters and setters go here
}
```

As a developer, we should always create POJO classes aligned to JSON payload to bind JOSN 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 Jakson 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>
```

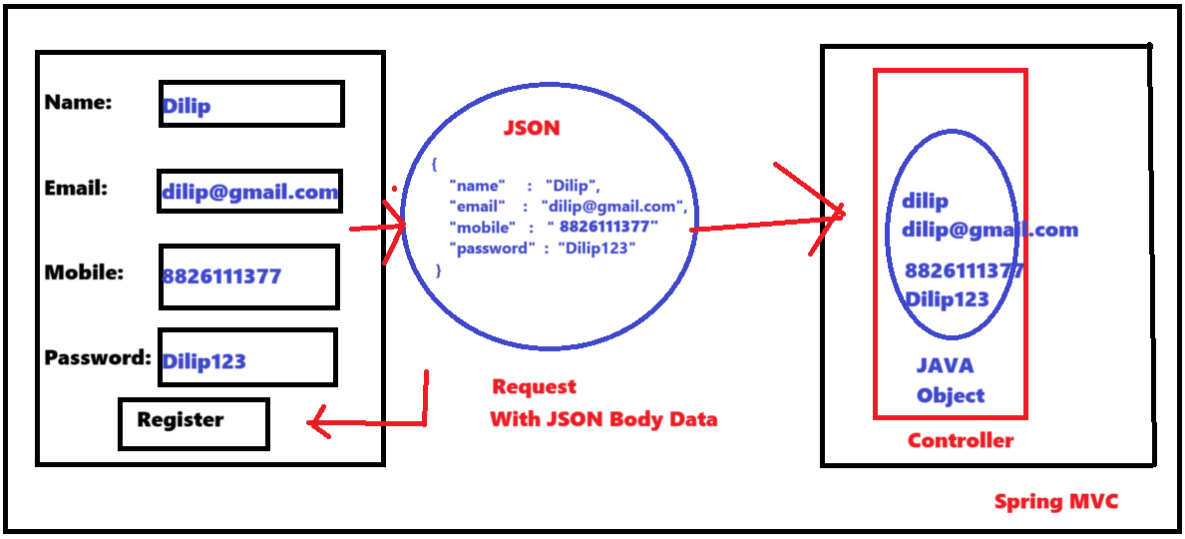
```
		</dependency>
		<dependency>
			<groupId>com.fasterxml.jackson.core</groupId>
			<artifactId>jackson-databind</artifactId>
			<version>2.15.2</version>
		</dependency>
		<dependency>
			<groupId>com.fasterxml.jackson.core</groupId>
			<artifactId>jackson-annotations</artifactId>
			<version>2.15.2</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;
        }
}
```

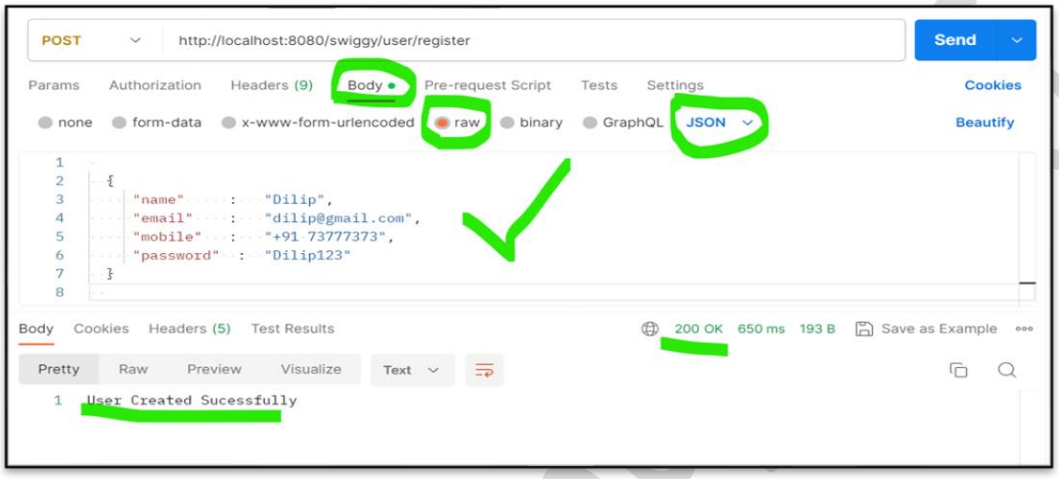
6. Now Create A controller and inside an endpoint for User Register Request Handling.

```
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";
        }
}
```

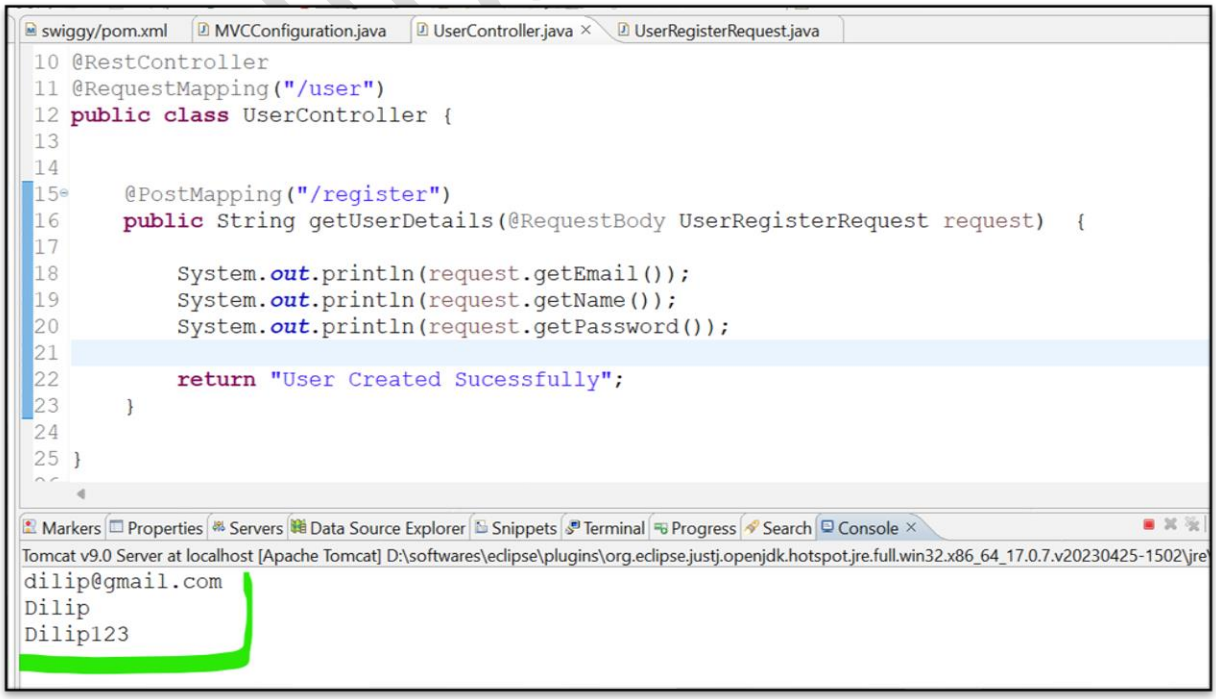
In Above, We are used **@RequestBody** for binding/mapping incoming JSON request to JAVA Object at method parameter layer level. Means, Spring MVC internally maps JSON to JAVA with help of Jackson API jar files.

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.

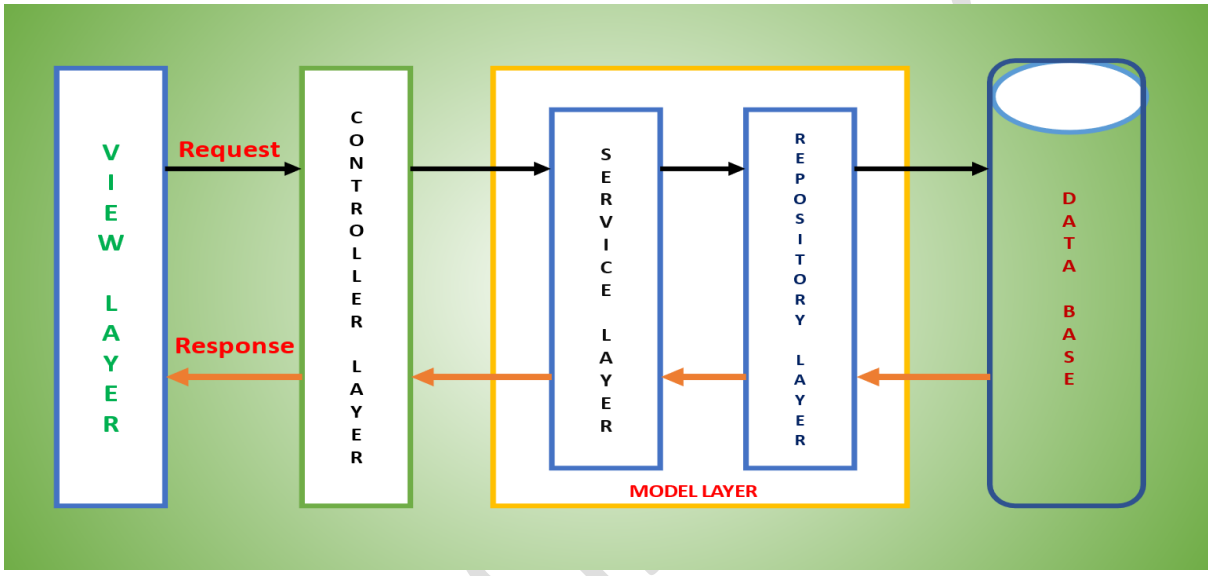


After Clicking on **Send** Button, Summited Request to Spring MVC REST Service Endpoint method and we got Response back with **200 Ok** status Code. We can See in Server Console, Request Data printed what we Received from Client level as JSON data.



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;
```

```java
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;
```

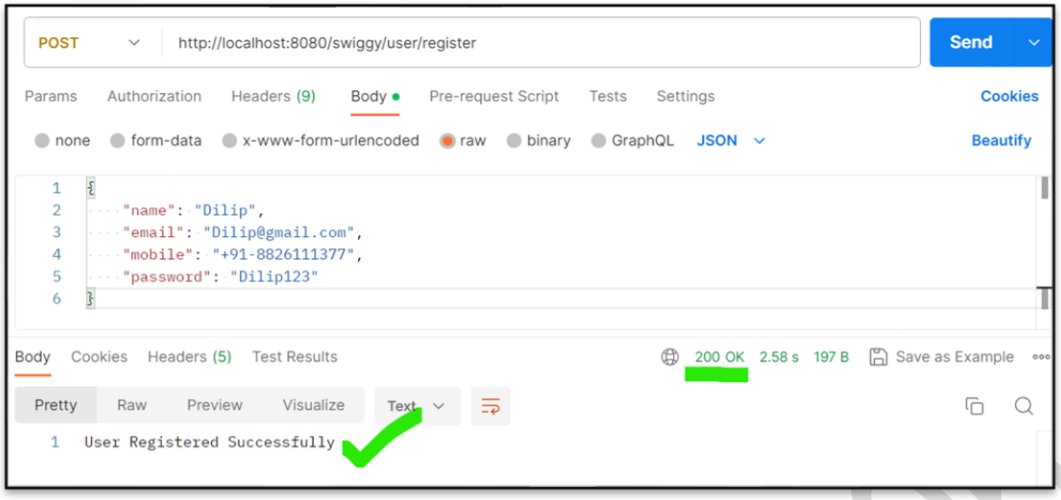
```
import com.swiggy.user.repository.UserRepository;
import com.swiggy.user.request.UserRegisterRequest;
import com.swiggy.user.response.UserRegisterResponse;

@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.



- Verify In Database. Data Inserted.

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;
```

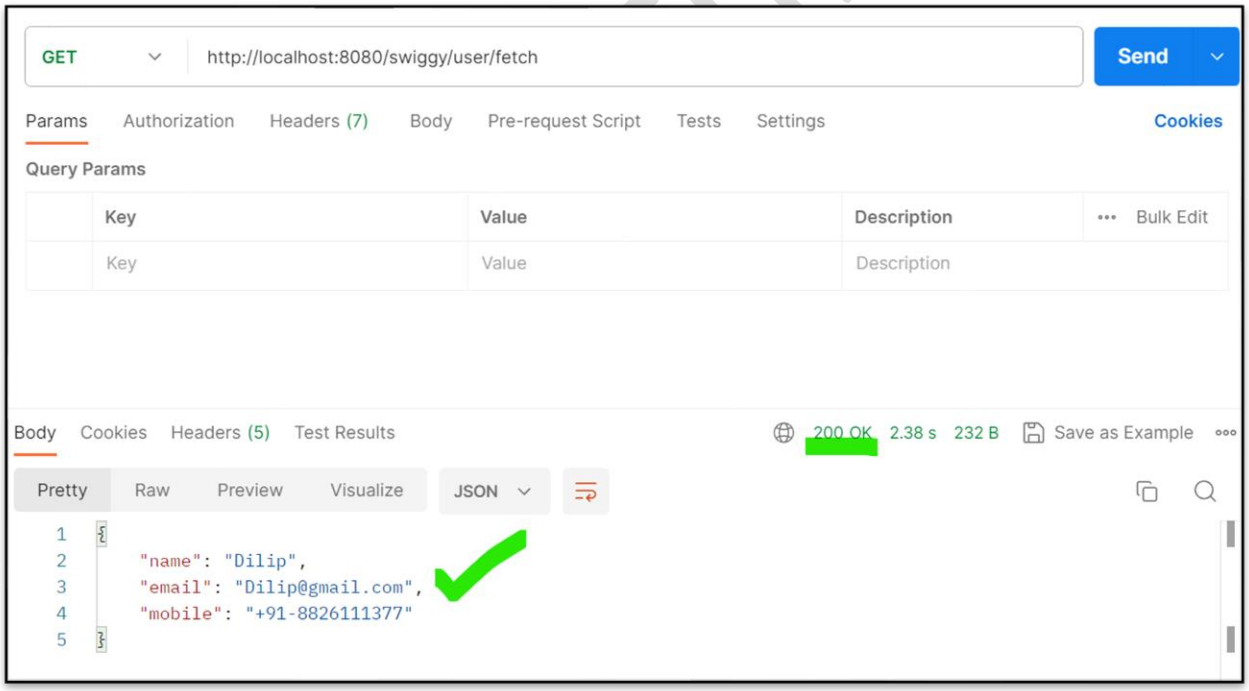
```
	@PostMapping("/register")
	public String registerUserDetails(@RequestBody UserRegisterRequest request) {
		// Controller -> Service and From Service receiving Result
		String response = userService.registerUserDetails(request);
		return response;
	}

	// 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();
	}
}
```

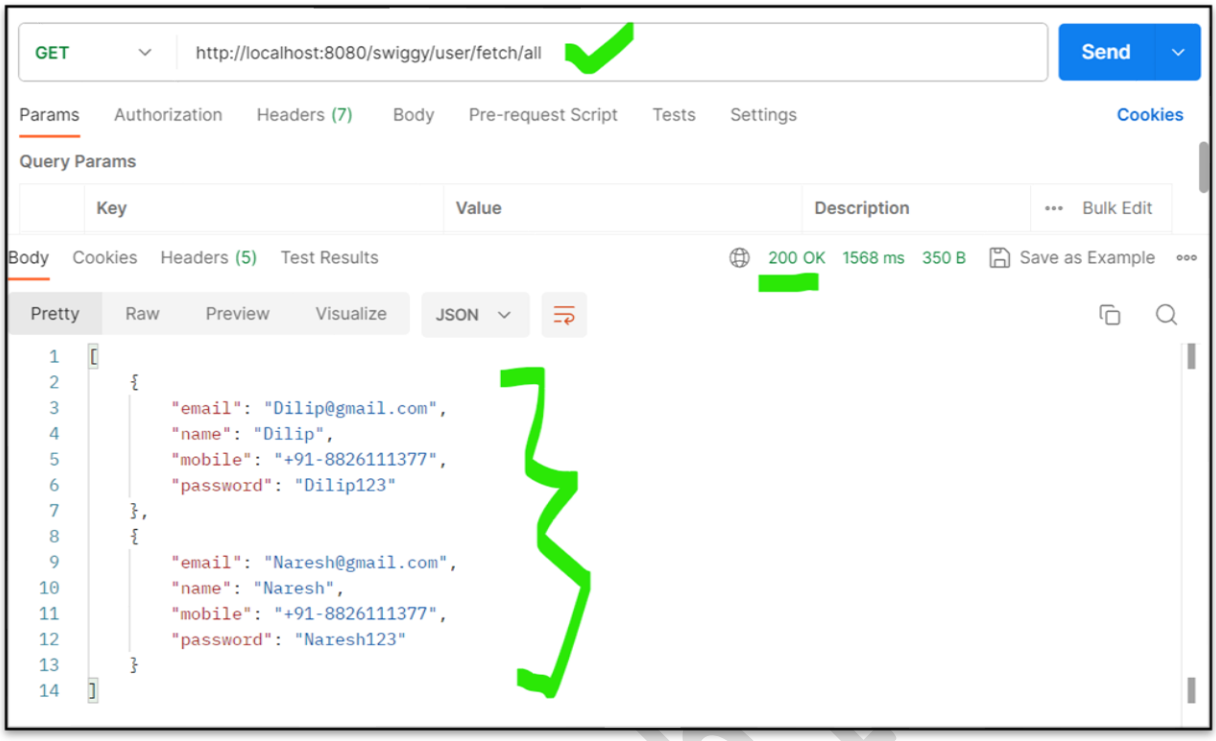
UserService.java

```
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.**





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{}.

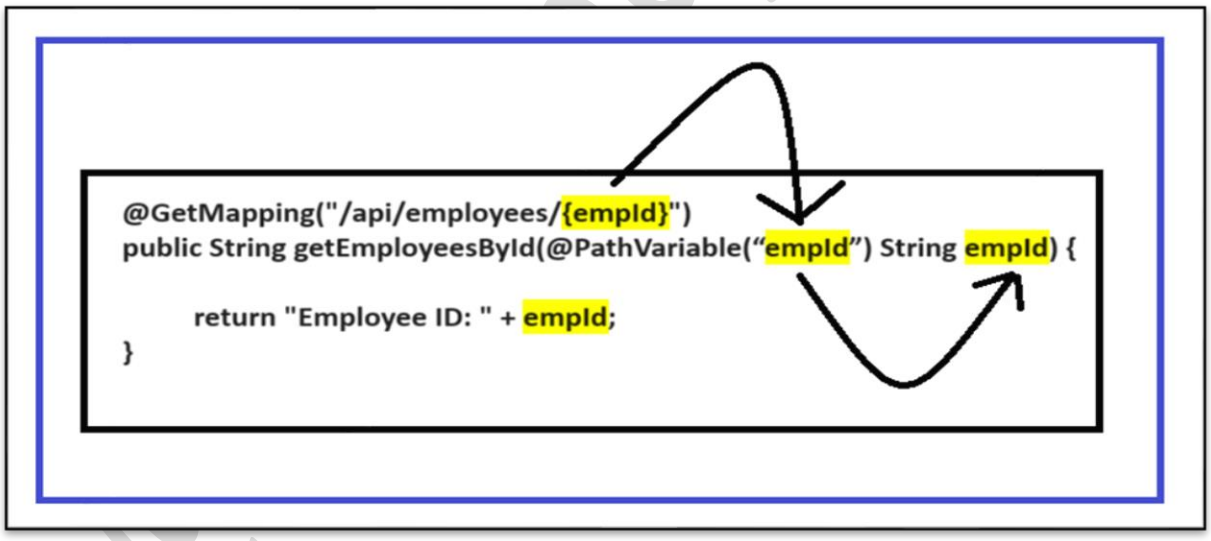
Package of Annotation: org.springframework.web.bind.annotation.PathVariable;

Examples:

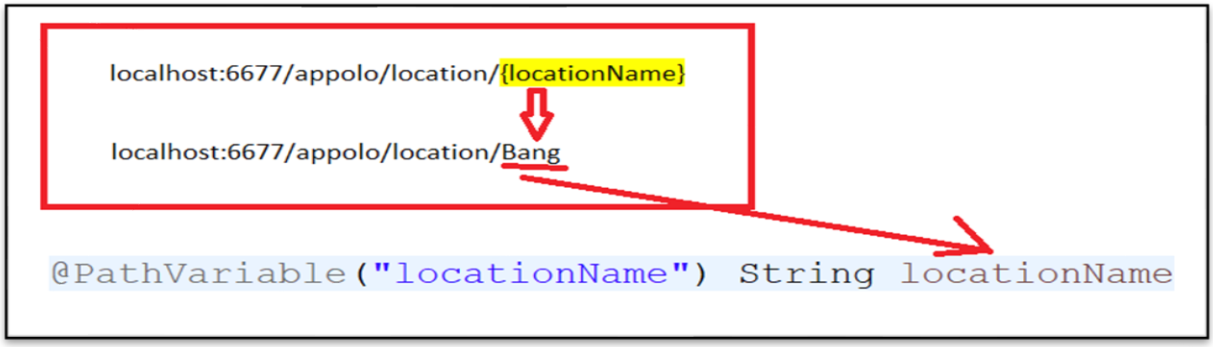
URI with Template Path variables : /location/{**locationName**}/pincode/**{pincode}**
URI with Data replaced : /location/**Hyderabad**/pincode/**500072**

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.

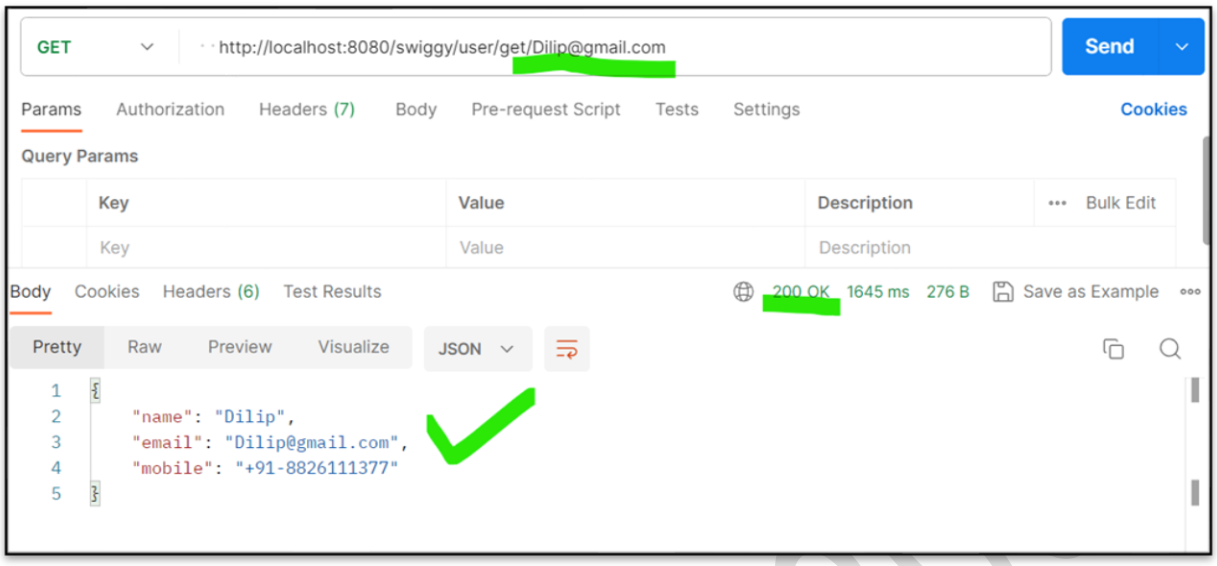
```
@RequestMapping(value = "/get/{emailId}", method = RequestMethod.GET)
public UserRegisterResponse getUserByEmailId(@PathVariable("emailId") String email) {
        return userService.getUserDetails(email);
}
```

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



Multiple Path Variable as part of URI:

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

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

Example: /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>	getOrdersByemaailIDAndStaus(@PathVariable	String
emailId,
						@PathVariable("status") String orderStaus ){
	   List<OrderResponse> orders =
				orderService.getOrdersByemaailIDAndStaus(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 orderReposity;

	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());
		orderReposity.save(order);
		return "Order Creeated Successfully";
	}

	 public List<OrderResponse> getOrdersByemaailIDAndStaus(String emailId,
	                                                         String orderStaus) {
			List<SwiggyOrders> orders =
				orderReposity.findByEmailIdAndOrderstatus(emailId, orderStaus);

				 List<OrderResponse>  allOrders =  orders.stream().map(
				 v -> new OrderResponse(
				  v.getOrderID(),
				  v.getOrderstatus(),
				  v.getAmount(),
				  v.getEmailId(),
				  v.getCity()
			    )).collect(Collectors.toList());

		return allOrders;
	}
}
```

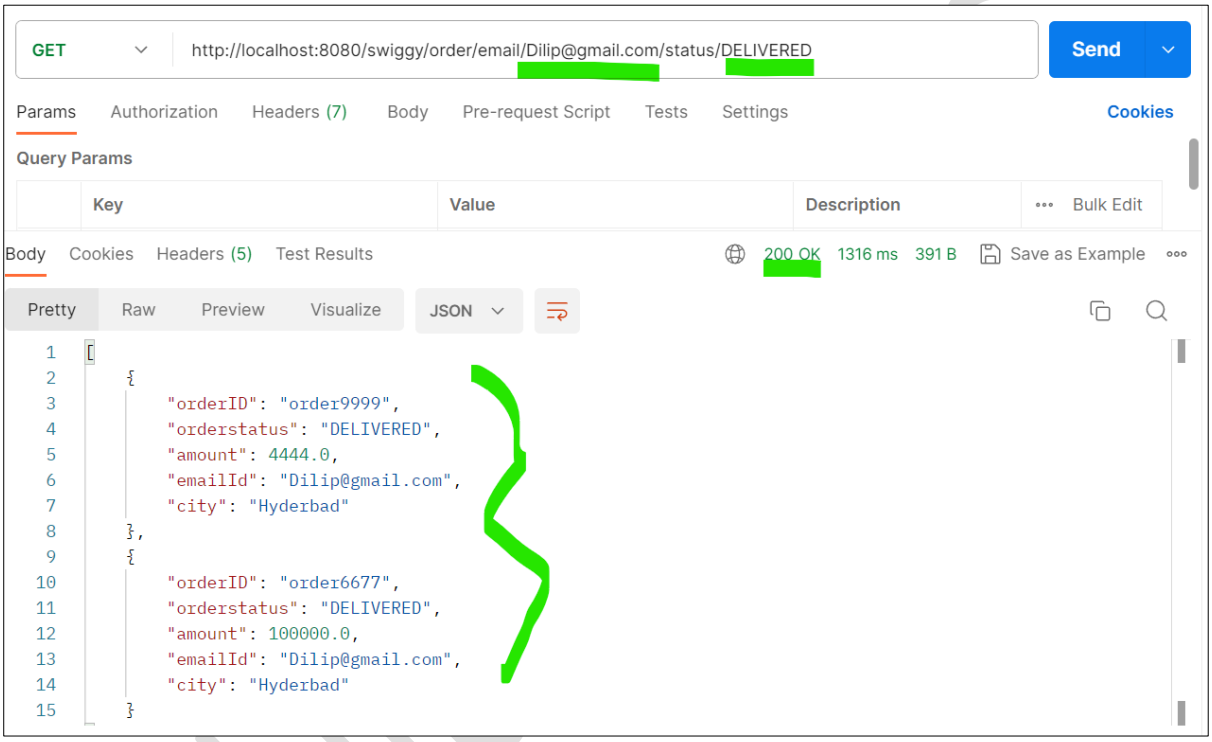
- **Create Repository : OrderRepository.java**

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 orderStaus);
}
```

From Postman Test end point: URL formation, replacing Path variables with real values



- **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

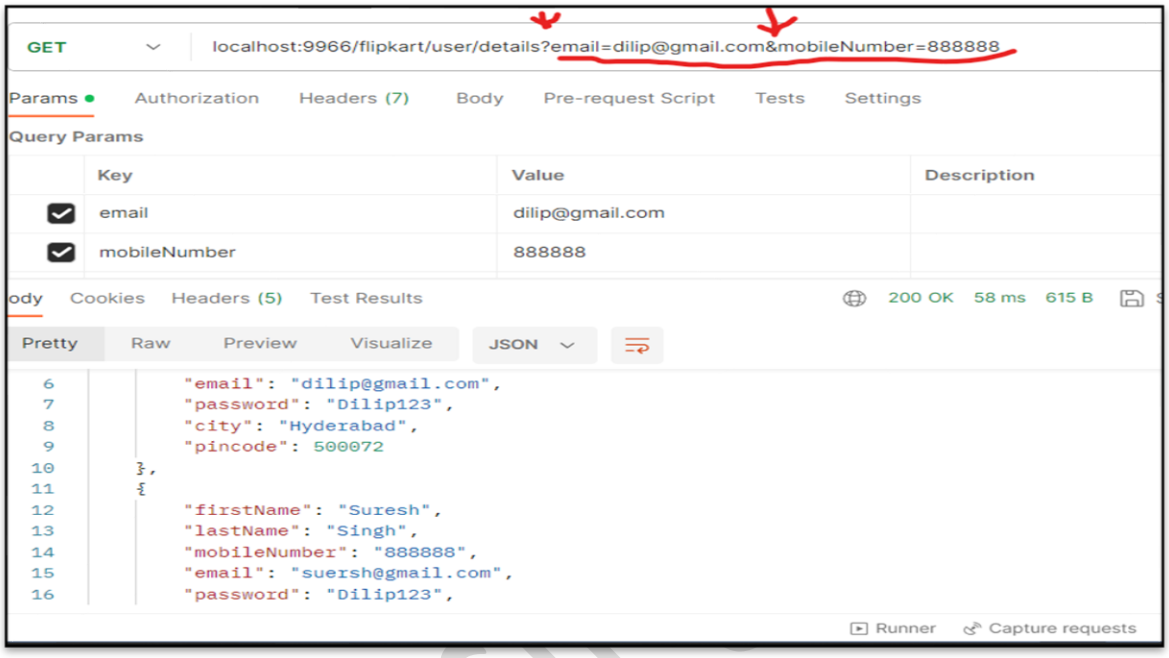
Method in controller:

```
@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>**



HTTP status codes in building RESTful API's:

HTTP status codes are three-digit numbers that are returned by a web server in response to a client's request made to a web page or resource. These codes indicate the outcome of the request and provide information about the status of the communication between the client (usually a web browser) and the server. They are an essential part of the HTTP (Hypertext Transfer Protocol) protocol, which is used for transferring data over the internet. HTTP defines these standard status codes that can be used to convey the results of a client's request.
The status codes are divided into five categories.

1xx: Informational	Communicates transfer protocol-level information.
2xx: Success	Indicates that the client's request was accepted successfully.
3xx: Redirection	Indicates that the client must take some additional action in order to complete their request.
4xx: Client Error	This category of error status codes points the finger at clients.
5xx: Server Error	The server takes responsibility for these error status codes.

Some of HTTP status codes summary being used mostly in REST API creation

1xx Informational:

This series of status codes indicates informational content. This means that the request is received and processing is going on. Here are the frequently used informational status codes:

100 Continue: This code indicates that the server has received the request header and the client can now send the body content. In this case, the client first makes a request (with the Expect: 100-continue header) to check whether it can start with a partial request. The server can then respond either with 100 Continue (OK) or 417 Expectation Failed (No) along with an appropriate reason.

101 Switching Protocols: This code indicates that the server is OK for a protocol switch request from the client.

102 Processing: This code is an informational status code used for long-running processing to prevent the client from timing out. This tells the client to wait for the future response, which will have the actual response body.

2xx Success:

This series of status codes indicates the successful processing of requests. Some of the frequently used status codes in this class are as follows.

200 OK: This code indicates that the request is successful and the response content is returned to the client as appropriate.

201 Created: This code indicates that the request is successful and a new resource is created.

204 No Content: This code indicates that the request is processed successfully, but there's no return value for this request. For instance, you may find such status codes in response to the deletion of a resource.

3xx Redirection:

This series of status codes indicates that the client needs to perform further actions to logically end the request. A frequently used status code in this class is as follows:

304 Not Modified: This status indicates that the resource has not been modified since it was last accessed. This code is returned only when allowed by the client via setting the request headers as If-Modified-Since or If-None-Match. The client can take appropriate action on the basis of this status code.

4xx Client Error:

This series of status codes indicates an error in processing the request. Some of the frequently used status codes in this class are as follows:

400 Bad Request: This code indicates that the server failed to process the request because of the malformed syntax in the request. The client can try again after correcting the request.

401 Unauthorized: This code indicates that authentication is required for the resource. The client can try again with appropriate authentication.

403 Forbidden: This code indicates that the server is refusing to respond to the request even if the request is valid. The reason will be listed in the body content if the request is not a HEAD method.

404 Not Found: This code indicates that the requested resource is not found at the location specified in the request.

405 Method Not Allowed: This code indicates that the HTTP method specified in the request is not allowed on the resource identified by the URI.

408 Request Timeout: This code indicates that the client failed to respond within the time window set on the server.

409 Conflict: This code indicates that the request cannot be completed because it conflicts with some rules established on resources, such as validation failure.

5xx Server Error:

This series of status codes indicates server failures while processing a valid request. Here is one of the frequently used status codes in this class:

500 Internal Server Error: This code indicates a generic error message, and it tells that an unexpected error occurred on the server and that the request cannot be fulfilled.

501 (Not Implemented): The server either does not recognize the request method, or it cannot fulfil the request. Usually, this implies future availability (e.g., a new feature of a web-service API).

REST Specific HTTP Status Codes:

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

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

PUT - Update Resource : **200 Ok** : Successfully Updated Data
If not i.e. Resource Not Found Data
404 Not Found : Successfully Processed but Data Not available

GET - Read Resource : **200 Ok** : Successfully Retrieved Data
If not i.e. Resource Not Found Data
404 Not Found : Successfully Processed but Data Not available

DELETE - Deleting Resource : **204 No Content**: Successfully Deleted Data
If not i.e. Resource Not Found Data
404 Not Found : Successfully Processed but Data Not available

Binding HTTP status codes and Response in Spring:

To bind response data and relevant HTTP status code with endpoint in side controller class, we will use predefined Spring provided class **ResponseEntity.**

ResponseEntity:

ResponseEntity represents the whole HTTP response: status code, headers, and body. As a result, we can use it to fully configure the HTTP response. If we want to use it, we have to return it from the endpoint, Spring takes care of the rest. ResponseEntity is a generic type. Consequently, we can use any type as the response body. This will be used in Controller methods as well as in RestTemplate.

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 **ResponseEntity**<T> with Response Object Data Type at method declaration as Return type of method.
2. We should bind actual Response Data Object with Http Status Codes by passing as Constructor Parameters of **ResponseEntity** class, and then we returning that **ResponseEntity** Object to HTTP Client.

Few Examples of Controller methods with ResponseEntity:

```
@RestController
public class NetBankingController {

   @PostMapping("/create")
   @ResponseStatus(value = HttpStatus.CREATED)    //Using 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.

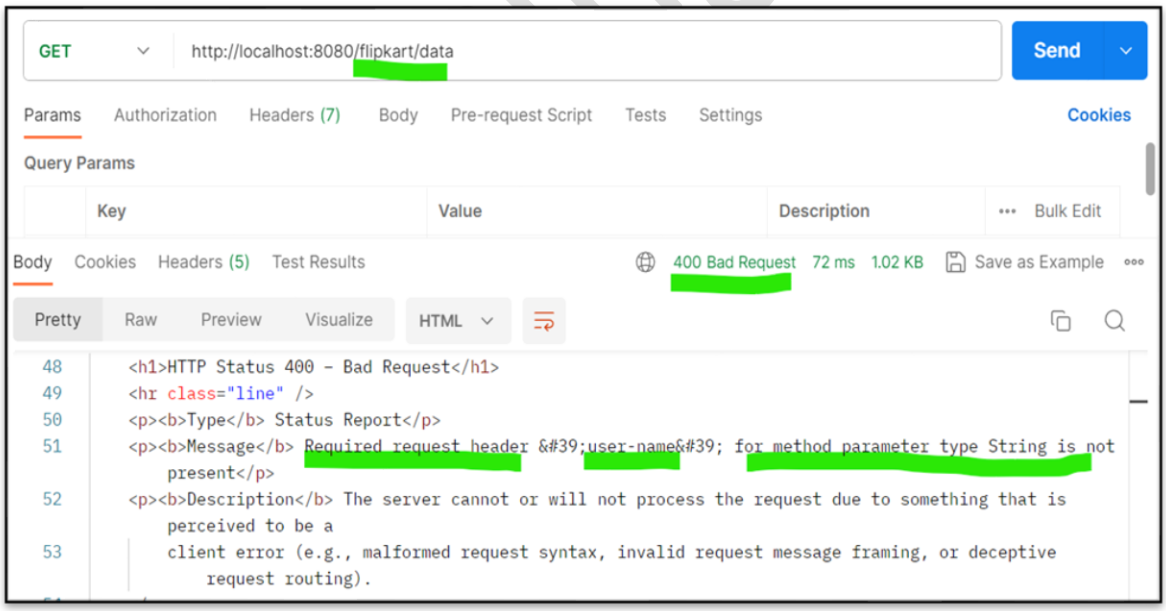
```
@GetMapping("/endpoint")
public ResponseEntity<String> handleRequest(@RequestHeader("Header-Name") String
```

```
                                                        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;
        }
}
```

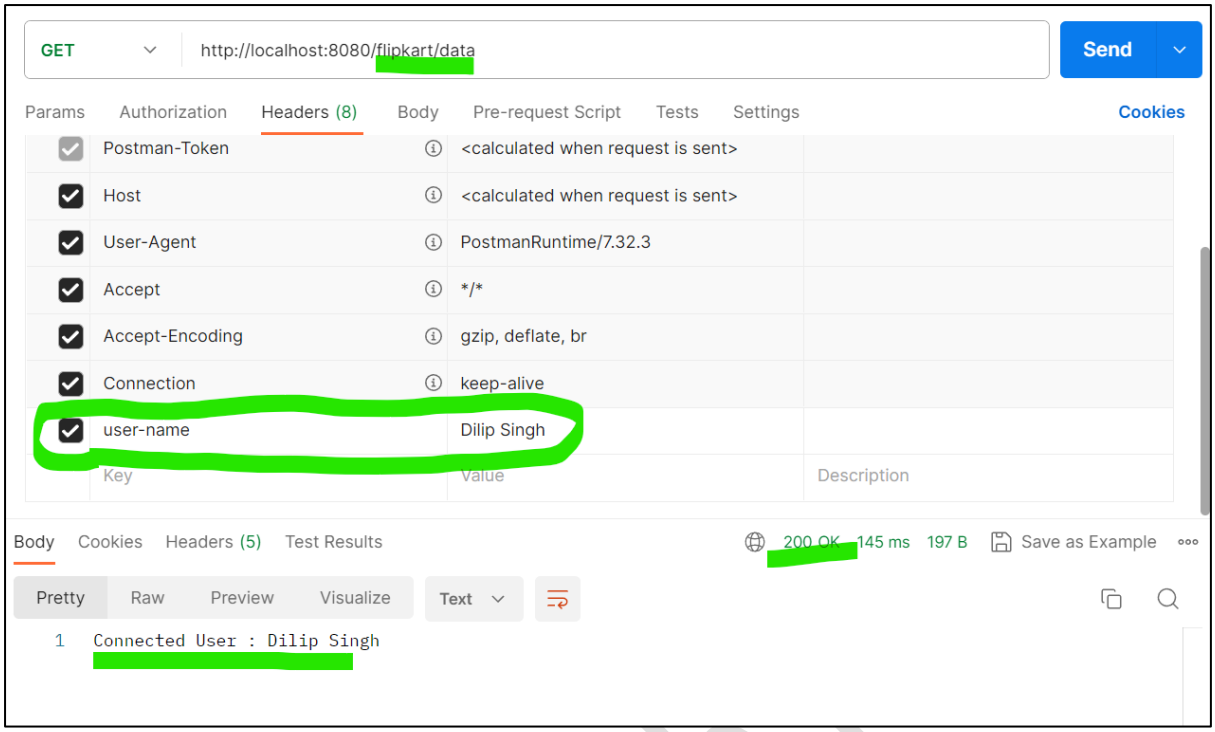
Testing: Without Sending Header and Value from Client, Sending Request to Service.



Result : We Got Bad request like Header is Missing i.e. Header is Mandatory by default if we defined in Controller method.

Setting Header in Client: i.e. In Our Case From Postman:

In Postman, Add header user-name and its value under Headers Section. Now request is executed Successfully.



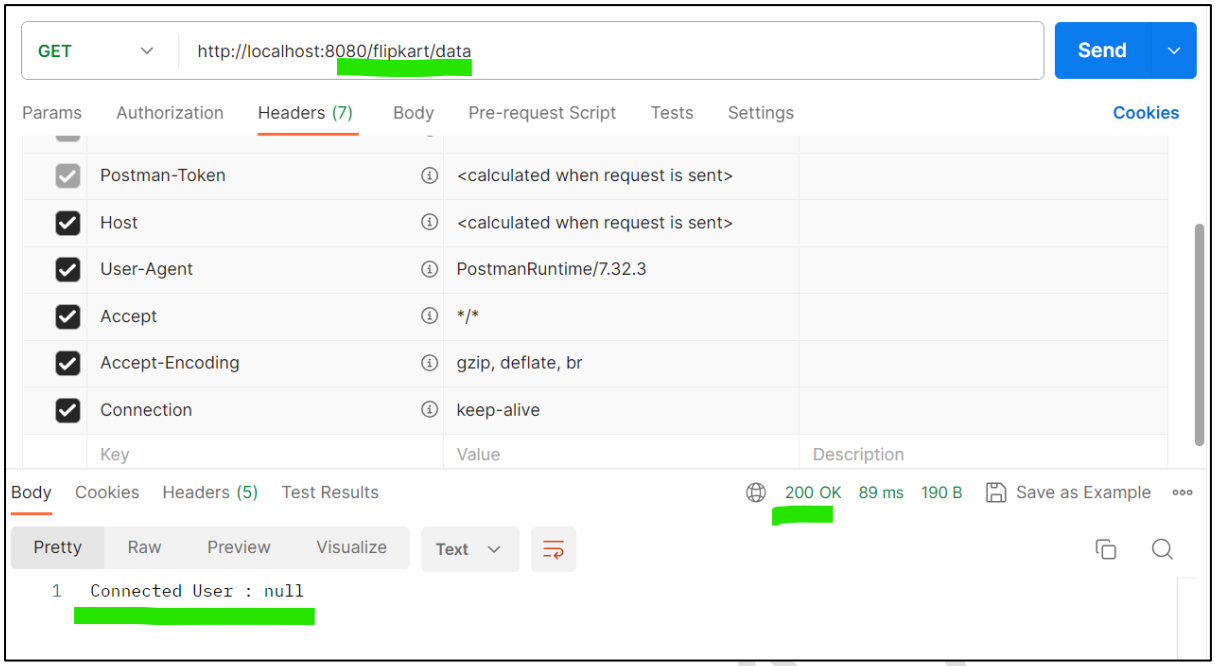
Optional Header:

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.**

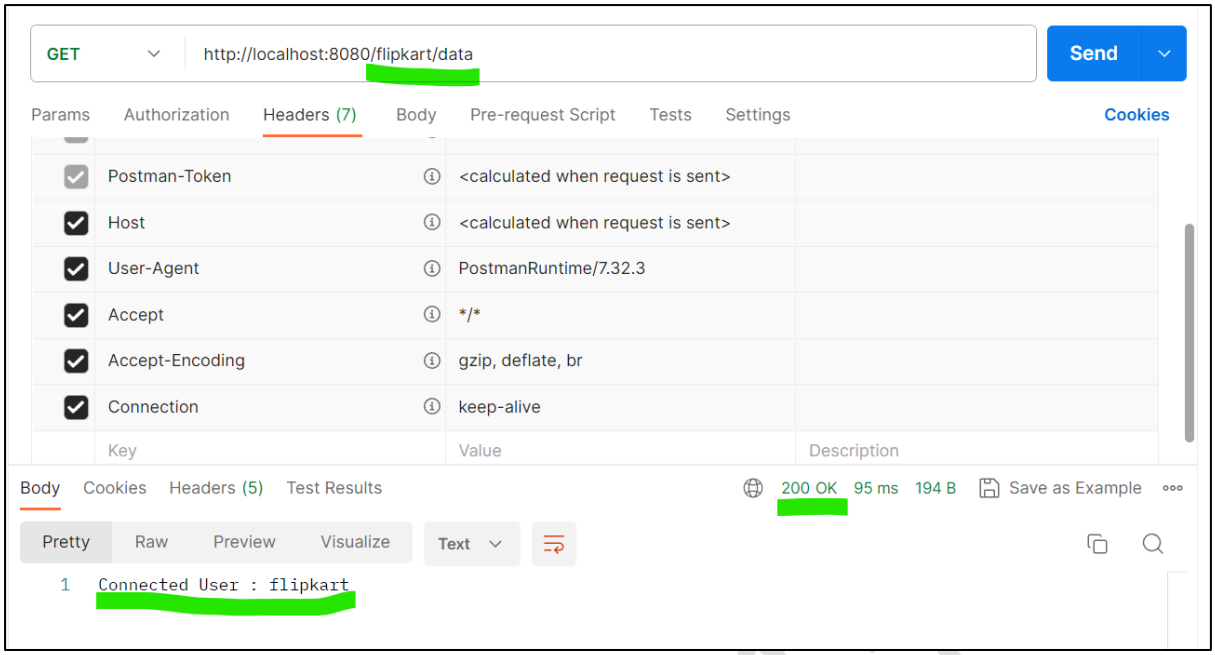


Default Value Of Header:

- **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.**



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:

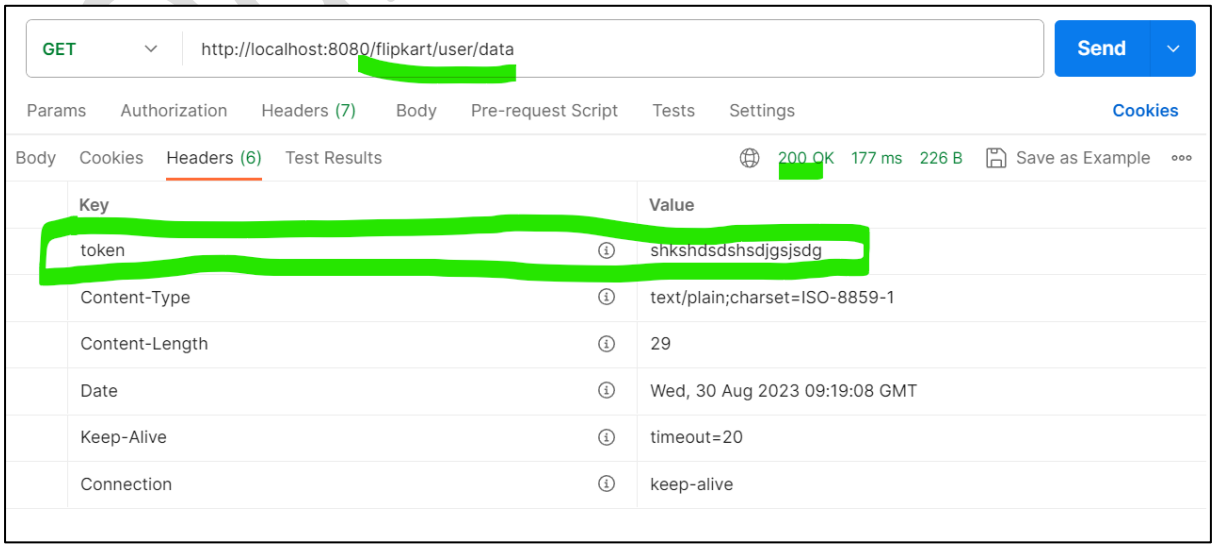
In a Response:

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 it's value as part of response Body.**

```
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

	// Header is Part of Response, i.e. Should be set from Server Side.
	@GetMapping("/user/data")
	public ResponseEntity<String> testResponseHeaders() {
		HttpHeaders headers = new HttpHeaders();
		headers.set("token", "shkshdsdshsdjgsjsdg");
	return new ResponseEntity<String>("Sending Response with Headers", headers,
                                                                HttpStatus.OK);
	}
}
```

Testing: **Trigger endpoint from Client: Got Token and its value from Service in Headers.**

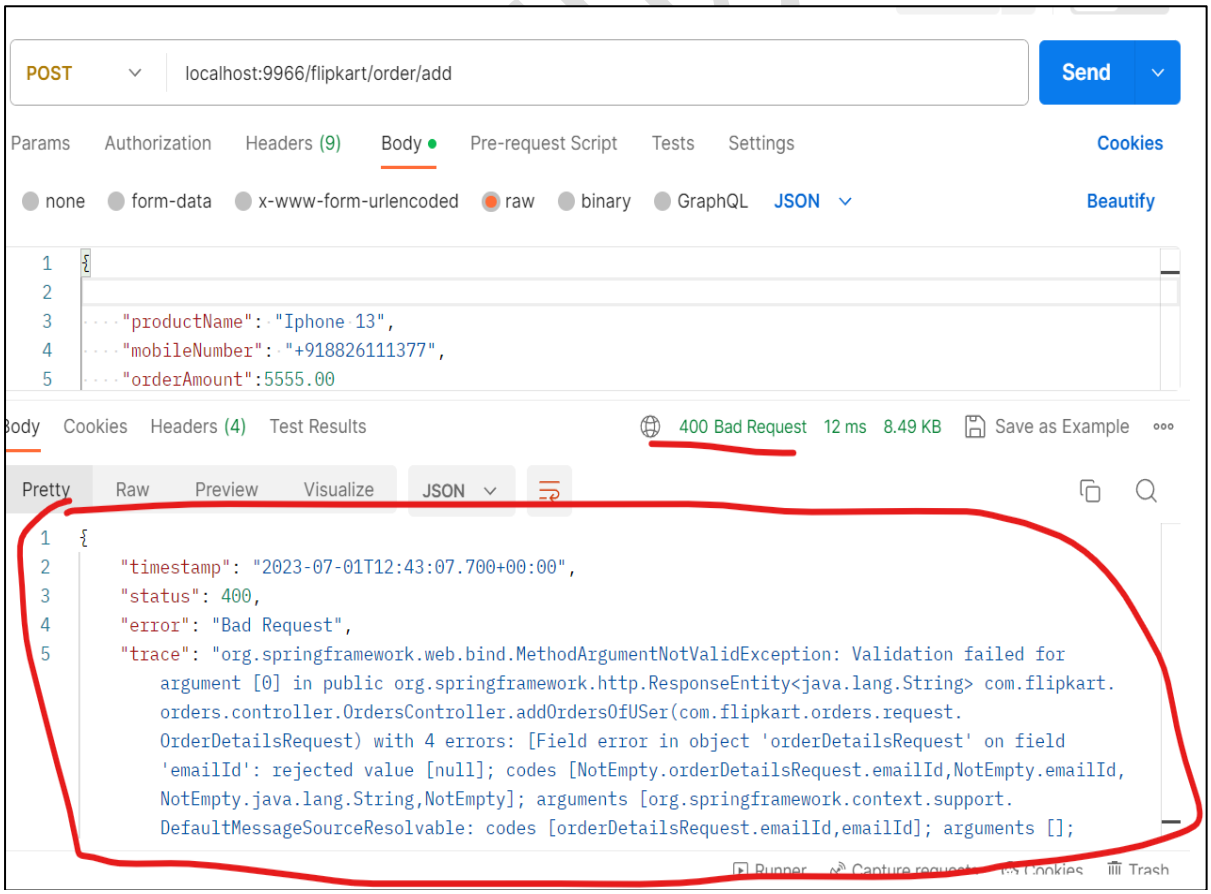


Exception Handling in Spring MVC Controllers:

What I have to do with Errors or Exceptions ?

Spring brought **@ExceptionHandler** & **@ControllerAdvice** annotations for handling Exceptions thrown at controller layer. So we can handle exceptions and will be forwarded meaning full Error response messages with response status code to HTTP clients.

If we are not handled exceptions then we will see Exception stack trace as shown in below at HTTP client level as a response. As a Best Practice we should show meaningful Error Response messages.

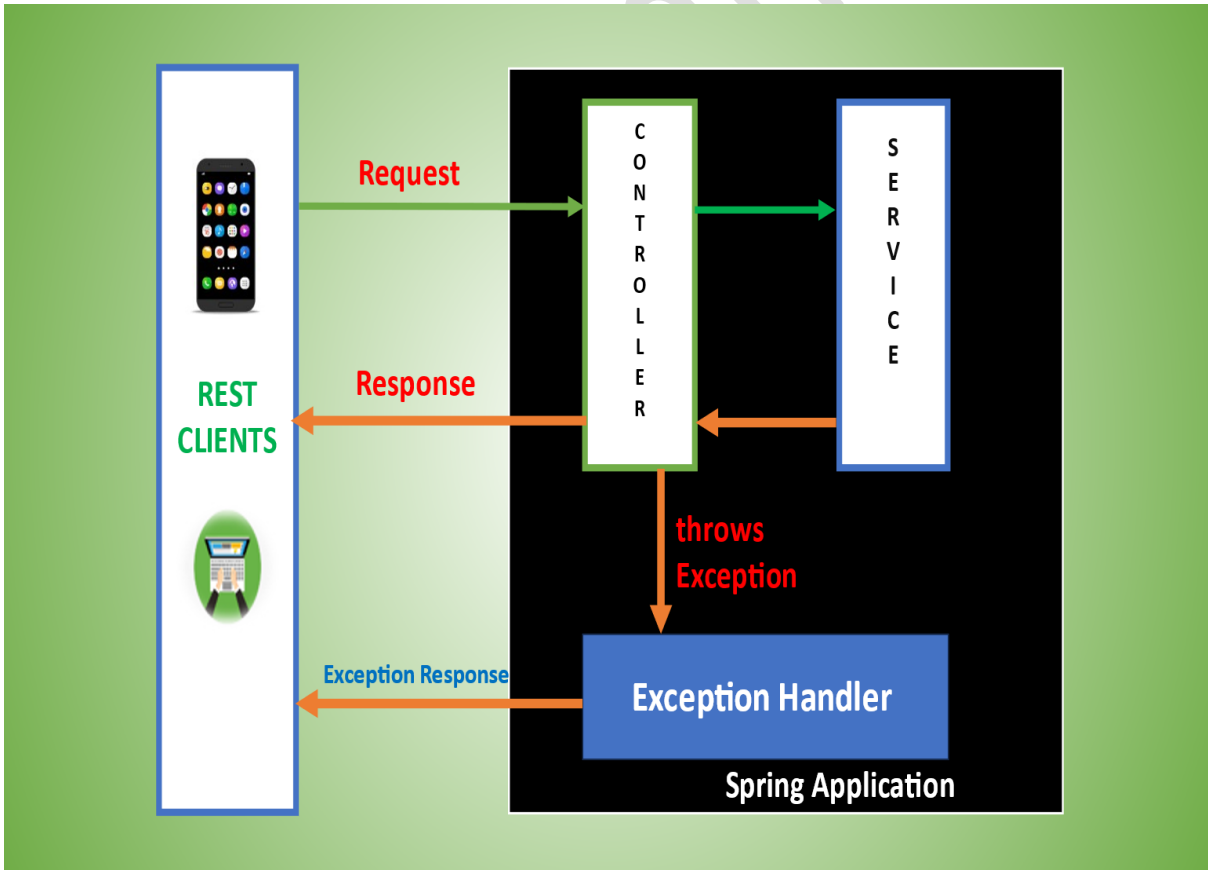


Note: Spring provided other ways as well to handle exceptions but controller advice and Exception handler will provide better way of exception handling.

@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 **ResposeEntity** response.

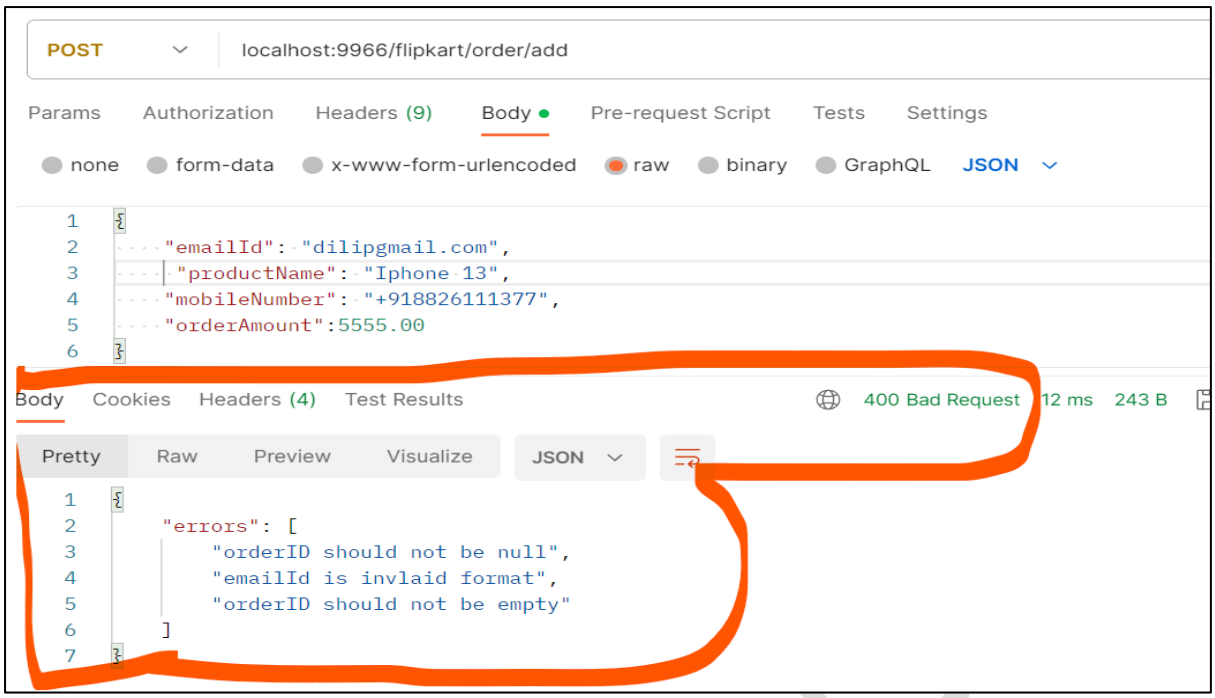
One thing to keep in our mind here is to match the exceptions declared with @ExceptionHandler to the exception used as the argument of the method.

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(ArithmeticException.class)
	public ResponseEntity<?> handleArithmeticException(ArithmeticException ex,
                                                        HttpServletRequest request) {

		return new ResponseEntity<>("Please Exception Details ",
                                                HttpStatus.INTERNAL_SERVER_ERROR);
	}

// Below Exception handler method will work for all child exceptions when we are not
//handled those specifically.
	@ExceptionHandler(Exception.class)
	public ResponseEntity<?> handleException(Exception ex, HttpServletRequest
                                                                request) {

	return new ResponseEntity<>("Please check Exception Details. ",
                                                HttpStatus.INTERNAL_SERVER_ERROR);
	}
}
```

- **Now see How we are getting Error response with meaningful messages when Request Body validation failed instead of complete Exception stack trace.**

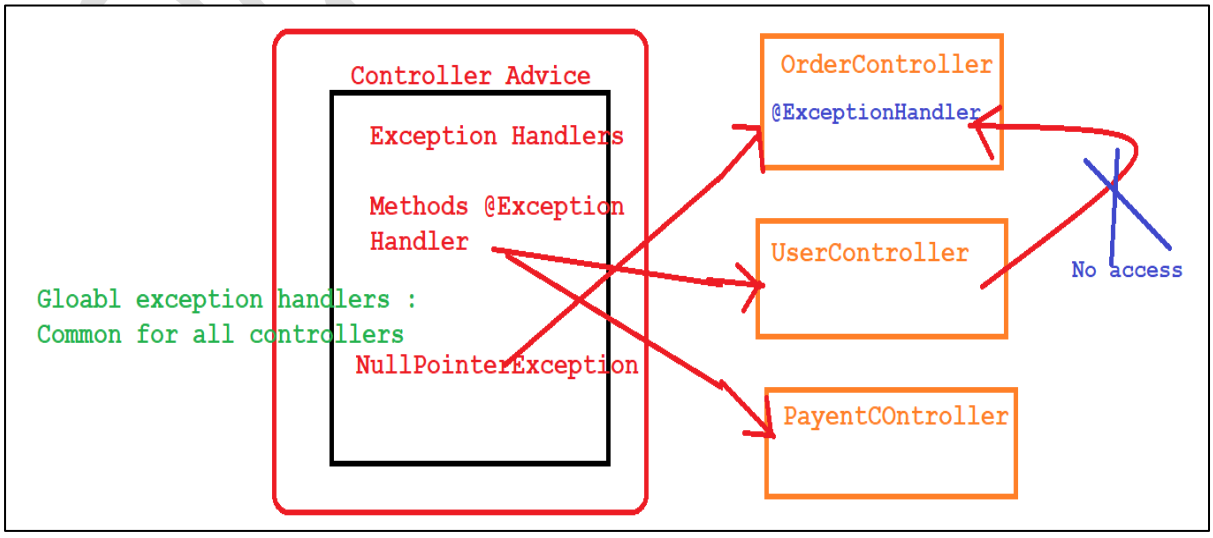


How it is working?

Whenever an exception occurred at controller layer due to any reason, immediately controller will check for relevant exceptions handled as part of Exception Handler or not. If handled, then that specific exception handler method will be executed and response will be forwarded to clients. If not handled, then entire exception error stack trace will be forwarded to client an it's not suggestable.

Which Exception takes Priority if we defined Child and Parent Exceptions Handlers?

From above example, if **NullPointerException** occurred then **handleNullpointerException()** method will be executed even though we have logic for parent **Exception** handling i.e. Priority given to child exception if we handled and that will be returned as response data. Similarly we can define multiple controller advice classes with different types of Exceptions along with relevant Http Response Status Codes.



Producing and Consuming REST API services:

Producing REST Services:

Producing REST services is nothing but creating Controller endpoint methods i.e. Defining REST Services on our own logic. As of Now we are created/produced multiple REST API Services with different examples by writing controller layer and URI mapping methods.

Consuming REST Services:

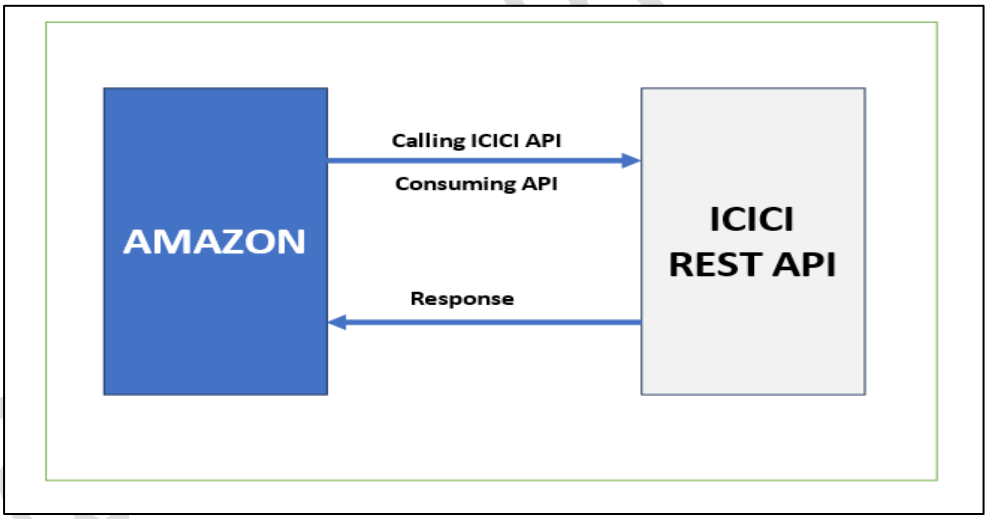
Consuming REST services is nothing but integrating other application REST API services from our application logic.

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

In This case:

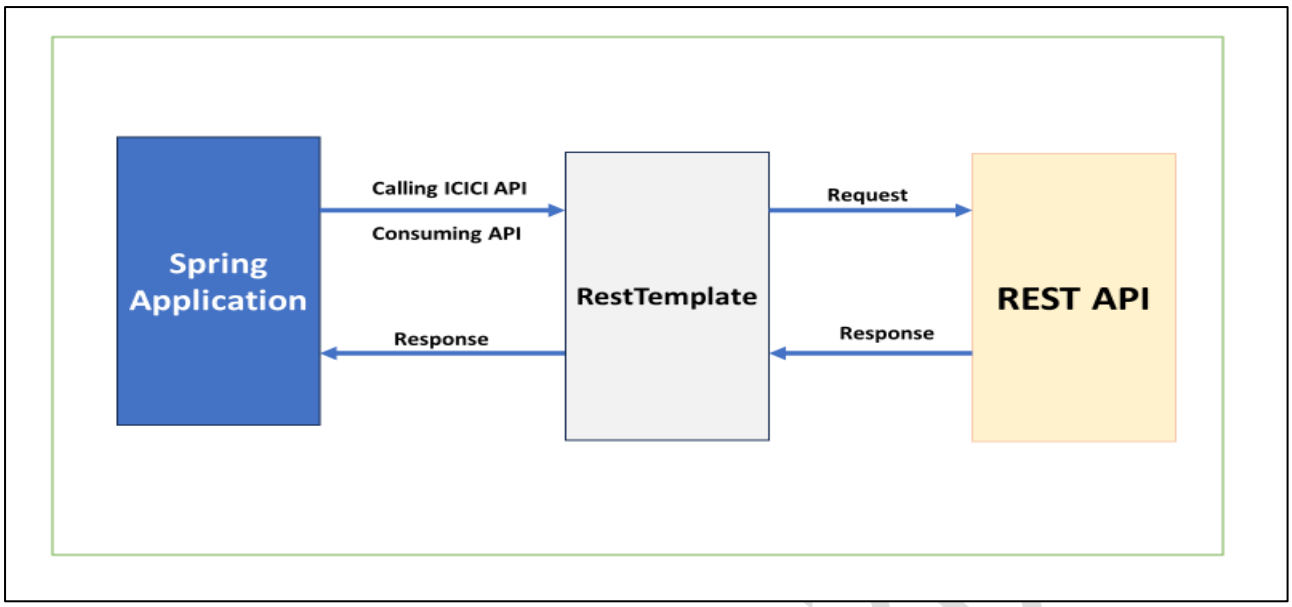
Producer is : ICICI

Consumer is : Amazon



In Spring MVC, Spring Provided an HTTP or REST client class called as **RestTemplate** from package **org.springframework.web.client. RestTemplate** class provided multiple utility methods to consume REST API services from one application to another application.

RestTemplate is 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.**

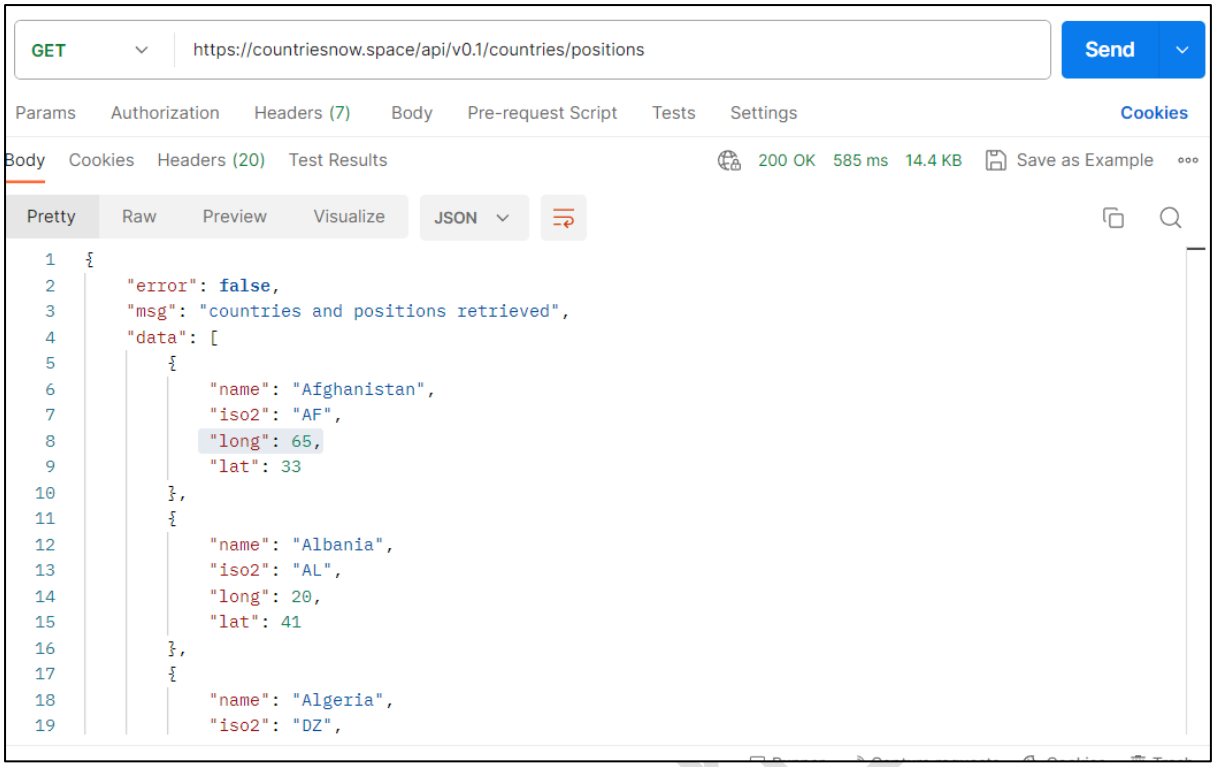
Now I am giving only API details of Pharmacy Application as swagger documentation. Because in Realtime Projects, swagger documentation or Postman collection data will be shared to Developers team, but not source code. So we will try to consume by seeing Swagger API documentation of tother application.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



Based on Response, we should create Response POJO classes aligned to JSON Payload.

Country.java

```
public class Country {
	private String name;
	private String iso2;
	private int lat;

	//Setters and Getters
}
```

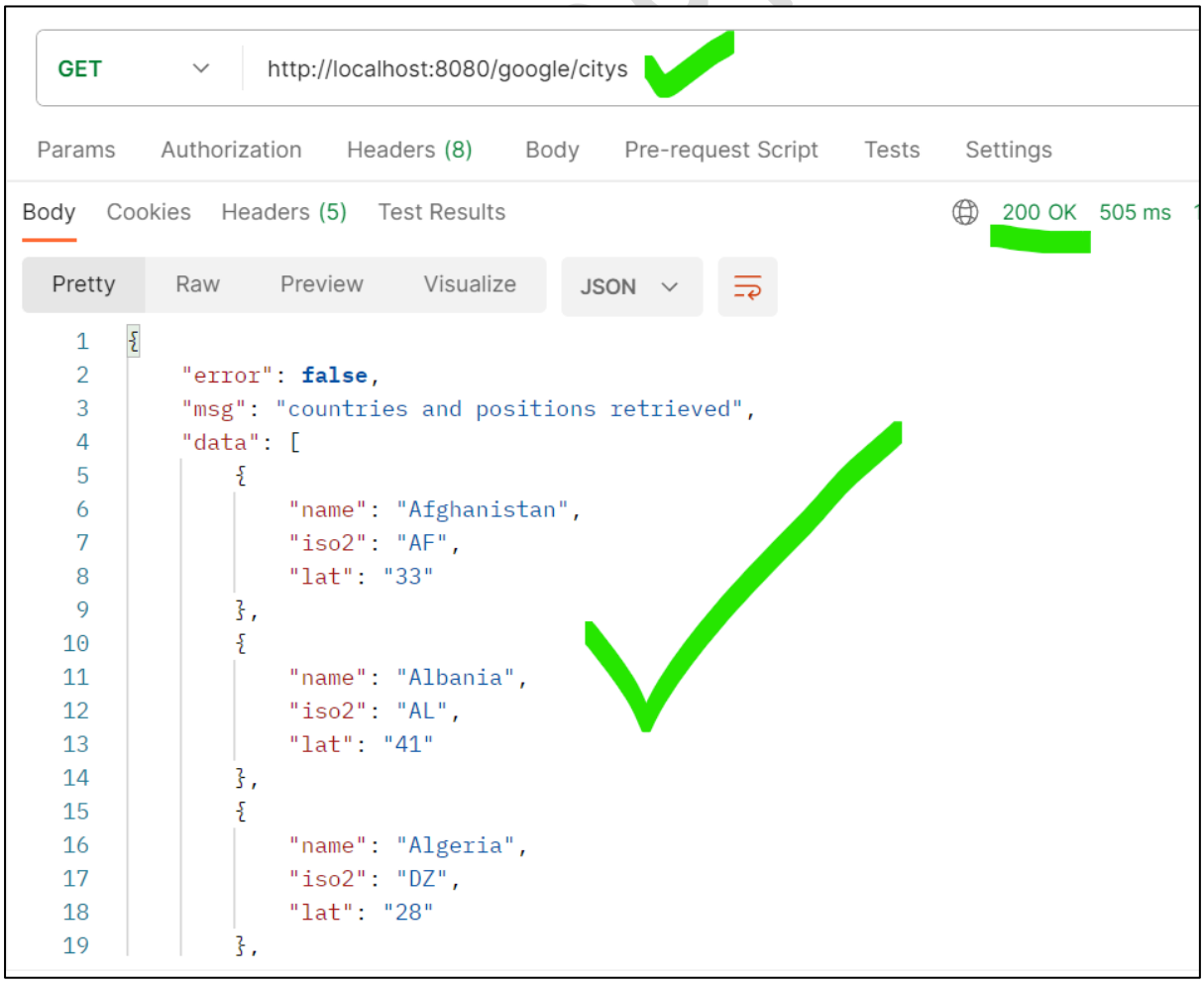
CountriesResponse.java

```
public class CountriesResponse {

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

		//Setters and Getters

}
```

Consuming Logic:

```
public CountriesResponse loadCities() {

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

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

        System.out.println(respone);

        return respone;
}
```

Testing from our Application:



XML Request and XML Response in REST API:

As of Now, By Default we are used JOSN 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 exiting 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:

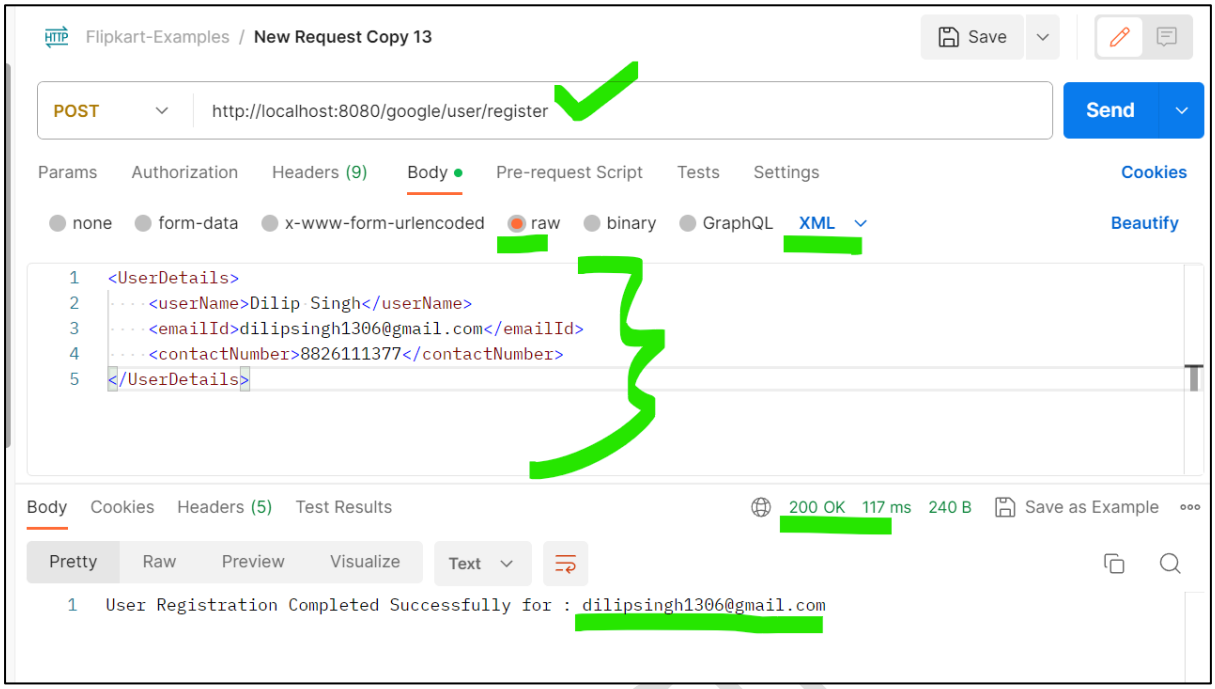
- **Inside Controller class: Create a New Service with Request Body:**

Below Service will accept either JSON or XML data formats as Request Payload. We have to send an Header **Content-Type** and it's associated values **application/xml** or **application/json** depends on out 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 give throw Exception with status code "**415 Unsupported Media Type**".

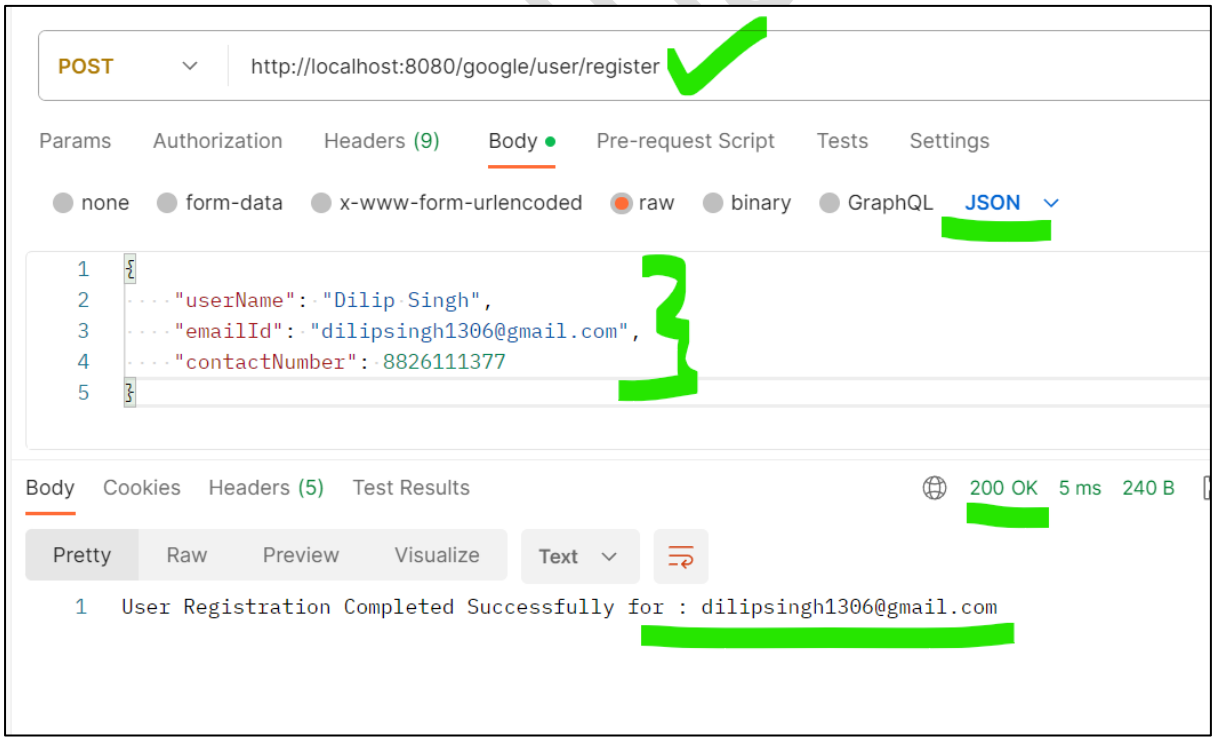
```
@PostMapping("/user/register")
public String RegisterUserInfo(@RequestBody UserDetails userDetails) {
	//Transfer Request Data to Service and Service to Repository
	System.out.println("Registering User Data for :"+userDetails.getEmailId());
	return "User Registration Completed Successfully for : "+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



JSON Payload: Postman : Please Select Data Type as JSON in Body-> raw -> JSON



Creating REST Service which Returns Response wither XML or JSON Data Format.

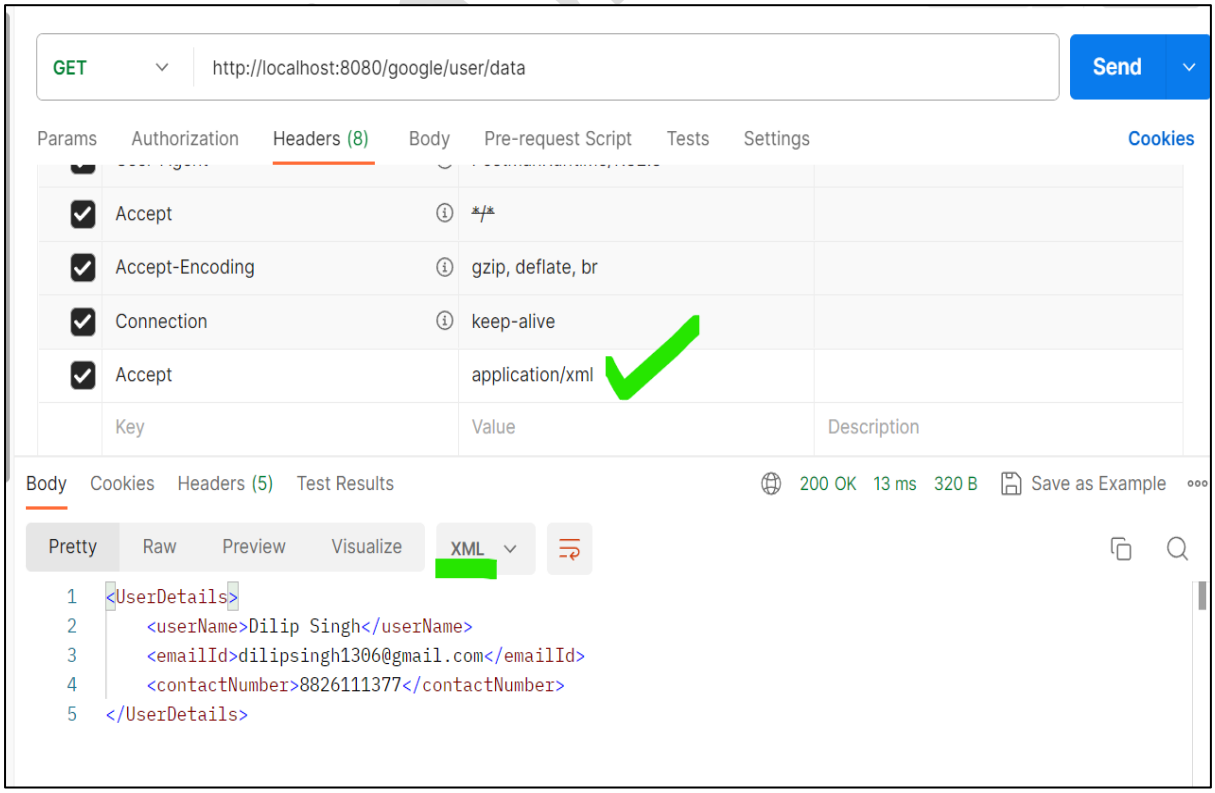
```
@GetMapping("/user/data")
public UserDetails getUserInfo() {
        //Assume Data Received from Service Layer
        UserDetails userData = new UserDetails();
        userData.setUserName("Dilip Singh");
        userData.setEmailId("dilipsingh1306@gmail.com");
        userData.setContactNumber(8826111377l);
        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:



JSON Response: