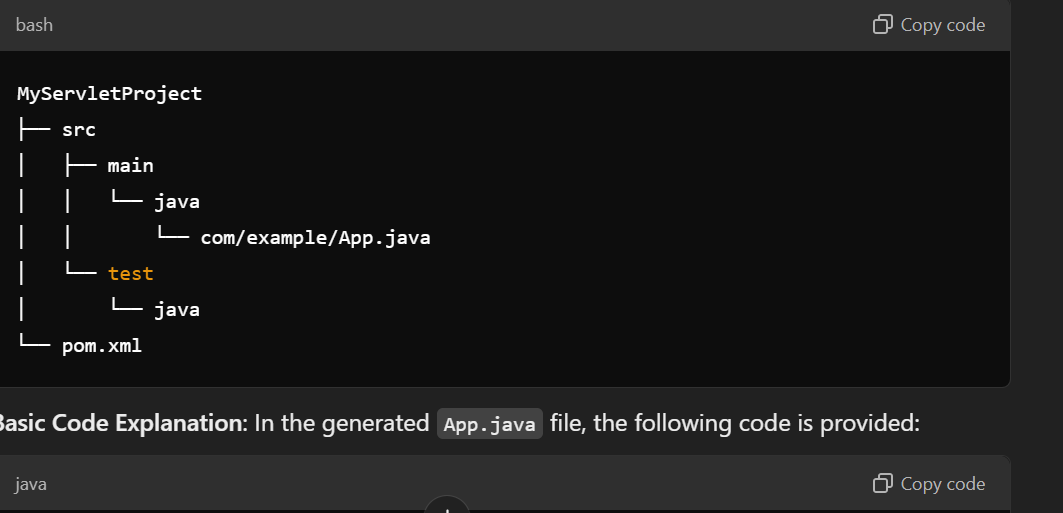
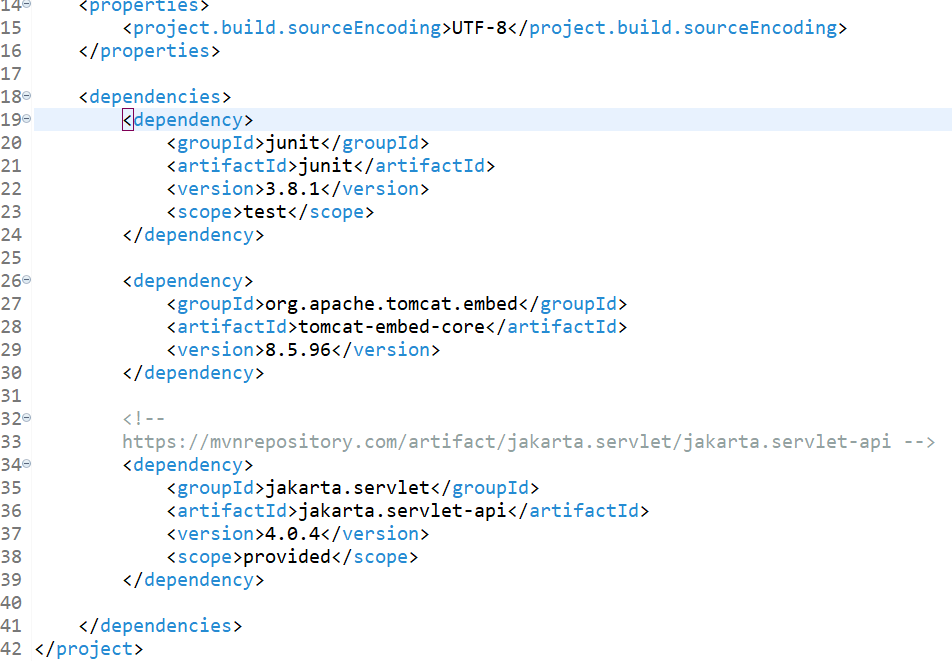
**SPRING BOOT WEB**

1. **Creating a Maven Project for a Simple Servlet**
   1. **Set Up the Maven Project**:

* Choose **org.apache.maven: maven-archetype-quickstart** as the archetype in your IDE or command line to set up the structure. This archetype provides a basic setup for Java applications with directories for Java source and test files.
* Name the project, choose a groupId (like com.example), and artifactId (like MyServletProject).
  1. **Project Structure**:
* **src/main/java**: Contains application code.
* **src/test/java**: Contains test code.
* **pom.xml**: The Maven configuration file that manages dependencies, build configuration, and other project settings.



POM> XML

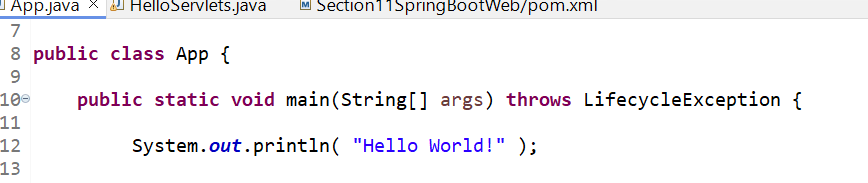


**1.3 Dependencies**:

* junit: For unit testing (scope: test).
* tomcat-embed-core: Provides the embedded Tomcat server.
* jakarta.servlet-api: Supplies servlet classes (scope: provided for runtime only).

This setup ensures you have the foundational tools to run a web application with servlets in Spring Boot.

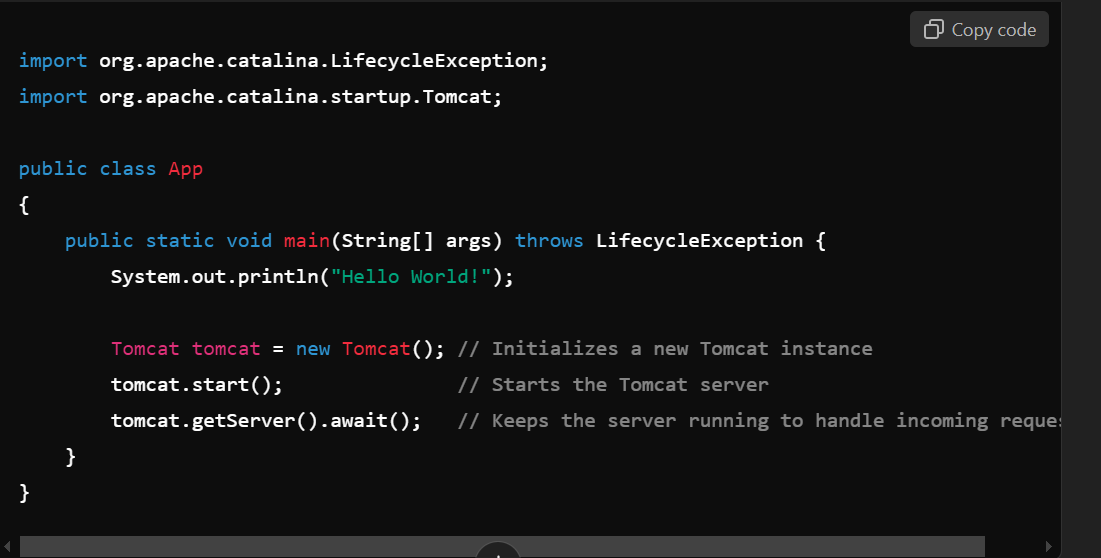
App.java



This simple output confirms that the Maven setup is working. Now you’re ready to extend this basic project by adding servlet functionality, configuring an embedded server (e.g., Tomcat), and setting up pom.xml dependencies specific to servlets and Spring Boot.

1. **Running tomcat**
   1. **Main Class App.java**

The App class starts the embedded Tomcat server.

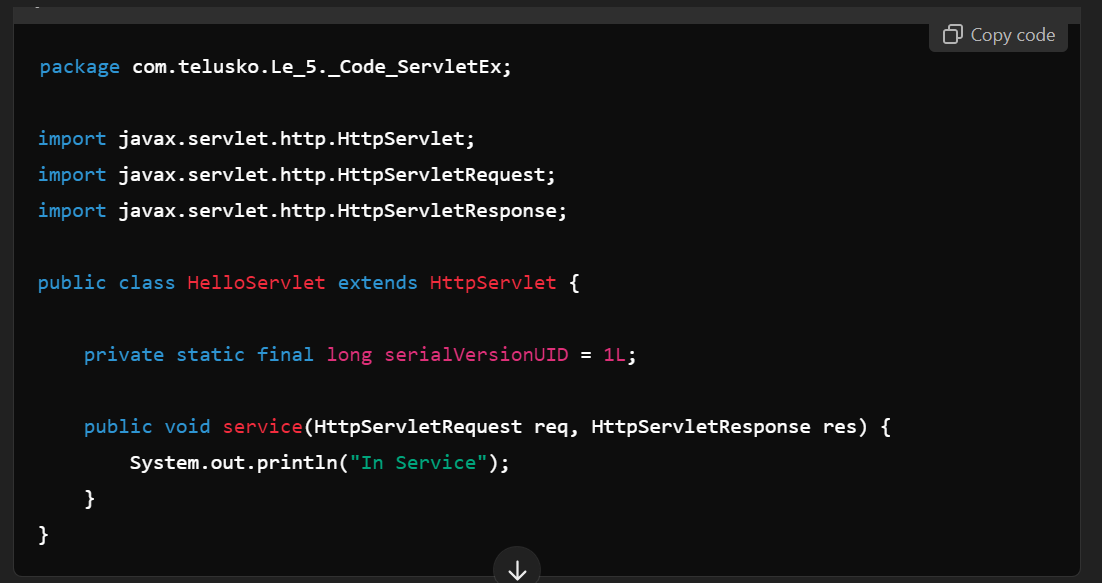
****

**2.2 Explanation:**

* Tomcat tomcat = new Tomcat();: Initializes a new embedded Tomcat server instance.
* tomcat.start();: Starts the Tomcat server.
* tomcat.getServer().await();: Puts the server in a listening state, keeping it running for handling requests

**2.3 Servlet Class (HelloServlet)**

The HelloServlet class extends HttpServlet and overrides the service() method, handling requests.



**2.4 Explanation:**

* **service() Method**: Logs "In Service" when handling HTTP requests but doesn’t produce any HTTP response. This is why no page loads on a direct request.

**2.5 Hitting http://localhost:8080**

If you navigate to http://localhost:8080, it will return a **404 error** because:

* The Tomcat server started, but the servlet wasn’t mapped to an endpoint.
* Without proper endpoint mapping, Tomcat doesn’t know how to route incoming requests to the HelloServlet.

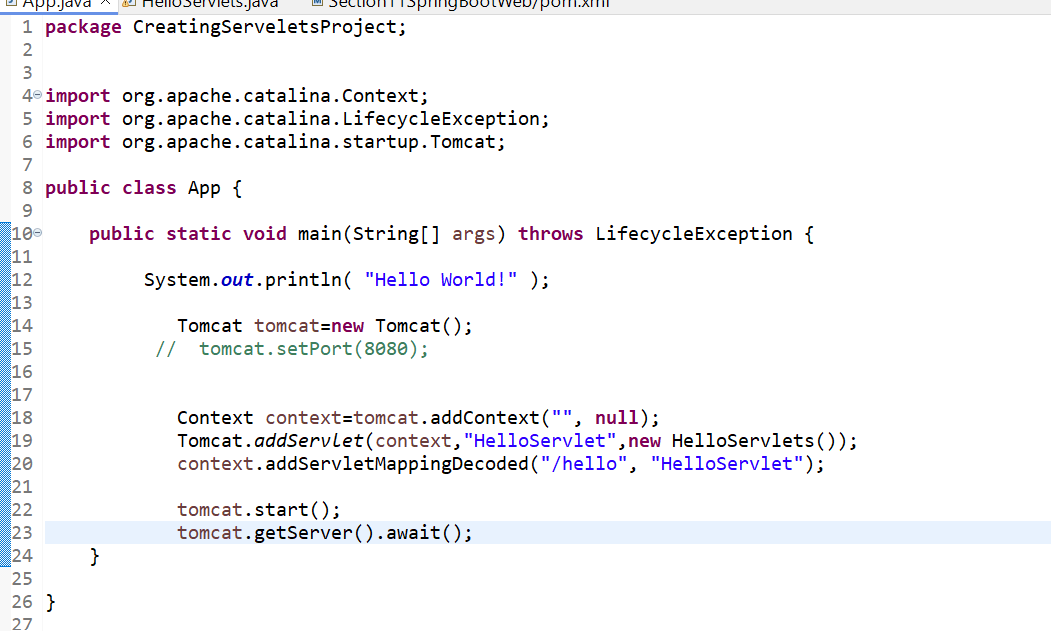
**To Fix:**

* **Map the servlet to a URL pattern** by adding code to App.java that explicitly registers HelloServlet with a path, e.g., /hello.
* Rebuild and rerun, then accessing http://localhost:8080/hello should trigger the service() method, printing “In Service” to the console.

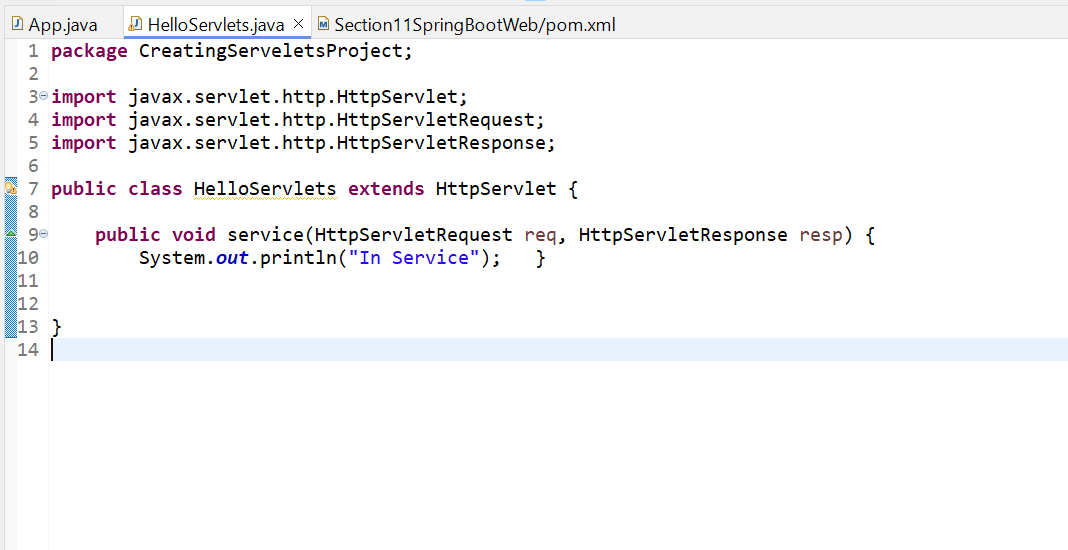
1. **Servlet Mapping**

This code demonstrates how to map a servlet in an embedded Tomcat server using Java code. Let’s go through each part:

App.java

****

Hello Servlets. Java

****

**3.1Code Explanation**

**3.1.1** Initialize and Start Tomcat:

Tomcat tomcat = new Tomcat();

Creates a new embedded Tomcat instance.

**3.1.2 Set Up a Context**:

***Context context = tomcat.addContext("", null);***

* + - 1. **Purpose**: The context is the environment or scope in which servlets are registered. Here, an empty string "" represents the **root context**, meaning any mappings created here apply directly under the base URL.E
      2. **Example**: Setting the context as "" means http://localhost:8080/hello maps directly to the root. If the context were "app1", requests would need to be prefixed with /app1, like <http://localhost:8080/app1/hello>.
    1. **Add and Map the Servlet**:

***Tomcat.addServlet(context, "HelloServlet", new HelloServlets());***

* The addServlet method registers a servlet within the specified context. It requires:
* context: The context within which this servlet operates.
* "HelloServlet": The servlet’s name (or identifier) used to refer to it within Tomcat.
* new HelloServlets(): An instance of the servlet class that Tomcat will use to process requests.

**Example:** This step creates a servlet instance under the name "HelloServlet". Think of it like naming a store item so it can be referred to easily for requests.

**Execution Flow**: When addServlet runs, Tomcat registers "HelloServlet" under the given context, but the servlet is not yet accessible to users without mapping it to a URL.

**3.1.4 Mapping a URL Path to the Servlet using addServletMappingDecoded**

***context.addServletMappingDecoded("/hello", "HelloServlet");***

* **Purpose**: This method maps a **URL path** ("/hello") to the servlet name ("HelloServlet"), so when users visit this URL, Tomcat will know which servlet to call.
* **Example**: If you visit http://localhost:8080/hello, this path matches the "/hello" mapping and directs the request to "HelloServlet".

**Execution Flow**: When a request is made to /hello, Tomcat uses addServletMappingDecoded to match the path with "HelloServlet". Since "HelloServlet" refers to HelloServlets, Tomcat triggers its service() method, executing any defined behavior for this servlet.

* + 1. **Start the Server and Await Requests**:

***tomcat.start();***

***tomcat.getServer().await();***

Starts the server and keeps it running to handle requests.

With this setup, accessing http://localhost:8080/hello invokes the HelloServlets servlet. This code effectively sets up servlet routing within the Tomcat instance.

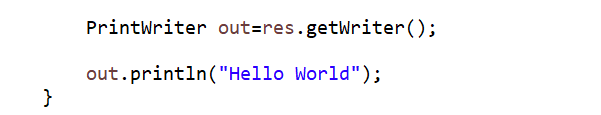
**Complete Example and Explanation of Flow**

1. **Define Context**: Sets up the environment or root URL where servlets are registered.
2. **Register Servlet with addServlet**: Registers "HelloServlet" within the context as an instance of HelloServlets.
3. **Map URL to Servlet with addServletMappingDecoded**: Associates the path /hello with "HelloServlet", ensuring requests to /hello are routed to HelloServlets.

After implementing this setup, running the server allows HelloServlets to respond to requests at http://localhost:8080/hello. If configured correctly, hitting this URL triggers the servlet’s service() method, which handles the response logic.

This flow provides a basic setup for managing and responding to specific URL paths using an embedded Tomcat server with servlet mappings in Java.

**4.Response Output to the Servlet**

****

* **PrintWriter out = res.getWriter();:** This retrieves a **PrintWriter** object from the **HttpServletResponse (res).** The **PrintWriter** is used to send character-based output (like HTML or text) to the client (browser).
* **out.println("Hello World");:** This writes **"Hello World**" to the response, which will be sent back to the client’s browser.

**4.0.1 Execution Flow:**

When a request is made to /hello, the servlet executes the service() method, and this code sends the text "Hello World" back to the browser, displaying it on the page.

Now, when you hit http://localhost:8080/hello, you should see "Hello World" printed on the page.

Now full code is



By switching from the service() method to doGet(), you are making the servlet specifically handle **HTTP GET requests**.

**4.1 Key Differences:**

* **service()**: This method is a general-purpose method that handles all types of HTTP requests (GET, POST, etc.). It’s usually overridden if you want to customize handling for all request types.
* **doGet()**: This method is specifically designed to handle **GET requests** (e.g., when a user visits a URL in their browser). By using doGet(), you’re focusing on requests that retrieve data from the server (typically read operations).

**4.1.1 Execution Flow:**

* When you visit http://localhost:8080/hello, the server calls the doGet() method.
* Inside doGet(), the PrintWriter writes "Hello World" to the response, and this text is sent back to the browser.

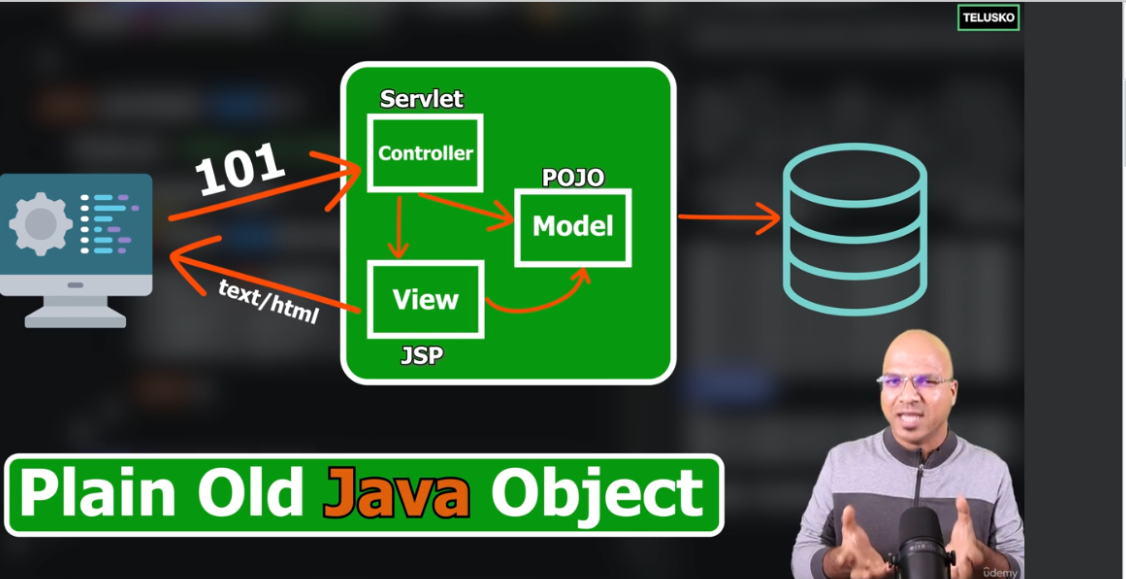
This method is more appropriate for responding to GET requests, such as when a browser requests a webpage.

**5.Introduction of MVC**

This diagram explains the **Model-View-Controller (MVC)** architecture in Java web development, specifically using Servlets and JSP (JavaServer Pages):

* **Controller (Servlet)**: Manages the application flow. It receives the HTTP request from the browser (client) and interacts with the model.
* **Model (POJO)**: Represents the business logic, often interacting with the database to fetch or store data.
* **View (JSP)**: Handles the presentation layer, generating HTML or text to display data to the user.

The **POJO (Plain Old Java Object)** represents the model's simplicity.



Here’s the flow of the **Model-View-Controller (MVC)** concept as shown in the diagram:

* **User Request**: A user sends a request (HTTP GET/POST) through the browser, like http://localhost:8080/hello.
* **Controller (Servlet)**: The Servlet (acting as the controller) receives the request, processes it, and interacts with the **Model** (POJO or business logic).
* **Model**: The Model fetches or updates data, often interacting with the database.
* **View (JSP)**: After the controller gets data from the model, it forwards it to the **View** (JSP) for rendering.
* **Response**: The **View (JSP)** generates HTML (or text) and sends it back to the client (browser) as the response.

**5.0.1 Flow Summary:**

* The **Controller** handles incoming requests.
* The **Model** contains the business logic and interacts with data.
* The **View** is responsible for displaying the data to the user in the browser (HTML or text).

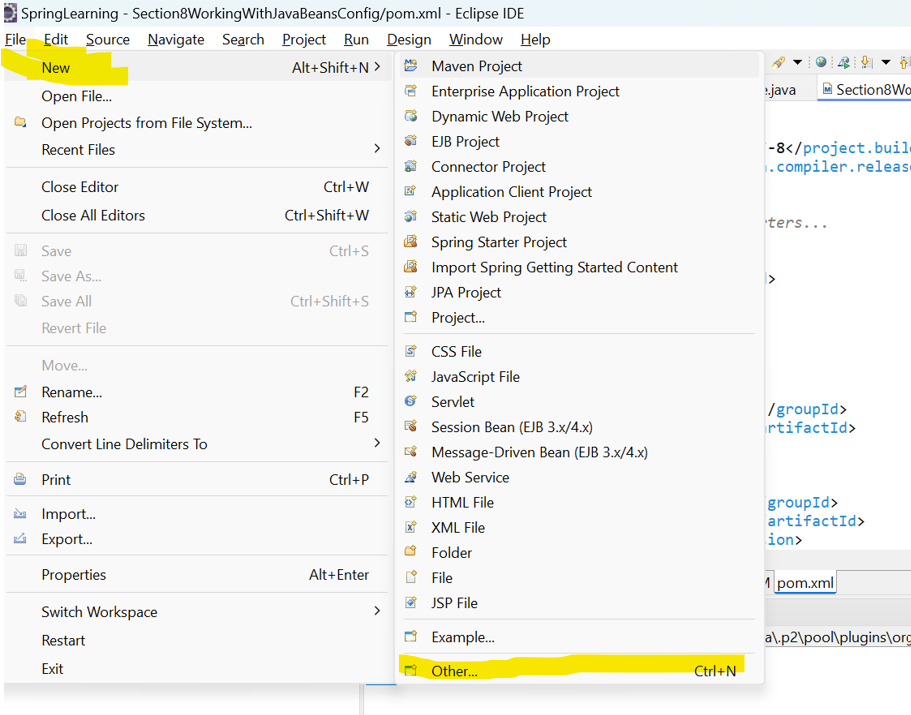
In this flow, the **Controller** orchestrates interactions between the **Model** and the **View**:

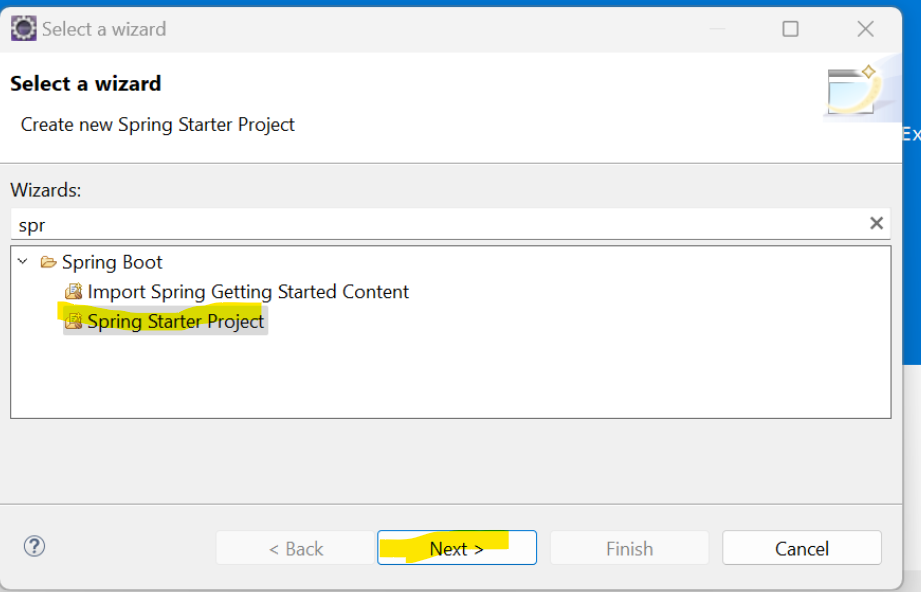
* **Model** handles data processing (fetching from the database or performing business logic).
* **View** presents that data back to the client in an easy-to-read format.

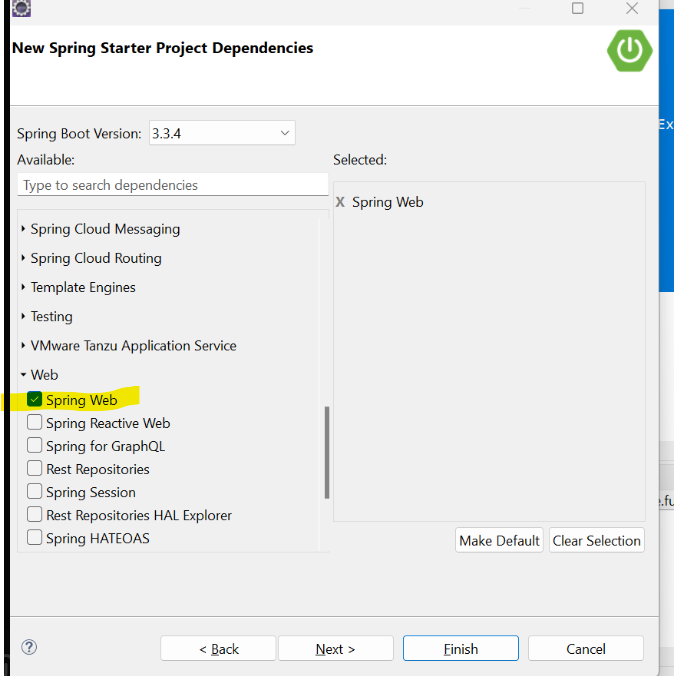
This separation of concerns allows better organization and scalability of web applications. Each layer (Model, View, and Controller) has its defined responsibility, making the system more maintainable and flexible.

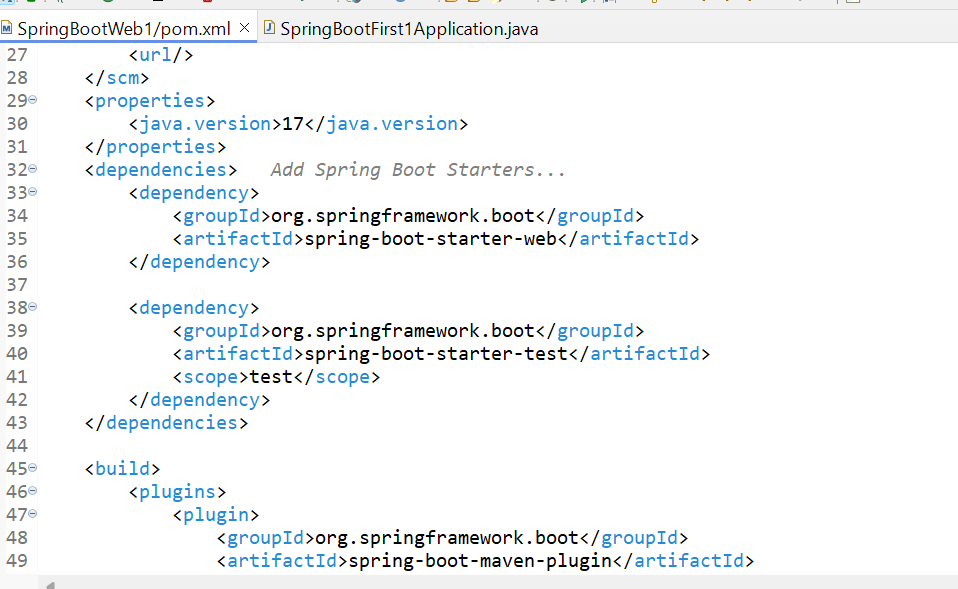
**6. Creating SpringBoot Web App Project**

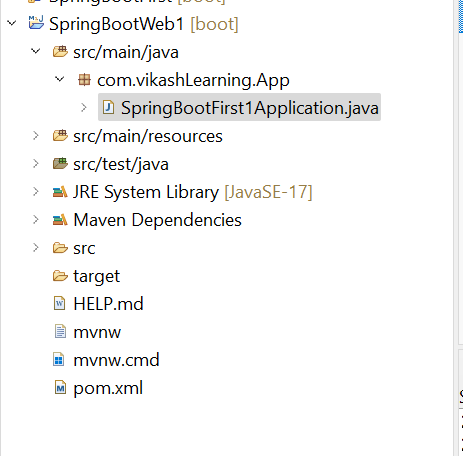
**Prerequisite**

****

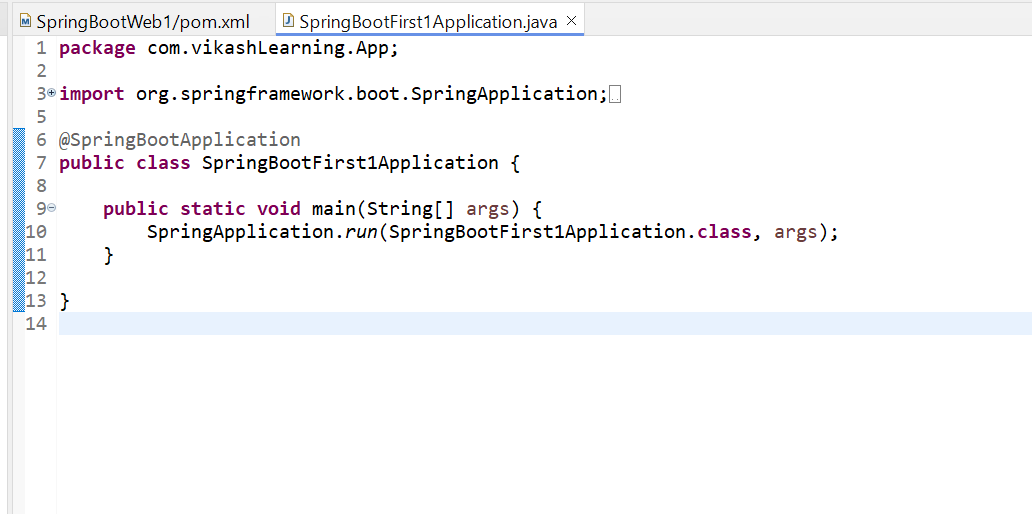
****

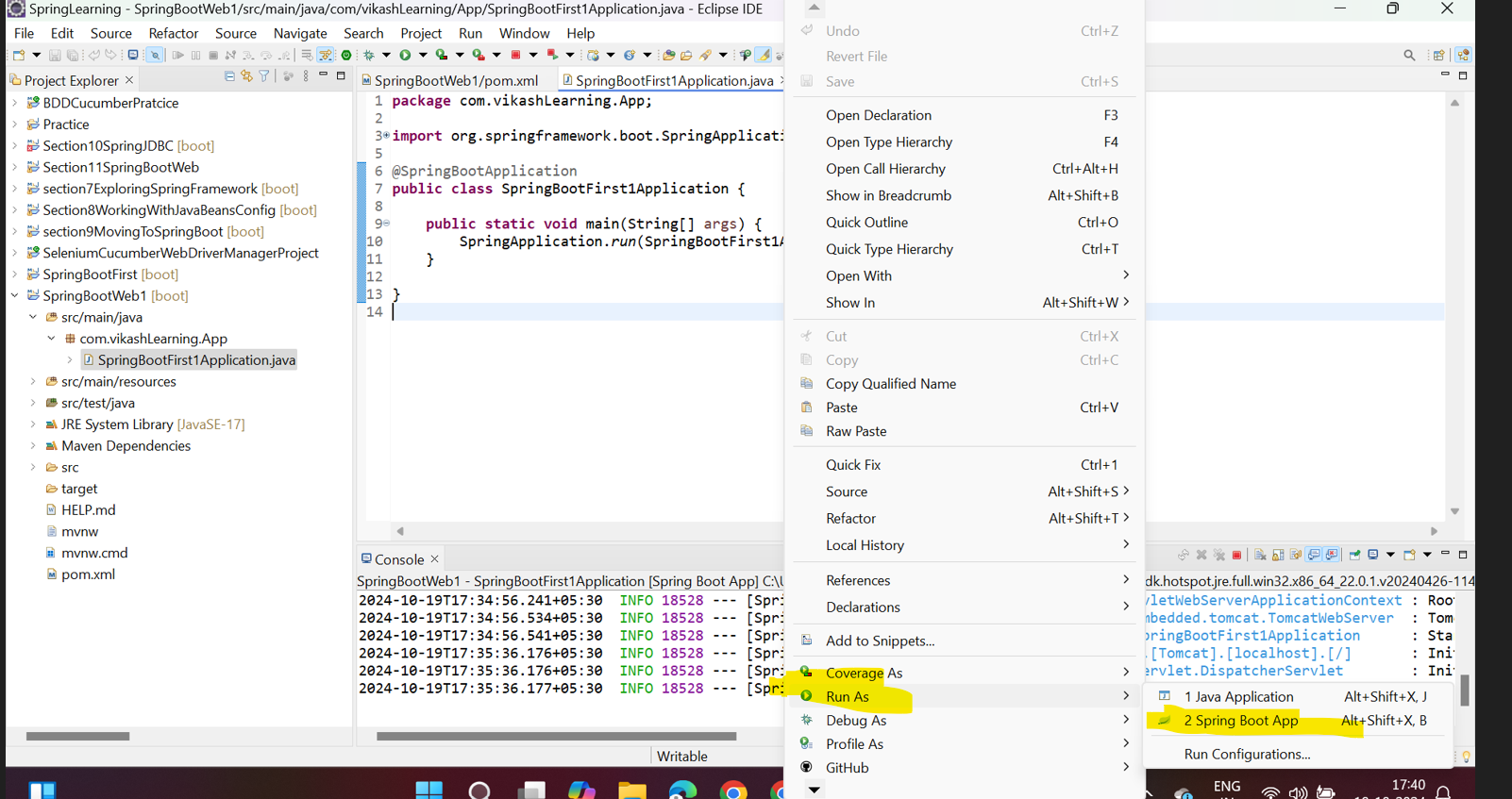
****

****

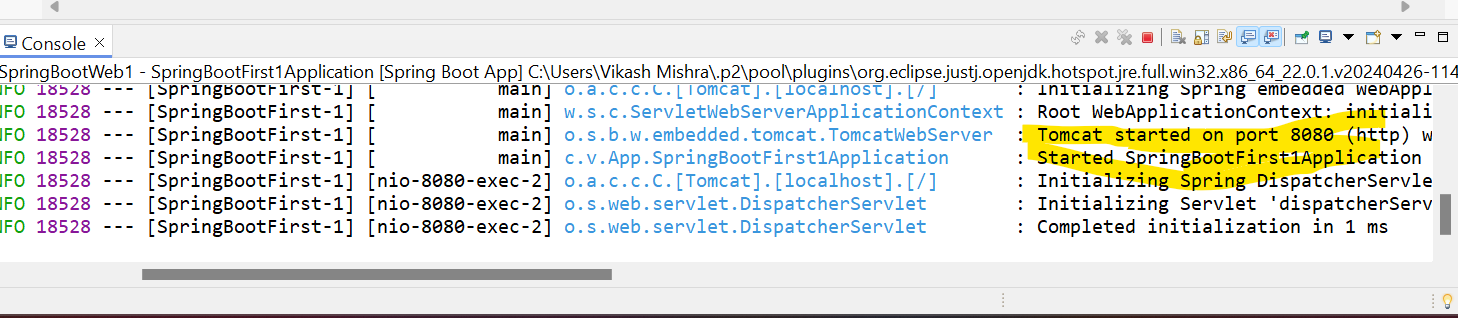
****

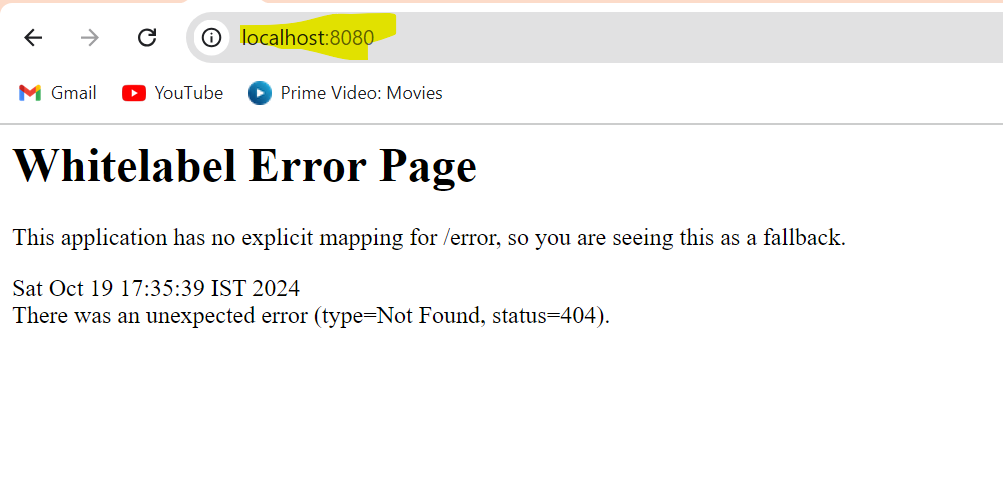
**6.1 Check the appilication which is running or not**





Console



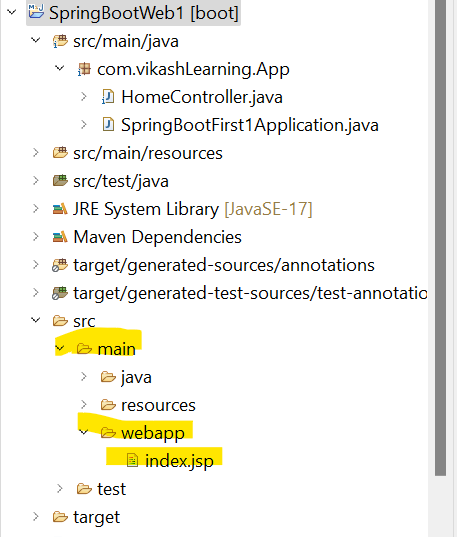
****

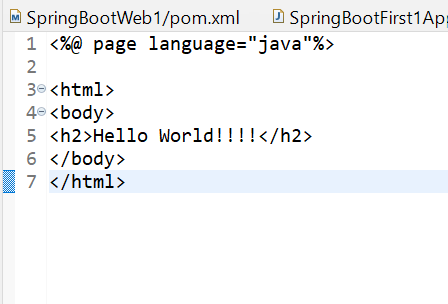
This means it work and looking for home page and at this time we don’t have home page so it is showing 404 we have used localhost8080 bcz tomcat by default use port 8080

**7. Creating JSP file**

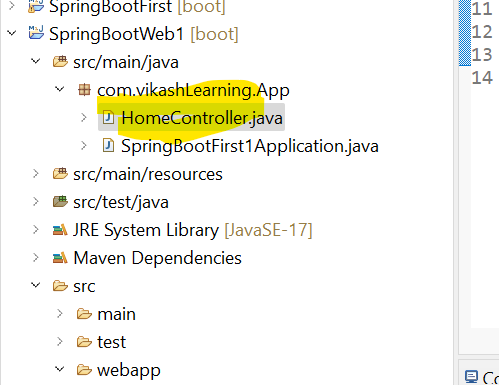
We have to create webapp folder in src file and inside that we have to create index.jsp. because servlet look for web app.

Src/main/webapp/index.jsp

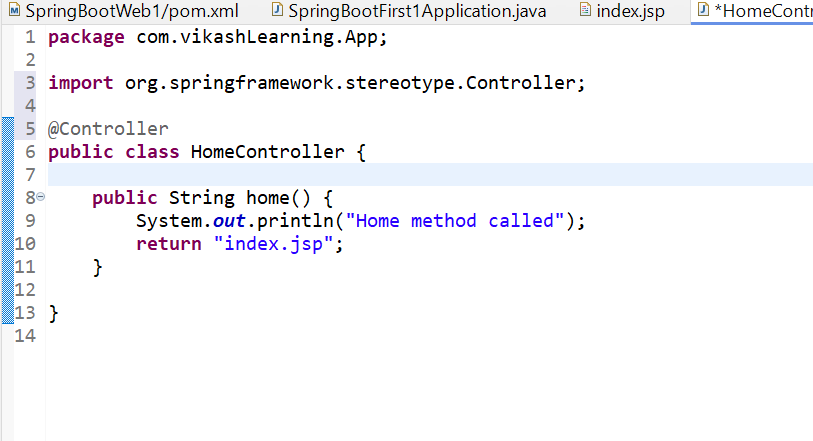




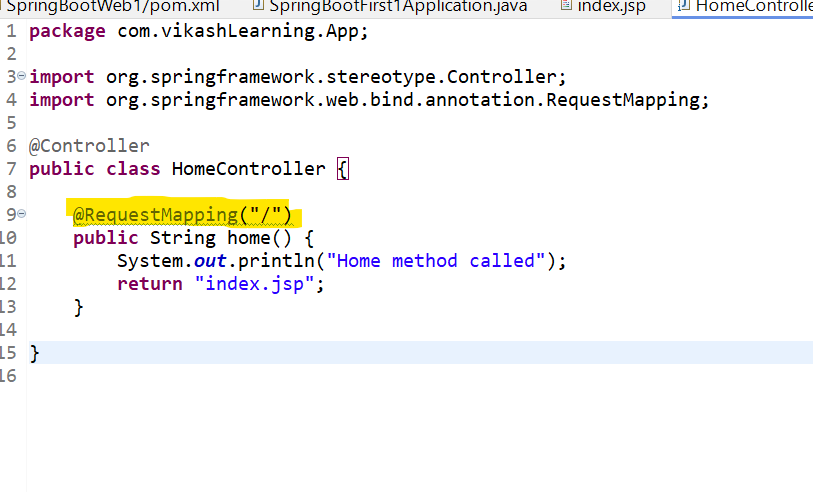
**8. Create Controller an Request Mapping**

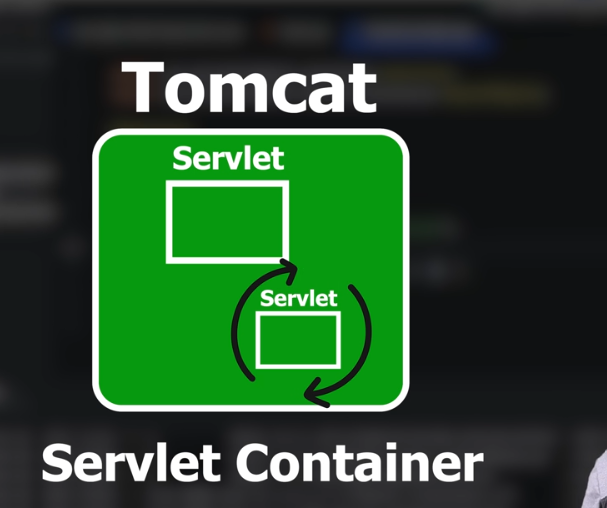


**8.1.1 Create HomeController class**

****

But this will not work here we have to map the URL. So for mapping we have to use below annotation





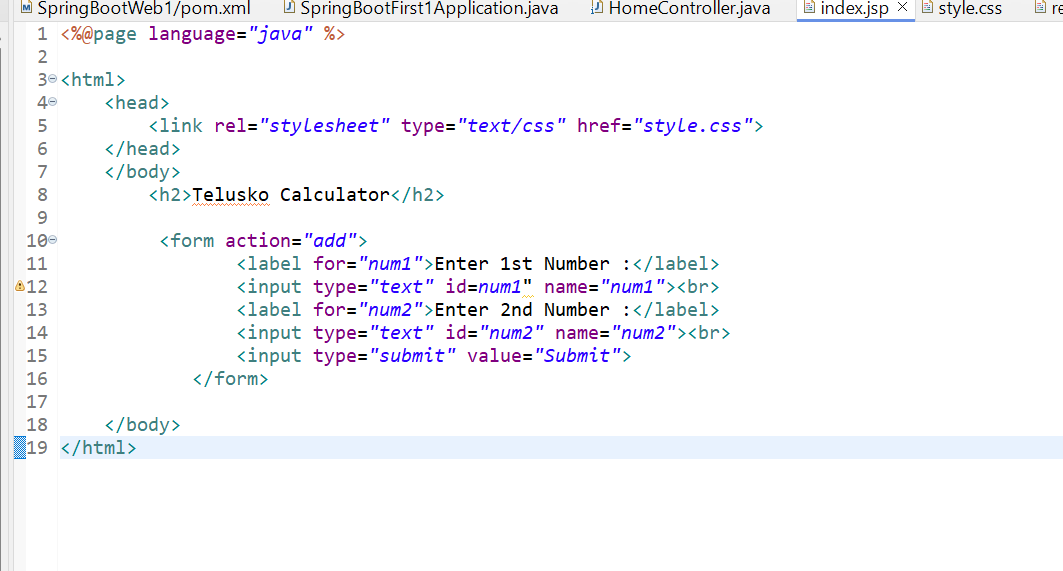
Before run the we have to convert **JSP into Servl**et because spring boot do not support JSP. So we have to add the dependency of **tomcat-jasper** which help to **convert JSP file in servlets** and we have use **same version as tomcat**. So add the below dependency in your POM>XML file.

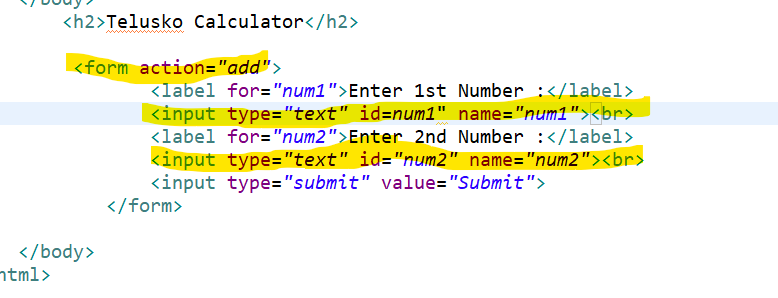


**9. Sending Data to controller**

In the **index.jsp** form, the data is sent to the controller when the form is submitted. The form action is defined as "**add**", which calls the **add()** method in the controller.

Create JSp file





This Css file   
  
**body** {

font-family: *'Roboto',* *sans-serif*;

background-color: *#FAFAFA*;

color: *#333*;

text-align: *center*;

padding: *50px*;

}

**h2** {

color: *#795548*;

margin-bottom: *30px*;

}

**form** {

background-color: *#FFFFFF*;

padding: *30px*;

border-radius: *4px*;

display: *inline-block*;

box-shadow: *0* *2px* *4px* *rgba(0,0,0,0.2)*;

max-width: *400px*;

width: *100%*;

box-sizing: *border-box*;

margin-top: *20px*;

}

**input**[type="text"]**,** **input**[type="submit"] {

border: *1px* *solid* *#E0E0E0*;

border-radius: *4px*;

padding: *10px* *15px*;

margin: *10px* *0*;

width: *calc(100%* - *30px)*;

box-sizing: *border-box*;

font-size: *16px*;

}

**input**[type="text"] {

background-color: *#fff*;

}

**input**[type="submit"] {

background-color: *#795548*;

color: *white*;

border: *none*;

font-weight: *bold*;

cursor: *pointer*;

outline: *none*;

transition: *background-color* *0.3s,* *box-shadow* *0.3s*;

}

**input**[type="submit"]*:hover* {

background-color: *#5d4037*;

}

**input**[type="submit"]*:focus* {

box-shadow: *0* *0* *8px* *rgba(121,* *85,* *72,* *0.6)*;

}

**label** {

font-size: *18px*;

color: *#795548*;

font-weight: *normal*;

margin: *15px* *0* *5px*;

}

/\* Ripple Effect \*/

@**keyframes** **ripple** {

0% {

transform: *scale(0)*;

opacity: *1*;

}

*100*% {

transform: *scale(4)*;

opacity: *0*;

}

}

**input**[type="submit"]*::after* {

content: *''*;

display: *block*;

position: *absolute*;

width: *100%*;

height: *100%*;

top: *0*;

left: *0*;

pointer-events: *none*;

background-image: *radial-gradient(circle,* *#fff* *10%,* *transparent* *10.01%)*;

background-repeat: *no-repeat*;

background-position: *50%*;

transform: *scale(10,10)*;

opacity: *0*;

transition: *transform* *.5s,* *opacity* *1s*;

}

**input**[type="submit"]*:active::after* {

transform: *scale(0,0)*;

opacity: *.2*;

transition: *0s*;

}

**p** {

color: *#333*;

font-size: *24px*;

line-height: *1.6*;

}

*.result-container* {

background-color: *#FFFFFF*;

padding: *30px*;

border-radius: *4px*;

display: *inline-block*;

box-shadow: *0* *2px* *4px* *rgba(0,0,0,0.2)*;

max-width: *400px*;

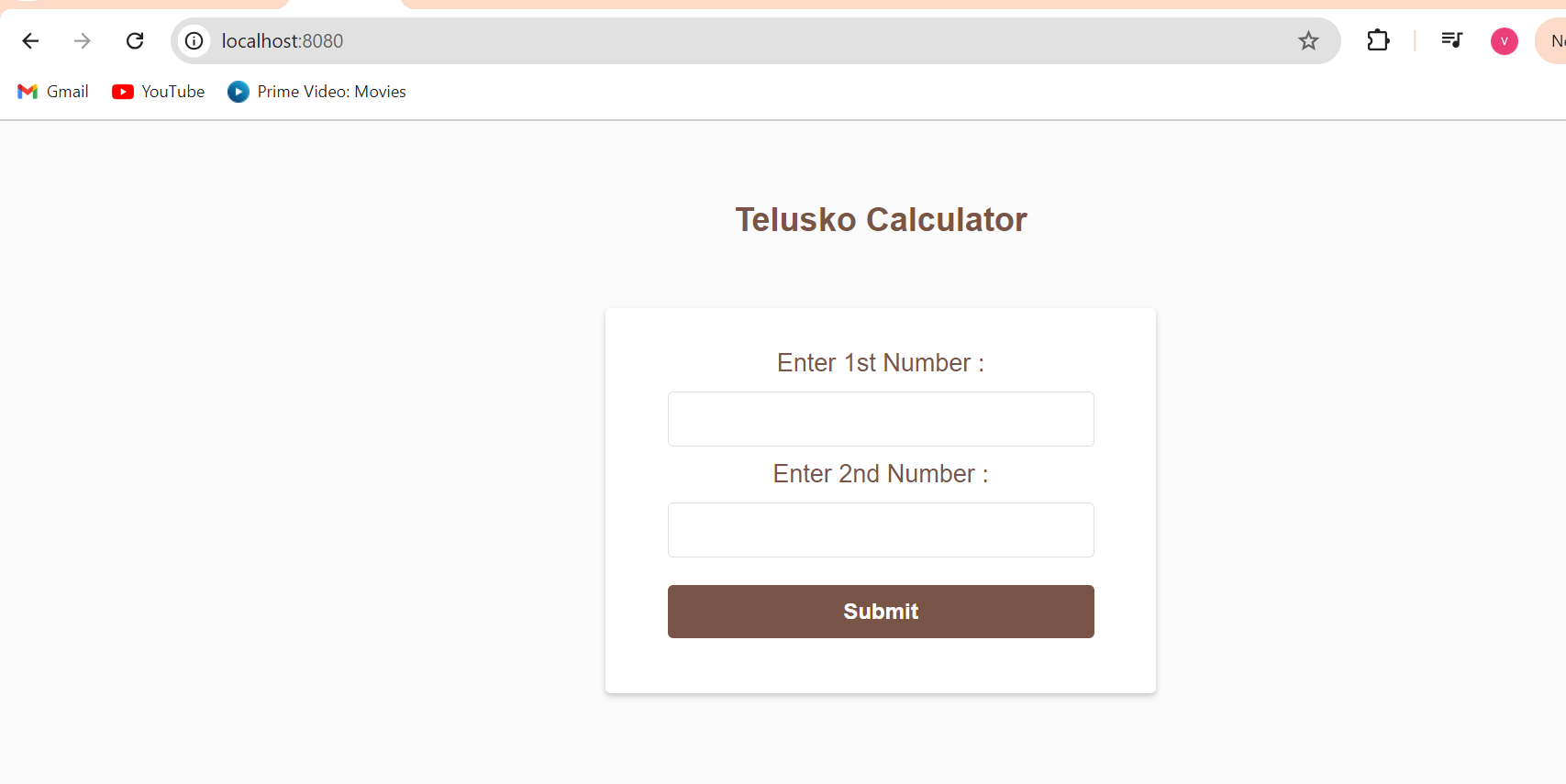
width: *100%*;

box-sizing: *border-box*;

margin-top: *20px*;

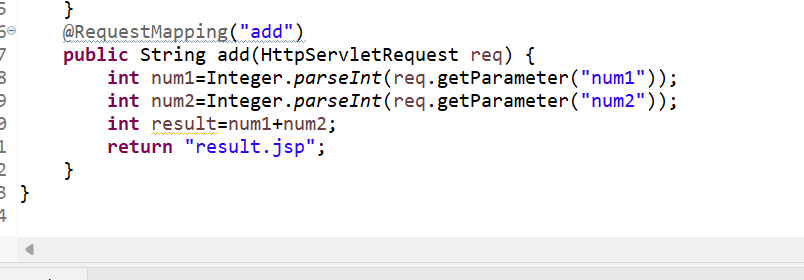
}

Pager created



**10. Accepting data in servlet way**

The HomeController's **add()** method receives the form data using **HttpServletRequest:**

****

This method extracts the values of num1 and num2 from the request, calculates the result, and returns the JSP page to be displayed.

**11. Display Data on Result page.**

Currently, the result.jsp does not display the calculated result. To display the result, you will need to modify the JSP page and possibly store the result in a session or request attribute.

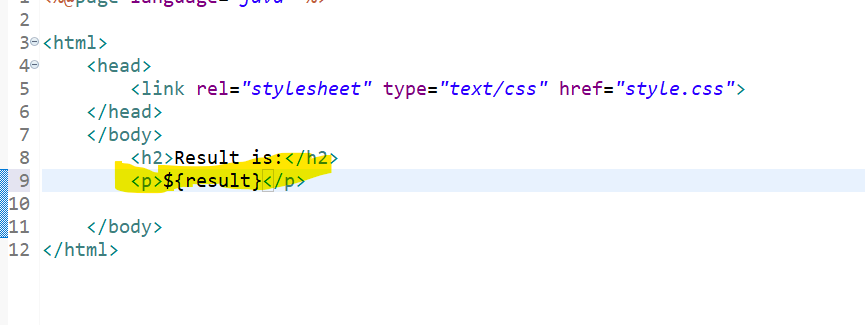
**11.1 Modifying for Session Management**

You can modify the **add()** method to store the result in the request object, which will be passed to **the result.jsp page**:

****

**11.2Displaying the Result in result.jsp**

Update your result.jsp to display the calculation result:

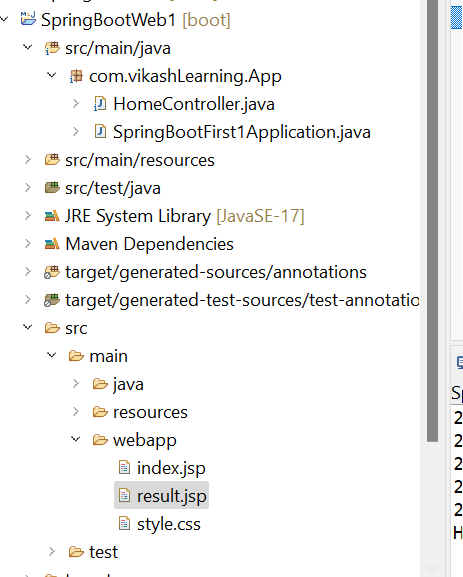
****

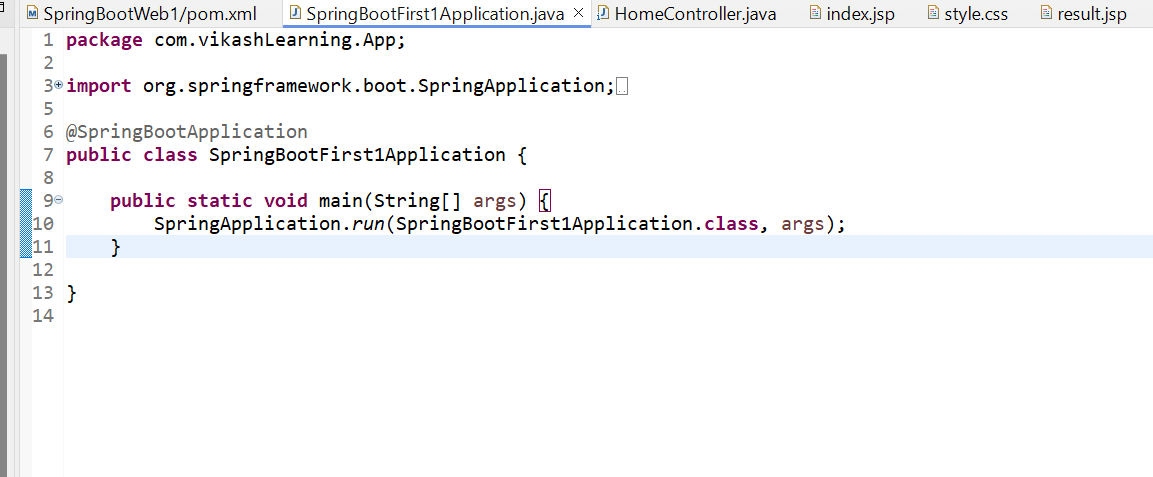
**11.3 Flow Explanation:**

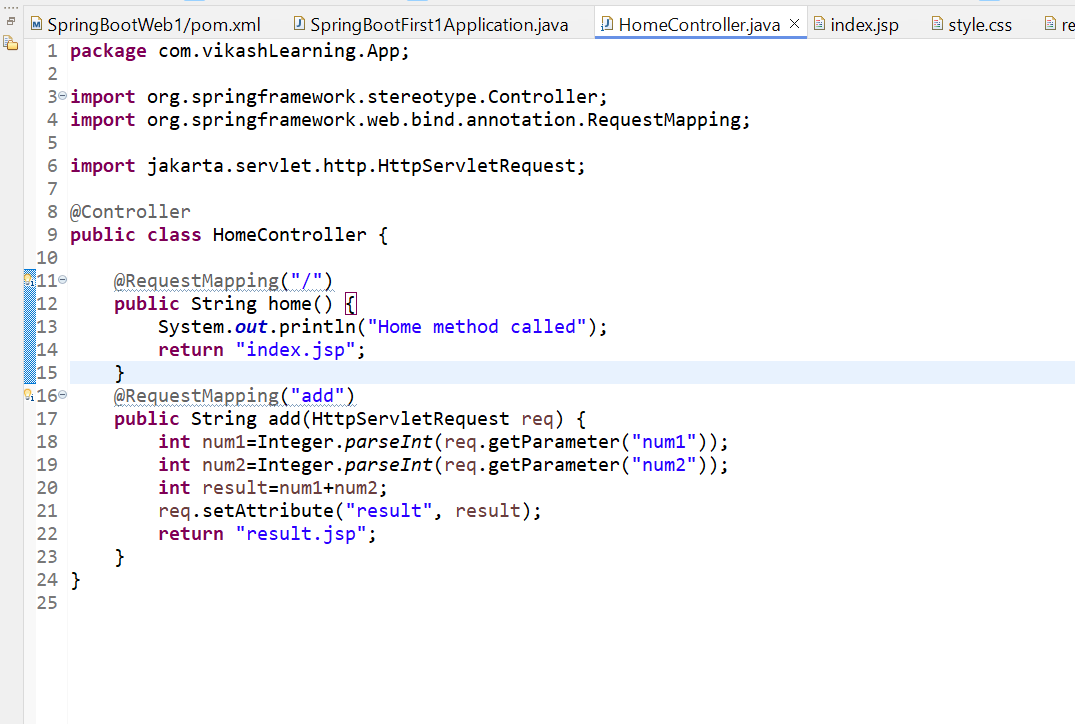
* Sending Data to Controller: The form in **index.jsp** sends data (num1 and num2) to the controller using a **GET** request.
* Accepting Data in **Controller (Servlet**): The controller's add() method processes the request, extracts the form data, performs the addition, and stores the result in the request.
* Displaying Data on the Result Page: The **result.jsp** page retrieves the result from the request object and displays it.
* This completes the flow from submitting the form, processing data in the controller, and displaying the result on the result page.

So here it the full code

Structure









**body** {

font-family: *'Roboto',* *sans-serif*;

background-color: *#FAFAFA*;

color: *#333*;

text-align: *center*;

padding: *50px*;

}

**h2** {

color: *#795548*;

margin-bottom: *30px*;

}

**form** {

background-color: *#FFFFFF*;

padding: *30px*;

border-radius: *4px*;

display: *inline-block*;

box-shadow: *0* *2px* *4px* *rgba(0,0,0,0.2)*;

max-width: *400px*;

width: *100%*;

box-sizing: *border-box*;

margin-top: *20px*;

}

**input**[type="text"]**,** **input**[type="submit"] {

border: *1px* *solid* *#E0E0E0*;

border-radius: *4px*;

padding: *10px* *15px*;

margin: *10px* *0*;

width: *calc(100%* - *30px)*;

box-sizing: *border-box*;

font-size: *16px*;

}

**input**[type="text"] {

background-color: *#fff*;

}

**input**[type="submit"] {

background-color: *#795548*;

color: *white*;

border: *none*;

font-weight: *bold*;

cursor: *pointer*;

outline: *none*;

transition: *background-color* *0.3s,* *box-shadow* *0.3s*;

}

**input**[type="submit"]*:hover* {

background-color: *#5d4037*;

}

**input**[type="submit"]*:focus* {

box-shadow: *0* *0* *8px* *rgba(121,* *85,* *72,* *0.6)*;

}

**label** {

font-size: *18px*;

color: *#795548*;

font-weight: *normal*;

margin: *15px* *0* *5px*;

}

/\* Ripple Effect \*/

@**keyframes** **ripple** {

0% {

transform: *scale(0)*;

opacity: *1*;

}

*100*% {

transform: *scale(4)*;

opacity: *0*;

}

}

**input**[type="submit"]*::after* {

content: *''*;

display: *block*;

position: *absolute*;

width: *100%*;

height: *100%*;

top: *0*;

left: *0*;

pointer-events: *none*;

background-image: *radial-gradient(circle,* *#fff* *10%,* *transparent* *10.01%)*;

background-repeat: *no-repeat*;

background-position: *50%*;

transform: *scale(10,10)*;

opacity: *0*;

transition: *transform* *.5s,* *opacity* *1s*;

}

**input**[type="submit"]*:active::after* {

transform: *scale(0,0)*;

opacity: *.2*;

transition: *0s*;

}

**p** {

color: *#333*;

font-size: *24px*;

line-height: *1.6*;

}

*.result-container* {

background-color: *#FFFFFF*;

padding: *30px*;

border-radius: *4px*;

display: *inline-block*;

box-shadow: *0* *2px* *4px* *rgba(0,0,0,0.2)*;

max-width: *400px*;

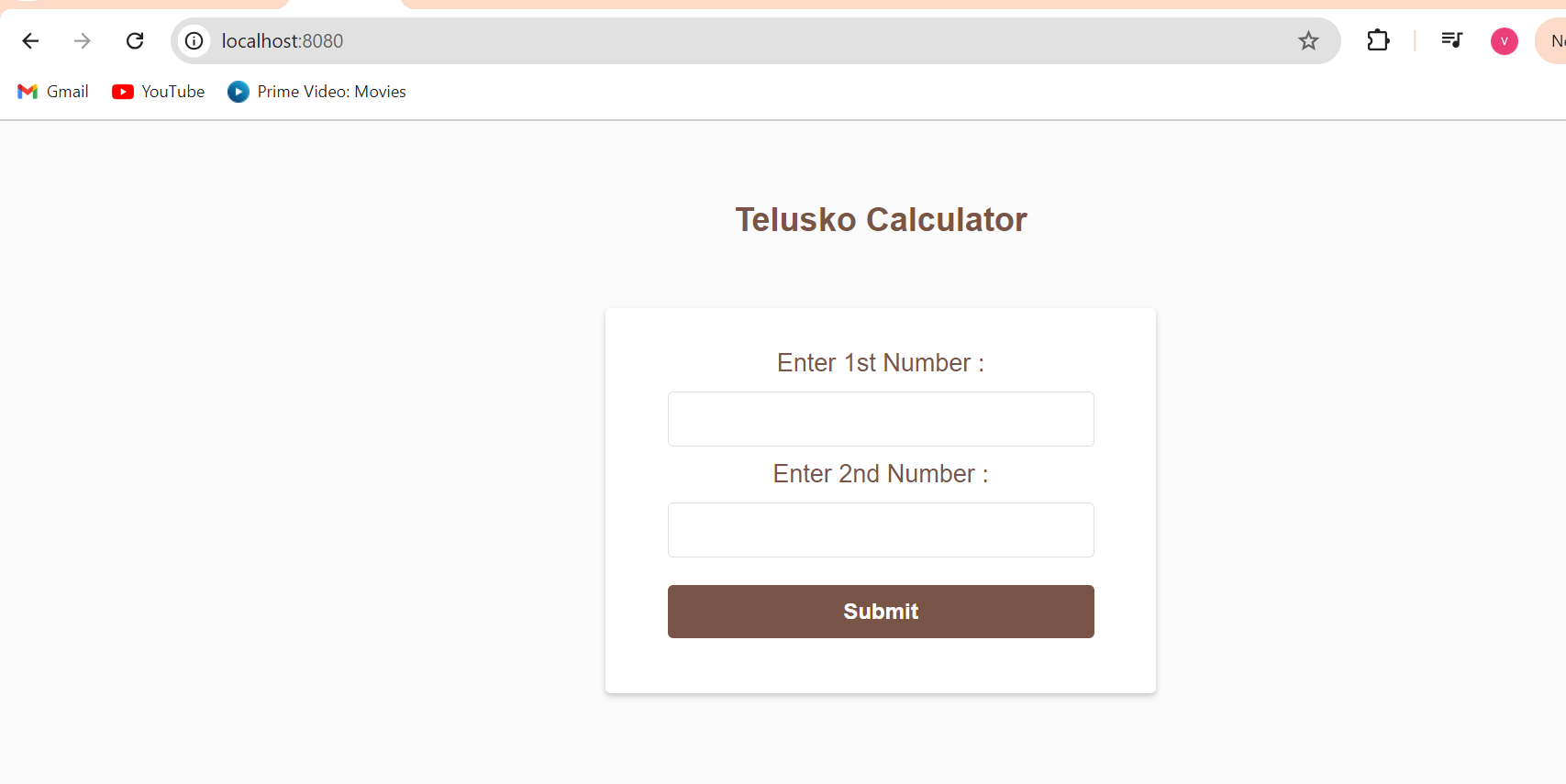
width: *100%*;

box-sizing: *border-box*;

margin-top: *20px*;

}

Page created





This is resulting page and observe the URL.

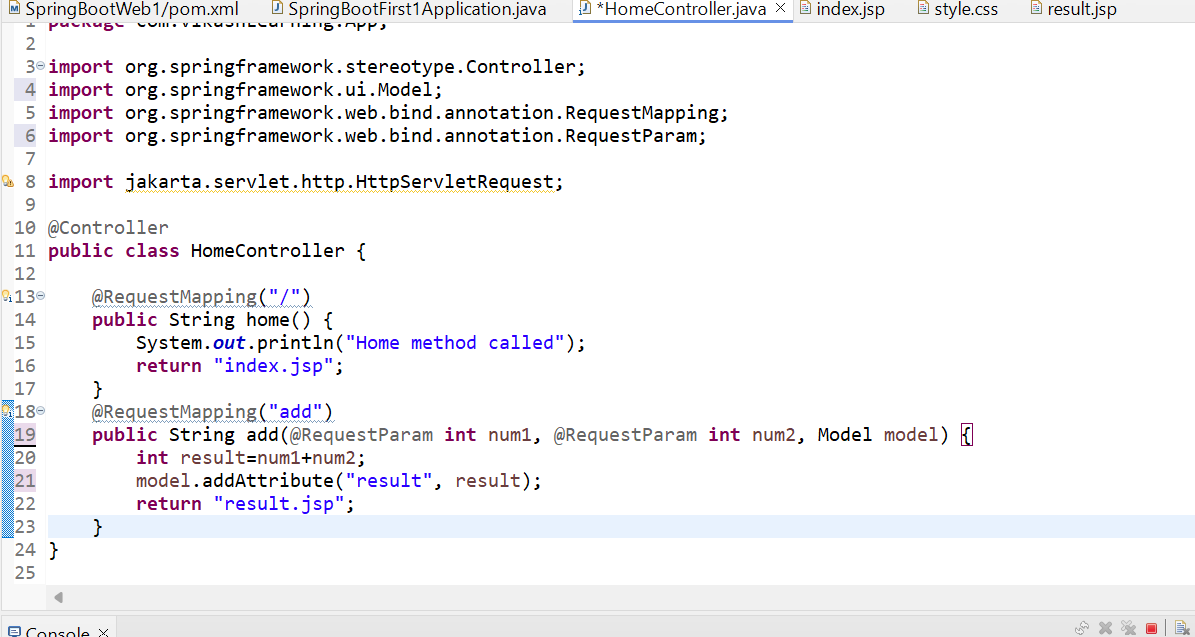


**12. Request param and Model Object**

To refactor the code to use Spring's **@RequestParam** instead of **HttpServletRequest**, you can directly extract the parameters from the URL in a more Spring-friendly way. Here's how you can modify the code:

**12.1 HomeController (Using @RequestParam):**

You can replace HttpServletRequest with @RequestParam to retrieve the values directly from the query string.

****

* @RequestParam: It automatically binds the request parameter (num1 and num2) to method arguments.
* Model: Spring’s Model is used to pass data to the view (JSP in this case).

**12.2. index.jsp (No Changes Required):**

The form stays the same. Spring will automatically map the form fields to the @RequestParam.

**12.3 result.jsp (No Changes Required)**:

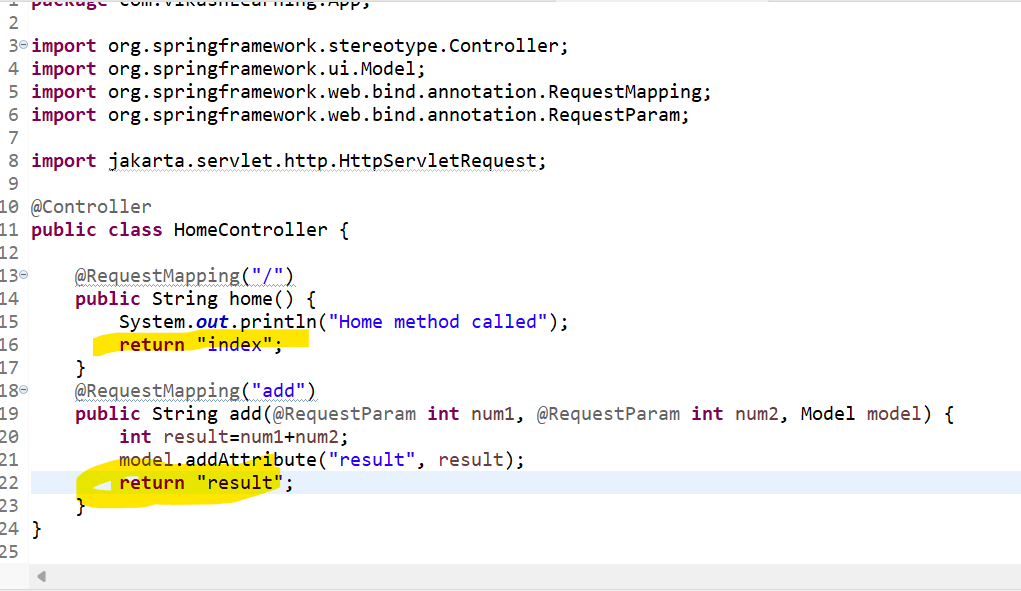
**12.4 Flow Explanation:**

1. **Sending Data to Controller**: When the form in index.jsp is submitted, it sends a GET request to /add with num1 and num2 as query parameters.
2. **Processing Data with @RequestParam**: The controller's add() method uses @RequestParam to directly capture the values from the URL, calculates the result, and stores it in the Model.
3. **Displaying Data on Result Page**: The result is displayed using ${result} in result.jsp, which retrieves the value from the Model.

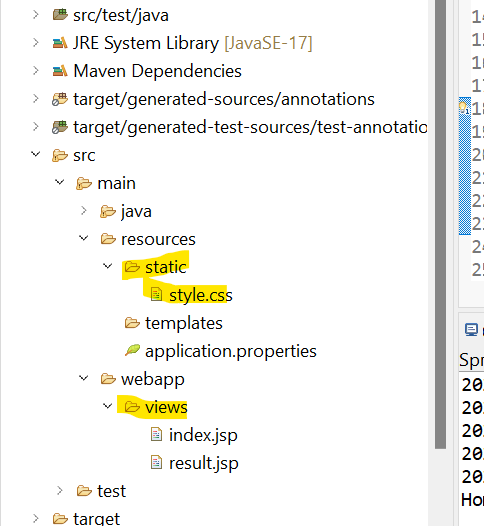
This way, you no longer rely on the HttpServletRequest and can use Spring’s more declarative way of handling request parameters.

**13. Setting Prefix and suffix**

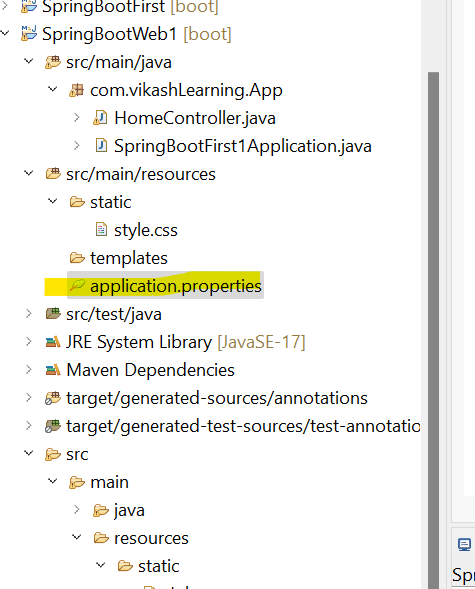
**13.1 Remove .jsp extension from result.jsp and index.jsp**

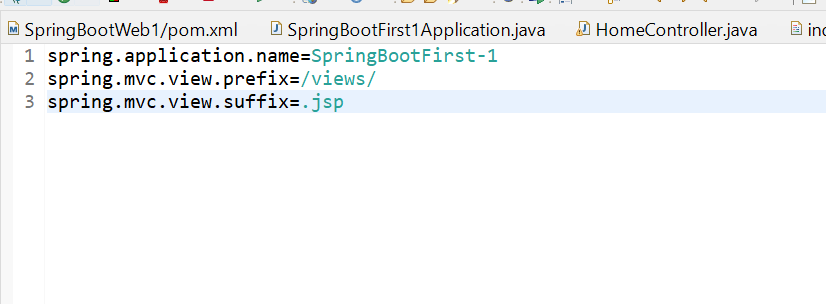
****

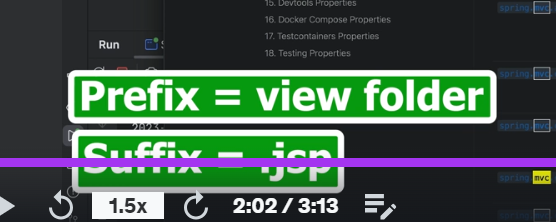
**13.2 Move Index.jsp and result.jsp file in src/mian/webpage/views/index.jsp ,result.jsp folder and style.css in src/mian/resources/static/styl.css**

****

**13.3 In src/main/resources/application.properties update below texts.**

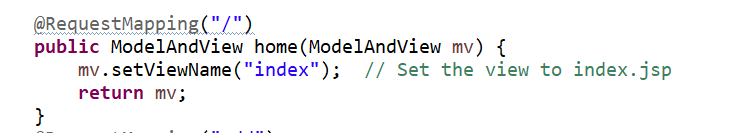
****

****



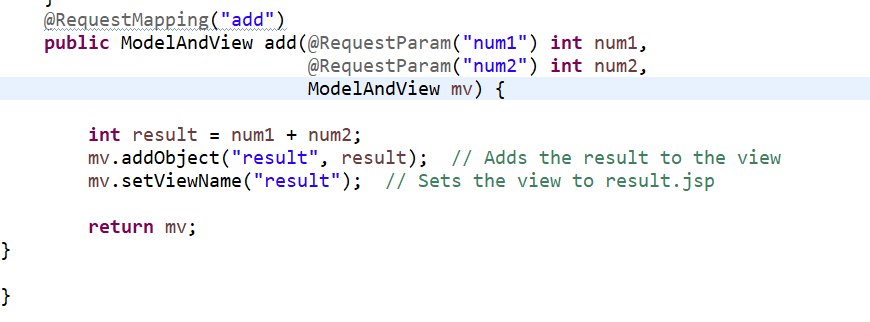
**14. ModelAndView**

**14.1 Home Method (/):**

****

* Purpose: This method handles the root (/) URL request.
* mv.setViewName("index"): Sets the view to index.jsp. It doesn't pass any dynamic data, just renders the view.

**14.2 Add Method (/add):**

****

* Purpose: This method processes /add requests.
* @RequestParam: Captures num1 and num2 from the request parameters.
* mv.addObject("result", result): Passes the calculated result to the view.
* mv.setViewName("result"): Specifies that the result.jsp page should render.

Both methods use ModelAndView to control both the view and any data being sent to the view, following a consistent pattern.

14.3 Flow of add() Method:

* Request: When a user submits a form to /add, the method captures the form parameters using @RequestParam. For example, num1=5 and num2=7 are passed.
* Calculation: The method adds the two numbers: int result = num1 + num2;.
* mv.addObject("result", result**):**
  + **"**result" is the key (string name) that will be used in the view (in result.jsp).
  + result is the value (the sum of num1 and num2).
  + This method passes the result to the view. Essentially, you are telling the view (JSP file) that the result value is available under the key "result".
* mv.setViewName("result"): This sets the view to result.jsp, telling Spring to render this JSP page.

**14.3 How It Works in JSP (result.jsp):**

In result.jsp, you can access the result value using ${result}. This retrieves the sum that was added to the model through mv.addObject("result", result).

**Example Flow:**

1. User submits a form: /add?num1=5&num2=7
2. Spring captures the num1 and num2 values.
3. The method calculates 5 + 7 = 12 and stores it in result.
4. mv.addObject("result", 12) passes 12 to result.jsp.
5. mv.setViewName("result") tells Spring to render the result.jsp page with ${result} displaying.

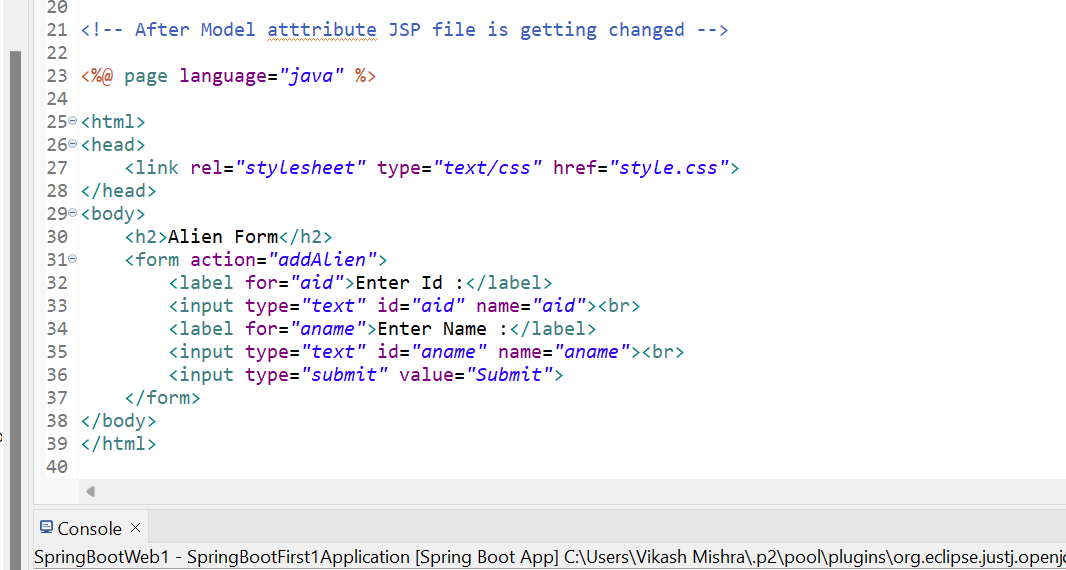
This makes the result value accessible in the view, and Spring will render it on the result page.

**15.Need of Model Attribute and Use of model attribute**

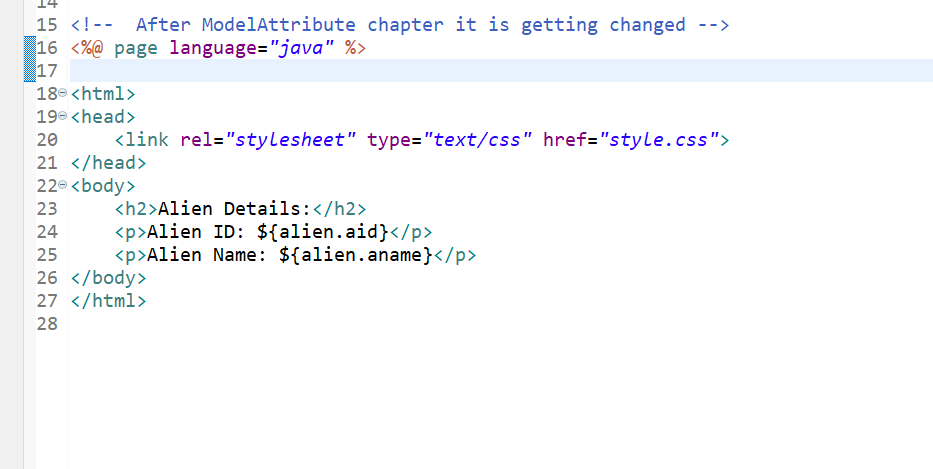
* 1. **Need of @ModelAttribute:**
* When dealing with forms that map multiple fields to complex objects, like the Alien class with aid and aname, **@ModelAttribute** simplifies binding form data to an object. It helps to automatically bind incoming form data to the object fields, reducing boilerplate code for handling multiple **@RequestParam.**
  1. **Use of @ModelAttribute:**
* The @ModelAttribute annotation in Spring automatically binds form data to a model object (e.g., Alien). It also adds the model to the view so it can be accessed in the HTML files without manual data extraction.

**15.3 Updated codes**

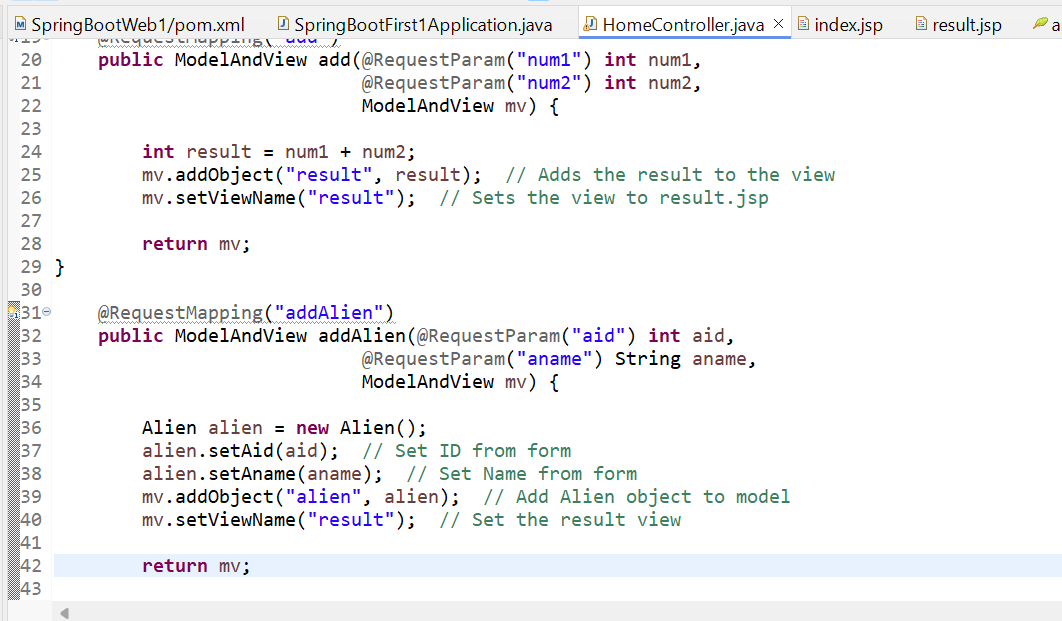
**15.3.1 Index.jsp**

****

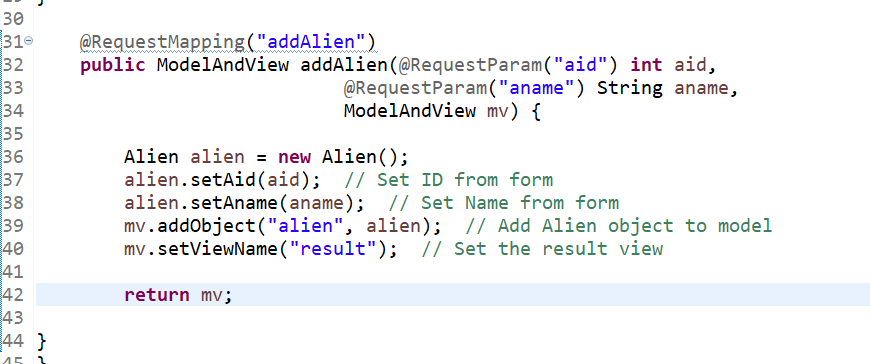
**15.3.2 Result.jsp**

****

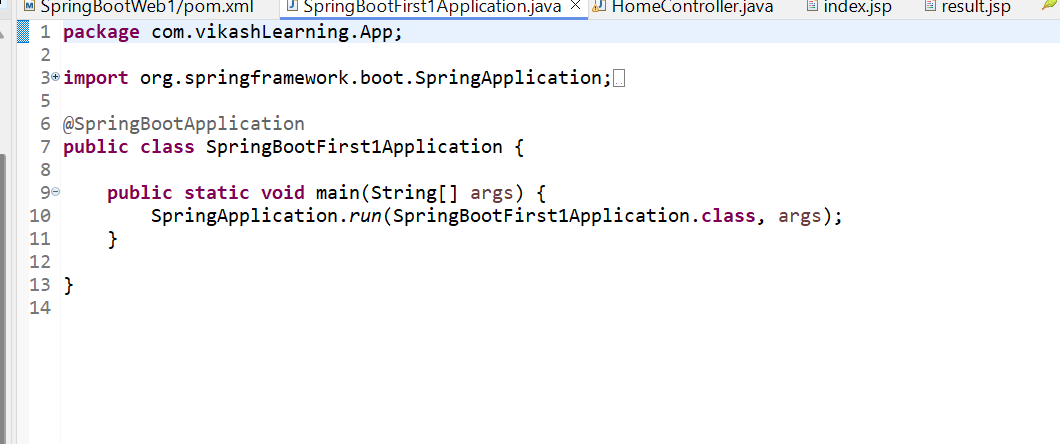
**15.3.3 HomeController.java**

****

**15.3.4 Especially focus on below part**

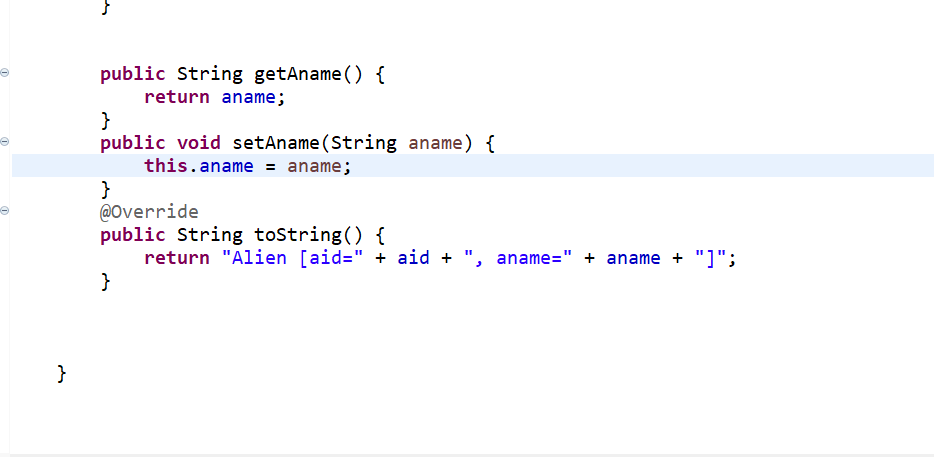
****

**15.3.5 SpringBootApplication.java**

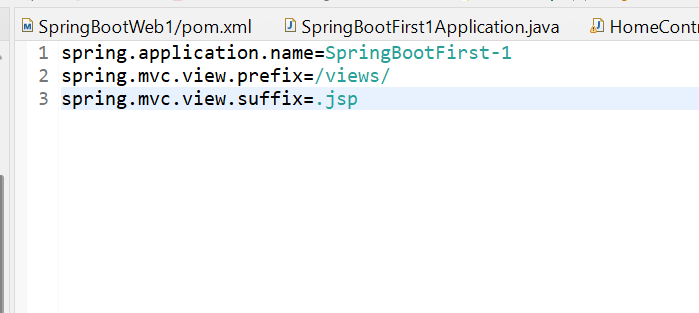
****

**15.3.6 Add one class Alient.java**

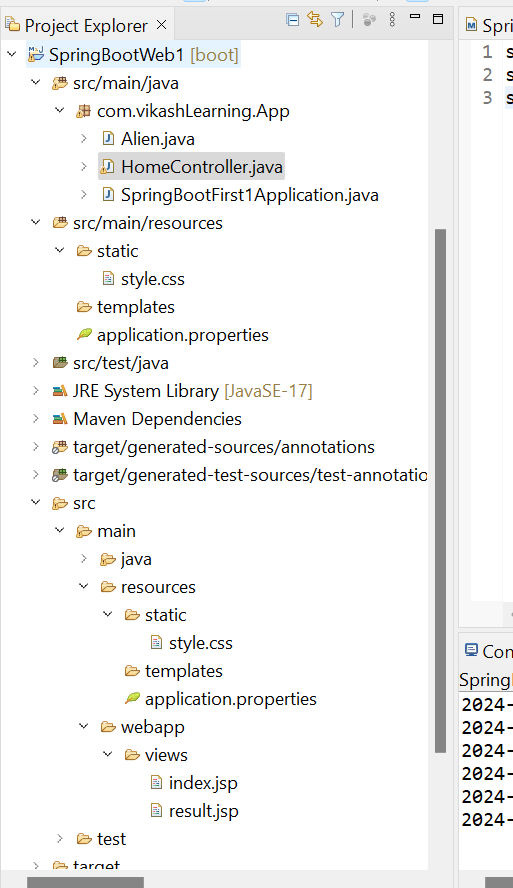




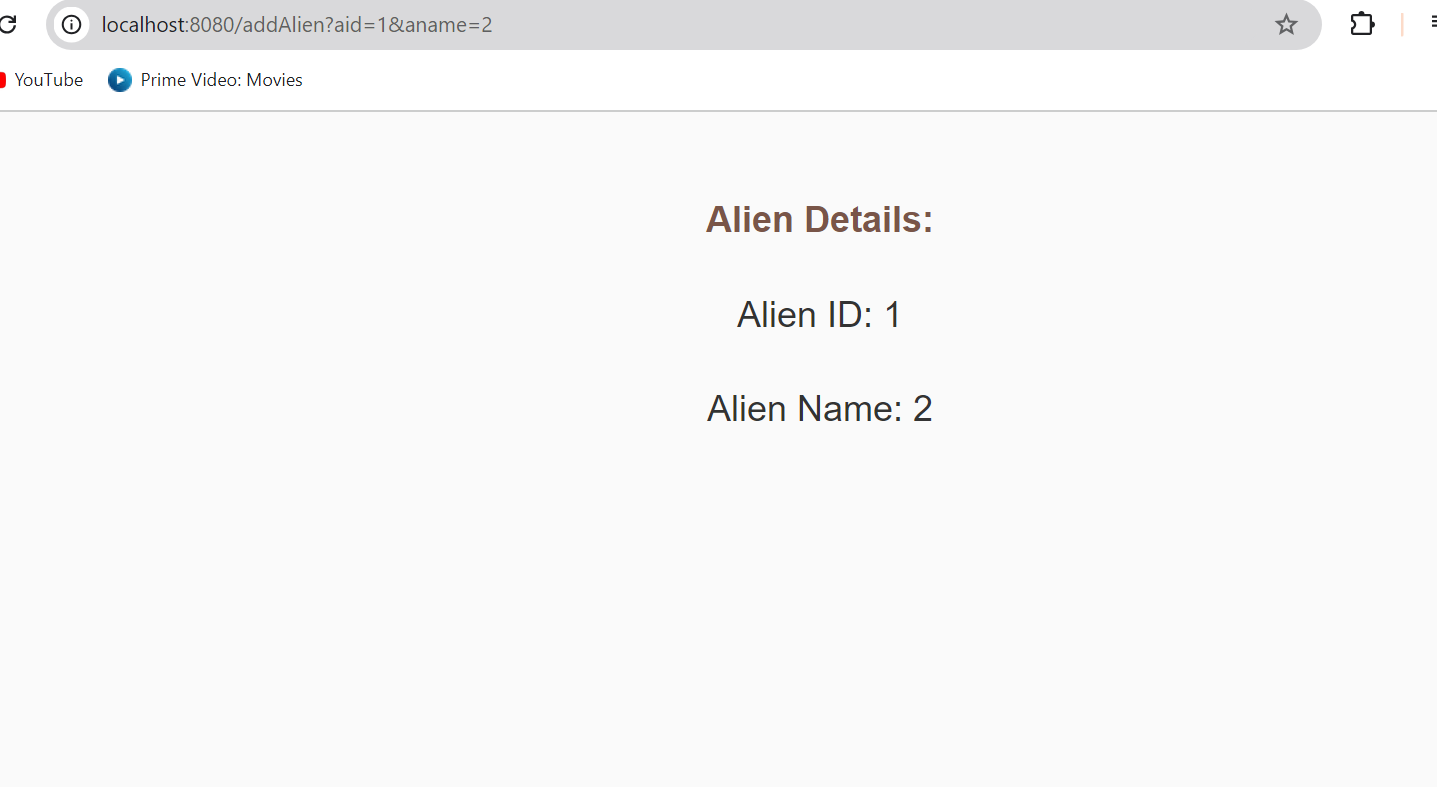
**15.3.7 Application.properties**



**15.3.8 Structure**



**15.3.9 Result page**

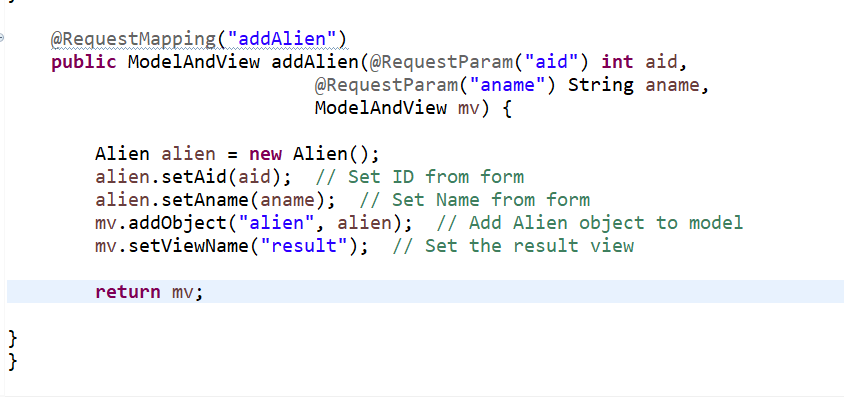


**15.4 Now the difference**

**15.4.1 Using @RequestParam**

In this version, the form data (aid, aname) is captured using @RequestParam and then manually set to the Alien object.

**15.4.1.1 HomeController.java (With @RequestParam):**



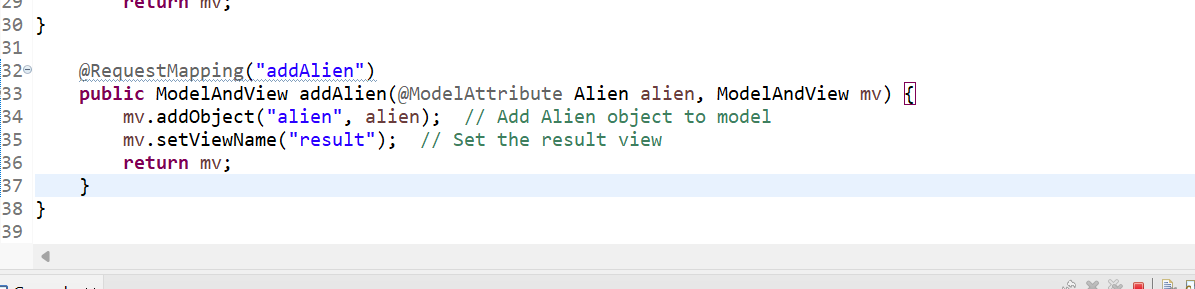
**15.4.1.2 Explanation:**

* **@RequestParam("aid") and @RequestParam("aname")**: Spring extracts the values from the form data using @RequestParam, then we manually set them into the Alien object using alien.setAid() and alien.setAname().
* **Form Submission**: Data is sent via URL parameters like addAlien?aid=1&aname=John.

**15.4.2 Using @ModelAttribute (Refactor)**

Now, we'll refactor this to use @ModelAttribute, which automatically binds the form data to the Alien object without needing manual setting.

#### **15.4.2.1 HomeController.java (With @ModelAttribute):**



**15.4.2.2 Explanation of @ModelAttribute:**

* **@ModelAttribute Alien alien**: Spring automatically binds the form fields (aid, aname) directly to the Alien object's fields.
* No need to manually call setAid() or setAname(). The form data is bound to the object and made available for use.

**15.4.3 Key Difference:**

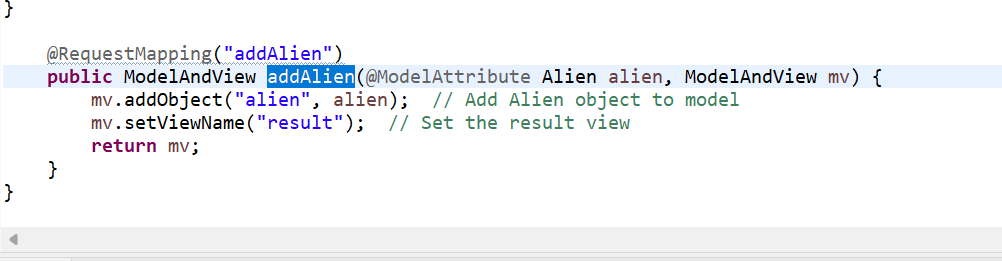
* **@RequestParam** requires you to manually handle each field.
* **@ModelAttribute** binds the entire form to the object, simplifying the process.

**15.4.4 Example Flow:**

* **With @RequestParam**:
  1. Form sends aid and aname via request parameters.
  2. Controller maps them manually to the Alien object.
* **With @ModelAttribute**:
  1. Form data is directly bound to the Alien object without manual mapping.

This illustrates how @ModelAttribute simplifies data binding when dealing with objects, compared to @RequestParam.

**15.5 Updated add alien method**



**15.5.1 Explanation:**

* **Automatic Data Binding**: Spring automatically binds the form data to the Alien object using the field names from the form that match the Alien class properties (aid and aname).
* **ModelAttribute Optional**: Since the parameter is an object (like Alien), Spring automatically assumes @ModelAttribute is being used. So explicitly using @ModelAttribute is optional.
* **View Name**: It returns the view name (result), which corresponds to result.jsp or result.html (depending on your setup).

This version simplifies your method by omitting manual mapping.

**15.5.2 Workflow Explanation of Updated Code:**

1. **Form Submission**: The user submits a form containing aid and aname fields, which correspond to the Alien object's properties.
2. **Data Binding**: Spring automatically binds the form data (via input fields aid and aname) to the Alien object because the form fields have the same names as the Alien class properties.
3. **Controller**: In the addAlien method, the Alien object is created and populated with the form data.
4. **View Rendering**: The method returns the view name "result", leading Spring to render the result.jsp page with access to the Alien object.

**15.6 When to Use @ModelAttribute:**

* **Optional Usage**:
  + **When**: If your form input names exactly match the fields of the object (like Alien), you don’t need to explicitly add @ModelAttribute. Spring will automatically bind the form fields to the object.
  + **Example**: public String addAlien(Alien alien) will work if the form fields match Alien's fields (aid, aname).
* **Mandatory Usage**:
  + **When**: You must use @ModelAttribute("aliasName") if you want to bind the form data to an object **with a different alias** in the view or when you need to pre-populate a form with data.
  + **Example**: public String addAlien(@ModelAttribute("alien") Alien alien) when you want to refer to it as "alien" in the view.

**15.7 Summary:**

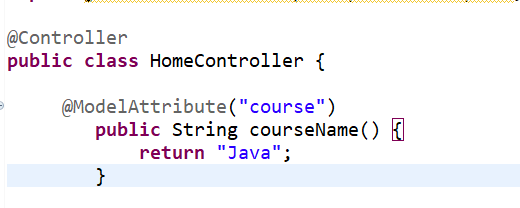
* **Use @ModelAttribute** if:
  + You need to give the object a specific alias.
  + You want to pre-populate data in forms.

**16. Model Pre-population with @ModelAttribute**

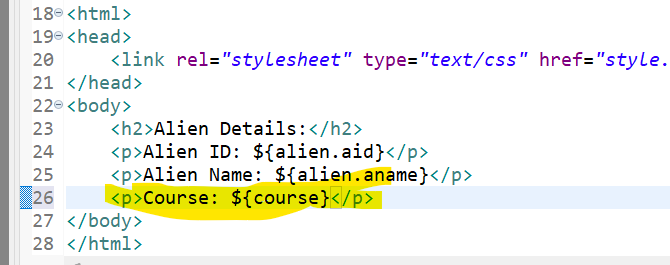
This allows automatic data population into the model before the controller method runs, making that data available to views across the application. It's useful for injecting common values like page titles, course names, or other shared data into multiple views without explicitly adding them in every controller method.

**16.1 Update Homecontroller. Java file**

Add the below snipped code



**16.2 Updated result.jsp:**



**16.3 Explanation:**

* **@ModelAttribute("course")**: Adds the string "Java" to the model with the key "course".
* **${course}**: In the JSP, ${course} displays "Java" on the result page without any additional code in the method handling logic.

Now, whenever result.jsp is rendered, it will also display the course data alongside the Alien details.

**16.4 When and Why to Use:**

* **Pre-populate a Model with Data**: This method adds a value ("Java") to the model with the key "course". When any view is rendered, the model will automatically contain this key-value pair, and "Java" can be accessed in the view using ${course}.
* **Useful in Forms and Views**: It is often used when you want to bind static or pre-defined data to your views before rendering.

**16.5 Example Use Case:**

* You might use @ModelAttribute in an educational platform where you need to show the current course name (like "Java") on multiple pages without manually passing it every time.

In summary, @ModelAttribute can be used to populate the model with common, reusable data that can be accessed across multiple views.