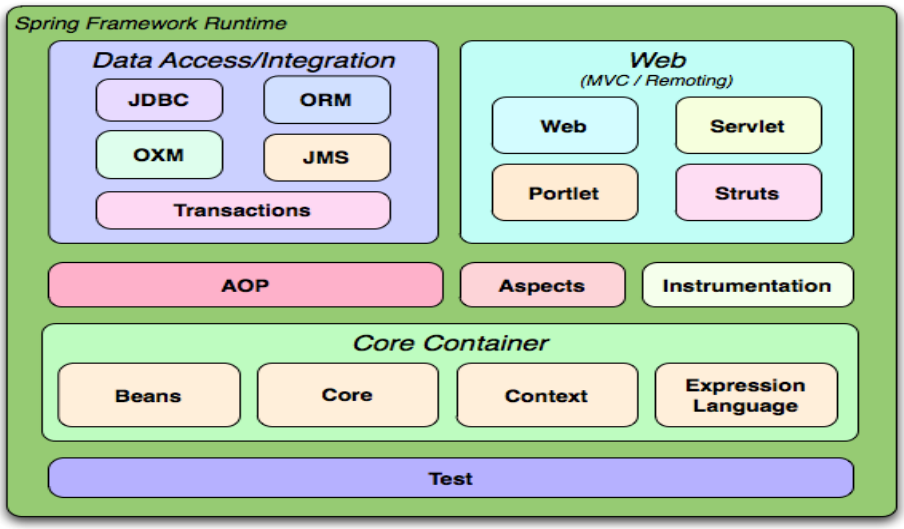
Spring / Spring Boot Web/MVC Modules

Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning. The formal name, "Spring Web MVC," comes from the name of its source module (spring-webmvc), but it is more commonly known as "Spring MVC". A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

A Spring MVC provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet. Here, DispatcherServlet is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the SpringBoot-starter-web module to get up and running quickly.



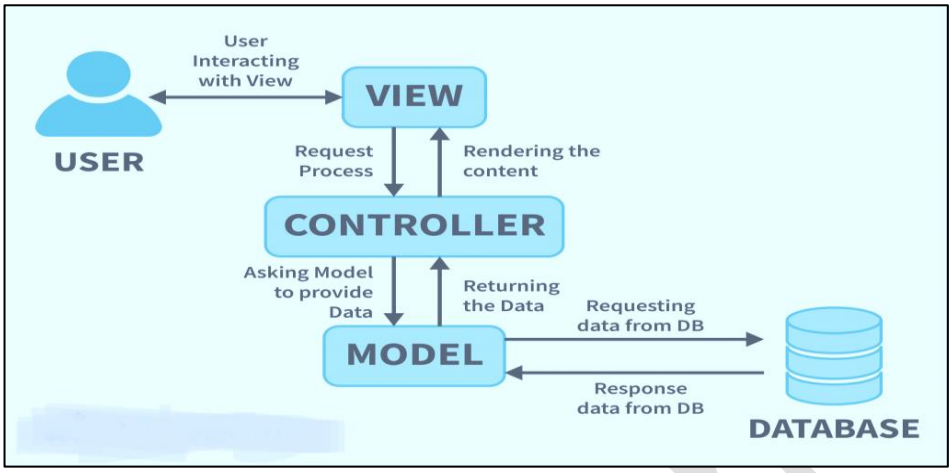
What is MVC?

MVC stands for Model-View-Controller, and it is a widely used architectural pattern in software development, particularly for building user interfaces and web applications. MVC is designed to separate an application into three interconnected components, each with a specific responsibility:

MVC Architecture becomes so popular that now most of the popular frameworks follow the MVC design pattern to develop the applications. Some of the popular Frameworks that follow the MVC Design pattern are:

- **JAVA Frameworks: Sprint, Spring Boot.**
- **Python Framework: Django.**
- **NodeJS (JavaScript): ExpressJS.**
- **PHP Framework: Cake PHP, Phalcon, PHPixie.**
- **Ruby: Ruby on Rails.**

- **Microsoft.NET: ASP.net MVC.**



Model:

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it 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.

Here's how the MVC pattern works in a typical scenario:

1. A user interacts with the View (e.g., clicks a button or submits a form).
2. The View forwards the user input to the Controller.
3. The Controller processes the input, potentially querying or updating the Model.
4. The Model is updated if necessary, and the Controller retrieves data from the Model.
5. The Controller sends the updated data to the View.
6. The View renders the data and presents it to the user.

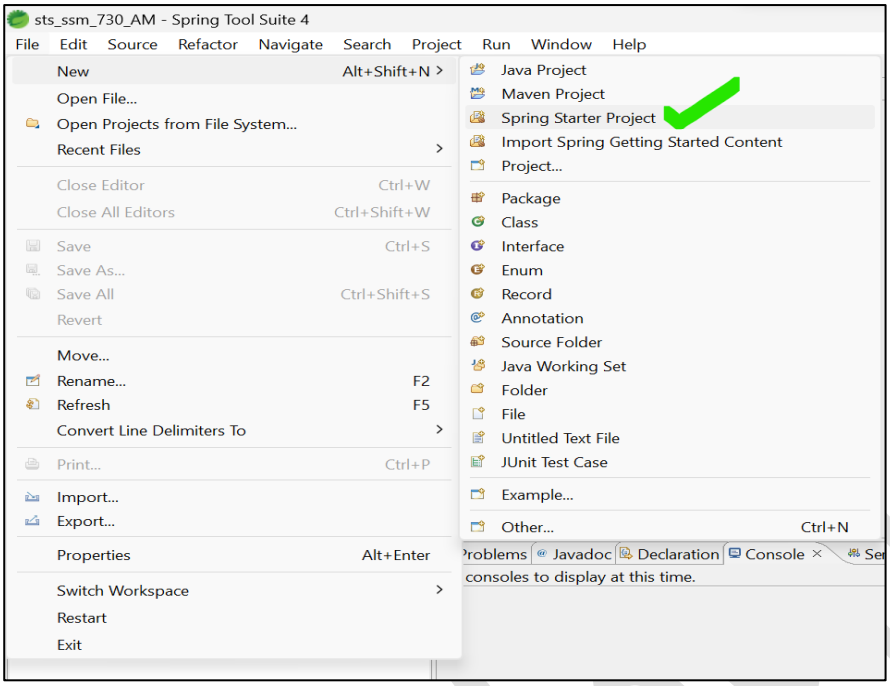
Advantages of Spring MVC Framework

- **Separate roles** - The Spring MVC separates each role, where the model object, controller, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
- **Light-weight** - It uses light-weight servlet container to develop and deploy your application.
- **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
- **Rapid development** - The Spring MVC facilitates fast and parallel development.
- **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
- **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
- **Flexible Mapping** - It provides the specific annotations that easily redirect the page.

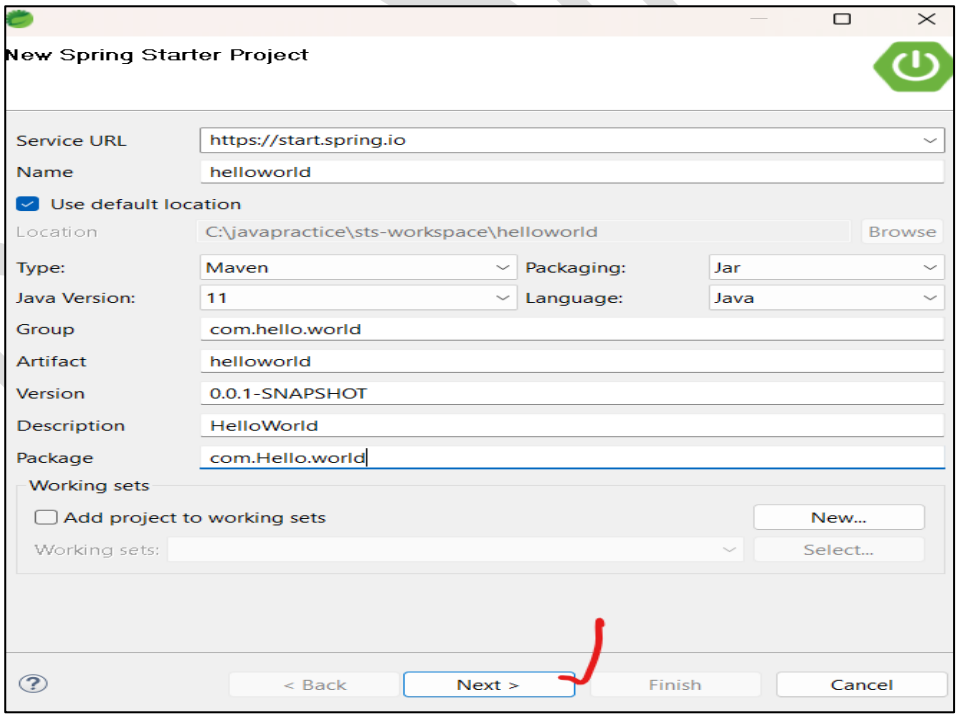
MVC is a foundational pattern used in various software development paradigms, including desktop applications, web applications, and mobile applications. Different platforms and frameworks may implement MVC in slightly different ways, such as Model-View-Presenter (MVP) or Model-View-View-Model (MVVM), but the core principles of separation of concerns and data flow remain consistent.

Creating SpringBoot Web Application:

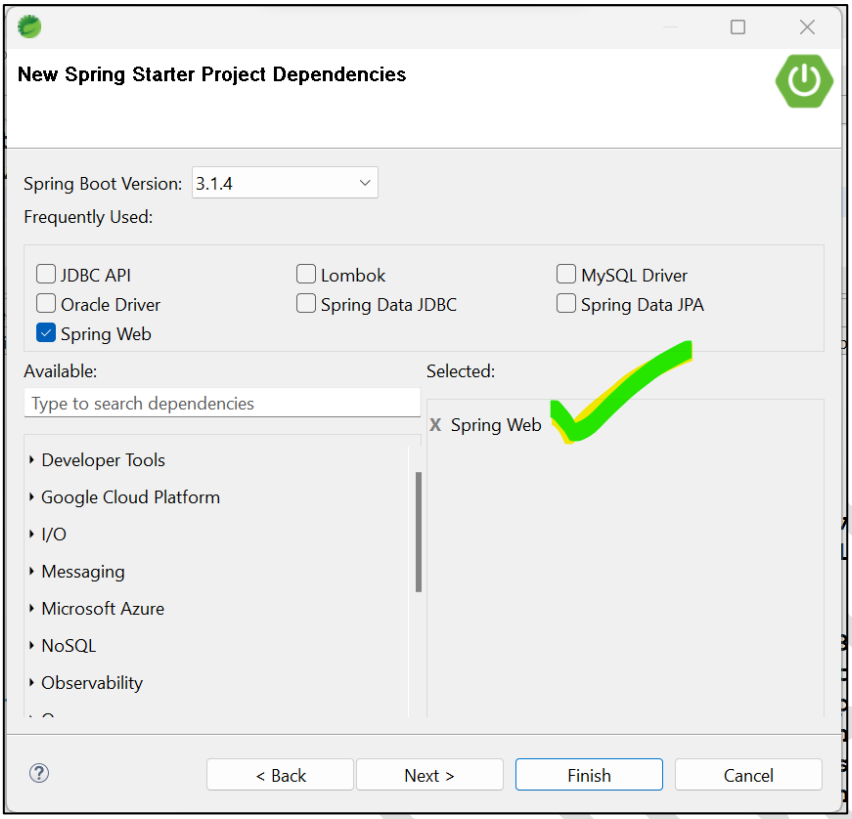
- **Open STS : File-> New > Spring Starter Project**



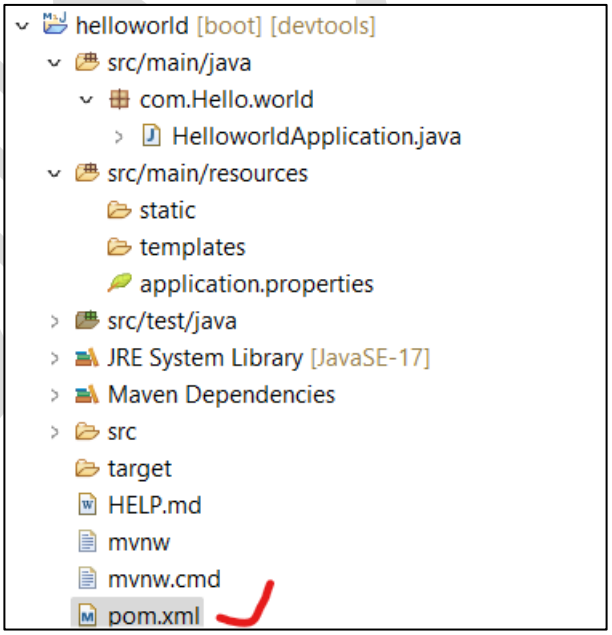
- **Fill All Project details as shown below and click on Next.**



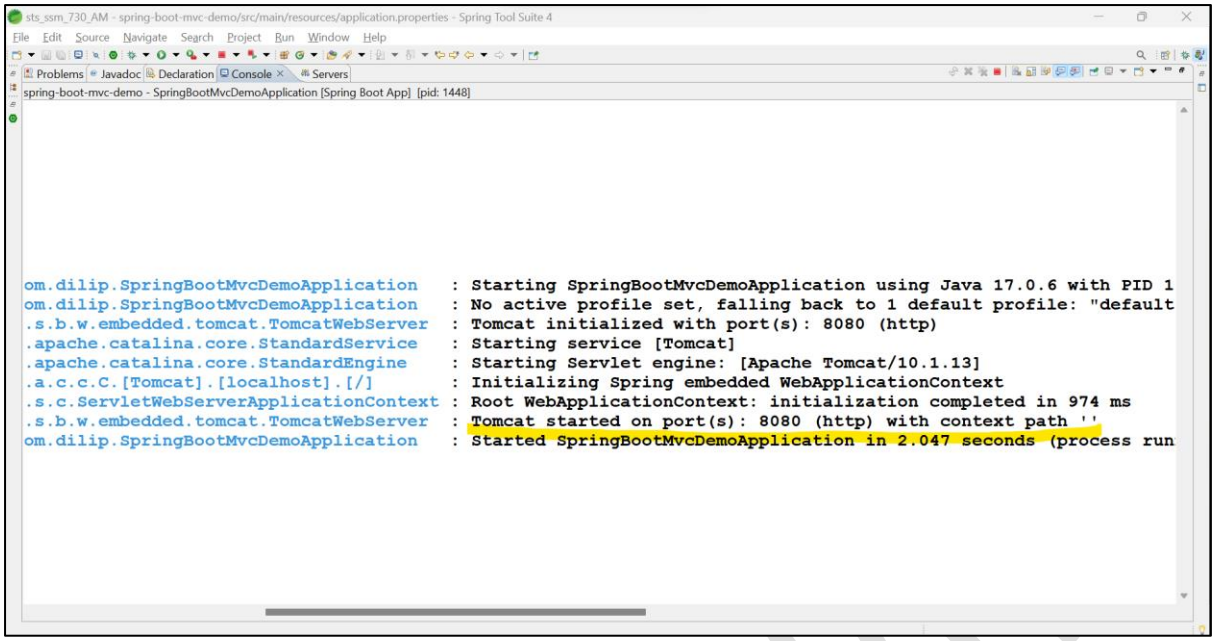
- **In Next Page, Add Spring Boot Modules/Starters as shown below and click on finish.**



- **After finish the project look like this all your dependencies in pom.xml file**



- Now Run your Application as **Spring Boot App / java application** from Main Method Class.



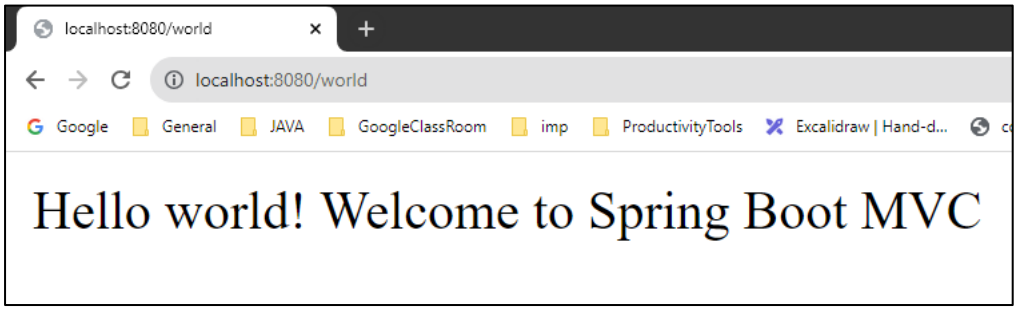
- Integrated Server Started with Default Port : **8080** with context path ' '. i.e., if we won't give any port number then default port number will be **8080**. If we want to change default port number then, we should add a property and its value in **application.properties**.
- By Default Spring Boot application will be deployed with empty context path ' '. If we want to change default context path then, we should add a property and its value in **application.properties**.

So our application base URL will be always : http://localhost:8080/
Requirement: Now Let me add an Endpoint/URL to print Hello World Message.

Controller Class:

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

@Controller
public class HelloWorldController {
	@GetMapping("/world")
	@ResponseBody
	public String printHelloWrold() {
		return "Hello world! Welcome to Spring Boot MVC";
	}
}
```

Execute/Call above Endpoint: We will access Endpoints/URLs from Http Clients like Browsers.



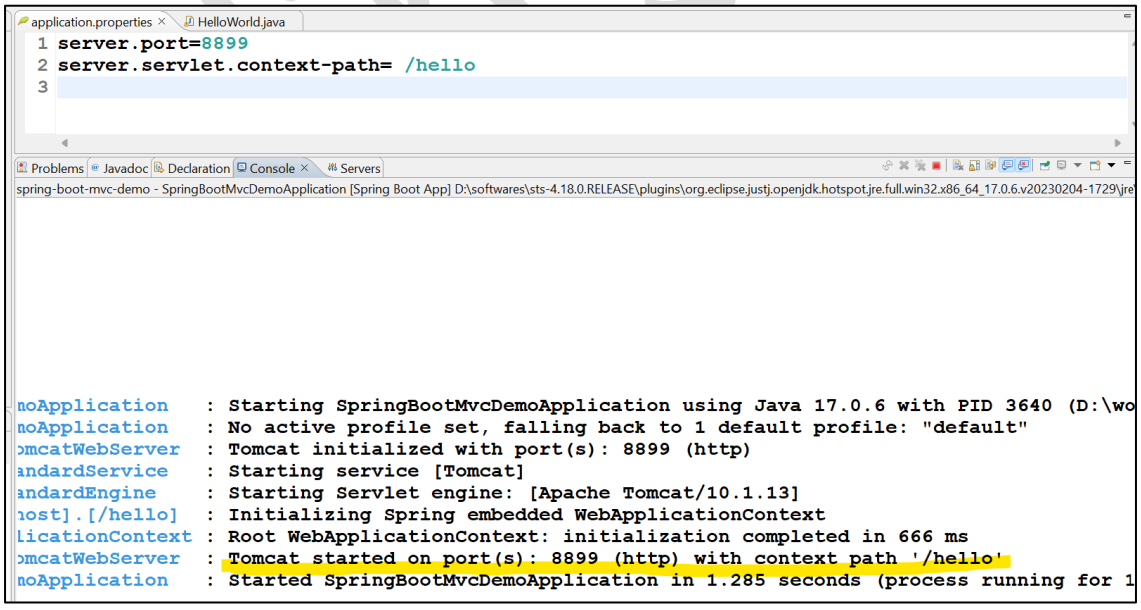
Changing Default Port Number and Context path of Application:

Now open **application. Properties** file and add below predefined properties and provide required values.

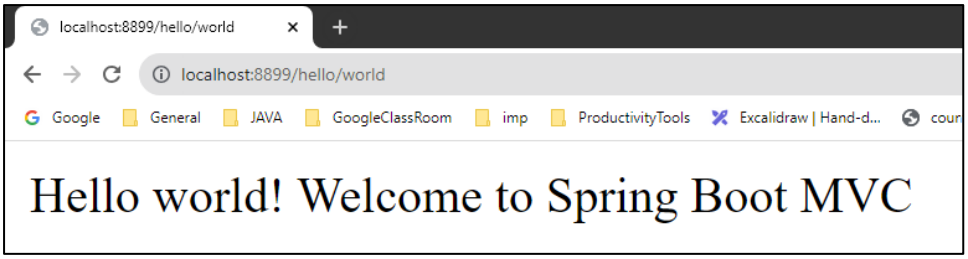
```
server.port=8899
server.servlet.context-path= /hello
```

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

So our application base URL will become now always : http://localhost:8899/hello

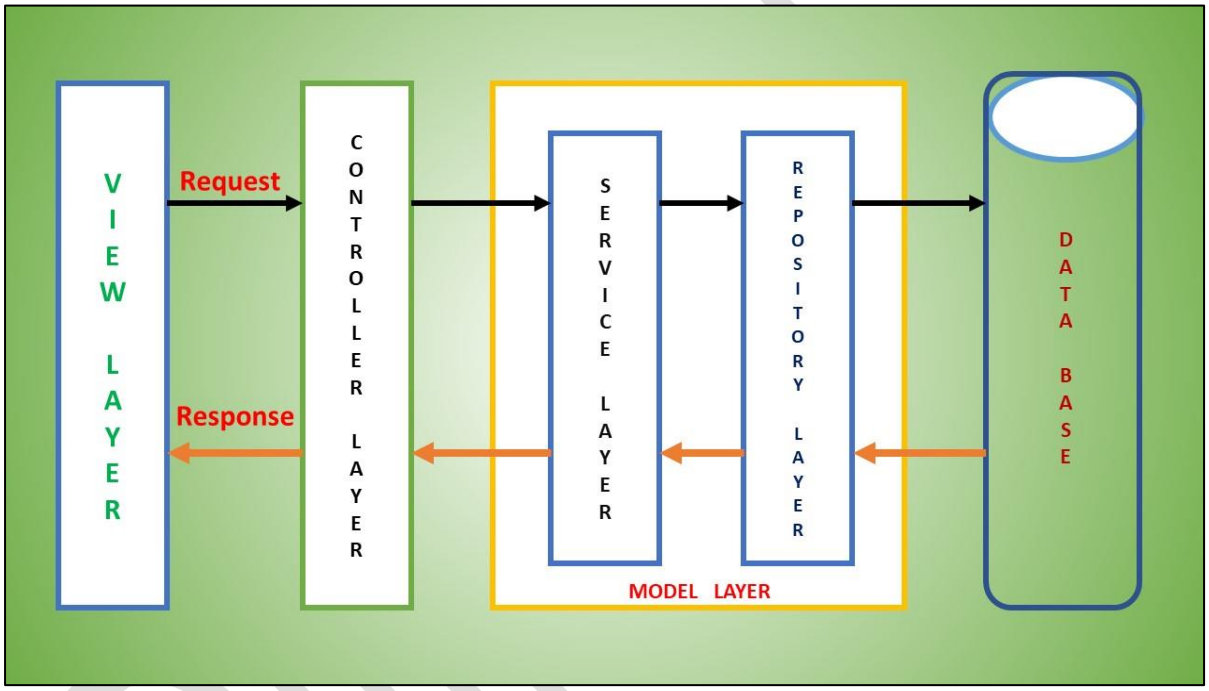


Output: **Call "/world" endpoint :** http://localhost:8899/hello/world



Spring MVC Application Internal Workflow:

Usually, Spring MVC Application follows below architecture on high level.



Internal Workflow of Spring MVC Application i.e., Request & Response Handling:

The Spring Web **Model-View-Controller** (MVC) framework is designed around a **Front Controller Design Pattern** i.e. **DispatcherServlet** that handles all the HTTP requests and responses across Spring Web application. The request and response processing workflow of the Spring Web MVC **DispatcherServlet** is illustrated in the following diagram.

Front Controller:

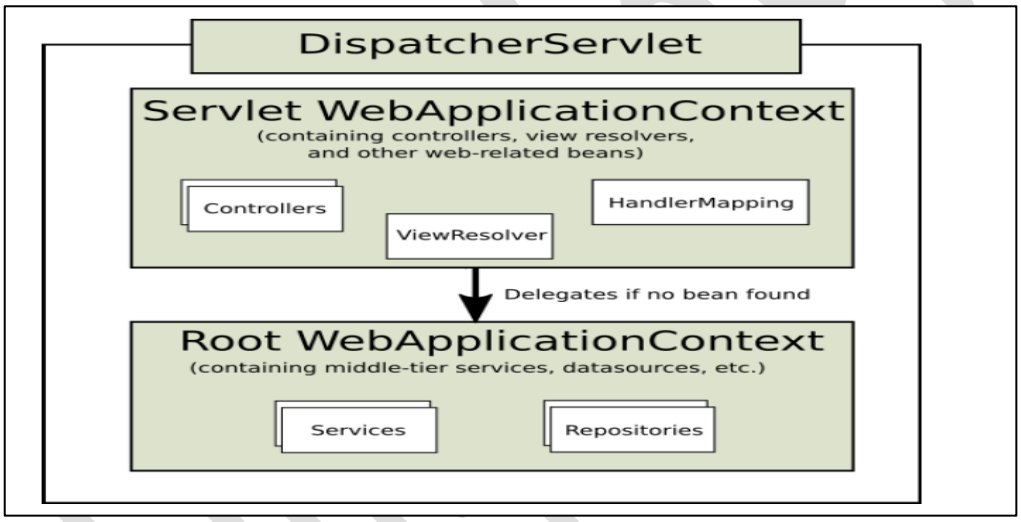
A **front controller** is defined as a controller that handles all requests for a Web Application. **DispatcherServlet** servlet is the front controller in Spring MVC that intercepts every request and then dispatches requests to an appropriate controller. The **DispatcherServlet** is a Front Controller and one of the most significant components of the Spring MVC web framework. A Front Controller is a typical structure in web applications that receives requests and delegates their processing to other components in the application. The **DispatcherServlet** acts as a single entry point for client requests to the Spring MVC web application, forwarding them to the appropriate Spring MVC controllers for processing.

DispatcherServlet is a front controller that also helps with view resolution, error handling, locale resolution, theme resolution, and other things.

Request: The first step in the MVC flow is when a request is received by the Dispatcher Servlet. The aim of the request is to access a resource on the server.

Response: Response is made by a server to a client. The aim of the response is to provide the client with the resource it requested, or inform the client that the action it requested has been carried out; or else to inform the client that an error occurred in processing its request.

Dispatcher Servlet: Now, the **DispatcherServlet** with the help of Handler Mappings understands the Controller class name associated with the received request. Once the **DispatcherServlet** knows which Controller will be able to handle the request, it will transfer the request to it. **DispatcherServlet** expects a **WebApplicationContext** (an extension of a plain **ApplicationContext**) for its own configuration. **WebApplicationContext** has a link to the **ServletContext** and the Servlet with which it is associated.



The **DispatcherServlet** delegates to special beans to process requests and render the appropriate responses.

All the above-mentioned components, i.e. **HandlerMapping**, Controller, and **ViewResolver** are parts of **WebApplicationContext** which is an extension of the plain **ApplicationContext** with some extra features necessary for web applications.

HandlerMapping:

In Spring MVC, the **DispatcherServlet** acts as front controller – receiving all incoming HTTP requests and processing them. Simply put, the processing occurs by passing the requests to the relevant component with the help of handler mappings.

HandlerMapping is an interface that defines a mapping between requests and handler objects. The HandlerMapping component parses a Request and finds a Handler that handles the Request, which is generally understood as a method in the Controller.

Now Define Controller classes inside our Spring Boot MVC application:

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

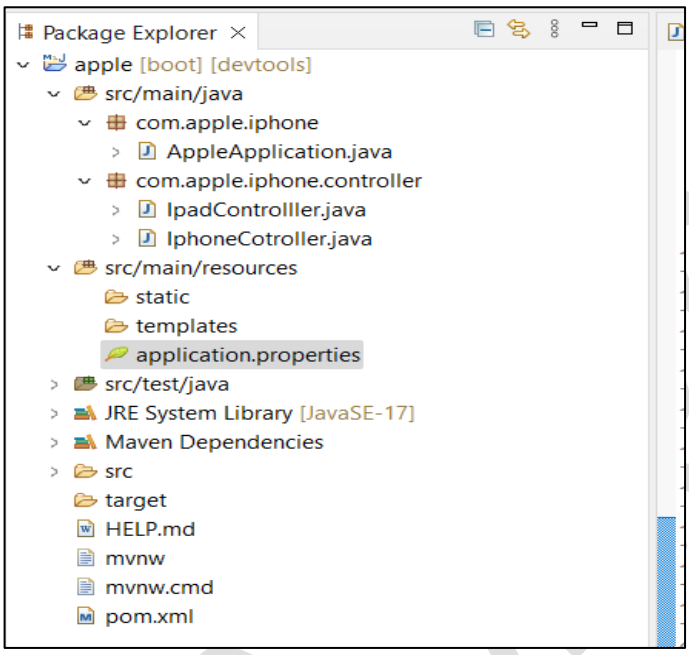
```
package com.apple.iphone.controller;

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 Iphone 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";
        }
}
```

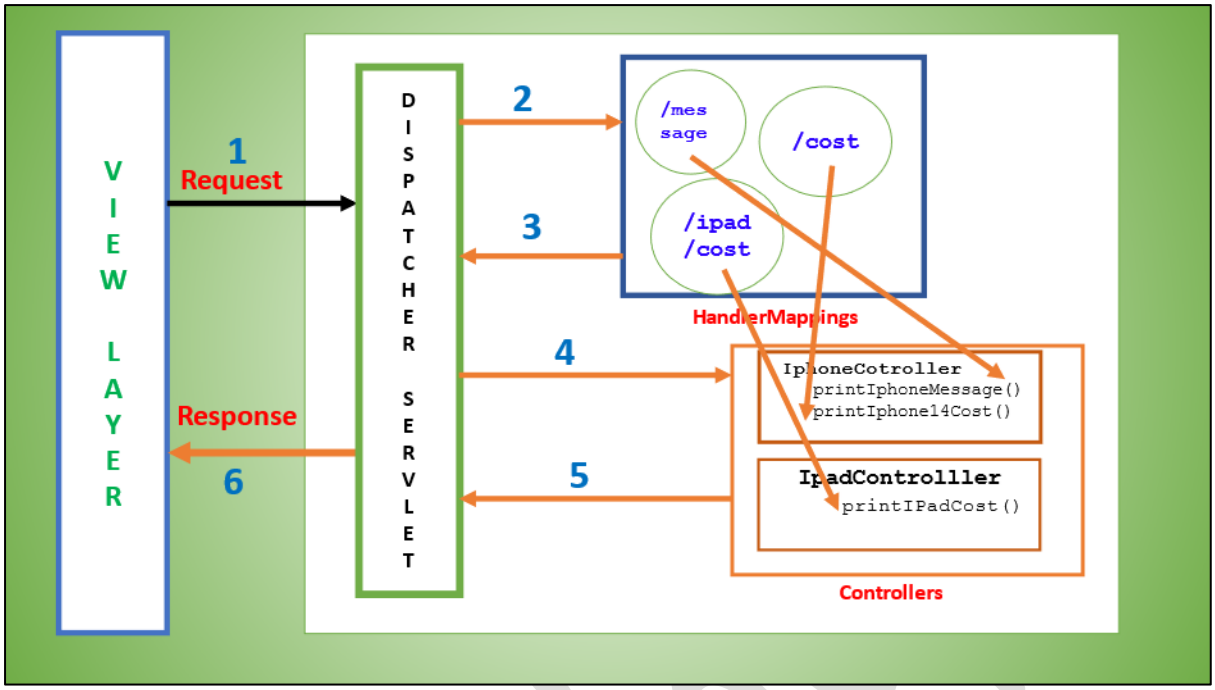
Project Folder and File Structure:



Now when we start our project as Spring Boot Application, Internally Project deployed to tomcat server and below steps will be executed.

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

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



After Starting our Spring Boot Application, when we are sending a request, Following is the sequence of events happens corresponding to an incoming HTTP request to **DispatcherServlet**:

For example, we sent a request to our endpoint from browser:

http://localhost:6655/apple/message

- After receiving an HTTP request, **DispatcherServlet** consults the **HandlerMappings** by passing URI (**/message**) and HTTP Method type GET.
- Then **HandlerMappings** will checks all mappings information with above Details, If details are mapped then **HandlerMapping** will returns Controller Class Name and Method Name.
- If Details are not mapped/found in mappings, then **HandlerMappings** will provide an error message to **DispatcherServlet** with Error Details.
- After **DispatcherServlet** Receiving appropriate Controller and its associated method of endpoint URI, then **DispatcherServlet** forwards all request body and parameters to controller method and executes same.
- The Controller takes the request from **DispatcherServlet** and calls the appropriate service methods.
- The service method will set model data based on defined business logic and returns result or response data to Controller and from Controller to **DispatcherServlet**.
- If We configured **ViewResolver**, The **DispatcherServlet** will take help from **ViewResolver** to pick up the defined view i.e. JSP files to render response of for that specific request.
- Once view is finalized, The **DispatcherServlet** passes the model data to the view which is finally rendered on the browser.
- **If no ViewResolver configured** then Server will render the response on Browser or ANY Http Client as default test/JSON format response.

NOTE: As per REST API/Services, we are not integrating Frontend/View layer with our controller layer i.e. We are implementing individual backend services and shared with Frontend Development team to integrate with Our services. Same Services we can also share with multiple third party applications to interact with our services to accomplish the task. So We are continuing our training with REST services implantation point of view because in Microservices Architecture communication between multiple services happens via REST APIS integration across multiple Services.

Controller Class:

In Spring Boot, the controller class is responsible for processing incoming HTTP web requests, preparing a model, and returning the view to be rendered as a response on client. The controller classes in Spring are annotated either by the **@Controller** or the **@RestController** annotation.

@Controller: org.springframework.stereotype.Controller

The **@Controller** annotation is a specialization of the generic stereotype **@Component** annotation, which allows a class to be recognized as a Spring-managed component. **@Controller** annotation indicates that the annotated class is a controller. It is a specialization of **@Component** and is autodetected through class path/component scanning. It is typically used in combination with annotated handler methods based on the **@RequestMapping** annotation.

@ResponseBody:

Package: org.springframework.web.bind.annotation.ResponseBody:

We annotated the request handling method with ***@ResponseBody***. This annotation enables automatic serialization of the return object into the ***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:

Package: org.springframework.web.bind.annotation.RequestMapping

This Annotation for mapping web requests onto methods in request-handling classes i.e. controller classes with flexible method signatures. **@RequestMapping** is Spring MVC's most common and widely used annotation.

This Annotation has the following optional attributes.

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

consumes	String[]	media types that can be consumed by the mapped handler. Consists of one or more media types one of which must match to the request `Content-Type` header.
		
		`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: **@RequestMapping** without any attributes with method level

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

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

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

@RequestMapping("/message"):

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

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

Example : With method attribute and one value:

```
@RequestMapping(value="/message", method = RequestMethod.GET)
@ResponseBody
public String printIphoneMessage() {
        return " Welcome to Iphone World.";
}
```

Now above handler method will work only for HTTP **GET** request call. If we try to request with any HTTP methods other than **GET**, we will get error response as

```
"status": 405,
"error": "Method Not Allowed",
```

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Oct 03 17:35:31 IST 2023
There was an unexpected error (type=Method Not Allowed, status=405).

Example : method attribute having multiple values i.e. Single Handler method

```
@RequestMapping(value="/message",
                        method = {RequestMethod.GET, RequestMethod.POST})
@ResponseBody
public String printIphoneMessage() {
        return " Welcome to Iphone World.";
}
```

Now above handler method will work only for HTTP **GET** and **POST** requests calls. If we try to request with any HTTP methods other than **GET**, **POST** we will get error response as:

```
"status": 405,
"error": "Method Not Allowed",
```

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Oct 03 17:35:31 IST 2023
There was an unexpected error (type=Method Not Allowed, status=405).

i.e. we can configure one URL handler method with multiple HTTP methods request.

Example : With Multiple URI values and method values:

```
@RequestMapping(value = { "/message", "/msg/iphone" },
                        method = { RequestMethod.GET, RequestMethod.POST })
@ResponseBody
public String printIphoneMessage() {
        return " Welcome to Iphone World.";
}
```

Above handler method will support both **GET** and **POST** requests of URI's mappings **"/message", "/msg/iphone".**

RequestMethod:

RequestMethod is Enumeration(Enum) of HTTP request methods. Intended for use with the **RequestMapping.method()** attribute of the **RequestMapping** annotation.

ENUM Constant Values : GET, POST, PUT, DELETE, HEAD, OPTIONS, PATCH, TRACE

Example : multiple Handler methods with same URI and different HTTP methods.

We can Define Same URI with multiple different handler/controller methods for different HTTP methods. Depends on incoming HTTP method request type specific handler method will be executed.

```
@RequestMapping(value = "/mac", method = RequestMethod.GET)
@ResponseBody
public String printMacMessage() {
        return " Welcome to MAC World.";
}

@RequestMapping(value = "/mac", method = RequestMethod.POST)
@ResponseBody
public String printMac2Message() {
        return " Welcome to MAC 2 World.";
}
```

Declaring @RequestMapping at Class Level:

We can use **@RequestMapping** with class definition level to create the base URI of that specific controller i.e. All URI mappings of that controller will be preceded with class level URI value always.

For example:

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

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

@Controller
@RequestMapping("/ipad")
public class 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 URI's as shown below.

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

@GetMapping:

Package: org.springframework.web.bind.annotation.GetMapping

This Annotation used for mapping HTTP **GET** requests onto specific handler methods. The **@GetMapping** annotation is a composed version of **@RequestMapping** annotation that acts as a shortcut for **@RequestMapping(method = RequestMethod.GET)**.

The **@GetMapping** annotated methods handle the HTTP GET requests matched with the given URI value.

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

@PostMapping:

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

@PutMapping:

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

@DeleteMapping:

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

View Layer / JSP files Integration in Spring Boot MVC:

- Create A Spring Boot Web Application

- By default embedded tomcat server will not support JSP functionalities inside a Spring Boot MVC application. So, In order to work with JSP, we need to add below dependency in Spring boot MVC.

```
<dependency>
   <groupId>org.apache.tomcat.embed</groupId>
   <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

JSP ViewResolver Configuration in application.properties file:

What is ViewResolver?

In Spring MVC, a **ViewResolver** is an essential component responsible for resolving logical view names returned by controller methods into actual view implementations that can be rendered and returned to the client. It plays a crucial role in the web application's flow by mapping logical view names to views, which can be JSP pages, HTML templates, or any other type of view technology supported by Spring.

InternalResourceViewResolver:

InternalResourceViewResolver is a class in the Spring Framework used for view resolution in a Spring web application. It's typically used when you are working with Java Server Pages (JSP) as your view technology. **InternalResourceViewResolver** helps map logical view names returned by controllers to physical JSP files within your application.

Here's how **InternalResourceViewResolver** works:

1. **Controller Returns a Logical View Name:** In a Spring web application, when a controller method processes an HTTP request and returns a logical view name (e.g., "home" or "dashboard"), this logical view name is returned to the Spring MVC framework.

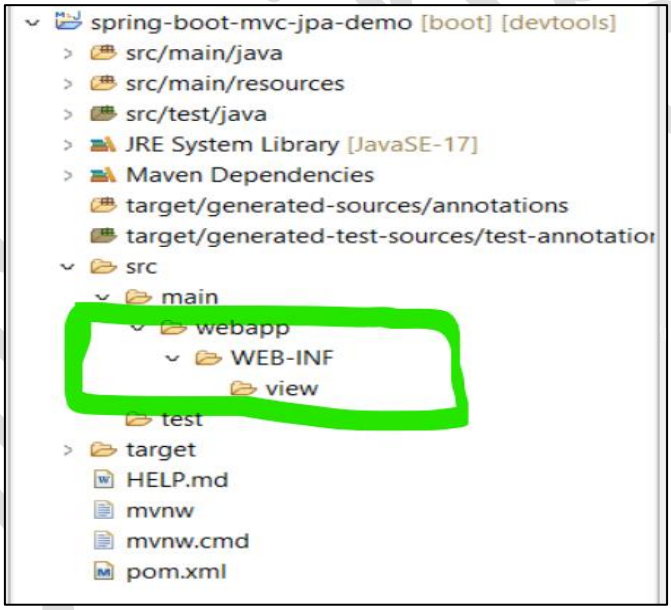
2. **InternalResourceViewResolver Resolves the View:** The **InternalResourceViewResolver** is configured in the Spring application context, and it is responsible for resolving logical view names. It combines a prefix and suffix with the logical view name to construct the actual path to the JSP file. For example, if the prefix is **"/WEB-INF/view/"** and the suffix is **".jsp,"** and the logical view name is "**home**," then the resolver constructs the view path as **"/WEB-INF/view/home.jsp**"

3. **JSP Is Rendered:** Once the view path is resolved, the JSP file at that path is executed. Any dynamic data is processed, and the JSP generates HTML content that is sent as a response to the client's browser.

- Add below View resolver properties in **application.properties** file to configure view names i.e. JSP files.

```
spring.mvc.view.prefix=/WEB-INF/view/
spring.mvc.view.suffix=.jsp
```

- Based on above configuration of property **prefix** value, we have to create folders inside our Spring Boot MVC application.
- Create a folder **webapp** inside **src -> main**
- Inside **webapp**, create another folder **WEB-INF**
- Inside **WEB-INF**, create another folder **view**
- Inside **view** Folder, We will create our JSP files.



- Now create a JSP file inside view folder and invoke it from Controller Method.

Create JSP file : **hello.jsp**

```
<html>
<head>
<title>Spring Boot MVC</title>
</head>
<body>
        <h1>Welcome to Spring Boot MVC with JSP</h1>
</body>
```

```
</html>
```

- Now create a controller method and invoke above jsp file.

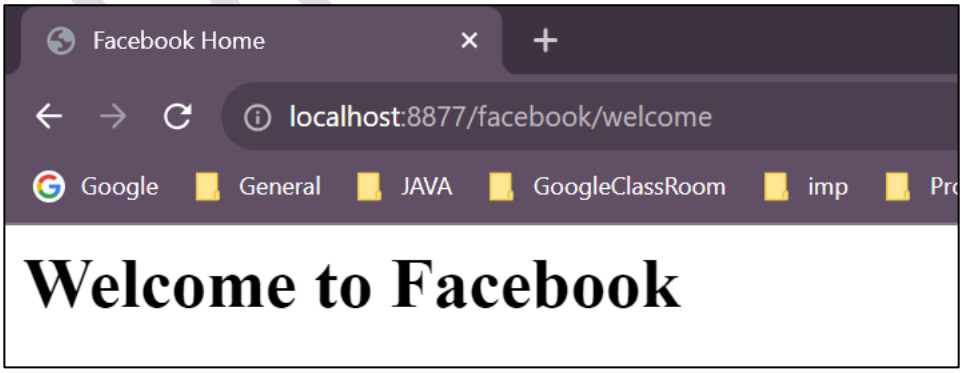
```
package com.facebook.controller;

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

@Controller
public class UserController {
	@GetMapping("/welcome")
	public String sayHello() {
		return "hello";
	}
}
```

Testing: **Send a Request to above URI method:**

- Internally, **DispatcherServlet** will forwards the request to jsp file as per our Internal Resource View Resolver configuration data, i.e. inside folder **/WEB-INF/view/** with suffix **.jsp** by including jsp file name **"hello".**

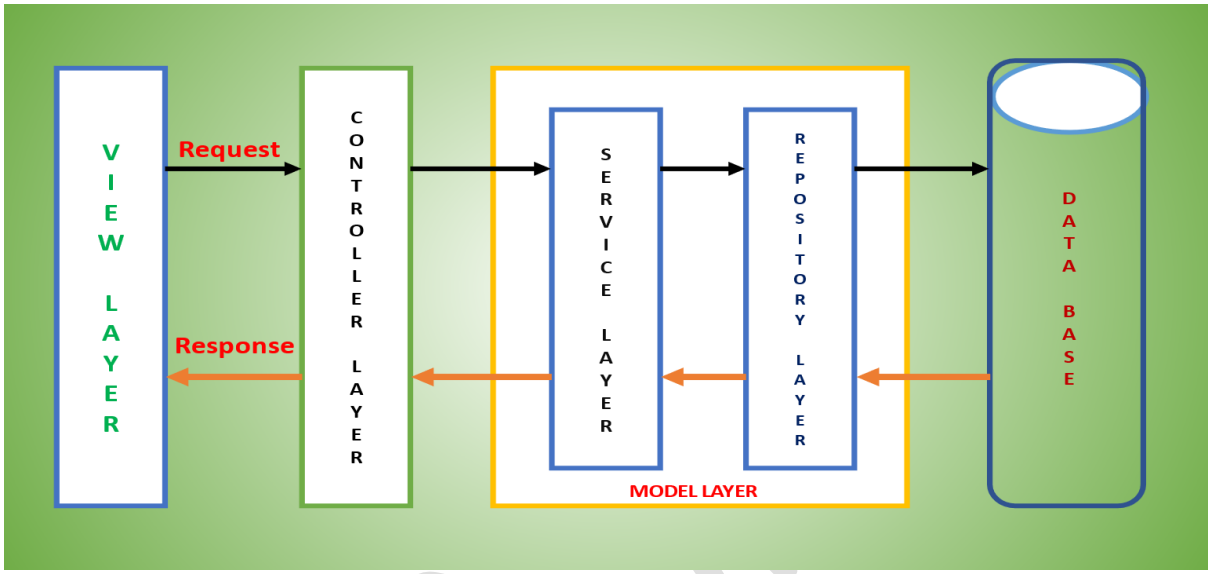
prefix + view name + suffix = /WEB-INF/view/hello.jsp

- Above JSP loaded as Response now in client/browser side.



Service Layer in Spring MVC:

A service layer is a layer in an application that facilitates communication between the controller and the persistence/repository layer. Additionally, business logic should be written inside service layer. It defines which functionalities you provide, how they are accessed, and what to pass and get in return. Even for simple CRUD cases, introduce a service layer, which at least translates from DTOs to Entities and vice versa. A Service Layer defines an application's boundary and its set of available operations from the perspective of interfacing client layers. It encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations.



Spring MVC application Architecture

We are going annotate with **@Service** is annotated on class to say spring, this is my Service Layer.

Create An Example with Service Layer:

Controller Class:

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

        //Logic

}
```

Service Class:

```
@Service
public class UniversityAdmissionsService {

        //Logic
```

```
}
```

Now integrate Service Layer class with Controller Layer i.e. injecting Service class Object into Controller class Object. So we will use **@Autowired** annotation to inject service in side controller.

```
@Controller
public class UniversityAdmissionsController {

        //Injecting Service Class Object : Dependency Injection
        @Autowired
        UniversityAdmissionsService service;

        //Logic
}
```

From above, We are integrated controller with service layer. Now inside Service class, we will write Business Logic and then data should be passed to persistence layer.

Now return values of service class methods are passed to Controller class level. This is how we are using service layer with controller layer. Now we should integrate Service layer with DAO Layer to Perform DB operations. We will have multiple examples together of all three layer.

Repository Layer:

Repository Layer is mainly used for managing the data with database in a Spring Application. A huge amount of code is required for working with the databases, which can be easily reduced by Spring Data modules. It consists of JPA and JDBC modules. There are many Spring applications that use JPA technology, so these development procedures can be easily simplified by Spring Data JPA. As we discussed earlier in JPA functionalities, Now we have to integrate JPA Module to our existing application.

Repository Interface:

```
@Respository
public interface AdmissionsRepository extends JpaRepository {
                //JPA Methods
}
```

Repository Integration with Service Class:

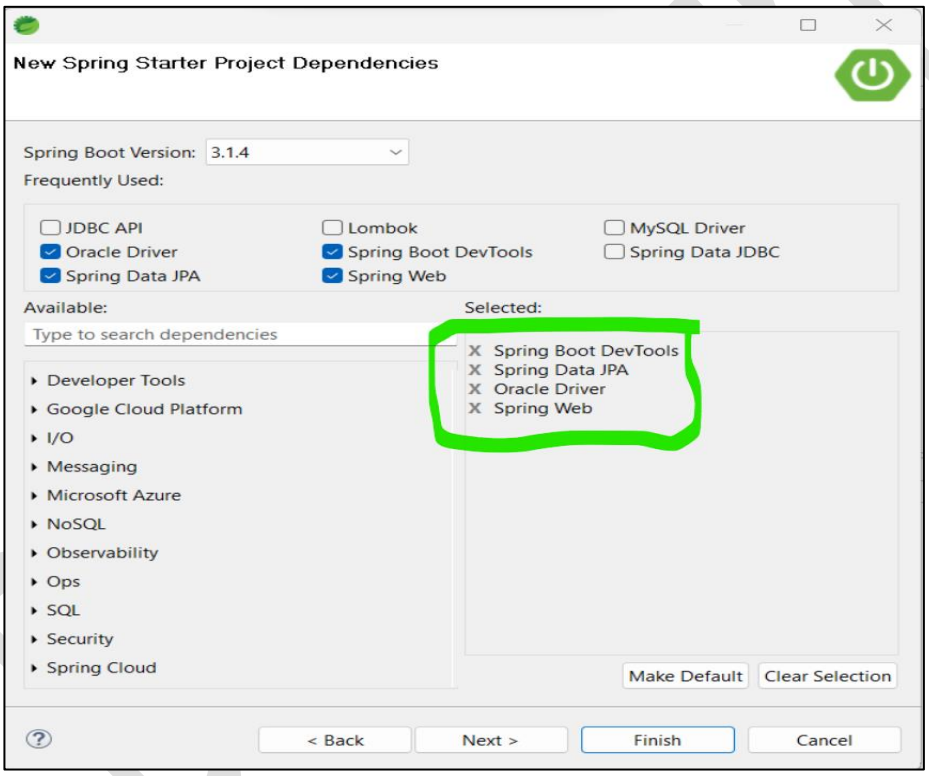
```
@Service
public class UniversityAdmissionsService {
```

```
        @Autowired
        AdmissionsRepository repository;

        //Logic

}
```

Spring Boot Web/MVC + JSP+ JPA Example:

Requirement: **Create a Project and implement User Registration and Login Flows.**

- Create a Spring Project with Web and JPA Modules.



- Add Database details and server port details inside **application.properties**.

```
server.port=9999
server.servlet.context-path=/tekteacher

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

spring.jpa.show-sql=true
```

```
spring.jpa.hibernate.ddl-auto=none
```

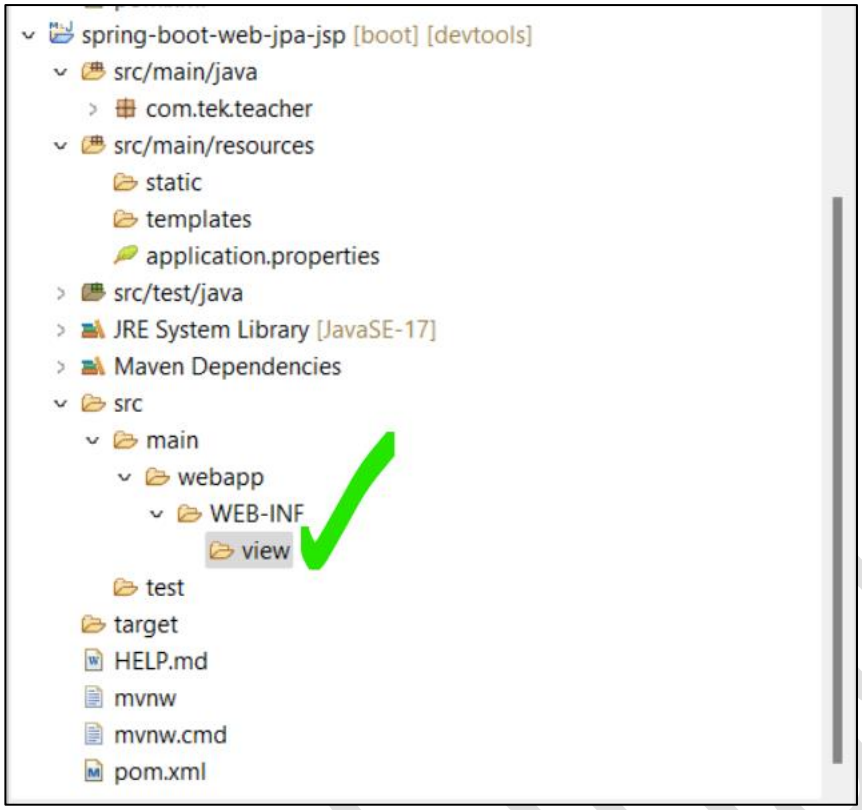
- Now Add Dependency of JSP inside **pom.xml** file.

```
<dependency>
   <groupId>org.apache.tomcat.embed</groupId>
   <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

- Now Add **ViewResolver** properties of JSP inside **application.properties** file.

```
server.port=9999
server.servlet.context-path=/tekteacher

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

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update

#JSP
spring.mvc.view.prefix=/WEB-INF/view/
spring.mvc.view.suffix=.jsp
```

- Create **view** folder as per prefix value inside our application.



- Create a JSP file for User Registration Form User Interface : **user-register.jsp**

```
<html>
<head>
    <title> User Register</title>
</head>
<body>
        <form action="user/register" method="POST">
                Name : <input type="text" name="name" /> <br />
                Email Id  : <input type="text" name="email" /> <br />
                Contact Number  : <input type="text" name="contact" /> <br />
                Password : <input type="password" name="pwd" /> <br />
                <input type="submit" value="Register" /> <br />
        </form>
</body>
</html>
```

- Create another JSP file for User Registration Result Message, whether User Account Created or Not : **result.jsp**

```
<html>
<head>
	<title> Result</title>
</head>
<body>
	${message}
</body>
</html>
```

- Create a DTO class for retrieving details from **HttpServeletRequest** Object in side Controller method.

```
package com.tek.teacher.dto;

public class UserReigtserDto {

	private  String name;
	private  String emailId;
	private  String contact;
	private  String password;

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getEmailId() {
		return emailId;
	}
	public void setEmailId(String emailId) {
		this.emailId = emailId;
	}
	public String getContact() {
		return contact;
	}
	public void setContact(String contact) {
		this.contact = contact;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
```

```
	}
}
```

- Create Controller class and Methods for loading User Registration Page and reading data from Registration page. Once Receiving Data at controller we should store it inside database.

Controller Class : **UserController.java**

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.servlet.ModelAndView;
import com.tek.teacher.dto.UserReigtserDto;
import com.tek.teacher.service.UserService;
import jakarta.servlet.http.HttpServletRequest;


@Controller
public class UserController {

	@Autowired
	UserService userService;


	//for loading HTML UI Form
	@GetMapping("register")
	public String sayHello() {
		return "register";
	}

	// From Action Endpoint for User Registration
	@PostMapping("user/register")
	public ModelAndView registerUser(HttpServletRequest request) {

		//Extracting Data From HttpServletRequest to DTO
		UserReigtserDto userReigtserDto = new UserReigtserDto();
		userReigtserDto.setName(request.getParameter("name"));
		userReigtserDto.setEmailId(request.getParameter("email"));
		userReigtserDto.setContact(request.getParameter("contact"));
		userReigtserDto.setPassword(request.getParameter("pwd"));
```

```
                String result = userService.userRegistration(userReigtserDto);

                ModelAndView modelAndView = new ModelAndView();
                //setting result jsp file name
                modelAndView.setViewName("result");
                modelAndView.addObject("message", result);

                return modelAndView;
        }
}
```

- Now Create Service Layer class and respective method for storing User Information inside database.

```
package com.tek.teacher.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.tek.teacher.dto.UserReigtserDto;
import com.tek.teacher.entity.UsersInfo;
import com.tek.teacher.repository.UserRepository;

@Service
public class UserService {

        @Autowired
        UserRepository repository;

        public String userRegistration(UserReigtserDto userReigtserDto) {

                // convert dto instance  to entity object
                UsersInfo user = new UsersInfo();
                user.setContact(userReigtserDto.getContact());
                user.setEmailId(userReigtserDto.getEmailId());
                user.setName(userReigtserDto.getName());
                user.setPassword(userReigtserDto.getPassword());
                repository.save(user);

                return "User Registration Successful.";
        }
}
```

- Now create JPA Entity class for Database Operations, with columns related to User Details.

```
package com.tek.teacher.entity;
```

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

@Entity
@Table
public class UsersInfo{

	@Id
	@Column
	private String emailId;

	@Column
	private String name;

	@Column
	private String contact;

	@Column
	private String password;

	public String getEmailId() {
		return emailId;
	}
	public void setEmailId(String emailId) {
		this.emailId = emailId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getContact() {
		return contact;
	}
	public void setContact(String contact) {
		this.contact = contact;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
```

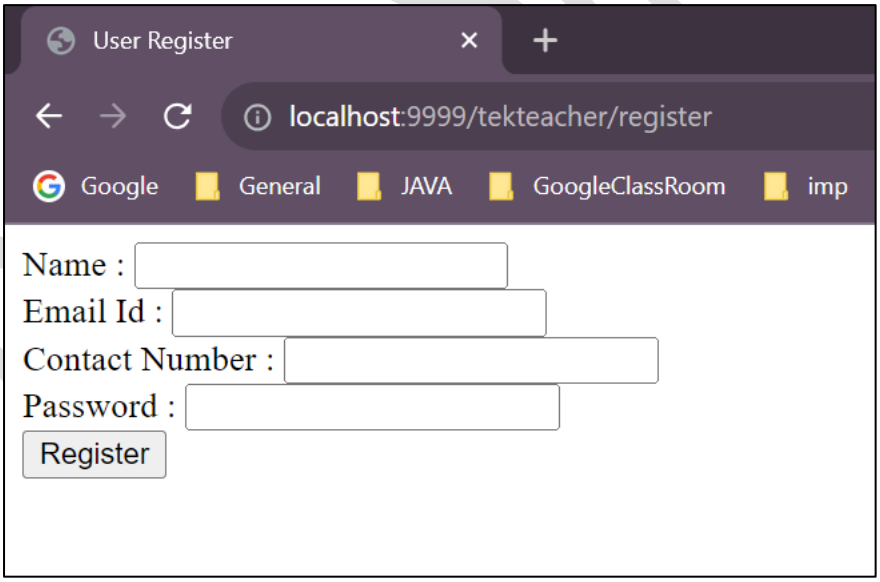
```
        }
}
```

- Now create a JPA Repository.

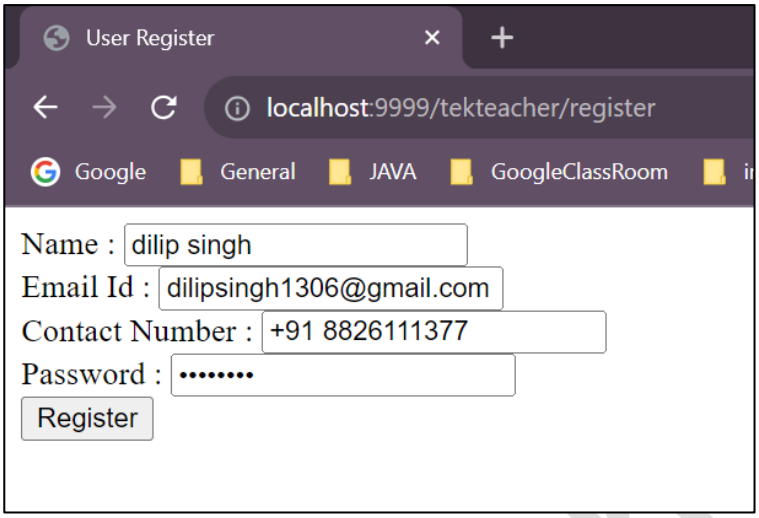
```
package com.tek.teacher.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.tek.teacher.entity.UsersInfo;

@Repository
public interface UserRepository extends JpaRepository<UsersInfo, String>{

}
```

Testing/Execution: **Start Our Spring Boot Application**

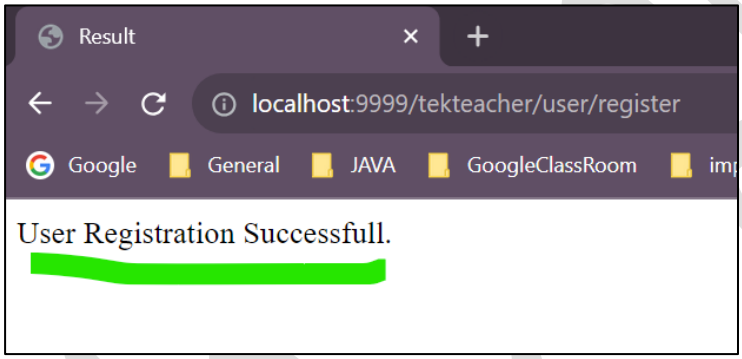
- Now Access register URI : http://localhost:9999/tekteacher/register
- This URL will load Form for entering User Details as followed.



- Now Enter User Information and then click on Register button. Internally it will trigger another endpoint **"user/register"**



Response :



Finally User Information Successfully Stored Inside Database.

Requirement: Login of User

- **Create Login UI Form :** login.jsp

```
<html>
<head>
<title>Login User</title>
</head>
<body>
	<form action="loginCheck" method="POST">
		Email : <input type="text" name="email" /> <br />
		Password : <input type="password" name="pwd" /> <br />
		<input type="submit" value="Login" /> <br />
	</form>
</body>
</html>
```

- **Now Create Controller Method For Login Page/Form Loading.**

```
@GetMapping("login")
public ModelAndView loadLoginPage() {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("login");
        return modelAndView;
}
```

- **Now Create Controller Method For Receiving Login Form Data and validation of User Details.**

```
@PostMapping("/loginCheck")
public ModelAndView validateUser(HttpServletRequest request) {

    String result = userService.validateUser(request.getParameter("email"),
                                                request.getParameter("pwd"));

     ModelAndView modelAndView = new ModelAndView();
     modelAndView.setViewName("result");
     modelAndView.addObject("message", result);
    return modelAndView;
}
```

- **Create Another Method in Service Class to connect with Repository layer**

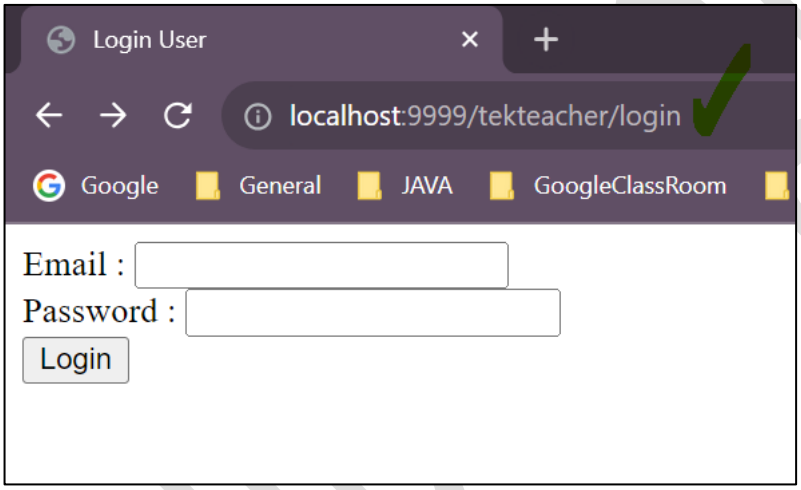
```
public String validateUser(String emailId, String password) {
        // Verify in data base
        List<FacebookUsers> users = repository.findByEmailIdAndPassword(emailId,
                                                                        password);
         if (users.size() == 0) {
                return "Invalid Credentials. Please Try again";
         } else {
                return "Welcome to FaceBook, " + emailId;
         }
}
```

- **Create a custom findBy.. JPA method inside Repository Interface.**

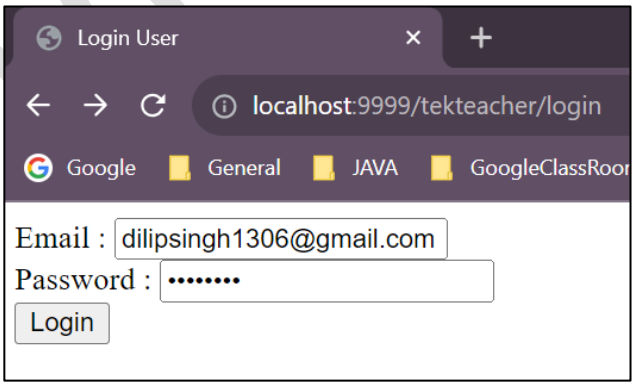
```
package com.facebook.repository;
```

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

@Repository
public interface UserRepository extends JpaRepository<FacebookUsers, String>{

    List<FacebookUsers> findByEmailIdAndPassword(String emailId, String password);

}
```

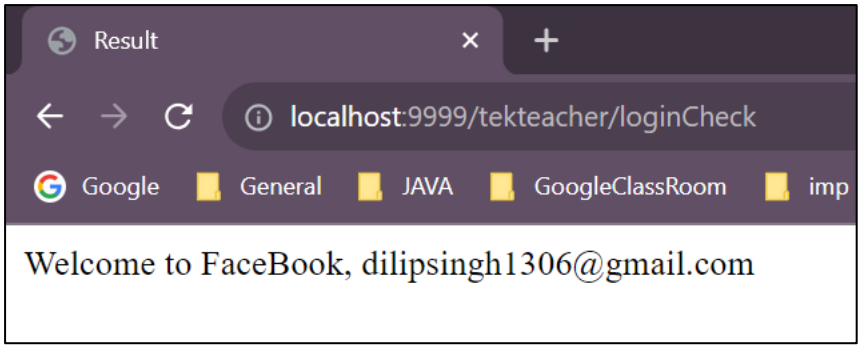
Testing: **Load Login From.**



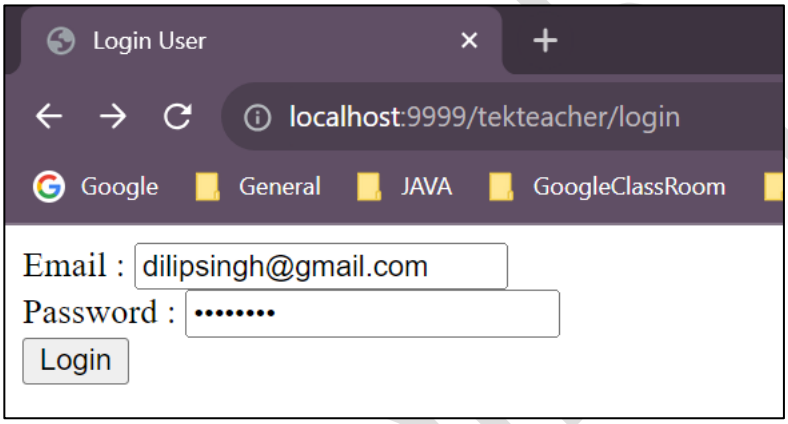
Now enter User Login Information.
Valid User Credentials: Entered Valid Email Id and Password



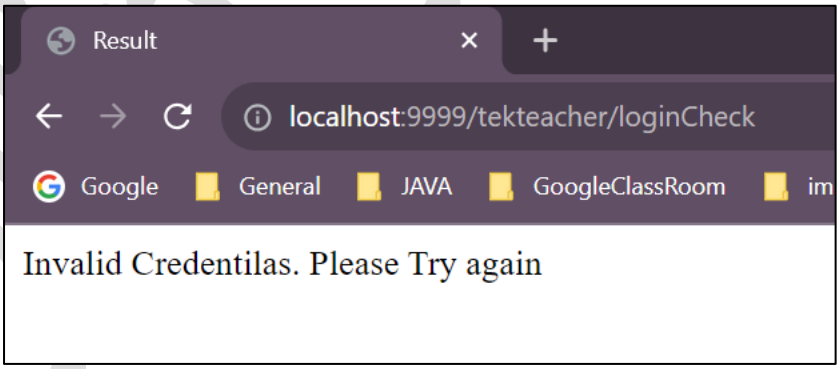
Response :



Invalid User Credentials: Entered Invalid Email Id and Password



Response :



How to Choose HTTP methods?

Generally we will choose HTTP method depends on Data Base Operations of the requirement i.e. When we are implementing Handler methods finally as part of implantation which database query is executed as explained as follows.

CRUD Operations vs HTTP methods:

Create, Read, Update, and Delete — or **CRUD** — are the four major functions used to interact with database applications. The acronym is popular among programmers, as it provides a quick reminder of what data manipulation functions are needed for an application

to feel complete. Many programming languages and protocols have their own equivalent of **CRUD**, often with slight variations in how the functions are named and what they do. For example, SQL — a popular language for interacting with databases — calls the four functions **Insert, Select, Update,** and **Delete**. CRUD also maps to the major HTTP methods.

Although there are numerous definitions for each of the CRUD functions, the basic idea is that they accomplish the following in a collection of data:

NAME	DESCRIPTION	SQL EQUIVALENT
Create	Adds one or more new entries	Insert
Read	Retrieves entries that match certain criteria (if there are any)	Select
Update	Changes specific fields in existing entries	Update
Delete	Entirely removes one or more existing entries	Delete

Generally most of the time we will choose HTTP methods of an endpoint based on Requirement Functionality performing which operation out of CRUD operations. This is a best practice of creating REST API's.

CRUD	**HTTP**
CREATE	POST
READ	GET
UPDATE	PUT
DELETE	DELETE

Webservices:

Web services are a standardized way for different software applications to communicate and exchange data. They enable interoperability between various systems, regardless of the programming languages or platforms they are built on. Web services use a set of protocols and technologies to enable communication and data exchange between different applications, making it possible for them to work together without any issues.

Web services are used to integrate different applications and systems, regardless of their platform or programming language. They can be used to provide a variety of services, such as:

- Information retrieval
- Transaction processing
- Data exchange
- Business process automation

There are two main types of web services:

1. SOAP (Simple Object Access Protocol) Web Services:

SOAP is a protocol for exchanging structured information using XML. It provides a way for applications to communicate by sending messages in a predefined format. SOAP web services offer a well-defined contract for communication and are often used in enterprise-level applications due to their security features and support for more complex scenarios.

2. REST (Representational State Transfer) Web Services:

REST is an architectural style that uses HTTP methods (GET, POST, PUT, DELETE) to interact with resources in a stateless manner. RESTful services are simple, lightweight, and widely used due to their compatibility with the HTTP protocol. They are commonly used for building APIs that can be consumed by various clients, such as web and mobile applications**.** The choice of web service type depends on factors such as the nature of the application, the level of security required, the complexity of communication, and the preferred data format.

REST API:

RESTful API is an interface that two computer systems use to exchange information securely over the internet. Most business applications have to communicate with other internal and third-party applications to perform various tasks.

API: An API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information consumer. An application programming interface (API) defines the rules that you must follow to communicate with other software systems. Developers expose or create APIs so that other applications can communicate with their applications programmatically. For example, the ICICI application exposes an API that asks for banking users, Card Details , Name, CVV etc.. When it receives this information, it internally processes the users data and returns the payment status.

REST is a set of architectural style but not a protocol or a standard. API developers can implement REST in a variety of ways. When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information or representation is delivered in one of several formats like JSON or XML via HTTP Protocol.

JSON is the most generally popular format to use because, despite its name, it's language-agnostic, as well as readable by both humans and machines.

REST API architecture that imposes conditions on how an API should work. REST was initially created as a guideline to manage communication on a complex network like the internet. You can use REST-based architecture to support high-performing and reliable communication at scale. You can easily implement and modify it, bringing visibility and cross-platform portability to any API system.

Clients: Clients are users who want to access information from the web. The client can be a person or a software system that uses the API. For example, developers can write programs that access weather data from a weather system. Or you can access the same data from your browser when you visit the weather website directly.

Resources: Resources are the information that different applications provide to their clients/users. Resources can be images, videos, text, numbers, or any type of data. The machine that gives the resource to the client is also called the server. Organizations use APIs to share resources and provide web services while maintaining security, control, and authentication. In addition, APIs help them to determine which clients get access to specific internal resources.

API developers can design APIs using several different architectures. APIs that follow the REST architectural style are called REST APIs. Web services that implement REST architecture are called RESTful web services. The term RESTful API generally refers to RESTful web APIs. However, you can use the terms REST API and RESTful API interchangeably.

The following are some of the principles of the REST architectural style:

Uniform Interface: The uniform interface is fundamental to the design of any RESTful webservice. It indicates that the server transfers information in a standard format. The formatted resource is called a representation in REST. This format can be different from the internal representation of the resource on the server application. For example, the server can store data as text but send it in an HTML representation format.

Statelessness: In REST architecture, statelessness refers to a communication method in which the server completes every client request independently of all previous requests. Clients can request resources in any order, and every request is stateless or isolated from other requests. This REST API design constraint implies that the server can completely understand and fulfil the request every time.

Layered system: In a layered system architecture, the client can connect to other authorized intermediate services between the client and server, and it will still receive responses from the server. Sometimes servers can also pass on requests to other servers. You can design your RESTful web service to run on several servers with multiple layers such as security, application, and business logic, working together to fulfil client requests. These layers remain invisible to the client. We can achieve this as part of Micro Services Design.

What are the benefits of RESTful APIs?

RESTful APIs include the following benefits:

Scalability: Systems that implement REST APIs can scale efficiently because REST optimizes client-server interactions. Statelessness removes server load because the server does not have to store past client request information.

Flexibility: RESTful web services support total client-server separation. Platform or technology changes at the server application do not affect the client application. The ability to layer application functions increases flexibility even further. For example, developers can make changes to the database layer without rewriting the application logic.

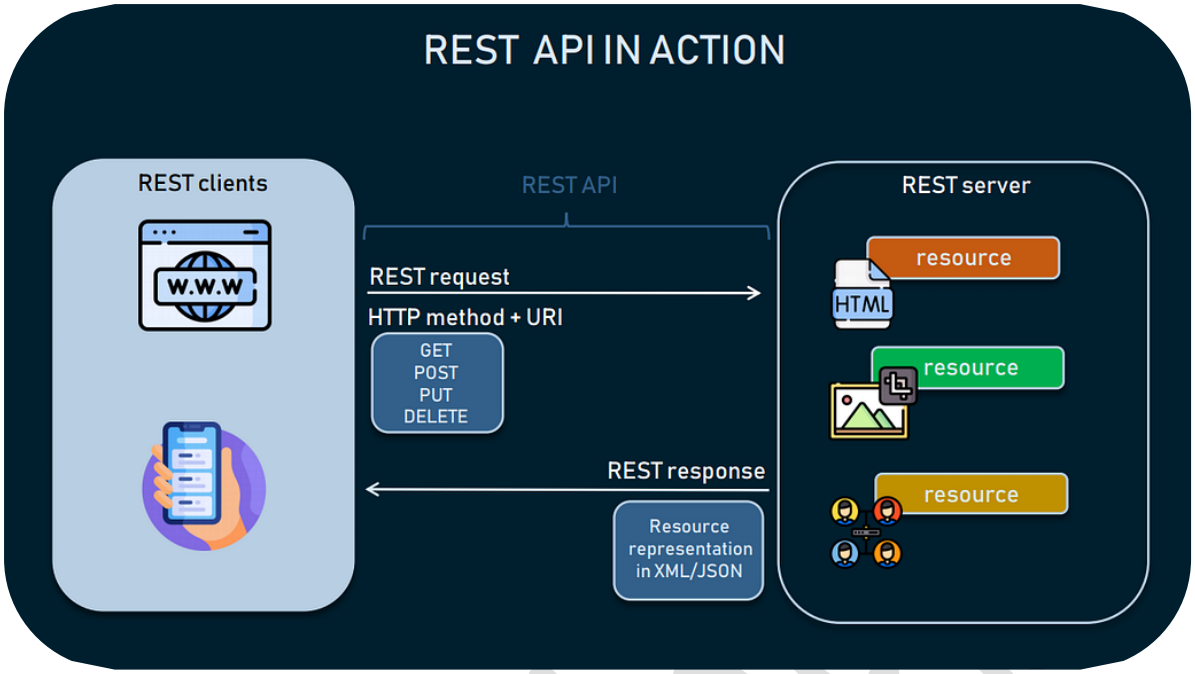
Platform and Language Independence: REST APIs are platform and language independent, meaning they can be consumed by a wide range of clients, including web browsers, mobile devices, and other applications. As long as the client can send HTTP requests and understand the response, it can interact with a REST API regardless of the technology stack used on the server side. You can write both client and server applications in various programming languages without affecting the API design. We can also change the technology on both sides without affecting the communication.

Overall, REST APIs provide a simple, scalable, and widely supported approach to building web services. These advantages in terms of simplicity, platform independence, scalability, flexibility, and compatibility make REST as a popular choice for developing APIs in various domains, from web applications to mobile apps and beyond.

How RESTful APIs work?

The basic function of a RESTful API is the same as browsing the internet. The client contacts the server by using the API when it requires a resource. API developers explain how the client should use the REST API in the server application with API documentation. These are the general steps for any REST API call integration:

1. The client sends a request to the server. The client follows the API documentation to format the request in a way that the server understands.
2. The server authenticates the client and Request and confirms that the client has the right to make that request.
3. The server receives the request and processes it internally.
4. The server returns a response to the client. The response contains information that tells the client whether the request was successful. The response also includes any information that the client requested.



The REST API request and response details are vary slightly depending on how the API developers implemented the API.

What does the RESTful API client request contain?

RESTful APIs require requests to contain the following main components:

URI (Unique Resource Identifier) : The server identifies each resource with unique resource identifiers. For REST services, the server typically performs resource identification by using a Uniform Resource Locator (URL). The URL specifies the path to the resource. A URL is similar to the website address that you enter into your browser to visit any webpage. The URL is also called the request endpoint and clearly specifies to the server what the client requires.

HTTP Method: Developers often implements RESTful APIs by using the Hypertext Transfer Protocol (HTTP). An HTTP method tells the server what it needs to do with the resource. The following are four common HTTP methods:

- **GET:** Clients use GET to access resources that are located at the specified URL on the server.
- **POST:** Clients use POST to send data to the server. They include the data representation with the request body. Sending the same POST request multiple times has the side effect of creating the same resource multiple times.
- **PUT:** Clients use PUT to update existing resources on the server. Unlike POST, sending the same PUT request multiple times in a RESTful web service gives the same result.
- **DELETE:** Clients use DELETE request to remove the resource.

HTTP Headers: Request headers are the metadata exchanged between the client and server.

Data: REST API requests might include data for the POST, PUT, and other HTTP methods to work successfully.

Parameters: RESTful API requests can include parameters that give the server more details about what needs to be done. The following are some different types of parameters:

- **Path parameters** that specify URL details.
- **Query/Request parameters** that request more information about the resource.
- **Cookie parameters** that authenticate clients quickly.

What does the RESTful API server response contain?

REST principles require the server response to contain the following main components:

Status line: The status line contains a three-digit status code that communicates request success or failure.

2XX codes indicate success
4XX and 5XX codes indicate errors
3XX codes indicate URL redirection.

The following are some common status codes:

200: Generic success response
201: POST method success response as Created Resource
400: Incorrect/Bad request that the server cannot process
404: Resource not found

Message body: The response body contains the resource representation. The server selects an appropriate representation format based on what the request headers contain i.e. like JSON/XML formats. Clients can request information in XML or JSON formats, which define how the data is written in plain text. For example, if the client requests the name and age of a person named John, the server returns a JSON representation as follows:

```
{
        "name":"John",
        "age":30
}
```

Headers: The response also contains headers or metadata about the response. They give more context about the response and include information such as the server, encoding, date, and content type.

As per REST API creation Guidelines, we should choose HTTP methods depends on the Database Operation performed by our functionality, as We discussed previously.

REST Services Implementation in Spring MVC:

Spring MVC is a popular framework for creating web applications in Java. Implementing RESTful web services in Spring MVC involves using the Spring framework to create endpoints that follow the principles of the REST architectural style. It can be used to create RESTful web services, which are web services that use the REST architectural style.

RESTful services allow different software systems to communicate over the internet using standard HTTP methods, like GET, POST, PUT, and DELETE. These services are based on a set of principles that emphasize simplicity, scalability, and statelessness.

In REST Services implementation, Data will be represented as **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 more 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 is :

```
{
   "name":"John",
    "age":30,
    "car":null
}
```

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

JSON Syntax Rules:

JSON syntax is derived from JavaScript object notation syntax:

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

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

Example: `"name" : "John"`

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

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

JSON vs XML:

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

JSON Example

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

XML Example:

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

```
                <lastName>Jones</lastName>
            </employee>
        </employees>
```

JSON is Like XML Because

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

When A Request Body Contains JSON/XML data Format, then how Spring MVC/JAVA language handling Request data?

Here, We should **Convert JSON/XML data to JAVA 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.

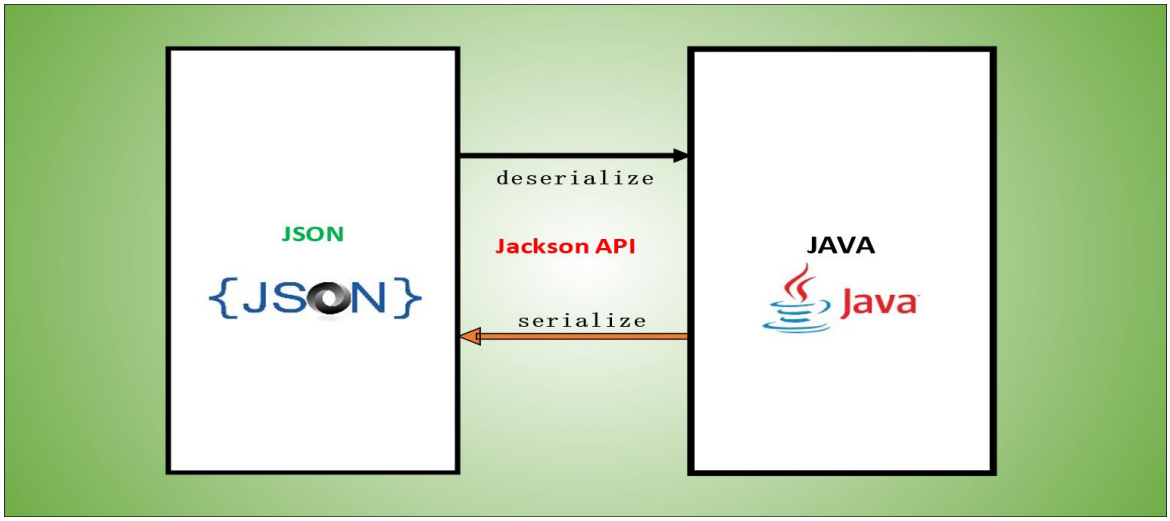
For these conversions, we have few pre-defined solutions in Market like Jackson API, GSON API, JAXB etc..

JSON to JAVA Conversion:

Spring MVC supports Jackson API which will take care of un-marshalling/mapping JSON request body to Java objects. We should add **Jackson API dependencies** explicitly In Spring MVC, If It is Spring Boot MVC application Jackson API jars will be added internally. We can use **@RequestBody** Spring MVC annotation to deserialize/un-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 + "]";
	}
```

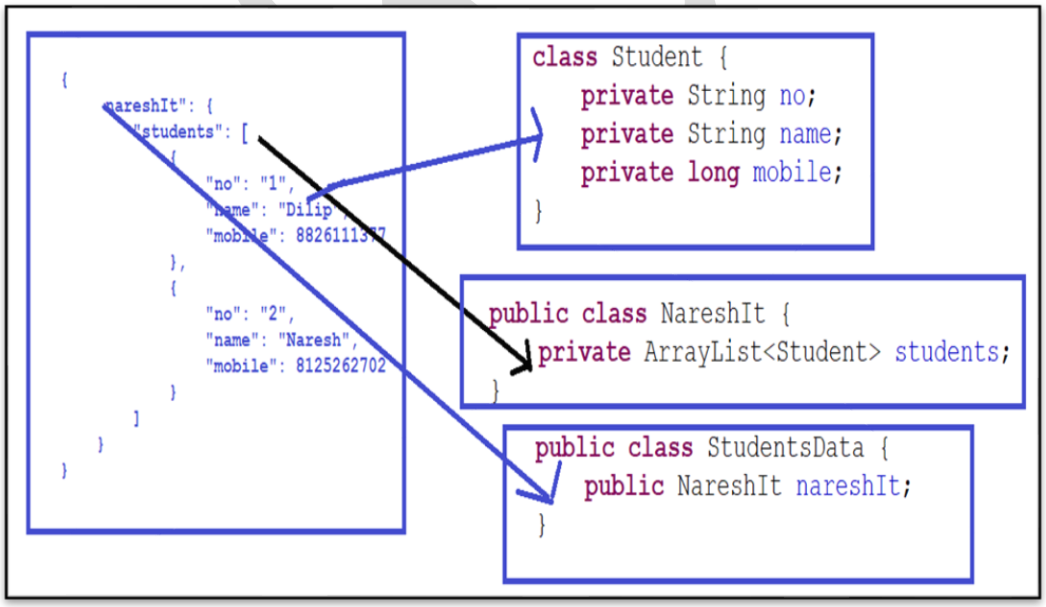
```
}
```

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.

@RestController = @Controller + @ResponseBody

```
@Controller
public class MAcBookController {

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

	@GetMapping(path = "/iphone/details")
	@ResponseBody
	public String getIphoneDetail() {
		return "Iphone Details : Price 150000. Model 15 Pro";
	}
}
```

- If we Used **@RestController** with controller class, removed @**ResponseBody** at all handler mapping methods level.

```
@RestController
public class MAcBookController {

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

	@GetMapping(path = "/iphone/details")
	public String getIphoneDetail() {
		return "Iphone Details : Price 150000. Model 15 Pro";
	}
}
```

That's all, **@RestController** is just like a shortcut annotation to avoid declaring **@ResponseBody** on every controller handler mapping method inside a class.

@RequestBody Annotation:

The **@RequestBody** annotation in Spring is used to bind the HTTP request body to a method parameter. This means that Spring will automatically deserialize the request body into a Java object and that object is then passed to the method as a parameter. The **@RequestBody** annotation can be used on controller methods.

For example, the following controller method accepts a User object as a parameter:

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
        // ...
}
```

- When you send a POST request to the **/users** endpoint with a JSON body containing the user data, Spring will automatically deserialize the JSON into a User object and pass it to the **createUser()** method.

- The **@RequestBody** annotation is a powerful tool that makes it easy to work with HTTP request bodies in Spring. It is especially useful for developing REST APIs.

Here are some additional things to keep in mind about the **@RequestBody** annotation:

- The request body must be in a supported media type, such as JSON, XML.
- Spring will use an appropriate HTTP message converter to deserialize the request body.
- If the request body cannot be deserialized, Spring will throw a **HttpMessageNotReadableException**.

Postman API Testing Tool:

Postman is a popular and widely used API (Application Programming Interface) testing and development tool. It provides a user-friendly interface for sending HTTP requests to APIs and inspecting the responses. Postman offers a range of features that make it valuable for developers, testers, and anyone working with APIs:

Some of the key features of Postman API Tools include:

- **API client:** Postman provides a powerful API client that allows you to send HTTP requests to any API and inspect the responses. The API client supports a wide range of authentication protocols and response formats.

- **API testing:** Postman provides a powerful API testing framework that allows you to create and execute tests for your APIs. Postman tests can be used to validate the functionality, performance, and security of your APIs.
- **API design:** Postman can be used to design your API specifications in OpenAPI, RAML, GraphQL, or SOAP formats. Postman's schema editor makes it easy to work with specification files of any size, and it validates specifications with a built-in linting engine.
- **API documentation:** Postman can be used to generate documentation for your APIs in a variety of formats, including HTML, Markdown, and PDF. Postman documentation is automatically generated from your API requests, so it is always up-to-date.
- **API monitoring:** Postman can be used to monitor your APIs for performance and availability issues. Postman monitors can be configured to send alerts when your APIs are unavailable or when they are not performing as expected.

Postman is a powerful tool that can help you to streamline your API development workflow. It is used by developers and teams of all sizes to build, test, document, and monitor APIs.

Here are some examples of how Postman API Tools can be used:

- A developer can use Postman to explore a new API and learn how to use it.
- A QA engineer can use Postman to create and execute tests for an API.
- A DevOps engineer can use Postman to monitor an API for performance and availability issues.
- A product manager can use Postman to generate documentation for an API.
- A sales engineer can use Postman to demonstrate an API to a customer.

Project Setup:

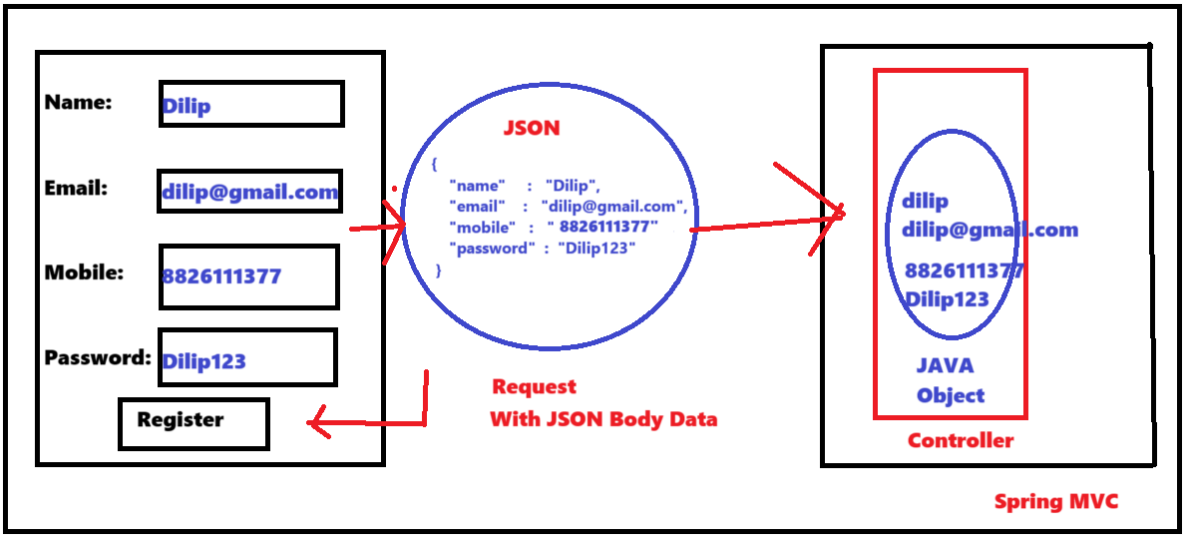
1. Create Spring Boot Web Module Project
2. Now Create an endpoints or REST Services for below requirement.

Requirement: **Write a Rest Service for User Registration. User Details Should Be :**

- **User Name**
- **Email Id**
- **Mobile**
- **Password**

- Create a JSON Mapping for above Requirement with dummy values.

```
{
  "name"     :  "Dilip",
  "email"    :  "dilip@gmail.com",
  "mobile"   :  "+91 73777373",
  "password" :  "Dilip123"
}
```



3. Before Creating Controller class, we should Create JAVA POJO class which is compatible with JSON Request Data. So create a JAVA class, as discussed previously. Which is Responsible for holding Request Data of JSON.

```
package com.swiggy.user.request;

public class UserRegisterRequest {

	private String name;
	private String email;
	private String mobile;
	private String password;

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public String getMobile() {
		return mobile;
	}
	public void setMobile(String mobile) {
		this.mobile = mobile;
```

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

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

```
package com.swiggy.user.controller;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.swiggy.user.request.UserRegisterRequest;

@RestController
@RequestMapping("/user")
public class UserController {

	@PostMapping("/register")
	public String getUserDetails(@RequestBody UserRegisterRequest request){
		System.out.println(request.getEmail());
		System.out.println(request.getName());
		System.out.println(request.getPassword());
		return "User Created Successfully";
	}
}
```

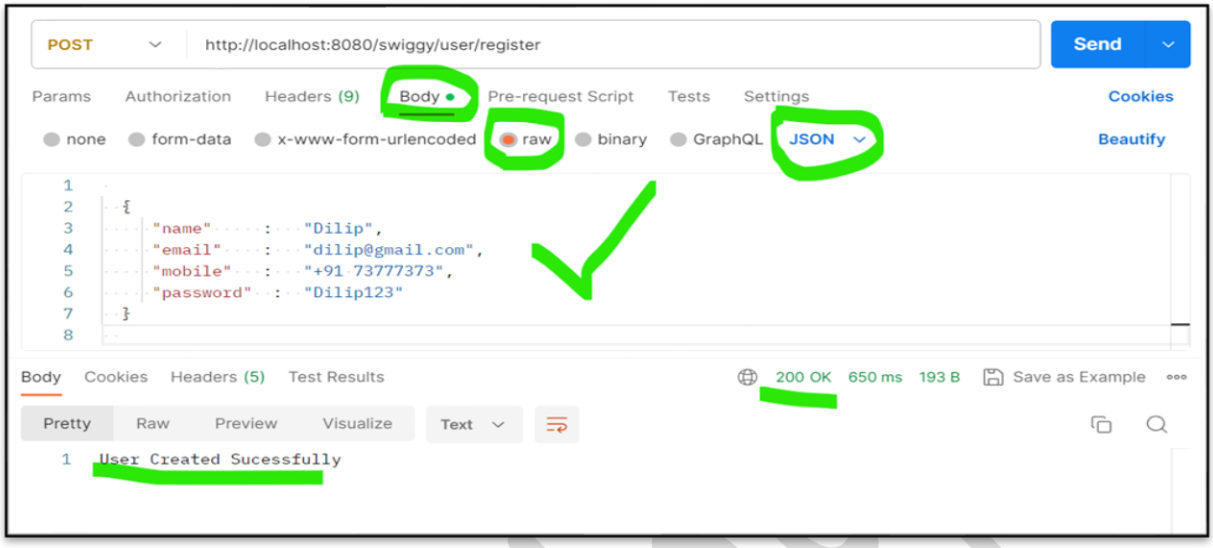
In Above, We are used **@RequestBody** for binding/mapping incoming JSON request to JAVA Object at method parameter layer level. Means, Spring MVC internally maps JSON to JAVA with help of Jackson API jar files.

5. Deploy Your Application in Server Now. After Deployment we have to Test now weather it Service is working or not.

6. Now We are Taking Help of **Postman** to do API/Services Testing.

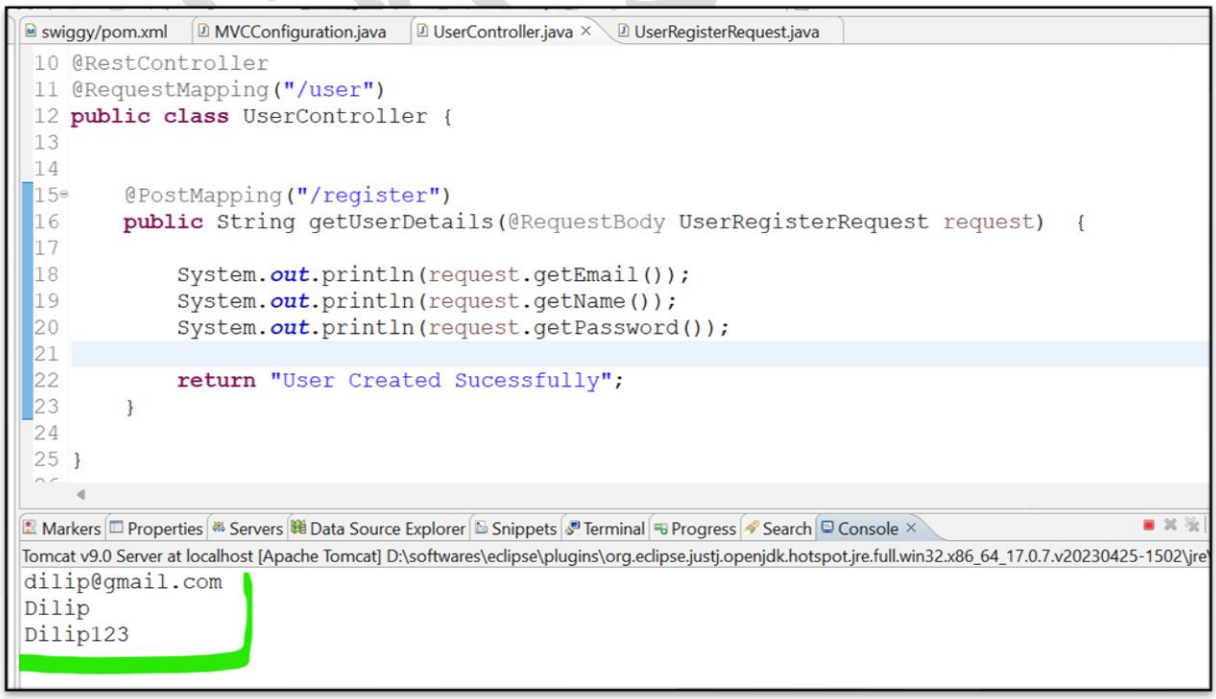
   - Open Postman
   - Now Click on Add request
   - Select Your Service HTTP method
   - And Enter URL of Service
   - Select Body

- Select raw
- Select JSON
- Enter JSON Body as shown in Below.



After Clicking on **Send** Button, Summited Request to Spring MVC REST Service Endpoint method and we got Response back with **200 Ok** status Code.

In Below, We can See in Server Console, Request Data is printed what we Received from Client/Postman level as JSON data.



Path Variables:

Path variable is a template variable called as place holder of URI, i.e. this variable path of URI. **@PathVariable** annotation can be used to handle template variables in the request URI mapping, and set them as method parameters. Let's see how to use **@PathVariable** and its various attributes. We will define path variable as part of URI in side curly braces{}.

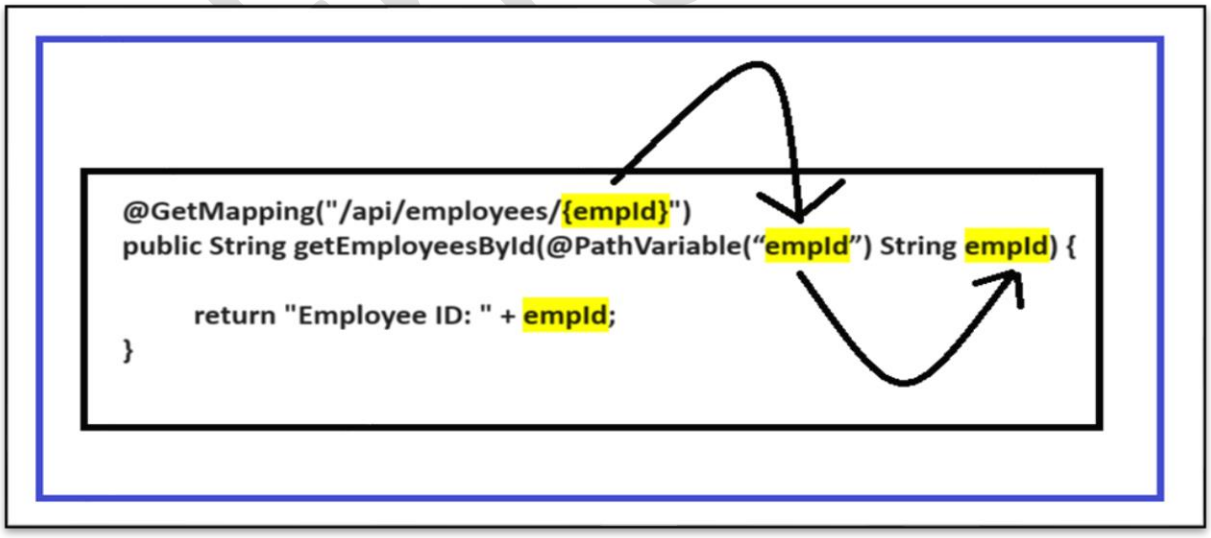
Package of Annotation: org.springframework.web.bind.annotation.PathVariable;

Examples:

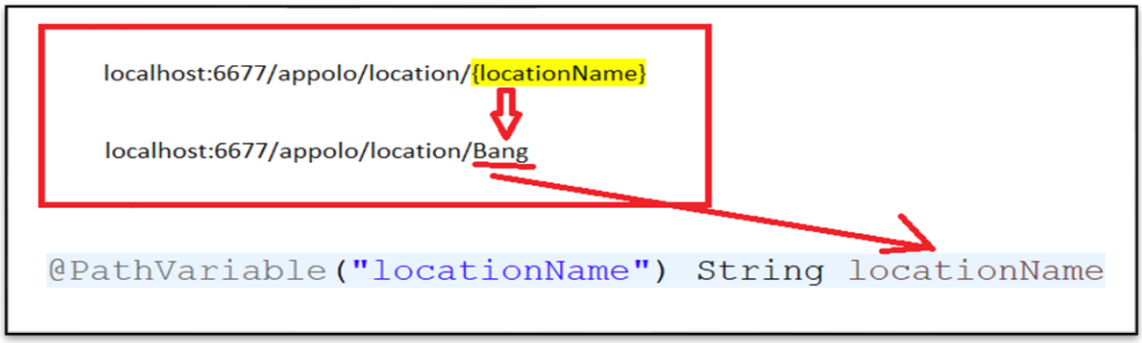
URI with Template Path variables : /location/{**locationName**}/pincode/**{pincode}**
URI with Data replaced : /location/**Hyderabad**/pincode/**500072**

Example for endpoint URI mapping in Controller : /api/employees/{empId}

```
@GetMapping("/api/employees/{empId}")
public String getEmployeesById(@PathVariable("empId") String empId) {

        return "Employee ID: " + empId;
}
```



Example 2: Path variable Declaration



Requirement : Get User Details with User Email Id.

In these kind of requirements, like getting Data with Resource ID's. We can Use Path Variable as part of URI instead of JSON mapping and equivalent Request Body classes. So Create a REST endpoint with a Path Variable of Email ID.

UserController.java : Add Below Logic In existing User Controller.

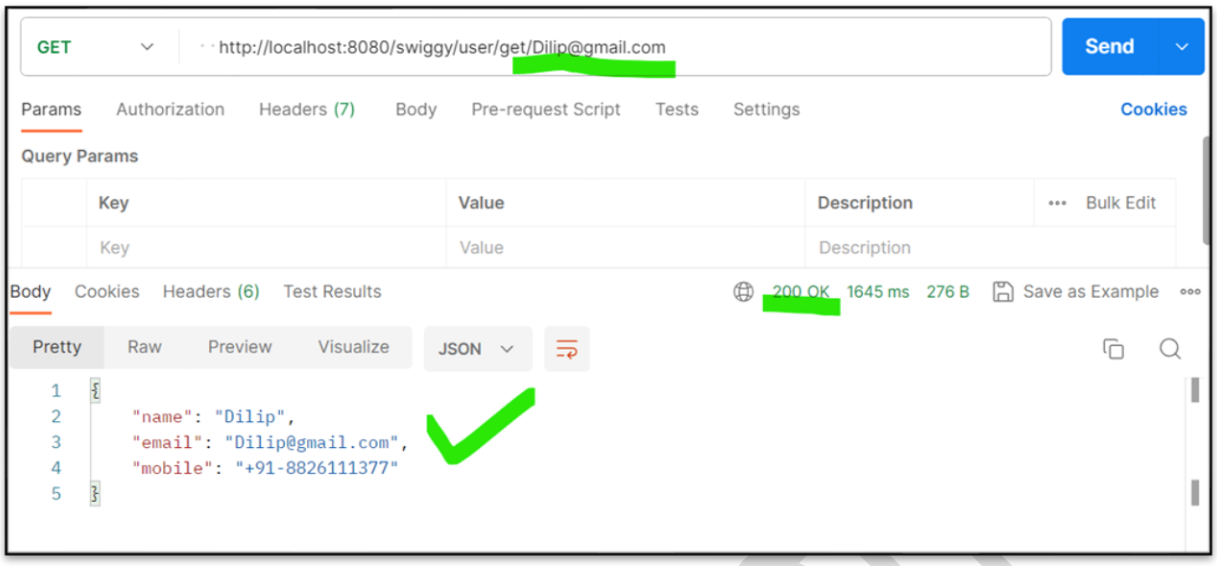
```
@RequestMapping(value = "/get/{emailId}", method = RequestMethod.GET)
public UserRegisterResponse getUserByEmailId(@PathVariable("emailId") String email) {
        return userService.getUserDetails(email);
}
```

- **Now Add Method in Service Class for interacting with Repository Layer.**

Method inside Service Class : UserService.java

```
        public UserRegisterResponse getUserDetails(String email) {
                SwiggyUsers user = repository.findById(email).get();
                //Entity Object to DTO object
                UserRegisterResponse response = new UserRegisterResponse();
                response.setEmail(user.getEmail());
                response.setMobile(user.getMobile());
                response.setName(user.getName());
                return response;
        }
```

Testing: Pass Email Value in place of PATH variable



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.

Syntax : **/{pathvar1}/{pathvar2}**
Example: **/pharmacy/{location}/pincode/{pincode}**

Requirement : Add Order Details as shown in below.

- Order ID
- Order status
- Amount
- Email Id
- City

After adding Orders, Please Get Order Details based on **Email Id** and **Order Status.**

In this case, we are passing values of Email ID and Order Status to find out Order Details. Now we can take **Path variables** here to fulfil this requirement.

- **Create an endpoints for adding Order Details and Getting Order Details with Email ID and Order Status.**

Create Request, Response and Entity Classes.

OrderRequest.java

```
package com.swiggy.order.request;

public class OrderRequest {

	private String orderID;
	private String orderstatus;
	private double amount;
	private String emailId;
	private String city;

	public String getOrderID() {
		return orderID;
	}
	public void setOrderID(String orderID) {
		this.orderID = orderID;
	}
	public String getOrderstatus() {
		return orderstatus;
	}
	public void setOrderstatus(String orderstatus) {
		this.orderstatus = orderstatus;
	}
	public double getAmount() {
		return amount;
	}
	public void setAmount(double amount) {
		this.amount = amount;
	}
	public String getEmailId() {
		return emailId;
	}
	public void setEmailId(String emailId) {
		this.emailId = emailId;
	}
	public String getCity() {
		return city;
	}
	public void setCity(String city) {
		this.city = city;
	}
}
```

- **OrderResponse.java**

```
package com.swiggy.order.response;
```

```
public class OrderResponse {

	private String orderID;
	private String orderstatus;
	private double amount;
	private String emailId;
	private String city;

	public OrderResponse() {
	}
	 public OrderResponse(String orderID, String orderstatus, double amount,
	                                                    String emailId,  String city) {
		this.orderID = orderID;
		this.orderstatus = orderstatus;
		this.amount = amount;
		this.emailId = emailId;
		this.city = city;
	}
	public String getOrderID() {
		return orderID;
	}
	public void setOrderID(String orderID) {
		this.orderID = orderID;
	}
	public String getOrderstatus() {
		return orderstatus;
	}
	public void setOrderstatus(String orderstatus) {
		this.orderstatus = orderstatus;
	}
	public double getAmount() {
		return amount;
	}
	public void setAmount(double amount) {
		this.amount = amount;
	}
	public String getEmailId() {
		return emailId;
	}
	public void setEmailId(String emailId) {
		this.emailId = emailId;
	}
	public String getCity() {
		return city;
	}
	public void setCity(String city) {
```

```
                this.city = city;
        }
}
```

➢ **Entity Class :** SwiggyOrders.java

```
package com.swiggy.order.entity;

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

@Entity
@Table(name = "swiggy_orders")
public class SwiggyOrders {

        @Id
        @Column
        private String orderID;
        @Column
        private String orderstatus;
        @Column
        private double amount;
        @Column
        private String emailId;
        @Column
        private String city;

        public String getOrderID() {
                return orderID;
        }
        public void setOrderID(String orderID) {
                this.orderID = orderID;
        }
        public String getOrderstatus() {
                return orderstatus;
        }
        public void setOrderstatus(String orderstatus) {
                this.orderstatus = orderstatus;
        }
        public double getAmount() {
                return amount;
        }
        public void setAmount(double amount) {
                this.amount = amount;
```

```
	}
	public String getEmailId() {
		return emailId;
	}
	public void setEmailId(String emailId) {
		this.emailId = emailId;
	}
	public String getCity() {
		return city;
	}
	public void setCity(String city) {
		this.city = city;
	}
}
```

➢ **Controller class : OrderController.java**

```
package com.swiggy.order.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.swiggy.order.request.OrderRequest;
import com.swiggy.order.response.OrderResponse;
import com.swiggy.order.service.OrderService;

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

	@Autowired
	OrderService orderService;

	@PostMapping(value = "/create")
	public String createOrder(@RequestBody OrderRequest request) {
		return  orderService.createOrder(request);
	}

	@GetMapping("/email/{emailId}/status/{status}")
	public List<OrderResponse>  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 reposity;

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

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

            List<OrderResponse>  allOrders =  orders.stream().map(
                    v -> new OrderResponse(
                      v.getOrderID(),
                      v.getOrderstatus(),
```

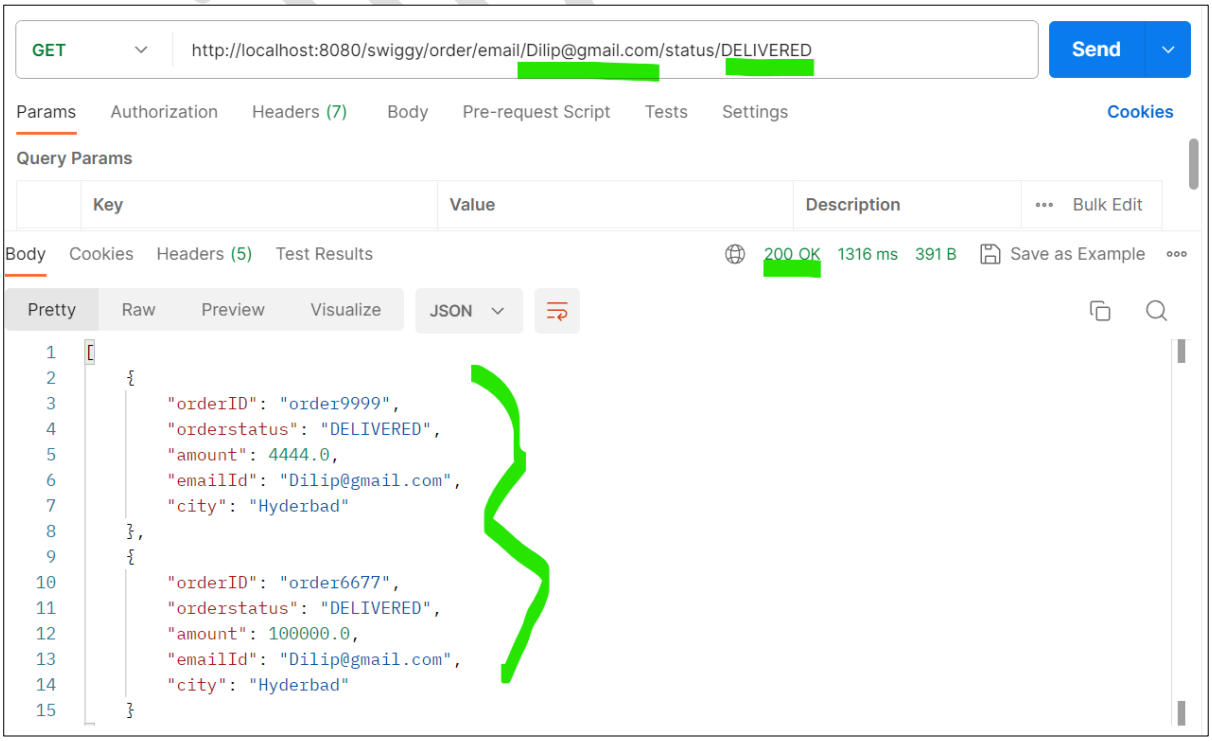
```
                        v.getAmount(),
                        v.getEmailId(),
                        v.getCity()
                    )).collect(Collectors.toList());

                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>

```java
@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 :

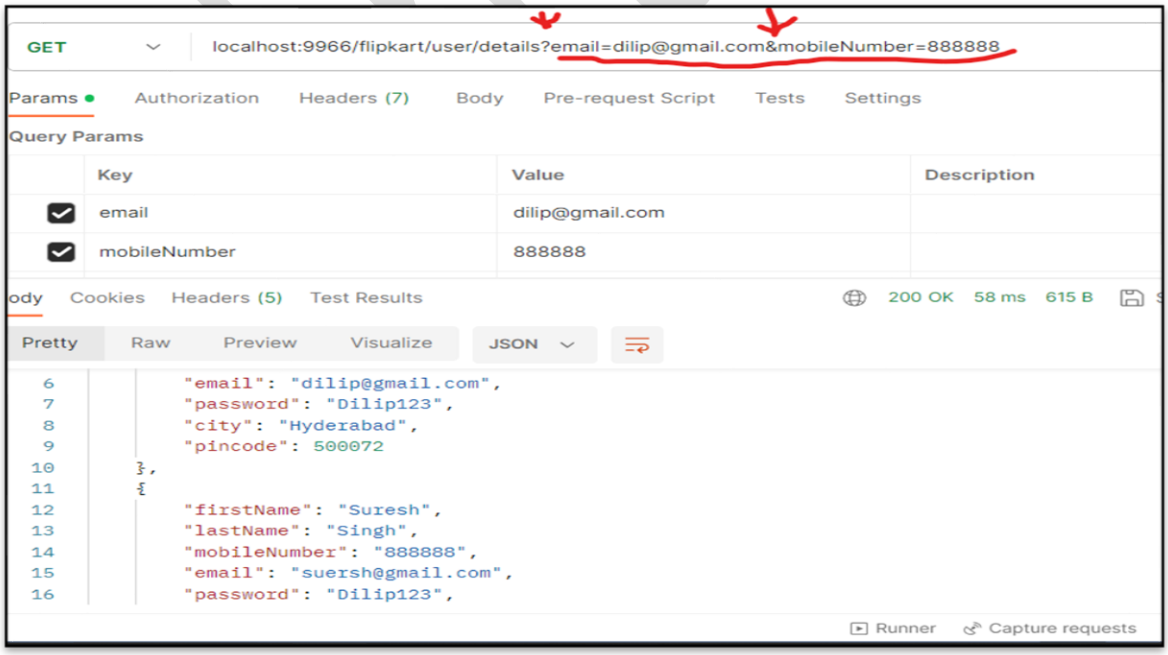
Requirement: Please Get User Details by using either email or mobile number

Method in controller:

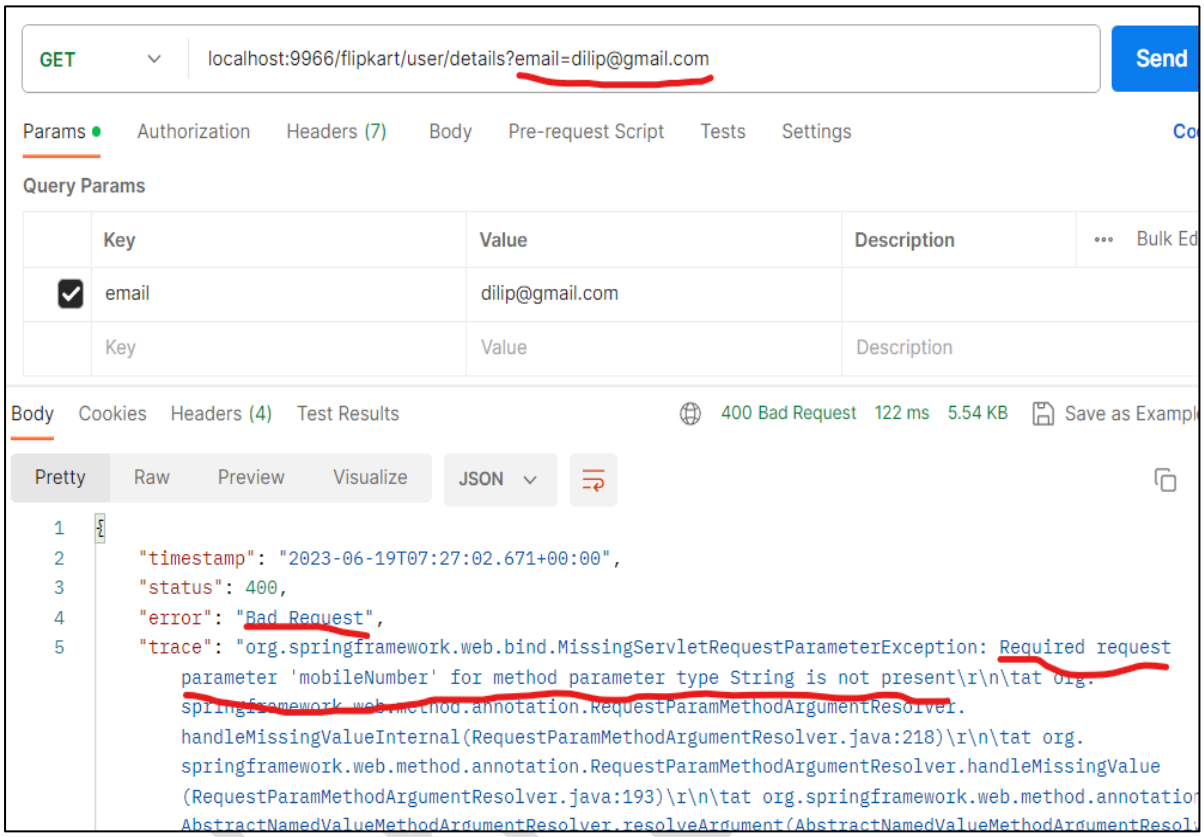
```java
@GetMapping("/details")
public List<Users> getUsersByEmailOrMobile(@RequestParam String email,
                                           @RequestParam String  mobileNumber) {

        //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>**



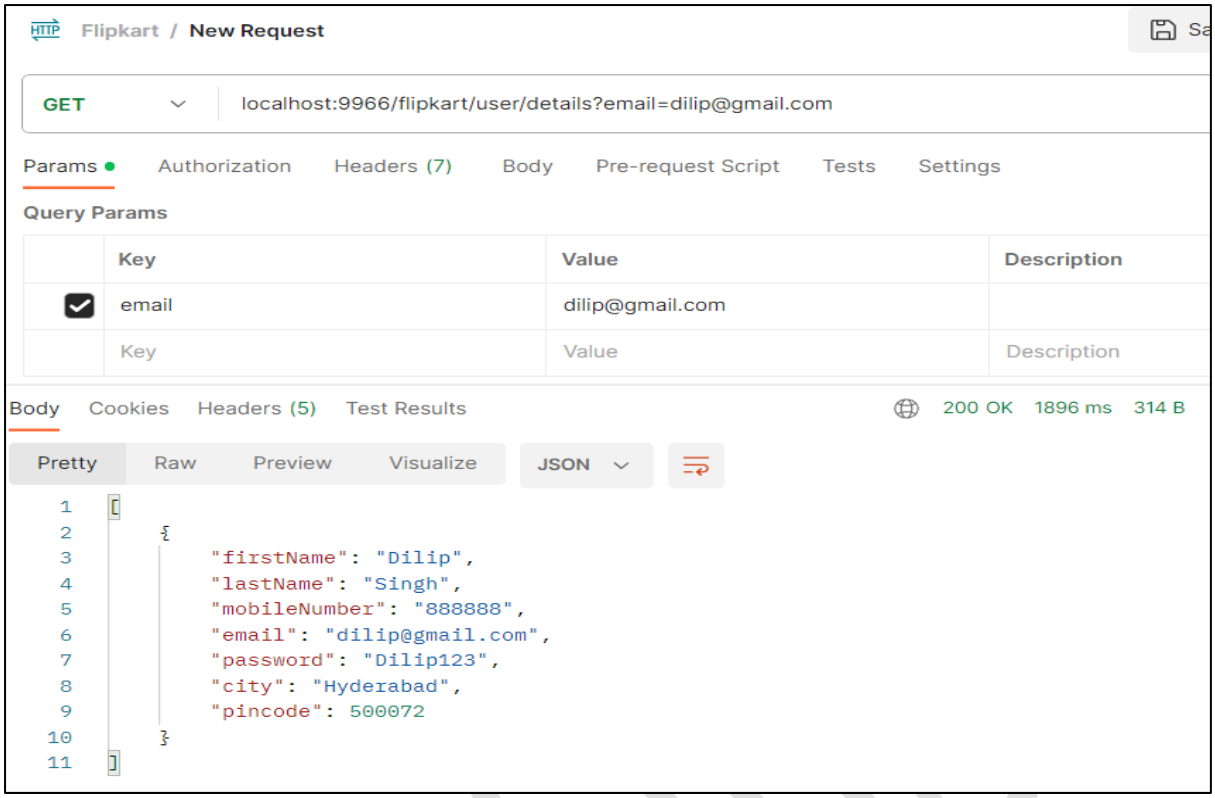
Note: By Default every Request Parameter variable is Required i.e. we should pass Query Parameter and its value as part of URL always. If we are missed any parameter, then we will get bad request.



If we want to make sure any request parameter as optional, then we have to use attribute **required=false** in side **@RequestParam** annotation. Now let's make Request Parameter **mobileNumber** as an Optional in controller.

```
@GetMapping("/details")
public List<Users> getUsersByEmailOrMobile(@RequestParam String email,
                                           @RequestParam(required = false) String mobileNumber) {

    List<Users> response = service.getUsersByEmailOrMobile(email, mobileNumber);
    return response;
}
```

Testing Endpoint: Now mobileNumber Request Parameter is missing in URI, but still our endpoint is working only with one Request parameter email.



Mapping a Multi-Value Parameter: A single @RequestParam can have multiple values:

```
@GetMapping("/api")
@ResponseBody
public String getUsers(@RequestParam List<String> id) {
        return "IDs are " + id;
}
```

And Spring MVC will map a comma-delimited id parameter:

URI: **/api?id=1,2,3**

Or we can pass a list of separate id parameters as part of URL

URI : **/api?id=1&id=2**

Mapping All Parameters:

We can also have multiple parameters without defining their names or count by just using a Map:

```
@GetMapping("/api")
public String getUsers(@RequestParam Map<String,String> allParams) {
        return "Parameters are " + allParams.entrySet();
}
```

Now we can read all Request Params from Map Object as Key and Value Pairs and we will utilize as per requirement.

When to use Query Param vs Path Variable:

As a best practice, almost of developers are recommending following way. If you want to identify a resource, you should use Path Variable. But if you want to sort or filter items on data, then you should use query parameters. So, for example you can define like this:

```
/users                                  # Fetch a list of users
/users?occupation=programmer&skill=java # Fetch a list of java programmers

/users/123                              # Fetch a user who has id 123
```

Swagger UI With SpringBoot Applications:

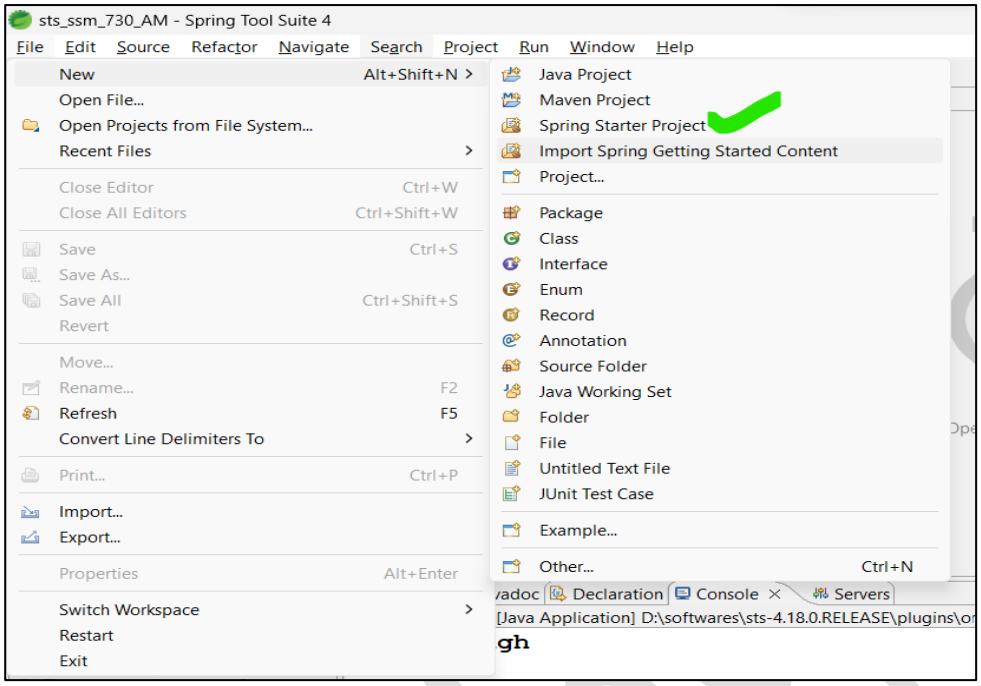
"Swagger is a set of rules, specifications and tools that help us document our APIs." "By using Swagger UI to expose our API's documentation, we can save significant time."

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

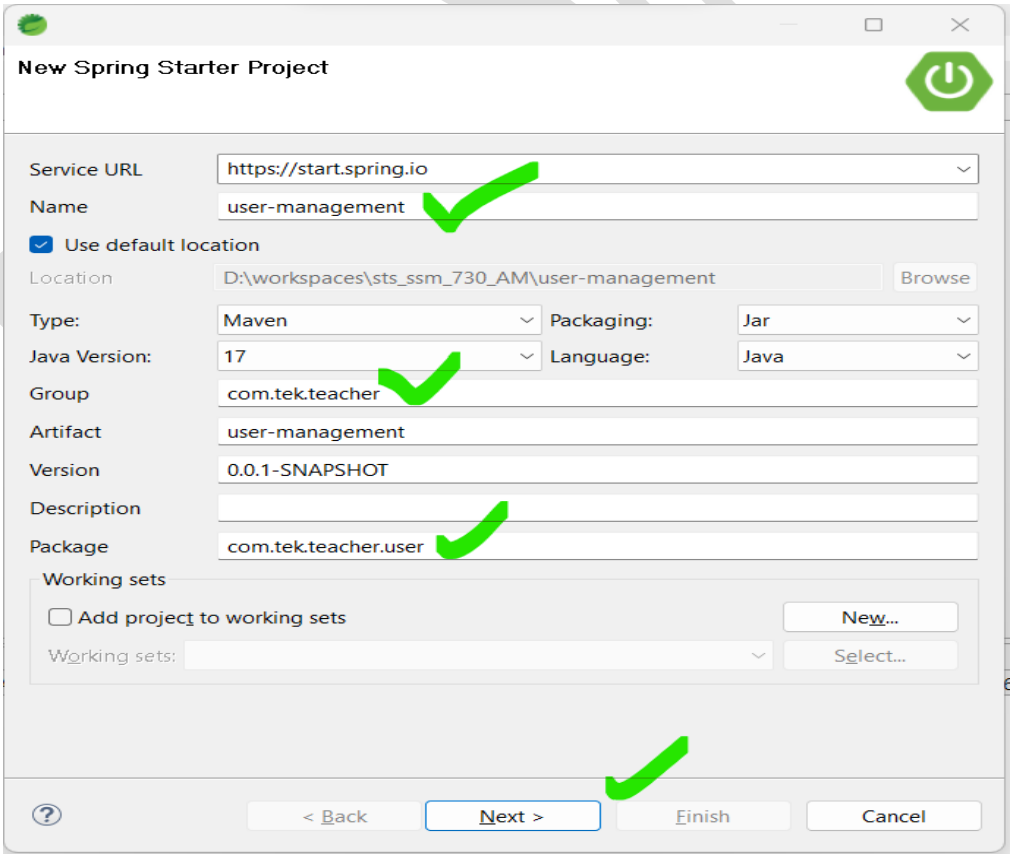
Swagger UI is one of the platform's attractive tools. In order for documentation to be useful, we will need it to be browsable and to be perfectly organized for easy access. It is for this reason that writing good documentation may be tedious and use a lot of the developers' time.

Create Spring Boot Web Application:

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

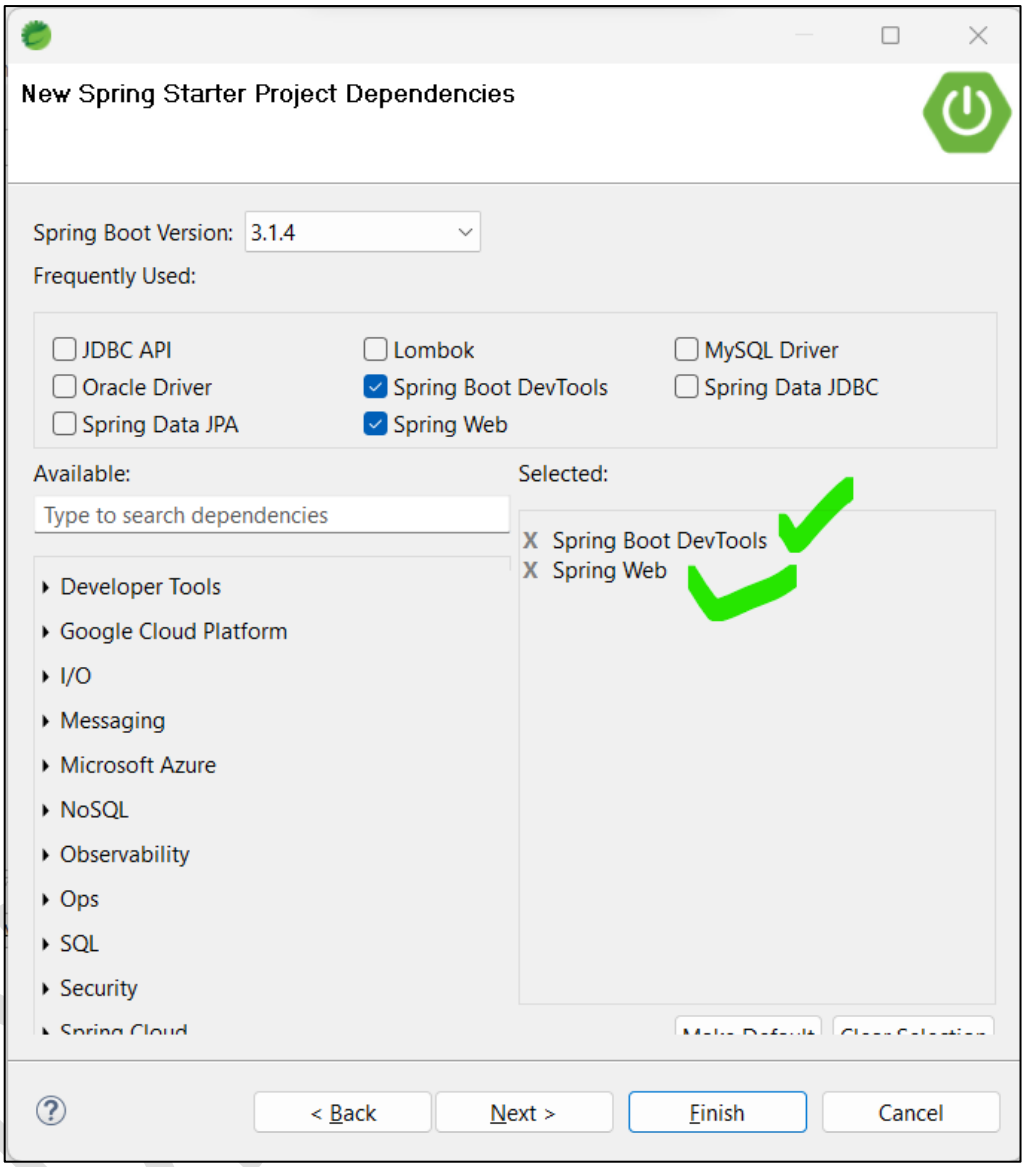


2. Fill All Project details as shown below and click on Next.



3. In Next Page, Add Spring Boot Modules/Starters as shown below and click on finish.

NOTE: Spring Web is mandatory, because REST Service Documentation we should do with Swagger.



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

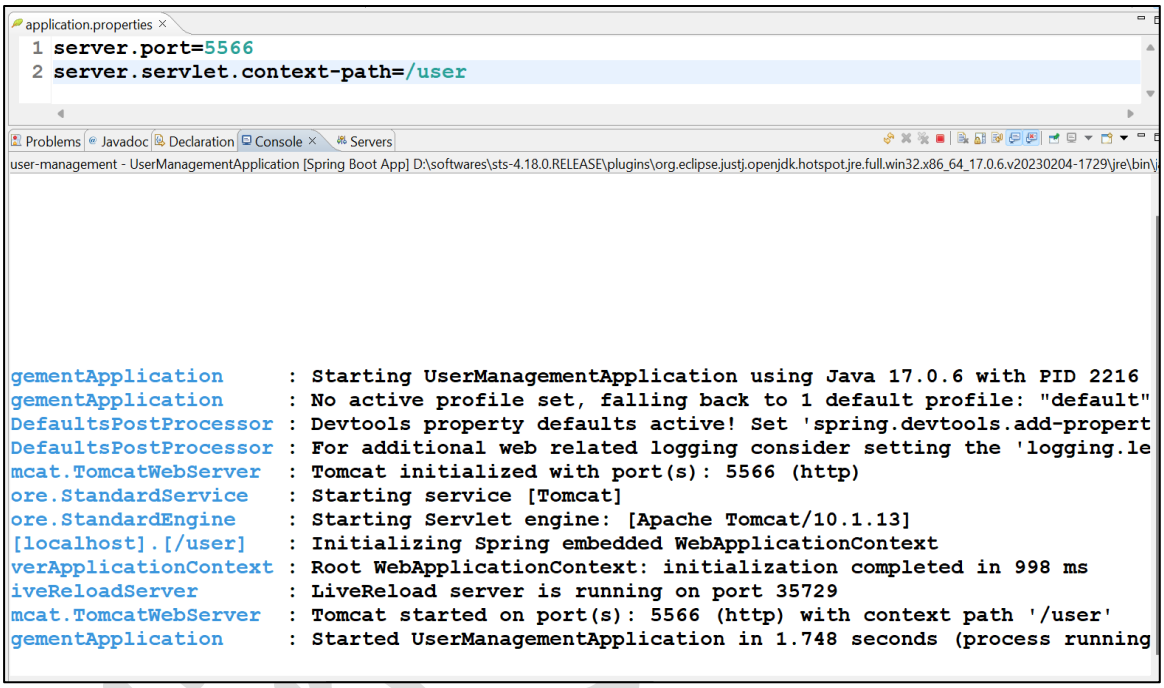
<!--Swagger/OpenAPI Documentation starter-->

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.1.0</version>
</dependency>
```

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

```
server.port=5566
server.servlet.context-path=/user
```

6. Now start your spring Boot application

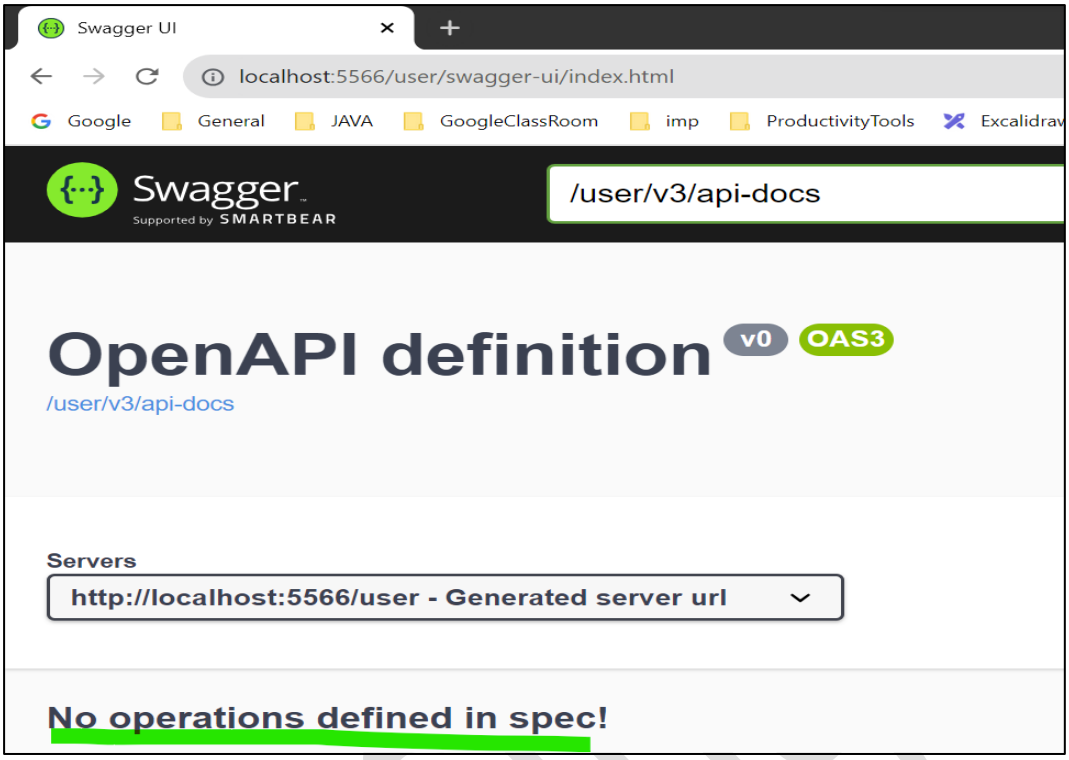


7. Enter URL in Browser for OpenAPI Swagger Documentation of Web services. Then you can Swagger UI page with empty Services List. Because Our application not contained any web services.

NOTE: The Swagger UI page will be available at **http://server:port/context-path/swagger-ui.html** and the OpenAPI description will be available at the following URL for JSON format: **http://server:port/context-path/v3/api-docs.** Documentation can be available in YAML format as well, on the following path : **/v3/api-docs.yaml**

server: The server name, hostname or IP
port: The server port
context-path: The context path of the application

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



Successfully Our SpringBoot Application Integrated with OpenAPI/Swagger documentation.

Adding REST Services to our application, to see Swagger API documentation:

- Now Create A controller Class in Our Application : `UserController.java`
- Adding REST Services inside controller class.

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

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

	@RequestMapping(method=RequestMethod.POST, path = "/create")
	public String createUser(@RequestBody CreateUserRequest request) {
		return "User Created Successfully ";
	}

	@RequestMapping(method=RequestMethod.GET, path = "/id/{userID}")
	public CreateUserResponse createUser(@PathVariable String userID) {
```

```
                System.out.println("Loading Values of user : " + userID);
                CreateUserResponse response = new CreateUserResponse();
                response.setEmail("Tekk.Teacher@gmail.com");
                response.setFirstName("Tek");
                response.setLastName("Teacher");

                return response;
        }
}
```

- Create Request and Response DTO classes: `CreateUserRequest.java`

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

public class CreateUserRequest {

        private String firstName;
        private String lastName;
        private String email;
        private String password;
        private long mobile;
        private float income;
        private String gender;

        public String getFirstName() {
                return firstName;
        }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
        public String getLastName() {
                return lastName;
        }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        public String getPassword() {
                return password;
        }
```

```
	public void setPassword(String password) {
		this.password = password;
	}
	public long getMobile() {
		return mobile;
	}
	public void setMobile(long mobile) {
		this.mobile = mobile;
	}
	public float getIncome() {
		return income;
	}
	public void setIncome(float income) {
		this.income = income;
	}
	public String getGender() {
		return gender;
	}
	public void setGender(String gender) {
		this.gender = gender;
	}
}
```

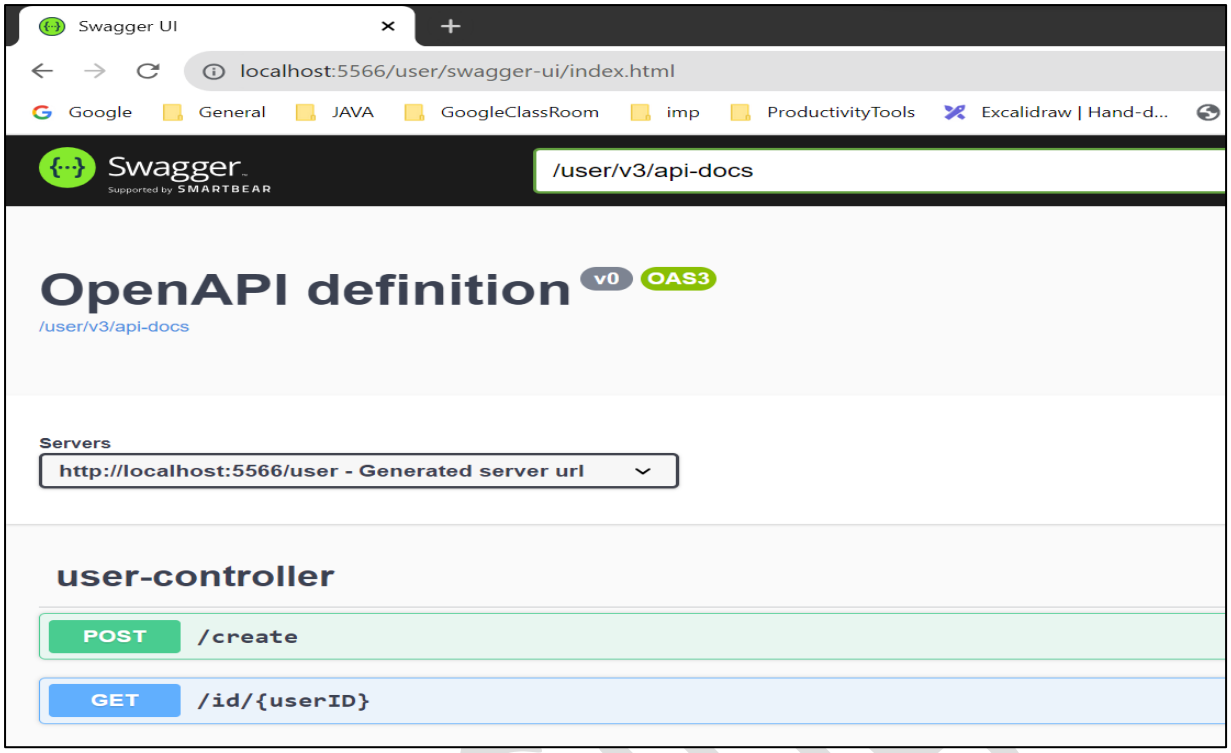
CreateUserResponse.java

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

public class CreateUserResponse {

	private String firstName;
	private String lastName;
	private String email;
	private long mobile;
	private float income;
	private String gender;

	public String getFirstName() {
		return firstName;
	}
	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}
	public String getLastName() {
		return lastName;
	}
	public void setLastName(String lastName) {
```

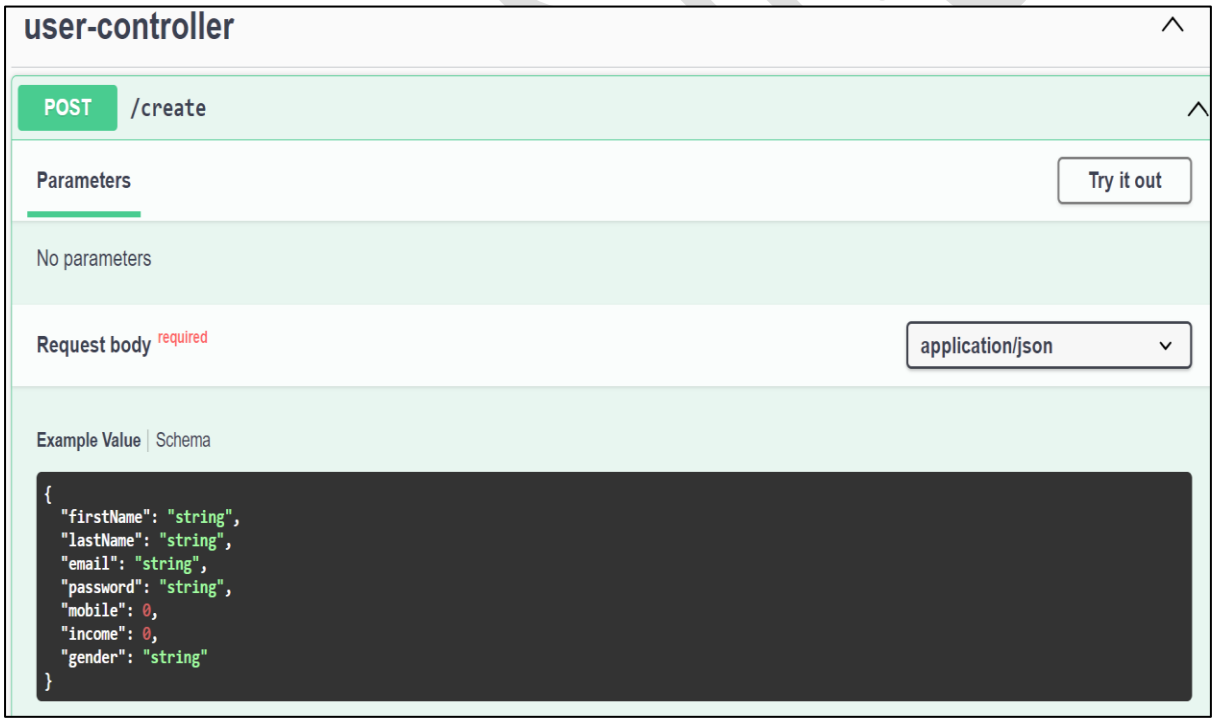
```
                this.lastName = lastName;
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        public long getMobile() {
                return mobile;
        }
        public void setMobile(long mobile) {
                this.mobile = mobile;
        }
        public float getIncome() {
                return income;
        }
        public void setIncome(float income) {
                this.income = income;
        }
        public String getGender() {
                return gender;
        }
        public void setGender(String gender) {
                this.gender = gender;
        }
}
```

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

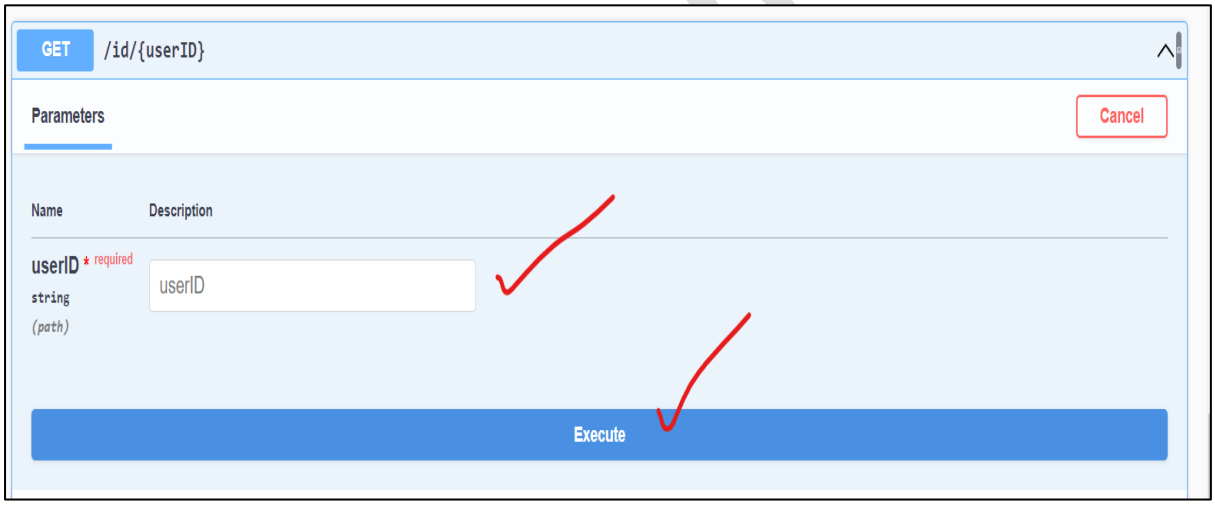
Swagger UI URL : http://localhost:5566/user/swagger-ui/index.html



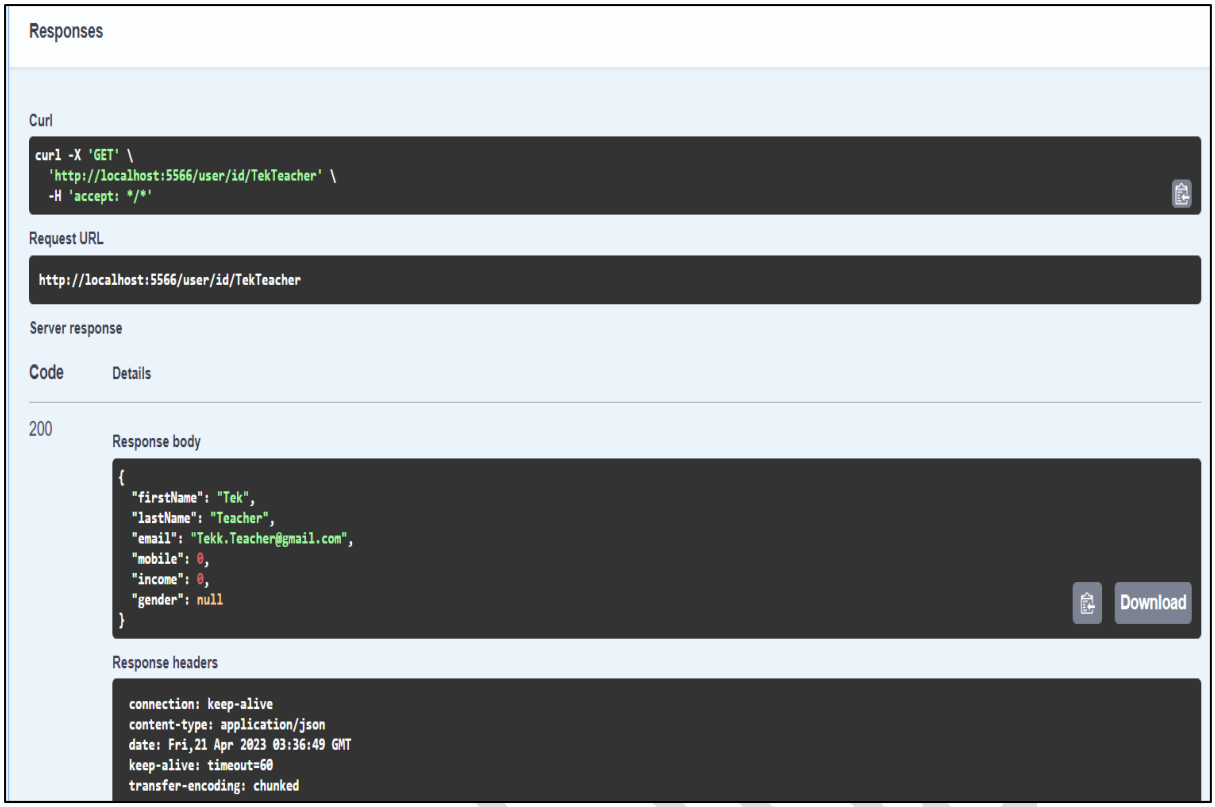
➢ **You can expand above both endpoints and look for payload details.**



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



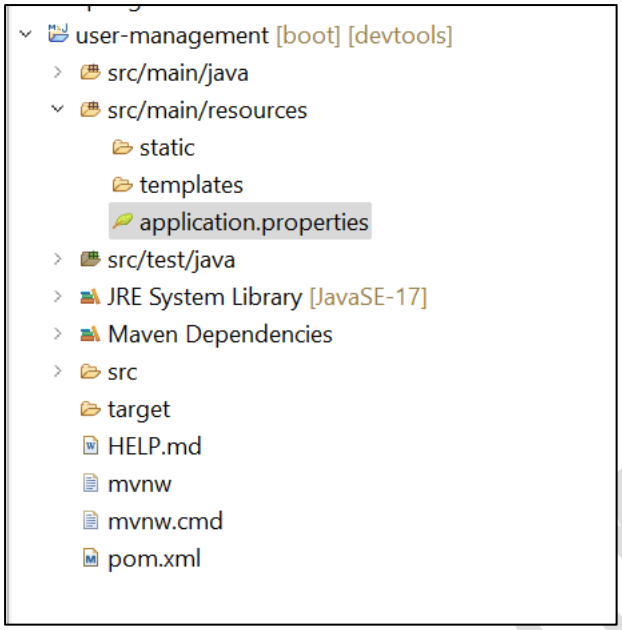
- After clicking on **Execute** You will get Response in response Section.



This is how we can integrate and use swagger UI API Documentation Tool with our applications to share all REST API data in UI format.

Spring Boot – application.yml / application.yaml File:

In Spring Boot, whenever we create a new Spring Boot Application in spring starter, or inside an IDE (Eclipse or STS) a file is located inside the src/main/resources folder named as application.properties file.



So in a spring boot application, **application.properties** file is used to write the application-related property into that file. This file contains the different configuration values which is required to run the application. The type of property like changing the port, database connectivity and many more.

In place of **properties** file, we can use **YAML/YML** based configuration files to achieve same behaviour.

What is this YAML/YML file?

YAML stands for **Yet Another Markup Language**. YAML is a data serialization language that is often used for writing configuration files. So YAML configuration file in Spring Boot provides a very convenient syntax for storing logging configurations in a hierarchical format. The application.properties file is not that readable. So most of the time developers choose application.yml file over application.properties file. YAML is a superset of JSON, and as such is a very convenient format for specifying hierarchical configuration data. YAML is more readable and it is good for the developers to read/write configuration files.

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

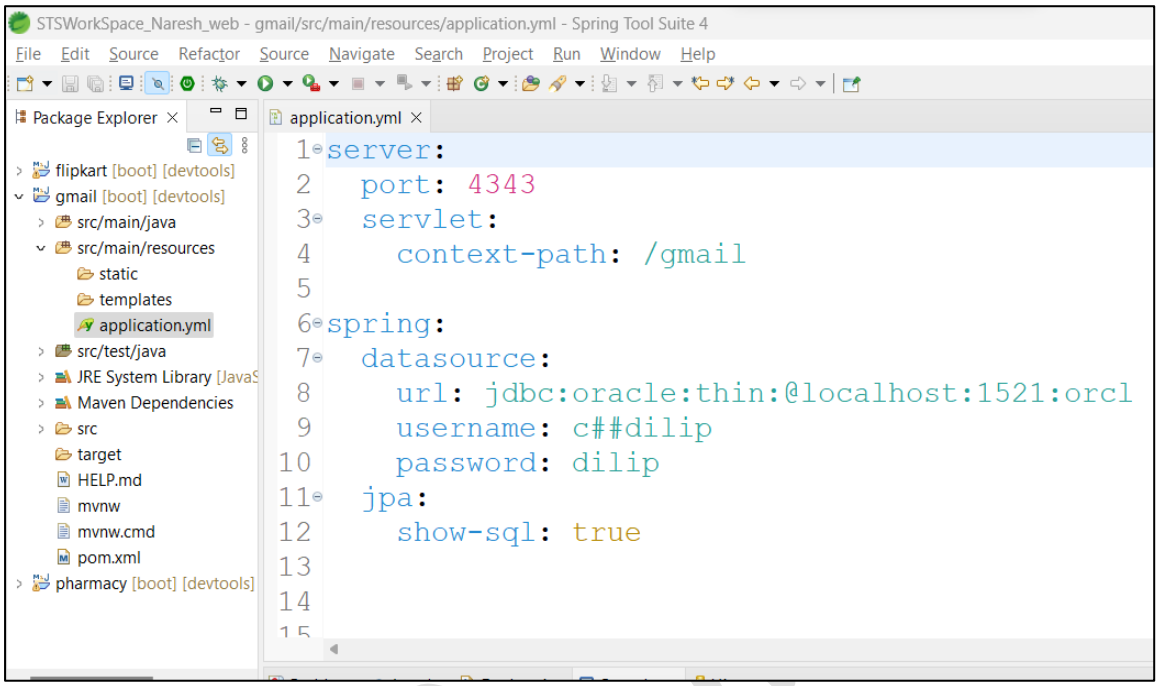
Now let's see some examples for better understanding :

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

```
server.port=4343
server.servlet.context-path=/gmail
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
```

```
spring.datasource.username=c##dilip
spring.datasource.password=dilip
spring.jpa.show-sql=true
```

If we written same properties content in application.yml:

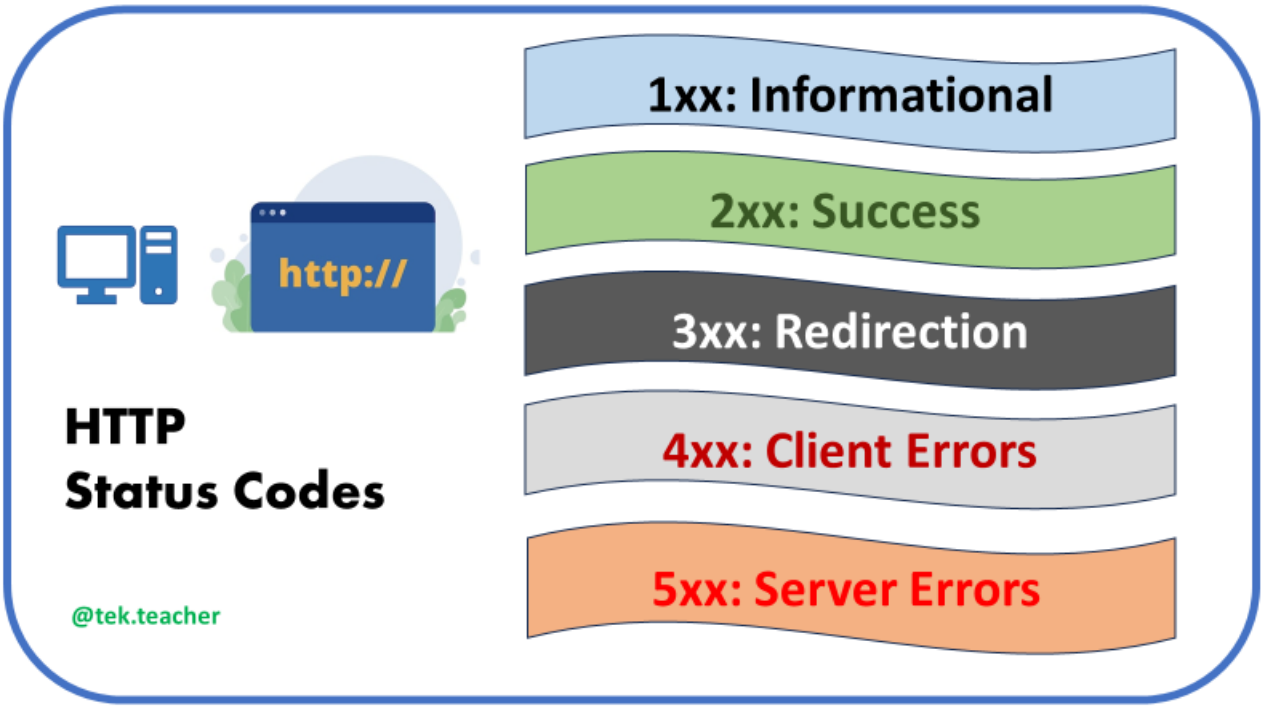


- Similarly, we can add all other Properties in same format always as per YAML scriprt rules and regulations.
- Now we can start our Spring Boot application as usual and continue development and testing activities.

HTTP status codes in building RESTful API's:

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

The status codes are divided into five categories.



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

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

1xx Informational:

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

100 Continue:

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

101 Switching Protocols:

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

102 Processing:

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

2xx Success:

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

200 OK:

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

201 Created:

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

204 No Content:

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

3xx Redirection:

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

304 Not Modified:

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

4xx Client Errors:

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

400 Bad Request:

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

401 Unauthorized:

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

403 Forbidden:

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

404 Not Found:

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

405 Method Not Allowed:

This code indicates that the HTTP method specified in the request is not allowed on the resource identified by the URI.

408 Request Timeout:

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

409 Conflict:

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

5xx Server Errors:

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

500 Internal Server Error:

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

501 (Not Implemented):

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

REST API Specific HTTP Status Codes:

Generally we will have likewise below scenarios and respective status codes in REST API services. For Example,

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

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

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

DELETE - Delete : **204 No Content**: Successfully Deleted Data
If not i.e. Resource Not Found Data

404 Not Found : Successfully Processed but Data Not available

Binding HTTP status codes and Response in Spring:

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

ResponseEntity:

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

This can be used as return value from an Controller URI method:

```
@RequestMapping("/handle")
public ResponseEntity<T> handle() {
    // Logic
    return new ResponseEntity<T>(ResponseData, ResponseHeaders, StatusCode);
}
```

Points to be noted:

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

Few Examples of Controller methods with ResponseEntity:

```
@RestController
public class NetBankingController {

@PostMapping("/create")
@ResponseStatus(value = HttpStatus.CREATED)     //Using Annotation
public String createAccount(@Valid @RequestBody AccountDetails accountDetails) {

        return "Created Net banking Account. Please Login.";
}

@PostMapping("/create/loan")    //Using Class
public ResponseEntity<String> createLoan(@Valid @RequestBody AccountDetails details) {

        return new ResponseEntity<>("Created Loan Account.", HttpStatus.CREATED);
```

```
}
}
```

Another Example:

```
@RestController
public class OrdersController {

@RequestMapping(value = "/product/order", method = RequestMethod.PUT)
public ResponseEntity<String> updateOrders(@RequestBody OrderUpdate request) {

        String result = orderService. updateOrders(request);
        if (result.equalsIgnoreCase("Order ID Not found")) {
                return new ResponseEntity<String>(result, HttpStatus.NOT_FOUND);
        }
                return new ResponseEntity<String>(result, HttpStatus.OK);
        }
}

@GetMapping("/orders")
public ResponseEntity<Order>  getOrders(@RequestParam("orderID") String orderID) {
        Order response = service.getOrders(orderID);
        return new ResponseEntity<Order>( response, HttpStatus.OK);
}
}
```

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

Headers in Spring MVC:

HTTP headers are part of the Hypertext Transfer Protocol (HTTP), which is the foundation of data communication on the World Wide Web. They are metadata or key-value pairs that provide additional information about an HTTP request or response. Headers are used to convey various aspects of the communication between a client (typically a web browser) and a server.

HTTP headers can be classified into two main categories: request headers and response headers.

Request Headers:

Request headers are included in an HTTP request sent by a client to a server. They provide information about the client's preferences, the type of data being sent, authentication credentials, and more. Some common request headers include:

- **User-Agent**: Contains information about the user agent (usually a web browser) making the request.
- **Accept**: Specifies the media types (content types) that the client can process.
- **Authorization**: Provides authentication information for accessing protected resources.
- **Cookie**: Sends previously stored cookies back to the server.
- **Content-Type**: Specifies the format of the data being sent in the request body.

Response Headers:

Response headers are included in an HTTP response sent by the server to the client. They convey information about the server's response, the content being sent, caching directives, and more. Some common response headers include:

- **Content-Type**: Specifies the format of the data in the response body.
- **Content-Length**: Specifies the size of the response body in bytes.
- **Set-Cookie**: Sets a cookie in the client's browser for managing state.

HTTP headers are important for various purposes, including negotiating content types, enabling authentication, handling caching, managing sessions, and more. They allow both clients and servers to exchange additional information beyond the basic request and response data. Proper understanding and usage of HTTP headers are essential for building efficient and secure web applications.

Spring MVC provides mechanisms to work with HTTP headers both in requests and responses. Here's how you can work with HTTP headers in Spring MVC.

Handling Request Headers:

Accessing Request Headers: Spring provides the **@RequestHeader** annotation that allows you to access specific request headers in your controller methods. You can use this annotation as a method parameter to extract header values.

In Spring Framework's MVC module, **@RequestHeader** is an annotation used to extract values from HTTP request headers and bind them to method parameters in your controller methods. This annotation is part of Spring's web framework and is commonly used to access and work with the values of specific request headers.

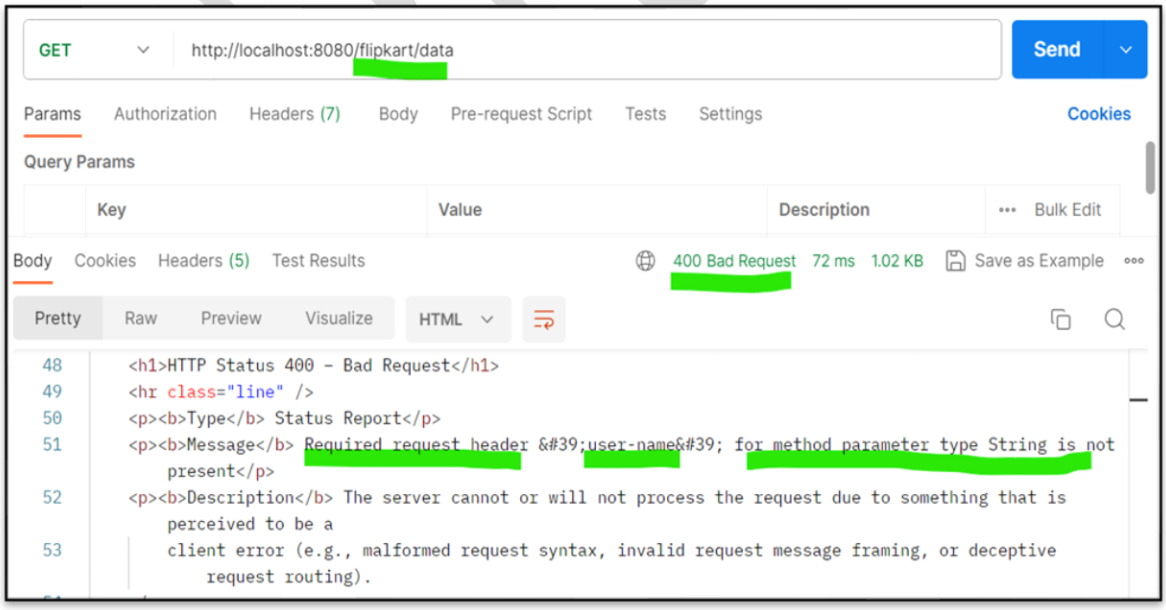
```
@GetMapping("/endpoint")
public ResponseEntity<String> handleRequest(@RequestHeader("Header-Name") String
                                                                    headerValue) {
   // Do something with the header and other values

}
```

Example: **Define a header user-name inside request:**

- Header and its Value should come from Client while they are triggering this endpoint.

```
package com.flipkart.controller;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;

@RestController
public class OrderController {
	//Header is Part of Request, i.e. Should be Come from Client Side.
	@GetMapping("/data")
	public String testHeaders(@RequestHeader("user-name") String userName) {
		return "Connected User : " + userName;
	}
}
```

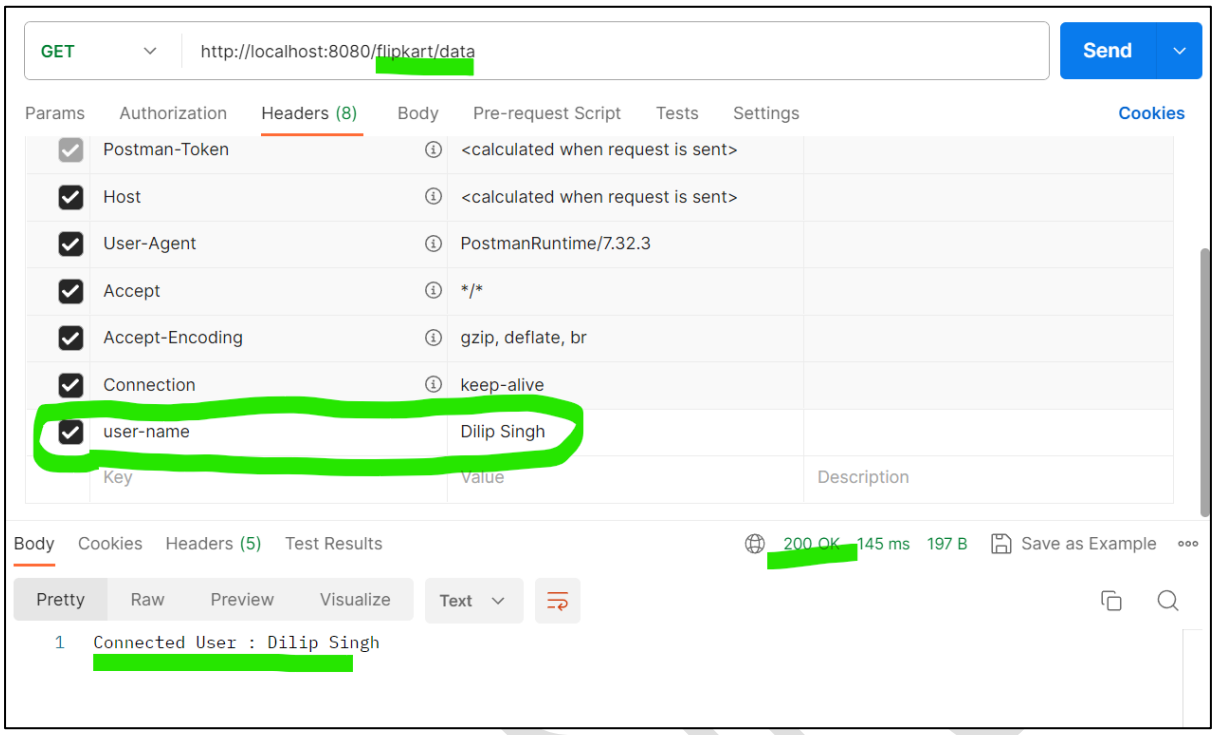
Testing: **Without Sending Header and Value from Client, Sending Request to Service.**



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

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

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



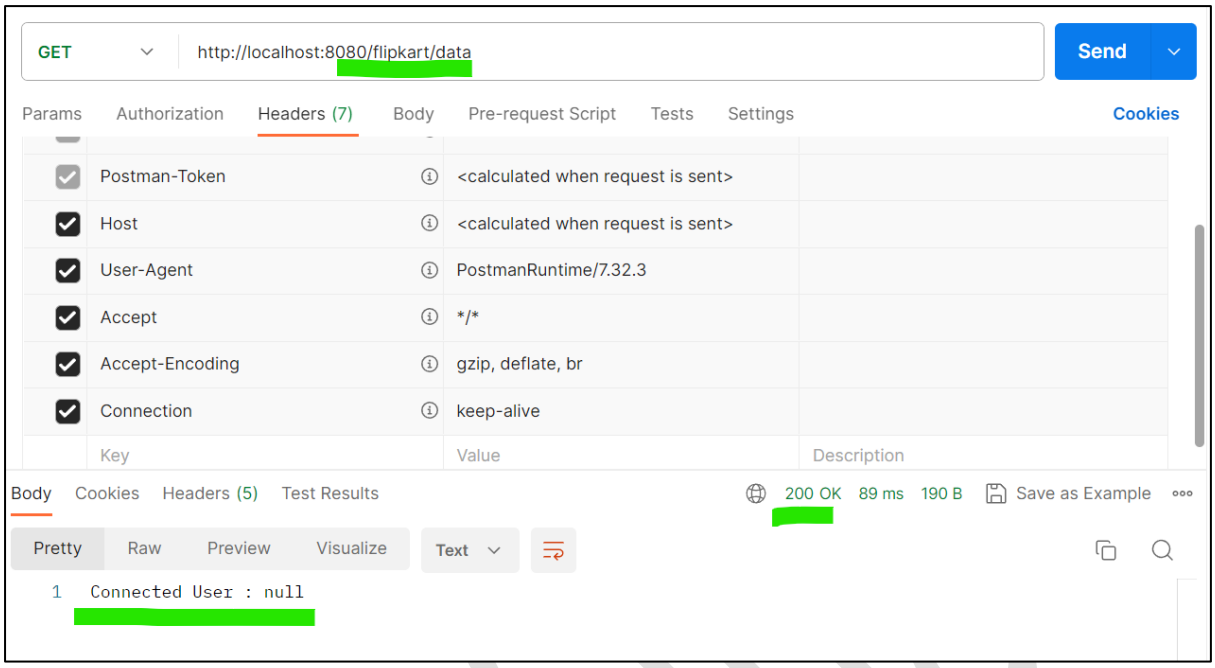
Optional Headers:

If we want to make Header as an Optional i.e. non mandatory. we have to add an attribute of **required** and Its value as **false**.

```
package com.flipkart.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

	//Header is Part of Request, i.e. Should be Come from Client Side.
	@GetMapping("/data")
	public String testHeaders(@RequestHeader(name = "user-name",
                                                 required = false)  String userName) {
		return "Connected User : " + userName;
	}
}
```

Testing:

- **No header Added, So Header value is null.**

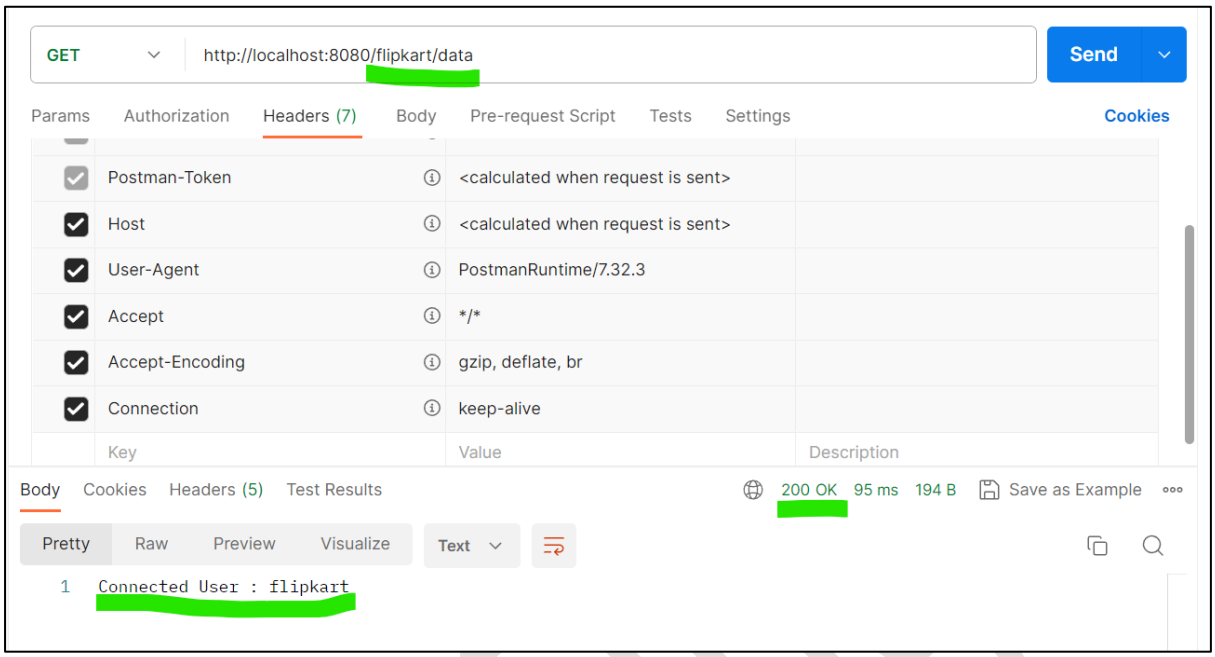


Default Value Of Header:

- **We can Set Header Default Value also in case if we are not getting it from Client. Add an attribute defaultValue and its value.**

```
package com.flipkart.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

	//Header is Part of Request, i.e. Should be Come from Client Side.
	@GetMapping("/data")
	public String testHeaders(@RequestHeader(name = "user-name", required = false,
						defaultValue = "flipkart") String userName) {
		return "Connected User : " + userName;
	}
}
```

Testing: **Without adding Header and its value, triggering Service. Default Value of Header user-name is flipkart is considered by Server as per implementation.**



Setting Response Headers:

In Spring MVC, response headers can be set using the **HttpServletResponse** object or the **ResponseEntity** class.

Here are some of the commonly used response headers in Spring MVC:

Content-Type: The MIME type of the response body.
Expires: The date and time after which the response should no longer be cached.
Last-Modified: The date and time when the resource was last modified.

The **HttpServletResponse** object is the standard way to set headers in a servlet-based application. To set a header using the **HttpServletResponse** object, you can use the **addHeader()** method.

For example:

```
HttpServletResponse response = request.getServletResponse();
response.addHeader("Content-Type", "application/json");
```

The **ResponseEntity** class is a more recent addition to Spring MVC. It provides a more concise way to set headers, as well as other features such as status codes and body content. To set a header using the **ResponseEntity** class, you can use the **headers()** method.

For example:

```
ResponseEntity<String> response = new ResponseEntity<>("Hello, world!", HttpStatus.OK);
response.headers().add("Content-Type", "application/json");
```

In another approach, We can create **HttpHeaders** instance and we can add multiple Headers and their values. After that, we can pass **HttpHeaders** instance to **ResponseEntity** Object.

HttpHeaders:

In Spring MVC, the **HttpHeaders** class is provided by the framework as a convenient way to manage HTTP headers in both request and response contexts. HttpHeaders is part of the **org.springframework.http** package, and it provides methods to add, retrieve, and manipulate HTTP headers. Here's how you can use the **HttpHeaders** class in Spring MVC:

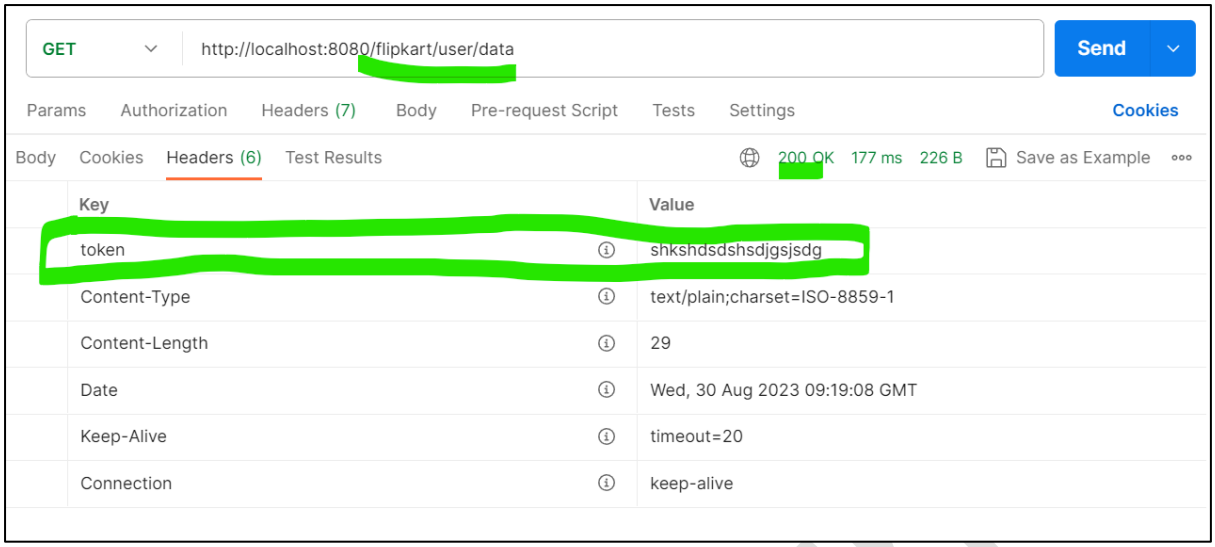
In a Response:

You can use **HttpHeaders** to set custom headers in the HTTP response. This is often done when you want to include specific headers in the response to provide additional information to the client.

Example: **Sending a Header and its value as part of response Body.**

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

@RestController
public class OrderController {

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

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

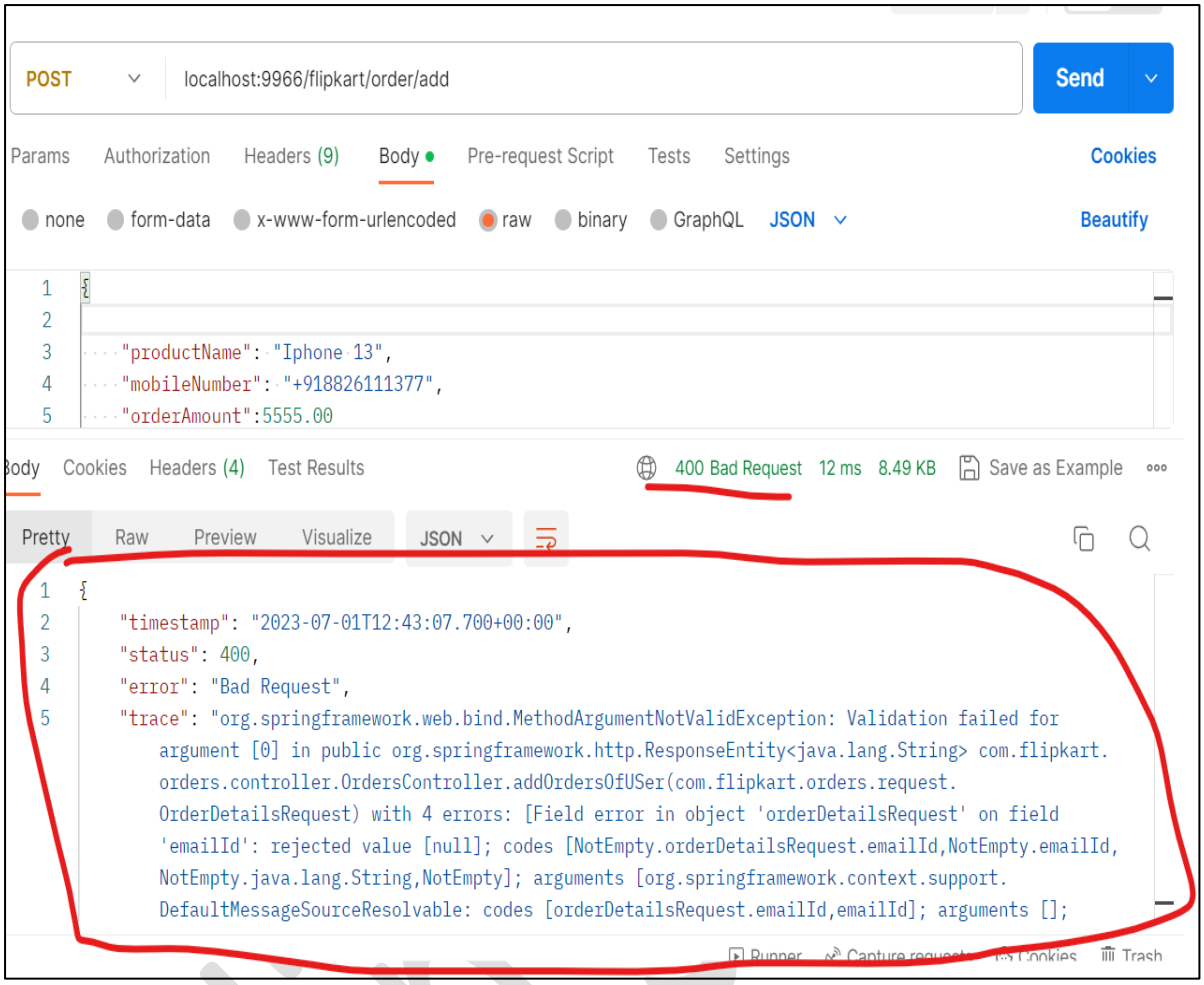


Exception Handling in Spring MVC Controllers:

What I have to do with Errors or Exceptions ?

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

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

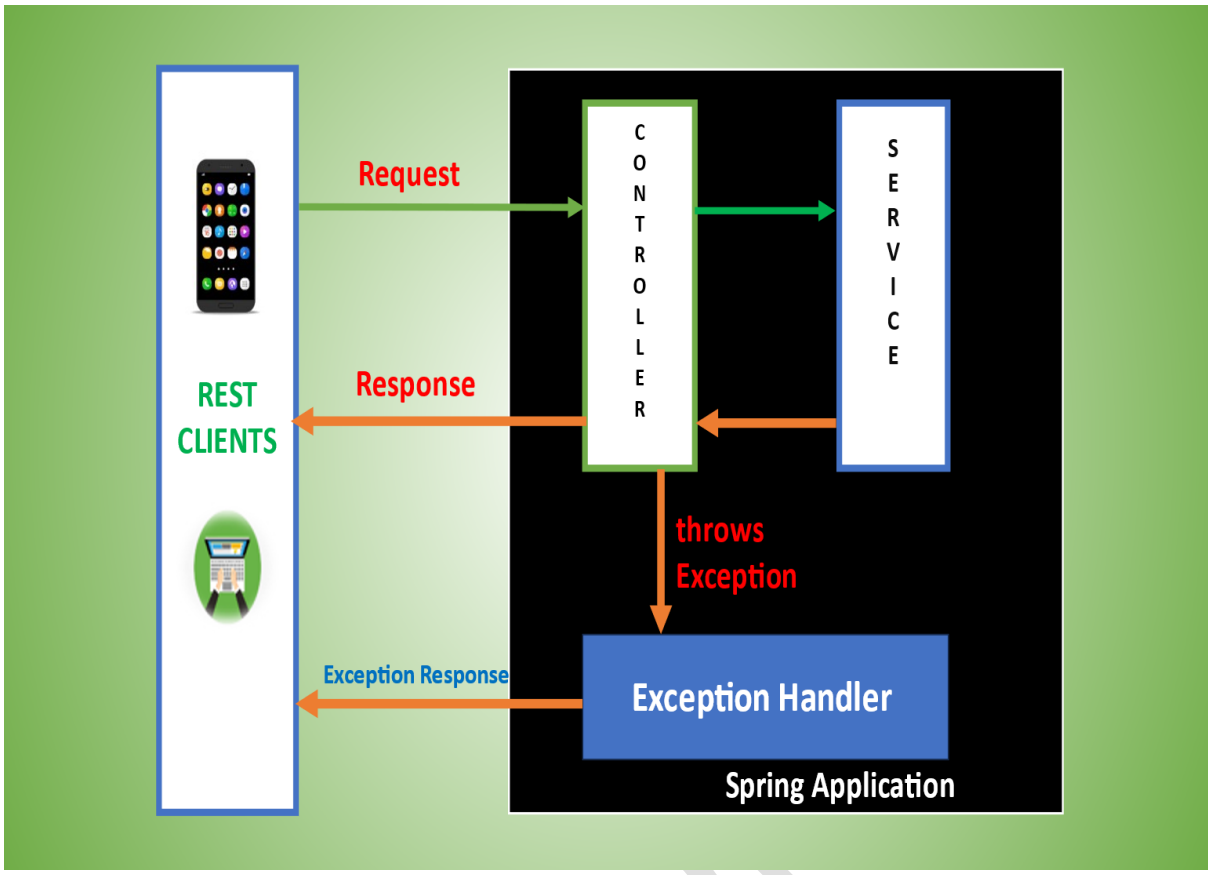


Note: Spring provided other ways as well to handle exceptions but controller advice and Exception handler will provide better way of exception handling.

@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.

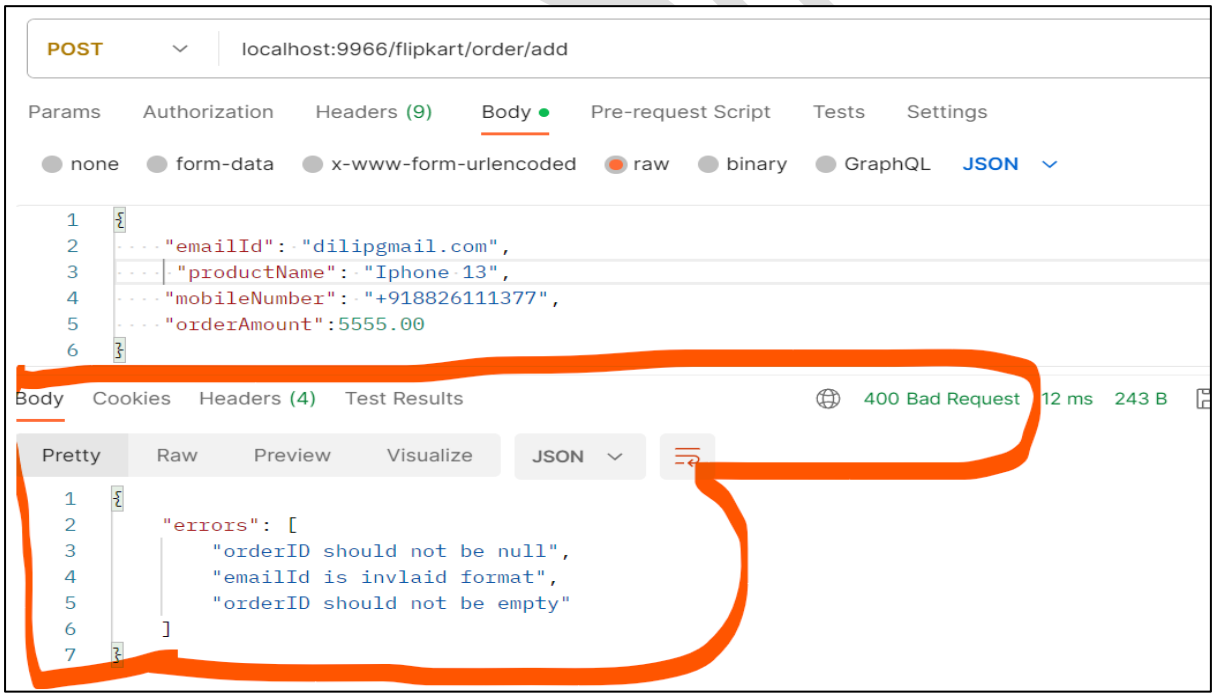
One thing to keep in our mind here is to match the exceptions declared with @ExceptionHandler to the exception used as the argument of the method.

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.**



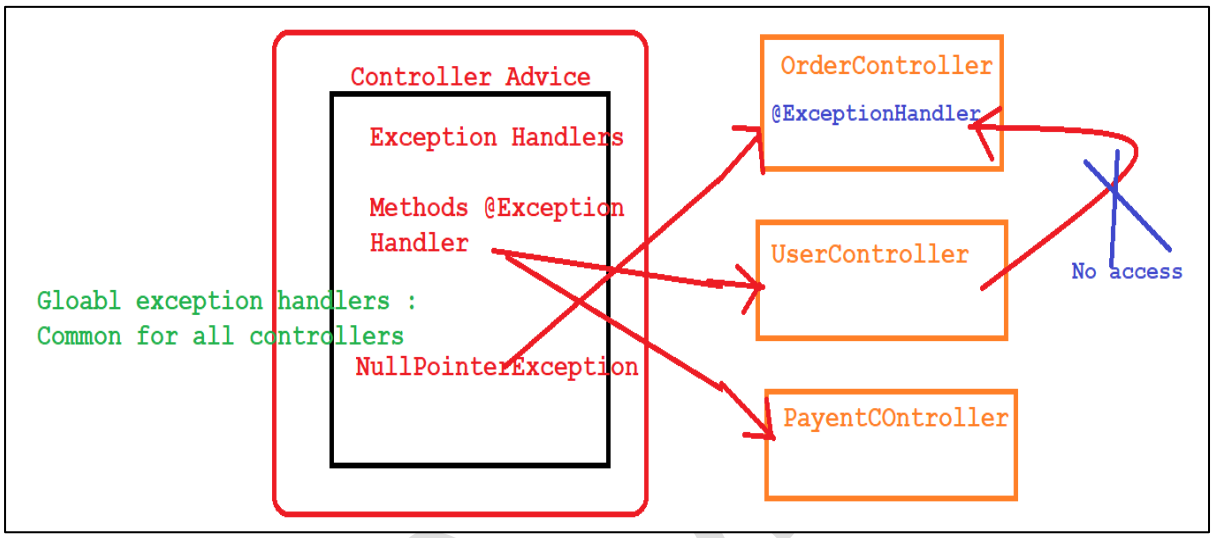
How it is working?

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.



@RequestMapping consumes and produces attributes:

Spring, by default, configures Jackson for parsing Java objects to JSON and converting JSON to Java objects as part of REST API request-response handling. When we want to support other Request and Response Data Formats in REST Services implementation, then we should define those respective data formats with help of **consumes** and **produces** attributes inside **@RequestMapping** annotation with endpoint methods. Same attributes and respective functionalities are applicable to shortcut annotations like **@GetMapping**, **@PostMapping** etc..

consumes:

Using a consumes attribute to narrow the mapping by the content type. **You can declare a shared consumes attribute at the class level i.e. applicable to all controller methods**. Unlike most other request-mapping attributes, **however, when used at the class level, a method-level consumes attribute overrides rather than extends the class-level declaration**.

The **consumes** attribute also supports negation expressions – for example, **!text/plain** means any content type other than **text/plain**.

MediaType class provides constants for commonly used media/content types, such as APPLICATION_JSON_VALUE and APPLICATION_XML_VALUE etc..

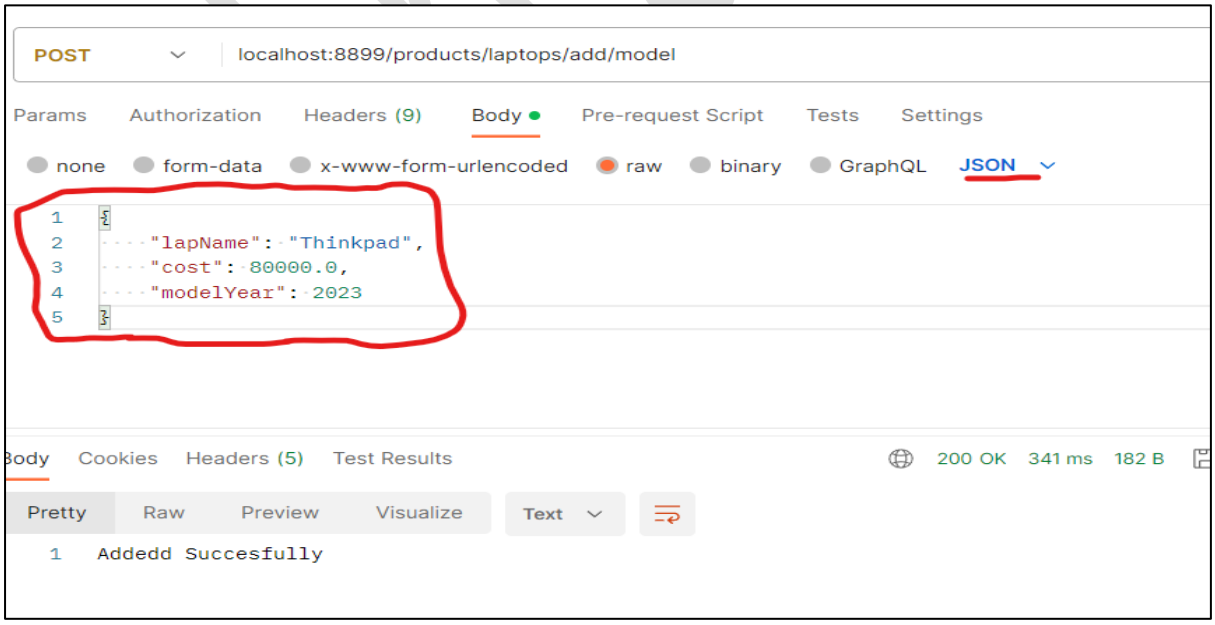
Now let's have an example, as below shown. Created an endpoint method, which accepts only JSON data Request by providing consumes ="application/json".

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

        return "Addedd Succesfully";
}
```

LaptopDetails.java : To Bind Request Body of JSON

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

        //Setters and Getters
}
```

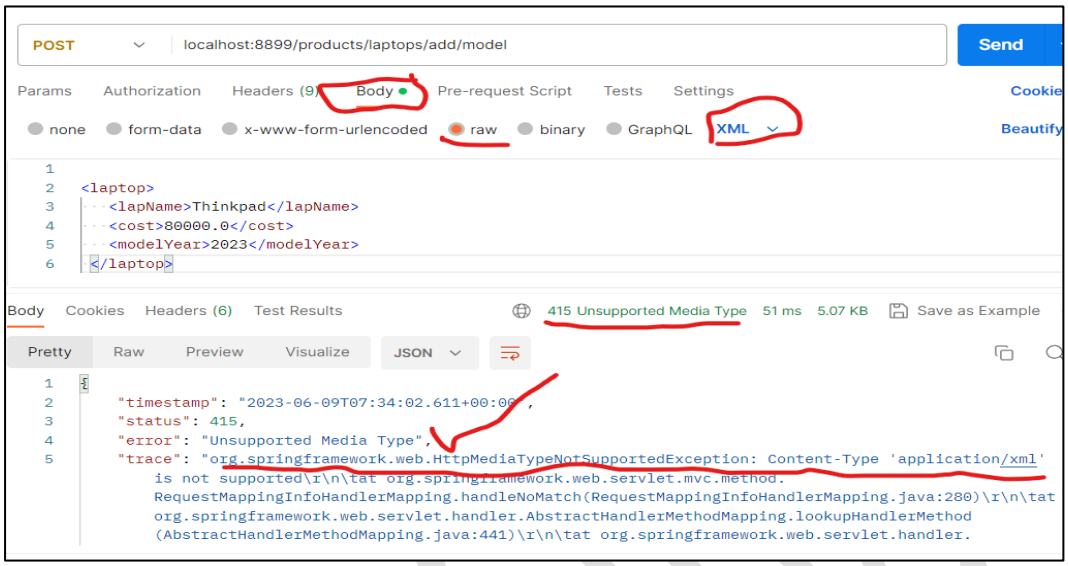
- Now Trigger Endpoint with JSON data in Request Body.



- **Now try to trigger same endpoint with XML Request Body.**

We will get an exception/error response as shown below.

```
"error": "Unsupported Media Type",
"trace": "org.springframework.web.HttpMediaTypeNotSupportedException: Content-Type 'application/xml'
```



Creating Endpoint which accepts only XML data Request Body:

To support XML request Body, we should follow below configurations/steps. Spring boot, by default, configures Jackson for parsing Java objects to JSON and converting JSON to Java objects as part of REST API request-response handling. To accept XML requests and send XML responses, there are two common approaches.

- Using Jackson XML Module
- Using JAXB Module

Start with adding Jackson's XML module by including the jackson-dataformat-xml dependency. Spring boot manages the library versions, so the following declaration is enough. Add below both dependencies in POM.xml file of application.

```
<dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

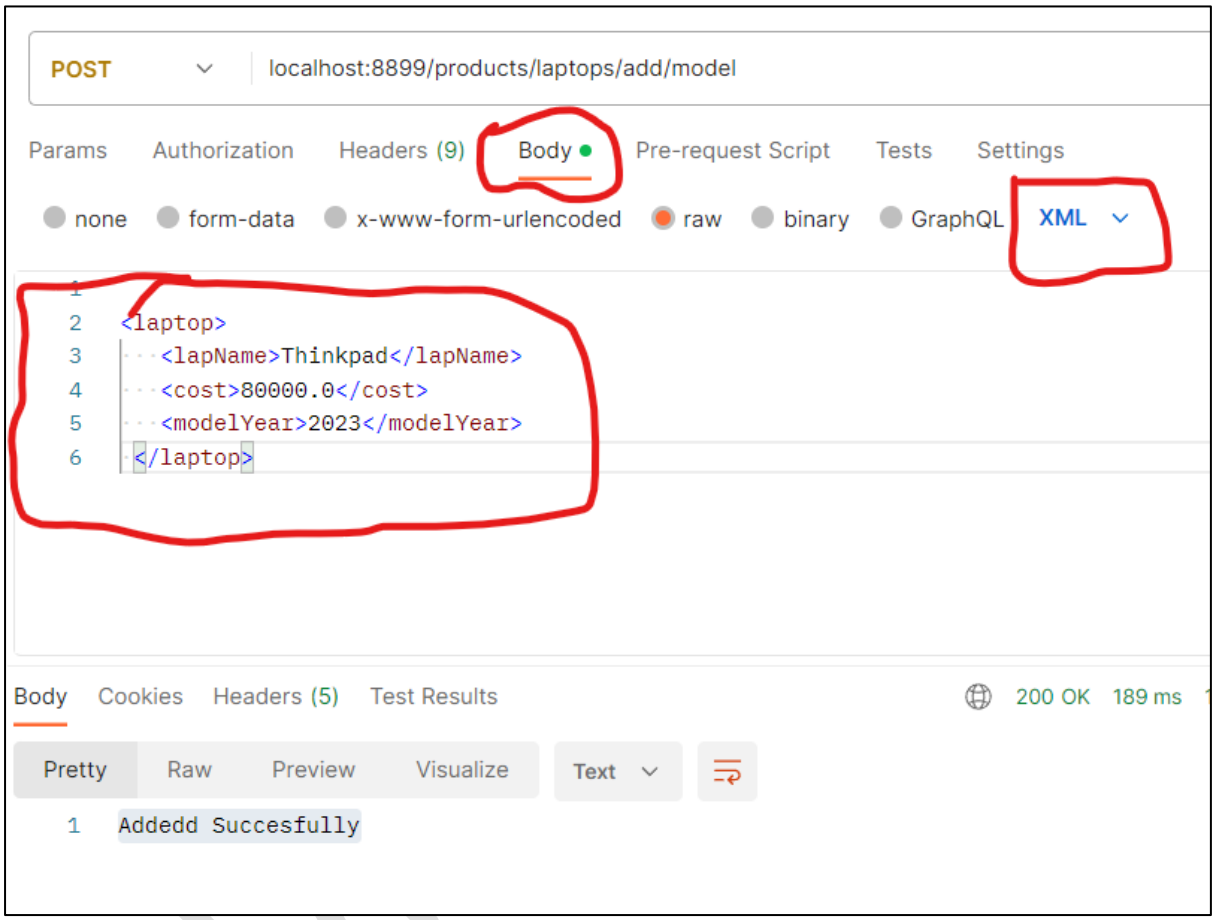
Now we can define and access REST API Services with XML data format.

➢ Creating a service which accepts only XML Request Body i.e. endpoint accepts now only XML request body but not JSON.

```
@RequestMapping(path = "/add/model", method = RequestMethod.POST,
                                                consumes ="application/xml")
```

```
public String addLaptopDetails(@RequestBody LaptopDetails details) {
                return "Added Successfully";
}
```

- Now Trigger Endpoint with XML Request data in Body.



- **Create endpoint which supports both JSON and XML Request Body.**

Below URI Request Mapping will support both XML and JSON Requests. We can pass multiple data types **consumes** attribute with array of values.

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

        return "Addedd Succesfully";
}
```

Spring Provided a class **MediaType** with Constant values of different Medi Types. We will use **MediaType** in **consumes** and **produces** attributes values.

consumes ={"application/json","application/xml"}
is equals to
consumes ={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE}

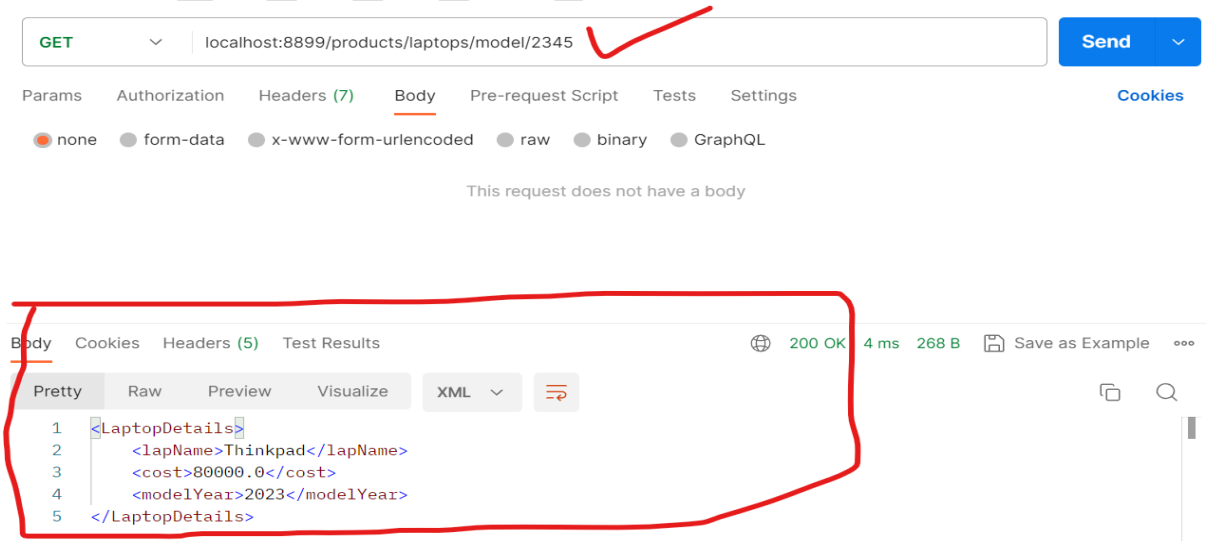
produces: with **produces** attributes, we can configure which type of Response data should be generated from Response object.

Endpoint Producing Only XML response:

Configure Request mapping with produces = MediaType.***APPLICATION_XML_VALUE***. So that now it will generate only XML response.

```
@RequestMapping(path = "/model/2345", method = RequestMethod.GET,
produces = MediaType.APPLICATION_XML_VALUE)
public LaptopDetails getLaptopDetails() {
        LaptopDetails lap = new LaptopDetails();
        lap.setCost(80000.00);
        lap.setLapName("Thinkpad");
        lap.setModelYear(2023);
        return lap;
}
```

Above endpoint generates only XML response for every incoming request.

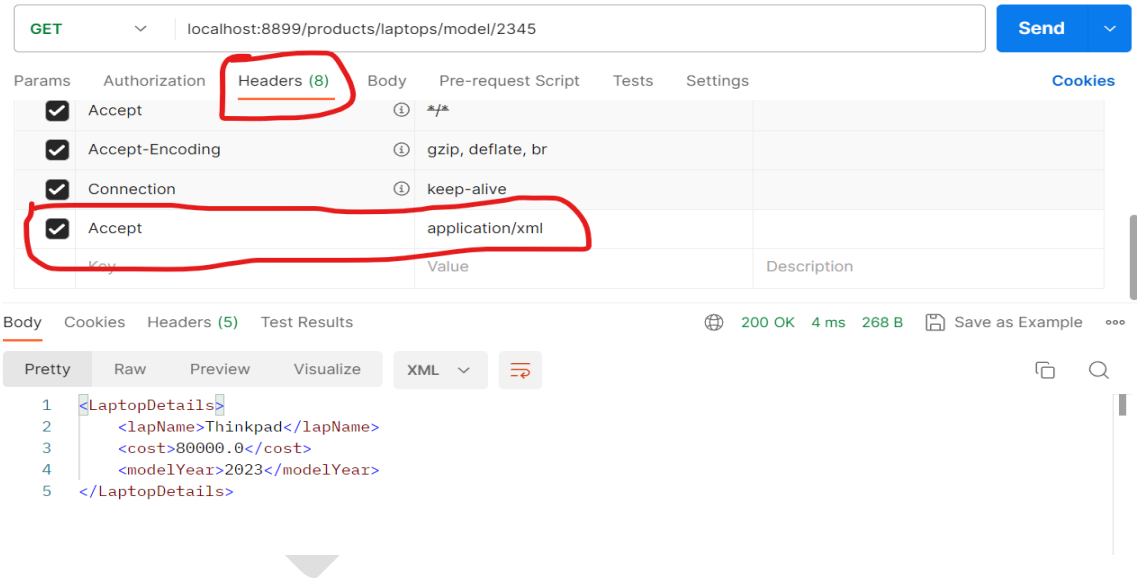


Creating an Endpoint Producing both JSON and XML response.

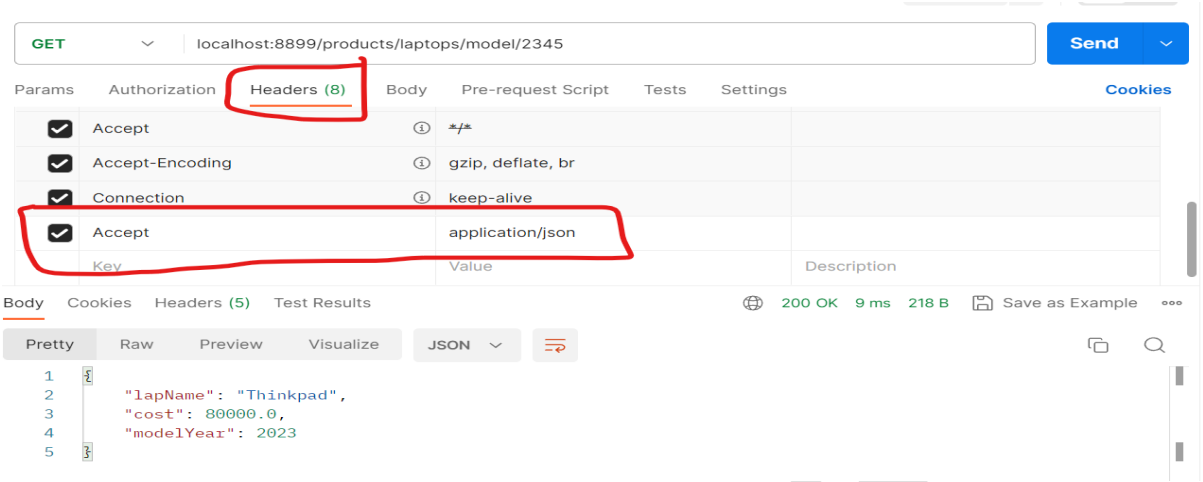
Configure Request mapping with **produces** attribute supporting both Media Types values i.e. array of values. So that now this endpoint generates either XML or JSON response depends on header **Accept** and its value. The HTTP **Accept** header is a request type header. The Accept header is used to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

```
@RequestMapping(path = "/model/2345", method = RequestMethod.GET, produces = {
MediaType.APPLICATION_XML_VALUE, MediaType.APPLICATION_JSON_VALUE})
	public LaptopDetails getLaptopDetails() {
		LaptopDetails lap = new LaptopDetails();
		lap.setCost(80000.00);
		lap.setLapName("Thinkpad");
		lap.setModelYear(2023);
		return lap;
	}
```

Request for XML response: Add Header **Accept** and value as **application/xml** as shown.



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



Producing and Consuming REST API services:

Producing REST Services:

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

Consuming REST Services:

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

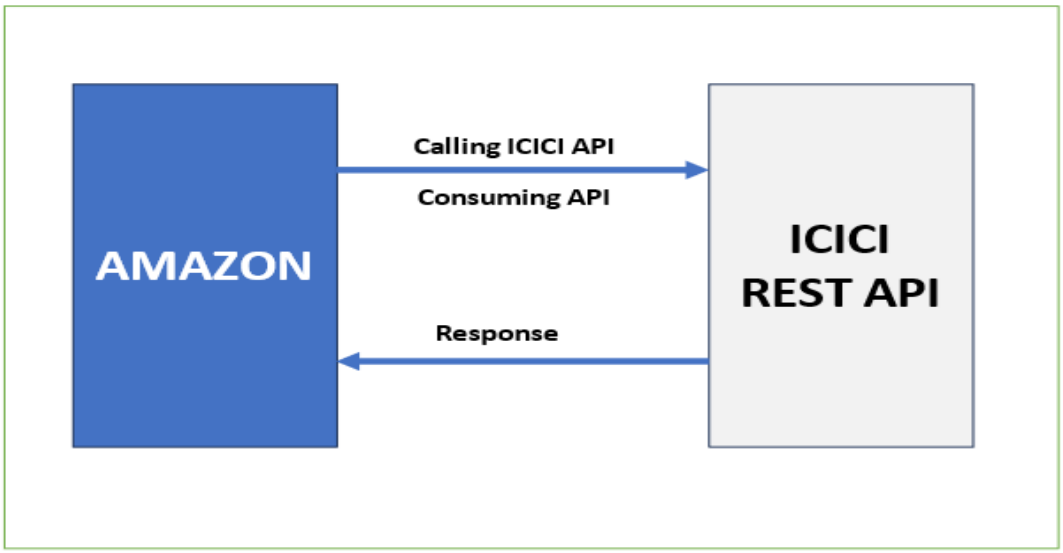
For Example,

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

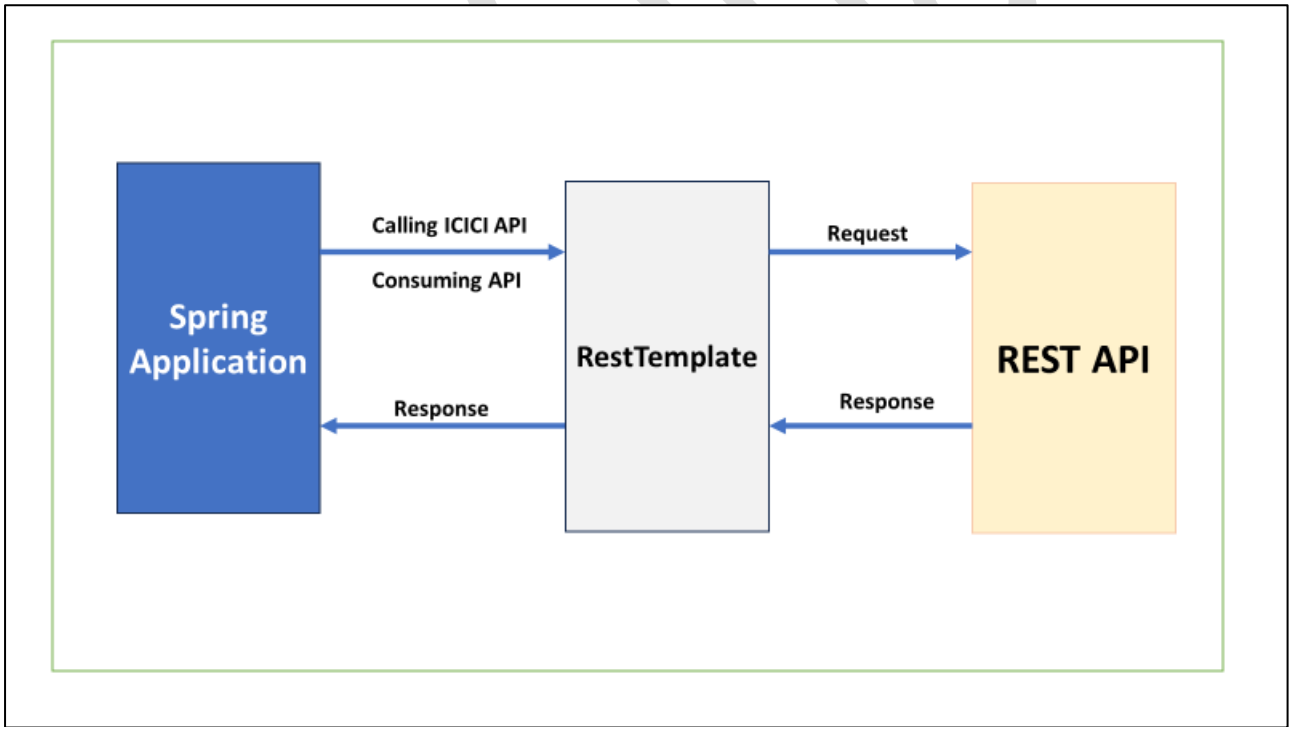
In This case:

Producer is : ICICI

Consumer is : Amazon



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



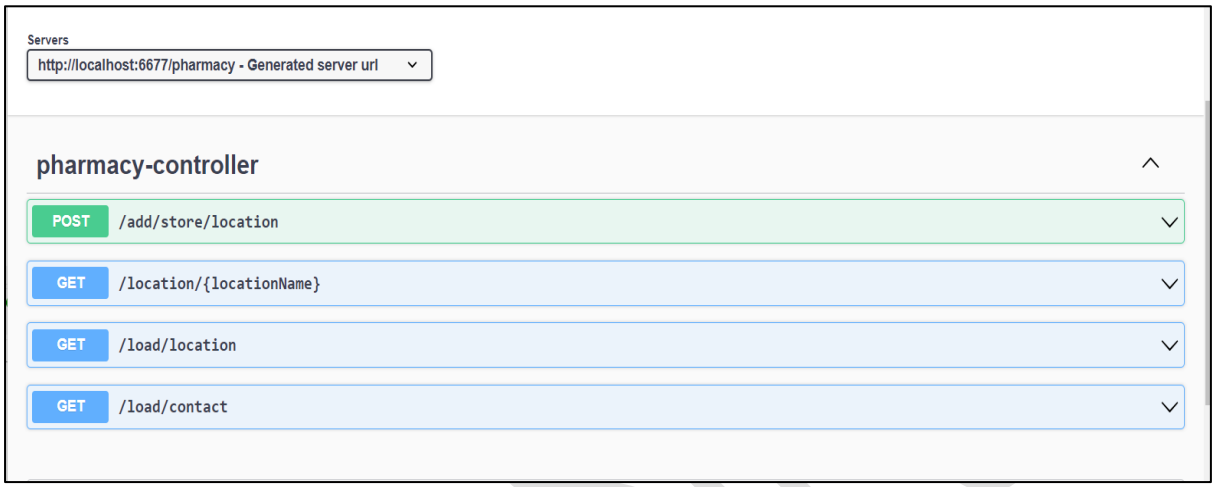
RestTemplate is used to create applications that consume RESTful Web Services. You can use the **exchange()** or specific http methods to consume the web services for all HTTP methods.

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

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

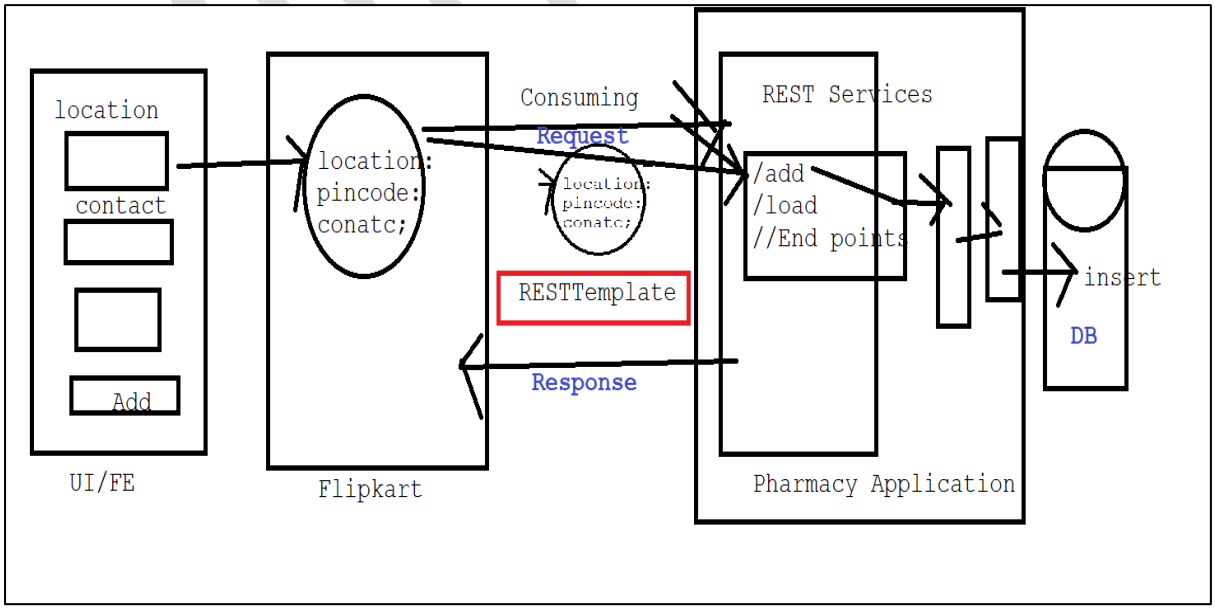
NOTE: Please makes sure other application running always to consume REST services.

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

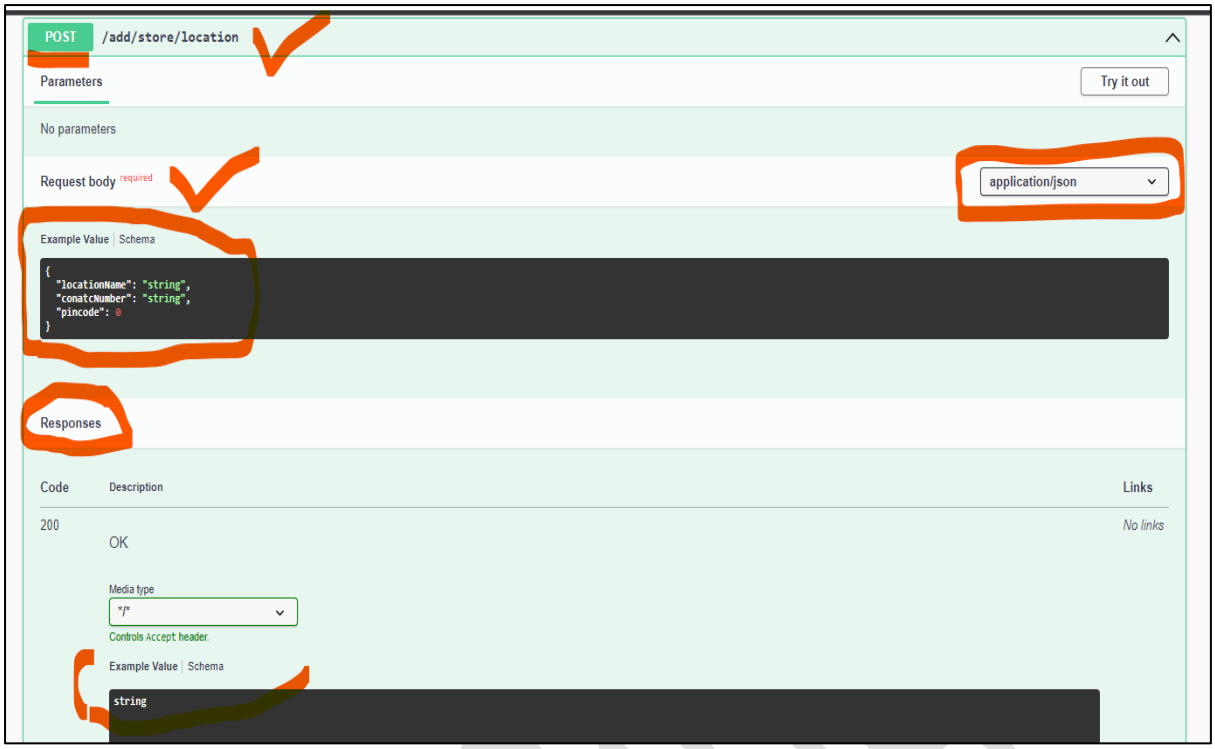


Consuming REST Services with Request Body:

Requirement : Now I want to call Rest service **/add/store/location** of pharmacy application from my Flipkart application.



Go to swagger and expand details of **/add/store/location** in swagger documentation.



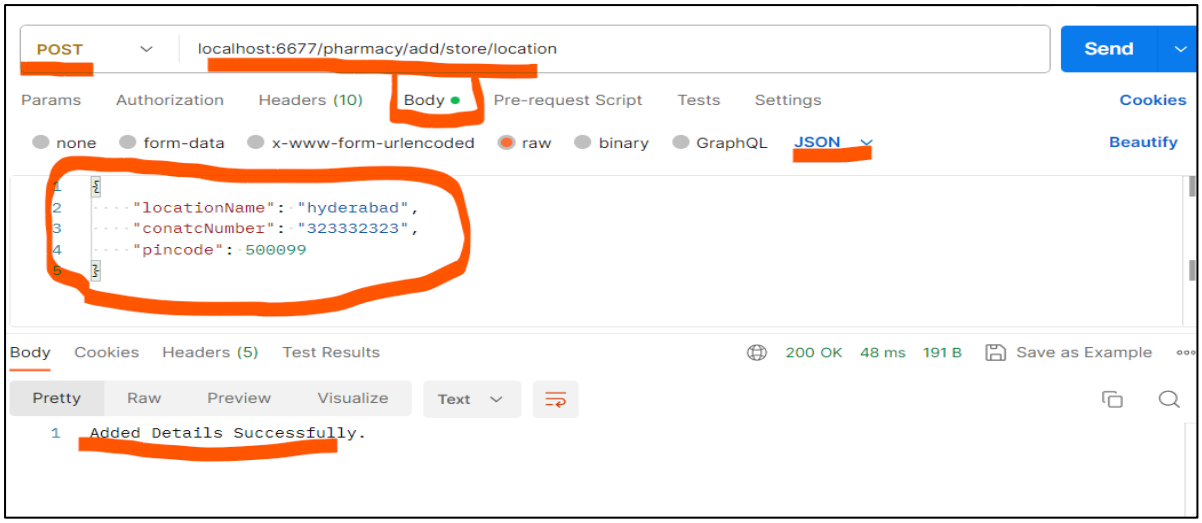
From above swagger snap, we should understand below points for that API call.

1. URL : http://localhost:6677/pharmacy/add/store/location
2. HTPP method: POST
3. Request Body Should contain below payload structure

```
{
   "locationName": "hyderabad",
   "conatcNumber": "323332323",
   "pincode": 500099
}
```

4. Response Receiving as String Format.

Same we can see in Postman as shown below.



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

Now assume we are receiving data from UI/Frontend to Flipkart application and that data we are transferring to Pharmacy API with help of **RestTemplate**.

NOTE : All code changes will happen only in flipkart application.

```
@RestController
@RequestMapping("/pharmacy")
public class PharmacyController {

	@Autowired
	PharmacyService pharmacyService;

	@PostMapping("/add/location")
	public String addPharmacyDetails(@RequestBody PharmacyLocation request) {
		return pharmacyService.addPharmacyDetails(request);
	}
}
```

- Now in Service class, we should write logic of integrating Pharmacy endpoint for adding store details as per swagger notes.
- Crate a POJO class which is equal to JOSN Request payload of Pharmacy API call.

PharmacyData.java : This Object will be used as Request Body

```
public class PharmacyData {

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

        //setters and getters
}
```

HttpEntity:

HttpEntity class is used to represent an HTTP request or response entity. It encapsulates/binds the HTTP message's headers and body. You can use **HttpEntity** to customize the headers and body of the HTTP request before sending it using **RestTemplate**. It provides more control and flexibility over the request or response compared to simpler methods like getForEntity(), postForObject(), etc.

Here's how you can use **HttpEntity** in **RestTemplate**:

- Now In service layer, Please map data from controller layer to API request body class.

```
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpMethod;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
import com.flipkart.dto.PharmacyData;
import com.flipkart.pharmacy.request.PharmacyLocation;

@Service
public class PharmacyService {

public String addPharmacyDetails(PharmacyLocation location) {

//complete URL of pharmacy endpoint.
String url = "http://localhost:6677/pharmacy/add/store/location";
```

```
        //Mapping from flipkart request object to JSON payload Object of class i.e. PharmacyData
        // Java Object which should be aligned to Pharmacy POST end point Request body.
        PharmacyData data = new PharmacyData();
        data.setConatcNumber(location.getContact());
        data.setLocationName(location.getLocation());
        data.setPincode(location.getPincode());

        // converting our java object to HttpEntity : i.e. Request Body
      HttpEntity<PharmacyData> body = new HttpEntity<PharmacyData>(data);
       RestTemplate restTemplate = new RestTemplate();
       return   restTemplate.exchange(url, HttpMethod.POST, body, String.class).getBody();
}
}
```

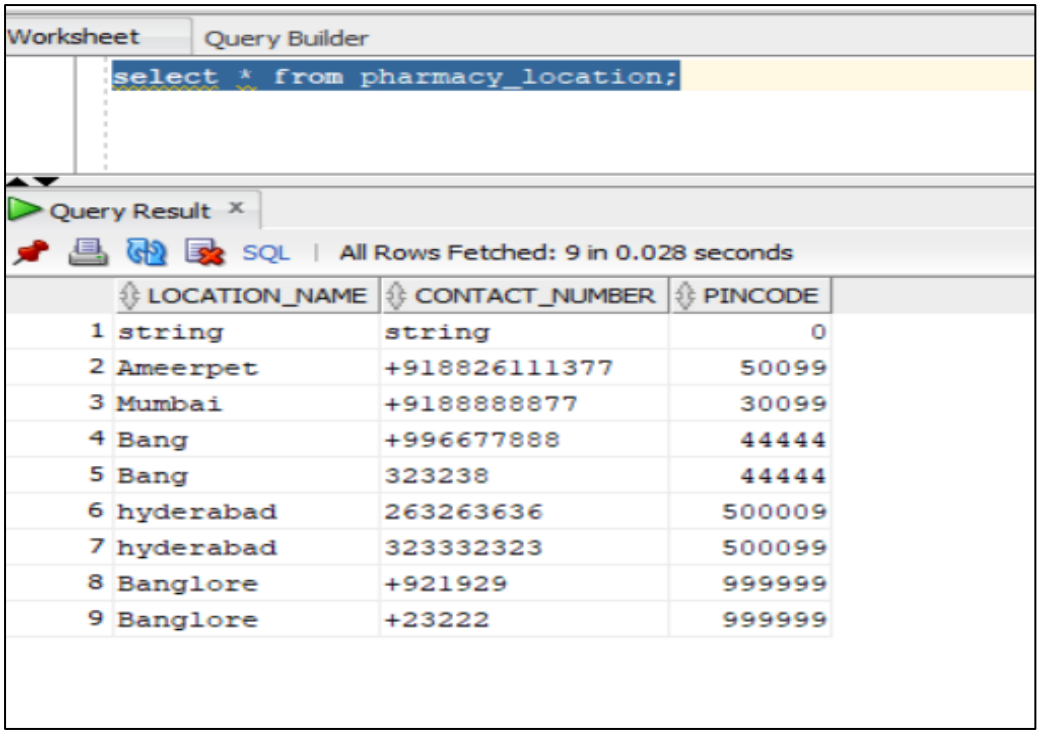
- Now Test it from Postman and check pharmacy API call triggered or not i.e. check data is inserted in DB or not from pharmacy application.

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

- **Now create Request body as per our controller request body class.**

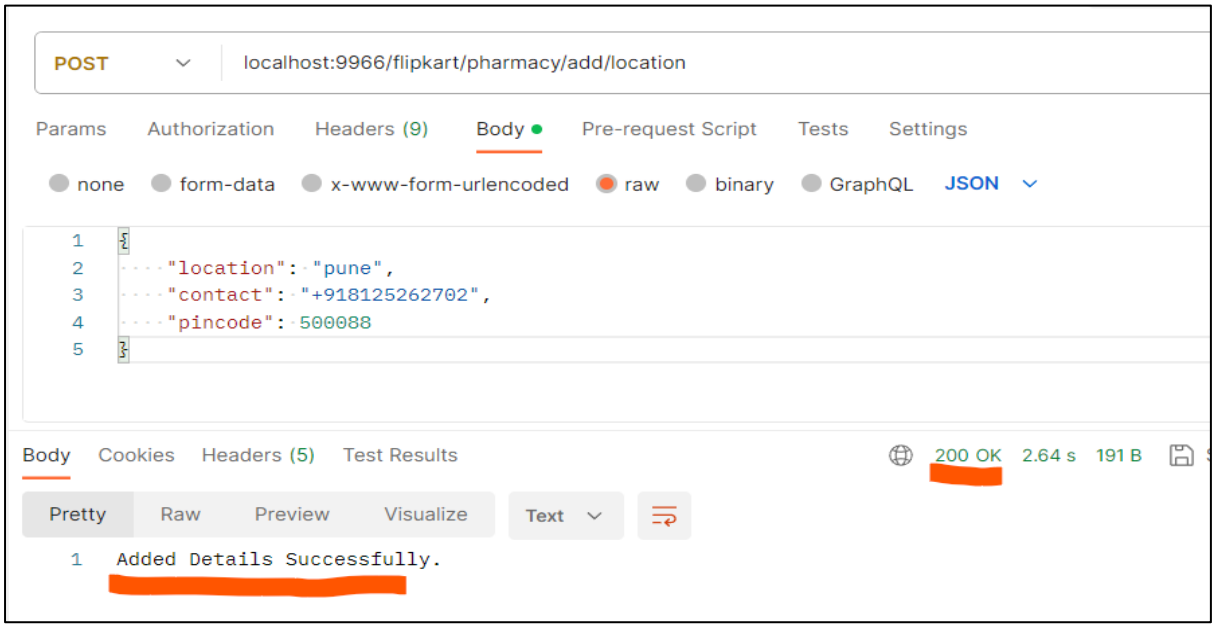
```
{
   "location": "pune",
   "contact": "+918125262702",
   "pincode": 500088
}
```

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

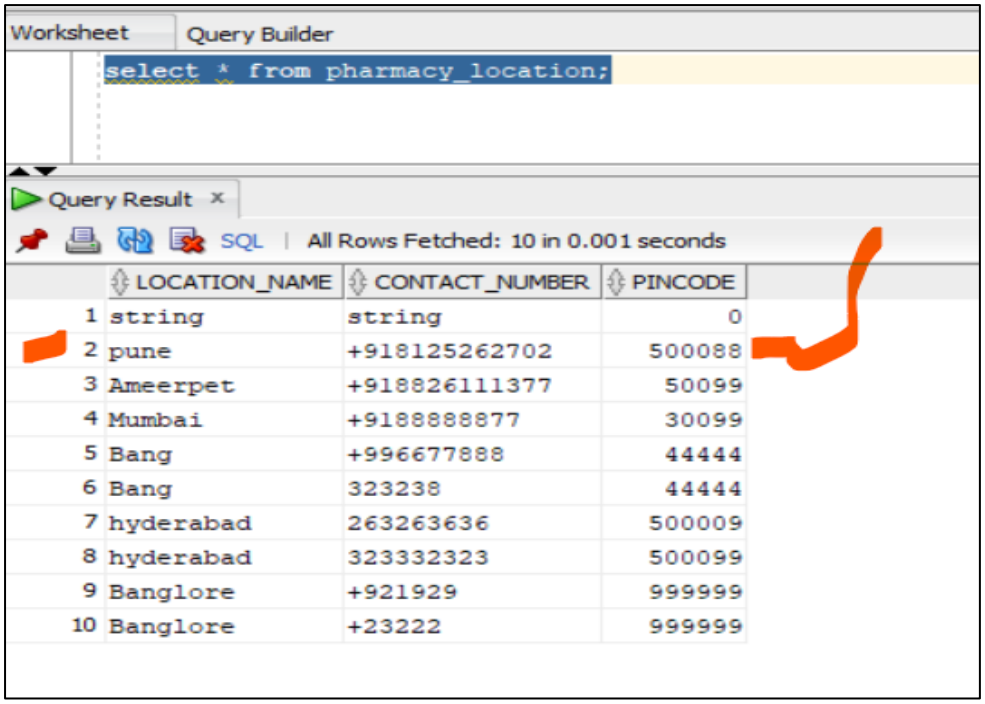


	LOCATION_NAME	CONTACT_NUMBER	PINCODE
1	string	string	0
2	Ameerpet	+918826111377	50099
3	Mumbai	+9188888877	30099
4	Bang	+996677888	44444
5	Bang	323238	44444
6	hyderabad	263263636	500009
7	hyderabad	323332323	500099
8	Banglore	+921929	999999
9	Banglore	+23222	999999

- **Now from postman send request as per flipkart controller method.**



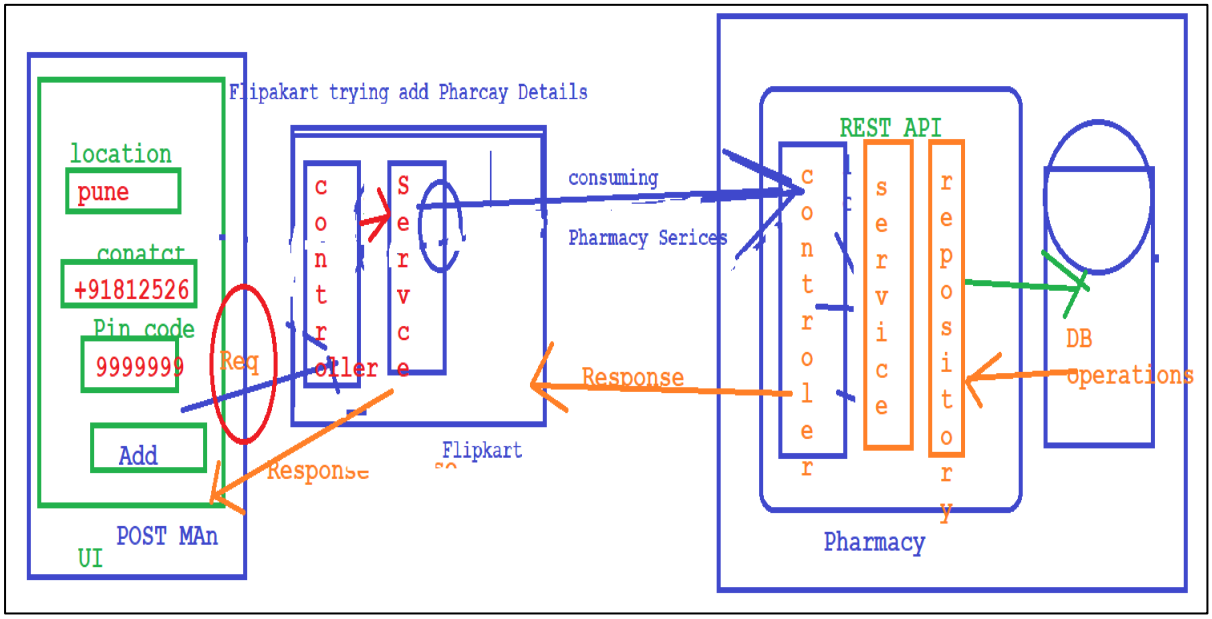
- Request executed successfully and you got response from Pharmacy API of post REST API call what we integrated. Verify In Database record inserted or not. It's inserted.



	LOCATION_NAME	CONTACT_NUMBER	PINCODE
1	string	string	0
2	pune	+918125262702	500088
3	Ameerpet	+918826111377	50099
4	Mumbai	+9188888877	30099
5	Bang	+996677888	44444
6	Bang	323238	44444
7	hyderabad	263263636	500009
8	hyderabad	323332323	500099
9	Banglore	+921929	999999
10	Banglore	+23222	999999

Internal Execution/Workflow:

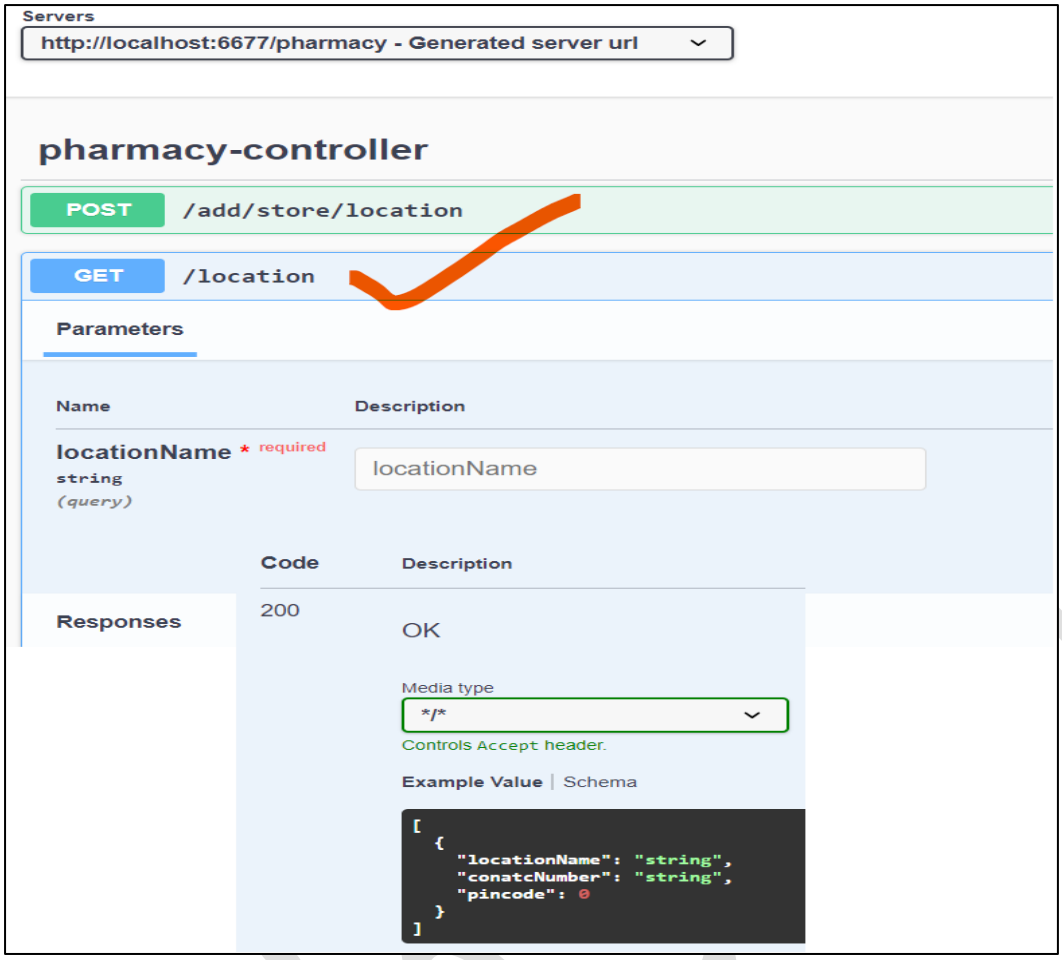
When we are sending data to flipkart app, now flipkart app forwarded data to pharmacy application via REST API call.



<u>Now Let's integrate Path variable and Query Parameters REST API Services:</u>

<u>Consuming API Services with Query Parameters:</u>

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



Consuming GET API Service with Query Parameter:

In **RestTemplate**, to handle Query Parameters Spring provided flexibility with **Hashmap** Object i.e. Configuring Query parameters with values key and values. Above Service Producing Response as JSON array of Objects. So create Response Class.

PharmacyResponse.java:

```
public class PharmacyResponse {

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

	//Setters and Getters
}
```

- **From above API details, we have one Request Parameter : locationName.**

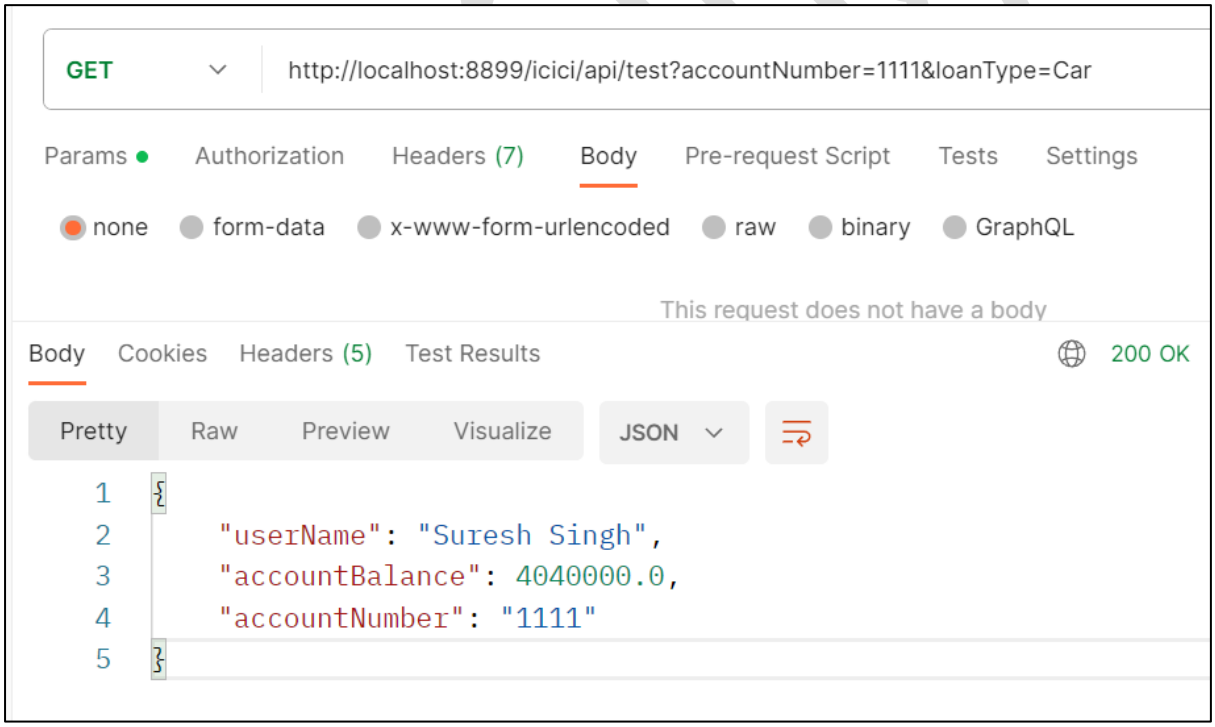
```
public List<PharmacyResponse> loadDetailsByLocationName(String location) {
```

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

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

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

Consuming Another Example with Query Parameters:

- Consume Below REST Service which contains Query Parameters From our Application.

Source of REST Service to be consumed:



- **Logic for Consumption:**

```
package com.flipkart.service;

import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.UriComponentsBuilder;
```

```
public class RestTemplateExample {

   public static void main(String[] args) {

      // Create a RestTemplate instance
      RestTemplate restTemplate = new RestTemplate();

      // Define the base URL of the API
      String baseUrl = "http://localhost:8899/icici/api/test";

      // Create query parameters using UriComponentsBuilder
      UriComponentsBuilder builder = UriComponentsBuilder.fromHttpUrl(baseUrl)
            .queryParam("accountNumber", "122334455")
            .queryParam("loanType", "House");

      // Build the final URL with query parameters
      String finalUrl = builder.toUriString();

      // Make a GET request to the API
      ResponseEntity<String> response = restTemplate.getForEntity(finalUrl, String.class);

      // Process the response
      if (response.getStatusCode().is2xxSuccessful()) {
         String responseBody = response.getBody();
         System.out.println(responseBody);
      } else {
         System.err.println("Request failed with status code: " + response.getStatusCode());
      }
   }
}
```

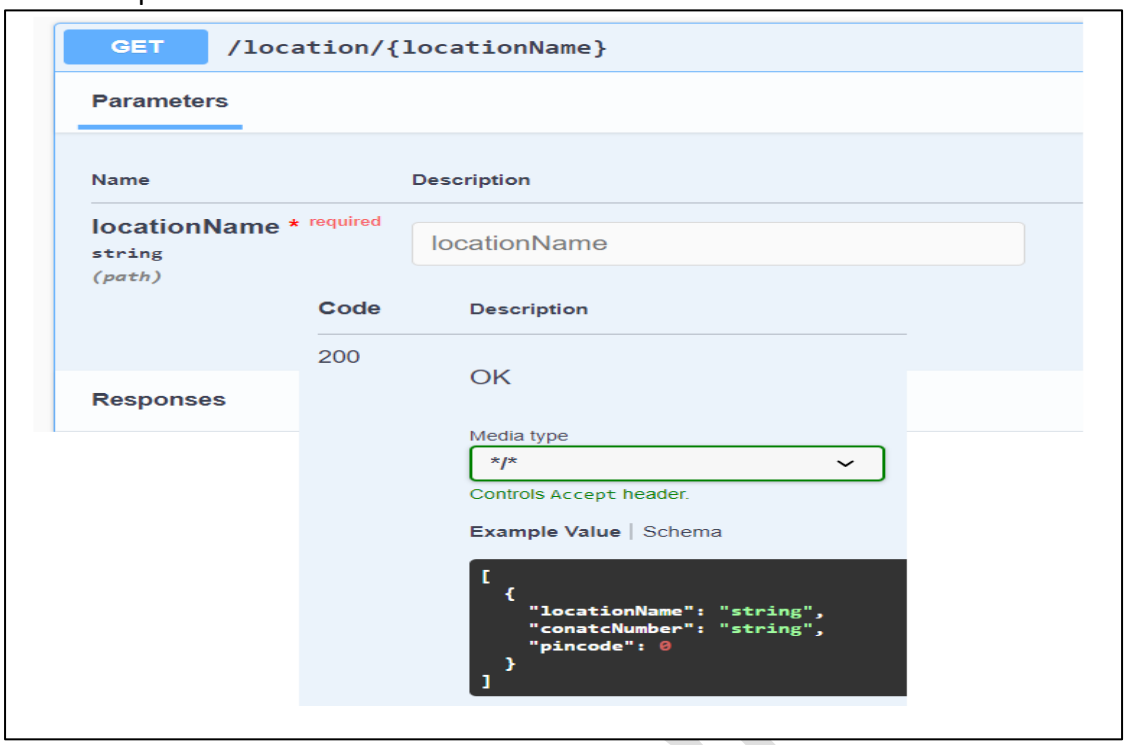
Output:

```
{
      "userName":"Suresh Singh",
      "accountBalance":4040000.0,
      "accountNumber":"122"
}
```

Consuming GET API Service with Path Parameter:

Example : Consume below Service which contains Path Variable.

In **RestTemplate**, to handle Path Parameters Spring provided flexibility with Hashmap Object or Object Type Variable Arguments as part of exchange() i.e. Configuring Path parameters with values. Above Service Producing Response as JSON array of Objects. So create Response Class.



PharmacyResponse.java

```
public class PharmacyResponse {

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

	//Setters and Getters
}
```

- **From above API details, we have one Path Parameter : locationName.**

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

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

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

	RestTemplate restTemplate = new RestTemplate();
	List<PharmacyResponse> respone = restTemplate.exchange(url, HttpMethod.GET,
```

```
                                                    null, List.class, values).getBody();

    return respone;
}
```

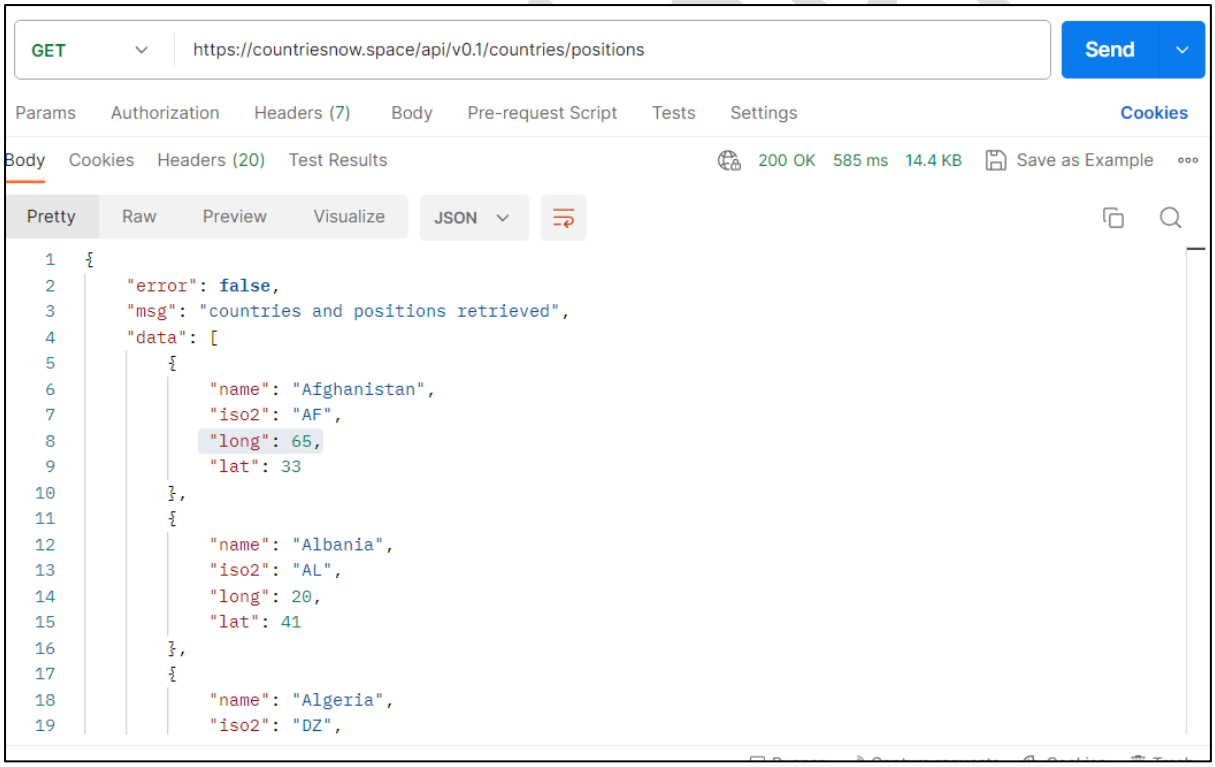
NOTE: We can handle both Path variable and Query Parameters of a single URI with Hashmap Object. i.e. We are passing values to keys. Internally spring will replace values specifically.

Integration of One More REST API Service:

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

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

Above API call, Producing JSON Response, as shown in below Postman. Depends on Response we should create JAVA POJO classes to serialize data from JAVA to JSON and vice versa.



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

Country.java

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

```
        //Setters and Getters
}
```

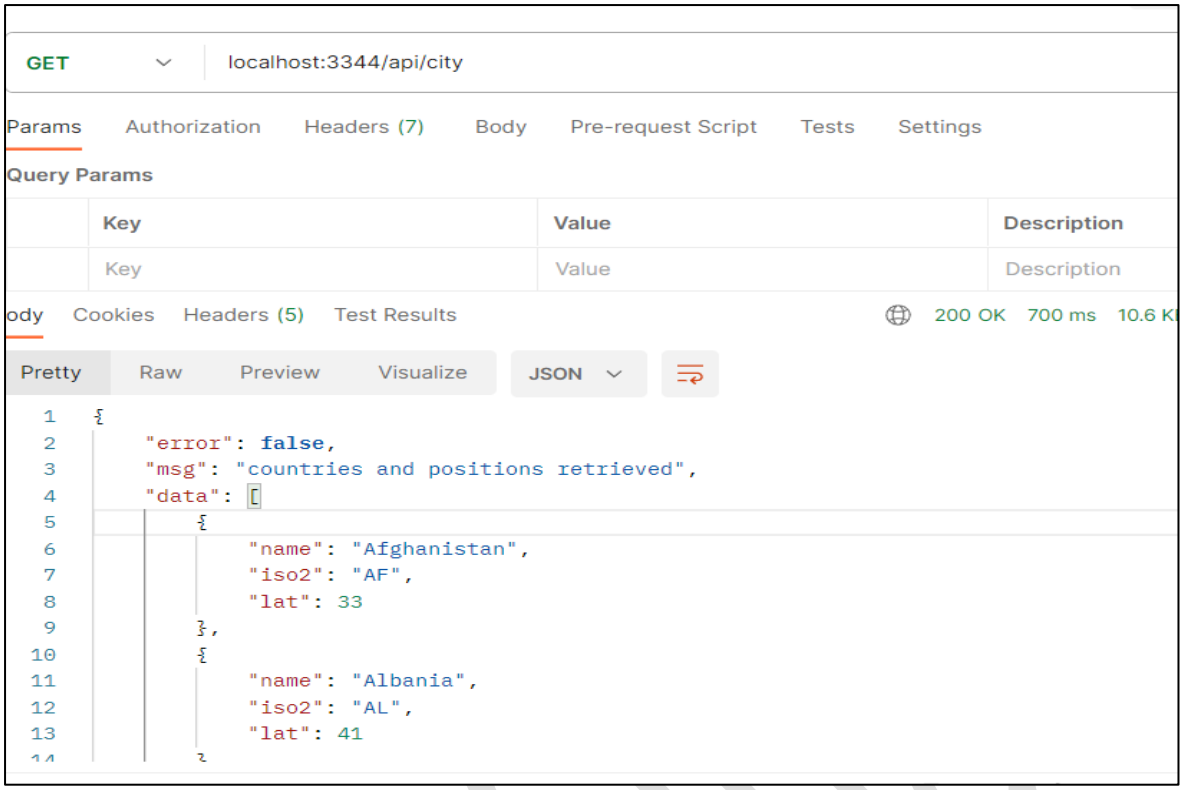
CountriesResponse.java

```
public class CountriesResponse {

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

        //Setters and Getters
}
```

API Consuming Logic:

```
public CountriesResponse loadCities() {

        String url = "https://countriesnow.space/api/v0.1/countries/positions";

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

        System.out.println(respone);
        return respone;
}
```

- **Testing from our Application Postman:**



How to Pass Headers with RestTemplate when Consuming REST Services:

In Spring's **RestTemplate**, we can work with HTTP headers by using the **HttpHeaders** class. You can add, retrieve, and manipulate headers in both requests and responses. Here's how we can work with headers in **RestTemplate**:

Adding Headers to a Request:

In this example, we create an **HttpHeaders** object and set custom headers. We can add headers to your HTTP request before sending it using RestTemplate. Here's an example of how to add headers to a request:

```
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

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

      // Create a RestTemplate instance
      RestTemplate restTemplate = new RestTemplate();

      // Define the request URL
      String url = "https://api.example.com/api/resource";
```

```
        // Create an HttpHeaders object to set custom headers
        HttpHeaders headers = new HttpHeaders();
        headers.set("Authorization", "Bearer yourAccessToken");
        headers.set("Custom-Header", "Custom-Value");
        headers.add("token", "dss3232444gt54t5tgrgtry54y5ydsdsdsdsdsdsd");

        HttpEntity<Object> entity = new HttpEntity<Object>(headers);

        // Create a HttpEntity with the custom headers
        ResponseEntity<String> responseEntity =
                                restTemplate.exchange(url, HttpMethod.POST, entity, String.class);

        // Process the response
        if (responseEntity.getStatusCode().is2xxSuccessful()) {
            String responseBody = responseEntity.getBody();
            System.out.println("Response: " + responseBody);
        } else {
            System.err.println("Request failed with status code: " +
                                                                responseEntity.getStatusCode());
        }
    }
}
```

Accessing Headers in a Response:

We can access response headers when you receive a response from the server. Here's an example:

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class RestTemplateResponseHeadersExample {

    public static void main(String[] args) {
        // Create a RestTemplate instance
        RestTemplate restTemplate = new RestTemplate();

        // Define the request URL
        String url = "https://api.example.com/api/resource";

        // Send a GET request and receive the entire ResponseEntity for the response
        ResponseEntity<String> responseEntity = restTemplate.getForEntity(url, String.class);

        // Access response headers
        HttpHeaders responseHeaders = responseEntity.getHeaders();
        String contentType = responseHeaders.getFirst("Content-Type");
```

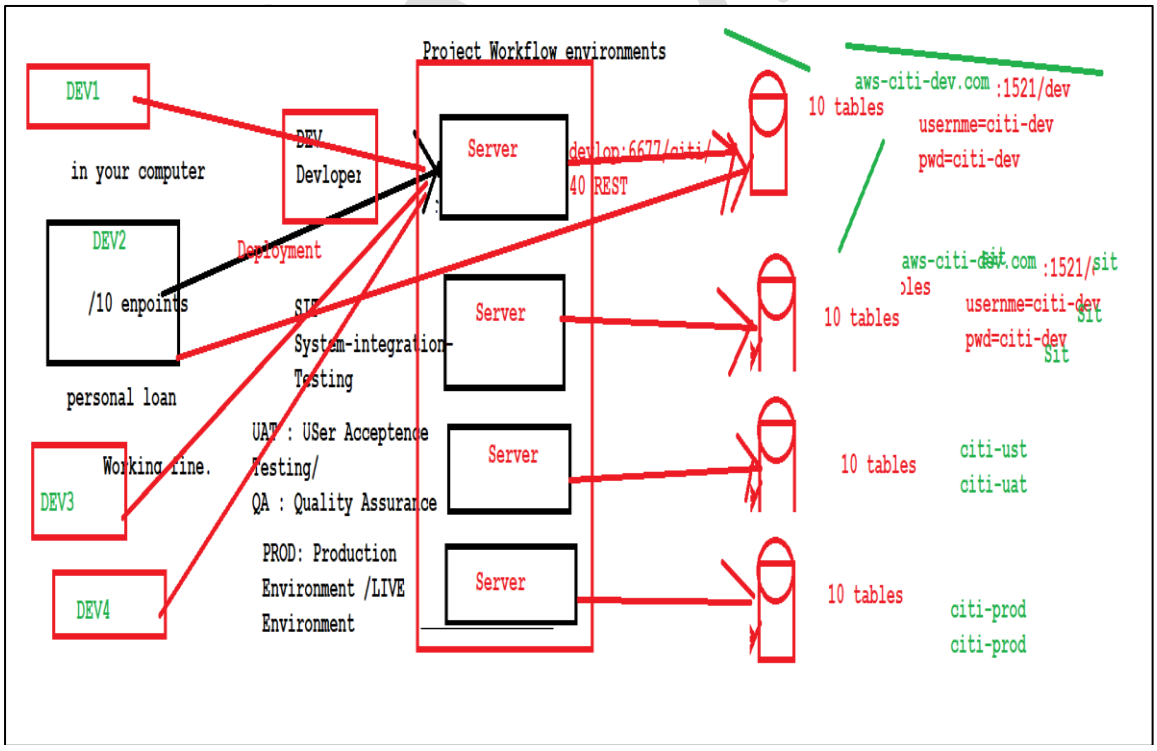
```
        long contentLength = responseHeaders.getContentLength();

        System.out.println("Content-Type: " + contentType);
        System.out.println("Content-Length: " + contentLength);

        // Access the response body
        String responseBody = responseEntity.getBody();
        System.out.println("Response Body: " + responseBody);
    }
}
```

In this example, we use **responseEntity.getHeaders()** to access the response headers and then retrieve specific header values using **responseHeaders.getFirst("Header-Name").**

Working with headers allows you to customize your requests and process responses more effectively in your **RestTemplate** interactions.

Spring Boot Profiles:

Every enterprise application has many environments, like: Dev, Sit, UAT, Prod. Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments.

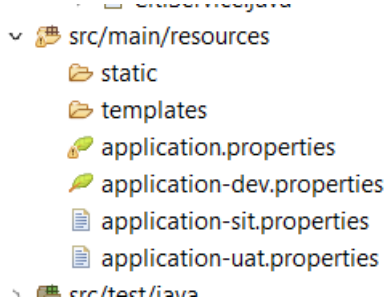


Each environment requires a setting that is specific to them. For example, in DEV, we do not need to constantly check database consistency. Whereas in UAT and PROD, we need to.

These environments host specific configurations called Profiles.

How Do we Maintain Profiles?

This is simple — properties files!
We make properties files for each environment and set the profile in the application accordingly, so it will pick the respective properties file. Don't worry, we will see how to set it up.



In this demo application, we will see how to configure different databases at runtime based on the specific environment by their respective profiles.

As the DB connection is better to be kept in a property file, it remains external to an application and can be changed. We will do so here. But, Spring Boot — by default — provides just one property file (**application.properties**). So, how will we segregate the properties based on the environment?

The solution would be to create more property files and add the "**profile**" name as the suffix and configure Spring Boot to pick the appropriate properties based on the profile.

Then, we need to create three application-<profile>.properties:

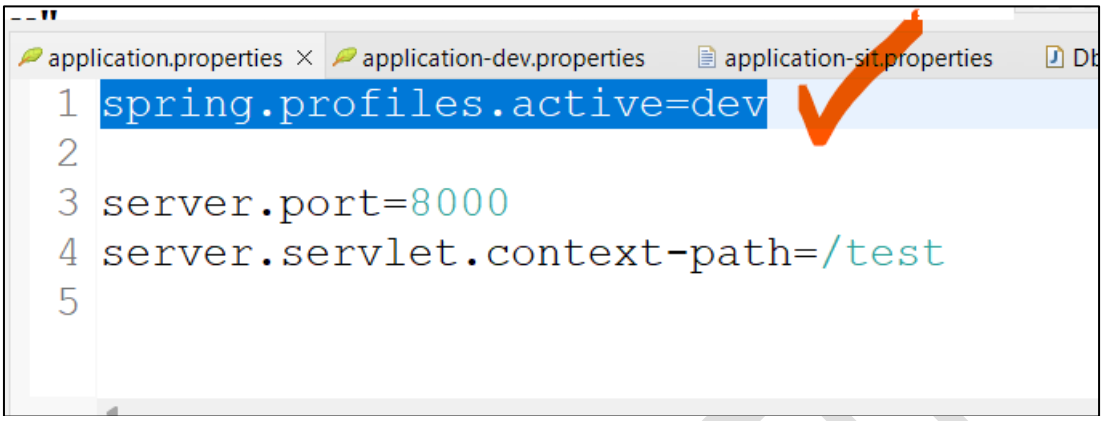
- application-dev.properties
- application-sit.properties
- application-uat.properties

Of course, the **application.properties** will remain as a master properties file, but if we override any key in the profile-specific file, then it will take priority.

Generally profile files will be created specific to Environments in projects. So we will configure properties and value which are really related to that environment. For example, In Real time Projects implementation, we will have different databases i.e. different database hostnames , user name and passwords for different environments. We will define common properties and values across all environments in side main **application.properties** file.

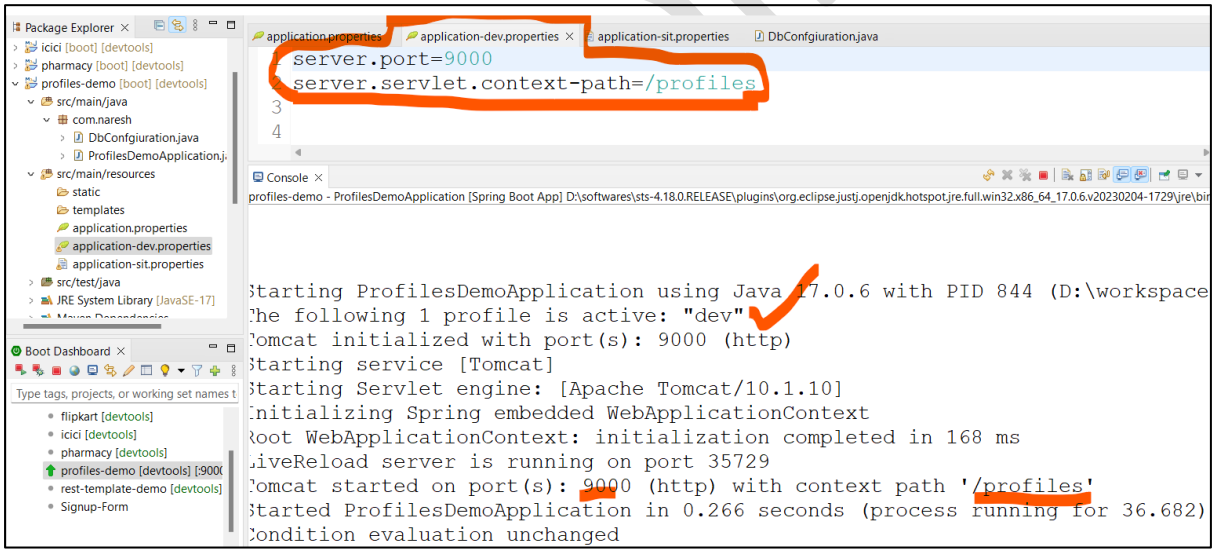
How to run Application with Specific Profile:

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



Now run application as SpringBoot or Java application, SpringBoot will load by default properties of **application.properties** and loads configured profiles properties file **application-dev.properties** file.

NOTE: Whenever we have same property in main **application.properties** and **application-<profile>.properties,** priority given to profile specific property and it's value.



This is how we are running application with specific profile i.e. loading specific profiles properties file.

Now, we are done with properties files. Let's configure in the Configuration classes to pick the correct properties.

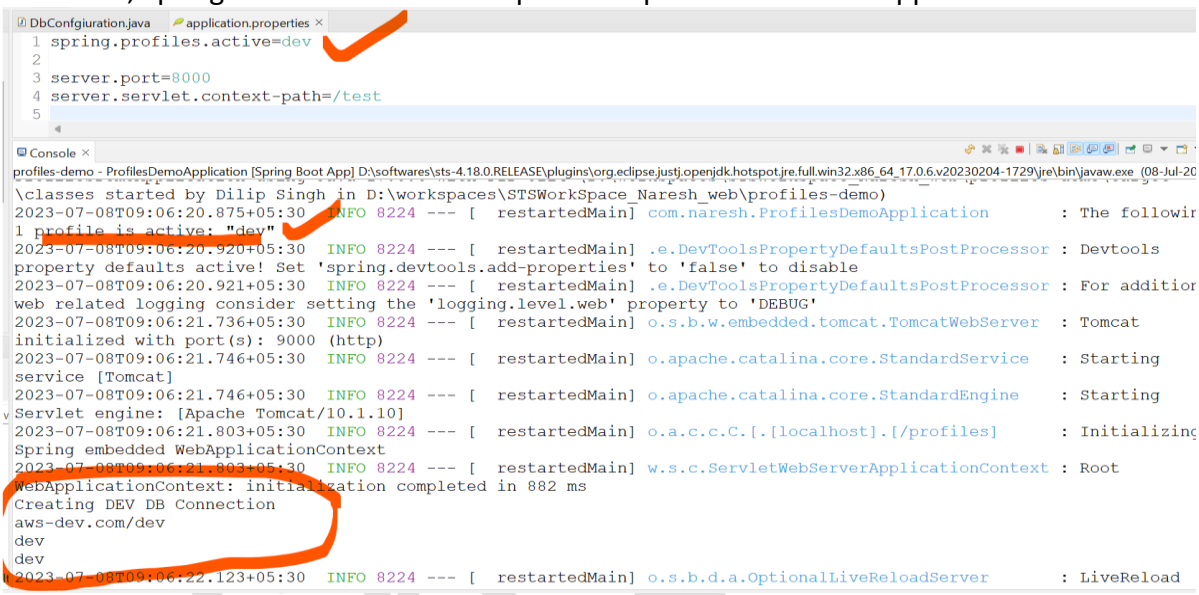
For Example, Database Connection should be created for specific profile or environment from configuration class.

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
public class DbConfgiuration {

	@Value("${db.hostName}")
	String hostName;

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

	@Value("${db.password}")
	String password;

	@Profile("sit")
	@Bean
	public String getSitDBConnection() {
		System.out.println("SIT Creating DB Connection");
		System.out.println(hostName);
		System.out.println(userName);
		System.out.println(password);
		return "SIT DB Connection Sccusessfu.";
	}

	@Profile("dev")
	@Bean
	public String getDevDBConnection() {
		System.out.println("Creating DEV DB Connection");
		System.out.println(hostName);
		System.out.println(userName);
		System.out.println(password);

		return "DEV DB Connection Sccusessfu.";
	}
}
```

We have used the **@Profile("dev")** and **@Profile("sit")** for specific profiles to pickup properties and create specific bean Objects. So when we start our application with "**dev**" profile, only **@Profile("dev")** bean object will be created not **@Profile("sit")** object i.e. The other profile beans will not be created at all.

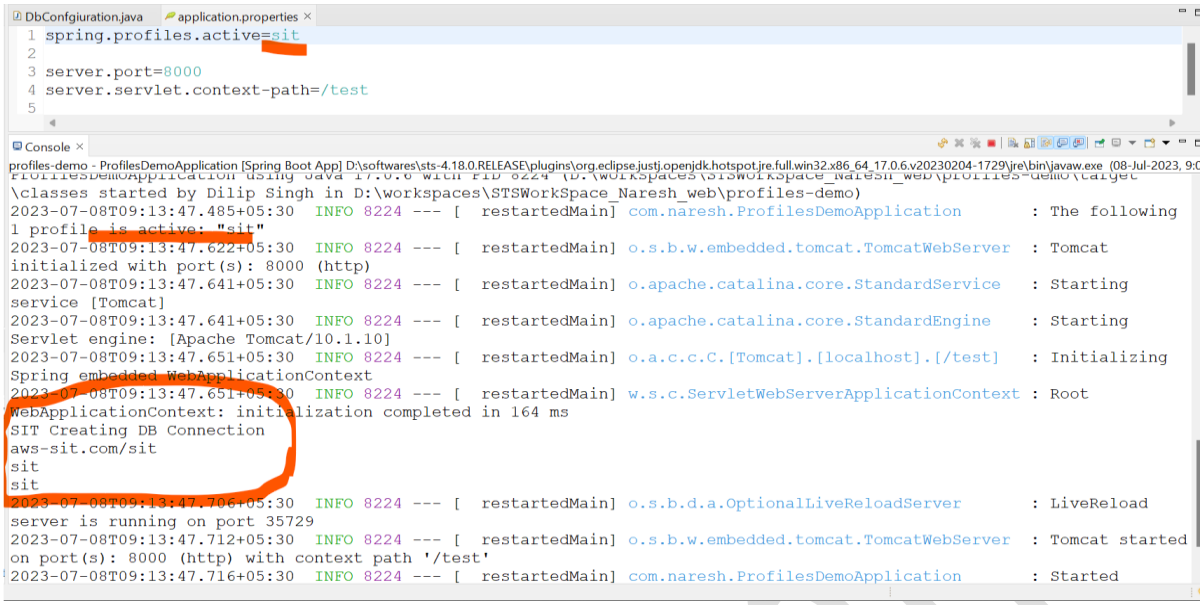
How application knows that this is DEV or SIT profile? how do we do this?
We will use **application.properties** with property **spring.profiles.active**=<profile>

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



We are not seeing any details of **sit** profile bean configuration i.e. skipped Bean creation because active profile is **dev**.

Now Let's change our active profile to **sit** and observe which Bean object created and which are ignored by Spring.



That's it! We just have to change it in **application.properties** to let Spring Boot know which environment the code is deployed in, and it will do the magic with the setting.

SpringBoot Actuator:

In Spring Boot, an actuator is a set of endpoints that provides various production-ready features to help monitor and manage your application. It exposes useful endpoints that give insights into your application's health, metrics, environment, and more. Actuators are essential for monitoring and managing your Spring Boot application in production environments.

To enable the Spring Boot Actuator, you need to add the relevant dependencies to your project. In most cases, you'll want to include the `spring-boot-starter-actuator` dependency in your pom.xml (Maven) or build.gradle (Gradle) file.

For Maven:

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

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the **health** endpoint provides basic application health information.

The built-in endpoints are auto-configured only when they are available. Most applications choose exposure over HTTP, where the ID of the endpoint and a prefix of **/actuator** is mapped to a URL. For example, by default, the health endpoint is mapped to **/actuator/health**

Some of endpoints are:

ID	Description
beans	Displays a complete list of all the Spring beans in your application.
health	Shows application health information.
info	Displays arbitrary application info.
loggers	Shows and modifies the configuration of loggers in the application.

Exposing Endpoints:

By default, only the **health** endpoint is exposed. Since Endpoints may contain sensitive information, you should carefully consider when to expose them. To change which endpoints are exposed, use the following specific **include** and **exclude** properties:

Property

management.endpoints.web.exposure.exclude=<endpoint>,<endpoint>
management.endpoints.web.exposure.include=<endpoint>,<endpoint>

***** can be used to select all endpoints. For example, to expose everything over HTTP except the env and beans endpoints, use the following properties:

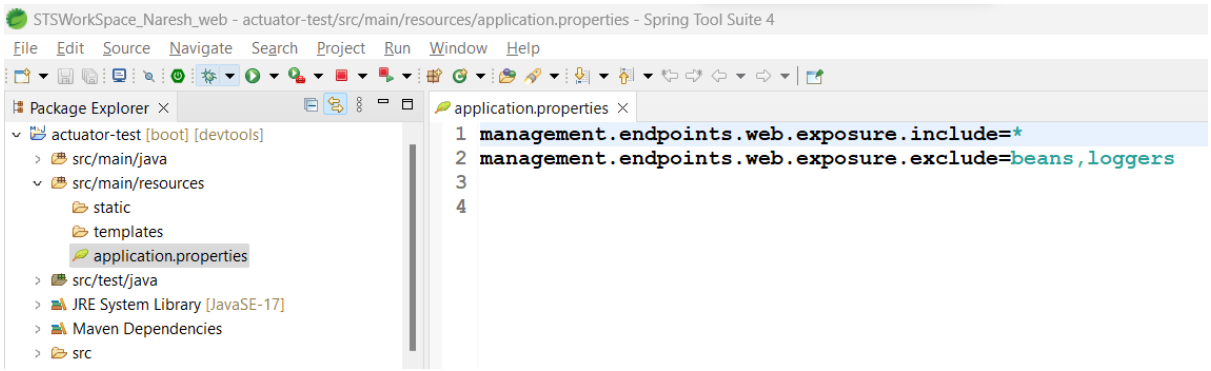
Properties:

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

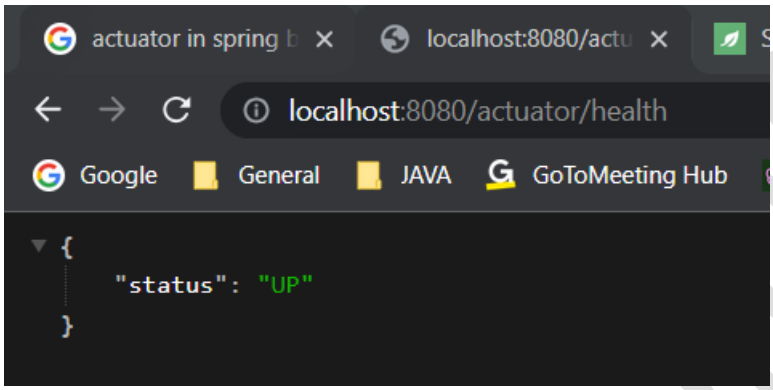
For security purposes, only the **/health** endpoint is exposed over HTTP by default. You can use the **management.endpoints.web.exposure.include** property to configure the endpoints that are exposed.

Before setting the **management.endpoints.web.exposure.include**, ensure that the exposed actuators do not contain sensitive information, are secured by placing them behind a firewall, or are secured by something like Spring Security.

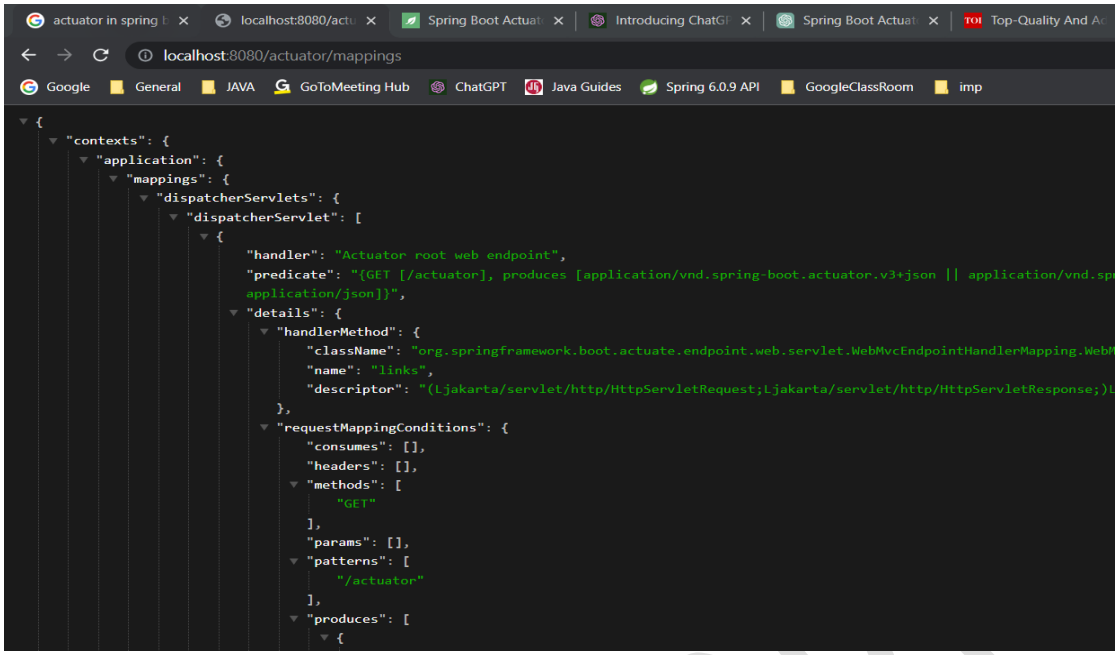
Configure Properties in **application.properties**:



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



Similarly, [http://localhost:8080/**actuator/mappings**](http://localhost:8080/actuator/mappings)



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

Thank you
Dilip Singh