

ENTERPRISE APPLICATION DEVELOPMENT

USING JAVA EE (ENTERPRISE EDITION)



CONTENTS

Introduction	I
creating the java server (tomcat)	6
Java servlets	11
session handling in java servlets	23
java jsp	26
MVC DDesign architecture	36
More on servlet lifecycle.....	46
working with the get and post methods in servlets	49
More about java JSP	50
introduction to Enterprise java beans (EJB)	55
jms – java message service.....	60
introduction to rest api.....	61
building rest api with java.....	66
creating REST api using spring.....	74
connecting to databases (mysql)	88
microservice architecture in rest api	94
spring security flow 3.0	106
Authentication in spring boot.....	108
<i>Summary of Key Concepts:</i>	117
<i>Example Flow</i>	118
connecting spring boot to mongo db.....	126
connecting mailjet api	131
connecting ms azure blob storage (azurite)	134
using oAuth in spring boot.....	136
spring boot essentials.....	141
commen mistakes in session tokens	144

INTRODUCTION

Main platforms in java programming language:

- Java Standard edition (SE)
- Java enterprise edition (SE)
- Java micro edition (ME)
- Java FX

Name	Description
Java SE	<ul style="list-style-type: none">• Provides core functionality.• Defines from basic OOP concepts to GUI development with XML.• This platform consists of virtual machine development tools and other java tools.
Java EE	<ul style="list-style-type: none">• This is built on the java SE platform.• This provides an API and a runtime environment (A runtime environment is a software layer that provides the necessary resources and services for a program to execute. It acts as an intermediary between the application and the underlying operating system) to run large scale, multi-tiered, scalable and secure network applications.• It provides libraries for database access (JDBC, JPA), remote method invocation (RMI), messaging (JMS), web services, XML processing, and defines standard APIs for Enterprise JavaBeans, servlets, Java Server Pages etc.
Java ME	<ul style="list-style-type: none">• This provides an API and a small-footprint virtual machine for smaller devices.• The API is a subset of the Java SE API, along with special class libraries useful for small device application development.• Java ME applications are often clients of Java EE platform services.
Java FX	<ul style="list-style-type: none">• For web development using java

MORE ABOUT JAVA EE

1. Web Specifications of Java EE

- **Servlet**- This specification defines how you can manage HTTP requests either in a synchronous or asynchronous way. It is low level, and other specifications depend on it.

Synchronous in HTTP

- In synchronous communication, the client sends a request to the server and waits (blocks) until it receives a response. The client cannot perform other tasks while waiting for the response.
- The operations are performed in a sequential manner. Each request must be completed before the next one can be processed.
- Example: Traditional web page loading, where the browser sends a request for a web page and waits for the server to send back the entire page before displaying it.

Asynchronous in HTTP

- In asynchronous communication, the client sends a request to the server and continues to perform other tasks without waiting for the response. The response is handled whenever it arrives.
 - Multiple requests can be sent and processed concurrently. The client can handle multiple responses as they arrive, without blocking other operations.
 - Example: AJAX (Asynchronous JavaScript and XML) in web development, where the browser can send requests to the server and update parts of a web page without reloading the entire page.
- **WebSocket**- WebSocket is a computer communication protocol, and this API provides a set of APIs to facilitate WebSocket connections.
 - **Java Server Faces**- It is a service which helps in building GUI out of components.
 - **Unified Expression Language**- It is a simple language which was designed to facilitate web application developers.

2. Web Service Specifications of Java EE

- **Java API for RESTful Web Services**- It helps in providing services having Representational State Transfer schema.
- **Java API for JSON Processing**- It is a set of specifications to manage the information provided in JSON format.
- **Java API for JSON Binding**- It is a set of specifications provided for binding or parsing a JSON file into Java classes.
- **Java Architecture for XML Binding**- It allows binding of xml into Java objects.
- **Java API for XML Web Services**- SOAP is an xml-based protocol to access web services over http. This API allows you to create SOAP web services.

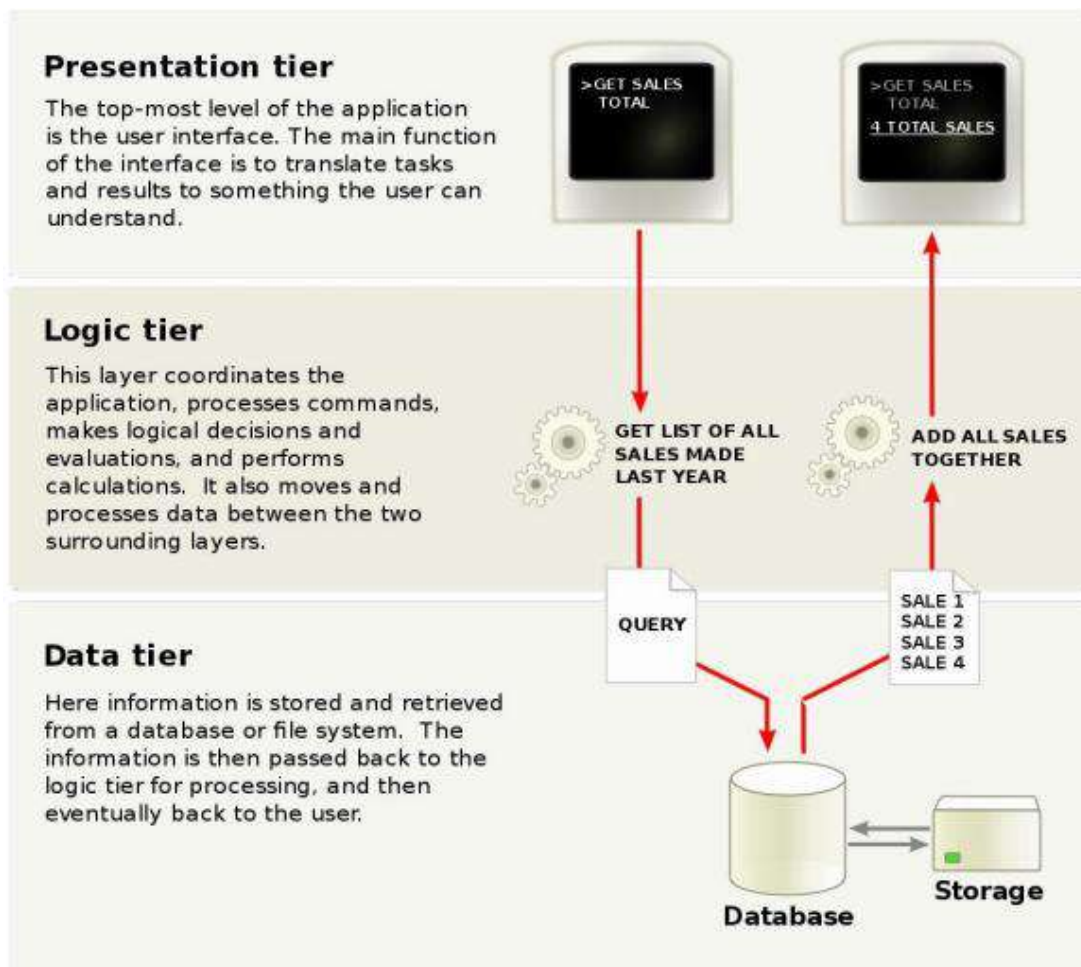
3. Enterprise Specifications of Java EE

- **Contexts and Dependency Injection**- It provides a container to inject dependencies as in Swing.
- **Enterprise JavaBean**- It is a set of lightweight APIs that an object container possesses in order to provide transactions, remote procedure calls, and concurrency control.
- **Java Persistence API**- These are the specifications of object-relational mapping between relational database tables and Java classes.
- **Java Transaction API**- It contains the interfaces and annotations to establish interaction between transaction support offered by Java EE. The APIs in this abstract from low-level details and the interfaces are also considered low-level.
- **Java Message Service**- It provides a common way to Java program to create, send and read enterprise messaging system's messages.

ENTERPRISE APPLICATIONS

- **Enterprise application is computer software used to satisfy the needs of an organization, schools, interest-based user groups, clubs, charities and governments.**

- The Java EE platform is designed to help developers to **create large-scale, multi-tiered, scalable, reliable, and secure network applications.**
- These applications are designed to solve problems encountered for large companies.
- The affective of Java EE to enterprise applications: this is designed to reduce the complexity of enterprise application development.
- **Characteristics:**
 - Distributed: A distributed application is software that run on multiple computers within a network to achieve a specific task.
 - Multi tired (N-tier architecture): Multi-tier Application is engineered to have the processing, data management and presentation functions physically and logically separated.



- Scalability: Scalability is an attribute that describes the ability of a software to grow and manage increased demand.

JAVA EE APPLICATION ARCHITECTURE

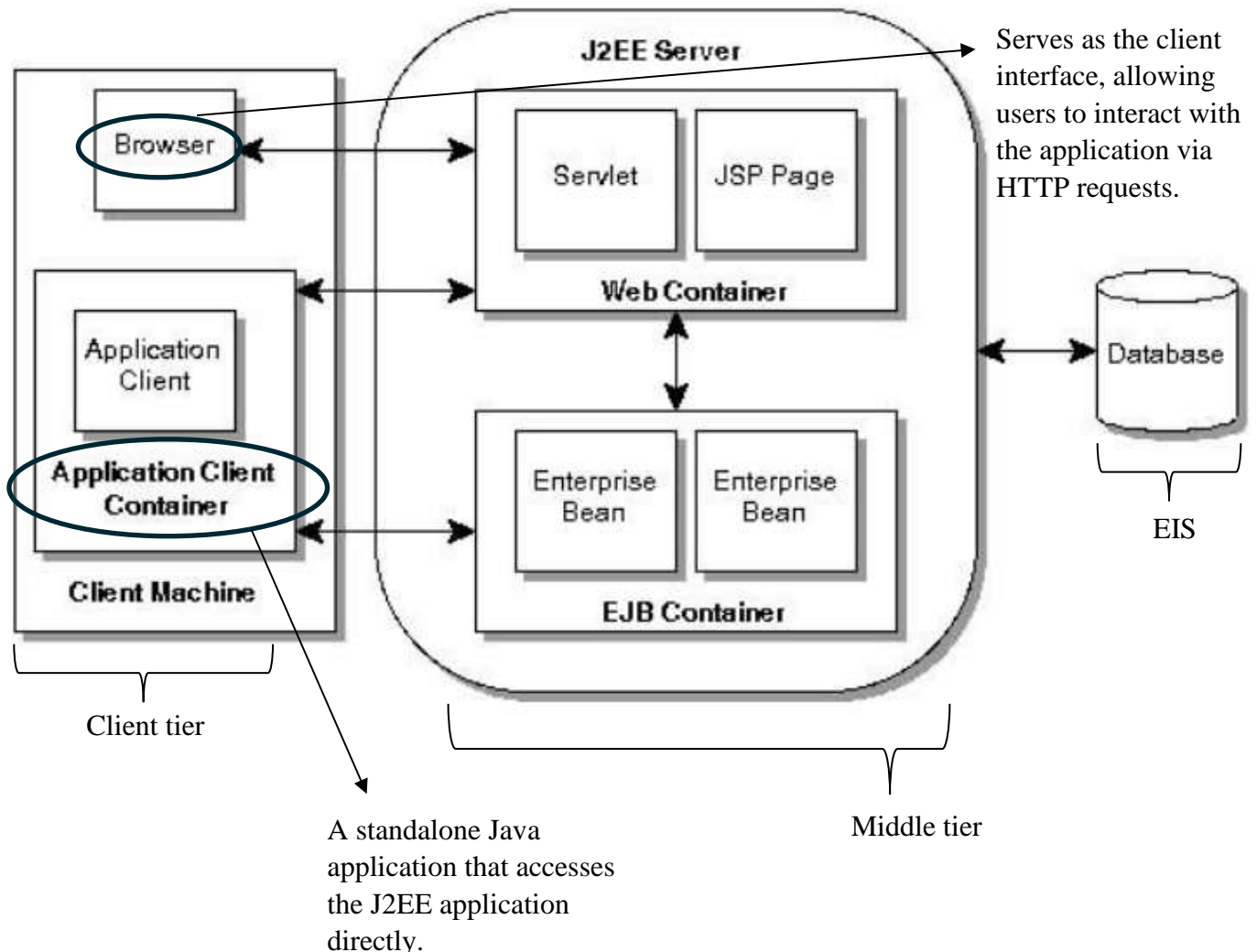
- **Multi tired applications:**
 - In multi tired applications the functionality of the entire application is separated into functional areas. They are called **tiers**.
 - Mainly, the multi tired application contains 3 main tiers. They are,
 - **Client tier:**
Makes the request to the middle tier.
 - **Middle tier:**

This tier has business functions, handles the client the client requests and processes them.

- **Data tier (Enterprise Information Systems tier or EIS):**

Store the application data in permanent data storage in the data tier or EIS.

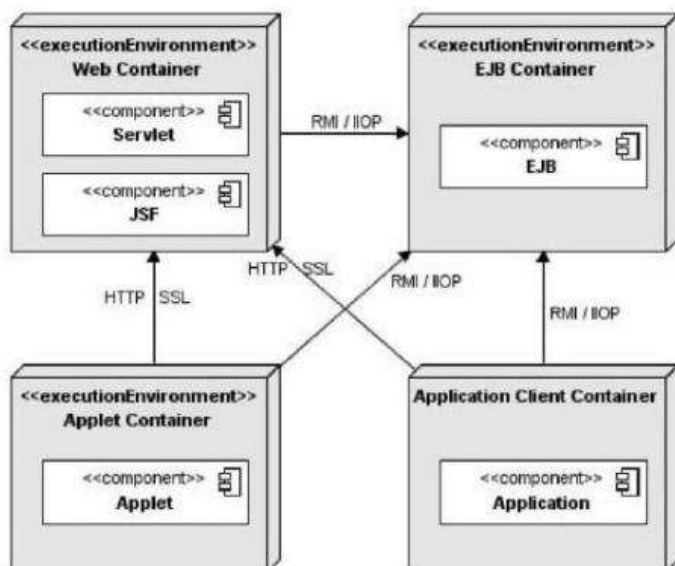
- This Java EE uses **multi tiered application architecture**.
- **Java EE is a set of specifications implemented by different containers.**
- The components use well-defined contracts to communicate.
- **Containers are runtime environments that provide certain services to the components hosted.**



- **Client tier:** Manages the execution of application client components. Application clients and their container run on the client.
 - Acts as the front end in the application (programs that interact with the user).
 - Get the inputs and convert to requests then forward to the server (middle tier).
 - Also, translates the server response and presents it to the user.
 - These clients could be web browsers, stand alone applications or another web servers.
 - The firewalls are used in this tier.
 - **Java applets:** Applets in Java were designed to provide interactive features to web applications that couldn't be achieved with HTML alone.
 - Features and outcomes:
 - Interactive Web Content, Platform Independence, Security, Rich User Interfaces, Client-side processing.
 - An **applet container** is a part of the Java EE architecture that manages the execution of applets.

- Middle tier (**application or web server**): The runtime portion of a Java EE product. A Java EE server provides EJB and web containers.
 - Consists of one or more sub tiers(containers).
 - This processes the request and provides the response to the user or client.
 - **Main containers (sub tiers) to process the request:**
 - **Web container (web tier)**
 - Manages the execution of web pages, servlets, and some EJB components for Java EE applications. Web components and their container run on the Java EE server.
 - Used to **develop web applications and provide internet functions to J2EE** application.
 - **Main components: servlets,JSP,HTML or XML**
 - **EJB container (EJB tier)**
 - Manages the execution of enterprise beans for Java EE applications. Enterprise beans and their container run on the Java EE server.
 - Used to deploy business applications and contains the collections of **Enterprise Java Beans (EJB)**. These EJB contains the business logic of applications.
 - The enterprise JavaBeans server and the enterprise JavaBeans container responsible for low level **system services** required to implement business logic of an enterprise java bean.
 - Examples for system services: resource pooling, State management, process management etc.
 - **JMS, Java mail etc.**
- EIS tier:
 - Provide the connectivity between java EE and non-java EE software.
 - Include Resources: Database Servers, Enterprise resource planning (ERP) systems, Legacy Systems.
 - This should be the backend for an application.
 - The developers can leverage the resources in the market and not need to replicate those : flexibility.

RELATIONSHIP AMOUNG THE CONTAINERS



JAVA EE SERVICES

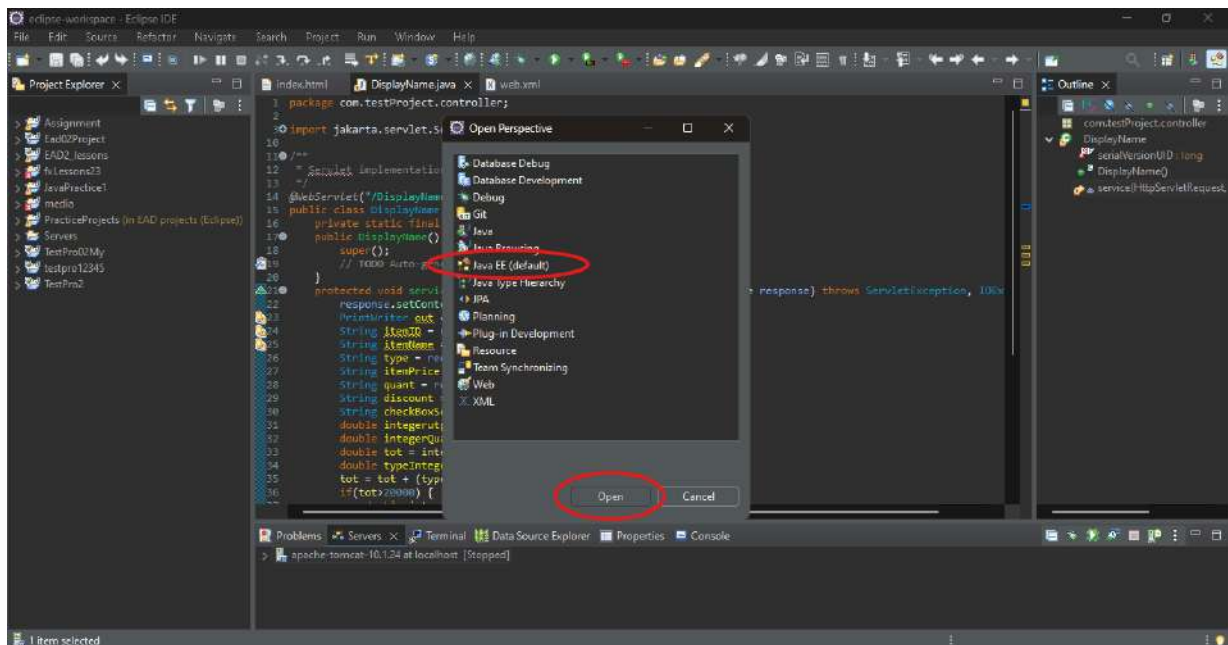
- Messaging: java message service (JMS) provides asynchronous messaging between disparate parts in the application.
- Transaction Provided by the Java Transaction API (JTA) implementation.
- Persistence: Offered by the Java Persistence API (JPA) which drives Object Relational Mapping (ORM) and an abstraction for the Database operations.
- Web service: JAXRS and JAXWS provide support for REST and SOAP style services respectively.
- Contexts & Dependency Injection CDI provides loosely coupled and type safe injection of resources.

JAVA EE BENEFITS

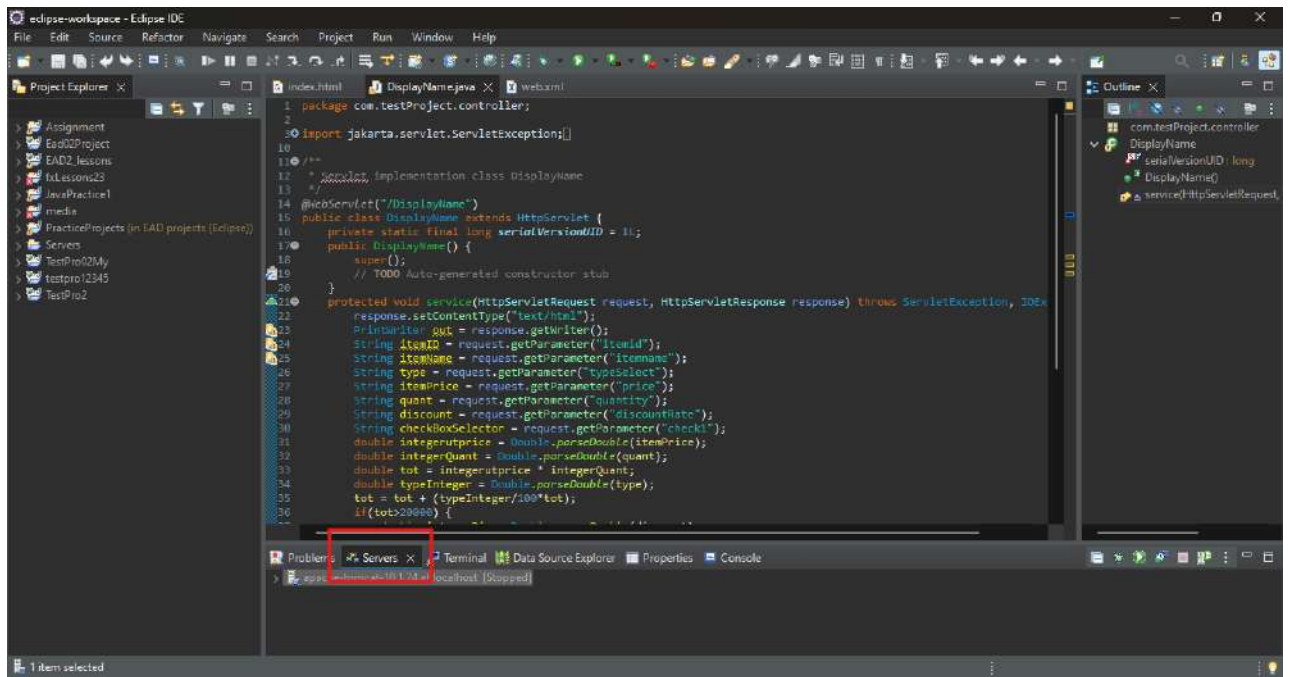
- High availability.
- Scalability.
- Integration with existing systems.
- Freedom to choose vendors of application servers, tools,
- Flexibility of scenarios and support to several types of clients programming productivity.

CREATING THE JAVA SERVER (TOMCAT)

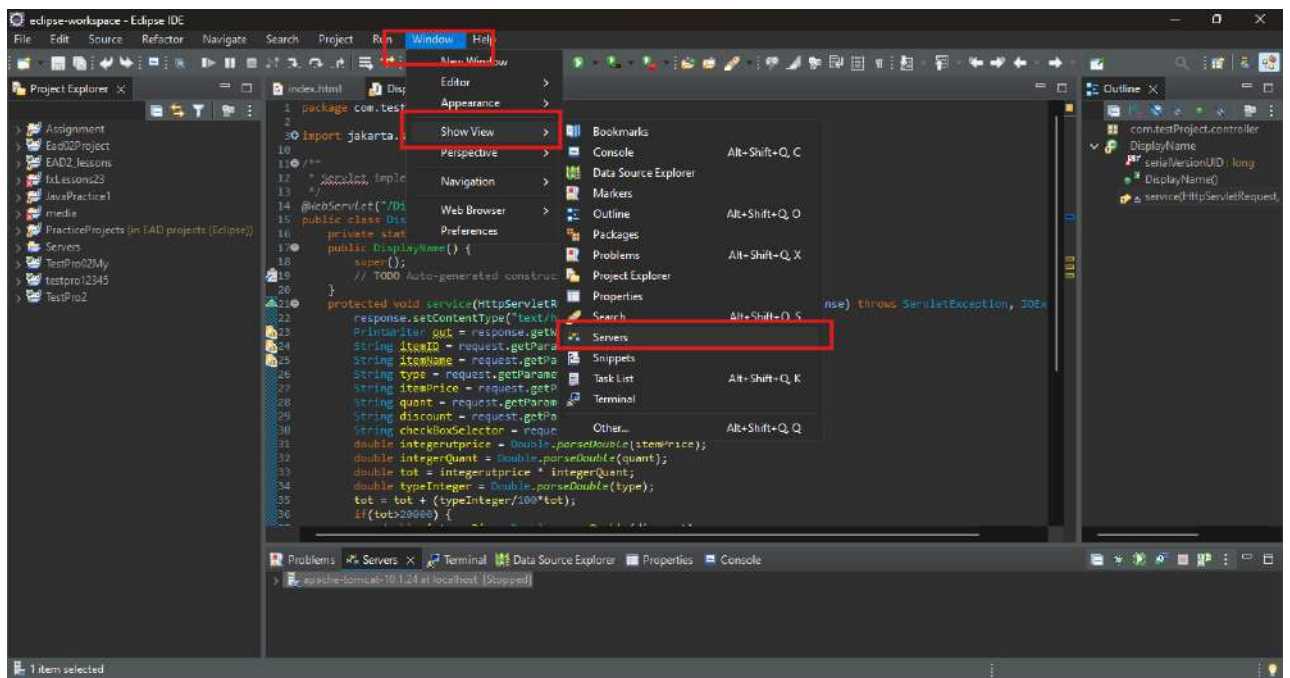
- First select the java version (select the Java EE).



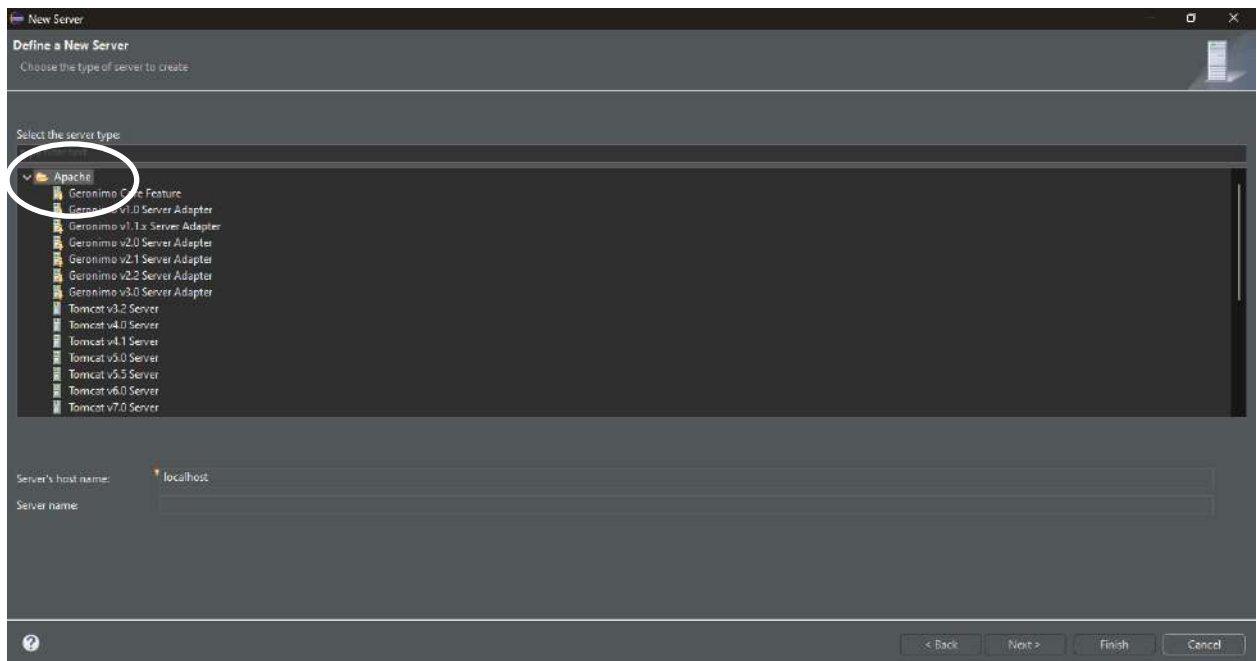
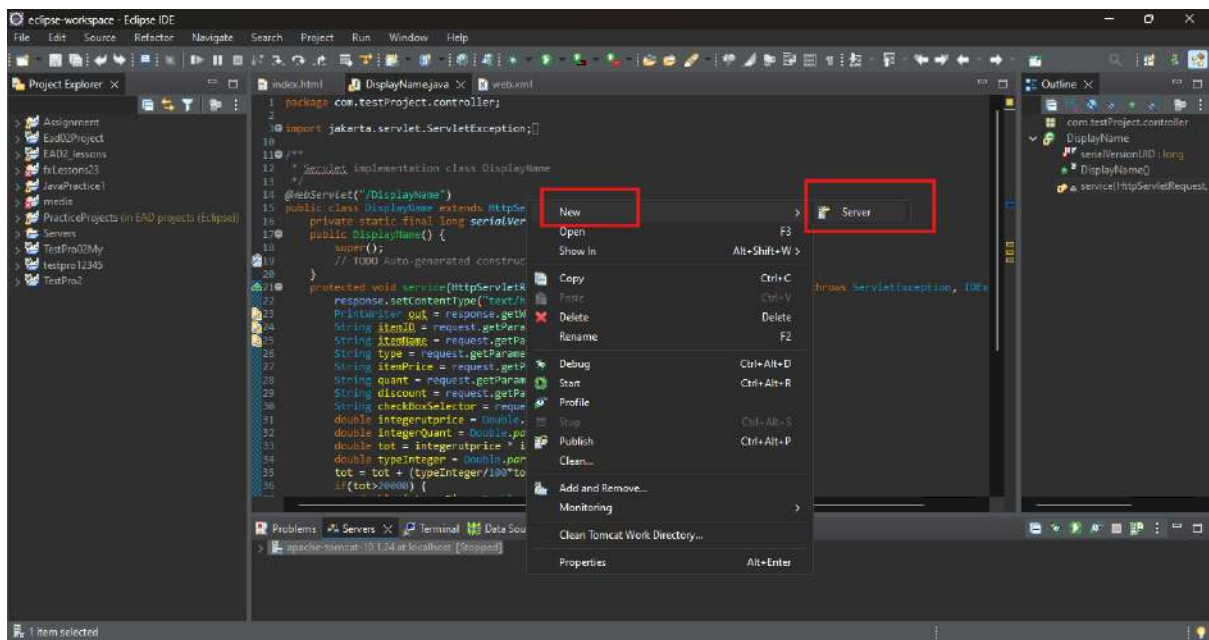
- Then you will be able to see a tab called “servers” like the following



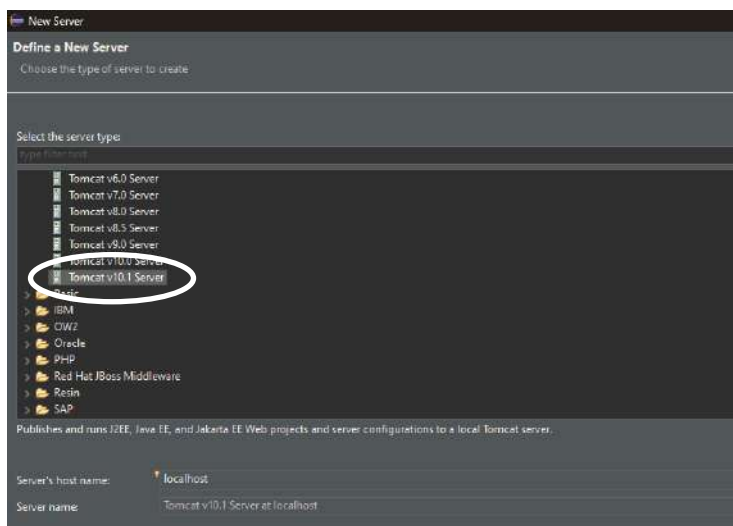
- But if that tab is not visible, select the visibility of the tab as below.



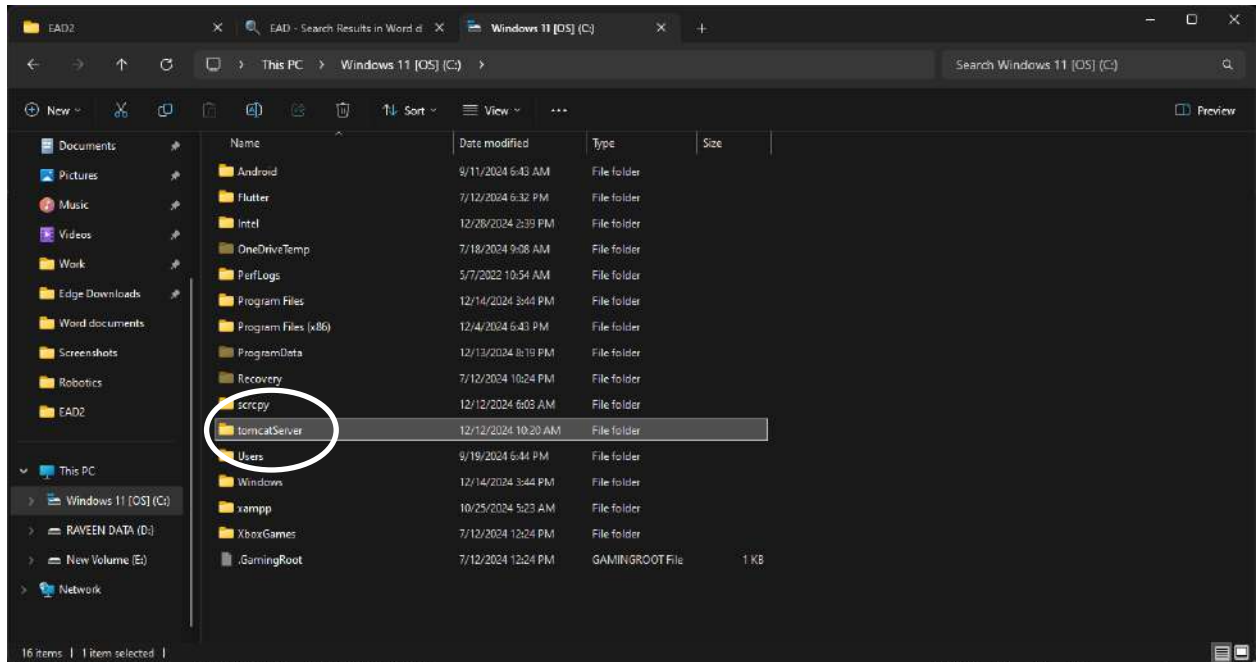
- Then install the tomcat server as below (right click the server tab space).



- In the apache section, find the tomcat server (select the Tomcat v10.1 version. The latest version does not have features of java servelets).



- Then create a separate folder to store that server in the C drive (windows drive) and create it in the root of the C drive.

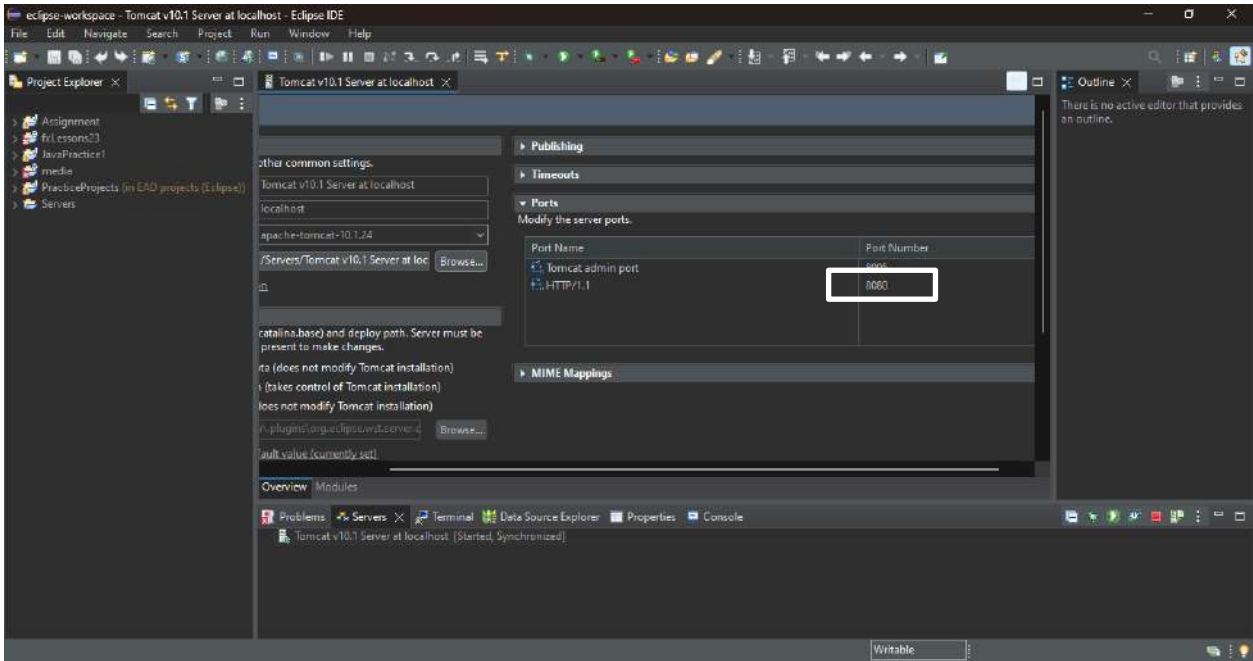


- Then select that folder and click the “download and install” to install the server in the specific location.



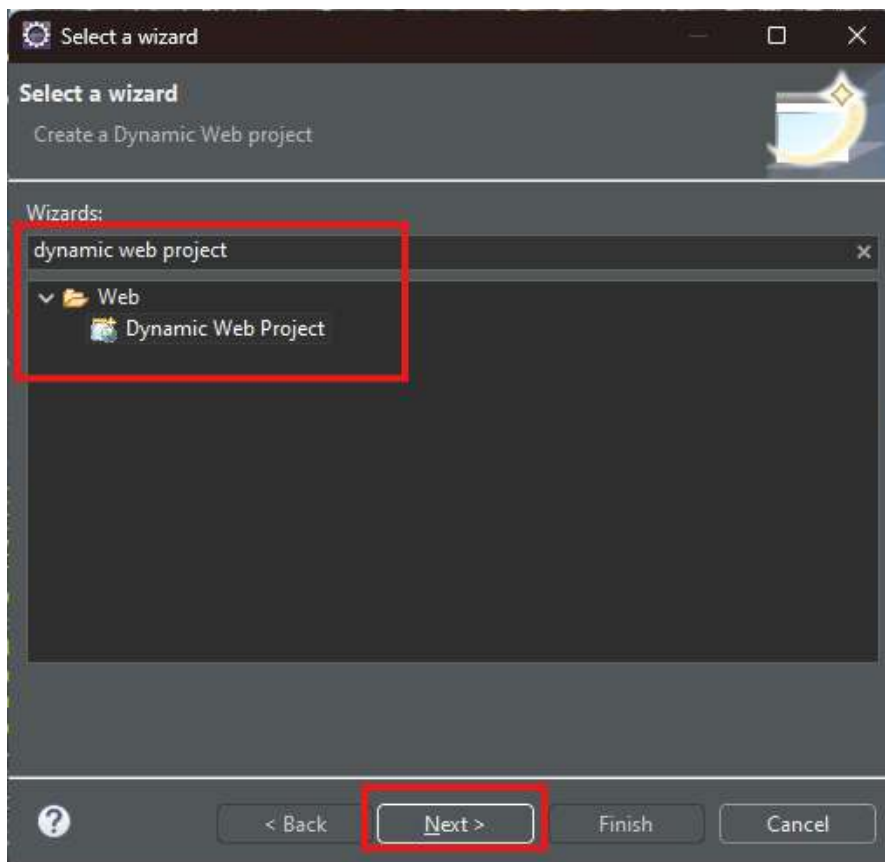
- Then the created server will be displayed as below.

- If the server does not start, go to the server properties and Make it 8080 and ctrl+s and close the window start the server again.

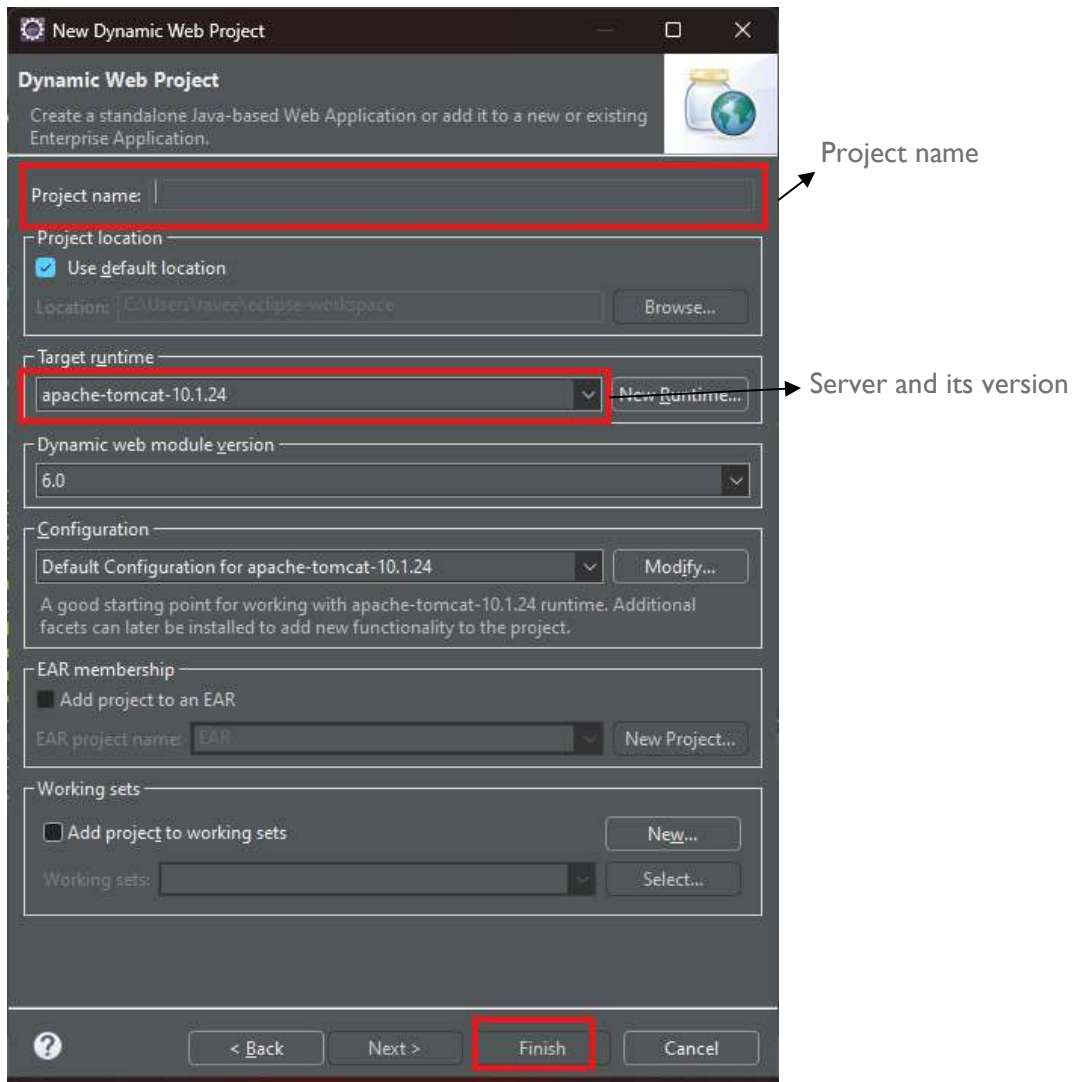


JAVA SERVLETS

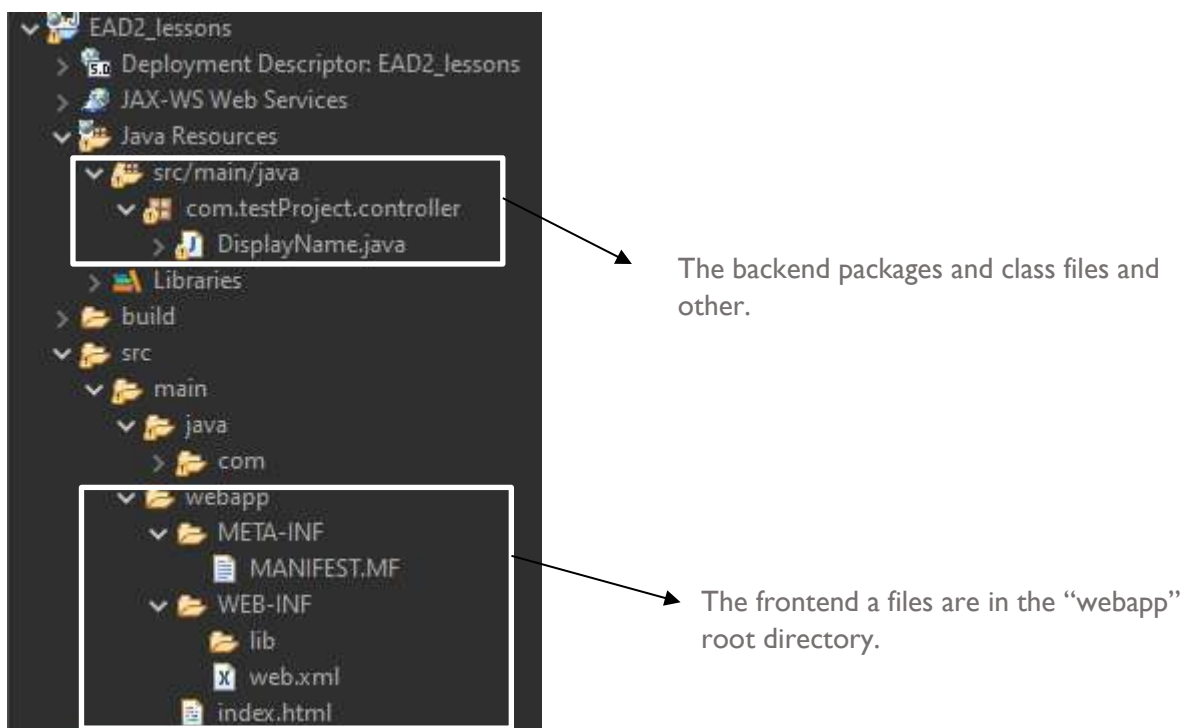
- To work with servlets, create a web project as below.
- Go to File → new → other and select Dynamic Web Project.



- And insert the project name and select the server that you are going to use in the project.

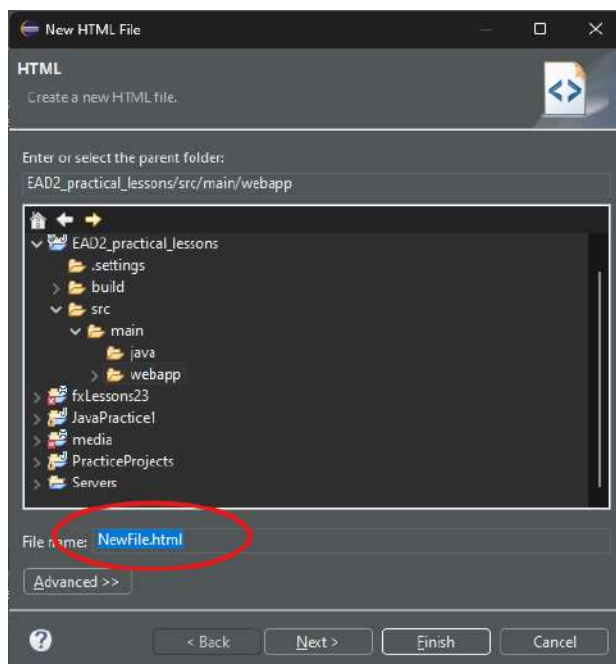
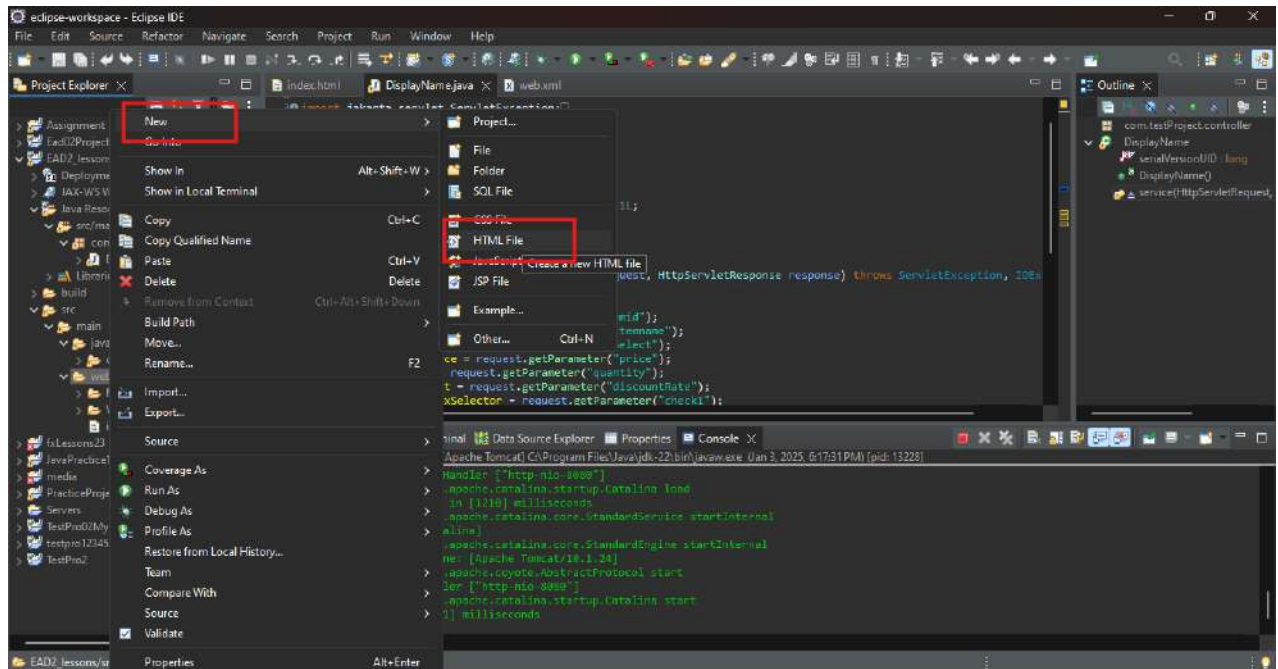


- Then the created project will be displayed as below.

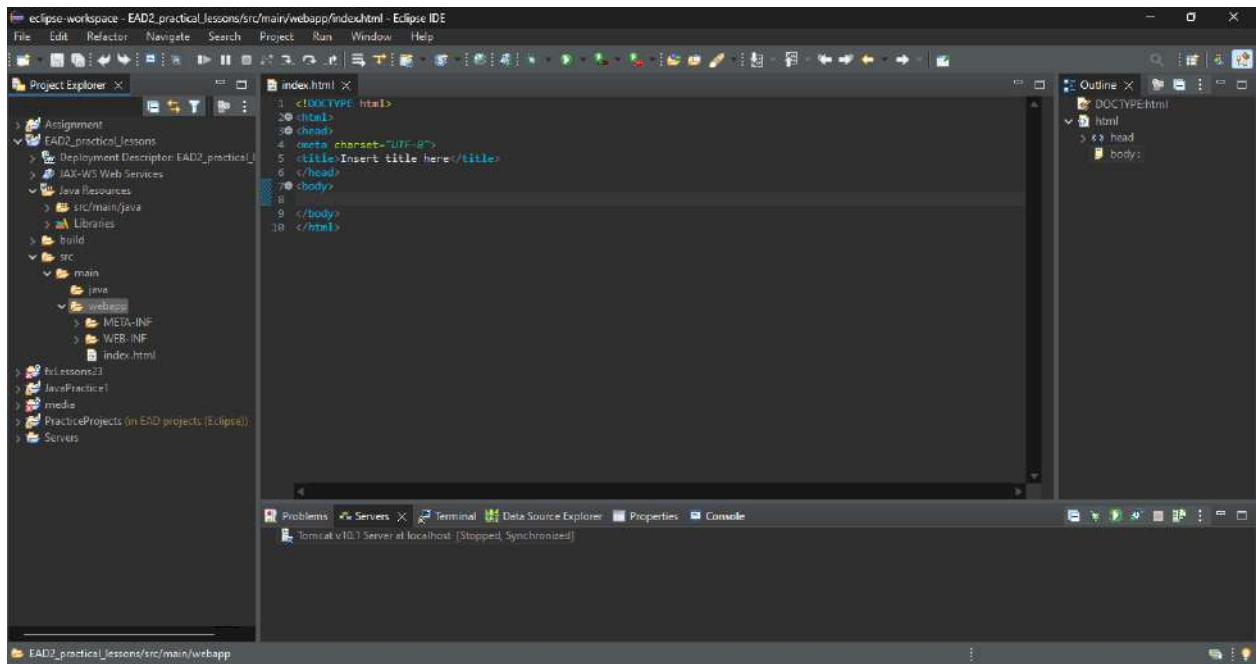


CREATING THE FRONT END OF THE PROJECT

- First create a HTML file to create the front end of the project as below (right click in the webapp folder and create a HTML file).

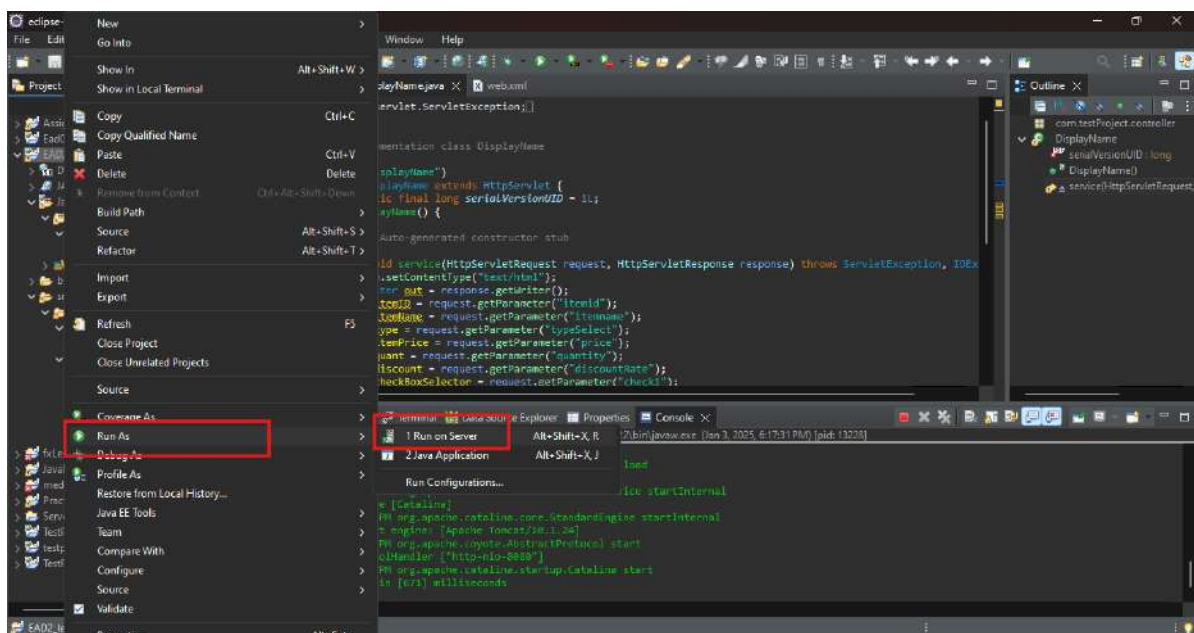


- When naming the first HTML file, name it as index.html or you need to configure that name in the web.xml.
- The created HTML file is shown below.

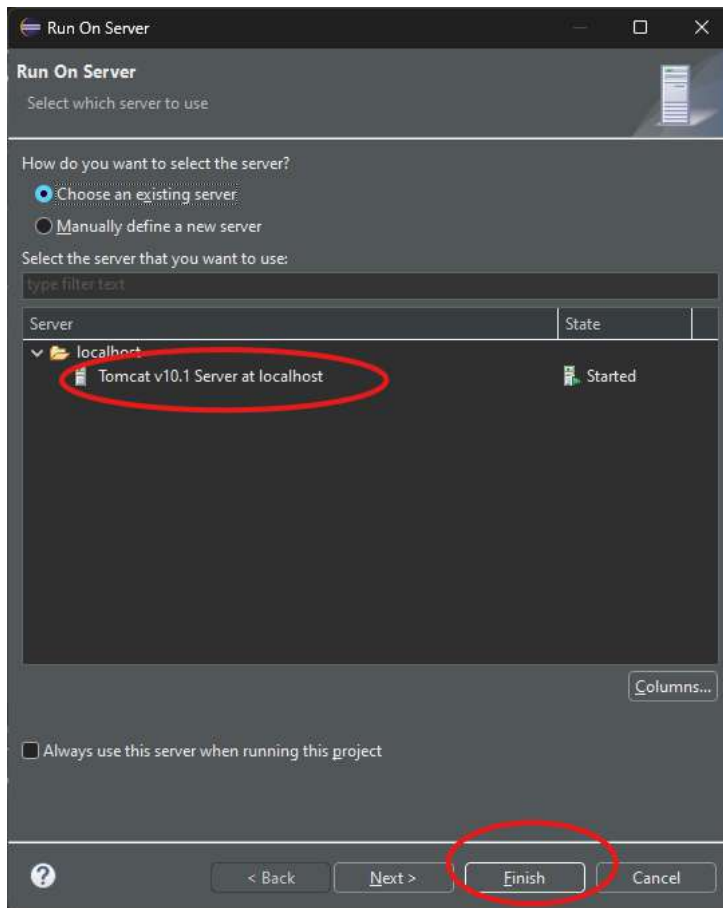


RUNNING THE WEB APPLICATION

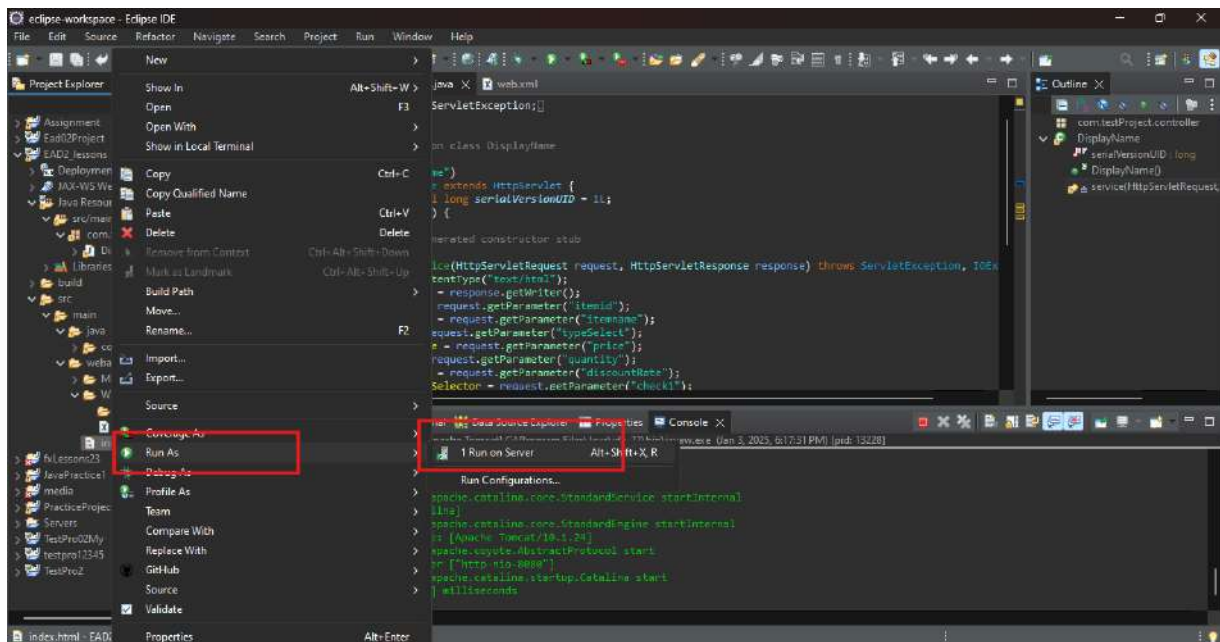
- First start the server as above and right click the project and start the application as following.



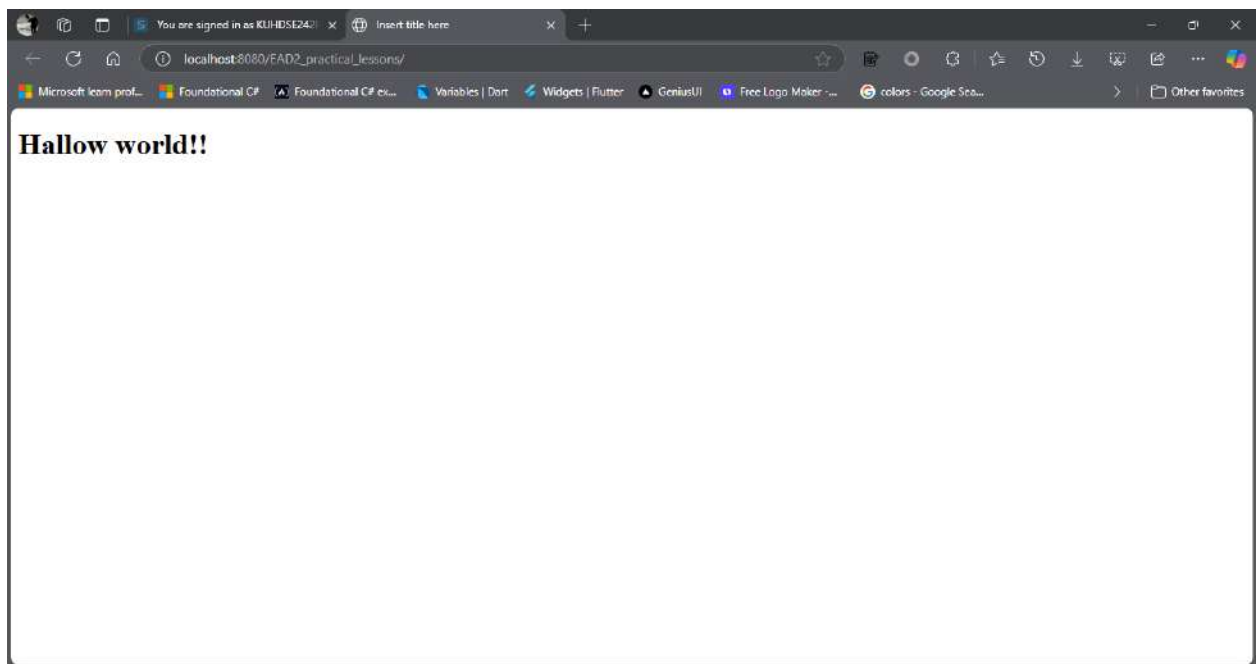
- Then select the required server as below and click finish.



- Or if you want to run a different HTML file, right click on the required HTML file and follow these steps.

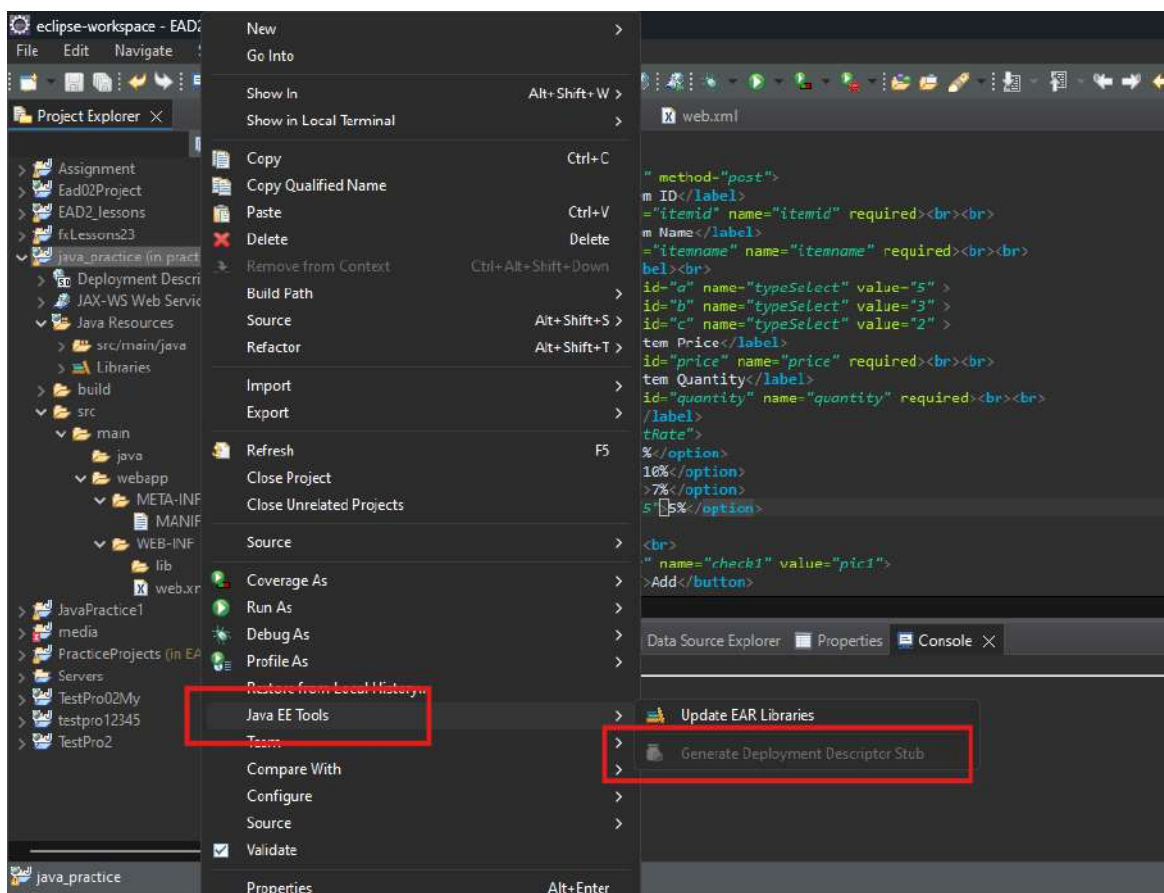


- Like before, select the server and run the application.
- Result →



GET THE WEB.XML FILE

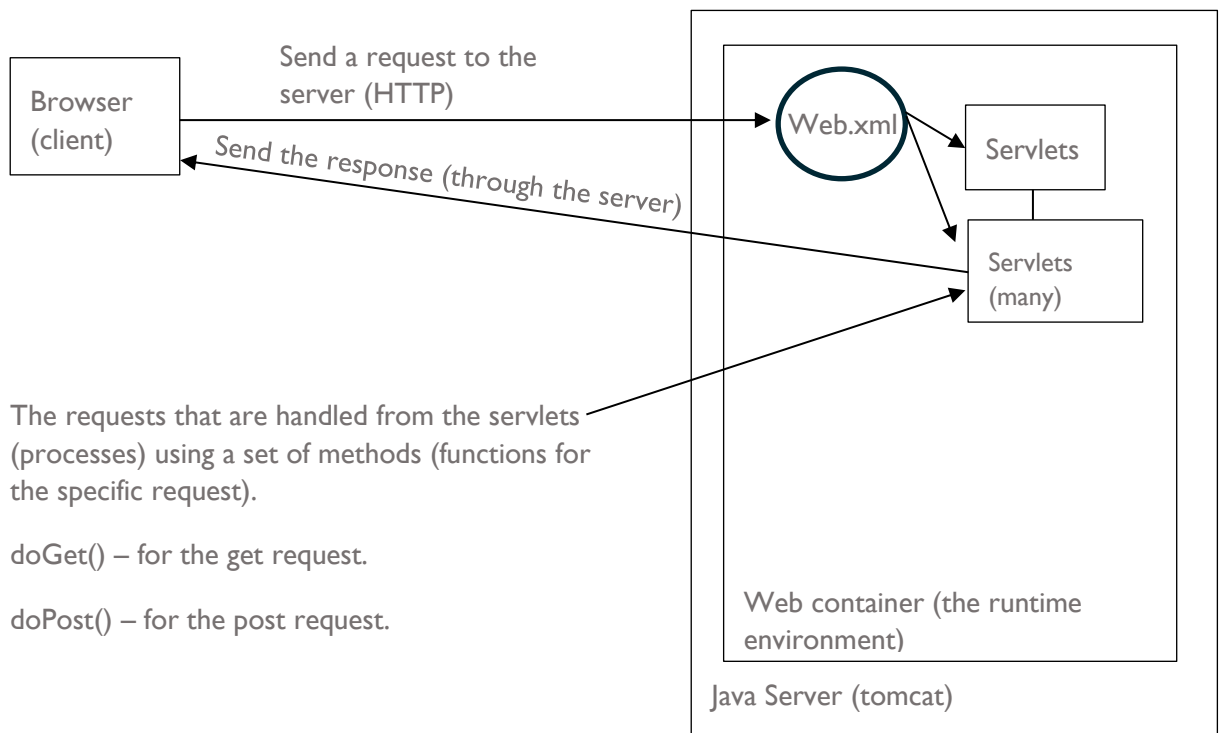
- Right click on the created project and do this.



CREATE THE SERVLET

- A servlet is a Java component that can be plugged into a Java-enabled web server to provide custom services. and then send a response back to the web server.

- Servlets work on the server side. Servlets are capable of handling complex requests obtained from the web server.
- How it works?



- This servlet is a java class which is inherited from the servlet class (HttpServlet).

```
@WebServlet("/task1")
public class task1 extends HttpServlet {
```

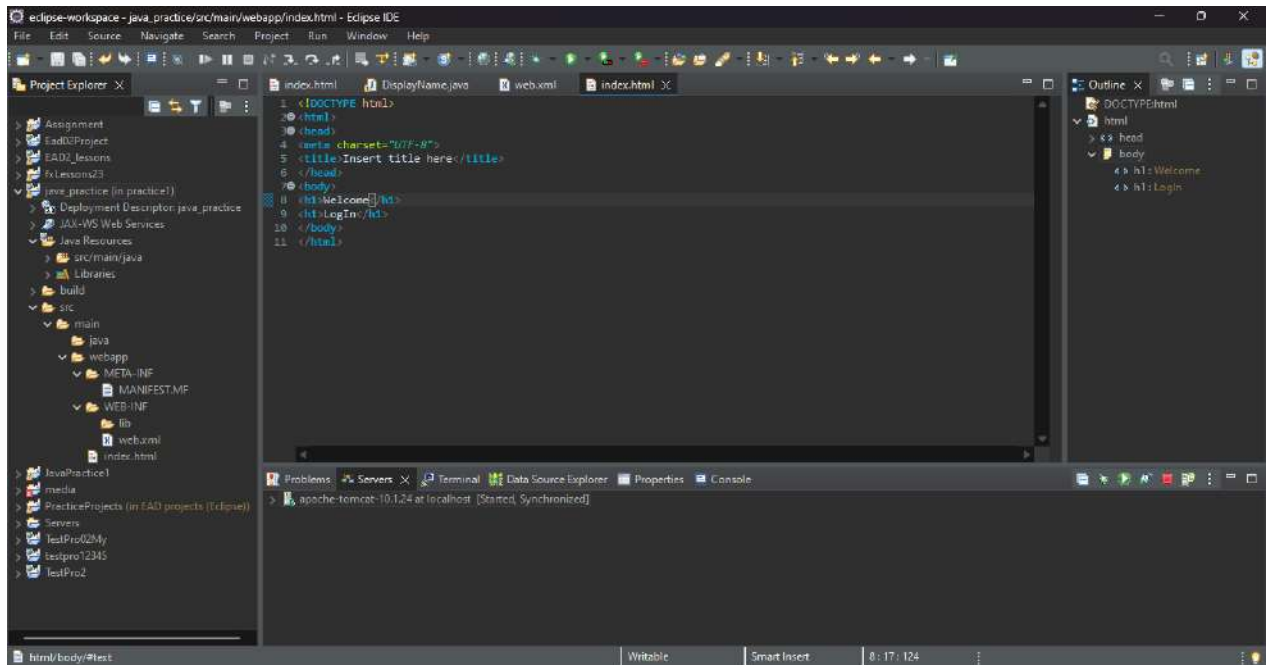
- According to the inheritance concept, functions (or the methods for the request handling) are the implementations of the various request handling methods (in the HttpServlet). Just calling the functions to process something in the inheriting class while inserting some information within it (these are corresponds to the HTTP request types).
- Example: when someone sends a HTTP request, that request go to the servlet and routes that request to the (using any backend process in the server) the methods related to the request type (eg: doGet()) while accessing to the different pieces of the request like resource and the request body.

Request ————> Specific method (like doGet())

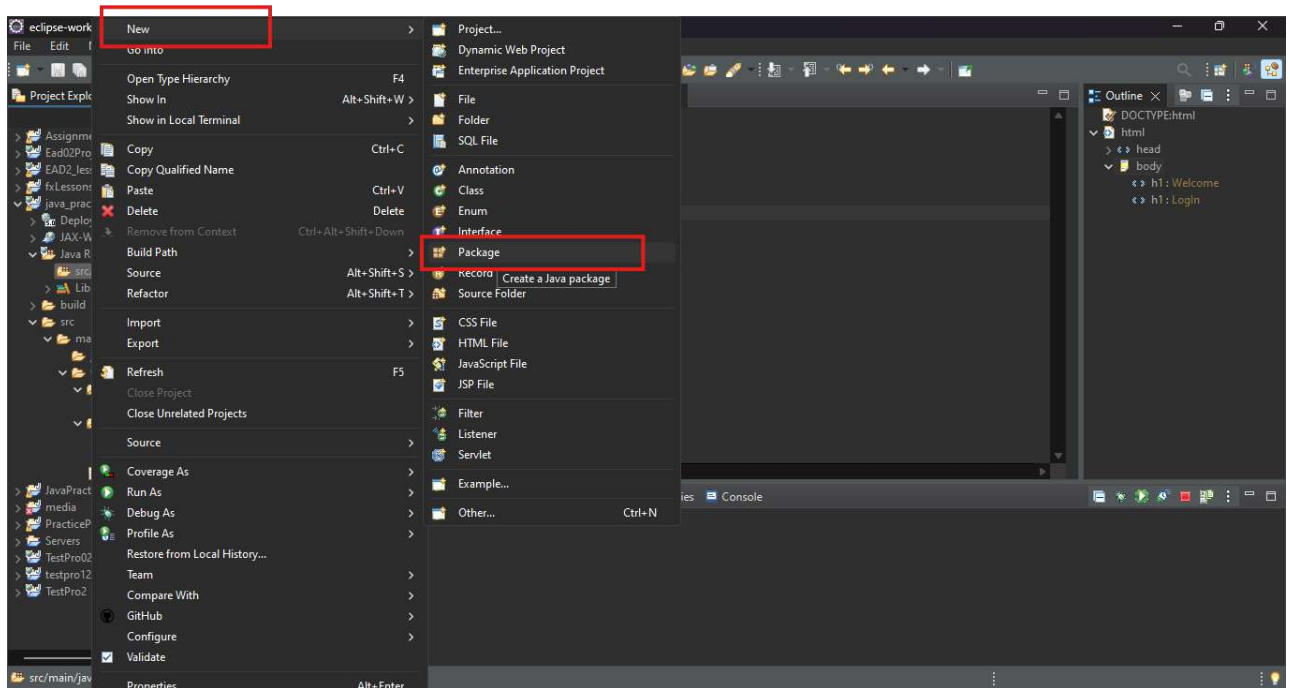
Routes (automatically occurs. Like calling
a method in the main method)

- Also, when there are many servlets, the server routes the request to the specific servlet. This is done using the web.xml file. For an example, when the client send a request like “post/bar”, the server looks in the web.xml file and finds the servlet that it should go to then routes the request to the appropriate servlet based on the web.xml file.
- **Note – A servlet container (also known as a web container) is the component of a web server that interacts with Java servlets. A web container is responsible for managing the lifecycle of servlets. (or acts as a runtime environment for the servlets). The servlets process the requests.**
- Main process.

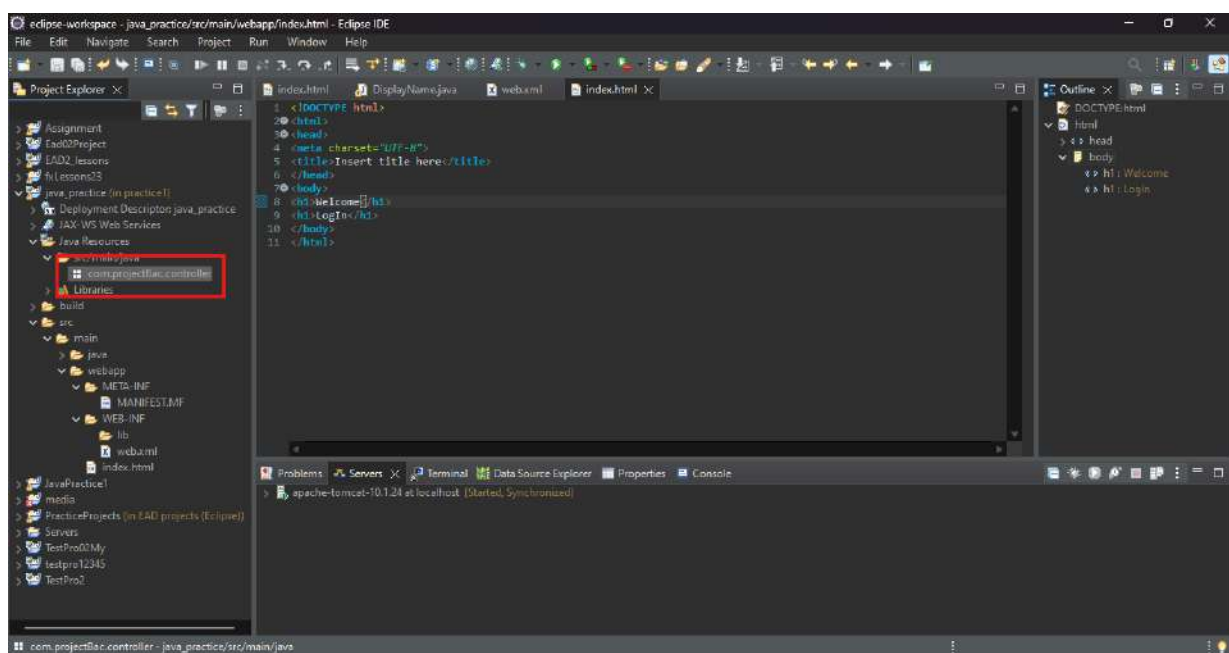
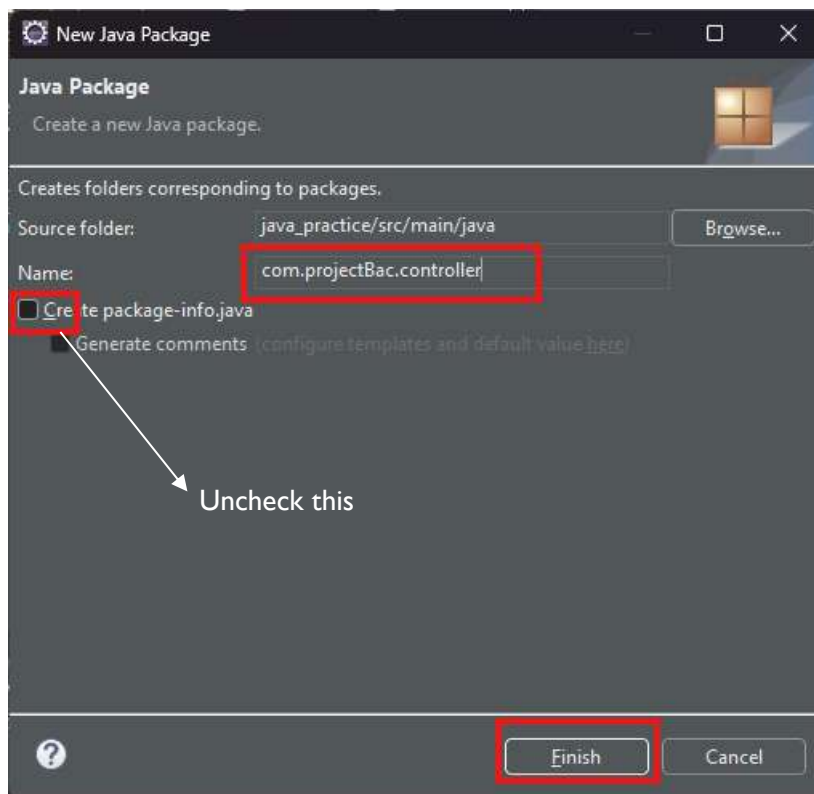
- The Clients send the request to the Web Server.
 - The Web Server receives the request.
 - The Web Server passes the request to the corresponding servlet.
 - The Servlet processes the request and generates the response in the form of output.
 - The Servlet sends the response back to the webserver.
 - The Web Server sends the response back to the client and the client browser displays it on the screen.
- Create the front end as before and develop the HTML content. Then test the front end.



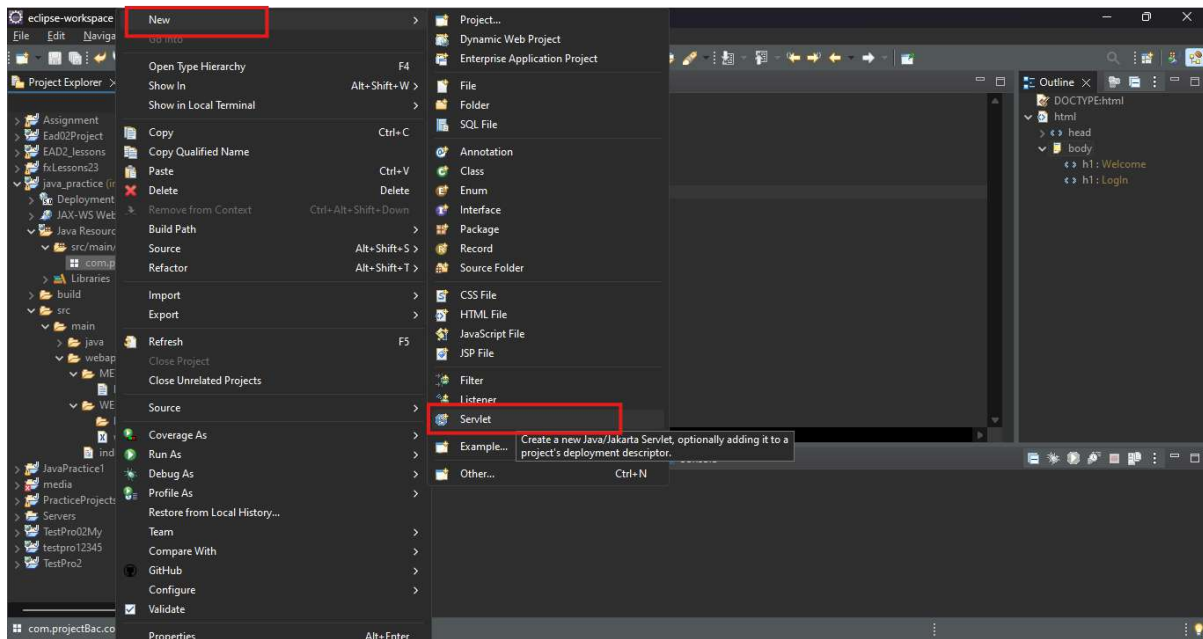
- Create a package in the SRC (src/main/java) folder (for the backend).



- When naming the package, insert the name like this, com.name_of_the_package.controller.



- Now create the servlet as follows. Right click on the created package and create the servlet inside of it.



- Insert the class name and create it.



- The code related to the requests.

```
package com.projectBac.controller;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```

@WebServlet("/task1")

public class task1 extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public task1() {

        super();

        // TODO Auto-generated constructor stub

    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        response.getWriter().append("Served at: ").append(request.getContextPath());

    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        // TODO Auto-generated method stub

        doGet(request, response);

    }

}

```

- Or we can use the “service” method which is a method that accepts both POST and GET requests.

```

protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    System.out.println("Hallow"); //prints an output from the servlet
}

```

- **HttpServletRequest request:** Represents the client's HTTP request.
- **HttpServletResponse response:** Represents the servlet's response to the client.

INTEGRATING HTML CONTENTS WITH SERVLETS

- In index.html or in any html file, create the HTML content and create a form for the actions.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>Welcome</h1>
<h1>LogIn</h1>
<form action="task1" method="post">
<button>Test</button>
</form>

```



```
</body>
</html>
```

- As the task, insert the name of the java file (or the class) which the servlet locate and insert the method as POST or GET.
- Then run (like before) that in the server and the output will print in the terminal.
- Java code :

```
package com.projectBac.controller;

import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/task1")

public class task1 extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public task1() {
        super();
    }

    protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        System.out.println("Hallow");

    }

}
```

SENDING DATA BETWEEN 2 SERVLETS

- The sending servlet :

```
protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    //System.out.println("Hallow");
    // Setting an attribute to pass data
    request.setAttribute("message", "Hello from SenderServlet!");

    // Forwarding the request to ReceiverServlet
    RequestDispatcher dispatcher = request.getRequestDispatcher("/t");
    dispatcher.forward(request, response);
}
```

- **request.setAttribute("message", "Hello from SenderServlet!");** : This line adds an attribute to the HttpServletRequest object. "message" is the name of the attribute. "Hello from SenderServlet!" is the value assigned to the key. This attribute is attached to the current HTTP request and can be retrieved

later by another servlet or JSP using `request.getAttribute("message")`. It is attached to the `HttpServletRequest` object, which represents the HTTP request received by the servlet.

- **RequestDispatcher dispatcher = request.getRequestDispatcher("/t");** : This line obtains a `RequestDispatcher` object for the **target resource. Path (/t)**: The target resource (another servlet, JSP, or static resource) is identified using the path. In this case: `"/t"` refers to a resource relative to the web application's root. **Effect**: The `RequestDispatcher` is used to forward the current request and response to the specified target resource.
- **dispatcher.forward(request, response);** : This line forwards the current request and response objects to the target resource (`/t`). Control is transferred to the resource specified in `getRequestDispatcher`. The target resource processes the same request and generates the response. Attributes set in the request object (like `"message"`) remain accessible to the target resource.
- In Java servlets, an attribute is a key-value pair used to share data between components (like servlets, JSPs, or filters) within the same application and during the same request or session lifecycle.

Attributes are stored in the following scope objects:

- Request Scope (`HttpServletRequest`)
- Session Scope (`HttpSession`)
- Application Scope (`ServletContext`)
- Receiving servlet (in the second servlet):

```
protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    //System.out.println("Hallow");
    // Retrieving the attribute set by SenderServlet
    String message = (String) request.getAttribute("message");

    response.setContentType("text/plain");
    response.getWriter().write("Received message: " + message);
}
```

- The HTML code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>Welcome</h1>
<h1>LogIn</h1>
<form action="task1" method="post">
<button>Test</button>
</form>
</body>
</html>
```

SESSION HANDLING IN JAVA SERVLETS

- HTTP is a “stateless” protocol, which means that each time a client requests a Web page, the client establishes a new connection with the Web server, and the server does not retain track of prior requests.

- Session tracking is the process of remembering and documenting customer conversations over time. Session management is another name for it.
- The term “stateful web application” refers to a web application that is capable of remembering and recording client conversations over time.
- The conversation of a user over a period of time is referred to as a session. In general, it refers to a certain period of time.
- The recording of the object in session is known as tracking.
- **Stateless: Each request from a client to the server is treated as an independent transaction that is unrelated to any previous request. The server does not retain any information about previous interactions.**
- **Stateful: The server maintains information about the client's state across multiple requests. This allows the server to remember previous interactions and provide a continuous experience.**
- Session Tracking employs Four Different techniques
 - Cookies
 - Hidden Form Field
 - URL Rewriting
 - HTTP Session
- Also, when we use another servlet, we can use the stored data with the session ID without any issues.

SESSION HANDLING USING HTTP SESSION

- First, create the HTML page as below (needed content with the form tag).

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>LogIn</h1>
<form action="task1" method="post">
<input type="text" name="userName" id="userName" placeholder="User Name"><br><br>
<input type="text" name="pass" id="pass" placeholder="Password"><br><br>
<button>Login</button>
</form>
</body>
</html>
```

- Then create the first servlet (to create and store the user data in the session).

```
protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        response.setContentType("text/html");
        String name = request.getParameter("userName");
        PrintWriter out = response.getWriter();
        out.println("Welcome" + name);
        HttpSession session=request.getSession();
        session.setAttribute("uname",name);
        out.print("<a href='task2'>visit</a>");
    }
}
```

```

        out.close();
    } catch (Exception ee) {
        System.out.println(ee.toString());
    }
}

```

- Explanation <later>
- We are creating another servlet to access the session data. We go to that servlet using an <a> tag with the href="second_servlet_name". Then we can move to the second servlet to access the session data.

```

protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        HttpSession session=request.getSession(false);
        String n=(String)session.getAttribute("uname");
        out.print("Hello "+n);

        out.close();
    } catch (Exception ee) {
        System.out.println(ee.toString());
    }
}

```

- HTML output.

Login

- First servlet output.

```
WelcomeRaveen visit
```

- Second servlet output.

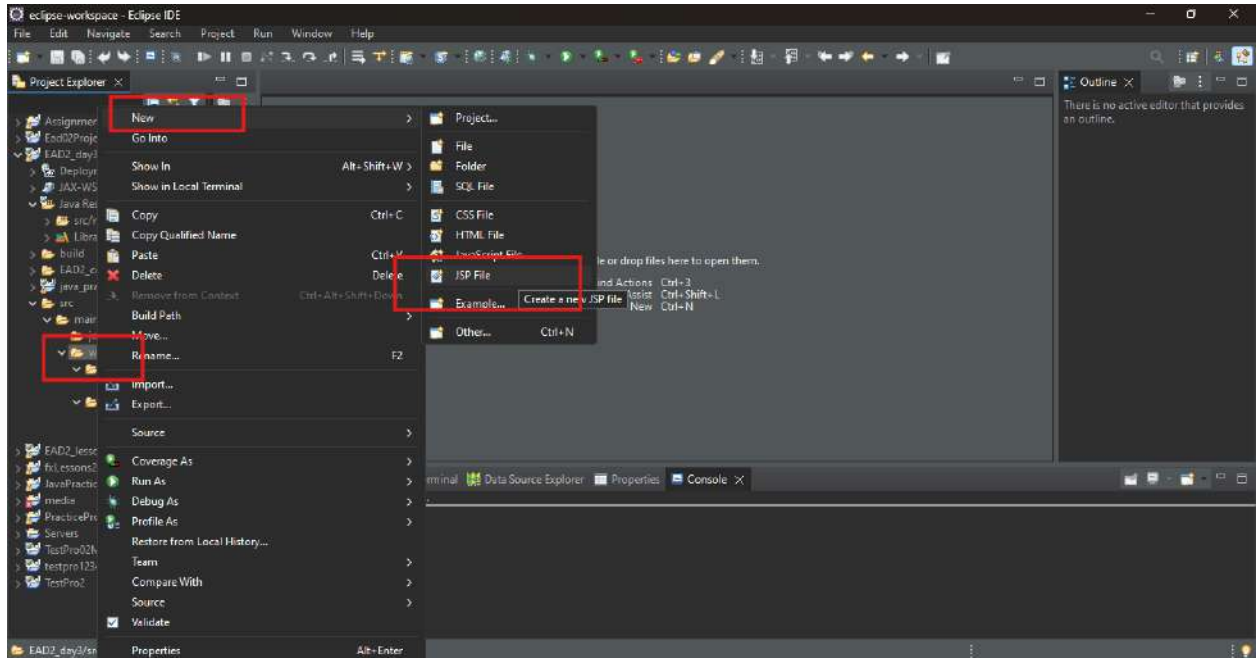
Hello Raveen

JAVA JSP

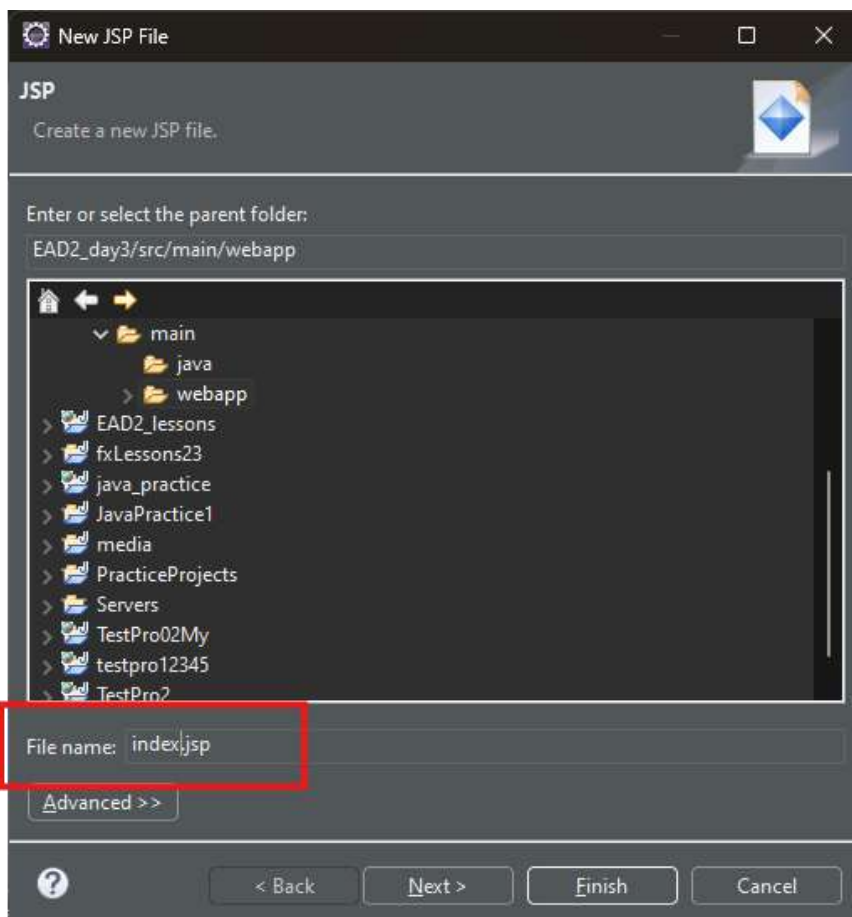
- JSP is like PHP, and we can insert HTML and use java language as the server-side language.

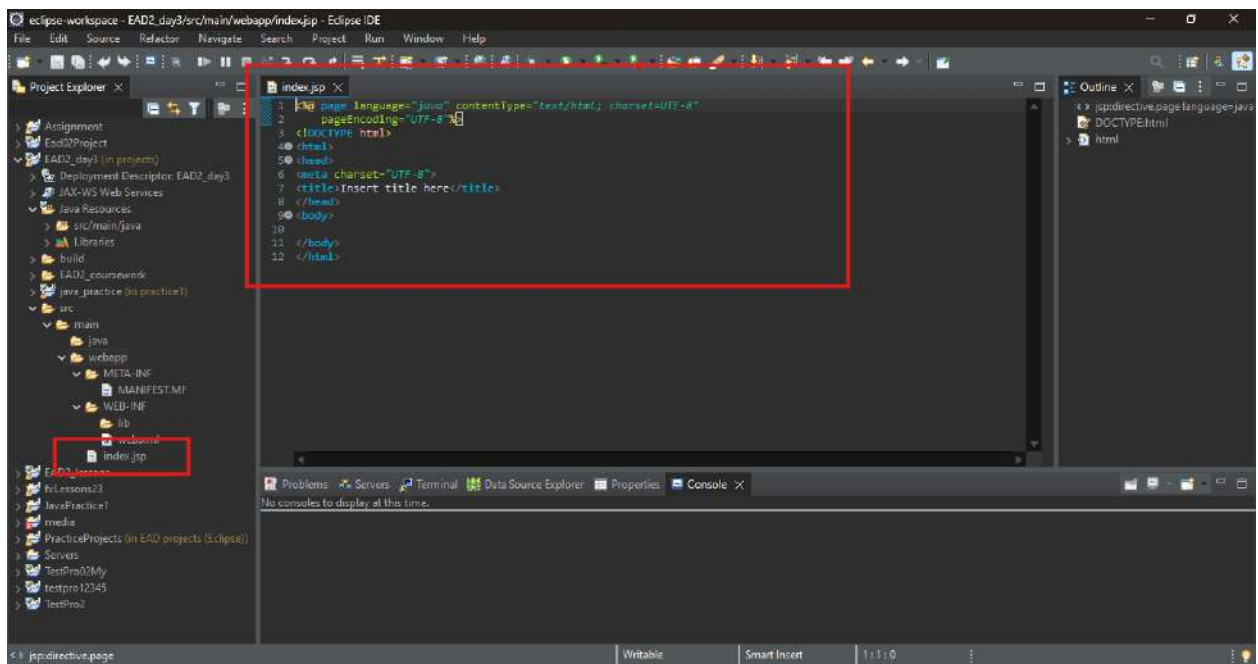
CREATING A JSP FILE

- In the webapp folder (root), create JSP file as follows,



- Now insert a name to the JSP file.





INTEGRATING JAVA LANGUAGE WITH HTML PART

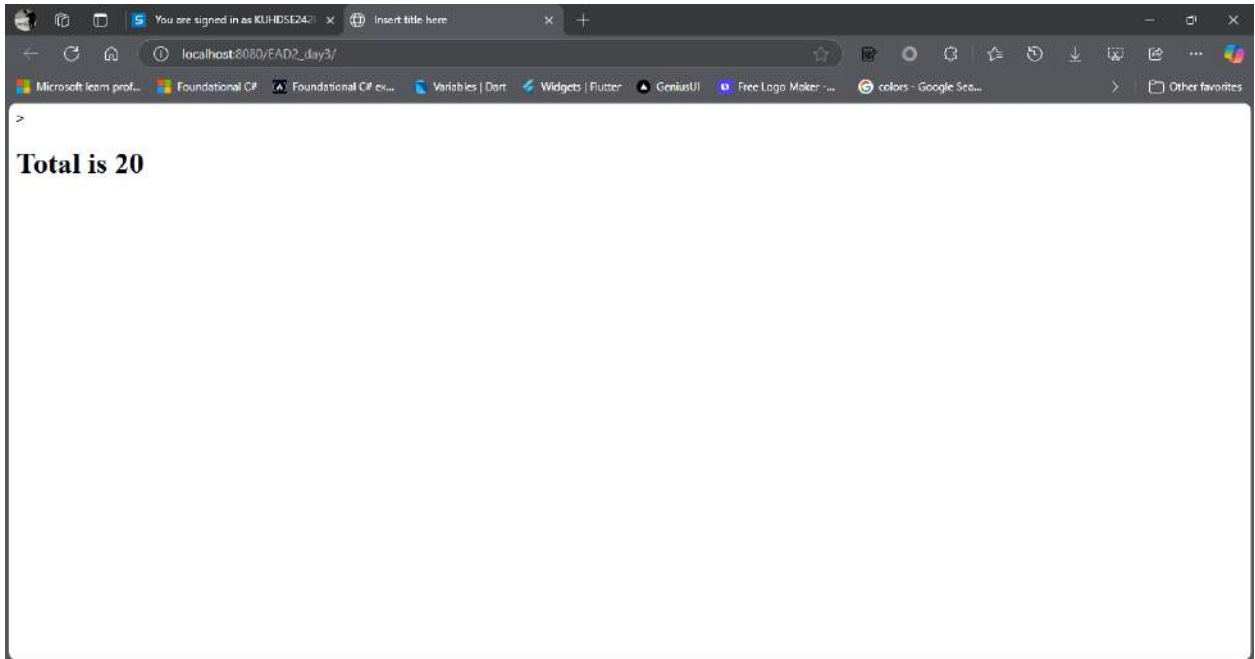
In Java Server Pages (JSP), the tags `<% %>`, `<%= %>`, and `<%! %>` serve distinct purposes for embedding Java code within HTML. The `<% %>` tag, known as a *scriptlet*, allows developers to insert executable Java code directly into the JSP page. This code executes during the request-processing phase and is typically used for logic like loops or conditional statements. The `<%= %>` tag, an *expression tag*, evaluates a Java expression and outputs its result directly into the HTML response, making it useful for dynamically rendering values (e.g., variables or method returns). The `<%! %>` tag, a *declaration tag*, defines class-level variables or methods in the generated servlet, allowing them to persist across multiple requests. While scriptlets and declarations were once common, modern JSP best practices often favor Expression Language (EL) or JSTL to separate logic from presentation, improving maintainability and reducing complexity.

- `<% %>` and `<%= %>` tag usage to integrate java codes.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

<%
int no1 = 10;
int no2 = 10;
int tot = no1 + no2;
%>
<h1>Total is <%= tot %></h1>
</body>
</html>
```

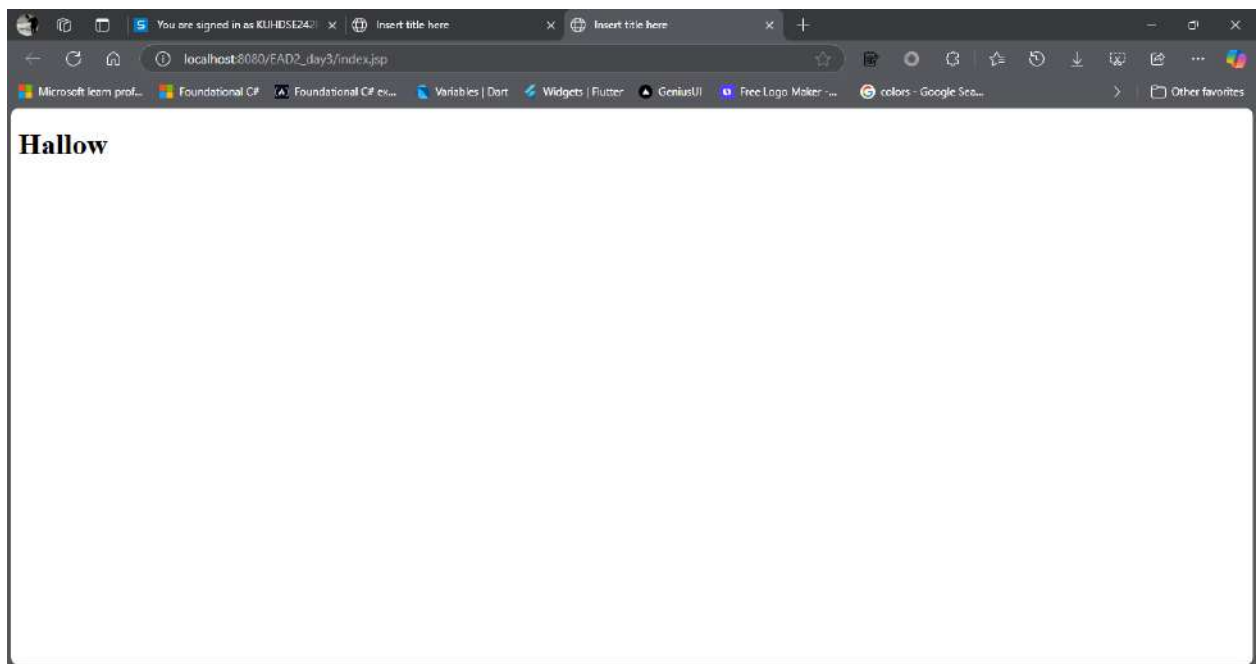
- Output →



- Use of the `<%!%>`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<%!
public String getMessage(){
    return "Hallow"; //process that is in the global (not in the servlet)
}
%>
<h1><%= getMessage() %></h1> //calling section of the global function in the JSP
</body>
</html>
```

- Output →



PROCESSING FROM JSP TO SERVLETS

- In the JSP (index.jsp)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<form method="post" action="CalculateServlet">
<h1>Employee salary calculator</h1>
<label>Employee ID</label>
<input type="text" name="emID" id="emID"><br><br>
<label>Employee Name</label>
<input type="text" name="emName" id="emName"><br><br>
<label>Employee Basic salary</label>
<input type="text" name="salary" id="salary"><br><br>
<label>Bonus</label>
<input type="text" name="bonus" id="bonus"><br><br>
<button>Calculate</button>
<button>Reset</button>
</form>
</body>
</html>
```

- In the servlet

```
protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        response.setContentType("text/html");
        String emID = request.getParameter("emID");
        String emName = request.getParameter("emName");
        double basic = Double.parseDouble(request.getParameter("salary"));
        double bonus = Double.parseDouble(request.getParameter("bonus"));
        double tot = basic + bonus;
        PrintWriter out = response.getWriter();
```

```

        // Provides a PrintWriter object that can be used to send text output (such
as HTML content) to the client.
        out.println("Total: " + tot + "\nEmployee ID: " + emID + "\nEmployee name: " +
emName);
    } catch (Exception ee) {
        System.out.println(ee.toString());
    }

    // Writes the value of the name variable to the HTTP response. If name is null,
the output will display null.

}

```

JSP TO SAME JSP

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<%
String emID = request.getParameter("emID");
String emName = request.getParameter("emName");
double basic = Double.parseDouble(request.getParameter("salary"));
double bonus = Double.parseDouble(request.getParameter("bonus"));
double tot = basic + bonus;
out.println("Total: " + tot + "\nEmployee ID: " + emID + "\nEmployee name: " + emName);
%>
</body>
</html>

```

Like “out” they has already been created (implicit objects)

Another example – request

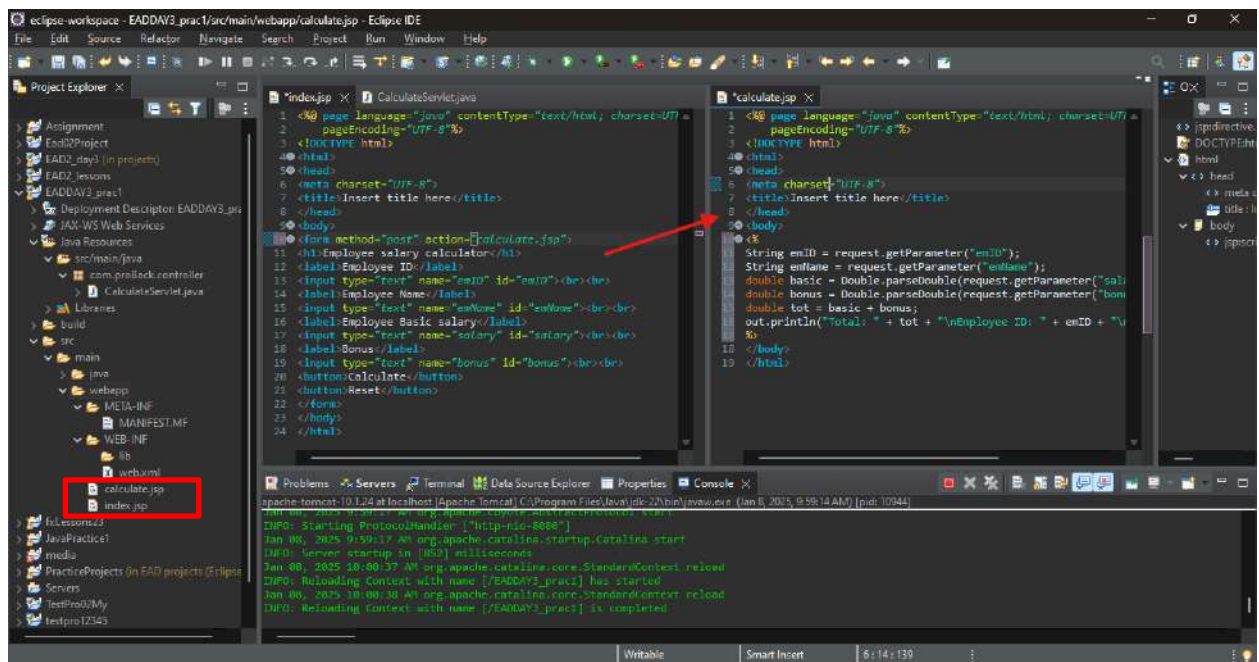
- The main reason for the presence of these objects, is JSP's are act as a servlet (so, we do not need to declare the objects but in the .java servlets files, we have to do that)
- The main specialty of JSP is we can integrate java codes (servlet codes) in the JSP HTML codes. (loke PHP codes within HTML codes.) but we can't do this in servlet files. To work with java servlets, JSP files will act as a servlet.
- In here we are running java backend with the html codes (like php in html codes within the jsp file).

CONNECTING TWO JSP FILES (JSP FRONTEND TO JSP BACKEND)

- First create the front end (in the index.jsp) as follows.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<form method="post" action="calculate.jsp"> //with the .jsp tag (the jsp file to jsp as the backend)
<h1>Employee salary calculator</h1>
<label>Employee ID</label>
<input type="text" name="emID" id="emID"><br><br>
<label>Employee Name</label>
<input type="text" name="emName" id="emName"><br><br>
<label>Employee Basic salary</label>
<input type="text" name="salary" id="salary"><br><br>
<label>Bonus</label>
<input type="text" name="bonus" id="bonus"><br><br>
<button>Calculate</button>
<button>Reset</button>
</form>
</body>
</html>
```

- Now create the backend (for the index.jsp) in a separate JSP file. (with implicit objects).
- Also, create the processing JSP file as follows,



```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
```

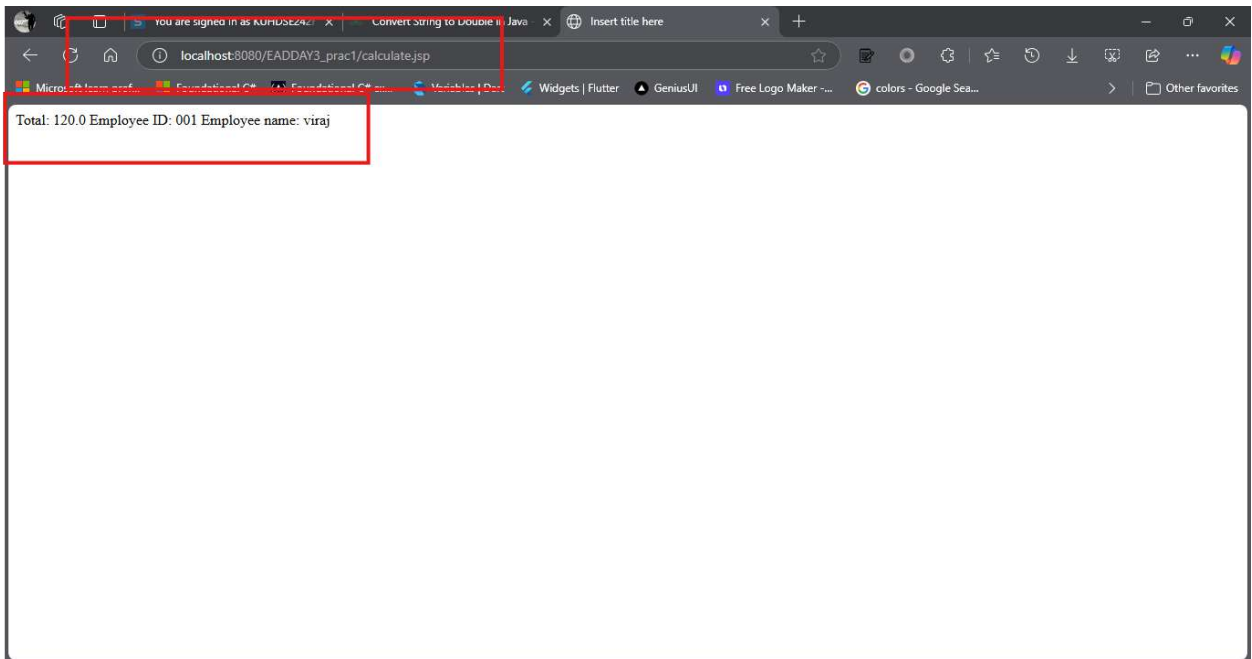


```

<title>Insert title here</title>
</head>
<body>
<%
String emID = request.getParameter("emID");
String emName = request.getParameter("emName");
double basic = Double.parseDouble(request.getParameter("salary"));
double bonus = Double.parseDouble(request.getParameter("bonus"));
double tot = basic + bonus;
out.println("Total: " + tot + "\nEmployee ID: " + emID + "\nEmployee name: " + emName);
%>
</body>
</html>

```

- Output →



JSP TO SERVLET TO JSP (METHOD I)

- Create the frontend as follows,

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<form method="post" action="calculate">

<h1>Calculate</h1>
<label>Product ID</label>
<input type="text" name="pro_id" id="pro_id"><br><br>
<label>Product Name</label>
<input type="text" name="pro_name" id="pro_name"><br><br>
<label>Price</label>
<input type="text" name="price" id="price"><br><br>
<label>Quantity</label>
<input type="text" name="qt" id="qt"><br><br>
<select id="dis" name="dis">

```

```

<option value="5">5%</option>
<option value="10">10%</option>
<option value="15">15%</option>
</select><br><br>
<label>Total</label>
<input type="text" name="totalShow" id="totalShow" value="<%= request.getAttribute("Total")
%>"><br><br>
<button>Calculate</button>
<button>Reset</button>
</form>
</body>
</html>

```

- As before, send the data from JSP to servlet, now we have to send data from servlet to JSP,
- In the servlet,

```

protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        String pro_id = request.getParameter("pro_id");
        String pro_name = request.getParameter("pro_name");
        double price = Double.parseDouble(request.getParameter("price"));
        double qt = Double.parseDouble(request.getParameter("qt"));
        double dis = Double.parseDouble(request.getParameter("dis"));
        PrintWriter out = response.getWriter();
        double tot = qt * price;
        if(dis == 5) {
            tot = tot + (tot * 0.05);
        }else if(dis == 10) {
            tot = tot + (tot * 0.1);
        }else {
            tot = tot + (tot * 1.5);
        }
        out.print("Total: " + tot);
        HttpSession session=request.getSession();
        session.setAttribute("total",tot);
        session.setAttribute("name",pro_name);
        response.sendRedirect("display.jsp");
        out.close();
    }catch(Exception ee) {
        System.out.println(ee.toString());
    }
}

```

- Now enter this code for the second JSP.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<%
try {
    response.setContentType("text/html");
    session=request.getSession(false);
    String n=(String)session.getAttribute("total").toString();
    String name=(String)session.getAttribute("name").toString();
    out.print("total " + n + "Name" + name);
}catch(Exception ee) {

```

```

System.out.println(ee.toString());
}

%>
</body>
</html>

```

MOVING BETWEEN DIFFERENT PAGES

```

protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        String pro_id = request.getParameter("pro_id");
        String pro_name = request.getParameter("pro_name");
        double price = Double.parseDouble(request.getParameter("price"));
        double qt = Double.parseDouble(request.getParameter("qt"));
        double dis = Double.parseDouble(request.getParameter("dis"));
        PrintWriter out = response.getWriter();
        double tot = qt * price;
        if(dis == 5) {
            tot = tot + (tot * 0.05);
        } else if(dis == 10) {
            tot = tot + (tot * 0.1);
        } else {
            tot = tot + (tot * 1.5);
        }
        out.print("Total: " + tot);

        response.sendRedirect("display.jsp");
        out.close();
    } catch (Exception ee) {
        System.out.println(ee.toString());
    }
}

```

JSP TO SERVLET TO JSP (METHOD 2)

- Send the data from JSP to the servlet (like done before).
- Sending data from JSP → servlet.

```

protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        String pro_id = request.getParameter("pro_id");
        String pro_name = request.getParameter("pro_name");
        double price = Double.parseDouble(request.getParameter("price"));
        double qt = Double.parseDouble(request.getParameter("qt"));
        double dis = Double.parseDouble(request.getParameter("dis"));
        PrintWriter out = response.getWriter();
        double tot = qt * price;
    } catch (Exception ee) {
        System.out.println(ee.toString());
    }
}

```

- Now in the servlet, set the values to the attributes in the request (In the same servlet).

```

protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {

```

```
String pro_id = request.getParameter("pro_id");
String pro_name = request.getParameter("pro_name");
double price = Double.parseDouble(request.getParameter("price"));
double qt = Double.parseDouble(request.getParameter("qt"));
double dis = Double.parseDouble(request.getParameter("dis"));
PrintWriter out = response.getWriter();
double tot = qt * price;
if(dis == 5) {
    tot = tot + (tot * 0.05);
}else if(dis == 10) {
    tot = tot + (tot * 0.1);
}else {
    tot = tot + (tot * 1.5);
}
out.print("Total: " + tot);
```

```
request.setAttribute("productID",productID);
request.setAttribute("productName",name);
request.setAttribute("Total",total);
RequestDispatcher dispatcher = request.getRequestDispatcher("display.jsp");
dispatcher.foward(request,response);
```

```
out.close();
}catch(Exception ee) {
    System.out.println(ee.toString());
}
}
```

- Used front end for both methods.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<form method="post" action="calculate">

<h1>Calculate</h1>
<label>Product ID</label>
<input type="text" name="pro_id" id="pro_id"><br><br>
<label>Product Name</label>
<input type="text" name="pro_name" id="pro_name"><br><br>
<label>Price</label>
<input type="text" name="price" id="price"><br><br>
<label>Quantity</label>
<input type="text" name="qt" id="qt"><br><br>
<select id="dis" name="dis">
<option value="5">5%</option>
<option value="10">10%</option>
<option value="15">15%</option>
</select><br><br>
<label>Total</label>
<input type="text" name="totalShow" id="totalShow" value="<%= request.getAttribute("Total")
%>"><br><br>
<button>Calculate</button>
<button>Reset</button>
</form>
</body>
</html>
```

- After moving to the second JSP, obtain the attached data (from the request) in the second JSP.

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta charset="UTF-8">
```

```
    <title>Insert title here</title>
```

```
</head>
```

```
<body>
```

```
    <h1>Product Details</h1>
```

```
    <p><strong>Product ID:</strong> <%= request.getAttribute("productId") %></p>
```

```
    <p><strong>Product Name:</strong> <%= request.getAttribute("productName") %></p>
```

```
    <p><strong>Net Total:</strong> <%= request.getAttribute("netTotal") %></p>
```

```
    <a href="index.jsp">Calculate Again</a>
```

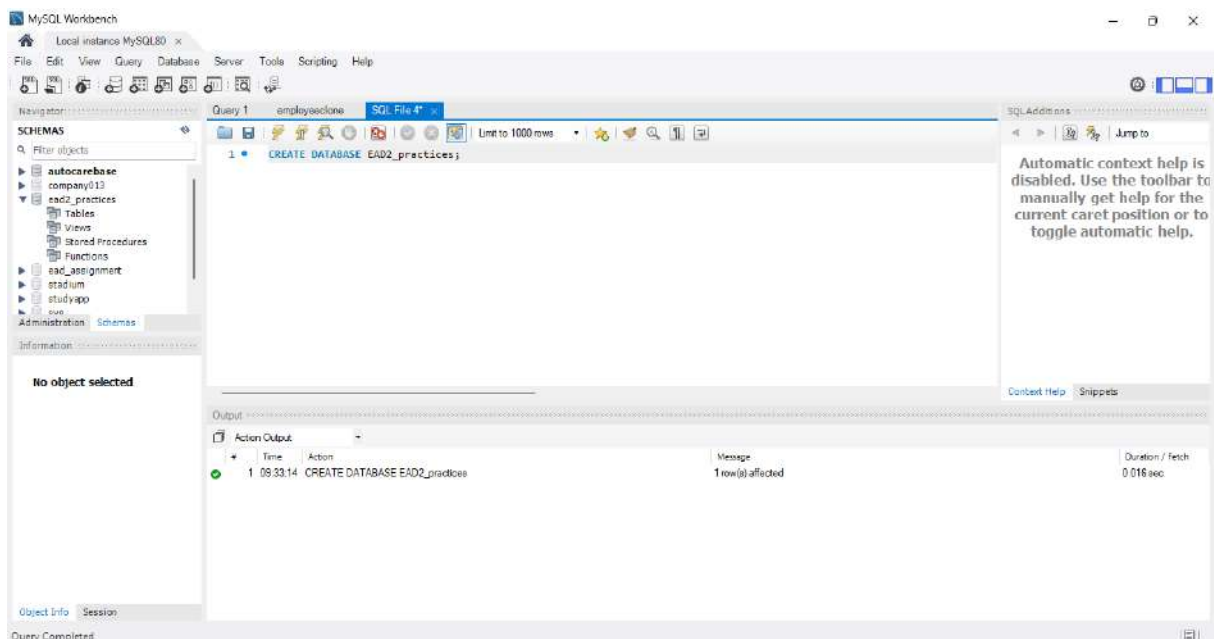
```
</body>
```

```
</html>
```

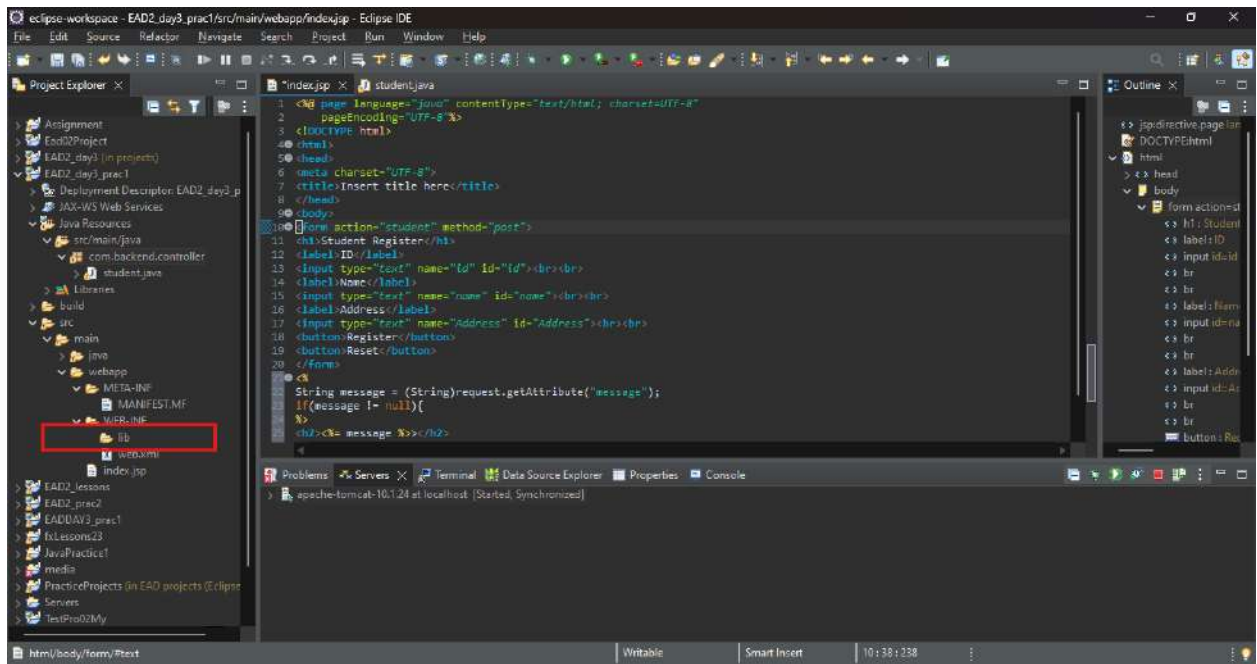
MVC DESIGN ARCHITECTURE

INTEGRATING MYSQL WORKBENCH

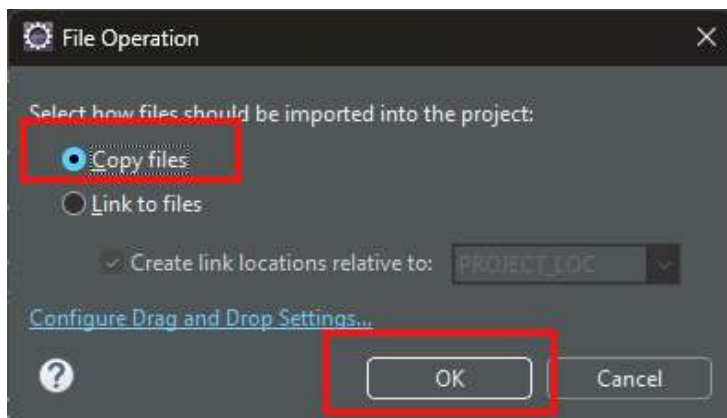
- First create the database in the current connection in the MYSQL workbench.



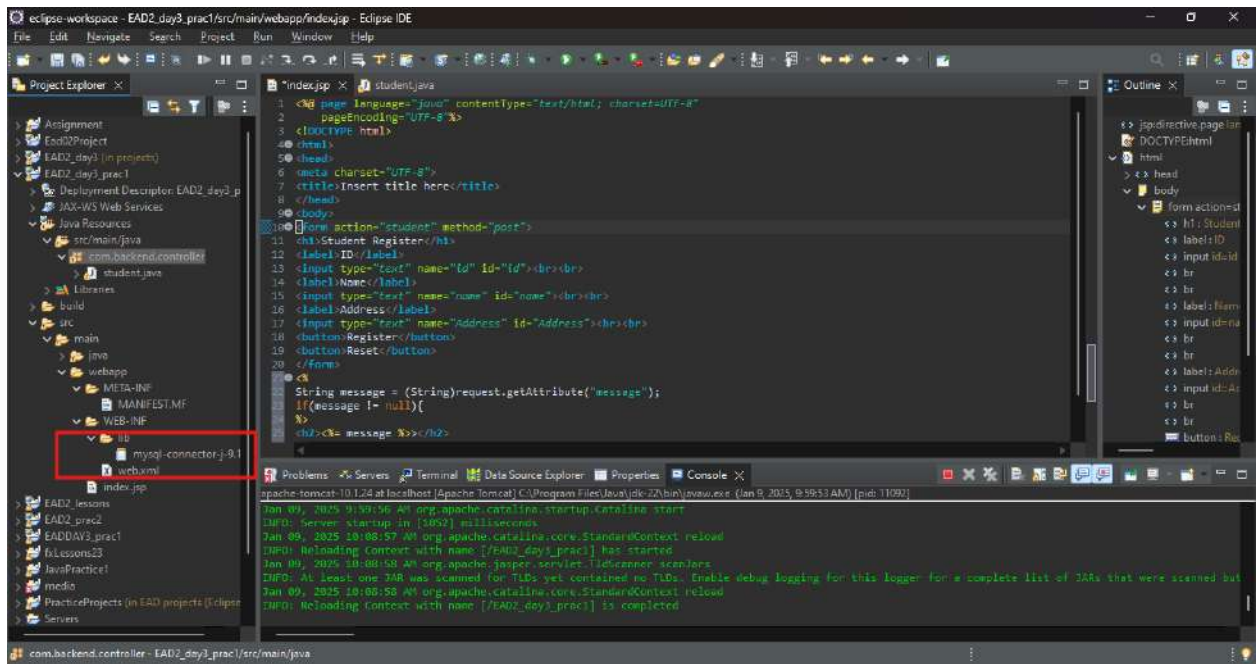
- Get the .jar file (related to MYSQL connectivity) from the maven repository and insert it to the java project.



- Copy the jar file as below. Do not link that.



- The inserted JAR file will be displayed as below,



- Create the frontend in the JSP page (index.jsp).

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<form action="student" method="post">
<h1>Student Register</h1>
<label>ID</label>
<input type="text" name="id" id="id"><br><br>
<label>Name</label>
<input type="text" name="name" id="name"><br><br>
<label>Address</label>
<input type="text" name="Address" id="Address"><br><br>
<button>Register</button>
<button>Reset</button>
</form>
<%
String message = (String)request.getAttribute("message");
if(message != null){
%>
<h2><%= message %></h2>
<% }%>
</body>
</html>
```

- In the servlet, obtain the data from the front end and send them to the database.

```

package com.backend.controller;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

```

```
@WebServlet("/student")
```

```
public class student extends HttpServlet {
```

Extend this class

Name of the database

Connection
information

```

    private static final long serialVersionUID = 1L;
    private String JDBC_URL = "jdbc:mysql://localhost:3306/lead2_practices";
    private String JDBC_USER = "root";
    private String JDBC_PASSWORD = "PASSWORD";

```

MYSQL port number

```
public student() {
```

```
    super();
```

```
    // TODO Auto-generated constructor stub
```

```
}
```

```

    protected void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

```

```
        try {
```

```
            response.setContentType("text/html");
```

```
            String id = request.getParameter("id");
```

```
            String name = request.getParameter("name");
```

```
            String address = request.getParameter("Address");
```

Obtained
data

```
            // PrintWriter out = response.getWriter();
```

```
            // out.println(id + "name" + name);
```

Creating the
connecting string

```

            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection connection = DriverManager.getConnection(JDBC_URL, JDBC_USER,
            JDBC_PASSWORD);

```

```
String sql = "INSERT INTO students VALUES (?, ?, ?);"
```

SQL Query

```
PreparedStatement statement = connection.prepareStatement(sql);
```

```
statement.setString(1, id);
```

```
statement.setString(2, name);
```

```
statement.setString(3, address);
```

Execute the
query

```
int rowsInserted = statement.executeUpdate();
```

```
connection.close();
```

```

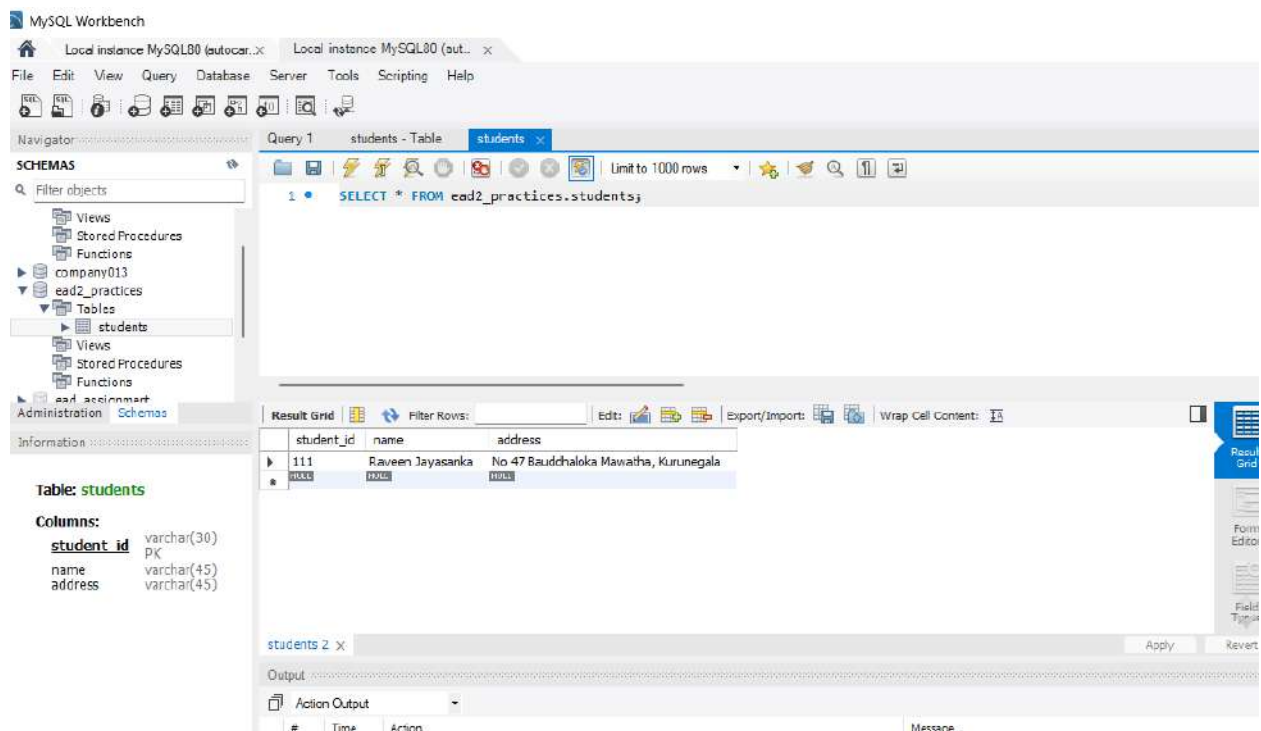
    if (rowsInserted > 0) {
        request.setAttribute("message", "Data saved successfully!");
    } else {
        request.setAttribute("message", "Failed to save data.");
    }

    }catch(Exception ee) {
        System.out.println(ee.toString());
    }

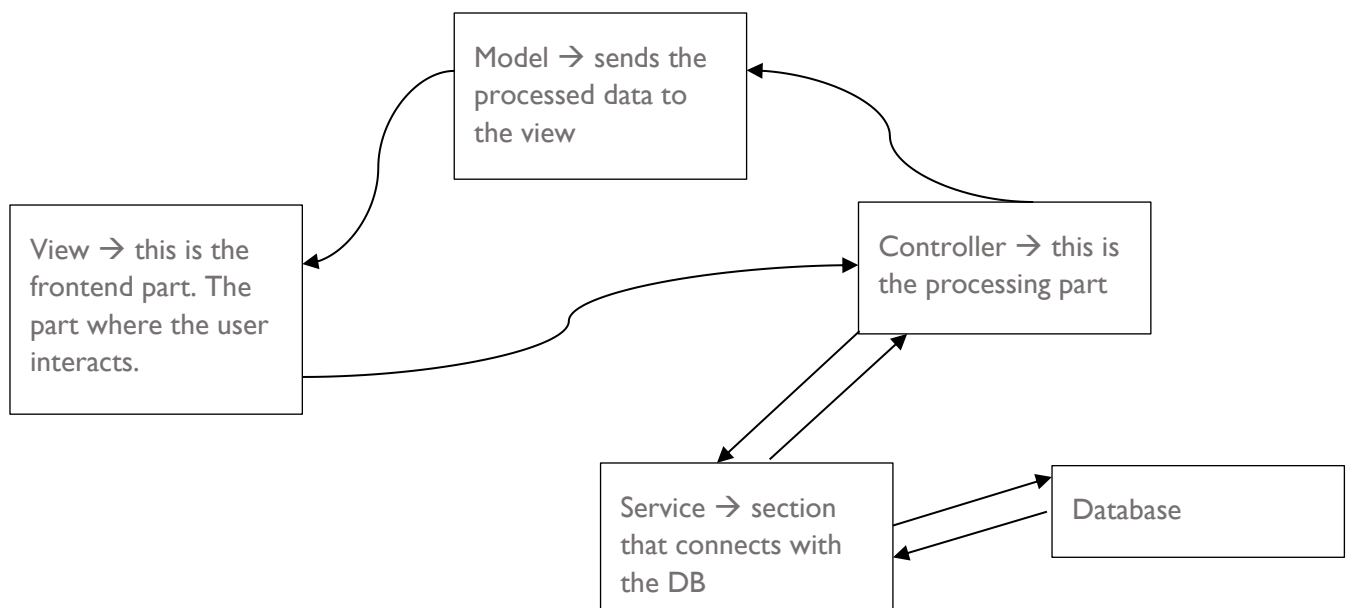
}
}
}

```

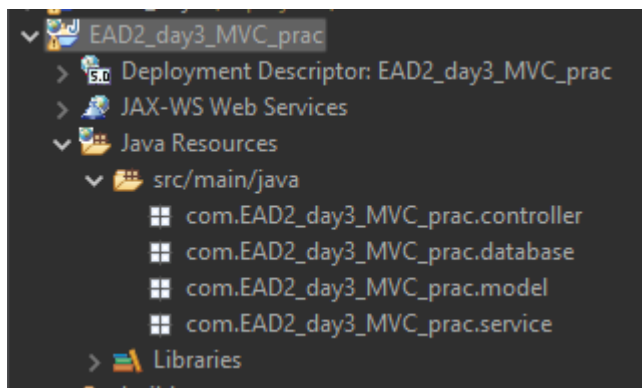
- Output →



PERFORMING MVC ARCHITECTURE CONCEPT



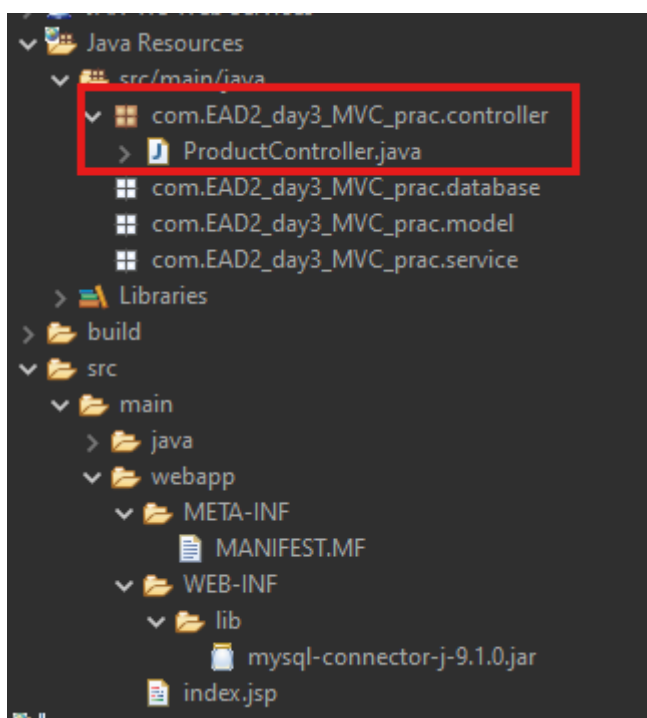
- Create these packages.



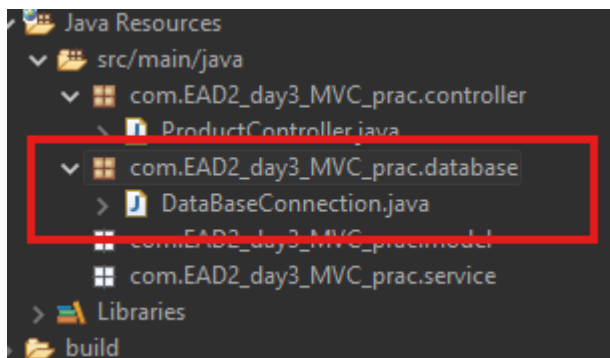
- First create the frontend part.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>Product registration</h1>
<form action="ProductController" method="post">
<label>Product number</label>
<input type="text" id="product_id" name="product_id" required><br><br>
<label>Name</label>
<input type="text" id="name" name="name" required><br><br>
<label>Price</label>
<input type="text" id="price" name="price" required><br><br>
<button type="submit">Enter</button>
</form>
</body>
</html>
```

- Now create the “controller” servlet in the controller package.



- Now create a java class file for the database connection (singalton).



- Create the database and perform the singalton pattern to connect to the database.

```
package com.EAD2_day3_MVC_prac.database;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class DataBaseConnection {
    private static final String JDBC_URL = "jdbc:mysql://localhost:3306/ABCShop";
    private static final String DB_USER = "root";
    private static final String DB_PASSWORD = "";
    private static Connection con = null;
    //Singleton design pattern
    public static Connection getConnection() throws Exception {
        if(con==null)
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            con = DriverManager.getConnection(JDBC_URL, DB_USER, DB_PASSWORD);
        }
        return con;
    }

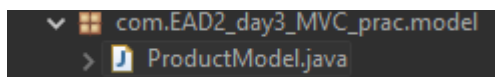
    public void closeConnection()
    {
        try {
            con.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

↓

Create the DB connection only once

→ Function to close the connection

- Now create the model class (.java class file).



- Create the data structure while creating getters, setters and constructors for the class attributes.

```

package com.EAD2_day3_MVC_prac.model;

public class ProductModel {
    private String productId;
    private String productName;
    private String productPrice;
}

public ProductModel(String productId, String productName, String productPrice) {
    super();
    this.productId = productId;
    this.productName = productName;
    this.productPrice = productPrice;
}

public String getProductId() {
    return productId;
}

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

public String getProductName() {
    return productName;
}

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

public String getProductPrice() {
    return productPrice;
}

public void setProductPrice(String productPrice) {
    this.productPrice = productPrice;
}
}

```

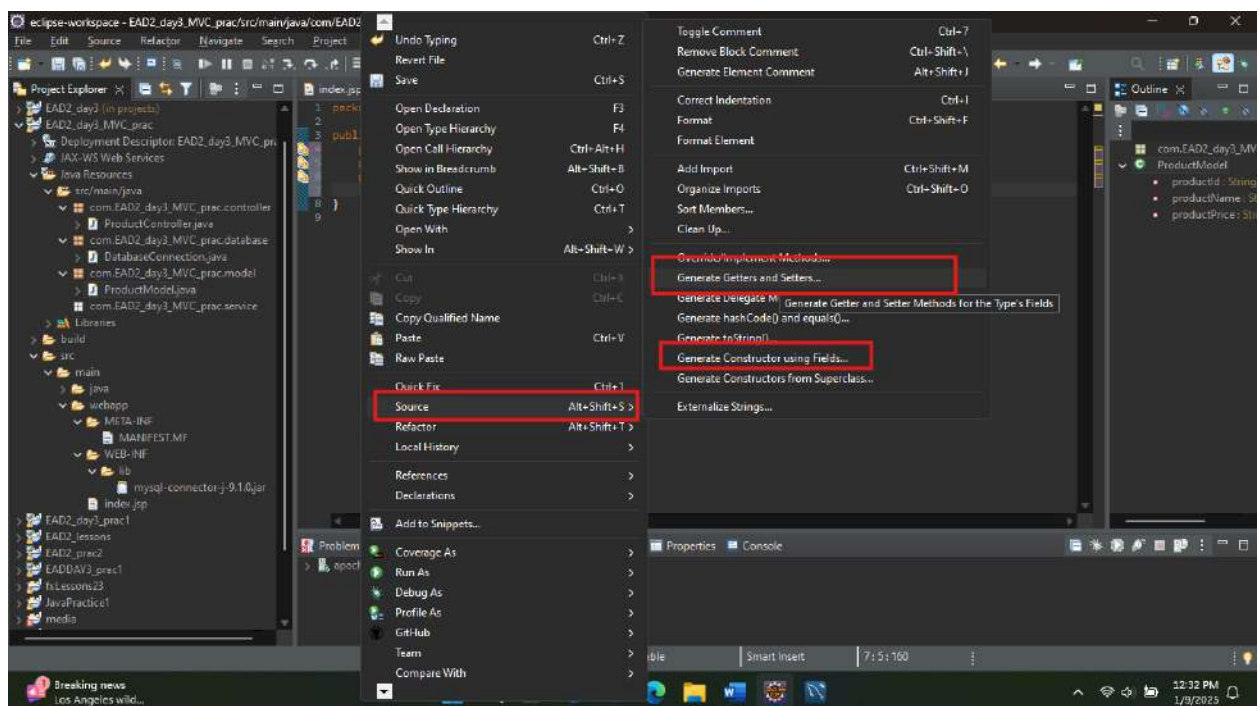
Attributes

Constructor to enter data

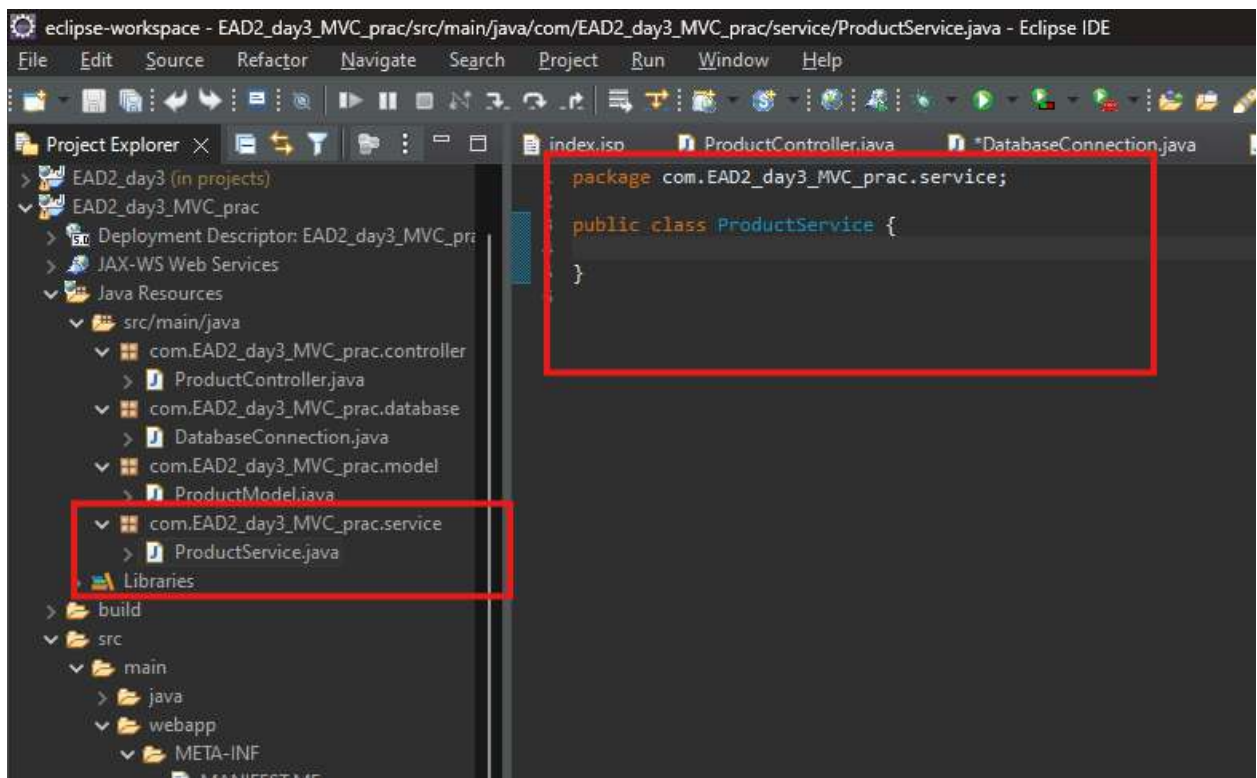
Setter for the attributes

getter for the attributes

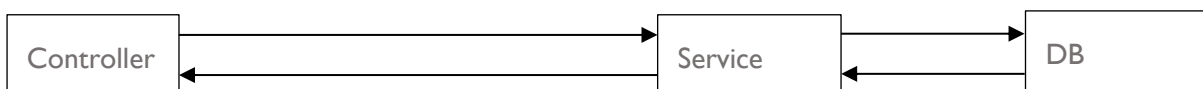
- The above methods can be generated automatically as follows.



- Now create the service class to communicate with the database.



- The data flow and the process will be as follows,



```

package com.EAD2_day3_MVC_prac.service;

import java.sql.Connection;
import java.sql.PreparedStatement;

import com.EAD2_day3_MVC_prac.database.DatabaseConnection;
import com.EAD2_day3_MVC_prac.model.ProductModel;

public class ProductService { //→service class to communicate with the DB
    public boolean saveProduct(ProductModel product) { //→ get the object of the data
        (from view) and returns the status of saved
        boolean isSaved = false;
        try {
            Connection connection = DatabaseConnection.getConnection(); //DB CONN (USING
            SINGALTON)
            String sql = "INSERT INTO product VALUES (?, ?, ?)"; //process with the DB
            PreparedStatement statement = connection.prepareStatement(sql);
            statement.setString(1, product.getProductid());
            statement.setString(2, product.getProductname());
            statement.setDouble(3, Double.parseDouble(product.getProductPrice()));
            int rowsInserted = statement.executeUpdate();
            isSaved = rowsInserted > 0;
            statement.close();
            connection.close();//close the connection
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
  
```

Annotations in the code:

- Annotation: "Returns a Boolean value" points to the `public boolean` return type.
- Annotation: "get the object of the data" points to the `product` parameter.
- Annotation: "Obtaining the connection string (using singleton pattern)" points to the `DatabaseConnection.getConnection()` call.
- Annotation: "Creating and executing the query" points to the SQL statement and its execution.
- Annotation: "Obtaining and setting data using the past object" points to the `statement.setString` and `statement.setDouble` calls.

```

    }
    return isSaved;
}
}

```

- Create the logic of the controller as follows,

```

@WebServlet("/processController")
public class processController extends HttpServlet {
    private static final long serialVersionUID = 1L;
    ProviderService service1; //→ create an object of the service class to work with the
service
    public processController() {
        super();
        service1 = new ProviderService(); //→ complete the object of the service class
    }

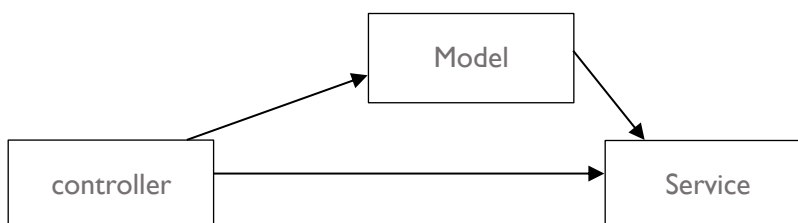
    protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException { //→ service class to act according to the requests from the
frontend
        String id = request.getParameter("id");
        String name = request.getParameter("name");
        String address = request.getParameter("address");
        String mobile = request.getParameter("mobile");
        StudentModel sm1 = new StudentModel(id,name,address,mobile); //→inserting the data to
the model class to access them from the service class while creating the object of the model
class.
        boolean isSaved = service1.saveProduct(sm1); //→ call the service function to send the
data while sending the model object (to access the data using getters and setters). This
function returns a boolean value to tell about whether it is saved or not.

        if (isSaved) {
            request.setAttribute("message", "Product registered successfully!");
        } else {
            request.setAttribute("message", "Error in registration. Please try again.");
        }
        RequestDispatcher dispatcher = request.getRequestDispatcher("index.jsp");
        dispatcher.forward(request, response);
    }
}

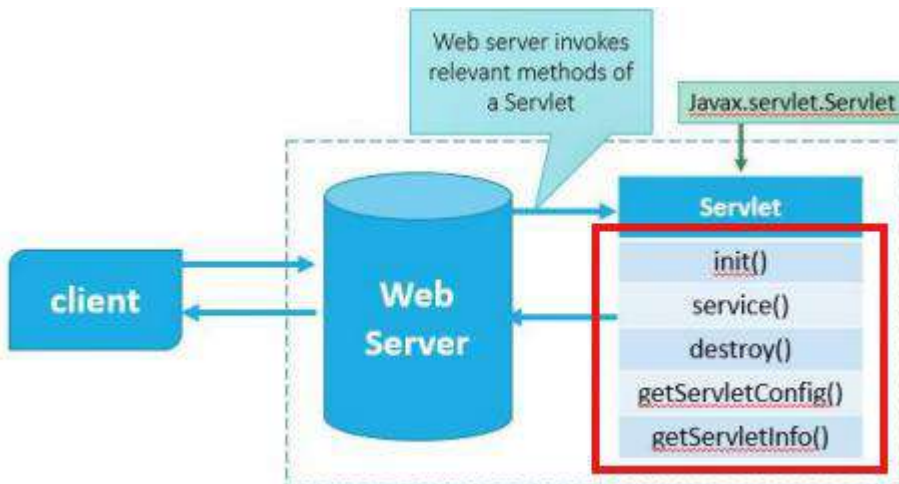
```

} Obtained data

The outputs from the service class will be inserted to the model class.



MORE ON SERVLET LIFECYCLE



- **Servlets run on the web server** platform as part of the same process as the web server itself.
- The **web server is responsible for initializing, invoking, and destroying** each servlet instance.
- The web server **communicates with the servlet using javax.servlet.Servlet interface**.
- This interface consists of three main components →
 - **init()**
 - **service()**
 - **destroy()**
- **init()** → When a servlet is first loaded, its **init()** method is invoked. This allows the servlet to perform any setup processing such as opening files or establishing connections to their servers.
- **service()** → Each request message from a client results in a single call to the servlet's **service()** method (**contains both GET and POST requests**). **This reads the request using the *ServletRequest* object with data from the client and send a reply to the client using *ServletResponse* object.**

in the servlet →

```
protected void service(HttpServletRequest request, HttpServletResponse response throws ServletException, IOException {
```

- **destroy()** → The **destroy()** method is called to allow your servlet to clean up any resources (such as open files or database connections) before the servlet is unloaded. This is optional.

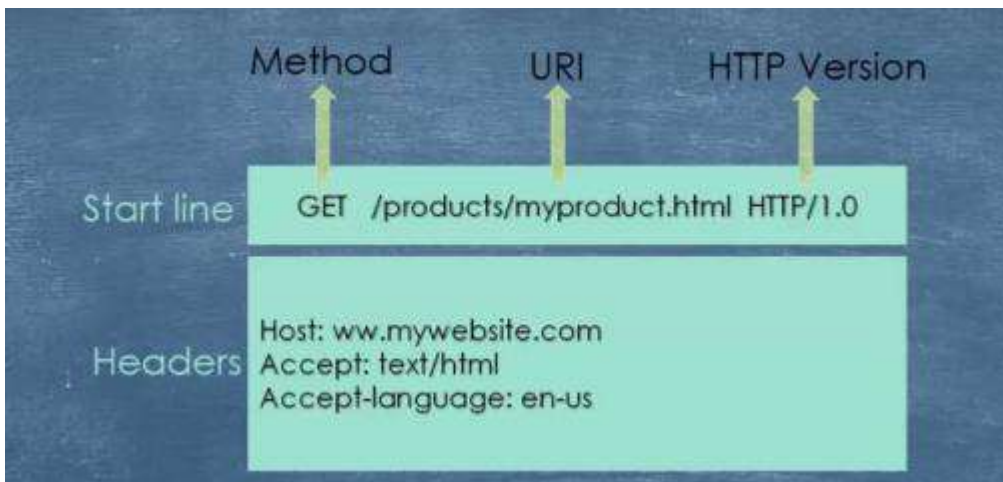
HOW THE CONTAINER HANDLES DYNAMIC REQUESTS

- Use Container (Apache Tomcat is one of the examples).
- sees that the request is for servlet, so create two objects:
 - 1) **HttpServletRequest**
 - 2) **HttpServletResponse** clicks or types a link that has a URL to a servlet instead of a static page.
- The container finds the correct servlet based on the URL in the request with the help deployment descriptor (web.xml) file, creates or allocates a thread for that request, and **passes the request and response objects to the servlet thread**.

- Container calls the servlets service() method, **on the type of request**, service calls doGet() or doPost() methods (example: assume that service calls the doGet() method. doGet() method generates dynamic page and adds the page in response object.).
- container converts the response object into HttpServletResponse object and destroys the response and request object.

HTTP REQUESTS

- The request sent by the computer to a web server that contains all sorts of potentially interesting information is known as HTTP requests.
- Including information →
 - The Request-line.
 - The source IP address and port.
 - The destination IP address, protocol, port and host.
 - The Requested URI (Uniform Resource Identifier).
 - The Http request method and Content.

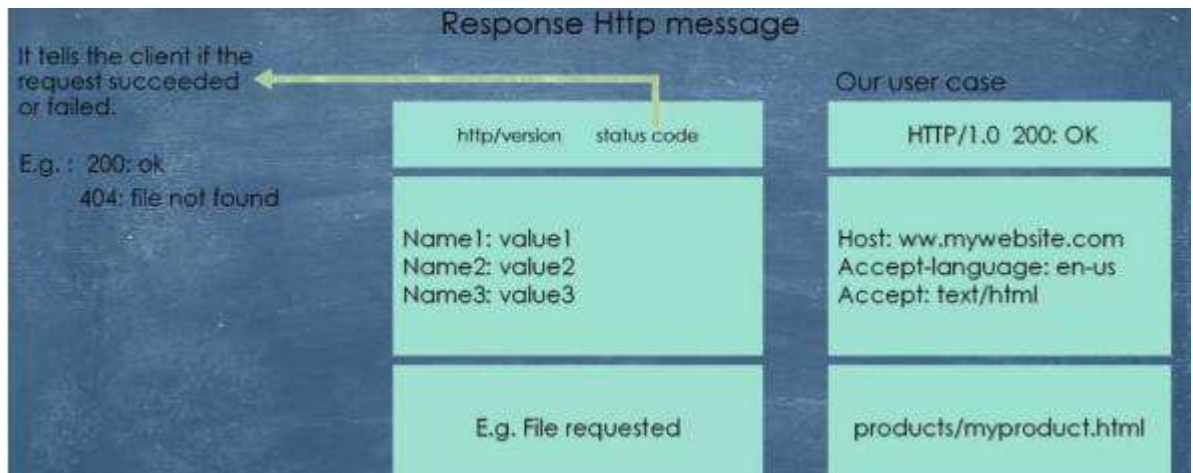


HTTP REQUEST TYPE

HTTP Request	Description
GET	The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.
POST	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
PUT	Replaces all current representations of the target resource with the uploaded content.
DELETE	Says to delete the resource at the requested URL.

HTTP RESPONSE

- Response data of the HTTP.
 - A Status code.
 - Http version.
 - Optionally a message-body.
 - Content type.
 - Actual content



HTTP STATUS CODES

Code number / range	Description
404	Request resource is not found in the server.
200	Request was processed normally.
401	Unauthorized access.
403	Forbidden.
500	Internal server error.
100-199	informational
200-299	Client requests successful.
300-399	Client request redirected. further action necessary.
400-499	Client requests incomplete.
500-599	Server error.

USING HTTP CODES IN JAVA SERVLETS

```
public class HelloWorld extends HttpServlet {  
    @Override  
    protected void doPost (HttpServletRequest req, HttpServletResponse resp) throws ServletException,  
        IOException {  
        resp.sendError (500, "Exception occurred while processing.");  
    }  
}
```

↓ ↓

Status code Error message

WORKING WITH THE GET AND POST METHODS IN SERVLETS

GET/POST METHOD

- First create the frontend of the application as follows,

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>Product registration</h1>
<form action="ProductController" method="get">
<label>Product number</label>
<input type="text" id="product_id" name="product_id" required><br><br>
<label>Name</label>
<input type="text" id="name" name="name" required><br><br>
<label>Price</label>
<input type="text" id="price" name="price" required><br><br>
<button type="submit">Enter</button>
</form>
</body>
</html>
```

Get method or change it to post to work with post

- Now create the servlet for the get method as follows,
- Insert the logic code inside the doGet() method. In post method, insert the code in the doPost() method.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        String proID = request.getParameter("product_id");
        String name = request.getParameter("name");
        double price = Double.parseDouble(request.getParameter("price"));
        PrintWriter out = response.getWriter();
        out.println(proID + "\n" + name);
        ProductModel mo1 = new ProductModel(proID, name, price);
        boolean isSaved = service.sendData(mo1);
        if(isSaved == true) {
            out.println("Data saved");
        }else {
            out.println("Data did not saved");
        }
    }catch(Exception ee) {
        System.out.println(ee.toString());
    }
}
```

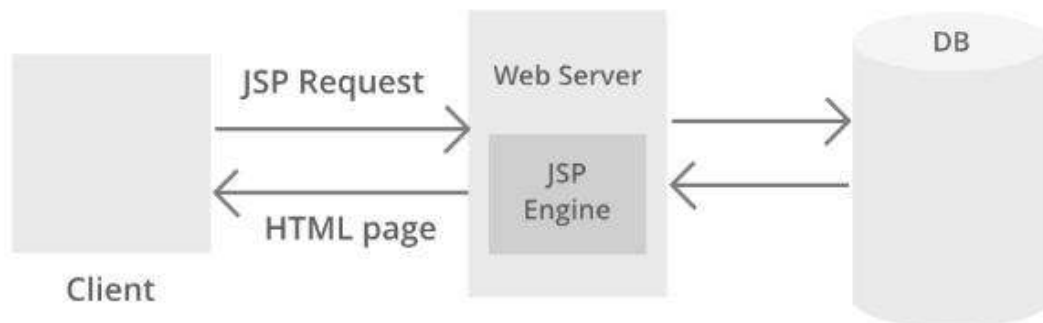
- Now run the application like normal. The only difference is the service() accepts all the HTTP methods. In this case we can use doGet() → GET method and doPost() → POST method.
- To receive the values, from the HTML form use these functions. Use the request function parameter.
 - getParameter() →
 - getParameterValues()
 - getParameterNames()

MORE ABOUT JAVA JSP

ADVANTAGES OF JAVA JSP

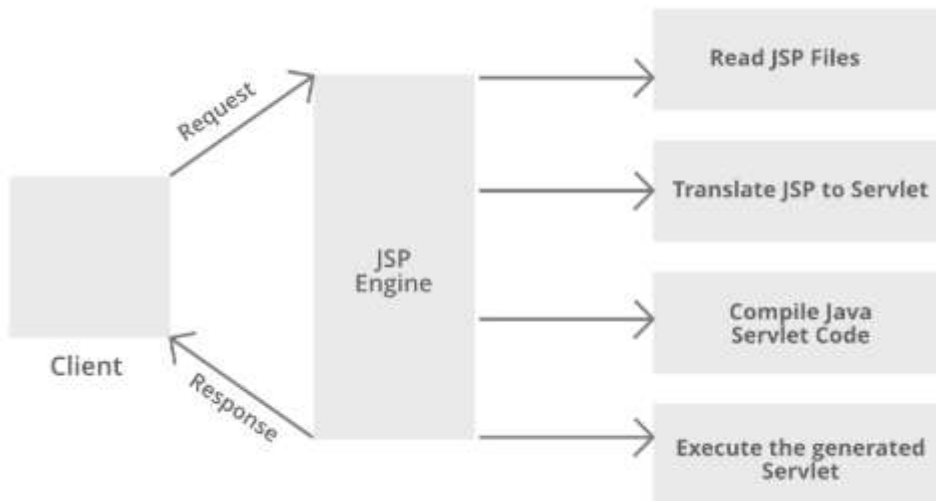
- contains implicit objects.
- Provides implicit/global exception handling mechanism.
- Provides an additional concept called custom tags development.
- Suitable for both java and non-java programmer.
- Increases readability of code because of tags.

JSP ARCHITECTURE



- Also, we consider the JSP page as a servlet.
- In servlets there are 3 constructors, initialize constructors, service constructors and destroy constructors.

JSP architecture is a 3-tier architecture. It has a Client, Web Server, and Database. The client is the web browser or application on the user side. Web Server uses a JSP Engine i.e. a container that processes JSP. For example, Apache Tomcat has a built-in JSP Engine. JSP Engine intercepts the request for JSP and provides the runtime environment for the understanding and processing of JSP files. It reads, parses, builds Java Servlet, Compiles and Executes Java code, and returns the HTML page to the client. The webserver has access to the Database. The following diagram shows the architecture of JSP.

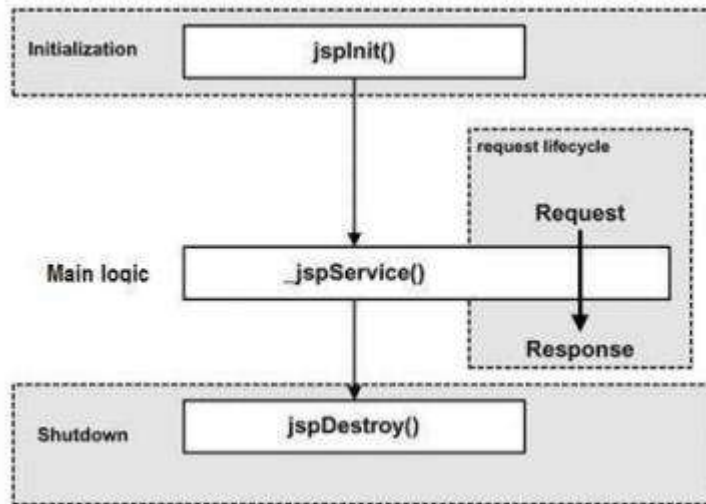


- **Step 1:** The client navigates to a file ending with the .jsp extension and the **browser initiates an HTTP request to the webserver. For example, the user enters the login details and submits the button. The browser requests a status.jsp page from the webserver.**
- **Step 2:** If the compiled version of JSP exists in the web server, it returns the file. Otherwise, the request is forwarded to the JSP Engine. This is done by recognizing the URL ending with **.jsp** extension.
- **Step 3:** The JSP Engine loads the JSP file and translates the JSP to Servlet(Java code). This is done by converting all the **template text into println() statements and JSP elements to Java code**. This process is called **translation**.
- **Step 4:** The JSP engine compiles the Servlet to an executable .class file. It is forwarded to the Servlet engine. This process is called **compilation** or request processing phase.

Then the JSP lifecycle runs ()

- **Step 5:** The **.class** file is executed by the Servlet engine which is a part of the Web Server. The output is an HTML file. The Servlet engine passes the output as an HTTP response to the webserver.
- **Step 6:** The web server forwards the HTML file to the client's browser.

JSP LIFECYCLE



- **jspInit()** → This method is invoked once during the lifecycle of a JSP page. It is called when the JSP page is initialized. This phase is akin to setting up the environment for the JSP page. Developers use this method to perform any resource allocation or setup activities required by the JSP page. For example, establishing database connections or loading configuration settings.
- **_jspService()** → This method is the core of the JSP lifecycle. It handles the processing of incoming requests and generating the appropriate response. Every request to the JSP page triggers a call to this method. The client requests are processed (to another JSP file or servlets).
 - **Request Handling:** When a client (such as a web browser) sends a request to the JSP page, the request is processed by the `_jspService()` method. This involves retrieving request parameters, processing data, and executing business logic.
 - **Response Generation:** After processing the request, the `_jspService()` method generates a response, which is sent back to the client. This response could be in the form of HTML, XML, JSON, or any other format suitable for the client.
- **jspDestroy()** → This method is invoked once when the JSP page is about to be destroyed. It is used to perform any cleanup activities before the JSP page is taken out of service. For instance, releasing resources like database connections or closing open files. This phase ensures that all resources are properly released, and the JSP page is cleaned up efficiently.
- **NOTE** → Once the JSP is compiled, the server doesn't need to retranslate and recompile it for every subsequent request. Instead, it can efficiently use the compiled servlet to handle multiple requests. This saves significant processing time.
- When the server finds the JSP page, it doesn't automatically generate a response. The mere act of finding the page doesn't account for the dynamic nature of web content. Here's why `_jspService()` is essential:
 - **Dynamic Content:** JSP pages often include dynamic content that depends on the request. For example, a JSP might display different content based on query parameters, user input, or session data. `_jspService()` processes this information to produce the correct output.
 - **Business Logic Execution:** The JSP may need to perform operations like querying a database, processing form data, or executing business logic before generating the response. `_jspService()` handles these operations.
 - **Consistency and Modularity:** Using `_jspService()` ensures that every request is processed in a consistent and modular way. It separates the logic of handling requests from the mere presence of the JSP file.
- when consider the requests, form tags and scriptlets tags are also considered as requests (HTTP).

- The `_jspService()` method takes an `HttpServletRequest` and an `HttpServletResponse` as its parameters as follows

```
void _jspService(HttpServletRequest request, HttpServletResponse response)

{ // Service handling code... }
```

JSP SCRIPTING ELEMENTS (THEORY)

- We use these tags to include the java code inside the HTML code.

Declarations (<%! %>)

- Used to define methods and instance variables. These do not produce any output to the client.
- The functions and variables defined are available to the JSP Page as well as to the servlet in which it is compiled

Scriptlets (<% %>)

- Used to embed java code in JSP pages. Contents of JSP go into `_JSPpageservice()` method.

Expressions (<%= %>)

- Used to write dynamic content back to the browser.
- If the output is an object, then the result of calling `toString` on the object is output to the browser

Directives (<%@ %>)

- Directives in JSP (Java Server Pages) are special instructions that are processed by the JSP engine during the translation phase. They provide global information about the entire JSP page and control how the JSP engine processes the page.
- Used for
 - Importing tag libraries (libraries to use within the HTML content)
 - Import required classes
 - Set output buffering options
 - Include content from external files

Page directives (<% page %>): This directive provides information about the JSP page, such as the scripting language used, content type, or buffer size. It can be used to import classes, set error pages.

Example:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%@ page import="java.util.Date" %>
```

Attributes that we can use in here:

- **Language:** (Default Java) Defines server-side scripting language (e.g. java)
- **Extends:** Declares the class which the servlet compiled from JSP needs to extend
- **Import:** Declares the packages and classes that need to be imported for using in the java code (comma separated list)

- Session: (Default true) Boolean which says if the session implicit variable is allowed or not. Sometimes we don't need a session to be created in JSP, and hence, we can set this attribute to false in that case.
- Buffer: defines buffer size of the jsp in kilobytes (if set to none no buffering is done)
- autoFlush: When true the buffer is flushed when max buffer size is reached (if set to false an exception is thrown when buffer exceeds the limit)
- isThreadSafe: (default true) If false the compiled servlet implements SingleThreadModel interface
- Info: String returned by the getServletInfo() of the compiled servlet
- errorPage: Defines the relative URI of web resource to which the response should be forwarded in case of an exception 10. contentType: (Default text/html) Defines MIME type for the output response
- isErrorPage: True for JSP pages that are defined as error pages

Include directive(<%@include%>): This directive is used to include the content of another file during the translation phase. The included file can be a static resource like HTML or another JSP file.

Example

```
<%@ include file="included.jsp" %>
```

Taglib directive(<%@taglib%>): This directive is used to declare a custom tag library that the JSP page will use. It identifies the location of the library and provides a prefix for the custom tags.

Example

```
<% @ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

Method to use

```
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:forEach var="item" items="${itemList}">
  <p>${item.name}</p>
</c:forEach>
```

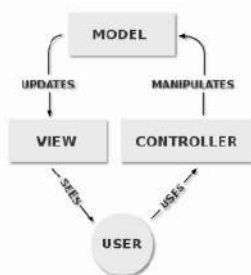
JSP IMPLICIT OBJECTS ALSO CALLED “PRE-DEFINED VARIABLES”

- JSP Implicit Objects are the Java objects that the JSP container makes available to developers on each page and developer can call them directly without being explicitly declared.
- If we consider the JSP as a servlet, we can use the objects that are used in the servlet but without declaring them (they are already declared).

Object	Description
Request	This is the HttpServletRequest object associated with the request.
response	This is the HttpServletResponse object associated with the response to the client.
out	This is the PrintWriter object used to send output to the client.
session	This is the HttpSession object associated with the request.
application	This is the ServletContext object associated with application context.
config	This is the ServletConfig object associated with the page.
pageContext	This encapsulates use of server-specific features like higher performance JspWriters .
page	This is simply a synonym for this , and is used to call the methods defined by the translated servlet class.
Exception	The Exception object allows the exception data to be accessed by designated JSP.

MVC THEORY

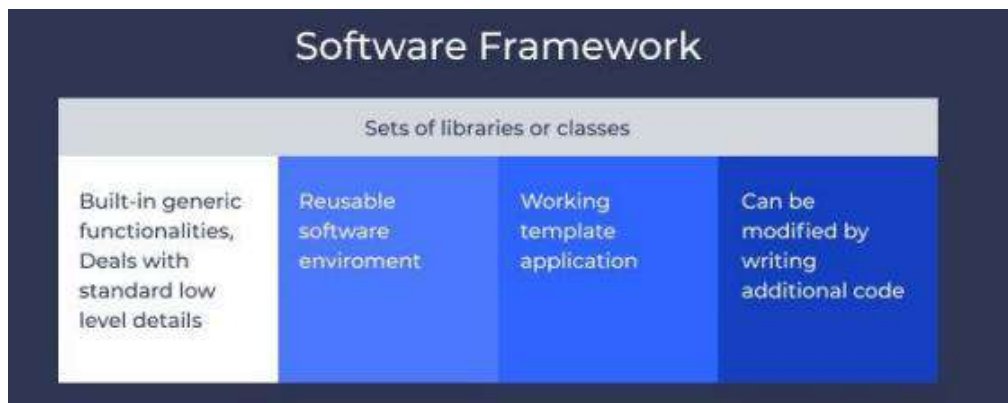
- MVC is a software architecture pattern which allows us to split a software application into three interconnected parts. These parts are Model, View, and Controller.
- Model: Model is an important part of MVC application. It manages the data which is used to represent the output using Views.
- View: Views are the presentation layer of MVC framework. Views are basically the templates where we write the scripts.
- Controller: handle the data coming from the user and responds with a relevant View with a specific Model if required. **Controllers are basically the classes which are responsible for handling the user's request.**



INTRODUCTION TO ENTERPRISE JAVA BEANS (EJB)

FRAMEWORK

- **software framework is a platform for developing software applications. It provides a foundation on which software developers can build programs for a specific platform.**
- These frameworks are built on top of programming languages and tools.



ADVANTAGES OF FRAMEWORKS

- free and open source.
- Contains documentation and community support.
- Eliminate the need to write a lot of repetitive code.
- Higher efficiency.
- Higher security (tested by many developers).

INTRODUCTION TO EJB

- **EJB is a framework for developing large-scale, distributed business applications on the java platform.**
- **An enterprise bean is a server-side component that encapsulates the business logic of an application.**
- The EJB enumeration aims to provide a standard way to implement the server-side business software typically found in enterprise applications.
- **A Web server contains only a web container, but an Application Server contains both a Web Container and an EJB Container.**
- To run an EJB application we need a server with the EJB container (GlassFish,, WebLogic etc..). It performs,
 - Life cycle management.
 - Security.
 - Transaction management.
 - Object pooling.

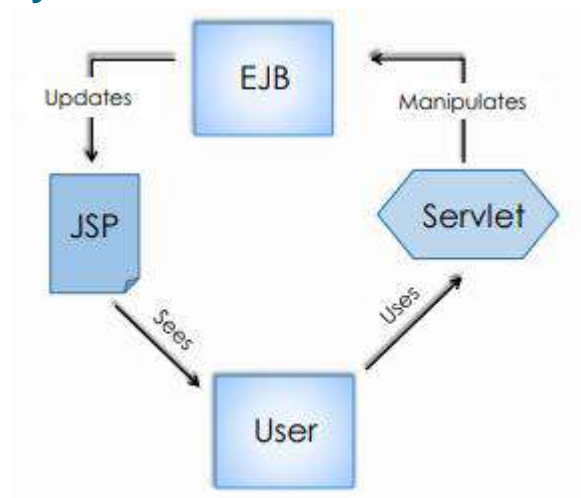
WHEN USE TO ENTERPRISE JAVA BEAN

- When applications need remote access (for distribution).
- When the application needs to be scalable (supports load balancing).
- When Application needs encapsulated business logic (EJB application is separated from presentation and persistent layer).

ADVANTAGES

- EJB are reusable.
- Can be combined visually using a development IDE.
- EJB provides convenient abstractions. So, it does not require you to write:
 - Multi-threaded, multiple access code.
 - Database access code (e.g. JDBC).
 - Network communication code (i.e. it uses RMI) for client/server communication.
 - Network communication code for EJB-to-EJB communication.
- EJBs from different businesses can • Transaction management code interact easily.

EJB AND MVC PATTERN



CONVERSATIONAL STATE

- A **session** is a single client's interaction with a server.
- The **session state** is **client-specific data that is accumulated during the session**.
- **This session state is also known as conversational state.**
- The conversational state can be maintained in the client, the server, or split between them. For example, the contents of the temporary shopping cart can be maintained in the client side or in the server object's instance variables. **The use of session beans to maintain client interactions is the standard method in any J2EE application.**

EJB TYPES

Session beans:

- Session bean encapsulates business logic only, it can be invoked by local, remote and webservice clients. Like calculators etc..
- **Instances of using session beans, when the bean needs to hold information about the client across method invocations and the bean's state represents the interaction between the bean and a specific client.**

- The life cycle of session bean is maintained by the application server (**EJB Container**).
 - **Stateless session beans:** It doesn't maintain state of a client between multiple method calls.
 - **Stateful session beans:** It maintains the state of a client across multiple requests. **conversational** state between multiple method calls is not maintained by the container in case of stateless session bean. The stateless bean objects are pooled by the EJB container to service the request on demand.
 - **Singleton Session Bean:** One instance per application, it is shared between clients and supports concurrent access. This needs when, the application needs an enterprise bean to perform tasks upon application startup and shutdown and when State needs to be shared across the application.

Entity bean:

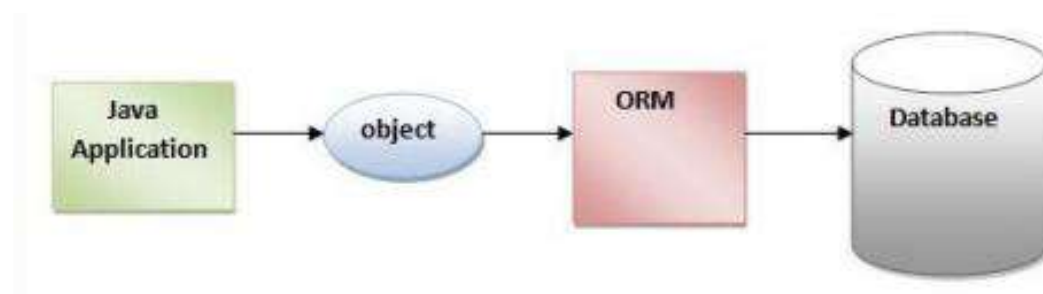
- **Entity bean represents the persistent data stored in the database and can be uniquely identified by a primary key.**
- Entity Beans are a type of Enterprise JavaBean (EJB) that represent persistent data stored in a relational database. They provide an object-oriented interface to this data and map database tables to Java objects. This allows developers to interact with the database using Java code instead of writing SQL queries.
- We use this method to access databases without using queries in java API creation (just create the structure of the database). This is done by entity beans.

Difference between entity and session beans: Entity beans are persistent, allow shared access, have primary keys, and can participate in relationships with other entity beans.

- **The ORM (object relational mapping) programming concept is used to perform the task of the entity bean programmatically.**

ORM (OBJECT RELATIONAL MAPPING)

- **Object-Relational Mapping (ORM)** is a programming technique used to convert data between incompatible type systems in object-oriented programming languages. **In simpler terms, it allows you to map objects in your code to rows in a database table, making it easier to work with databases using object-oriented concepts.**
- An ORM tool simplifies data creation, data manipulation and data access.
- It is a programming technique that maps the object to the data stored in the database.
- The ORM tool internally uses the JDBC API to interact with the database.
- These ORM's are available as frameworks and different frameworks for different languages.
 - **Hibernate** for Java
 - **Entity Framework** for .NET
 - **SQLAlchemy** for Python



ADVANTAGES OF HIBERNATE FRAMEWORK

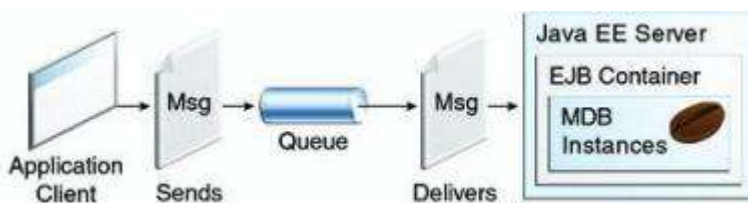
- Open Source and Lightweight.
- Fast Performance.
- Database Independent Query.
- Automatic Table Creation.
- Simplifies Complex Join.

JPA (JAVA PERSISTENCE API)

- Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools.
- The javax.persistence package contains the JPA classes and interfaces.
- Entity Beans have been largely replaced by the Java Persistence API (JPA) in modern enterprise applications. JPA provides a more flexible and powerful framework for managing persistent data, and it is easier to use and configure compared to Entity Beans.
- **This is the platform that provides a framework for managing relational data in Java applications.** JPA is designed to simplify the interaction between Java objects and relational databases, allowing developers to work with database records through Java objects rather than writing complex SQL queries.

Message driven beans:

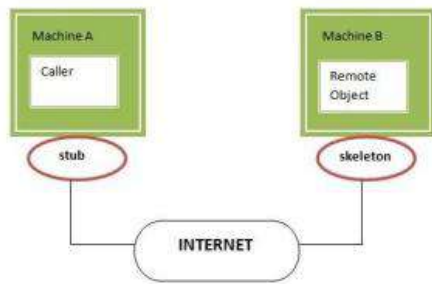
- Message-Driven Beans (MDBs) are a type of Enterprise JavaBean (EJB) that are used to process asynchronous messages.
- A message driven bean (MDB) is a bean that contains business logic. But it is invoked by passing the message. So, it is like JMS Receiver.
- **MDB asynchronously receives the message and processes it.** A message driven bean receives message from queue or topic.



- An application client sends a message (Msg) that needs to be processed asynchronously.
- The message is placed in a queue, which is a storage location for messages waiting to be processed. Queues enable asynchronous communication and ensure that messages are processed in a first-in, first-out (FIFO) order.
- The message is delivered from the queue to the messaging system that will handle its processing.
- The Java EE server contains an EJB (Enterprise JavaBean) container, which is responsible for managing the lifecycle of EJBs, including MDBs. Inside the EJB container, there are instances of Message-Driven Beans (MDBs) that listen to the queue. When a message arrives, an MDB instance is triggered to process the message. MDBs handle the business logic associated with the message and perform the necessary actions.

JAVA RMI (REMOTE METHOD INVOCATION)

- Allows an object running in one Java Virtual Machine (JVM) to invoke methods on an object running in another JVM. This makes it possible to execute methods and exchange data between distributed applications as if they were located on the same machine.
- **It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM (to build distributed applications).**
- RMI uses **stub** and **skeleton** object for communication with the remote object.
 - Stub: acts as the gateway for the client side.
 - Skeleton: acts as the gateway for the server-side object.



- A **remote object** is an object whose method can be invoked from another JVM.

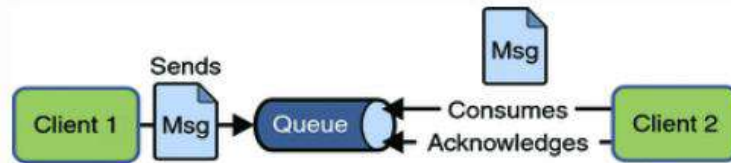
JMS – JAVA MESSAGE SERVICE

INTRODUCTION TO JMS

- JMS (Java Message Service) is an API that provides the facility to create, send and read messages. It provides loosely coupled, reliable and asynchronous communication.
- if we want to send message from one application to another, we need to use JMS API.
- EJB and JMS
 - **JMS** can be used to send messages from the order processing system to different systems handling inventory, payment, and email notifications.
 - **EJBs**, particularly **MDBs**, can be used to listen for these messages and process them asynchronously, ensuring that the order processing flow is efficient and robust.

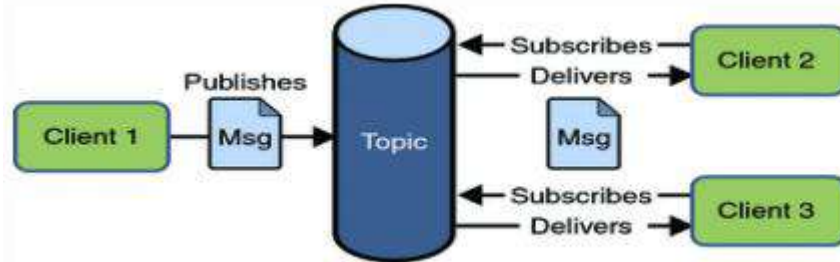
MESSAGING DOMAINS

- Two types of messaging domains,
 - Point to point message domain.
 - one message is delivered to one receiver only.
 - The Queue is responsible for holding the message until the receiver is ready.
 - Queue is used as a message-oriented middleware.



- Publisher/Subscriber Messaging Domain.

- In Pub/Sub model, one message is delivered to all the subscribers.
- Topic is used as a message-oriented middleware that is responsible to hold and deliver messages.



- A **Topic** acts as a message broadcast channel that allows multiple subscribers to receive the same message.
- **Topic** is an interface, not a class or a package. It belongs to the javax.jms package in the JMS API.

- Differences comparison,

Point to point message domain	Publisher/Subscriber Messaging Domain
delivered to one and only one JMS Receiver	Each message is delivered to multiple Consumers.
no timing dependency.	Pub/Sub model has some timing dependency.
sends acknowledgements to JMS Sender once it receives messages.	Acknowledgement is not required.

INTRODUCTION TO REST API

SERVICE ORIENTED ARCHITECTURE (SOA)

- A **Service-Oriented Architecture** or **SOA** is a design pattern which is designed to build distributed systems that deliver services to other applications through the protocol.
- **Service:** A service is a well-defined, self-contained function that represents unit of functionality. A service can exchange information from another service. It is not dependent on the state of another service.
- **Service connections:** Service consumer sends a service request to the service provider, and the service provider sends the service response to the service consumer. The service connection is understandable to both the service consumer and service provider.



WEB SERVICES

- The method of communication between two devices over the network or it is a collection of standards or protocols for exchanging information between two devices or application.

NECESSITIES OF WEB SERVICES

- Exposing the Existing Function on the network: other applications can use the functionality of your program.
- Interoperability: allow various applications to talk to each other and share data and services among themselves
- Standardized protocol: use standardized industry standard protocol for communication.
- Low-cost communication: due to SOAP over HTTP protocol.

TYPES OF WEB SERVICES

- SOAP web services.
- Restful web services.

SOAP – SIMPLE OBJECT ACCESS PROTOCOL

- SOAP is an XML-based protocol for accessing web services.
- SOAP is a W3C recommendation for communication between applications.
- SOAP is XML based, so it is platform independent and language independent. In other words, it can be used with Java, .Net or PHP language on any platform.

Advantages	Disadvantages
WS security: SOAP defines its own security known as WS Security.	Slow: SOAP uses XML format that must be parsed to be read. It defines many standards that must be followed and consumes more bandwidth.
Language & platform independent: SOAP web services can be written in any programming language and executed in any platform.	WSDL dependent: SOAP uses WSDL and doesn't have any other mechanism to discover the service.

WSDL

- **WSDL is a xml document containing information about web services such as method name, method parameter and how to access it.**
- **SOAP:** A protocol for exchanging information in the form of XML messages.

WSDL: An XML-based language for describing the functionalities offered by a web service, typically using SOAP as the protocol.

- WSDL is a part of UDDI(Universal Description, Discovery, and Integration). It acts as an interface between web service applications.

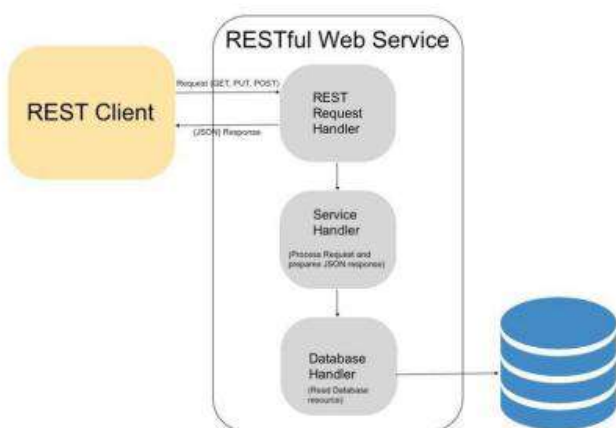
RESTFUL WEB SERVICES

- REST is an architectural style not a protocol.
- **A service which is built on the REST architecture is called a RESTful web service.**
- **The underlying protocol for REST is HTTP.**

MAIN FEATURES

- **Fast:** RESTful Web Services are fast because there is no strict specification like SOAP.
- **Language and Platform independent:** RESTful web services can be written in any programming language and executed in any platform.
- **Can use SOAP:** RESTful web services can use SOAP web services as the implementation.
- **Permits different data format:** RESTful web service permits different data format such as Plain Text, HTML, XML and JSON.

RESTFUL ARCHITECTURE



that every resource should be accessible via the normal HTTP commands of GET, POST, PUT, or DELETE. So, if someone wanted to get a file from a server, they should be able to issue the GET request and get the file. If they want to put a file on the server, they should be able to either issue the POST or PUT request. And finally, if they wanted to delete a file from the server, they issue the DELETE request.

Client: web browser.

Server: web server hosting the application.

Stateless: the state of the application is not maintained in REST. For example, if you delete a resource from a server using the DELETE command, you cannot expect that deleted information to be passed to the next request.

ELEMENTS OF REST

- Resources.
- Request verbs.
- Request headers.
- Request body.
- Response body.
- Response status codes.

HTTP METHODS

- GET: Provides read only access to a resource.
- POST: Used to create a new resource.
- DELETE: Used to remove a resource.
- PUT: Used to update an existing resource.
- PATCH: The PATCH method is used to apply partial modifications to a resource. Unlike the PUT method, which replaces the entire resource, PATCH updates only the specified fields.
- HEAD: The HEAD method is used to retrieve the headers of a resource without fetching its body. This is useful for obtaining metadata about the resource, such as its size, content type, and last modified date. For example, if you want to check whether an image exists and get its metadata without downloading the image itself, you can use a HEAD request.
- OPTIONS: The OPTIONS method is used to describe the communication options available for a resource or the server. It returns the allowed HTTP methods for the resource.

BEST PRACTICES WHEN BUILDING REST API

- Use PUT, POST and DELETE methods instead of the GET method to alter the state.
- Do not mix up singular and plural nouns. Keep it simple and use only plural nouns for all resources.
- If a resource is related to another resource use sub resources.
- Use HTTP headers for serialization formats.
- Handle Errors with HTTP status codes.

HTTP STATUS CODES

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

API RESPONSE CODES

HTTP Status Code	Description
200 OK	Successful.
201 Created	Created.
400 Bad Request	Bad input parameter. Error message should indicate which one and why.
401 Unauthorized	The client passed in the invalid Auth token. Client should refresh the token and then try again.
403 Forbidden	* Customer doesn't exist. * Application not registered. * Application try to access to properties not belong to an App. * Application try to trash/purge root node. * Application try to update contentProperties. * Operation is blocked (for third-party apps). * Customer account over quota.
404 Not Found	Resource not found.
405 Method Not Allowed	The resource doesn't support the specified HTTP verb.
409 Conflict	Conflict.
411 Length Required	The Content-Length header was not specified.
412 Precondition Failed	Precondition failed.
429 Too Many Requests	Too many request for rate limiting.
500 Internal Server Error	Servers are not working as expected. The request is probably valid but needs to be requested again later.
503 Service Unavailable	Service Unavailable.

DIFFERENCE BETWEEN SOAP AND REST

SOAP	REST
SOAP is a protocol .	REST is an architectural style .
SOAP stands for Simple Object Access Protocol .	REST stands for REpresentational State Transfer .
SOAP can't use REST because it is a protocol.	REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
SOAP uses services interfaces to expose the business logic .	REST uses URI to expose business logic .
JAX-WS is the Java API for SOAP web services.	JAX-RS is the Java API for RESTful web services.
SOAP defines standards to be strictly followed.	REST does not define too much standards like SOAP.
SOAP requires more bandwidth and resource than REST.	REST requires less bandwidth and resource than SOAP.
SOAP defines its own security .	RESTful web services inherits security measures from the underlying transport.
SOAP permits XML data format only.	REST permits different data format such as Plain text, HTML, XML, JSON etc.
SOAP is less preferred than REST.	REST more preferred than SOAP.

BUILDING REST API WITH JAVA

INTRODUCTION TO SPRING FRAMEWORK

- Spring is a lightweight and popular open-source Java-based framework.
- It is used to develop enterprise-level applications. It provides support to many other frameworks such as Hibernate, Tapestry, EJB, JSF, Struts, etc, so it is also called a framework of frameworks.
- It's an application framework and IOC (Inversion of Control) container for the Java platform. Spring contains several modules like IOC, AOP, DAO, Context, WEB MVC, etc
- Spring Boot is an open-source Java-based framework used to create a micro-Service. It is developed by Pivotal Team and is used to build stand-alone and production ready spring applications.

Advantages

Spring Boot offers the following advantages to its developers –

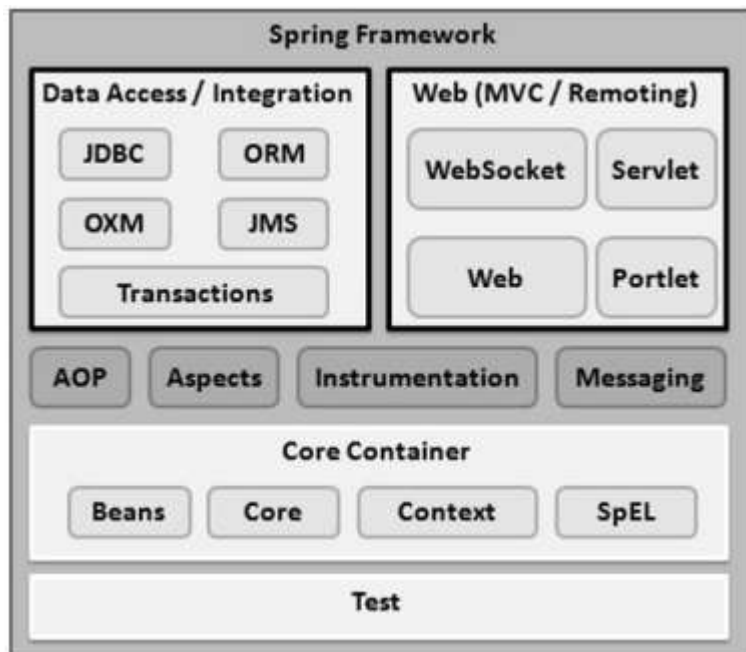
- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

Goals

Spring Boot is designed with the following goals –

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way
- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application

SPRING FRAMEWORK ARCHITECTURE



- **Core container:**
 - **Beans:** The fundamental unit in Spring, representing objects managed by the Spring IoC (Inversion of Control) container.
 - **Core:** Provides core features like dependency injection (DI) and IoC.
 - **Context:** Offers a way to access application objects and integrates with Java EE features.
 - **SpEL (Spring Expression Language):** A powerful expression language used to query and manipulate object graphs.
- **Data Access / Integration:**
 - **JDBC:** Simplifies database access by reducing boilerplate code.
 - **ORM:** Integrates with ORM frameworks like Hibernate, JPA, and MyBatis.
 - **OXM (Object-XML Mapping):** Supports working with XML using technologies like JAXB.
 - **JMS (Java Messaging Service):** Facilitates message-based communication.
 - **Transactions:** Provides transaction management, supporting both programmatic and declarative transactions.

TRANSACTIONS IN SPRING BOOT

A transaction is a sequence of operations performed as a single unit of work. In Spring Framework, transaction management ensures that database operations maintain **data integrity, consistency, and reliability**. If any part of the transaction fails, the entire transaction is rolled back.

Another explanation:

A transaction is a sequence of actions performed by the application that together pipelined to perform a single operation. For example, booking a flight ticket is also a transaction where the end user has to enter his information and then make a payment to book the ticket.

Main reason for requiring this concept:

Let's understand transactions with the above example, if a user has entered his information the user's information gets stored in the user_info table. Now, to book a ticket he makes an online payment and due to some reason(system failure) the payment has been canceled so, the ticket is not booked for him. But, the

problem is that his information gets stored on the user_info table. On a large scale, more than thousands of these things happen within a single day. So, it is not good practice to store a single action of the transaction(Here, only user info is stored not the payment info).

To overcome these problems, spring provides transaction management, which uses annotation to handle these issues. In such a scenario, spring stores the user information in temporary memory and then checks for payment information if the payment is successful then it will complete the transaction otherwise it will roll back the transaction and the user information will not get stored in the database.

Spring provides two types of transaction management:

1. Declarative Transaction Management (Recommended)

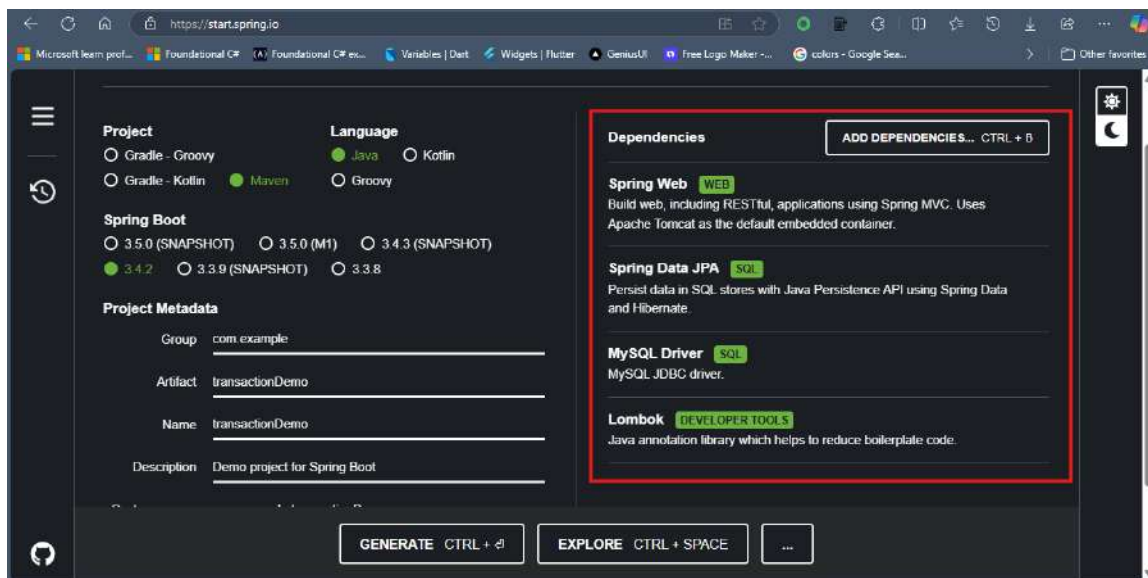
- Uses **annotations or XML configuration** to manage transactions.
- No need to write manual transaction handling code.
- Uses **@Transactional** annotation to define transactional boundaries.

2. Programmatic Transaction Management

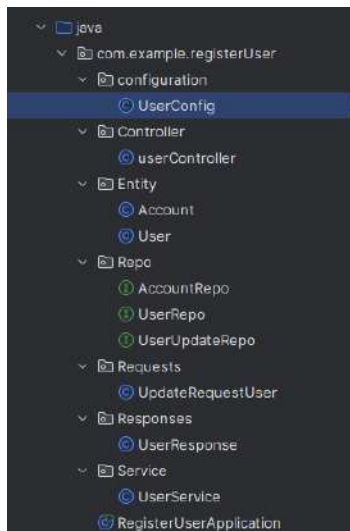
- Transactions are managed manually in the code using the **TransactionTemplate** or **PlatformTransactionManager**.
- Provides fine-grained control but leads to **boilerplate code**.

Declarative transaction management

- First create a spring boot project with the following dependencies.



- Create this file structure:



- Insert this annotation and throw exceptions to roll back. Use `@Transactional(rollbackFor = Exception.class)` to roll back in check exceptions.

```
package com.example.registerUser.Service;

import com.example.registerUser.Entity.Account;
import com.example.registerUser.Entity.User;
import com.example.registerUser.Repo.AccountRepo;
import com.example.registerUser.Repo.UserRepo;
import com.example.registerUser.Repo.UserUpdateRepo;
import com.example.registerUser.Requests.UpdateRequestUser;
import com.example.registerUser.Responses.UserResponse;
import jakarta.transaction.Transactional;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Example;
import org.springframework.data.domain.ExampleMatcher;

import java.util.List;
import java.util.Optional;

public class UserService {
    @Autowired
    private UserRepo userRepo;
    @Autowired
    private AccountRepo accountRepo;
    @Autowired
    private ModelMapper mapper;
    @Autowired
    private UserUpdateRepo userUpdateRepo;
    @Transactional ←
    public UserResponse insertData(User user) {
        User user2 = mapper.map(user, User.class);
        User savedUser = userRepo.save(user2);
        Account account = new
Account(savedUser.getPassword(), savedUser.getUser_Name());
        Account account2 = mapper.map(account, Account.class);
        Account savedAccount = accountRepo.save(account2);
        UserResponse userResponse = mapper.map(savedUser, UserResponse.class);
        return userResponse;
    }
}
```

- Enable the transaction in the application.

```

package com.example.registerUser;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootApplication
@EnableTransactionManagement
public class RegisterUserApplication {
    public static void main(String[] args) {
        SpringApplication.run(RegisterUserApplication.class, args);
    }
}

```

Web (MVC / Remoting)

- **WebSocket:** Supports real-time communication.
- **Servlet:** Provides integration with Java servlets.
- **Web:** Supports core web-related functionalities like multipart file uploads and initializing web applications.
- **Portlet:** Enables the development of Java portlets (for portal frameworks like Liferay).
 - Java Portlets are server-side components that process requests and generate dynamic content, typically for web applications. They are part of the Java Portlet Specification, which is standard for developing web portal applications.
 - **Modular Web Components:** Portlets are designed to be modular components of a web page. They can be added, removed, and rearranged without affecting other parts of the page.
 - **Request Handling:** Similar to servlets, portlets handle requests and generate responses. However, portlets are specifically designed to generate content fragments, not entire web pages.
 - **Portlet Container:** Portlets run within a portlet container, which provides the runtime environment and manages the lifecycle of portlets.
- **Portlet Lifecycle:**
 - Initialization (init): The portlet is initialized when it is first created.
 - Request Handling (processAction and render): The portlet processes different types of requests, including action requests (e.g., form submissions) and render requests (e.g., generating content for display).
 - Destruction (destroy): The portlet is destroyed when it is no longer needed.

- Example:

```

import javax.portlet.Portlet;
import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import java.io.IOException;

public class MyPortlet implements Portlet {

```

```

@Override

public void init() throws PortletException {

    // Initialization code here

}

@Override

public void processAction(ActionRequest request, ActionResponse response) throws
PortletException, IOException {

    // Action processing code here

}

@Override

public void render(RenderRequest request, RenderResponse response) throws PortletException,
IOException {

    response.getWriter().print("Hello from MyPortlet!");

}

@Override

public void destroy() {

    // Cleanup code here

}

}

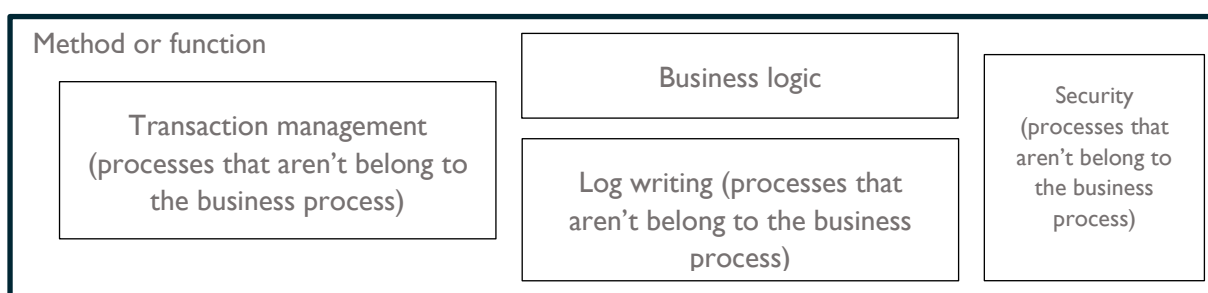
```

Other Modules

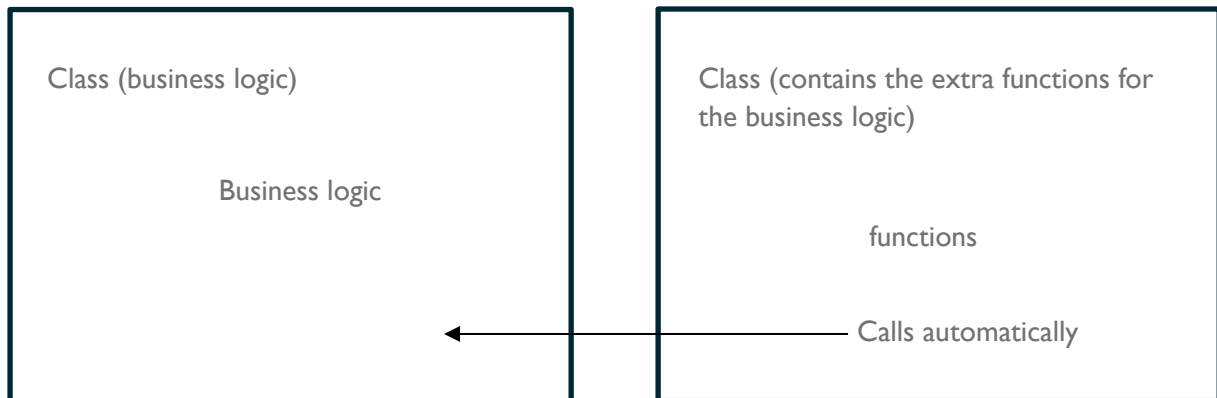
- **AOP (Aspect-Oriented Programming):** Enables aspect-oriented programming, allowing separation of cross-cutting concerns (like logging, security).
- **Aspects:** Works with AOP to define aspects (modular concerns).
- **Instrumentation:** Provides class instrumentation and class loader support.
- **Messaging:** Supports messaging frameworks like RabbitMQ.

ASPECT ORIENTED PROGRAMMING (AOP)

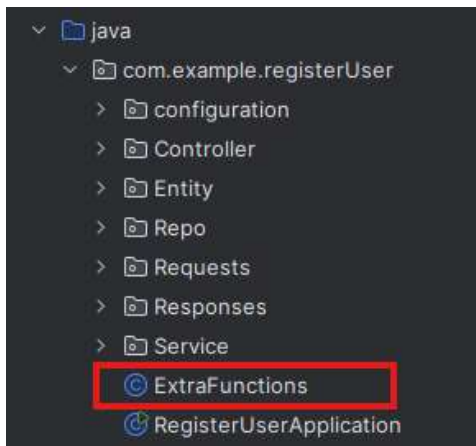
Aspect-Oriented Programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. Cross-cutting concerns are aspects of a program that affect multiple parts of the application, such as logging, security, or transaction management. These concerns can lead to code duplication and tangled code if not handled properly.



- In here we are separating these processes that do not belong to the business logic and calls those functions automatically.



- Main keywords:
 - Cross cutting concerns: extra functions that are not a part of the main business logic.
 - Aspects: section that the cross-cutting concerns are located. This may be a class.
 - Join point: A **join point** is a specific point in the application where we can plug-in the AOP **aspect**. Such as method execution, exception handling, variable modification... Most of the time, a **join point** represents a method execution.
 - Advise: An **advice** is an action taken at a particular **join point**. In other words, it is the actual code that gets executed before or after the **join point** or what we should call in the business logic section. **This advice is the method which contains the extra code in the aspect.** There are different types of advice.
 - **Before** : Executed before the method call.
 - **After** : Executed after the method call, regardless of its outcome.
 - **AfterReturning** : Executed after the method returns a result, but not if an exception occurs.
 - **Around** : Surrounds the method execution, allowing you to control the method execution and its result.
 - **AfterThrowing** : Executed if the method throws an exception.
 - Point cut: this is the section or the place where we call this extra function (advice).
 - Weaving : The process of linking aspects with the target object. Weaving can occur at compile-time, load-time, or runtime. Spring AOP performs runtime weaving using proxy-based mechanisms.
- Example for AOP in Spring:
 - Create a class for the advice.



- Define the advice and the logic as below:

```
package com.example.registerUser;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class ExtraFunctions {

    @Before("execution(public com.example.registerUser.Responses.UserResponse
com.example.registerUser.Service.UserService.getDataByID(..))")
    /*
    execution(...) → Matches method execution.
    public com.example.registerUser.Responses.UserResponse → Fully qualified return type.
    com.example.registerUser.Service.UserService.getDataByID(..) → Fully qualified class
    and method name.
    (..) → Matches any arguments.

    If you don't care about the return type, you can use *:
    //@Before("execution(* com.example.registerUser.Service.UserService.getDataByID(..))")

    * In Spring AOP, the execution expression matches methods based on their signature,
    which includes:
    ✓ Access modifier (public, private, etc.)
    ✓ Return type
    ✓ Method name
    ✓ Parameters

    If the return type is not correctly specified, the AOP pointcut won't match the method,
    and the advice will not be executed.
    * */
    public void log() {
        System.out.println("LOG WRITING");
    }
}
```

- Output →

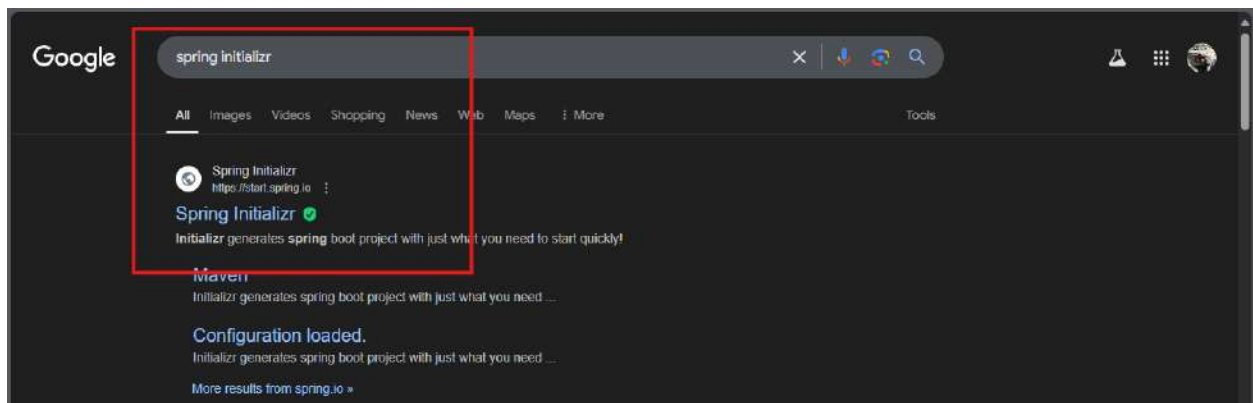
```
LOG WRITING
Hibernate:
select
  u1_0.userid,
  u1_0.address,
  u1_0.email,
  u1_0.password,
  u1_0.first_name,
  u1_0.last_name,
  u1_0.mobile_number,
  u1_0.nic_number,
  u1_0.user_name
from
  users u1_0
where
  u1_0.userid=?
```

Test Module

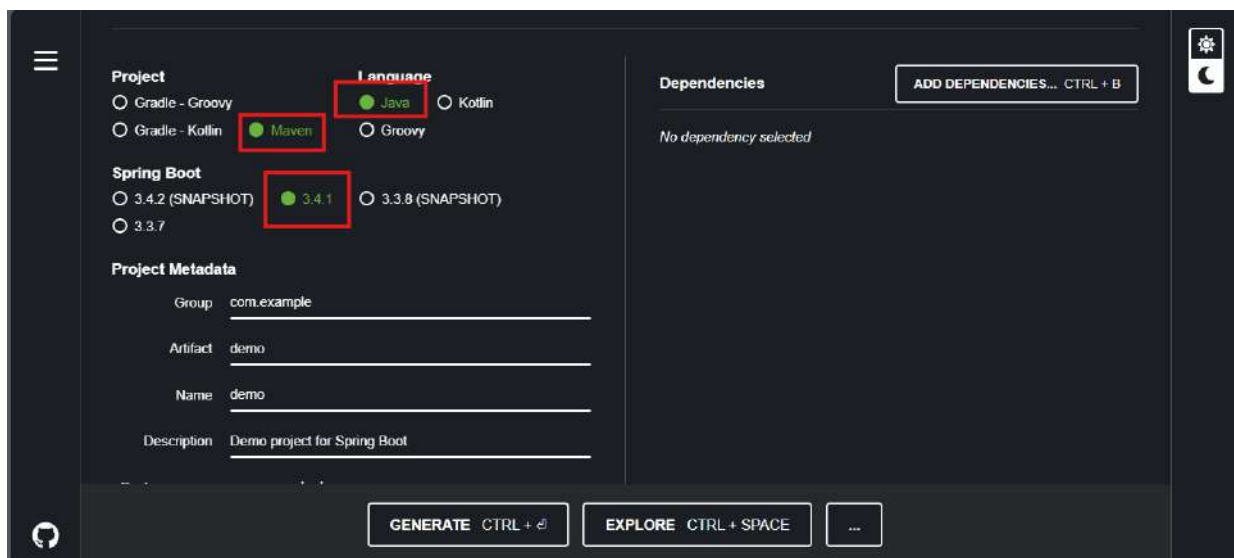
- Provides testing support for unit testing and integration testing with JUnit, TestNG, and Mock frameworks.

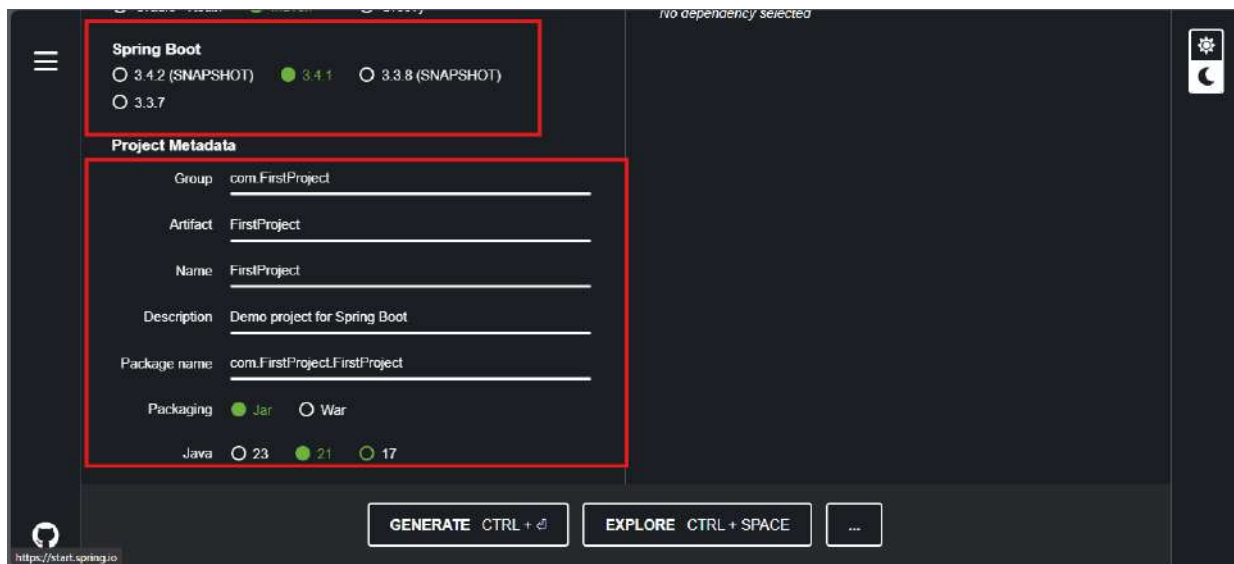
CREATING REST API USING SPRING

- First go to the spring initializr (<https://start.spring.io>)

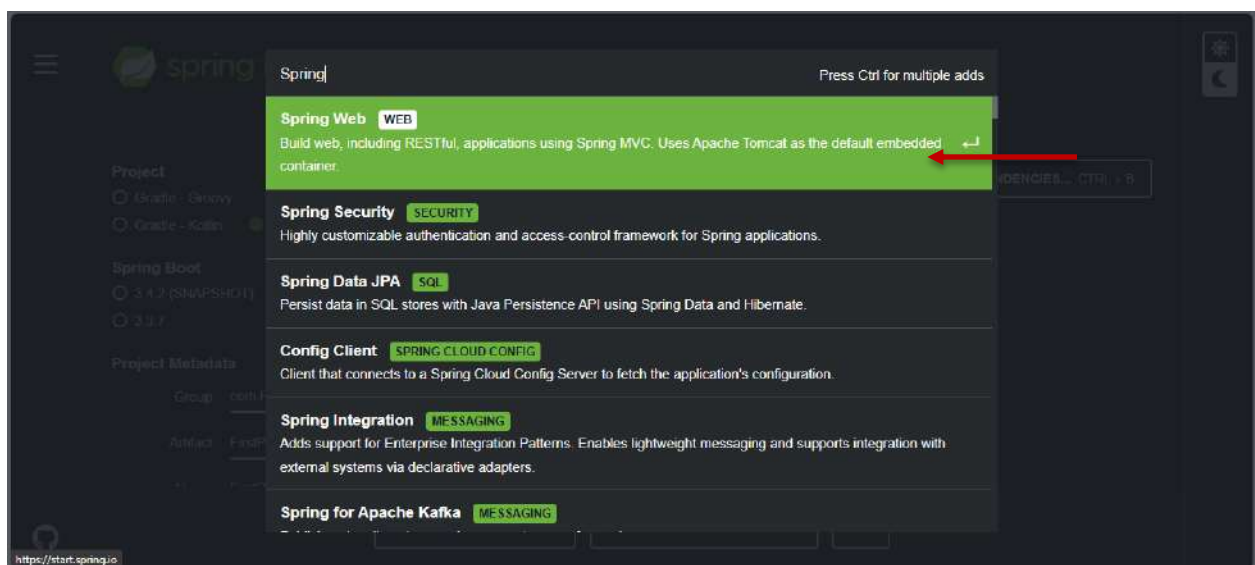
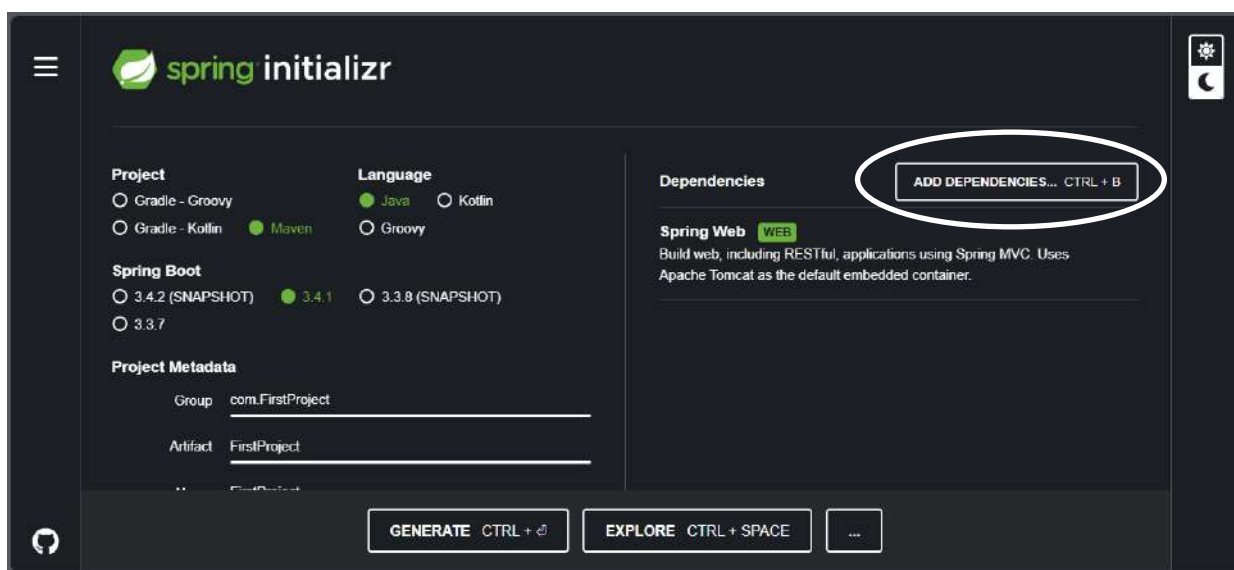


- Create the project with the following configurations.

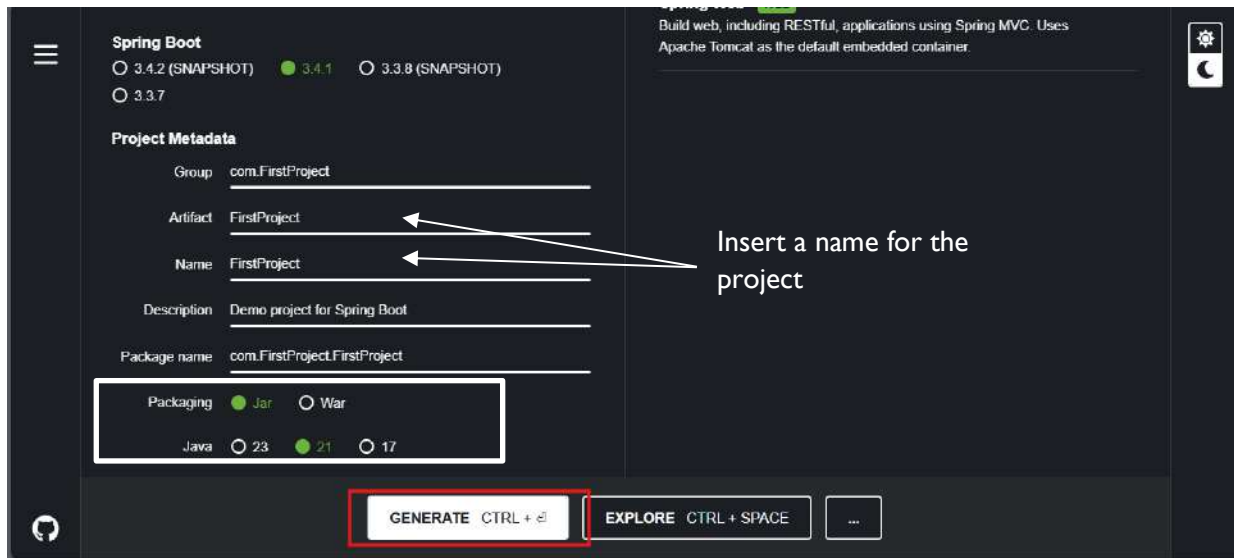




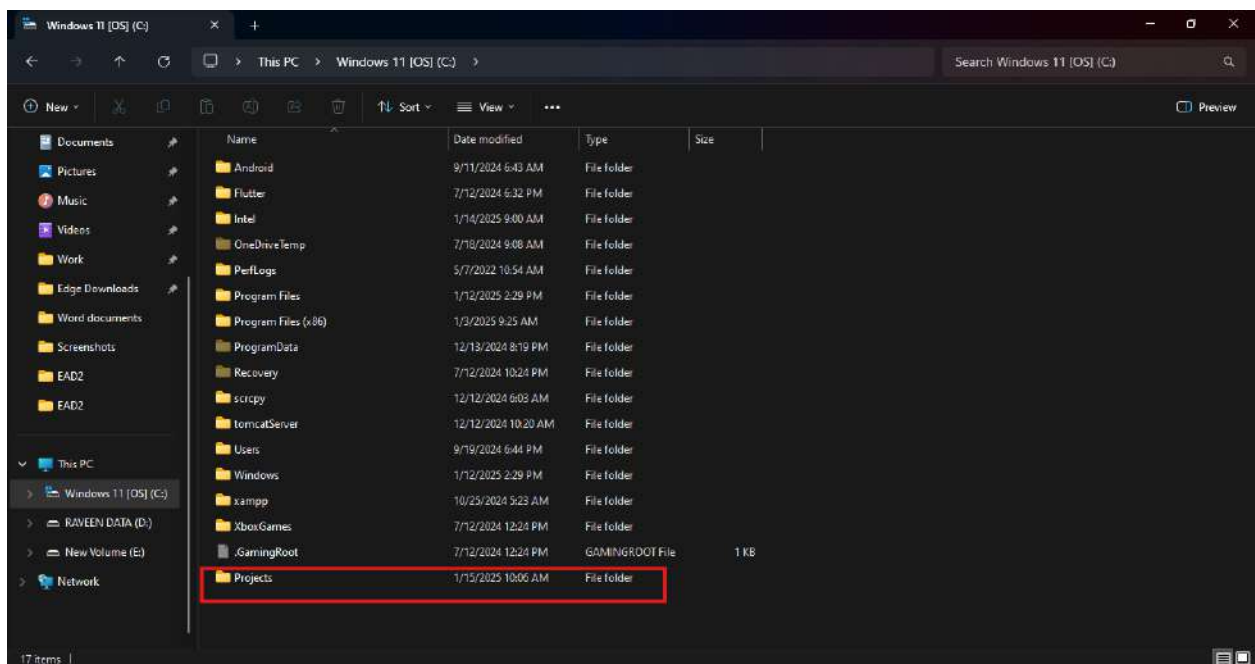
- Now add the dependencies as follows:



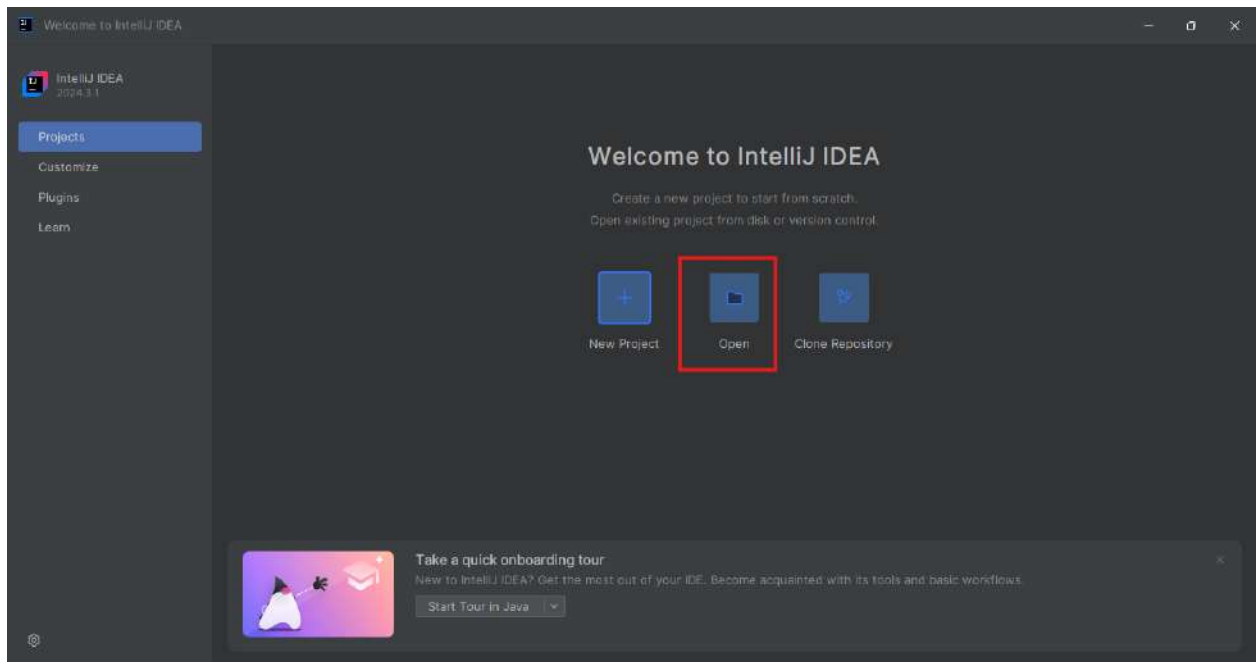
- Then add the following information:



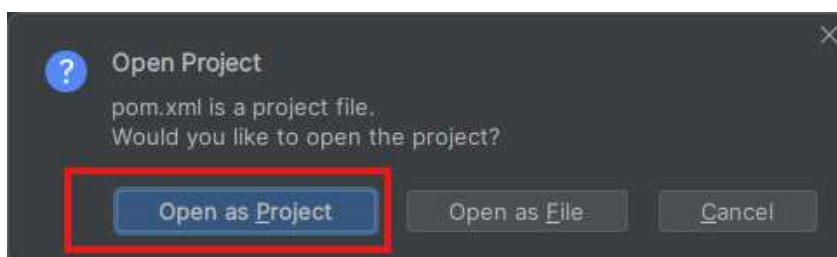
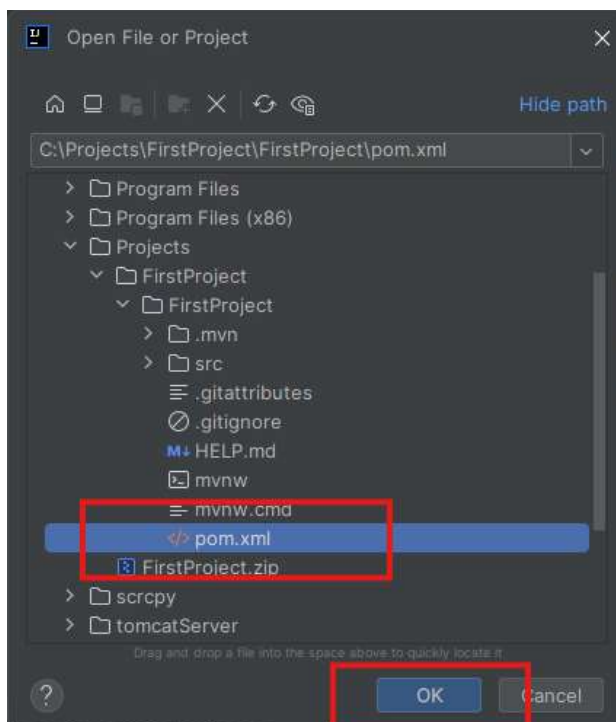
- Create a project in the C drive for the spring projects.



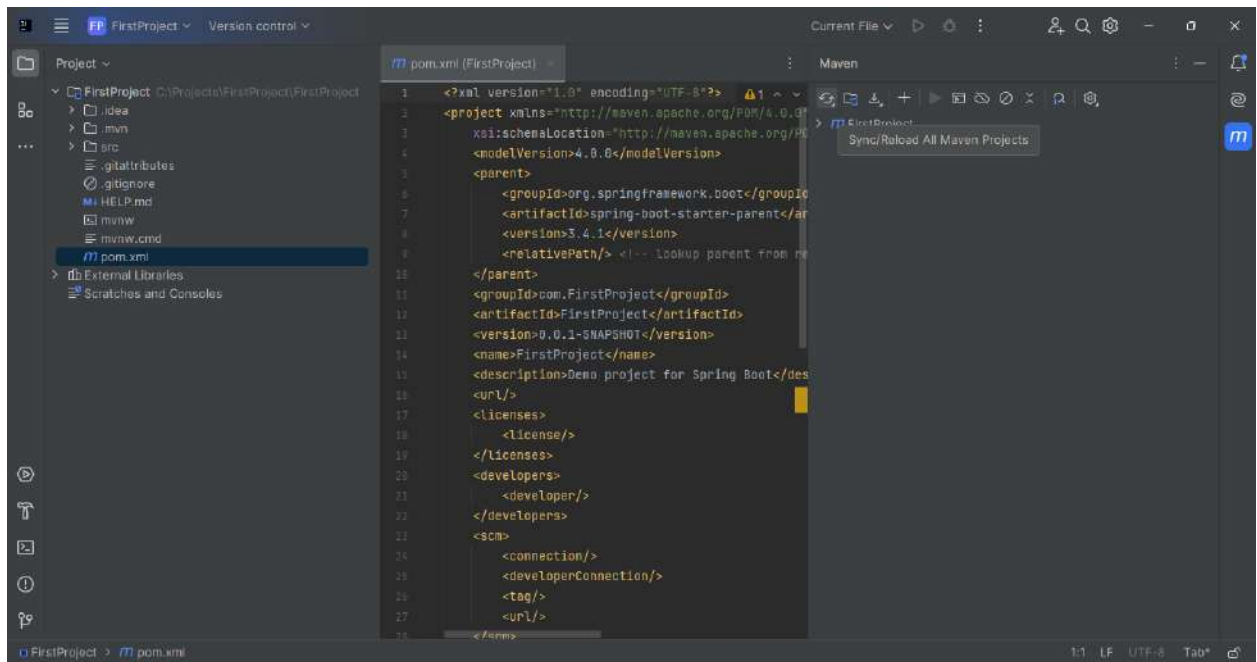
- Then add the downloaded project and unzip it.
- Open IntelliJ and open the project:



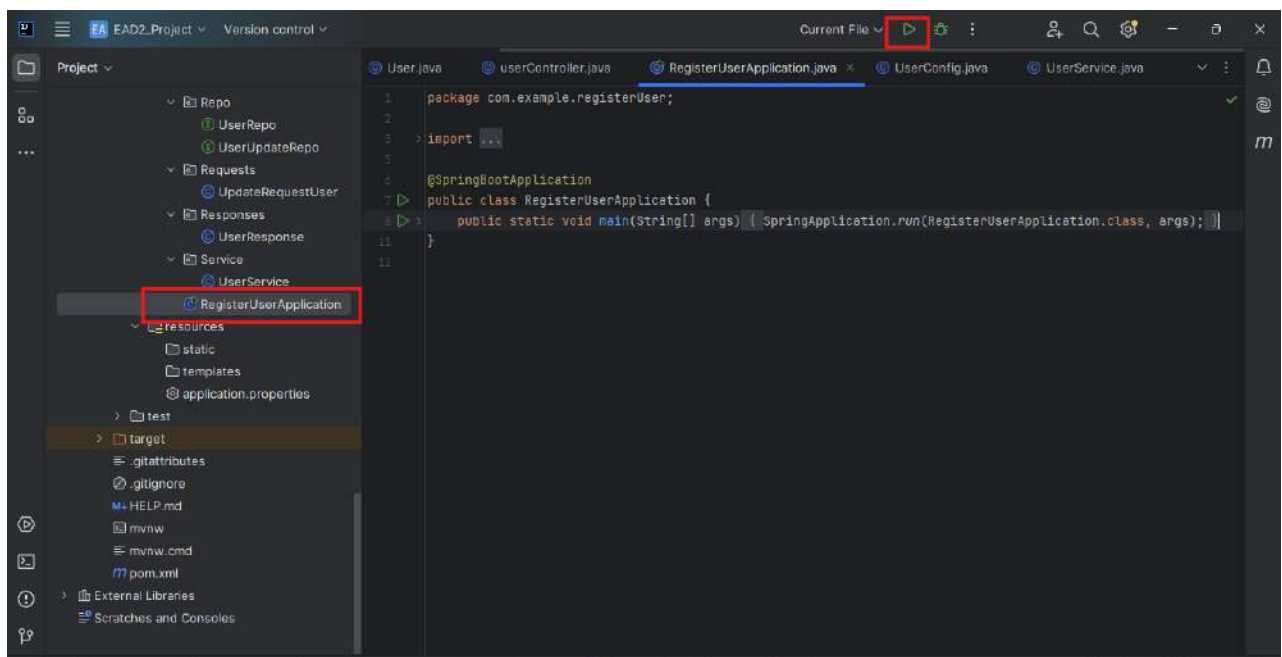
- Select the project and open the pom.xml file.



- Let the IDE complete the operations.
- The created project.



- To test the application, run the application as follows. When trying to run the application the first time, it may require to set up the SDK. To do that, click the version 22 folder and let the IDE run downloads. After that the application will run automatically.



```

package com.example.registerUser;

import org.springframework.boot.SpringApplication;

public class RegisterUserApplication {

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

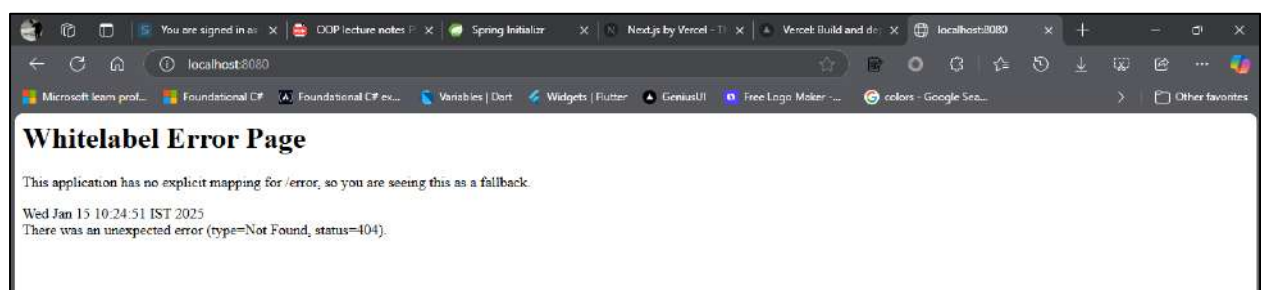
```

```

main] com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
main] com.netflix.discovery.DiscoveryClient : Application is null : false
main] com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
main] com.netflix.discovery.DiscoveryClient : Application version is -1: true
main] com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
main] com.netflix.discovery.DiscoveryClient : The response status is 200
main] com.netflix.discovery.DiscoveryClient : Starting heartbeat executor: renew interval is: 30
main] com.netflix.discovery.DiscoveryClient : InstanceInfoReplicator onDemand update allowed rate per min is 4
main] com.netflix.discovery.DiscoveryClient : Discovery Client initialized at timestamp 1739371416029 with initial instances count: 0
main] com.netflix.discovery.DiscoveryClient : Registering application REGISTER_PROCESS with eureka with status UP
main] com.netflix.discovery.DiscoveryClient : Saw local status change event StatusChangeEvent [timestamp=1739371416040, current=UP, previous=STARTING]
main] com.netflix.discovery.DiscoveryClient : DiscoveryClient_REGISTER_PROCESS/Raveen_Lenovo:register_process:8090: registering service...
main] com.netflix.discovery.DiscoveryClient : Tomcat started on port 8090 (http) with context path '/'
main] com.netflix.discovery.DiscoveryClient : Updating port to 8090
main] com.netflix.discovery.DiscoveryClient : Started RegisterUserApplication in 13.184 seconds (process running for 14.4)
main] com.netflix.discovery.DiscoveryClient : DiscoveryClient_REGISTER_PROCESS/Raveen_Lenovo:register_process:8090 - registration status: 204

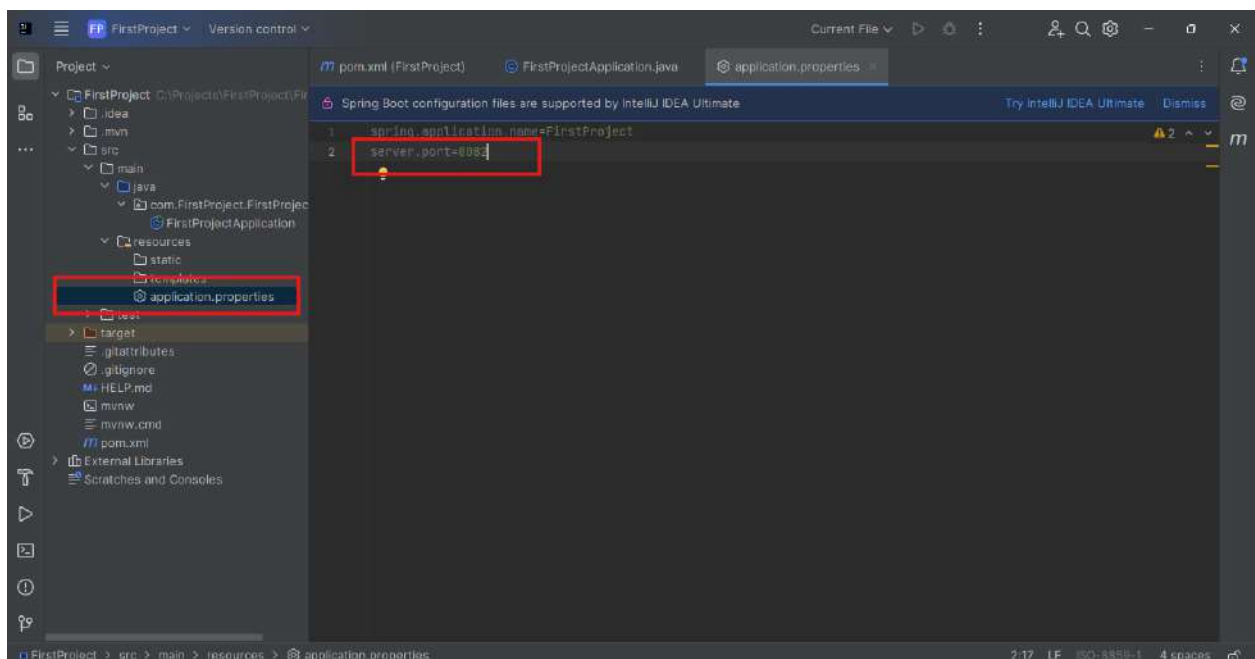
```

- When running the application, make sure the required port is free or not. Then type the address in the browser like this, localhost:<port number>.

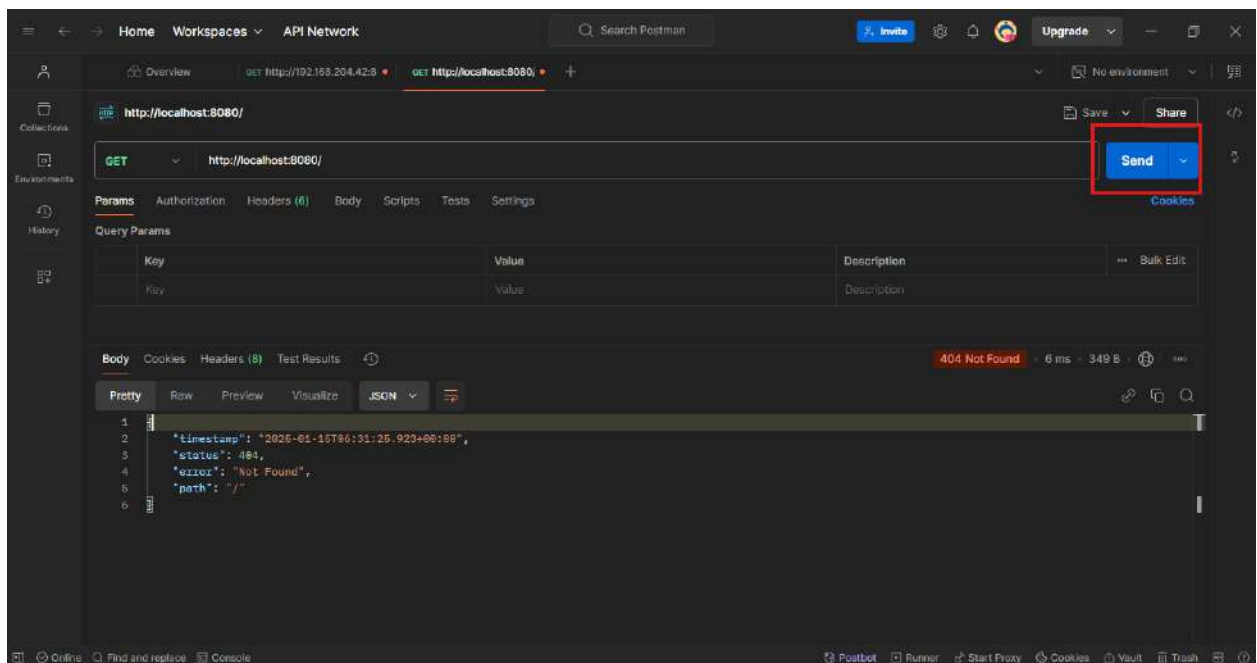
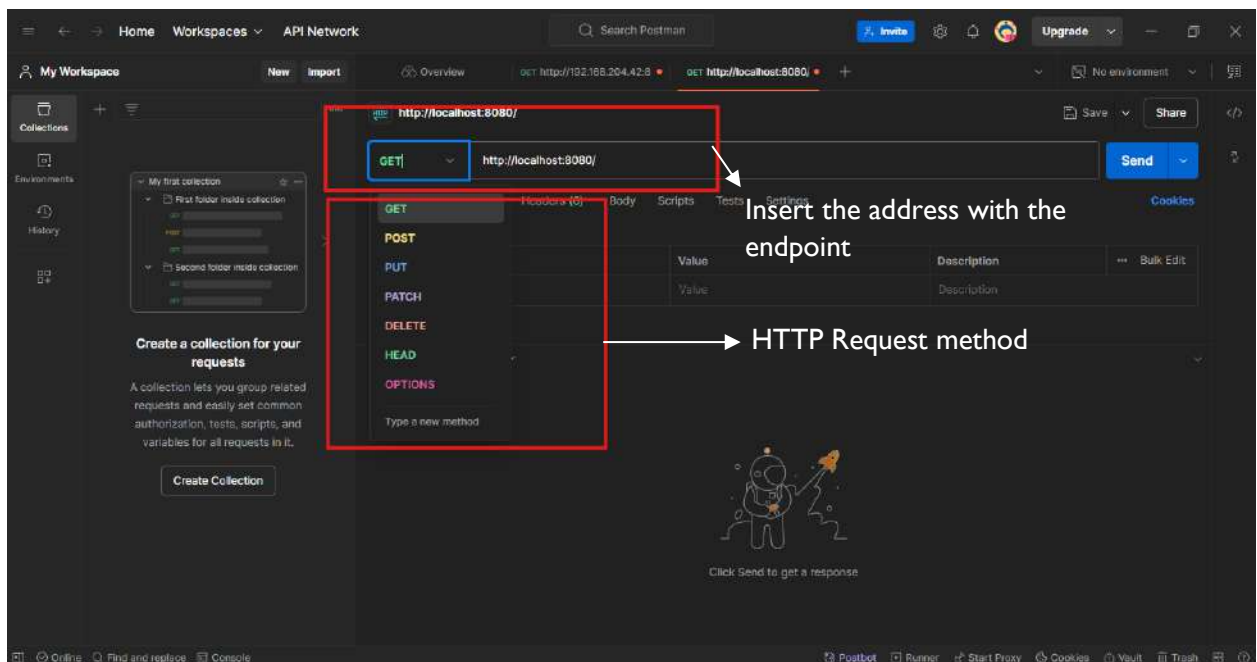


- The above page appears because there are no endpoints or logic.

Methods of changing the port.

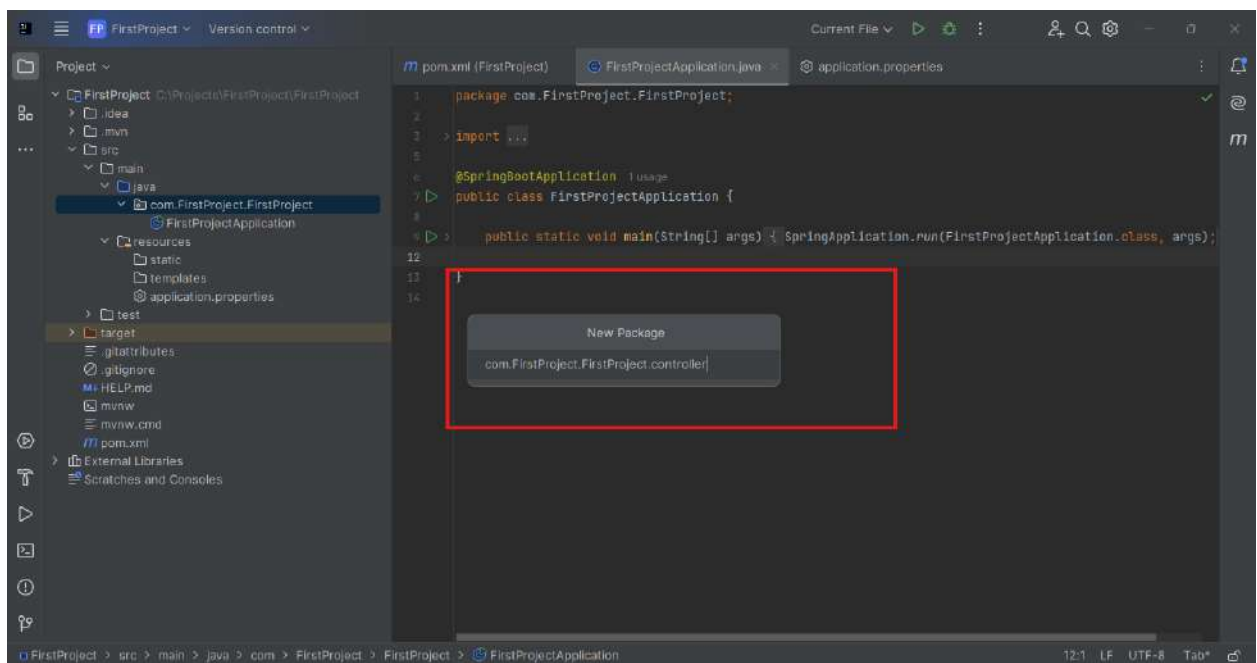
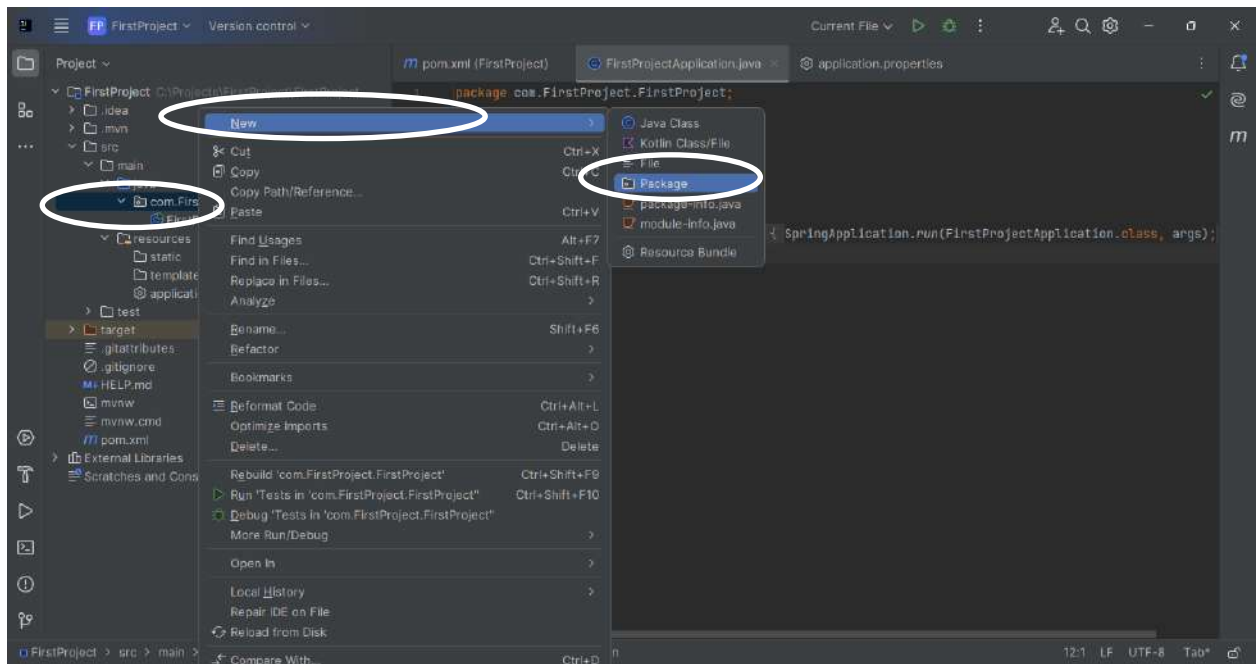


Testing the API using postman

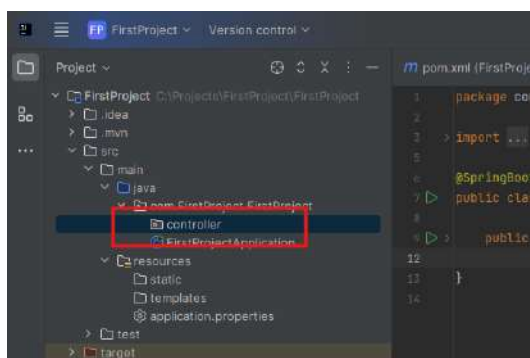


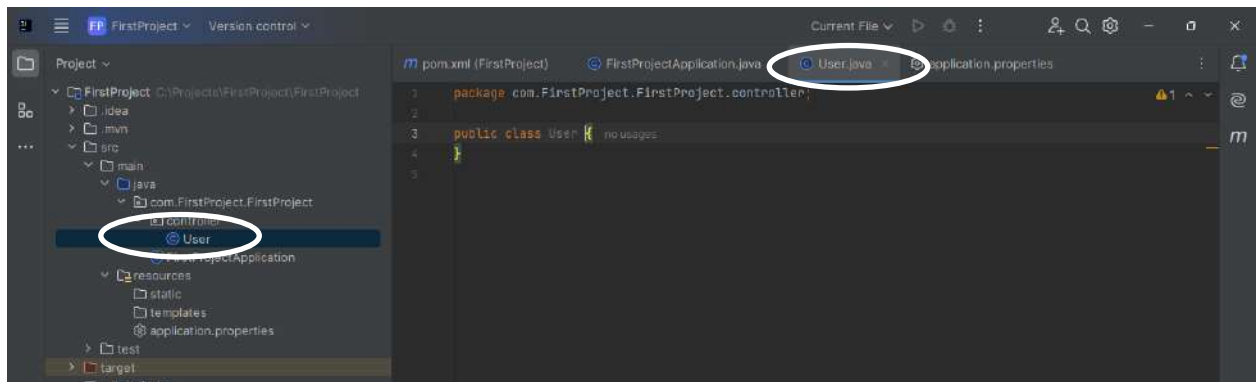
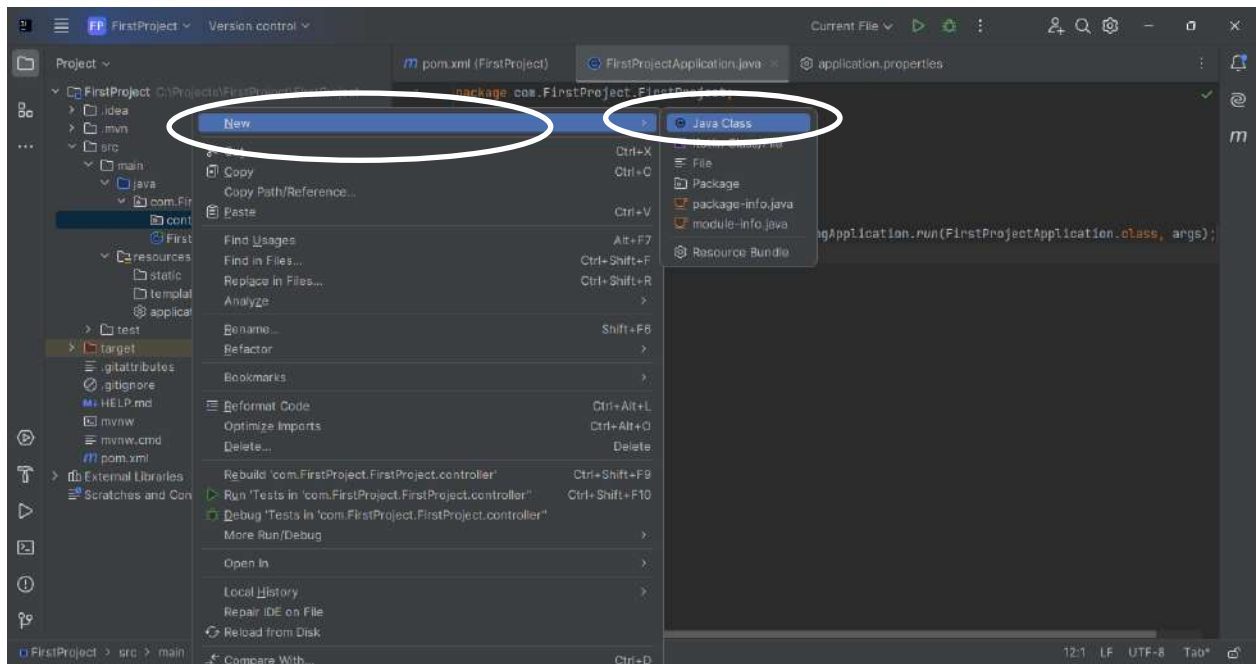
CREATING THE REST CONTROLLER (FOR ENDPOINTS)

- Create a new package for the controllers and name as follows:



- Then create a class for the controller.





- Create an endpoint to return a string.

```
package com.FirstProject.FirstProject.controller;

import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@CrossOrigin //to communicate with the 2 servers (for different requests)
@RequestMapping("api/test") //call section or accessing endpoint to the class
public class User {
    @GetMapping("/getMessage") //request endpoint
    public String getMessage() {
        return "Hallow world";
    }
}
```

- Use these annotations according to the HTTP requests.

- @PostMapping : for POST requests.
- @GetMapping : for GET requests.
- @PutMapping : for PUT requests.
- @DeleteMapping : for DELETE requests.

```
package com.FirstProject.FirstProject.controller;

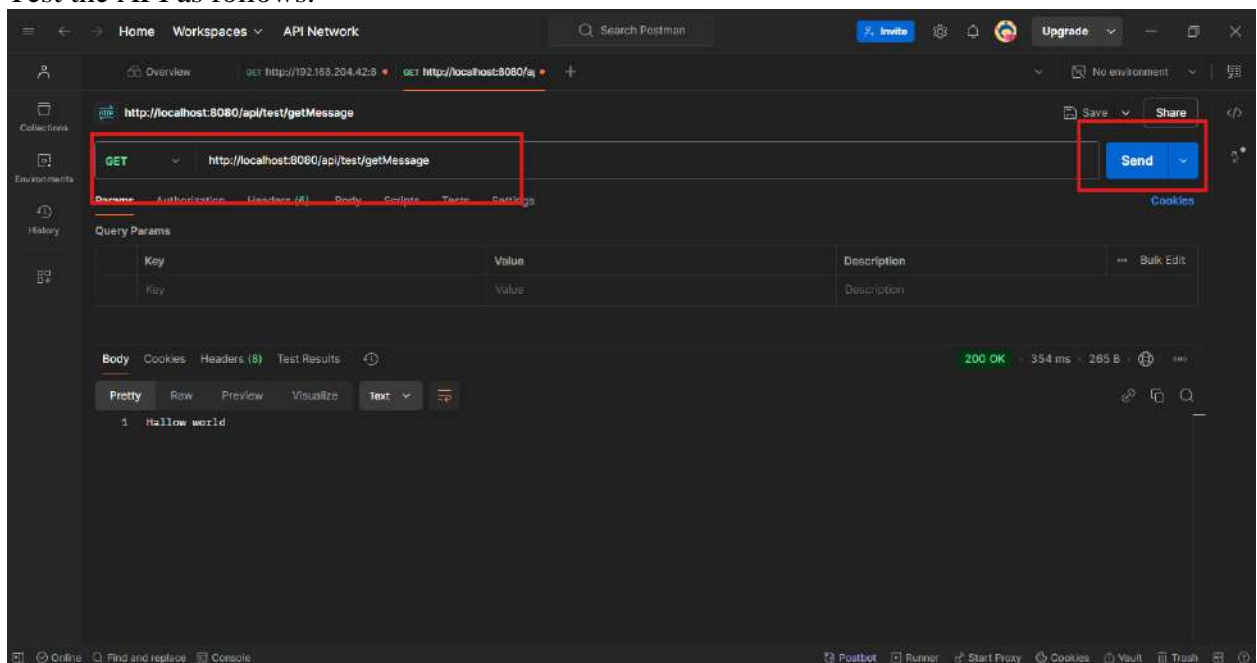
import org.springframework.web.bind.annotation.*;
```

```

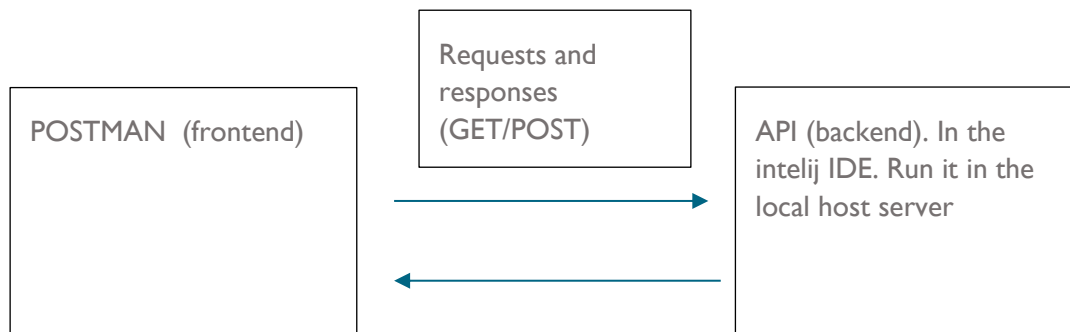
@RestController
@CrossOrigin //to communicate with the 2 servers (for different requests)
@RequestMapping("api/test")
public class User {
    @GetMapping("/Message")
    public String getMessage() {
        return "Hallow world";
    }
    @PostMapping("/Message")
    public String postMessage() {
        return "Hallow post";
    }
    @PutMapping("/Message")
    public String putMessage() {
        return "Hallow put";
    }
    @DeleteMapping
    public String deleteMessage() {
        return "Hallow Delete";
    }
}

```

- Test the API as follows.



- The postman acts as follows.



- It is possible to perform actions as follows.

```

package com.FirstProject.FirstProject.controller;
import org.springframework.web.bind.annotation.*;
@RestController

```

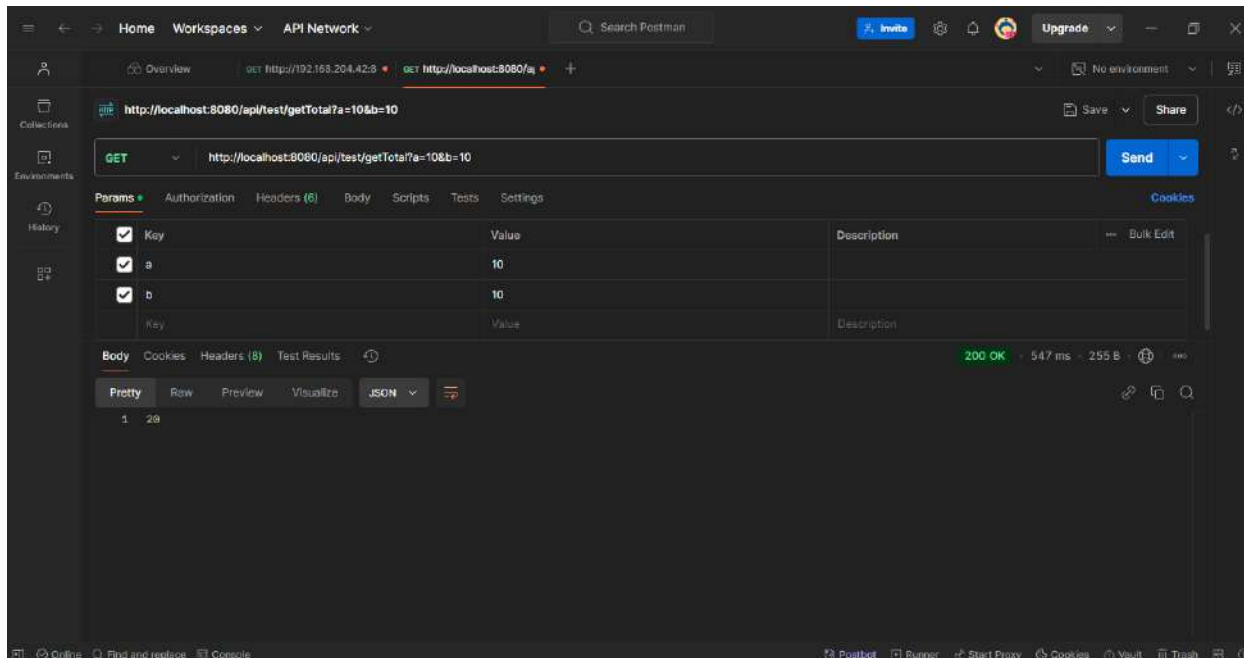


```

@CrossOrigin //to communicate with the 2 servers (for different requests)
@RequestMapping("api/test")
public class User {
    @GetMapping("/getTotal")
    public int addValues(@RequestParam int a,@RequestParam int b){
        return (a+b);
    }
}

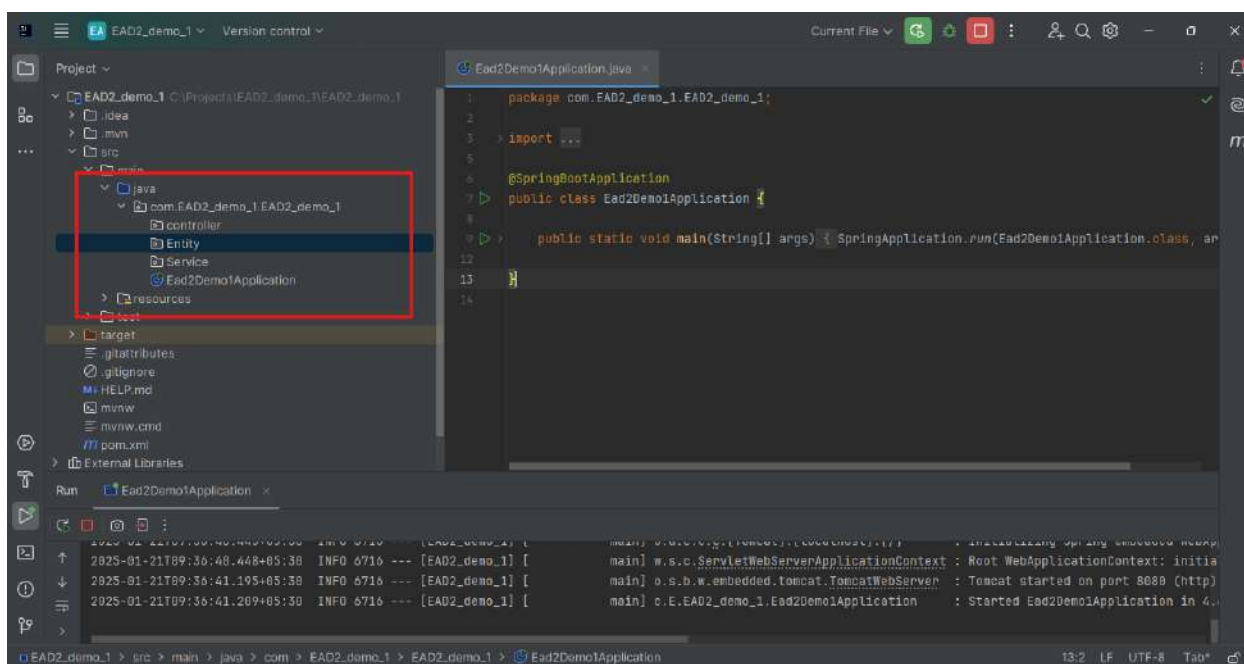
```

- Testing the API.



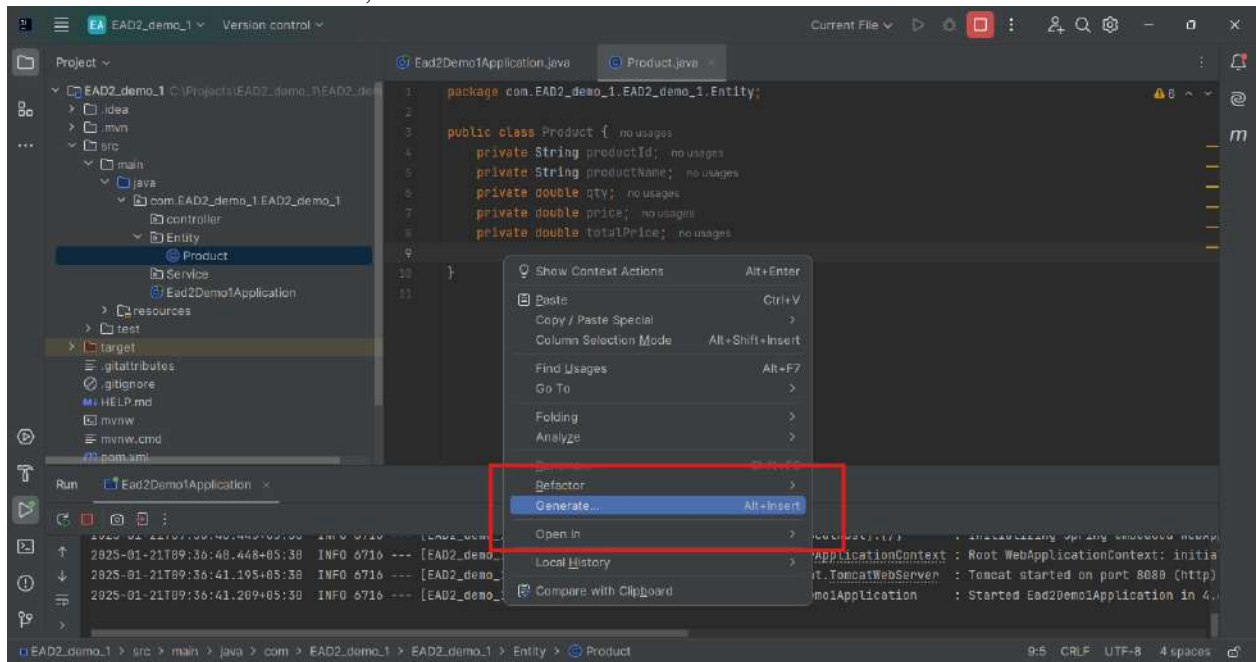
CREATE THE API USING MVC ARCHITECTURE

- For this create these packages.

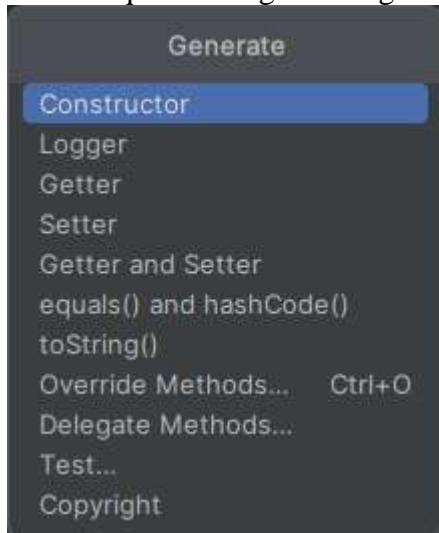


- Entity: this acts as a model or the table structure.
- Controller: this acts as the collection of endpoints to the API.
- Service: this works with the database and provides output to the controller.

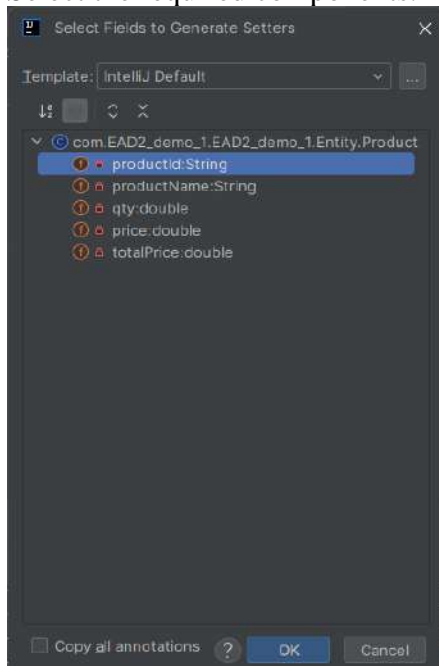
- Create the model as follows,



- Select required categories to generate.



- Select the required components.



- Generated sections.

```
package com.EAD2_demo_1.EAD2_demo_1.Entity;

public class Product {
    private String productId;
    private String productName;
    private double qty;
    private double price;
    private double totalPrice;

    public Product(String productId, String productName, double qty, double price,
double totalPrice) {
        this.productId = productId;
        this.productName = productName;
        this.qty = qty;
        this.price = price;
        this.totalPrice = totalPrice;
    }

    public Product() {
    }

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

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

    public void setQty(double qty) {
        this.qty = qty;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public void setTotalPrice(double totalPrice) {
        this.totalPrice = totalPrice;
    }

    public String getProductId() {
        return productId;
    }

    public String getProductName() {
        return productName;
    }

    public double getQty() {
        return qty;
    }

    public double getPrice() {
        return price;
    }

    public double getTotalPrice() {
        return totalPrice;
    }

    @Override
    public String toString() {
        return "Product{" +
            "productId='" + productId + '\'' +
```

```

        ", productName='" + productName + '\'' +
        ", qty=" + qty +
        ", price=" + price +
        ", totalPrice=" + totalPrice +
        '>';
    }
}

```

- This is the service logic.

```

package com.EAD2_demo_1.EAD2_demo_1.Service;

import com.EAD2_demo_1.EAD2_demo_1.Entity.Product;
import org.springframework.stereotype.Service;

@Service
public class ProductService {
    public Product getTotalAmount(Product product){
        //get the product object and do the modifications
        //and return the modified same object.
        double dis = 0;
        double total = (product.getPrice() * product.getQty());
        if(product.getQty()>10){
            dis = product.getPrice() * 0.10;
        } else if (product.getQty() > 5) {
            dis = product.getPrice() * 0.15;
        }else{
            dis = 0;
        }
        product.setTotalPrice(total - dis);
        return product;
    }
}

```

- We use the product as the model or the data structure (for the object creating and passing).

```

package com.EAD2_demo_1.EAD2_demo_1.controller;

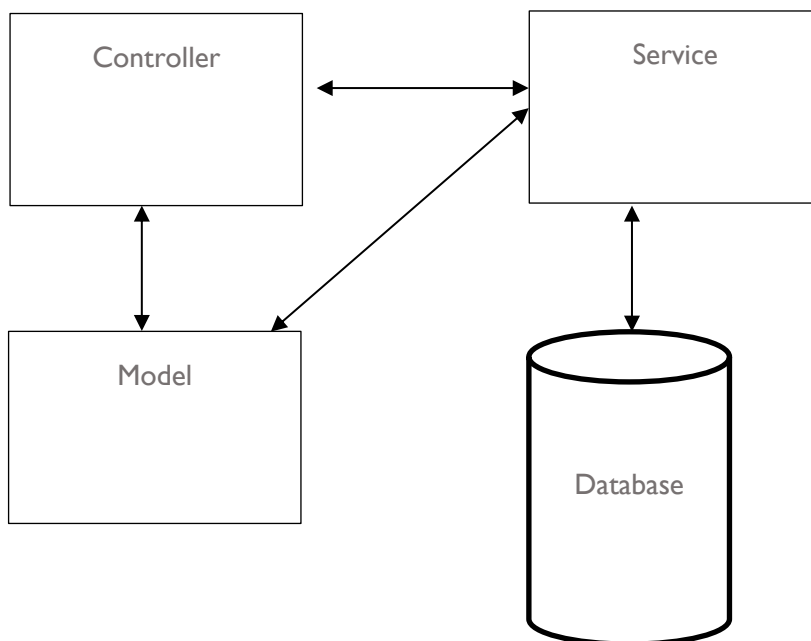
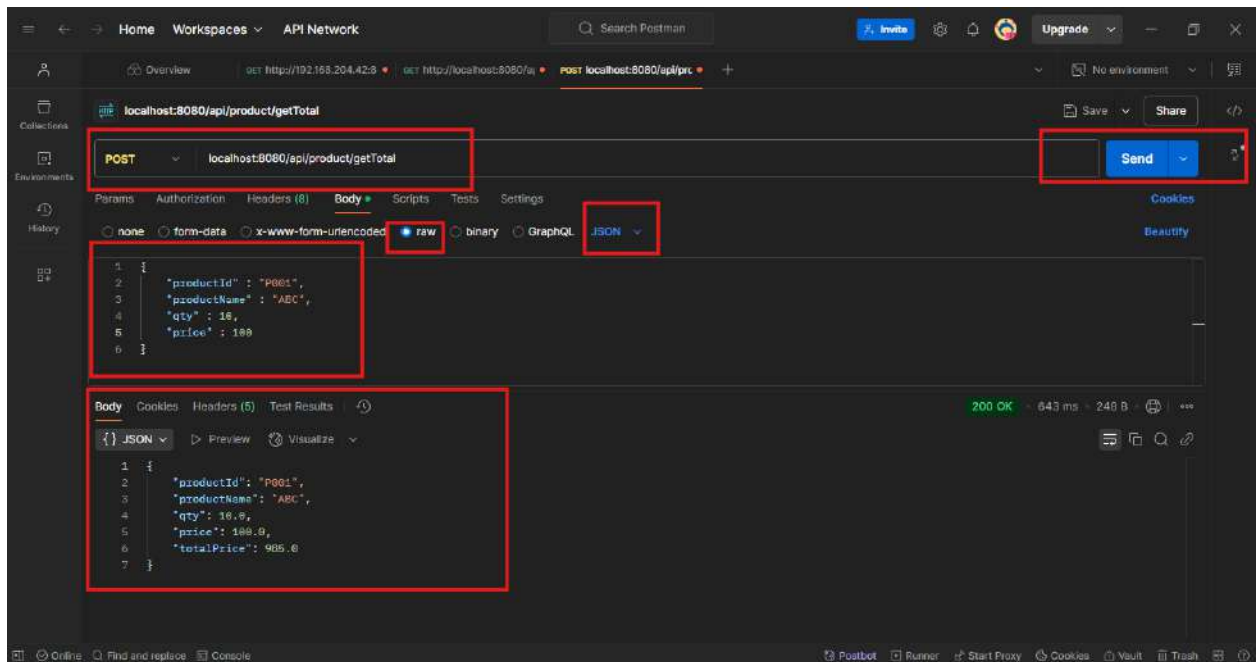
import com.EAD2_demo_1.EAD2_demo_1.Entity.Product;
import com.EAD2_demo_1.EAD2_demo_1.Service.ProductService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("api/product")
public class ProductController {
    //best practice: use the same class name as the object name
    @Autowired
    private ProductService productService;

    @PostMapping("/getTotal")
    public Product getTotal(@RequestBody Product product){
        return productService.getTotalAmount(product);
    }
}

```

- Testing the API.



CONNECTING TO DATABASES (MYSQL)

CONNECTING TO MYSQL

- Install these dependencies:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

```

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
  <dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.1.1</version>
    <scope>compile</scope>
  </dependency>

```

- Add these to the application.properties

```

eureka.instance.hostname=localhost
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
server.port=8070
spring.application.name=account_creation
spring.config.import=optional:configserver:http://localhost:8888
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/travel_project
spring.datasource.username=root
spring.datasource.password=raveen007
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true

```

- Create the entity class as follows (in the entity folder):

```

package com.example.createAccountService.Entity;

import jakarta.persistence.*;

import java.sql.Time;
import java.time.LocalDateTime;

@Entity
@Table(name = "users")
public class users {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id")
    private int user_id;
    @Column(name = "username")
    private String username;
    @Column(name = "email")
    private String email;
    @Column(name = "password_hash")
    private String password_hash;
    @Column(name = "created_at")
    private LocalDateTime created_at;

    public users(String username, String email, String password_hash, LocalDateTime created_at) {

```



```

        this.username = username;
        this.email = email;
        this.password_hash = password_hash;
        this.created_at = created_at;
    }

    public users() {

    }

    public int getUser_id() {
        return user_id;
    }

    public String getUsername() {
        return username;
    }

    public String getEmail() {
        return email;
    }

    public String getPassword_hash() {
        return password_hash;
    }

    public LocalTime getCreated_at() {
        return created_at;
    }

    public void setUser_id(int user_id) {
        this.user_id = user_id;
    }

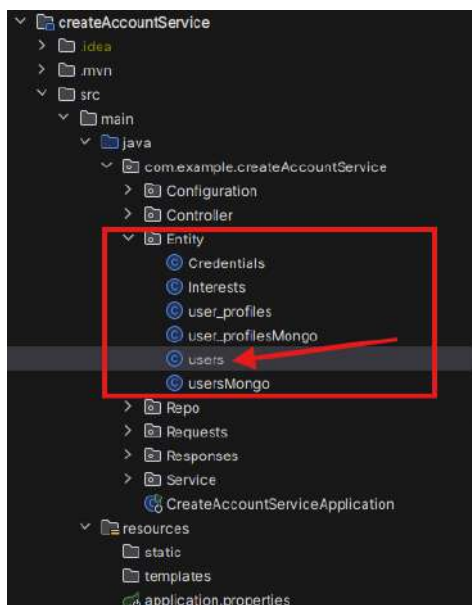
    public void setUsername(String username) {
        this.username = username;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void setPassword_hash(String password_hash) {
        this.password_hash = password_hash;
    }

    public void setCreated_at(LocalTime created_at) {
        this.created_at = created_at;
    }
}

```



- Create the table repository:

```
package com.example.createAccountService.Repo;

import com.example.createAccountService.Entity.users;
import org.springframework.data.jpa.repository.JpaRepository;

public interface usersRepo extends JpaRepository<users, Integer>{
}
```

- Create the service (to interact with the database):

```
package com.example.createAccountService.Service;

import com.example.createAccountService.Entity.Interests;
import com.example.createAccountService.Entity.users;
import com.example.createAccountService.Repo.InterestsRepo;
import com.example.createAccountService.Repo.usersRepo;
import com.example.createAccountService.Requests.Interest;
import com.example.createAccountService.Requests.InterestRequestList;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
@RequiredArgsConstructor
public class SaveAllRequests {
    @Autowired
    private ModelMapper mapper;
    @Autowired
    private InterestsRepo repo;
    @Autowired
    private usersRepo userRepo;
    public String saveAllRequests(List<Interest> allRequests){
        try{
            allRequests.forEach(interest -> {
                Optional<users>userOptional = userRepo.findById(interest.getUser_id());
                if(userOptional.isPresent()){
                    users getUser = userOptional.get();
                    Interests saveInterst = new
Interests(getUser,interest.getName(),interest.getCategory());
                    Interests savedObj = repo.save(saveInterst); //saves the data
                }
            });
            return "Saved all requests";
        }catch(Exception e){
            return "Error saving all requests: " + e.getMessage();
        }
    }
}
```

- Create the controller as follows:

```
@Autowired
private SaveAllRequests saveAll;

@PostMapping("/addAllRequests")
public String addRequests(@RequestBody InterestRequestList allObj){
    List<Interest> allRequests = allObj.getAllRequests();
    //System.out.println(allRequests);
    String status = saveAll.saveAllRequests(allRequests);
    return status;
}
```

CONNECTING MONGO DB

- Add this dependency to pom.xml:

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

- Add these to the application.properties:

```
#mongo db connection
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=travel-project-temp-db
```

- Create the entity for the table:

```
package com.example.createAccountService.Entity;

import jakarta.persistence.Column;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.time.LocalDateTime;
@Document("usersMongo")
public class usersMongo {
    @Id
    private String id;
    private String username;
    private String email;
    private String password_hash;
    private LocalDateTime created_at;
    private String verificationCode;

    public usersMongo(String username, String email, String password_hash, LocalDateTime created_at,
String verificationCode) {
        this.username = username;
        this.email = email;
        this.password_hash = password_hash;
        this.created_at = created_at;
        this.verificationCode = verificationCode;
    }

    public String getId() {
        return id;
    }

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

    public void setVerificationCode(String verificationCode) {
        this.verificationCode = verificationCode;
    }

    public String getVerificationCode() {
        return verificationCode;
    }

    public String getUsername() {
        return username;
    }

    public String getEmail() {
        return email;
    }

    public String getPassword_hash() {
        return password_hash;
    }
}
```

→ Name of the mongo DB document

```

    }

    public LocalTime getCreated_at() {
        return created_at;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void setPassword_hash(String password_hash) {
        this.password_hash = password_hash;
    }

    public void setCreated_at(LocalTime created_at) {
        this.created_at = created_at;
    }
}

```

- Create the repository according to this table:

```

package com.example.createAccountService.Repo;

import com.example.createAccountService.Entity.users;
import com.example.createAccountService.Entity.usersMongo;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface usersRepoMongo extends MongoRepository <usersMongo,String>{

}

```

- Create the service to enter data to the database:

```

package com.example.createAccountService.Service;

import com.example.createAccountService.Entity.Credentials;
import com.example.createAccountService.Entity.user_profilesMongo;
import com.example.createAccountService.Entity.usersMongo;
import com.example.createAccountService.Repo.user_profilesRepoMongo;
import com.example.createAccountService.Repo.usersRepoMongo;
import com.example.createAccountService.Responses.userMongoResponse;
import com.mailjet.client.ClientOptions;
import com.mailjet.client.MailjetClient;
import com.mailjet.client.errors.MailjetException;
import com.mailjet.client.transactional.*;
import com.mailjet.client.transactional.response.SendEmailsResponse;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;

import java.security.SecureRandom;

@RequiredArgsConstructor
public class insertTempData {
    @Autowired
    private ModelMapper mapper;
    @Autowired
    private usersRepoMongo urml;
    @Autowired
    private user_profilesRepoMongo upml;
    @Autowired
    private Credentials crel;
    public userMongoResponse saveUserTemp(usersMongo user, user_profilesMongo user_profile)
    throws MailjetException {
        try{
            usersMongo savedData = urml.save(user);
            try{
                user_profilesMongo savedData2 = upml.save(user_profile);
            }
        }
    }
}

```

```

    }catch(Exception ll){
        url1.deleteById(savedData.getId()); //delete data by ID
        userMongoResponse exceptionUser5 = new userMongoResponse();
        return exceptionUser5;
    }
    //update data
    user.setVerificationCode(String.valueOf(randomNumber));
    url1.save(user); //update the row data
    userMongoResponse u1 = new
userMongoResponse(savedData.getId(),savedData.getUsername(),savedData.getEmail(),savedData.getPa
ssword_hash(),savedData.getCreated_at(),user.getVerificationCode());
    return u1;
}catch(Exception ee){
    userMongoResponse exceptionUser5 = new userMongoResponse();
    return exceptionUser5;
}
}
}

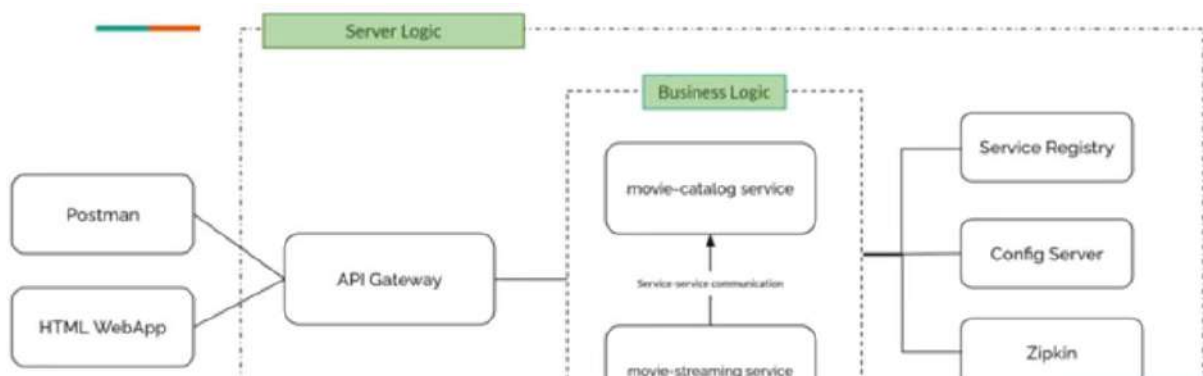
```

- Then execute this service function in the controller:

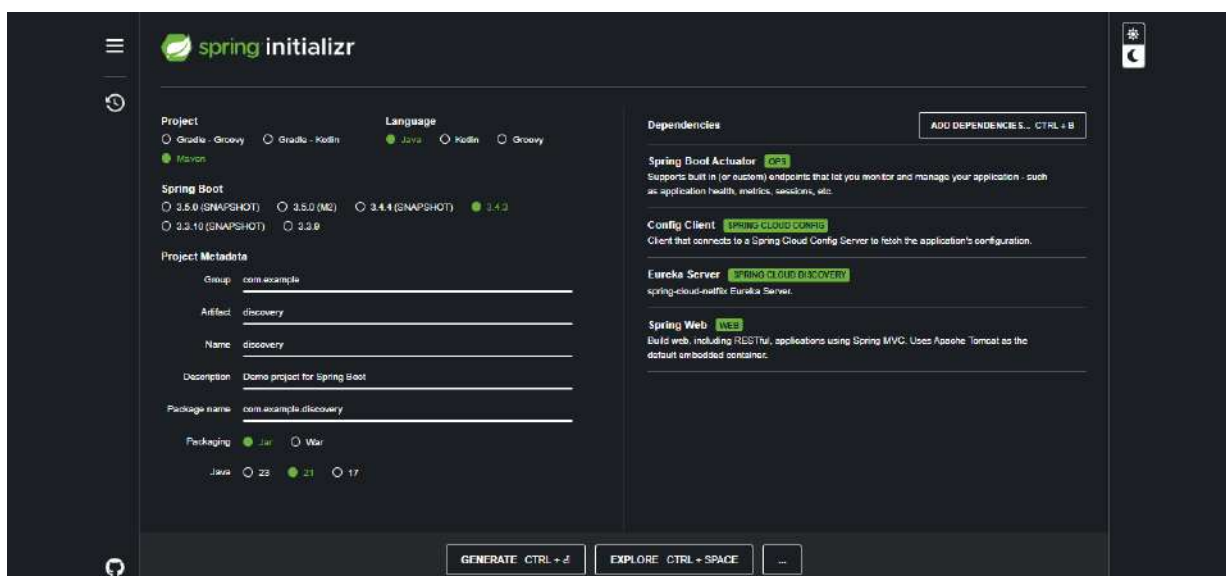
```
userMongoResponse us2 = it1.saveUserTemp(us1,us21);
```

MICROSERVICE ARCHITECTURE IN REST API

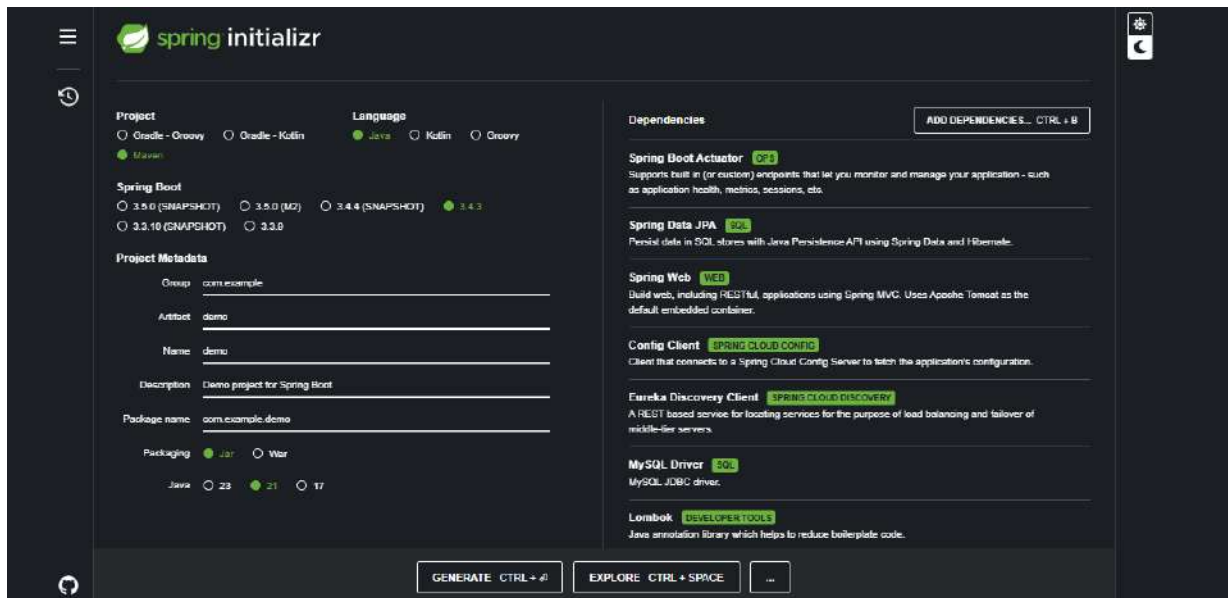
- Create the project for the discovery server.



- Add the dependencies (discovery)



- Create the project for the MySQL connectivity (service 1).



Project
☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Maven

Language
☒ Java ☐ Kotlin ☐ Groovy

Spring Boot
☐ 3.5.0 (SNAPSHOT) ☐ 3.5.0 (M2) ☐ 3.4.4 (SNAPSHOT) ☒ 3.4.3
☐ 3.3.10 (SNAPSHOT) ☐ 3.3.9

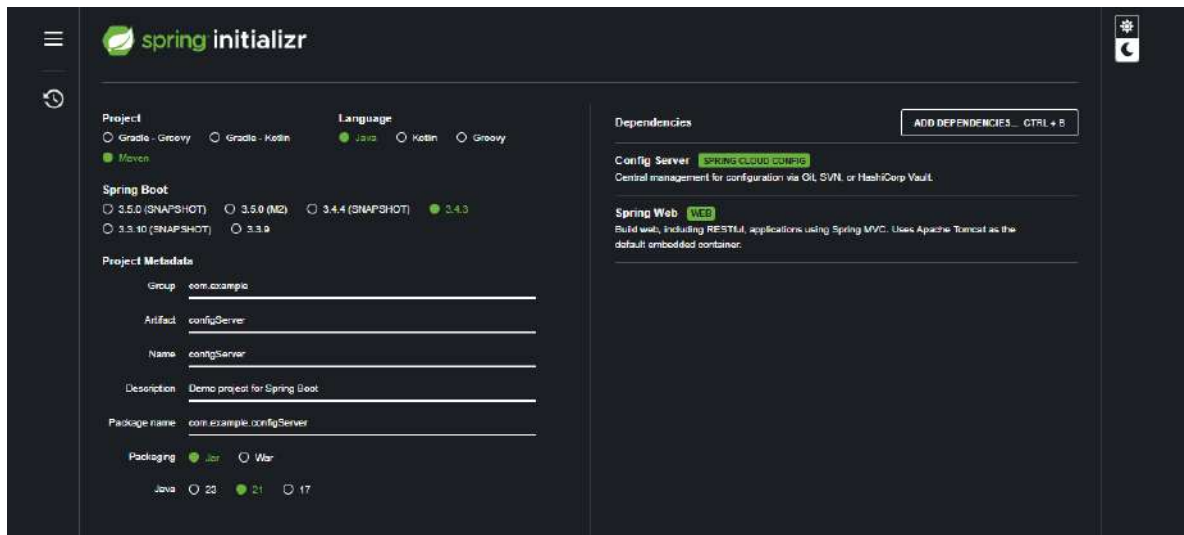
Project Metadata
 Group: com.example
 Artifact: demo
 Name: demo
 Description: Demo project for Spring Boot
 Package name: com.example.demo
 Packaging: ☒ Jar ☐ War
 Java: ☐ 23 ☒ 21 ☐ 17

Dependencies
 ADD DEPENDENCIES... CTRL + B

- Spring Boot Actuator** OPS
Supports built-in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- Spring Data JPA** SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- Spring Web** WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- Config Client** SPRING CLOUD CONFIG
Client that connects to a Spring Cloud Config Server to fetch the application's configuration.
- Eureka Discovery Client** SPRING CLOUD DISCOVERY
A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
- MySQL Driver** SQL
MySQL JDBC driver.
- Lombok** DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

GENERATE CTRL + G EXPLORE CTRL + SPACE ...

- For the config server.



Project
☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Maven

Language
☒ Java ☐ Kotlin ☐ Groovy

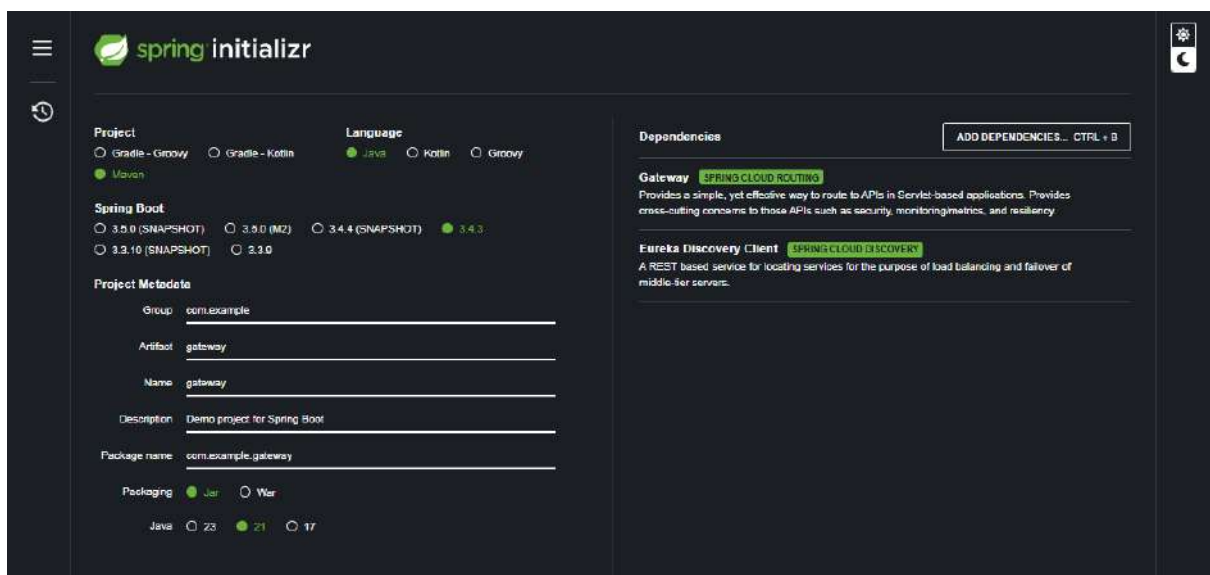
Spring Boot
☐ 3.5.0 (SNAPSHOT) ☐ 3.5.0 (M2) ☐ 3.4.4 (SNAPSHOT) ☒ 3.4.3
☐ 3.3.10 (SNAPSHOT) ☐ 3.3.9

Project Metadata
 Group: com.example
 Artifact: configServer
 Name: configServer
 Description: Demo project for Spring Boot
 Package name: com.example.configServer
 Packaging: ☒ Jar ☐ War
 Java: ☐ 23 ☒ 21 ☐ 17

Dependencies
 ADD DEPENDENCIES... CTRL + B

- Config Server** SPRING CLOUD CONFIG
Central management for configuration via Git, SVN, or HashiCorp Vault.
- Spring Web** WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

- For the gateway server.



Project
☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Maven

Language
☒ Java ☐ Kotlin ☐ Groovy

Spring Boot
☐ 3.5.0 (SNAPSHOT) ☐ 3.5.0 (M2) ☐ 3.4.4 (SNAPSHOT) ☒ 3.4.3
☐ 3.3.10 (SNAPSHOT) ☐ 3.3.9

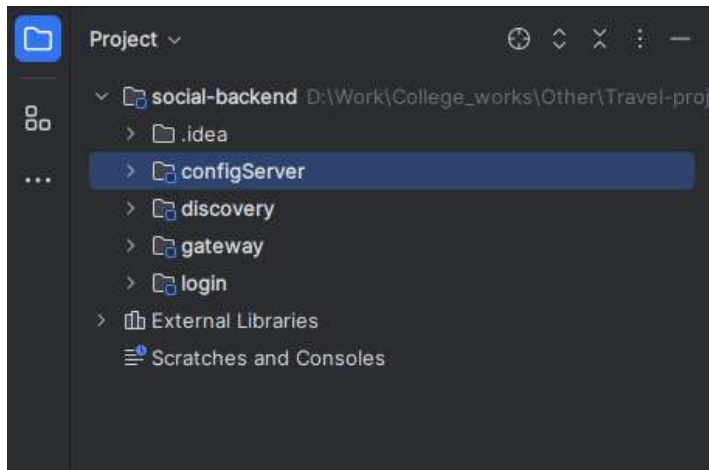
Project Metadata
 Group: com.example
 Artifact: gateway
 Name: gateway
 Description: Demo project for Spring Boot
 Package name: com.example.gateway
 Packaging: ☒ Jar ☐ War
 Java: ☐ 23 ☒ 21 ☐ 17

Dependencies
 ADD DEPENDENCIES... CTRL + B

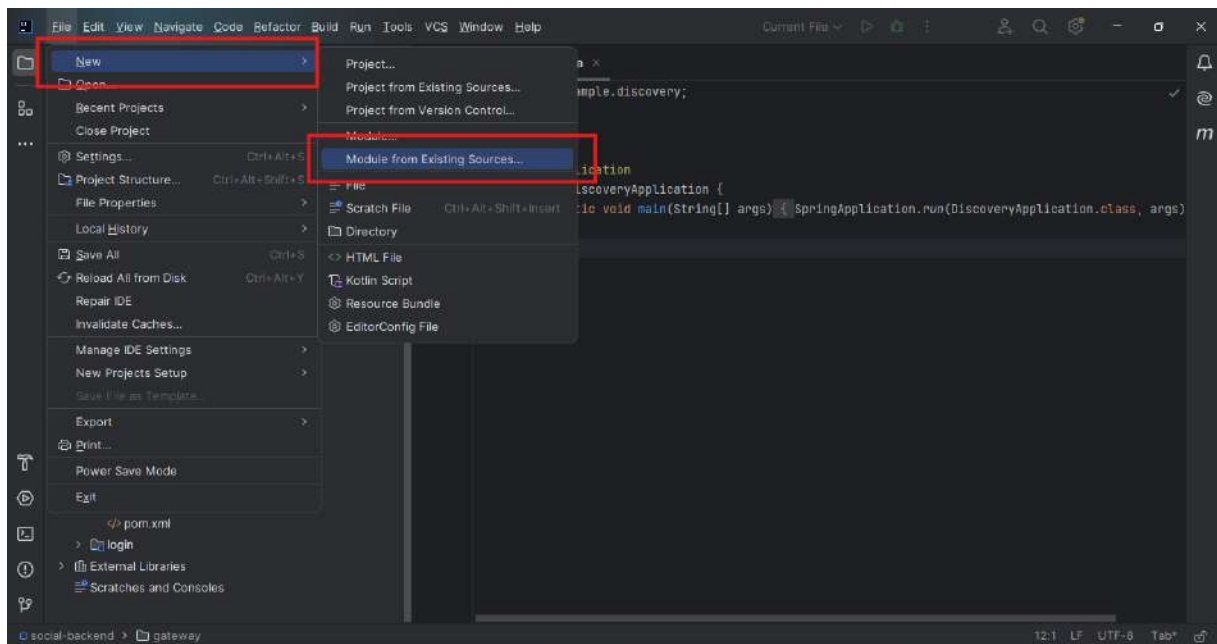
- Gateway** SPRING CLOUD ROUTING
Provides a simple, yet effective way to route to APIs in Service-based applications. Provides cross-cutting concerns to those APIs such as security, monitoring/metrics, and resiliency.
- Eureka Discovery Client** SPRING CLOUD DISCOVERY
A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.

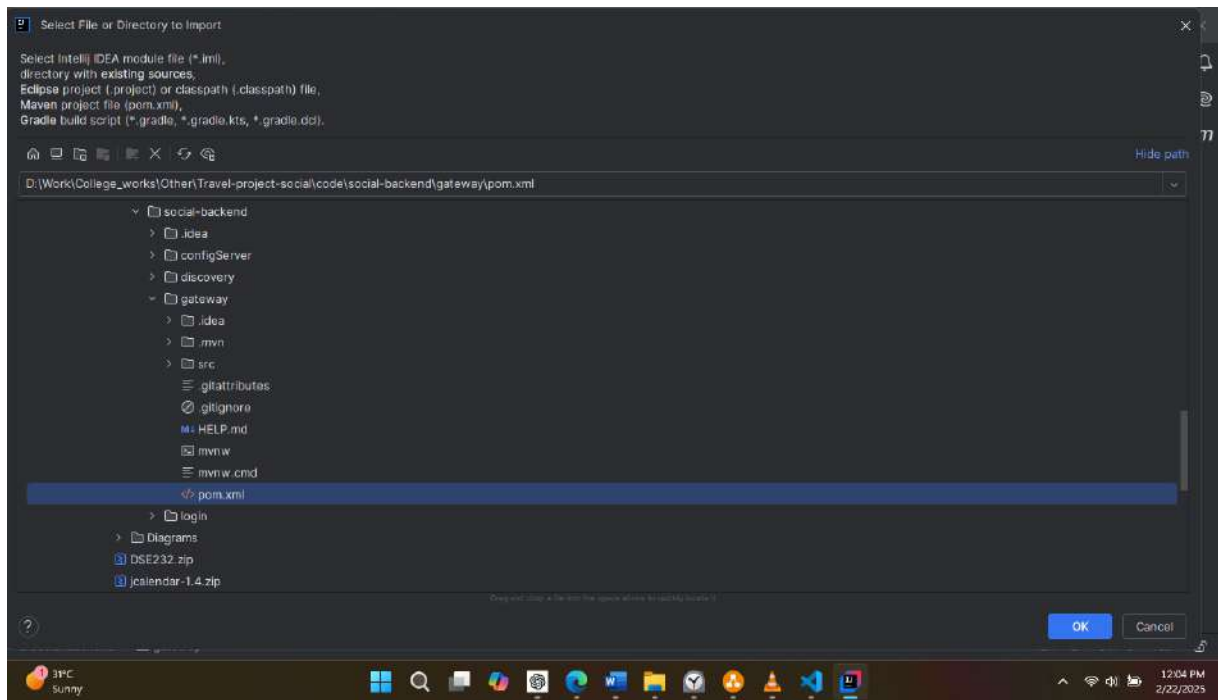
- Extract all the projects and insert them into a one folder and open It (if necessary, open the all extracted projects separately to configure maven).

- Then open the main folder.



- Then open modules from existing resources.
 - Click on the project.





CONFIGURING THE DISCOVERY SERVER

- Configure the eureka server in application.properties:

```
eureka.instance.hostname=localhost
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
eureka.client.serviceUrl.defaultZone=http://localhost:${server.port}/eureka/

server.port=8761

spring.application.name=eureka-server
spring.config.import=optional:configserver:http://localhost:8888/eureka
```

- In the main application (.java file), insert these annotations:

```
package com.example.discovery;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

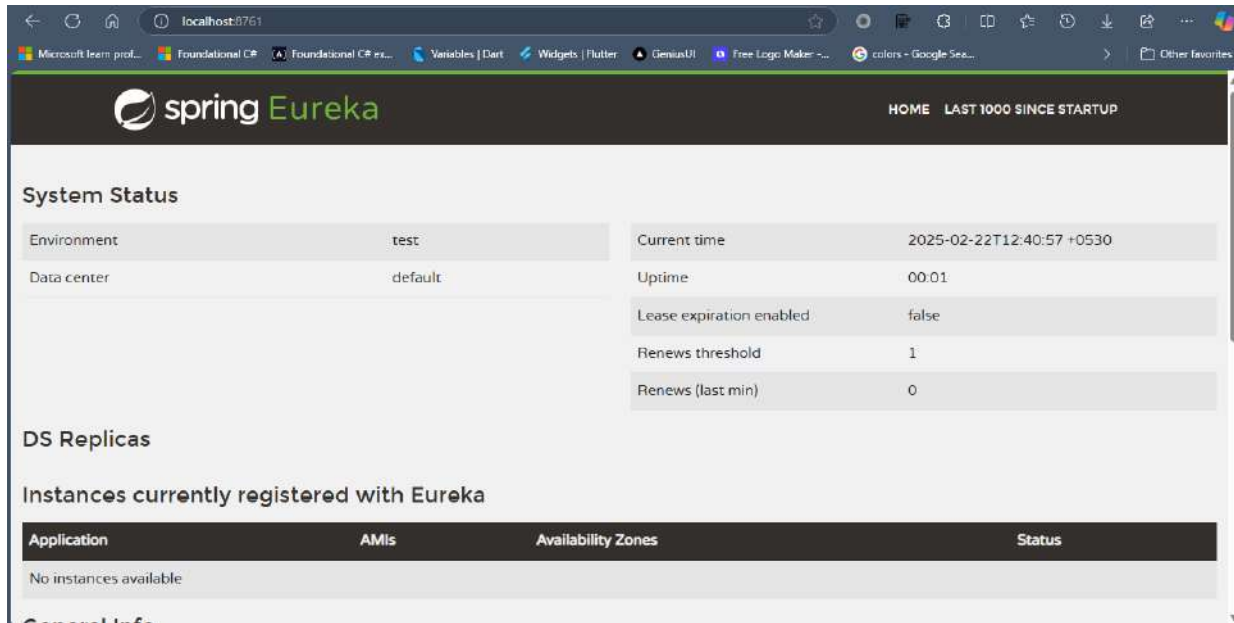
@EnableEurekaServer
@SpringBootApplication
public class DiscoveryApplication {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryApplication.class, args);
    }
}
```

- Then start the server:

```

Run  DiscoveryApplication x
14872 --- [eureka-server] [main] o.s.c.n.e.s.EurekaServiceRegistry : Registering application EUREKA-SERVER with eureka with status UP
14872 --- [eureka-server] [Thread-0] o.s.c.n.e.server.EurekaServerBootstrap : isAsms returned false
14872 --- [eureka-server] [Thread-0] o.s.c.n.e.server.EurekaServerBootstrap : Initialized server context
14872 --- [eureka-server] [Thread-0] c.n.e.r.PeerAwareInstanceRegistryImpl : Got 1 instances from neighboring DS node
14872 --- [eureka-server] [Thread-0] c.n.e.r.PeerAwareInstanceRegistryImpl : Renew threshold is: 1
14872 --- [eureka-server] [Thread-0] c.n.e.r.PeerAwareInstanceRegistryImpl : Changing status to UP
14872 --- [eureka-server] [Thread-0] o.s.c.n.e.s.EurekaServerBootstrap : Started Eureka Server
14872 --- [eureka-server] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8761 (http) with context path '/'
14872 --- [eureka-server] [main] o.s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8761
14872 --- [eureka-server] [main] o.o.discovery.DiscoveryApplication : Started DiscoveryApplication in 8.249 seconds (process running for 9.515)
14872 --- [eureka-server] [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
14872 --- [eureka-server] [nio-8761-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
14872 --- [eureka-server] [nio-8761-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
14872 --- [eureka-server] [nio-8761-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
14872 --- [eureka-server] [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
14872 --- [eureka-server] [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 2ms

```



REGISTERING MICROSERVICES TO THE DISCOVERY SERVER

- Create the logic and edit the application.properties as below:

```

eureka.instance.hostname=localhost
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

server.port=8090

spring.application.name=login_process
spring.config.import=optional:configserver:http://localhost:8888
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAuto
Configuration

```

- This registers the service to the discovery server (without a database). Or we can add a MySQL database while registering with the server.

```

eureka.instance.hostname=localhost
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

server.port=8090

spring.application.name=login_process
spring.config.import=optional:configserver:http://localhost:8888
#spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAut
oConfiguration

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

```

```

spring.datasource.url=jdbc:mysql://localhost:3306/trainService
spring.datasource.username=root
spring.datasource.password=raveen007

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true

```

- Then start the service.

```

main] c.n.discovery.InstanceInfoReplicator : InstanceInfoReplicator onDemand update allowed rate per min is 4
main] com.netflix.discovery.DiscoveryClient : Discovery Client initialized at timestamp 1740209387457 with initial instances count: 1
main] o.s.c.n.e.s.EurekaServiceRegistry : Registering application LOGIN_PROCESS with eureka with status UP
main] com.netflix.discovery.DiscoveryClient : Saw local status change event StatusChangeEvent [timestamp=1740209387475, current=UP, previous=STARTING]
main] com.netflix.discovery.DiscoveryClient : DiscoveryClient_LOGIN_PROCESS/Raveen_Lenovo:login_process:8090: registering service...
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8090 (http) with context path '/'
main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8090
main] com.example.login.LoginApplication : Started LoginApplication in 14.298 seconds (process running for 15.537)
main] com.netflix.discovery.DiscoveryClient : DiscoveryClient_LOGIN_PROCESS/Raveen_Lenovo:login_process:8090 - registration status: 284

```

- Refresh the eureka server.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
LOGIN_PROCESS	n/a (1)	(1)	UP (1) - Raveen_Lenovo:login_process:8090

General Info

Name	Value
total-avail-memory	81mb
num-of-cpus	4
current-memory-usage	62mb (76%)
server-uptime	00:21
registered-replicas	
unavailable-replicas	

CONNECTING MYSQL DATABASE

- Install these dependencies and create the project:

```

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

```

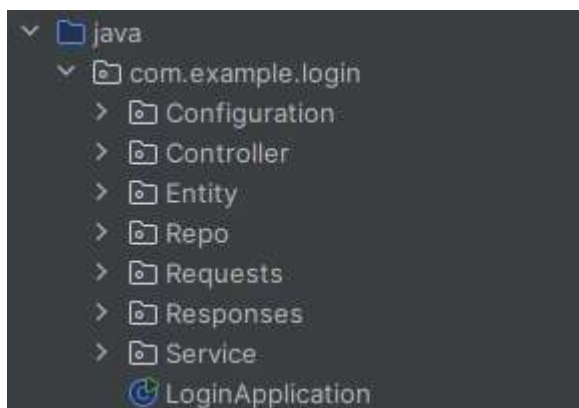
```

    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.1.1</version>
</dependency>

```

- Create the following file structure:



- First create the entity class. This maps the tables in the database. We can create one class file for each table.

```

package com.example.login.Entity;

import jakarta.persistence.*;

import java.sql.Time;

@Entity
@Table(name = "users")
public class users {

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "user_id")
private int user_id;
@Column(name = "username")
private String username;
@Column(name = "email")
private String email;
@Column(name = "password_hash")
private String password_hash;
@Column(name = "created_at")
private Time created_at;

public int getUser_id() {
    return user_id;
}

public String getUsername() {
    return username;
}

public String getEmail() {
    return email;
}

public String getPassword_hash() {
    return password_hash;
}

public Time getCreated_at() {
    return created_at;
}

public void setUser_id(int user_id) {
    this.user_id = user_id;
}

public void setUsername(String username) {
    this.username = username;
}

public void setEmail(String email) {
    this.email = email;
}

public void setPassword_hash(String password_hash) {
    this.password_hash = password_hash;
}

public void setCreated_at(Time created_at) {
    this.created_at = created_at;
}
}

```

- Second, create the repositories for the tables as interfaces.

```

package com.example.login.Repo;

import com.example.login.Entity.users;
import org.springframework.data.jpa.repository.JpaRepository;

public interface usersRepo extends JpaRepository<users, Integer> {
}

```

- Create the response and request classes.

- Request:

```
package com.example.login.Requests;

import jakarta.persistence.*;

import java.sql.Time;

@Entity
@Table(name = "users")
public class logIn_request {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id")
    private int user_id;
    @Column(name = "username")
    private String username;
    @Column(name = "password_hash")
    private String password_hash;

    public int getUser_id() {
        return user_id;
    }

    public String getUsername() {
        return username;
    }

    // public String getEmail() {
    //     return email;
    // }

    public String getPassword_hash() {
        return password_hash;
    }

    // public Time getCreated_at() {
    //     return created_at;
    // }

    public void setUser_id(int user_id) {
        this.user_id = user_id;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    // public void setEmail(String email) {
    //     this.email = email;
    // }

    public void setPassword_hash(String password_hash) {
        this.password_hash = password_hash;
    }

    // public void setCreated_at(Time created_at) {
    //     this.created_at = created_at;
    // }
}
```

- Response:

```
package com.example.login.Responses;

import jakarta.persistence.*;
```

```

import java.sql.Time;

@Entity
@Table(name = "users")
public class userResponse {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id")
    private int user_id;
    @Column(name = "username")
    private String username;
    @Column(name = "email")
    private String email;
    @Column(name = "password_hash")
    private String password_hash;
    @Column(name = "created_at")
    private Time created_at;

    public int getUser_id() {
        return user_id;
    }

    public String getUsername() {
        return username;
    }

    public String getEmail() {
        return email;
    }

    public String getPassword_hash() {
        return password_hash;
    }

    public Time getCreated_at() {
        return created_at;
    }

    public void setUser_id(int user_id) {
        this.user_id = user_id;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void setPassword_hash(String password_hash) {
        this.password_hash = password_hash;
    }

    public void setCreated_at(Time created_at) {
        this.created_at = created_at;
    }
}

```

- Create the service to interact with the database.
 - For selecting all data.

```
package com.example.login.Service;
```

```

import com.example.login.Entity.users;
import com.example.login.Repo.usersRepo;
import com.example.login.Requests.logIn_request;
import com.example.login.Responses.userResponse;
import jakarta.transaction.Transactional;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.*;

public class login_service {
    @Autowired
    private ModelMapper mapper;
    @Autowired
    private usersRepo repo;
    @Transactional
    public String logInProcess(logIn_request us){
        List<users> user = repo.findAll();
        String[] arr1 = new String[2];
        arr1[0] = "NONE";
        String userName = us.getUsername();
        String pass = us.getPassword_hash();
        user.forEach((p)->{
            if(p.getUsername().equals(userName) && p.getPassword_hash().equals(pass)){
                arr1[0] = "APPROVED";
            }
        });
        return arr1[0];
    }
}

```

- Create the configuration class as follows:

```

package com.example.login.Configuration;

import com.example.login.Service.login_service;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class userConfig {
    /*
     * In a Spring Boot application, defining beans using @Bean inside a @Configuration
     class like UserConfig explicitly tells Spring to create and manage instances of these
     objects (for AutoWire).
     *
     * Spring Boot automatically scans for components annotated with @Component, @Service,
     @Repository, etc., and registers them as Spring Beans.
     * If you annotate UserService with @Service, Spring will automatically manage it
     without needing a @Bean declaration.
     *
     * Why Use @Bean in UserConfig?
     *   When You Don't Want to Use @Service, @Component, or @Repository
     *   If UserService is from a third-party library or lacks annotations, you can
     manually declare it as a Spring Bean.
     *   When You Need to Customize Bean Initialization
     *   If a class requires custom setup before being used, defining it in a
     @Configuration class gives you more control.
     *   To Create and Share a Singleton Bean
     *   Spring Beans by default are singletons, meaning only one instance exists for
     the entire application.
     *   Example: ModelMapper is a utility that should be shared across the app.
     *   When You Need Conditional Bean Creation
     * */
    @Bean

```

```

public login_service logInBean() {
    return new login_service();
}
@Bean
public ModelMapper modelMapperBean() {
    return new ModelMapper();
}
}

```

- Finally create the controller and the endpoints:

```

package com.example.login.Controller;

import com.example.login.Entity.users;
import com.example.login.Requests.logIn_request;
import com.example.login.Responses.userResponse;
import com.example.login.Service.login_service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@CrossOrigin(origins = "http://localhost:5173")
@RequestMapping("user")
public class UserController {
    @Autowired
    private login_service logService;
    @GetMapping("/loginCheck")
    public String getMessage(@RequestBody logIn_request l1) {
        String test1 = "";
        if(l1.getUsername().equals("") || l1.getPassword_hash().equals("")){
            test1 = "ERROR";
            return test1;
        }else{
            test1 = logService.logInProcess(l1);
            return test1;
        }
    }
}

```

- Configure the application.properties (inside the resources folder) for MYSQL connections:

```

eureka.instance.hostname=localhost
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

server.port=8090

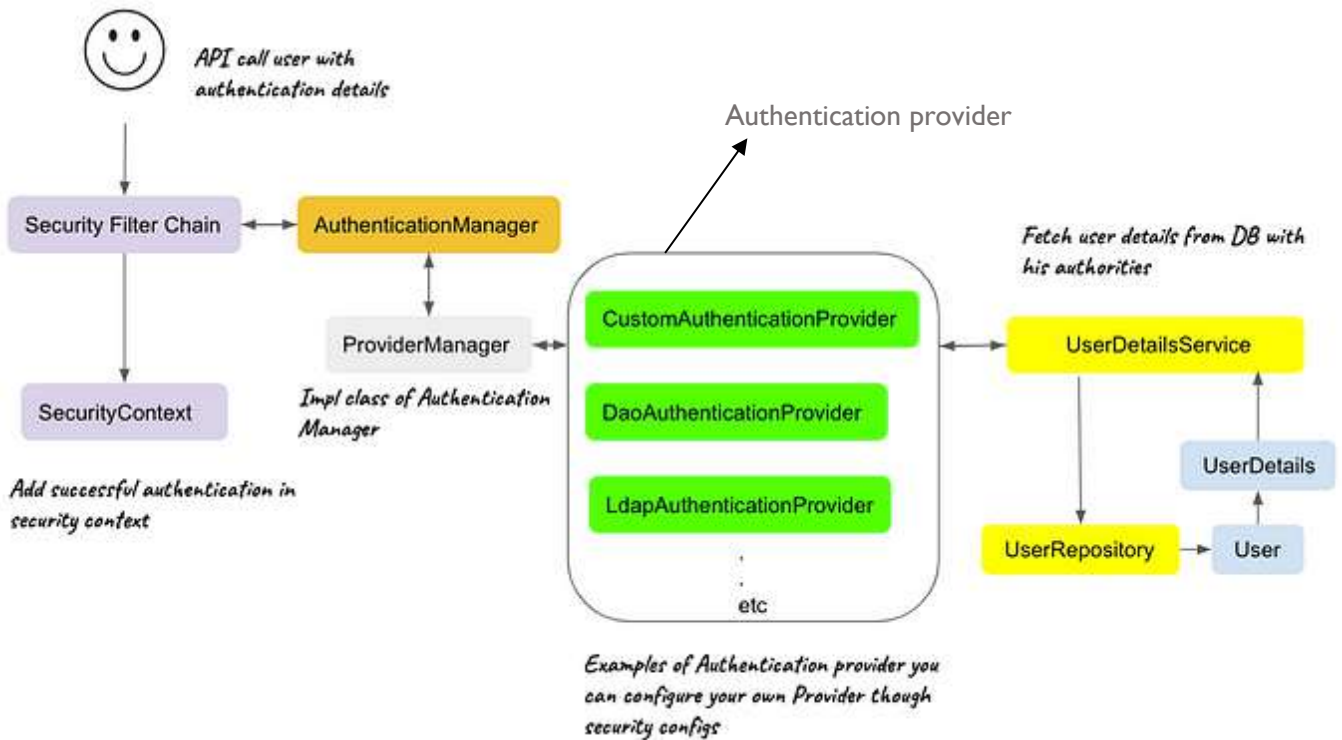
spring.application.name=login_process
spring.config.import=optional:configserver:http://localhost:8888
#spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/travel_project
spring.datasource.username=root
spring.datasource.password=raveen007

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true

```

SPRING SECURITY FLOW 3.0



1. Security Filter Chain

We most of the time use spring boot security in our applications to secure our application it can be either for authentication or authorization, but do we understand how spring internally handles the authentication and authorization.

The Security Filter Chain is a crucial component in Spring Security, responsible for processing incoming requests before they reach the application. It consists of a series of filters that handle various security-related tasks, such as authentication and authorization.

- **Security Filters:** When a user makes a call to a secured API, the request first passes through the security filter chain. Filters like UsernamePasswordAuthenticationFilter and BasicAuthenticationFilter extract the username and password from the request and create an authentication object.
- **Authentication Manager:** The AuthenticationManager interface, implemented by ProviderManager, processes the Authentication object. It selects the appropriate AuthenticationProvider based on the authentication class.
- **Authentication Provider:** These components handle specific types of authentications, such as LDAP or database user authentication. Custom authentication providers can also be configured.
- **UserDetailsService:** This service interacts with the database to fetch user details for authentication. The loadByUsername method retrieves user information, which is then used by the AuthenticationProvider.
- **Security Context:** Once authentication is successful, Spring Security saves the authenticated user details in SecurityContext and wraps it using SecurityContextHolder. If the SecurityContextHolder is populated, it assumes the user is authenticated.

Security Filters extracts the username and password from request and then spring security creates an authentication object out of it which then can be used for validation and actual authentication purpose.

2. Authentication manager

- **AuthenticationManager** is the component where our `authenticate(-)` method is there but **AuthenticationManager** is an interface and we have **ProviderManager** which is the implementation for our AuthenticationManager.

- Example:

```
public interface AuthenticationManager {  
    Authentication authenticate(Authentication authentication) throws  
    AuthenticationException;  
}
```

- **ProviderManager** is the class which contains the actual implementation of `authenticate(-)` method, it also contains the logic to choose **AuthenticationProvider** for authentication based on the Authentication class.
- so once we get the Authentication object from filters that object is passed to `authenticate(-)` method as method argument through AuthenticationManager -> ProviderManager.

3. Authentication Provider

- **AuthenticationProvider** are the components which are responsible to process the request for certain type of authentication, it can be LDAP authentication, database user authentication or even our custom authentication provider which we can use by configuring it through security configuration.
- In **AuthenticationProvider** we can get the user and perform the authentication.
- Example:

```
public interface AuthenticationProvider {  
    Authentication authenticate(Authentication authentication) throws  
    AuthenticationException;  
    boolean supports(Class<?> authentication);  
}
```

4. UserDetailsService

- We have **UserDetailsService** which then talks with Database to fetch the authenticating user details and they are used in **AuthenticationProvider**.
- The **UserDetailsService** has **loadByUsername(String username)** method which we can use to fetch user from DB.

5. Security Context

- The Security Context is a container object that:
 1. Stores information about the currently authenticated user (principal).
 2. Includes their credentials (if available) and authorities (roles or permissions).
 3. Provides a consistent way for the application to access the authenticated user's identity.
- A successful authentication object is created and stored in the security context.

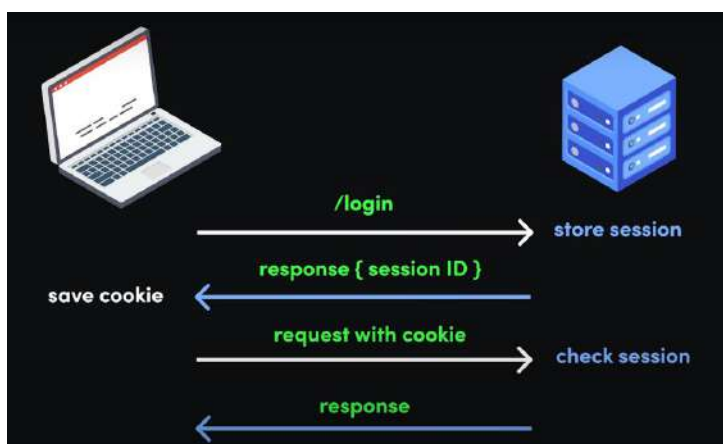
- Central to Spring Security, it holds authentication details (e.g., authenticated user and their roles).
- Reasons for using security context:
 1. Access control.
 2. Stateless Authentication.
 3. Session management.
 4. Thread-safety.
- In here we set an authentication object. This represents the principal (user) and their authentication details.
 1. Key attributes
 - **Principle:** Often a UserDetails object containing username, password, and roles.
 - **Credentials:** Represents the user's credentials (e.g., password). Usually set to null after authentication for security.
 - **Authorities:** Represents the user's roles or permissions. Example: [ROLE_USER,ROLE_ADMIN].
 - **Authenticated Flag:** Indicates whether the user has been successfully authenticated.
 - **Additional Details:** Metadata such as IP address, session ID, etc.

Source: [Spring Boot Security 3.0 Flow. we most of the times use spring boot... | by Shubham Wankhede / Medium](#)

AUTHENTICATION IN SPRING BOOT

In user authentication, there are 2 ways to implement that, the first method is sessions and JWT tokens.

- Session approach.

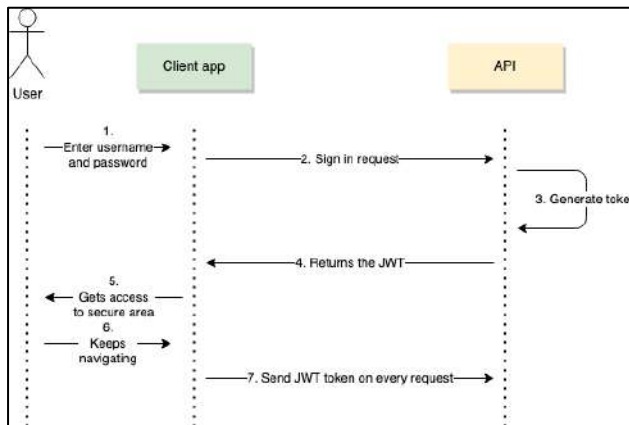


In Spring Security, a session is a mechanism used to maintain stateful authentication between a client (user) and the server. It allows a user to stay logged in across multiple requests without needing to re-authenticate every time. Like a train ticket which is valid for a limited time. In this case it is the same. Provides access for

the user after validating the login information for a limited time. But this exposes to attacks like CSRF (cross site request forgery). But the risk is low. But this session is stored in the server. This consumes

a lot of memory on the server or in a database. This may be a disadvantage when preparing the application for the production.

- JWT tokens.



This is same as sessions approach but instead of storing the session, the server generates a JWT (JSON Web token) and sends it to the client. Then stores in the local storage. But this token is not stored in the server. Then the token is used in every request. While the session is handled by the server, the token is handled by the client.

SESSION AUTHENTICATION

- First create a spring boot project with these dependencies:

```
<dependencies>
<!--START-->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.session</groupId>
<artifactId>spring-session-core</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-test</artifactId>
<scope>test</scope>
</dependency>
<!--END-->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<dependency>
<groupId>com.mysql</groupId>
<artifactId>mysql-connector-j</artifactId>
<scope>runtime</scope>
</dependency>
```

```

<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>3.1.1</version>
</dependency>
</dependencies>

```

- Note: some dependencies may not be available in spring initializer.
- First create the user entity (the table which includes the usernames and passwords).

```

package com.example.login.Entity;

import jakarta.persistence.*;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import java.sql.Time;
import java.util.Collection;
import java.util.List;

@Entity
@Table(name = "users")
public class users implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id")
    private int user_id;
    @Column(name = "username")
    private String username;
    @Column(name = "email")
    private String email;
    @Column(name = "password_hash")
    private String password_hash;
    @Column(name = "created_at")
    private Time created_at;

    public int getUser_id() {
        return user_id;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of();
    }

    @Override
    public String getPassword() {
        return password_hash;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {

```

```

        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

```

The security methods implemented in UserEntity (from UserDetails) are used by **Spring Security** to enforce authentication and authorization rules. They are automatically called when a user tries to log in or access protected resources.

```

    public String getUsername() {
        return username;
    }

    public String getEmail() {
        return email;
    }

    public String getPassword_hash() {
        return password_hash;
    }

    public Time getCreated_at() {
        return created_at;
    }

    public void setUser_id(int user_id) {
        this.user_id = user_id;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void setPassword_hash(String password_hash) {
        this.password_hash = password_hash;
    }

    public void setCreated_at(Time created_at) {
        this.created_at = created_at;
    }
}

```

- Main functions of these overridden functions:

Method	Purpose	Default	When to Change
<code>isAccountNonExpired()</code>	Checks if account is expired	<code>true</code>	If accounts expire (e.g., subscriptions)
<code>isAccountNonLocked()</code>	Checks if account is locked	<code>is_verified</code>	If accounts lock after failed attempts
<code>isCredentialsNonExpired()</code>	Checks if password is expired	<code>true</code>	If passwords expire after 90 days
<code>isEnabled()</code>	Checks if account is active	<code>true</code>	If accounts can be deactivated
<code>getAuthorities()</code>	Returns user roles	<code>ROLE_USER</code>	If users have multiple roles

- The method itself (`isAccountNonExpired()`, `isAccountNonLocked()`, etc.) **only returns a value**—it does not actively check or update the account status.

- However, **Spring Security automatically calls** these methods during authentication. The logic inside these methods **determines** whether the account should be considered expired, locked, or disabled.
- Then in the service class, override the loadUserByName function. This includes the process of retrieving the user's information using only username. The checking process will be done later.
- The returned values (true or false) **are used by Spring Security internally** to determine if login is allowed. You **can add custom logic** inside these methods (e.g., checking expiry date, login attempts).
- Flow in Spring Security:
 - User tries to log in → Spring loads UserDetails.
 - Spring calls isAccountNonExpired(), isAccountNonLocked(), etc.
 - If any method returns false, Spring blocks the login.
- Example for using custom logic:

@Override

```
public boolean isAccountNonExpired() {
    return expiryDate.isAfter(LocalDate.now());
}
```

- If you want to modify authentication behavior, check the **UserEntity** class.
- If you want to change how users are loaded, modify **UserDetailsService** in the **Service Layer**.

```
package com.example.login.Service;

import com.example.login.Entity.users;
import com.example.login.Repo.usersRepo;
import com.example.login.Requests.logIn_request;
import com.example.login.Responses.userResponse;
import jakarta.transaction.Transactional;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.*;

@RequiredArgsConstructor
public class login_service implements UserDetailsService {
    @Autowired
    private ModelMapper mapper;
    @Autowired
    private usersRepo repo;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        List<users> user = repo.findAll();
        String[] arr1 = new String[2];
        user.forEach((p) -> {
            if (p.getUsername().equals(username)) {
                arr1[0] = p.getUsername();
                arr1[1] = p.getPassword_hash();
            }
        });
    }
}
```

This includes the filtering the user details according to the username

```

    });
    return new User(
        arr1[0],
        arr1[1],
        new ArrayList<>()
    );
}
}

```

- The above class or the service will be called by the AuthenticationProvider. Not by the controller. This returns the required details to the AuthenticationProvider.
- Then create the AuthenticationProvider as follows. Create a new java class file in the service directory.

```

package com.example.login.Service;

import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Component;

import java.util.ArrayList;

@Component
//Marks this class as a Spring-managed component.
//Enables Spring to discover it during component scanning and include it in the application
context.
@RequiredArgsConstructor
//From Lombok, it generates a constructor for all final fields, simplifying dependency
injection.
public class UserAuthenticationProvider implements AuthenticationProvider {
    // This class implements the AuthenticationProvider interface, which defines custom logic for
    authentication.
    //Spring Security delegates the authentication process to instances of AuthenticationProvider.

    //Core method where the actual authentication logic is implemented.
    //Accepts an Authentication object that contains the user's credentials.
    @Autowired
    private UserDetailsService userDetailsService;
    /*The UserDetailsService is a Spring Security interface typically backed by a database or
    another persistent store.
    The loadUserByUsername() method retrieves a UserDetails object, which contains the user's
    credentials and authorities.*/
    @Autowired
    private PasswordEncoder passwordEncoder; // Used to securely validate passwords.
    @Override
    public Authentication authenticate(Authentication authentication) throws
    AuthenticationException {
        String username = authentication.getName();
        String password = authentication.getCredentials().toString();
        /*The authentication object contains the user's input credentials (provided during login).
        getName() retrieves the username */
        /*getCredentials() retrieves the password.*/
        UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        /*The userDetailsService fetches the user record from the data source (e.g., database) based on
        the username. Calls the method in the service class*/
        if(!password.equals(userDetails.getPassword())){
            throw new BadCredentialsException("Invalid username or password");
        }
        /*check weather the details are correct*/
        return new UsernamePasswordAuthenticationToken(username,password,new ArrayList<>());
        //return the token to the controller. This token represents a successful authentication and will
        be stored in the SecurityContext.
    }
}

```



```

@Override
public boolean supports(Class<?> authentication) {
    return authentication.equals(UsernamePasswordAuthenticationToken.class);
}
/*Ensures that this AuthenticationProvider can process UsernamePasswordAuthenticationToken
objects.*/
}

```

- It's not required to explicitly call this class or its authenticate function in the controller. Here's why:
 - How Authentication Works in Spring Security
 - The `UserAuthenticationProvider` class is automatically invoked by Spring Security as part of the authentication process.
 - When a controller receives a request that needs authentication (like a login endpoint), the `AuthenticationManager` is responsible for handling the authentication process. It delegates the work to the appropriate `AuthenticationProvider` (in this case, your custom `UserAuthenticationProvider`).
- Then create the controller as follows:

```

package com.example.login.Controller;

import com.example.login.Entity.users;
import com.example.login.Requests.logIn_request;
import com.example.login.Responses.userResponse;
import com.example.login.Service.login_service;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContext;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.context.SecurityContextHolderStrategy;
import org.springframework.security.web.context.HttpSessionSecurityContextRepository;
import org.springframework.security.web.context.SecurityContextRepository;
import org.springframework.web.bind.annotation.*;

import java.util.Map;

@RestController
@CrossOrigin(origins = "http://localhost:5173")
@RequiredArgsConstructor
@RequestMapping(path = "/api/auth")
public class UserController {
    @Autowired
    private login_service logService;
    // private final AuthService authService;
    private ModelMapper modelMapper;
    private SecurityContextRepository securityContextRepository = new
HttpSessionSecurityContextRepository();
    @Autowired
    private AuthenticationManager authenticationManager;
    private SecurityContextHolderStrategy securityContextHolderStrategy =
SecurityContextHolder.getContextHolderStrategy();
    @PostMapping("/login")
    public ResponseEntity<?> login(
        @RequestBody logIn_request loginRequest,
        HttpServletRequest request,
        HttpServletResponse response
    ) {
        try{

            UsernamePasswordAuthenticationToken token = UsernamePasswordAuthenticationToken
.unauthenticated(

```

```

        loginRequest.getUsername(), loginRequest.getPassword_hash()
    );
}

/**
Encapsulates the user's phone number and password into an unauthenticated token.
This token will be used to attempt authentication.
*/

    Authentication authentication = authenticationManager.authenticate(token);

/**
The AuthenticationManager authenticates the user credentials.
This involves calling AuthenticationProvider (e.g., DaoAuthenticationProvider), which validates
the credentials against a user store (like a database).this class (UserAuthenticationProvider)
implements the AuthenticationProvider interface, and its methods are invoked indirectly by
Spring Security as part of the authentication process. Let's break it down: The
AuthenticationManager is called in the login() method */

    SecurityContext context = securityContextHolderStrategy.createEmptyContext();
/**create the context object */
/** The Security Context is a container object that:
    Stores information about the currently authenticated user (principal).
    Includes their credentials (if available) and authorities (roles or permissions).
    Provides a consistent way for the application to access the authenticated user's
    identity.*/

    context.setAuthentication(authentication); //set context application from
authentication
/** The returned UsernamePasswordAuthenticationToken from the UserAuthenticationProvider is
stored in the SecurityContext in this code.
*/

    securityContextHolderStrategy.setContext(context);

    securityContextRepository.saveContext(context, request, response); //save the auth
context

    return ResponseEntity.ok(Map.of(
        "message", "Login successful",
        "user", authentication.getPrincipal()
    ));
} catch (Exception ee) {
    return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(Map.of(
        "message", ee.toString()
    ));
}
}
}

```

- Explanation:

1. **Encapsulation of User Input into an Unauthenticated Token:**

- The UsernamePasswordAuthenticationToken class represents the user's credentials (e.g., phone number and password).
- **Unauthenticated Token:**

What is the UsernamePasswordAuthenticationToken?

- It's an implementation of the Authentication interface, which represents the credentials (and potentially the authorities) of a principal (user) in Spring Security.
- Unauthenticated Token: Initially, the token only contains the user's input credentials (e.g., username and password). This means that it hasn't been verified or authenticated yet.
- Authenticated Token: After authentication is successful, the token will also include details about the authenticated user and their authorities (roles/permissions).
- Creates a token in an unauthenticated state. At this point, the application only knows what the user has provided but hasn't validated it yet.

- Created with user-provided credentials.
- Represents the initial state before any validation.
- Example:

```
java
CopyEdit
UsernamePasswordAuthenticationToken token =
    UsernamePasswordAuthenticationToken
        .unauthenticated(loginRequest.getPhone_number(),
            loginRequest.getPassword());
```

- The token holds:
 - Principal: User's phone number (or username).
 - Credentials: User's password.
 - Authorities: null (not yet populated).

2. Authentication Process Using the Token:

- The unauthenticated token is passed to the `AuthenticationManager` for validation.

```
java
CopyEdit
Authentication authentication = authenticationManager.authenticate(token);
```

- Steps:
 - **Delegation to Authentication Providers:**
 - The `AuthenticationManager` delegates validation to one or more `AuthenticationProviders` (e.g., `DaoAuthenticationProvider`).
 - **Credential Validation:**
 - User details (username, hashed password) are fetched from the database via `UserDetailsService`.
 - The input password is compared with the stored hashed password using a `PasswordEncoder`.

3. Result of Authentication:

- **Successful Authentication:**
 - A new, authenticated token is created containing:
 - User's identity (e.g., phone number).
 - Granted authorities (e.g., roles like `ROLE_USER` or `ROLE_ADMIN`).
 - Credentials are cleared for security.
 - Example:

```
java
CopyEdit
UsernamePasswordAuthenticationToken(
    principal: "user@example.com",
    credentials: null,
    authorities: [ROLE_USER, ROLE_ADMIN]
);
```

- **Failed Authentication:**
 - If validation fails, an exception (e.g., `BadCredentialsException`) is thrown.

4. Why Use an Unauthenticated Token First?:

- **Separation of Concerns:**
 - The unauthenticated token wraps user-provided data without validation.
 - Validation and authentication logic are handled centrally by the `AuthenticationManager` and its providers.
- **Security:**
 - The password is cleared from the token after successful authentication to reduce exposure.

- **Extensibility:**
 - The modular design allows for easy customization (e.g., integrating new authentication providers).

Summary of Key Concepts:

- **Unauthenticated Token:**
 - Contains raw user credentials before validation.
- **Authenticated Token:**
 - Created after successful authentication, includes user details and roles.
- **AuthenticationManager:**
 - Validates credentials via configured `AuthenticationProviders`.
- **Benefits:**
 - Centralized and reusable authentication flow.
 - Clear distinction between pre-validation and post-validation states.
 - Enhanced security and maintainability.

```
securityContextHolderStrategy.setContext(context);
```

1. **SecurityContext:**
 - **What is it?**
 - An object that stores security-related information for the current request or session.
 - Includes details like the authenticated user's credentials and authorities.
2. **SecurityContextHolderStrategy:**
 - **What is it?**
 - Defines how the `SecurityContext` is stored and retrieved.
 - Allows different storage mechanisms:
 - **Thread-Local Storage (Default):** The context is tied to the current thread.
 - **Inheritable Thread-Local Storage:** For parent-child threads.
 - **Custom Strategies:** HTTP sessions, shared memory, or external storage for distributed systems.

3. Thread-Local Storage (Default):

- **Behavior:**
 - Spring Security uses a `ThreadLocal` variable to store the `SecurityContext`:

```
java
CopyEdit
private static final ThreadLocal<SecurityContext> contextHolder = new
ThreadLocal<>();
```

- The context is set during a request and cleared after the request completes.

- **Example Scenario:**

- A user is authenticated.
- The `SecurityContext` is stored in thread-local storage for that request.
- After the request, the context is cleared to prevent cross-thread contamination.

- **Custom Strategy:**

- If configured, a custom `SecurityContextHolderStrategy` can store the context in shared data structures, HTTP sessions, or external storage.

4. Setting the Context for a Request:

- **How it works:**
 - The `setContext(context)` method associates a populated `SecurityContext` with the current execution thread or request:

```
java
CopyEdit
```

```
SecurityContext context =
securityContextHolderStrategy.createEmptyContext();
context.setAuthentication(authentication);
securityContextHolderStrategy.setContext(context);
```

- **Why this is important:**

- Ensures authentication state persists throughout the request lifecycle.
- Subsequent access to `SecurityContextHolder.getContext()` retrieves the same context.

5. Subsequent Access:

- The `SecurityContext` becomes globally accessible within the application for the current thread or request.
- Example of retrieving it:

```
java
CopyEdit
Authentication auth =
SecurityContextHolder.getContext().getAuthentication();
```

6. What Happens if `setContext(context)` is Omitted?:

- The `SecurityContext` will not be set.
- Calls to `SecurityContextHolder.getContext()` will return an empty or default context.
- **Impact:**
 - The application won't recognize the user as authenticated.
 - Authorization mechanisms (e.g., role checks) will fail.

7. Behind the Scenes of `setContext()`:

- **Thread-Local Association (Default):**
 - With `ThreadLocalSecurityContextHolderStrategy`, the context is stored in a `ThreadLocal` variable:

```
java
CopyEdit
public void setContext(SecurityContext context) {
    contextHolder.set(context);
}
```

- **Custom Strategy Implementation:**

- If a custom strategy is provided, the context might be stored differently (e.g., HTTP session or shared memory).

8. Why This Is Important:

- **Authentication Persistence:**
 - Maintains the authentication state throughout the lifecycle of a request.
- **Authorization:**
 - Security mechanisms (e.g., `@PreAuthorize`) depend on the `SecurityContext` for role/permission checks.
- **Thread Safety:**
 - Isolates the context to the current thread, preventing cross-thread contamination in multi-threaded environments.

Example Flow

1. User Logs In:

- The user submits credentials (e.g., username and password).
- The `AuthenticationManager` authenticates the user and returns an `Authentication` object.

2. Security Context Is Set:

- The `Authentication` object is stored in the `SecurityContext`:

```
java
CopyEdit
SecurityContext context =
securityContextHolderStrategy.createEmptyContext();
context.setAuthentication(authentication);
securityContextHolderStrategy.setContext(context);
```

3. Access Control:

- The `SecurityContext` is used to check if the user has sufficient permissions to access resources:

```
java
CopyEdit
Authentication auth =
SecurityContextHolder.getContext().getAuthentication();
```

4. Context Cleanup:

- After the request is processed, Spring Security clears the `SecurityContext` to prevent leakage between threads.

```
securityContextRepository.saveContext(context, request, response);
```

Key Components

1. SecurityContextRepository:

- Defines how the `SecurityContext` should be stored and retrieved.
- The implementation determines the storage mechanism.
- Common implementations include:
 - **HttpSessionSecurityContextRepository (default for session-based authentication):**
 - Stores the `SecurityContext` in the HTTP session.
 - **RequestAttributeSecurityContextRepository:**
 - Stores the `SecurityContext` as a request attribute (for single-request lifetimes).
 - **Custom implementations:**
 - Used for stateless authentication approaches, such as token-based systems (e.g., JWT).

2. saveContext(SecurityContext context, HttpServletRequest request, HttpServletResponse response):

- Saves the provided `SecurityContext` (containing the `Authentication` object) into the repository.
- Associates the saved context with the incoming `HttpServletRequest` and `HttpServletResponse`.

3. SecurityContext:

- Contains the `Authentication` details for the authenticated user.

What Does This Line Do?

```
java
CopyEdit
securityContextRepository.saveContext(context, request, response);
```

1. Retrieves the Current Authentication State:

- The context contains the Authentication object, populated earlier:

```
java
CopyEdit
SecurityContext context =
securityContextHolderStrategy.createEmptyContext();
context.setAuthentication(authentication);
```

2. Persists the SecurityContext:

- The SecurityContext is stored based on the SecurityContextRepository implementation:
 - Example with HttpSessionSecurityContextRepository:

```
java
CopyEdit
HttpSession session = request.getSession();
session.setAttribute(
SPRING_SECURITY_CONTEXT_KEY, context);
```

3. Links to the Request/Response:

- The HttpServletRequest and HttpServletResponse are used to manage the SecurityContext lifecycle:
 - The repository may add session cookies to the response or extract information from the request.

4. Prepares the Context for Future Requests:

- Ensures the authentication state persists across multiple HTTP requests.
- On subsequent requests, the repository retrieves the context:

```
java
CopyEdit
SecurityContext context = securityContextRepository.loadContext(request);
```

Why Is This Important?

1. Session-Based Authentication:

- Persists the user's login state after the initial login.
- Without saveContext, the user would need to re-authenticate on every request.

2. Decoupling Storage Mechanisms:

- Allows flexible storage mechanisms (e.g., session-based, request-based, token-based) without altering authentication logic.

3. Integration with Security Filters:

- Many Spring Security filters rely on the saved context to apply security checks (e.g., SecurityContextHolderAwareRequestFilter).
-

Flow of saveContext

1. Authentication:

- User logs in, and credentials are validated.
- A SecurityContext is created, and the Authentication object is set.

2. Context Saved:

- The SecurityContext is saved to the repository:

```
java
```



```
CopyEdit
securityContextRepository.saveContext(context, request, response);
```

3. Future Requests:

- For subsequent HTTP requests, the `SecurityContextRepository` retrieves the saved context:

```
java
CopyEdit
SecurityContext context = securityContextRepository.loadContext(request);
```

4. Authorization:

- Spring Security uses the loaded `SecurityContext` to determine access to resources.

Example: `HttpSessionSecurityContextRepository`

1. Session Created:

- If no session exists, one is created:

```
java
CopyEdit
HttpSession session = request.getSession(true);
```

2. Context Saved:

- The `SecurityContext` is stored in the session:

```
java
CopyEdit
session.setAttribute(
    SPRING_SECURITY_CONTEXT_KEY, context);
```

3. Cookie Added (Optional):

- If configured, a session cookie is added to the response to track the session.

4. Persistent Authentication:

- On subsequent requests, the saved context is loaded and used.

Custom Implementations

- For stateless systems (e.g., JWT):
 - Implement a custom `SecurityContextRepository`.
 - In the `saveContext` method, serialize the `Authentication` object into a JWT and add it to the response headers:

```
java
CopyEdit
response.setHeader("Authorization", "Bearer " + jwtToken);
```

What Happens Without `saveContext`?

- The `SecurityContext` will not persist beyond the current request.
- On subsequent requests:
 - No authentication information will be available.

- The user will need to log in again.

SUMMARY

1. User Input:

- The user submits a username and password via the login API.

2. AuthenticationManager:

- Receives the credentials wrapped in a UsernamePasswordAuthenticationToken.
- Delegates authentication to the appropriate AuthenticationProvider (in this case, UserAuthenticationProvider).

3. Custom Authentication Logic:

- The authenticate() method checks the credentials against the database.
- If successful, an authenticated UsernamePasswordAuthenticationToken is returned.
- If authentication fails, an exception is thrown.

4. Security Context:

- Spring Security stores the authenticated token in the SecurityContext for the current session or request.

Source: [How to create session authentication using spring boot? / by Ziaurrahman / Medium](#)

IMPORTANT

- When the authentication request is sent and when the authentication request is successful, the session ID will send to the client automatically with the response.
- Also, the browser or the web app uses that session ID automatically for the next requests (if credentials: "include" is enabled in JS).
- The endpoints of the API should be protected from unauthorized requests. Otherwise, the unauthorized requests can access the app API endpoints. Because the API or the server checks the arriving request is authorized.
- Also, it is required to send the session information (or the token) through the next requests.
- Security configuration (related to google authentication):

```
package com.example.LoginUsingGoogle.Configuration;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.annotation.web.configurers.SessionManagementConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.logout.HeaderWriterLogoutHandler;
import org.springframework.security.web.context.HttpSessionSecurityContextRepository;
import org.springframework.security.web.context.SecurityContextRepository;
import org.springframework.security.web.header.writers.ClearSiteDataHeaderWriter;
```

```

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration {
    private AuthenticationProvider authenticationProvider;
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        SecurityContextRepository securityContextRepository = new
HttpSessionSecurityContextRepository();
        http
            .csrf(AbstractHttpConfigurer::disable) // Disable CSRF for testing purposes
(enable it for production)
            .authorizeHttpRequests(authz -> authz
                .requestMatchers("/login", "/", "/oauth2/**",
"/login/oauth2/**").permitAll()
                .requestMatchers("/section2").authenticated()
                .anyRequest().authenticated() // Secure other endpoints
            )
            .oauth2Login(withDefaults())
            .oauth2Login(oauth2->oauth2.defaultSuccessUrl("/login-success",true))
            .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)) // No sessions for stateless
APIs
            .formLogin(AbstractHttpConfigurer::disable); // Disable default form login

        return http.build();
        /*
        * .oauth2Login(oauth2 -> oauth2
        .defaultSuccessUrl("/login-success") // Redirect here after OAuth2 success
        */
    }
}

```

- Configuration part II:

```

package com.example.LoginUsingGoogle.Configuration;

import com.example.LoginUsingGoogle.Service.FetchPrimaryDetails2;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.datatype.jsr310.JavaTimeModule;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.session.HttpSessionEventPublisher;

@Configuration
public class SupplierConfiguration {
    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }
    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http) throws Exception {
        return http.getSharedObject(AuthenticationManagerBuilder.class).build();
    }
    // @Bean
    // public ObjectMapper objectMapper() {
    //     return new ObjectMapper();
    // }
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}

```

```

@Bean
public FetchPrimaryDetails2 loginBean() {
    return new FetchPrimaryDetails2();
}
@Bean
public ModelMapper modelMapperBean() {
    return new ModelMapper();
}
@Bean
@Primary
public ObjectMapper objectMapper(){
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.registerModule(new JavaTimeModule());
    return objectMapper;
}
}

```

CONFIGURING THE PROJECT FOR AUTHENTICATED ACCESS

- First create another endpoint for the security (for authenticated requests only).

```

@GetMapping("/secTest")
@PreAuthorize("isAuthenticated()")
public String getMessage() {

    return "PROTECTED ACCESSED";
}

```

- “@PreAuthorize(“isAuthenticated()”)” checks the request is authenticated (we can access this after login process).
- Modify the security config file (as we created before) for the checking of authentication.

```

package com.example.login.Configuration;

import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.annotation.web.configurers.SessionManagementConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.logout.HeaderWriterLogoutHandler;
import org.springframework.security.web.context.HttpSessionSecurityContextRepository;
import org.springframework.security.web.context.SecurityContextRepository;
import org.springframework.security.web.header.writers.ClearSiteDataHeaderWriter;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration {
    private AuthenticationProvider authenticationProvider;
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        SecurityContextRepository securityContextRepository = new
HttpSessionSecurityContextRepository();
        http
            .csrf(AbstractHttpConfigurer::disable) // Disable CSRF for testing purposes
            (enable it for production)
            .authorizeHttpRequests(authz -> authz

```

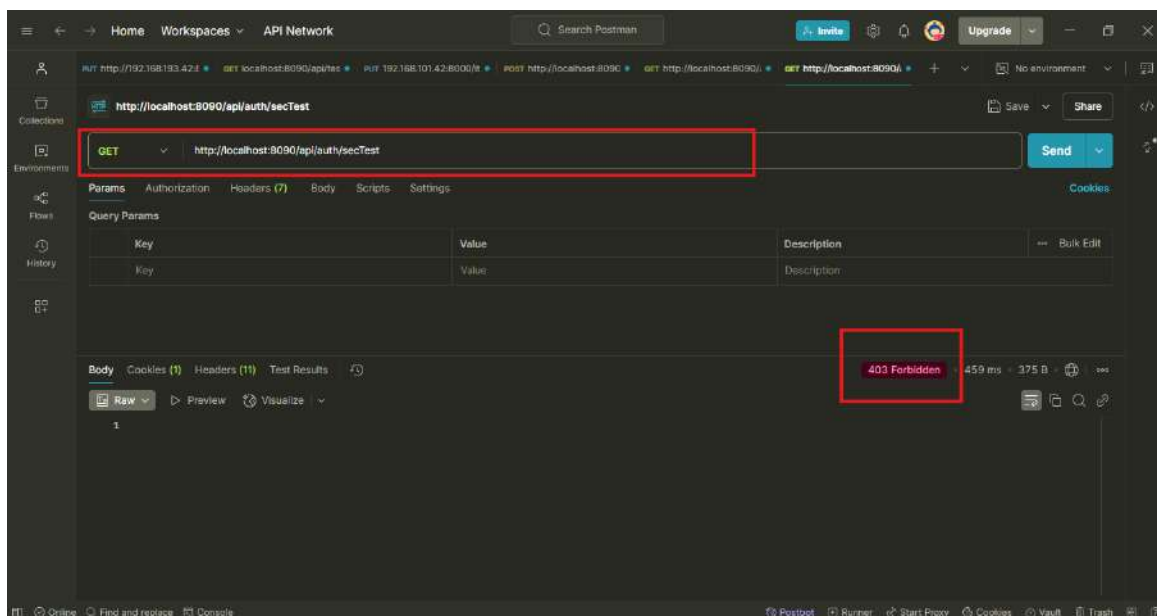
```

        .requestMatchers("/api/auth/login").permitAll()//allow all requests for
login
        .requestMatchers("/api/auth/secTest").authenticated()//allow for
authenticated requests to access this API endpoint.
        .anyRequest().authenticated() // Secure other endpoints
    )
    .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)) //This means Spring Security
will not store authentication in a session.
.formLogin(AbstractHttpConfigurer::disable); // Disable default form login

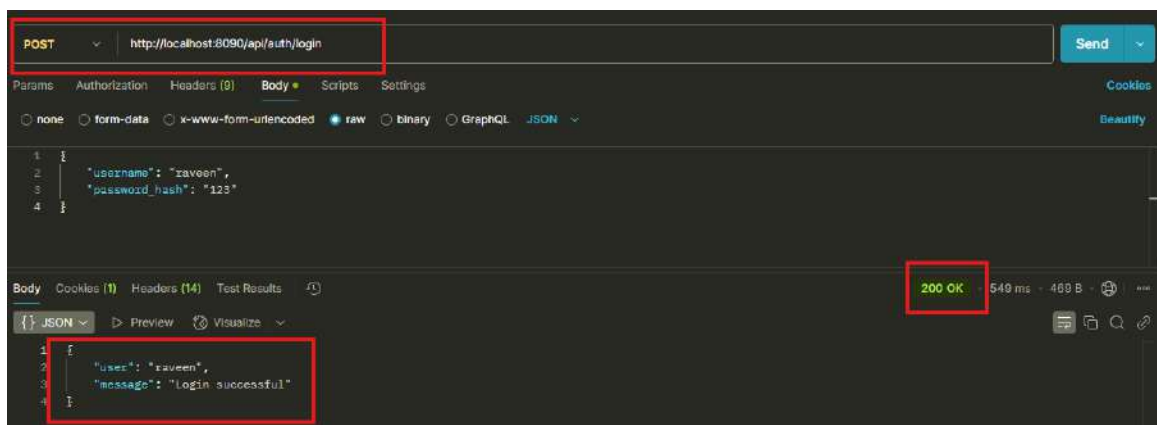
    return http.build();
}
}

```

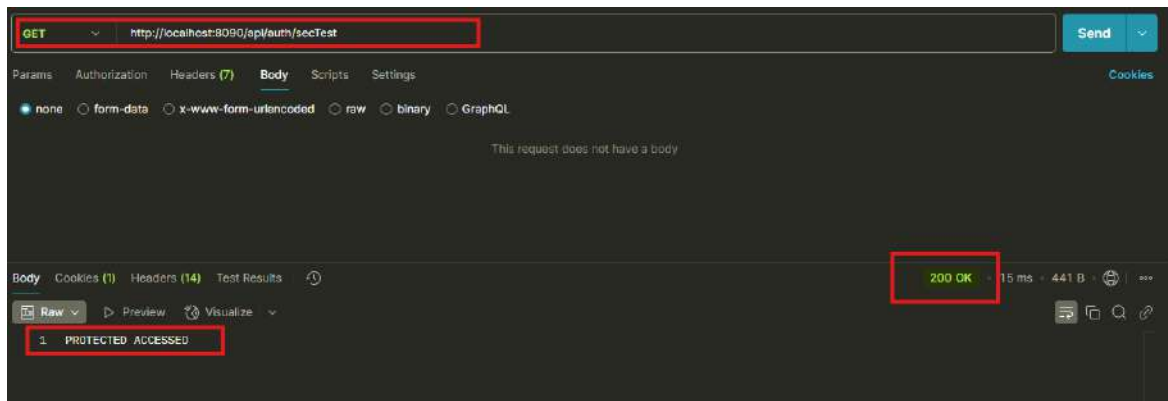
- Output →
 - We can't access the “secTest” endpoint without authentication.



- But when we authenticated,



- The endpoint “secTest” can be accessed.



CONNECTING SPRING BOOT TO MONGO DB

- Add this dependency:

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

- Then refresh the pom.xml.
- Add these properties to the application.properties.

```
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=travel-project-temp-db
```

- Create the entity class as follows:

```
package com.example.createAccountService.Entity;

import jakarta.persistence.Column;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.time.LocalDateTime;
@Document("usersMongo")
public class usersMongo {
    @Id
    private String id;
    private String username;
    private String email;
    private String password_hash;
    private LocalDateTime created_at;
    private String verificationCode;

    public usersMongo(String username, String email, String password_hash, LocalDateTime created_at,
String verificationCode) {
        this.username = username;
        this.email = email;
        this.password_hash = password_hash;
        this.created_at = created_at;
        this.verificationCode = verificationCode;
    }

    public String getId() {
        return id;
    }

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

```

public void setVerificationCode(String verificationCode) {
    this.verificationCode = verificationCode;
}

public String getVerificationCode() {
    return verificationCode;
}

public String getUsername() {
    return username;
}

public String getEmail() {
    return email;
}

public String getPassword_hash() {
    return password_hash;
}

public LocalTime getCreated_at() {
    return created_at;
}

public void setUsername(String username) {
    this.username = username;
}

public void setEmail(String email) {
    this.email = email;
}

public void setPassword_hash(String password_hash) {
    this.password_hash = password_hash;
}

public void setCreated_at(LocalTime created_at) {
    this.created_at = created_at;
}
}

```

- Create the repository as follows:

```

package com.example.createAccountService.Repo;

import com.example.createAccountService.Entity.users;
import com.example.createAccountService.Entity.usersMongo;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface usersRepoMongo extends MongoRepository <usersMongo,String>{

}

```

- Create the request class (optional):

```

package com.example.createAccountService.Requests;
import java.time.LocalTime;

public class usersMongoRequest {
    private String username;
    private String email;
    private String password_hash;
    private LocalTime created_at;

    public usersMongoRequest(String username, String email, String password_hash, LocalTime
created_at) {
        this.username = username;
        this.email = email;
        this.password_hash = password_hash;
        this.created_at = created_at;
    }

    public String getUsername() {
        return username;
    }
}

```



```

    }

    public String getEmail() {
        return email;
    }

    public String getPassword_hash() {
        return password_hash;
    }

    public LocalTime getCreated_at() {
        return created_at;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void setPassword_hash(String password_hash) {
        this.password_hash = password_hash;
    }

    public void setCreated_at(LocalTime created_at) {
        this.created_at = created_at;
    }
}

```

- Create a service class and perform the CRUD operations:

```

package com.example.createAccountService.Service;

import com.example.createAccountService.Entity.Credentials;
import com.example.createAccountService.Entity.user_profilesMongo;
import com.example.createAccountService.Entity.usersMongo;
import com.example.createAccountService.Repo.user_profilesRepoMongo;
import com.example.createAccountService.Repo.usersRepoMongo;
import com.example.createAccountService.Responses.userMongoResponse;
import com.mailjet.client.ClientOptions;
import com.mailjet.client.MailjetClient;
import com.mailjet.client.errors.MailjetException;
import com.mailjet.client.transactional.*;
import com.mailjet.client.transactional.response.SendEmailsResponse;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;

import java.security.SecureRandom;

@RequiredArgsConstructor
public class insertTempData {
    @Autowired
    private ModelMapper mapper;
    @Autowired
    private usersRepoMongo urm1;
    @Autowired
    private user_profilesRepoMongo upm1;
    public userMongoResponse saveUserTemp(usersMongo user, user_profilesMongo user_profile)
    throws MailjetException {
        //save data to mongo DB database
        usersMongo savedData = urm1.save(user);
        user_profile.setUser_id(savedData.getId());
        user_profilesMongo savedData2 = upm1.save(user_profile);
        //create random number
        SecureRandom secureRandom = new SecureRandom();
        int randomNumber = secureRandom.nextInt(100);
        //update data
        user.setVerificationCode(String.valueOf(randomNumber));
        urm1.save(user); //update the verification code
        userMongoResponse ul = new

```

```

userMongoResponse(savedData.getId(),savedData.getUsername(),savedData.getEmail(),savedData.getPa
ssword_hash(),savedData.getCreated_at(),user.getVerificationCode());
    return ul;
}
}

```

- Also create the config file (if @Service is not used):

```

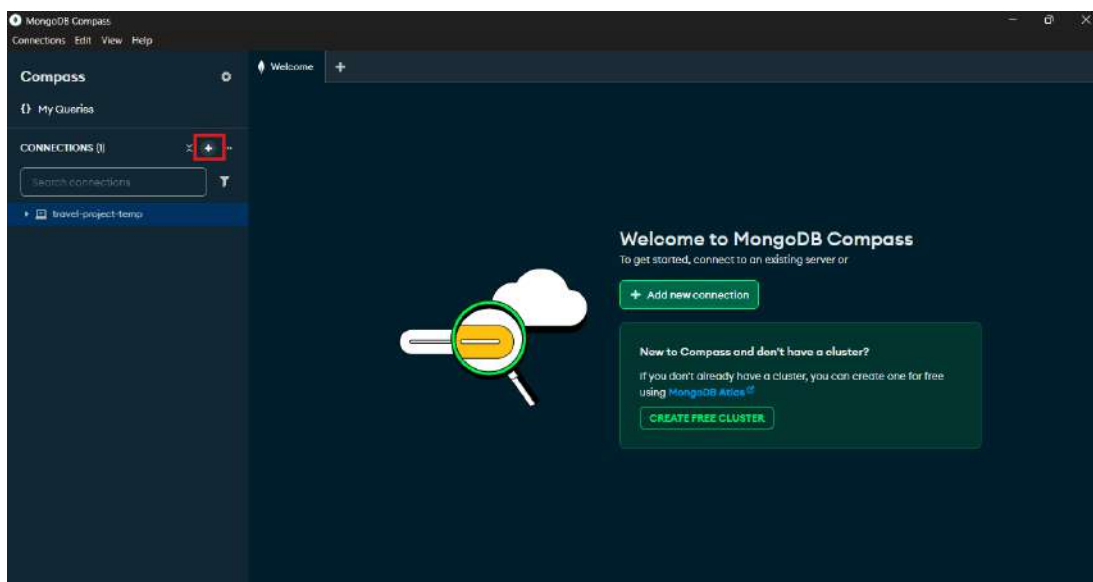
package com.example.createAccountService.Configuration;

import com.example.createAccountService.Service.checkVerification;
import com.example.createAccountService.Service.insertTempData;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

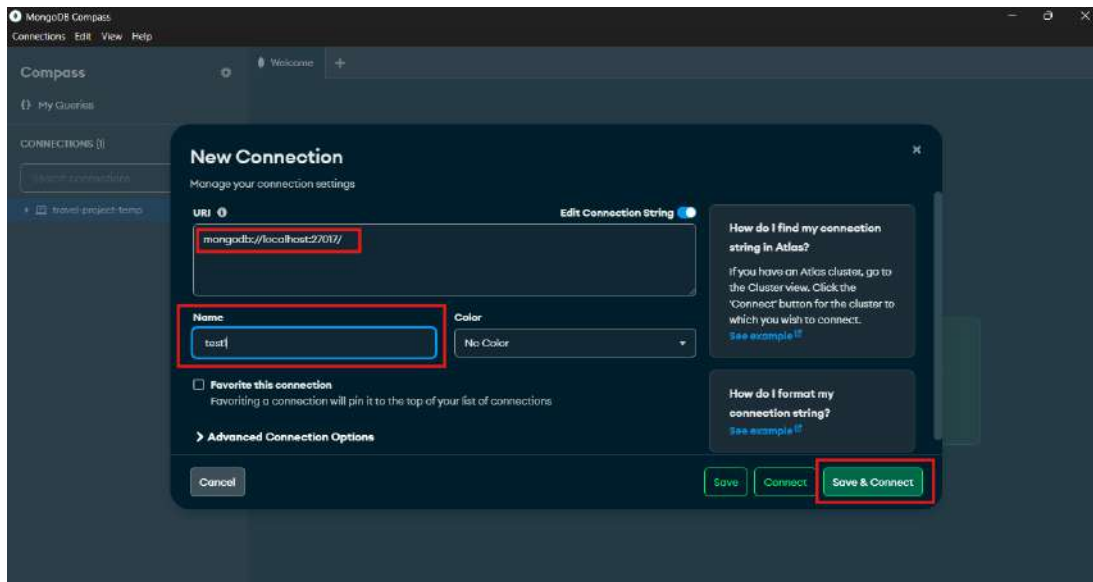
@Configuration
public class userConfig {
    @Bean
    public insertTempData logInBean() {
        return new insertTempData();
    }
    @Bean
    public ModelMapper modelMapperBean() {
        return new ModelMapper();
    }
}

```

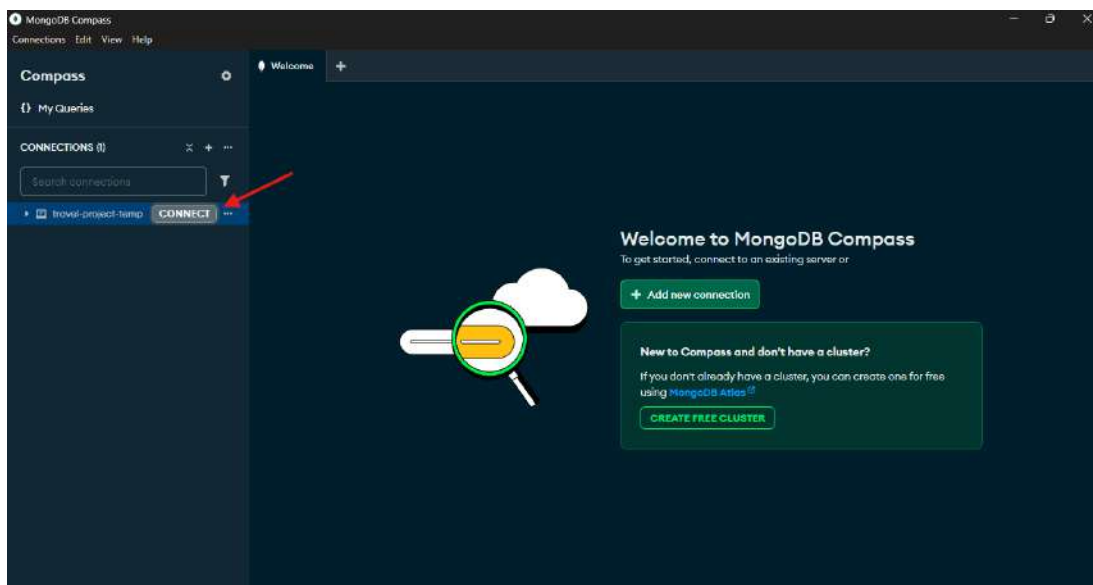
- Create a new mongo DB connection:



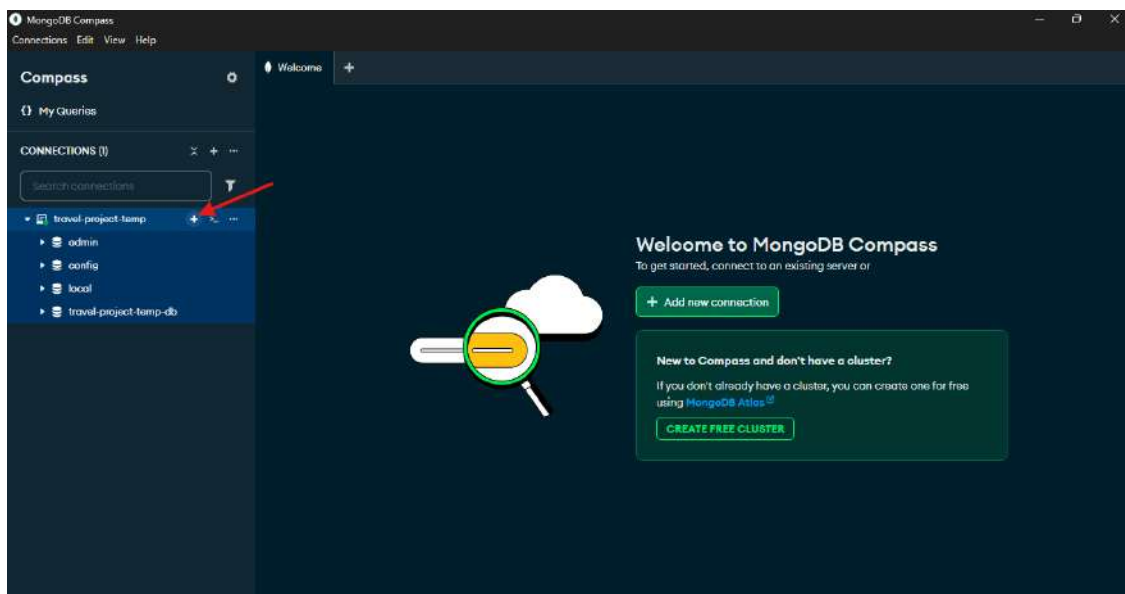
- Add the connection name and keep the settings default.



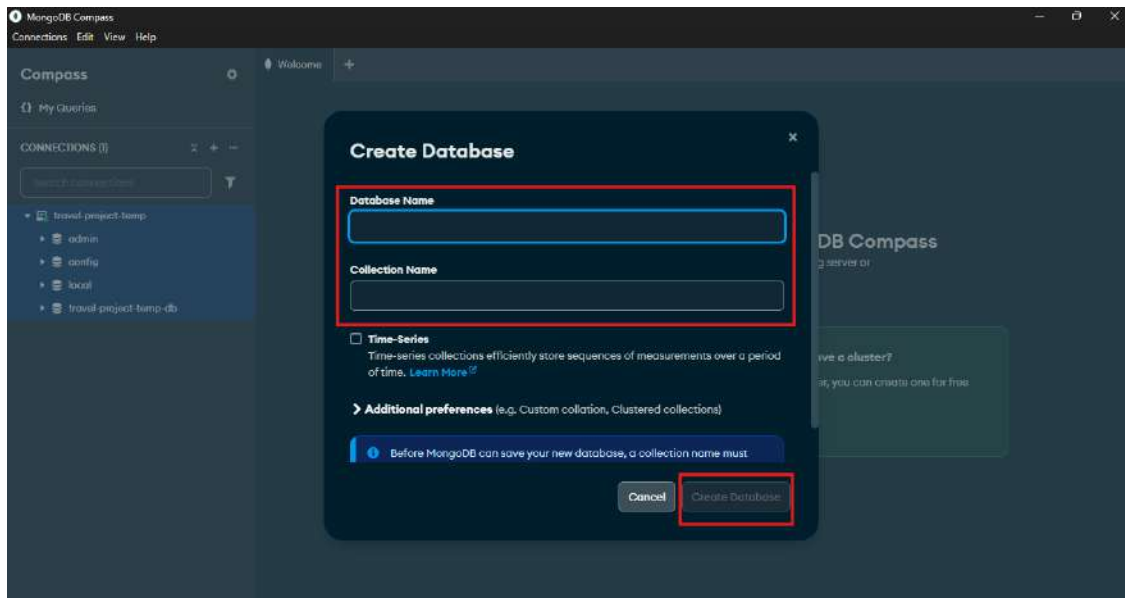
- Connect to the database:



- Create a database as follows:



- Add the database name and the collection name. then create the database.



- Also you can create the collections in the database but the collections will create automatically when using the code.

CONNECTING MAILJET API

- Create a mailjet account and obtain the API and the secret key.
- Add this dependency to the POM.xml.

```
<dependency>
  <groupId>com.mailjet</groupId>
  <artifactId>mailjet-client</artifactId>
  <version>5.2.5</version>
</dependency>
```

- Add these details to the application.properties.

```
#mailjet credentials
mailjet.api.key=your API KEY
mailjet.api.secret=your API secret key
```

- Create a class file to create the connection object (singleton pattern). This will output only one connection object.

```
package com.example.createAccountService.Entity;

import com.mailjet.client.ClientOptions;
import com.mailjet.client.MailjetClient;
import lombok.Getter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Credentials {
    @Value("${mailjet.api.key}") // Add these properties
    private String mailjetApiKey;

    @Value("${mailjet.api.secret}")
    private String mailjetApiSecret;
    private MailjetClient client = null;
    public MailjetClient getMailConnection() {
        if(client == null) {
            ClientOptions options = ClientOptions.builder()
```

```

        .apiKey(mailjetApiKey)
        .apiSecretKey(mailjetApiSecret)
        .build();
    MailjetClient client = new MailjetClient(options);
    return client;
}
return client;
}
}

```

- Send the e-mail with the HTML content as follows:

```

package com.example.createAccountService.Service;

import com.example.createAccountService.Entity.Credentials;
import com.example.createAccountService.Entity.user_profilesMongo;
import com.example.createAccountService.Entity.usersMongo;
import com.example.createAccountService.Repo.user_profilesRepoMongo;
import com.example.createAccountService.Repo.usersRepoMongo;
import com.example.createAccountService.Responses.userMongoResponse;
import com.mailjet.client.ClientOptions;
import com.mailjet.client.MailjetClient;
import com.mailjet.client.errors.MailjetException;
import com.mailjet.client.transactional.*;
import com.mailjet.client.transactional.response.SendEmailsResponse;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;

import java.security.SecureRandom;

@RequiredArgsConstructor
public class insertTempData {
    @Autowired
    private ModelMapper mapper;
    @Autowired
    private usersRepoMongo urml;
    @Autowired
    private user_profilesRepoMongo upml;
    @Autowired
    private Credentials crel;
    // @Value("${mailjet.api.key}") // Add these properties
    // private String mailjetApiKey;
    //
    // @Value("${mailjet.api.secret}")
    // private String mailjetApiSecret;
    public userMongoResponse saveUserTemp(usersMongo user, user_profilesMongo user_profile)
    throws MailjetException {
        usersMongo savedData = urml.save(user);
        user_profile.setUser_id(savedData.getId());
        user_profilesMongo savedData2 = upml.save(user_profile);
        //create random number
        SecureRandom secureRandom = new SecureRandom();
        int randomNumber = secureRandom.nextInt(100);
        MailjetClient client = crel.getMailConnection();
        TransactionalEmail message1 = TransactionalEmail
            .builder()
            .to(new SendContact(savedData.getEmail(), "stanislav"))
            .from(new SendContact("saliyaautocare52@gmail.com", "Mailjet integration test"))
            .htmlPart("<h1>Your verification code : " + randomNumber + "</h1>")
            .subject("This is the subject")
            .trackOpens(TrackOpens.ENABLED)
            .header("test-header-key", "test-value")
            .customID("custom-id-value")
            .build();
        SendEmailsRequest request = SendEmailsRequest
            .builder()
            .message(message1) // you can add up to 50 messages per request
            .build();
        SendEmailsResponse response = request.sendWith(client);
        //update data
        user.setVerificationCode(String.valueOf(randomNumber));
    }
}

```

```

        url1.save(user); //update the verification code
        userMongoResponse u1 = new
userMongoResponse(savedData.getId(),savedData.getUsername(),savedData.getEmail(),savedData.getPa
ssword_hash(),savedData.getCreated_at(),user.getVerificationCode());
        return u1;
    }
}

```

- Also, add a mailjet exception to the controller:

```

package com.example.createAccountService.Controller;

import com.example.createAccountService.Entity.user_profilesMongo;
import com.example.createAccountService.Entity.usersMongo;
import com.example.createAccountService.Requests.UserTempRequest;
import com.example.createAccountService.Requests.user_profilesMongoRequest;
import com.example.createAccountService.Requests.usersMongoRequest;
import com.example.createAccountService.Responses.userMongoResponse;
import com.example.createAccountService.Service.checkVerification;
import com.example.createAccountService.Service.insertTempData;
import com.mailjet.client.errors.MailjetException;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@CrossOrigin(origins = "http://localhost:5173")
@RequiredArgsConstructor
@RequestMapping(path = "/api/create")
public class UserController {
    @Autowired
    private insertTempData it1;
    @Autowired
    private checkVerification verificationObj;
    @PostMapping("/addTempUserMongo")
    public userMongoResponse addTempUserDetails(@RequestBody UserTempRequest request) throws
MailjetException {
        usersMongo users1 = new usersMongo(
            request.getUser().getUsername(),
            request.getUser().getEmail(),
            request.getUser().getPassword_hash(),
            request.getUser().getCreated_at(),
            ""
        );

        user_profilesMongo users_profile1 = new user_profilesMongo(
            request.getProfile().getFirst_name(),
            request.getProfile().getLast_name(),
            request.getProfile().getBio(),
            request.getProfile().getProfile_pic_url(),
            request.getProfile().getLatLocation(),
            request.getProfile().getLonLocation(),
            request.getProfile().getBirdth_date(),
            request.getProfile().getJoined_at(),
            request.getProfile().getAge(),
            request.getProfile().getMobile(),
            request.getProfile().getAddress(),
            request.getProfile().getJob(), ""
        );
        usersMongo us1 = new
usersMongo(users1.getUsername(),users1.getEmail(),users1.getPassword_hash(),users1.getCreated_at
(),"");
        user_profilesMongo us21 = new
user_profilesMongo(users_profile1.getFirst_name(),users_profile1.getLast_name(),users_profile1.g
etBio(),users_profile1.getProfile_pic_url(),users_profile1.getLatLocation(),users_profile1.getLo
nLocation(),users_profile1.getBirdth_date(),users_profile1.getJoined_at(),users_profile1.getAge(
),users_profile1.getMobile(),users_profile1.getAddress(),users_profile1.getJob(),"");
        userMongoResponse us2 = it1.saveUserTemp(us1,us21);
        return us2;
    }
    @GetMapping("/checkVerification")
    public int checkVerification(@RequestParam String tempDataID, @RequestParam String
verificationID){
        int status = verificationObj.checkVerificationFun(tempDataID,verificationID);
    }
}

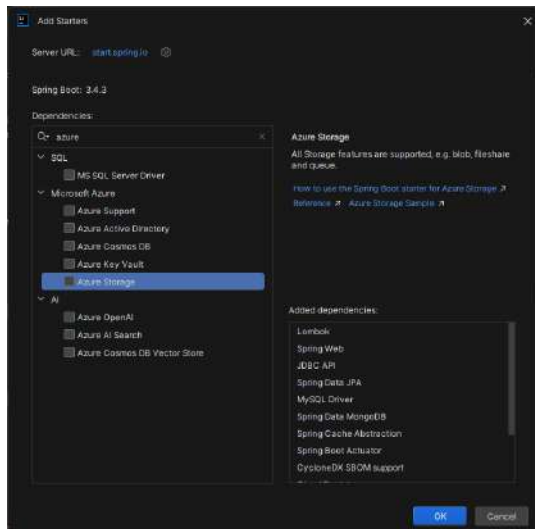
```

```

    return status;
}
}

```

CONNECTING MS AZURE BLOB STORAGE (AZURITE)



Or

- Add this dependency to pom.xml file.

```

<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-storage-blob</artifactId>
</dependency>

```

- Then add these to the config file:

```

package com.example.createAccountService.Configuration;

import com.azure.storage.blob.BlobServiceClient;
import com.azure.storage.blob.BlobServiceClientBuilder;
import com.example.createAccountService.Service.checkVerification;
import com.example.createAccountService.Service.insertTempData;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class userConfig {
    @Bean
    public insertTempData logInBean() {
        return new insertTempData();
    }
    @Bean
    public ModelMapper modelMapperBean() {
        return new ModelMapper();
    }
    @Bean
    public BlobServiceClient blobServiceClient() {
        return new BlobServiceClientBuilder()
            .endpoint("http://127.0.0.1:10000/devstoreaccount1")
            .credential(new com.azure.storage.common.StorageSharedKeyCredential(

```



```

        "devstoreaccount1",
        "ACCOUNT_KEY"
    ))
    .buildClient();
}
}

```

- Then create an endpoint to accept files in the controller:

```

@PostMapping("/saveProfileImage")
public String addProfileImage(@RequestParam("file") MultipartFile file, @RequestParam int
fileNumber) {
    try {
        String url = addImage.addImage(file, fileNumber);
        return url;
    } catch (Exception e) {
        return "ERROR Occured" + e.toString();
    }
    // return "passed";
}

```

- In the service add the file to the azure blob storage (create a separate class file in the service directory):

```

package com.example.createAccountService.Service;

import com.azure.storage.blob.BlobClient;
import com.azure.storage.blob.BlobContainerClient;
import com.azure.storage.blob.BlobServiceClient;
import com.azure.storage.blob.BlobServiceClientBuilder;
import org.springframework.stereotype.Service;
import org.springframework.web.multipart.MultipartFile;

import java.io.IOException;

@Service
public class profileImageAdder {
    private final BlobServiceClient blobServiceClient;

    public profileImageAdder() {
        // Use the correct Azurite connection string with the full development storage account
        key
        String connectionString =
"DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;AccountKey=KEY;BlobEndpoint=http://1
27.0.0.1:10000/devstoreaccount1;";

        this.blobServiceClient = new BlobServiceClientBuilder()
            .connectionString(connectionString)
            .buildClient();
    }

    public String addImage(MultipartFile file, int fileNumber) throws IOException {
        BlobContainerClient containerClient =
blobServiceClient.getBlobContainerClient("profileimages");
        if (!containerClient.exists()) {
            containerClient.create();
        }
        String ext =
file.getOriginalFilename().substring(file.getOriginalFilename().lastIndexOf("."));
        String newFileName = "userpic" + fileNumber + ext;
        BlobClient blobClient = containerClient.getBlobClient(newFileName);
        // Upload the file
        blobClient.upload(file.getInputStream(), file.getSize(), true);
        return blobClient.getBlobUrl();
    }
}

```

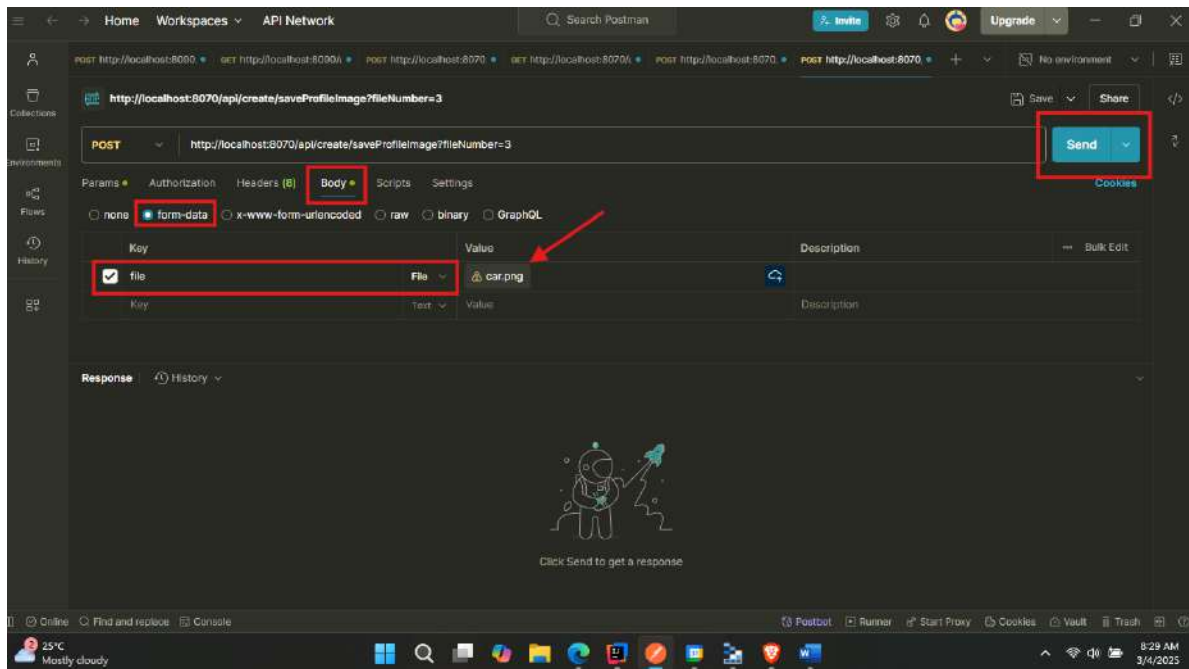
- Also, modify the application.properties as follows:

```

azure.storage.connection-
string=DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;AccountKey=ACCOUNT_KEY;BlobEnd
point=http://127.0.0.1:10000/devstoreaccount1;

```

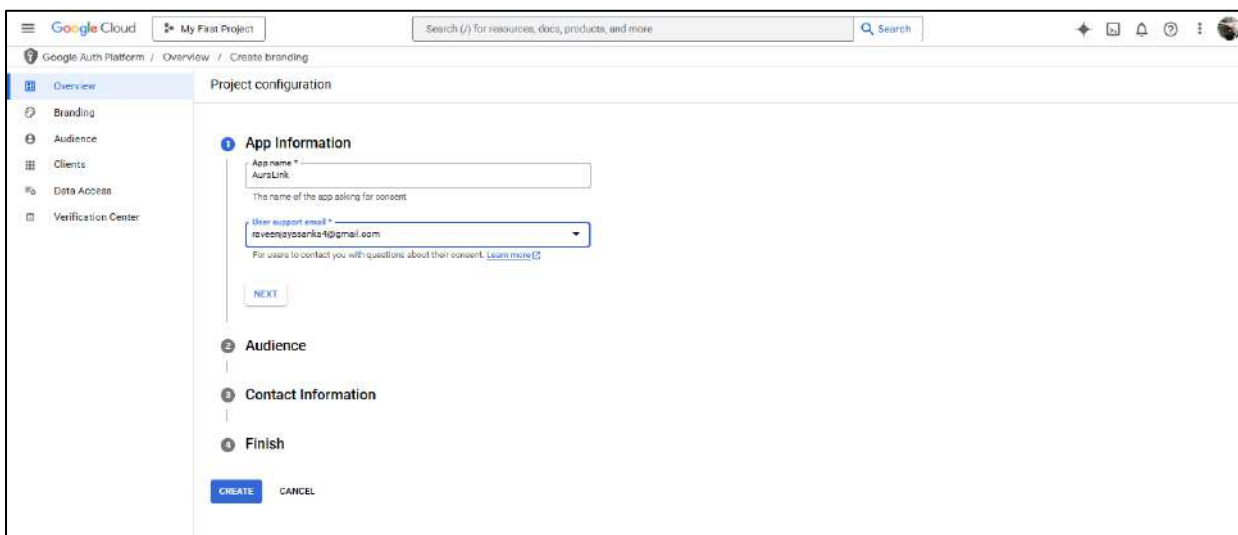
- Now test the service using postman as follows:



- Also, you can add other details as well.
- Note! Before sending the request start the emulator (azure) using the command “azurite” in the terminal. Also, install the azurite via NPM.

USING OAUTH IN SPRING BOOT

- First go to the google cloud console and go to API and services section.
- Then click “consent screen” to create a new screen.
- First enter the application information as follows:



- Select the Audience as “External”.

Google Cloud | My First Project | Search (/) for resources, docs, products, and more

Google Auth Platform / Overview / Create branding

Overview | Branding | Audience | Clients | Data Access | Verification Center

Project configuration

- 1 App Information
- 2 Audience
- 3 Contact Information
- 4 Finish

Audience

☐ Internal ⓘ

Only available to users within your organization. You will not need to submit your app for verification. [Learn more about user type](#)

☒ External ⓘ

Available to any test user with a Google Account. Your app will start in testing mode and will only be available to users you add to the list of test users. Once your app is ready to push to production, you may need to verify your app. [Learn more about user type](#)

NEXT

CREATE CANCEL

- Finally, insert the contact information and finish setup.
- After that, enter the application home page as follows:

Google Auth Platform / Branding

Overview | Branding | Audience | Clients | Data Access | Verification Center

Branding

App domain

To protect you and your users, Google only allows apps using OAuth to use Authorized Domains. The following information will be shown to your users on the consent screen.

Application home page

Provide users a link to your home page

Application privacy policy link

Provide users a link to your public privacy policy

Application terms of service link

Provide users a link to your public terms of service

Authorized domains ⓘ

When a domain is used on the consent screen or in an OAuth client's configuration, it must be pre-registered here. If your app needs to go through verification, please go to the [Google Search Console](#) to check if your domains are authorized. [Learn more](#) about the authorized domain limit.

+ ADD DOMAIN

Developer contact information

Email addresses *

These email addresses are for Google to notify you about any changes to your project.

OAuth configuration created! X

Google cloud console - Data Access - Google Auth Pla...

Update selected scopes

Only scopes for enabled APIs are listed below. To add a missing scope to this screen, find and enable the API in the [Google API Library](#) or use the Pasted Scopes text box below. Refresh the page to see any new APIs you enable from the Library.

Filter Enter property name or value

API	Scope	User-facing description
<input checked="" type="checkbox"/>	.../auth/userinfo.email	See your primary Google Account email address
<input checked="" type="checkbox"/>	.../auth/userinfo.profile	See your personal info, including any personal info you've made publicly available
<input checked="" type="checkbox"/>	openid	Associate you with your personal info on Google
<input type="checkbox"/>	Analytics Hub API .../auth/bigquery	View and manage your data in Google BigQuery and see the email address for your Google Account
<input type="checkbox"/>	Analytics Hub API .../auth/cloud-platform	See, edit, configure, and delete your Google Cloud data and see the email address for your Google Account.
<input type="checkbox"/>	BigQuery API .../auth/bigquery.readonly	View your data in Google BigQuery
<input type="checkbox"/>	BigQuery API .../auth/cloud-platform.read-only	View your data across Google Cloud services and see the email address of your Google Account.
<input type="checkbox"/>	BigQuery API .../auth/devstorage.full_control	Manage your data and permissions in Cloud Storage and see the email address for your Google Account.
<input type="checkbox"/>	BigQuery API .../auth/devstorage.read_only	View your data in Google Cloud Storage
<input type="checkbox"/>	BigQuery API .../auth/devstorage.read_write	Manage your data in Cloud Storage and see the email address of your Google Account

Rows per page: 10 1 - 10 of 25

Manually add scopes

If the scopes you would like to add do not appear in the table above, you can enter them here. Each scope should be on a new line or separated by commas. Please provide the full scope string (beginning with "https://"). When you are finished, click "Add to table".

- Create a new spring boot project and add these dependencies (related to the authentication and google people API):

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- Use Hibernate compatible with Spring Boot 3.4.3 -->
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>6.4.3.Final</version>
</dependency>
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>3.1.1</version>
</dependency>
<dependency>
```

```

    <groupId>com.google.api-client</groupId>
    <artifactId>google-api-client-gson</artifactId>
    <version>2.7.2</version>
</dependency>
<dependency>
    <groupId>com.google.api-client</groupId>
    <artifactId>google-api-client</artifactId>
    <version>1.27.0</version>
</dependency>
<dependency>
    <groupId>com.google.oauth-client</groupId>
    <artifactId>google-oauth-client-java6</artifactId>
    <version>1.11.0-beta</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.google.apis/google-api-services-people -->
<dependency>
    <groupId>com.google.apis</groupId>
    <artifactId>google-api-services-people</artifactId>
    <version>v1-rev537-1.25.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.google.oauth-client/google-oauth-client-jetty -->
<dependency>
    <groupId>com.google.oauth-client</groupId>
    <artifactId>google-oauth-client-jetty</artifactId>
    <version>1.39.0</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-netflix-eureka-client -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>4.2.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-config -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
    <version>4.2.0</version>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework.security/spring-security-test -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <version>6.4.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
<!-- default -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

```

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

- Then, create the controller class to fetch the information (basic) from account.

```
package com.example.SocialAuthLogIn.Controller;

import com.example.SocialAuthLogIn.Entity.users;
import com.example.SocialAuthLogIn.Services.checkUser;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.oauth2.client.authentication.OAuth2AuthenticationToken;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class AccessPoint {
    @Autowired
    private checkUser ck;
    @GetMapping("/")
    public List<users> getDetsils(OAuth2AuthenticationToken authentication){
        String fname =
authentication.getPrincipal().getAttributes().get("given_name").toString();
        String lname =
authentication.getPrincipal().getAttributes().get("family_name").toString();
        String picUrl = authentication.getPrincipal().getAttributes().get("picture").toString();
        String email = authentication.getPrincipal().getAttributes().get("email").toString();
        //      return "Name : " + fname;
        return ck.fetchAll();
    }
}
```

- Set up the application.properties file as follows:

```
eureka.instance.hostname=SocialAuthLogIn
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
server.port=8002
spring.application.name=socialLogin
spring.config.import=optional:configserver:http://localhost:8888
#spring.jpa.generate-ddl=true
#spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfigura
tion
#mysql setup
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/database_name
spring.datasource.username=username
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true
#Oauth setup - google
spring.security.oauth2.client.registration.google.client-id=your_client_id
spring.security.oauth2.client.registration.google.client-secret=your_client_secret
spring.security.oauth2.client.registration.google.scope=openid,profile,email,https://www.googlea
pis.com/auth/user.addresses.read,https://www.googleapis.com/auth/user.phonenumbers.read
```

SPRING BOOT ESSENTIALS

- ENUMS IN SPRING ENTITIES

```
@Entity
@Table(name="Posts")
public class Posts {
    public enum ContentType{
        VIDEO,
        IMAGE,
        TEXT,
        VIDEO_IMAGES
    }
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "postId")
    private int postId;
    @Enumerated(EnumType.STRING)
    @Column(name = "contentType")
    private ContentType contentType;
}
```

- Using foreign keys

```
@ManyToOne
@JoinColumn(name = "user_id", nullable = false)
private users user_id;
```

- Using @JsonProperty

```
package com.example.SupplierService.Request;
import com.fasterxml.jackson.annotation.JsonProperty;

import java.sql.Timestamp;
public class CreateSupplierRequestMongo {
    @JsonProperty("Fname")
    private String Fname;
    @JsonProperty("Lname")
    private String Lname;
    @JsonProperty("Address")
    private String Address;
    @JsonProperty("Email")
    private String Email;
    @JsonProperty("Phone")
    private String Phone;
    @JsonProperty("SupplierWeb")
    private String SupplierWeb;
    @JsonProperty("CreatedDate")
    private Timestamp CreatedDate;
    @JsonProperty("CreatedTime")
    private Timestamp CreatedTime;
    @JsonProperty("UserName")
    private String UserName;
    @JsonProperty("PasswordSupplier")
    private String PasswordSupplier;
    @JsonProperty("City")
    private String City;
    @JsonProperty("Country")
    private String Country;
    @JsonProperty("Status")
    private String Status;
    @JsonProperty("SupplierBankName")
    private String SupplierBankName;
    @JsonProperty("SupplierAccNo")
    private String SupplierAccNo;

    //    public CreateSupplierRequestMongo() {
    //    }

    public void setFname(String fname) {
        Fname = fname;
    }
}
```



```

public void setLname(String lname) {
    Lname = lname;
}

public void setAddress(String address) {
    Address = address;
}

public void setEmail(String email) {
    Email = email;
}

public void setPhone(String phone) {
    Phone = phone;
}

public void setSupplierWeb(String supplierWeb) {
    SupplierWeb = supplierWeb;
}

public void setCreatedDate(Timestamp createdDate) {
    CreatedDate = createdDate;
}

public void setCreatedTime(Timestamp createdTime) {
    CreatedTime = createdTime;
}

public void setUsername(String userName) {
    UserName = userName;
}

public void setPasswordSupplier(String passwordSupplier) {
    PasswordSupplier = passwordSupplier;
}

public void setCity(String city) {
    City = city;
}

public void setCountry(String country) {
    Country = country;
}

public void setStatus(String status) {
    Status = status;
}

public void setSupplierBankName(String supplierBankName) {
    SupplierBankName = supplierBankName;
}

public void setSupplierAccNo(String supplierAccNo) {
    SupplierAccNo = supplierAccNo;
}

public String getFname() {
    return Fname;
}

public String getLname() {
    return Lname;
}

public String getAddress() {
    return Address;
}

public String getEmail() {
    return Email;
}

public String getPhone() {
    return Phone;
}

```

```

}

public String getSupplierWeb() {
    return SupplierWeb;
}

public Timestamp getCreatedDate() {
    return CreatedDate;
}

public Timestamp getCreatedTime() {
    return CreatedTime;
}

public String getUserName() {
    return UserName;
}

public String getPasswordSupplier() {
    return PasswordSupplier;
}

public String getCity() {
    return City;
}

public String getCountry() {
    return Country;
}

public String getStatus() {
    return Status;
}

public String getSupplierBankName() {
    return SupplierBankName;
}

public String getSupplierAccNo() {
    return SupplierAccNo;
}

public CreateSupplierRequestMongo(String fname, String lname, String address, String email,
String phone, String supplierWeb, Timestamp createdDate, Timestamp createdTime, String userName,
String passwordSupplier, String city, String country, String status, String supplierBankName,
String supplierAccNo) {
    Fname = fname;
    Lname = lname;
    Address = address;
    Email = email;
    Phone = phone;
    SupplierWeb = supplierWeb;
    CreatedDate = createdDate;
    CreatedTime = createdTime;
    UserName = userName;
    PasswordSupplier = passwordSupplier;
    City = city;
    Country = country;
    Status = status;
    SupplierBankName = supplierBankName;
    SupplierAccNo = supplierAccNo;
}
}

```

- In your `CreateSupplierRequestMongo` class, your field names start with uppercase letters (e.g., `Fname`, `Lname`), but Spring's default JSON deserializer (Jackson) expects camelCase property names (e.g., `fname`, `lname`).
- When you send JSON with fields like `"Fname"`, Jackson is looking for a field named `fname` (lowercase first letter), not finding it, and leaving your fields null.
- Fixing this error :

2025-03-25T12:22:04.233+05:30 ERROR 7996 --- [LoginUsingGoogle] [nio-8094-exec-5] o.a.c.c.C.[.][.][dispatcherServlet] : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed: org.springframework.http.converter.HttpMessageConversionException: Type definition error: [simple type, class java.time.Instant]] with root cause
com.fasterxml.jackson.databind.exc.InvalidDefinitionException: Java 8 date/time type `java.time.Instant` not supported by default: add Module "com.fasterxml.jackson.datatype:jackson-datatype-jsr310" to enable handling (through reference chain: java.util.ImmutableCollections\$MapN["user"]->org.springframework.security.oauth2.core.oidc.user.DefaultOidcUser["authorities"]->java.util.Collections\$UnmodifiableSet[0]->org.springframework.security.oauth2.core.oidc.user.OidcUserAuthority["attributes"]->java.util.Collections\$UnmodifiableMap["exp"])

- First add this dependency:

```
<dependency>
  <groupId>com.fasterxml.jackson.datatype</groupId>
  <artifactId>jackson-datatype-jsr310</artifactId>
</dependency>
```

- Then configure the object mapper as follows in the config file:

```
@Bean
@Primary
public ObjectMapper objectMapper() {
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.registerModule(new JavaTimeModule());
    return objectMapper;
}
```

- Logout process in OAuth google account :

```
@GetMapping("/logOut")
public String logOut(HttpServletRequest request, HttpServletResponse response) {
    SecurityContextHolder.clearContext();
    HttpSession session = request.getSession(false);
    if (session != null) {
        session.invalidate();
    }
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            cookie.setValue("");
            cookie.setPath("/");
            cookie.setMaxAge(0);
            response.addCookie(cookie);
        }
    }
    return "redirect:https://accounts.google.com/logout";
}
```

- Working with ResponseEntity

```
@PostMapping("/checkVerification")
public ResponseEntity<String> checkVerification(String tempdataID, String verificationCode) {
    if (tempdataID == null || verificationCode == null) {
        return new ResponseEntity<String>("The values are null", HttpStatus.CREATED);
    }
    return null;
}
```

COMMON MISTAKES IN SESSION TOKENS

- Return correct value in the entity.

```
@Override
public String getUsername() {
    return getUserNam  ;
}
```

- Disable CSRF for testing

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration {
    private AuthenticationProvider authenticationProvider; no usages
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        SecurityContextRepository securityContextRepository = new HttpSessionSecurityContextRep
        http
            .csrf(AbstractHttpConfigurer::disable) // Disable CSRF for testing purposes (en
            .authorizeHttpRequests(AuthorizationManagerRequestMat... authz -> authz
                .requestMatchers(@"/login").permitAll()
                .requestMatchers(@"/section2").authenticated()
                .anyRequest().authenticated() // Secure other endpoints
            )
            .sessionManagement(SessionManagementConfigurer<HttpSecurity> session -> session.se
            .formLogin(AbstractHttpConfigurer::disable); // Disable default form login

        return http.build();
    }
}
```

- Create an empty constructor in the log in request. For the internal processing.

```
public class LoginRequest {
    // ...
    public void setPassword(String password) {
        Password = password;
    }
    public String getUsername() {
        return Username;
    }
    public String getPassword() {
        return Password;
    }
    public LoginRequest(String username, String password) {
        Username = username;
        Password = password;
    }
    public LoginRequest() {
    }
}
```

- Define the @Bean of the fetch primary details manually.

```
@Bean
public FetchPrimaryDetails loginBean() { return new FetchPrimaryDetails(); }
```

- When using OAuth with spring security, modify the security configuration as follows (try to use “/” endpoint in the controller for this task):

```

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    SecurityContextRepository securityContextRepository = new HttpSessionSecurityContextRepository();
    http
        .csrf(AbstractHttpConfigurer::disable) // Disable CSRF for testing purposes (enable it for production)
        .authorizeHttpRequests( AuthorizationManagerRequestMatcher::authz -> authz
            .requestMatchers( @"/login", @"/", @"/oauth2/**", @"/login/oauth2/**").permitAll()
            .requestMatchers( @"/section2").authenticated()
            .anyRequest().authenticated() // Secure other endpoints
        )
        .oauth2Login( @withDefaults() )
        .sessionManagement( SessionManagementConfigurer<HttpSecurity> session -> session.sessionCreationPolicy( SessionCreationPolicy.ALWAYS ) )
        .formLogin(AbstractHttpConfigurer::disable); // Disable default form login

    return http.build();
}

```

```

    .oauth2->oauth2.defaultSuccessUrl( defaultSuccessUrl: "/login-success", alwaysUse: true ))
    .sessionManagement( SessionManagementConfigurer<HttpSecurity> session -> session.sessionCreationPolicy( SessionCreationPolicy.IF_REQUIRED )) // No sessions for
    .formLogin(AbstractHttpConfigurer::disable); // Disable default form login

```

Handling exceptions of oracle procedures (in the controller and remove handlings in the service class)

```

package com.example.DashBoardSupplier.Controller;

import com.example.DashBoardSupplier.Entity.ProductCategories;
import com.example.DashBoardSupplier.Entity.Products;
import com.example.DashBoardSupplier.Entity.Supplier;
import com.example.DashBoardSupplier.Request.SaveNewItems;
import com.example.DashBoardSupplier.Service.InsertProducts;
import jdk.jfr.Category;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
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 org.springframework.web.multipart.MultipartFile;

import java.sql.Timestamp;
import java.util.Date;

@RestController
@RequiredArgsConstructor
public class AccessPoint {
    @Autowired
    private InsertProducts insertProducts;

    @PostMapping("/SaveProductsMain")
    public ResponseEntity<?> saveProduct(@RequestBody SaveNewItems saveNewItems) throws Exception {
        try{
            Date today = new Date();
            ProductCategories ctl = new ProductCategories();
            ctl.setCatID(saveNewItems.getCatID());
            Supplier sp = new Supplier();
            sp.setSupplierID(saveNewItems.getSupplierID());
            //set the image name
            Products products = new
Products(saveNewItems.getProductID(), saveNewItems.getDescription(), saveNewItems.getWeight(), saveNewItems.getStatus(), saveNewItems.getAmount(),
            saveNewItems.getExpiryDate(), saveNewItems.getBrand(), new
Timestamp(today.getDate()), saveNewItems.getPrice(), saveNewItems.getTags(), "image_name_temp", save

```

```

NewItems.getShippingCost(),
            ctrl, sp);
        Products newProduct = insertProducts.SaveProductsProcess(products); //send the image
file
        return new ResponseEntity<Products>(newProduct, HttpStatus.ACCEPTED);
    } catch (Exception e) {
        String errorMsg = e.getMessage();
        return new ResponseEntity<String>("Error occurred : " +
errorMsg, HttpStatus.CONFLICT);
    }
}
}

```

References:

[Spring Boot Security With UserDetailsService and Authentication Provider | by Shubham Wankhede | Medium](#)

[Spring Security - UserDetailsService and UserDetails with Example - GeeksforGeeks](#)