Introduction to Web Applications

Static Content: Web pages where content remains constant unless manually updated by a developer. Examples include plain HTML, CSS, and JavaScript files.

Dynamic Content: Web pages where the content is generated in real-time based on user input or server-side logic. Example: Content fetched from a database using server-side scripts.

Data Handling in Web Apps:

- **Server-side Processing**: Operations like querying databases, performing business logic, and generating responses handled on the server. Frameworks like Spring simplify server-side processing.

Servlet Container: A runtime environment for Java servlets that manages their lifecycle and handles HTTP requests.

Example: **Apache Tomcat**.

Creating a Basic Servlet Project

.jar (Java ARchive): A package of Java classes, libraries, and metadata bundled together for use in applications.

.war (Web Application Archive): A specialized archive format for web applications that includes servlets, JSP files, and resources.

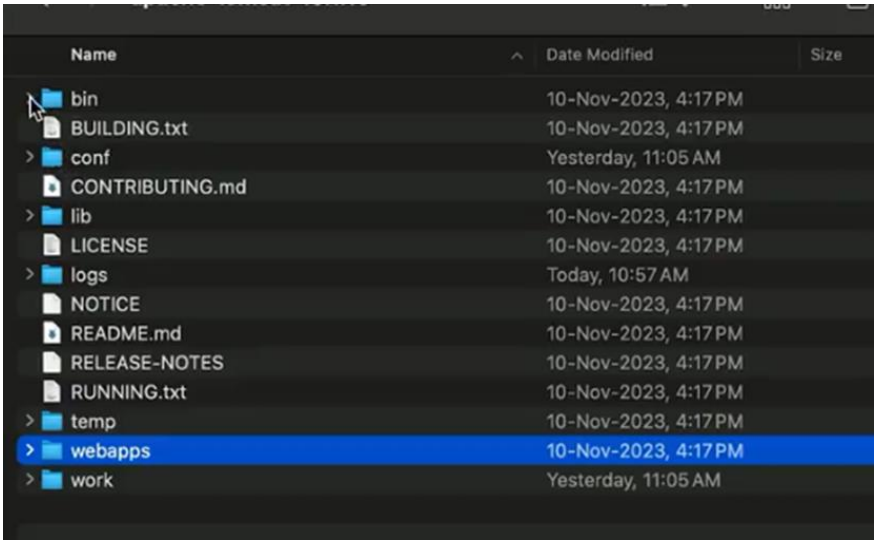
Tomcat Server:

- **Standalone Tomcat**: Requires manual deployment of .war files in the webapps directory.
- **Embedded Tomcat**: Integrated into your application via Maven dependencies, making it easier to manage.

Java Servlet API: Provides the standard interface for building servlets that handle HTTP requests and responses.

1) We need a Tomcat server

Download Tomcat 10

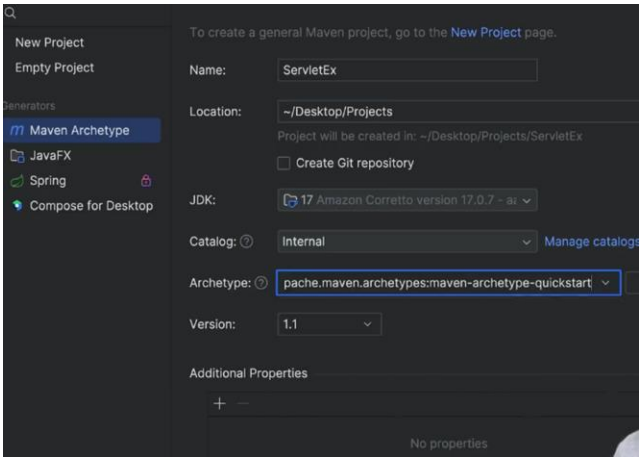


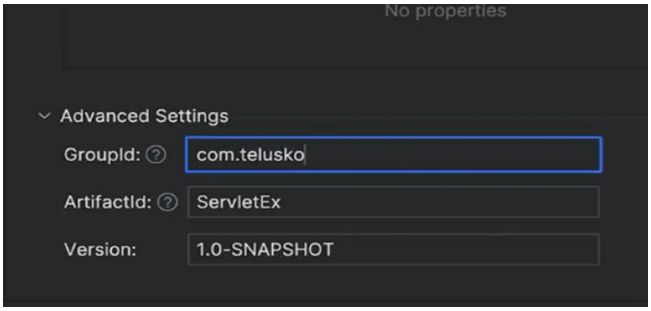
Name	Date Modified	Size
bin	10-Nov-2023, 4:17 PM	
BUILDING.txt	10-Nov-2023, 4:17 PM	
conf	Yesterday, 11:05 AM	
CONTRIBUTING.md	10-Nov-2023, 4:17 PM	
lib	10-Nov-2023, 4:17 PM	
LICENSE	10-Nov-2023, 4:17 PM	
logs	Today, 10:57 AM	
NOTICE	10-Nov-2023, 4:17 PM	
README.md	10-Nov-2023, 4:17 PM	
RELEASE-NOTES	10-Nov-2023, 4:17 PM	
RUNNING.txt	10-Nov-2023, 4:17 PM	
temp	10-Nov-2023, 4:17 PM	
webapps	10-Nov-2023, 4:17 PM	
work	Yesterday, 11:05 AM	

1. webapps: put our war file
2. In .bin folder we have startup.sh and shutdown.sh.

2)Embeded tomcat

Create a Maven project.



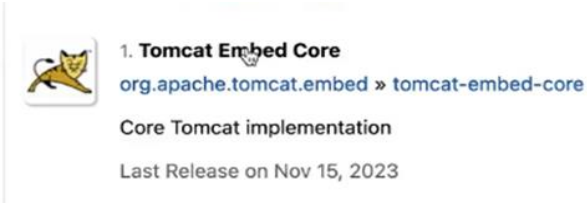


You need to add two dependencies.

To get dependencies: **[MVN Repository]**

1. Java Servlet API

2. **Tomcat Embeded Core**



Running an Embedded Tomcat Server

Tomcat Configuration:

- **Tomcat Instance**: new Tomcat() initializes an embedded Tomcat server.
- **Persistent Server**: tomcat.getServer().await() ensures the server keeps running and listens for requests.

HTTPServlet:

service() Method: Handles incoming HTTP requests and generates appropriate responses.

```
Tomcat tomcat = new Tomcat();
tomcat.start();
```

Persistent Server

Ensure the server keeps running:
tomcat.getServer().await();

Steps:

1. Create HelloServlet class
2. Making a Servlet by extending HttpServlet
3. Write the service(HttpServletRequest request, HttpServletResponse response) method.

=> For servlets, you need to send requests through a browser.

=> By default, embedded Tomcat is not running.

In the main method

```
Tomcat tomcat=new Tomcat();
tomcat.start();
```

```
public class App
{
    public static void main( String[] args ) throws LifecycleException {
        System.out.println( "Hello World!" );
        Tomcat tomcat = new Tomcat();
        tomcat.start();
    }
}
```

After doing that, something happens.

```
Dec 14, 2023 10:57:11 AM org.apache.catalina.core.StandardService startInternal
INFO: Starting service [Tomcat]
Dec 14, 2023 10:57:11 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-nio-8080"]

Process finished with exit code 0
```

But still *localhost:8080 is* not running.

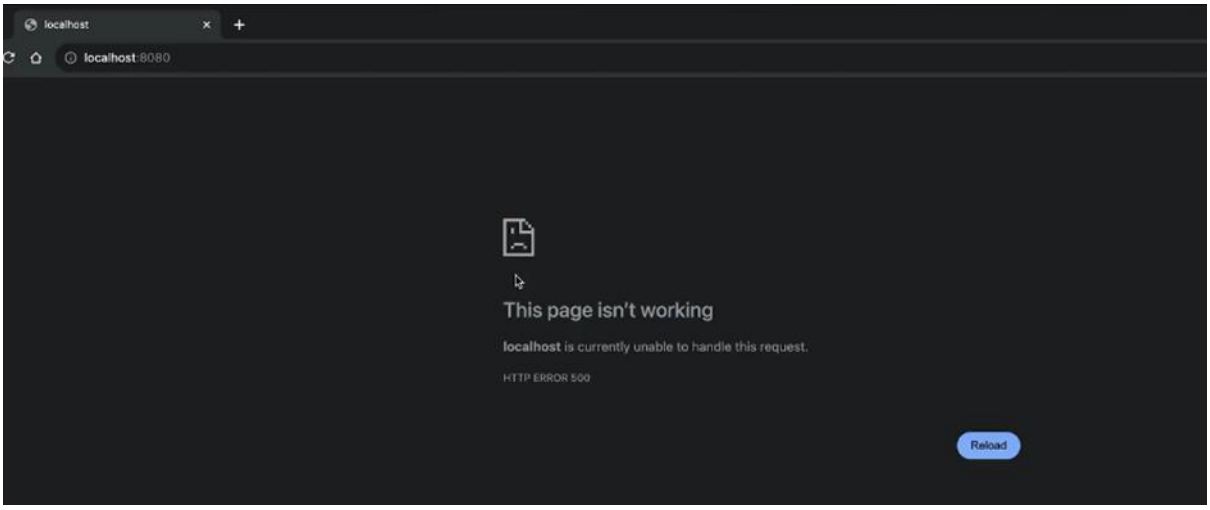
Now make Tomcat keep running.

Add tomcat.getServer().await()

```
public class App
{
    public static void main( String[] args ) throws LifecycleException {
        System.out.println( "Hello World!" );
        Tomcat tomcat = new Tomcat();
        tomcat.start();
        tomcat.getServer().await();
    }
}
```

```
Dec 14, 2023 10:58:09 AM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-nio-8080"]
Dec 14, 2023 10:58:09 AM org.apache.catalina.core.StandardService startInternal
INFO: Starting service [Tomcat]
Dec 14, 2023 10:58:09 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-nio-8080"]
```

Now we found the server does not stop.



But get some different error message (not able to handle the request).

How does Tomcat knows which page I handle?

=> Now we do mapping, which is in the next lecture.

Servlet Mapping

Servlet Mapping:

- **Annotation-based Mapping**: Using @WebServlet("/hello") for automatic URL mapping in external Tomcat.
- **Manual Mapping**: In embedded Tomcat, use methods like addServletMappingDecoded() to associate URLs with specific servlets.

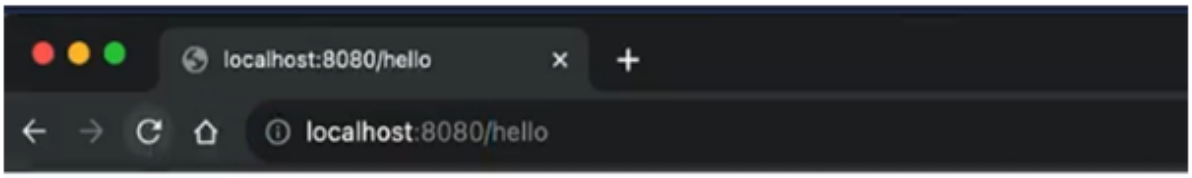
Manual Mapping for Embedded Tomcat:

```
Context context = tomcat.addContext("", null);
tomcat.addServlet("HelloServlet", new HelloServlet());
context.addServletMappingDecoded("/hello", "HelloServlet");
tomcat.start();
tomcat.getServer().await();
```

```
public class App
{
    public static void main( String[] args ) throws LifecycleException {
        System.out.println( "Hello World!" );
        Tomcat tomcat = new Tomcat();

        Context context = tomcat.addContext( contextPath: "", docBase: null);
        Tomcat.addServlet(context,  servletName: "HelloServlet", new HelloServlet());
        context.addServletMappingDecoded( s: "/hello",  s1: "HelloServlet");


        tomcat.start();
        tomcat.getServer().await();

    }
}
```

Output:

```
INFO: Starting service [Tomcat]
Dec 14, 2023 11:12:10 AM org.apache.catalina.core.StandardEngine startInterr
INFO: Starting Servlet engine: [Apache Tomcat/8.5.96]
Dec 14, 2023 11:12:10 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-nio-8080"]
In Service
```



Custom Port Configuration:

tomcat.setPort(8081) changes the default port from 8080.

Responding to Client Requests

PrintWriter: Used to write text responses to clients. For example:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<h2>Hello, World!</h2>");
```

HTTP Methods:

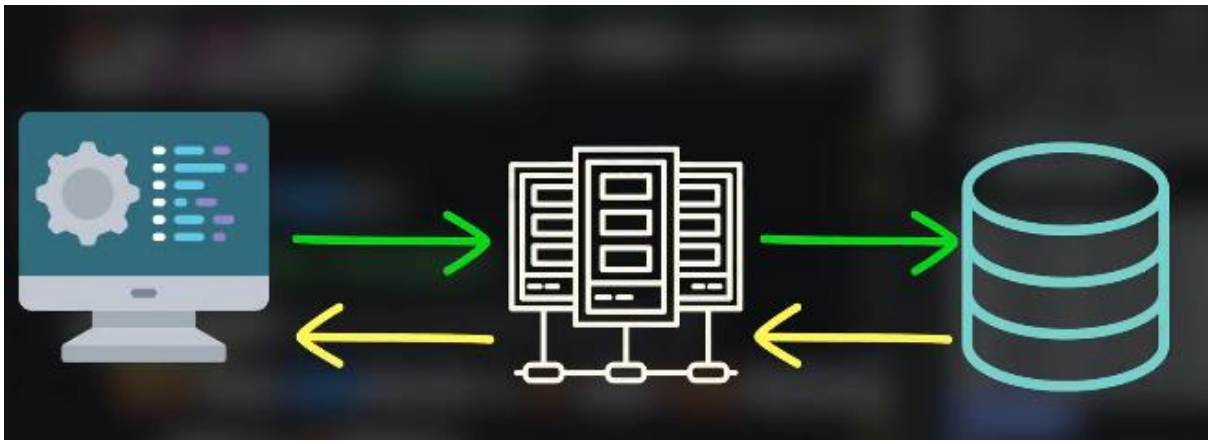
- **GET**: Retrieves data from the server without altering its state.
- **POST**: Sends data to the server for processing (e.g., form submissions).
- **PUT**: Updates existing resources.
- **DELETE**: Removes resources.

Introduction to MVC

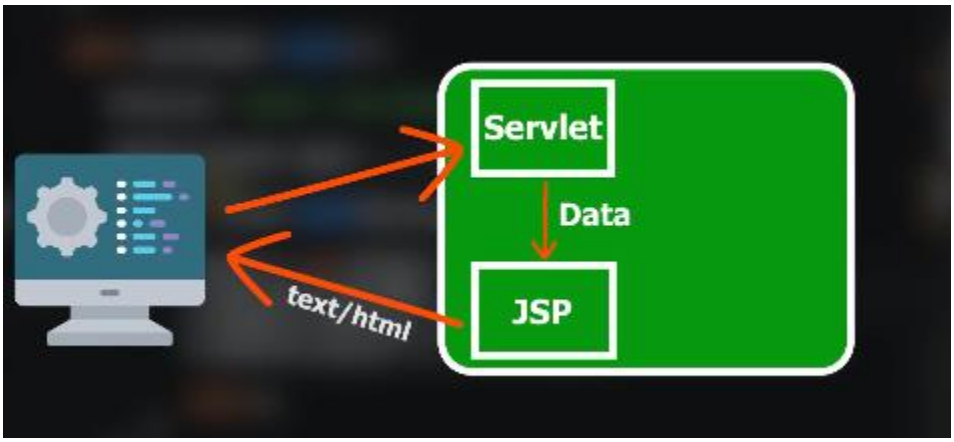
Model-View-Controller (MVC):

- **Model**: Represents application data and business logic.
- **View**: The presentation layer (e.g., JSP files).
- **Controller**: Handles user requests and interacts with the model to return responses.

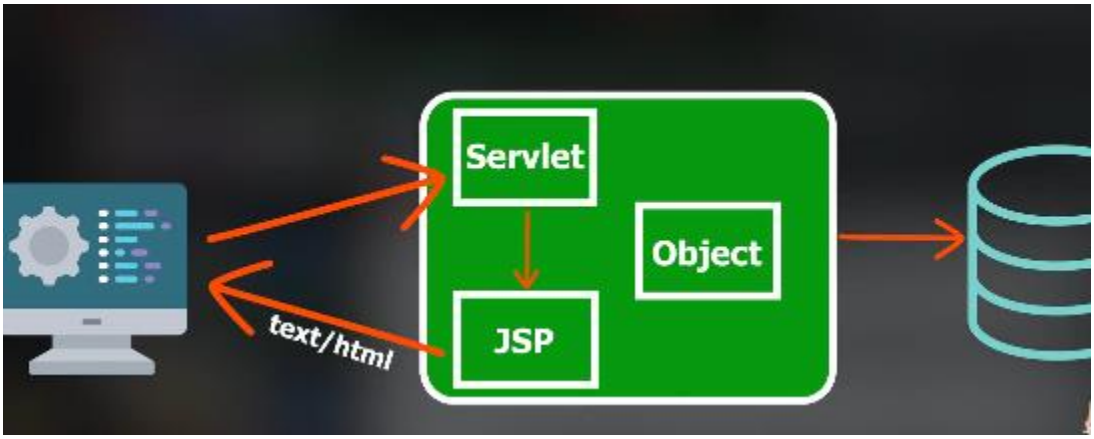
POJO (Plain Old Java Object): A simple Java class used to encapsulate data.



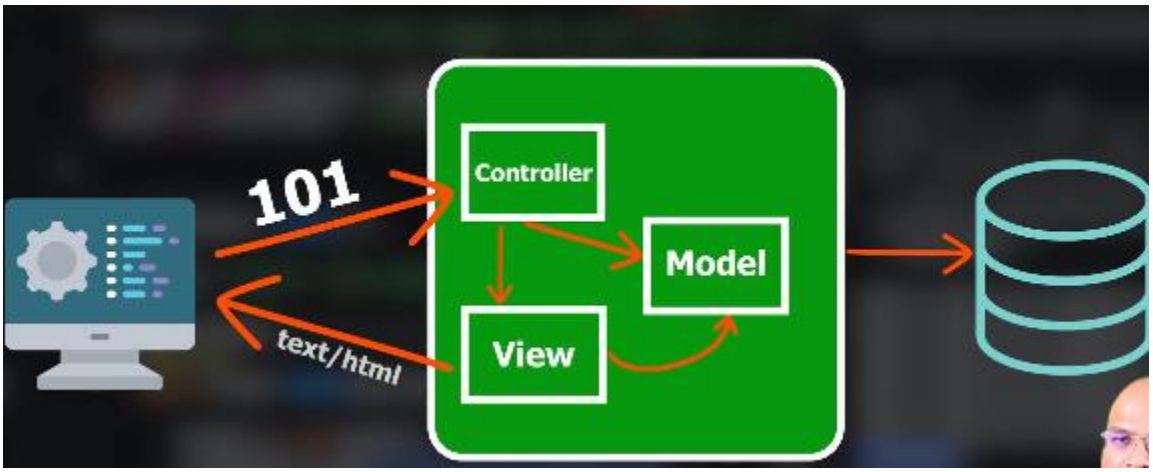
JSP - Java Servlet Page



Represent everything in object format.



MVC (Model View Controller) design pattern



A simple class is called POJO.

Create a Spring Boot Web Project

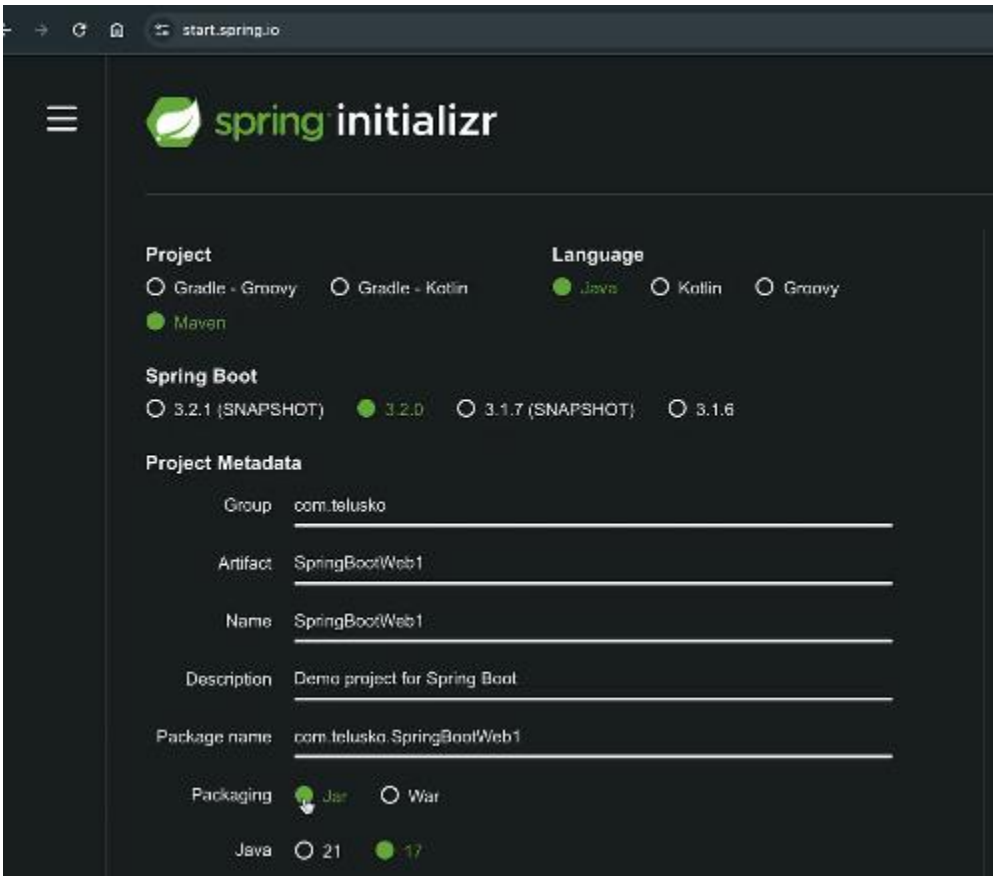
Spring Boot: A framework for building stand-alone, production-ready applications with minimal configuration.

Project Structure:

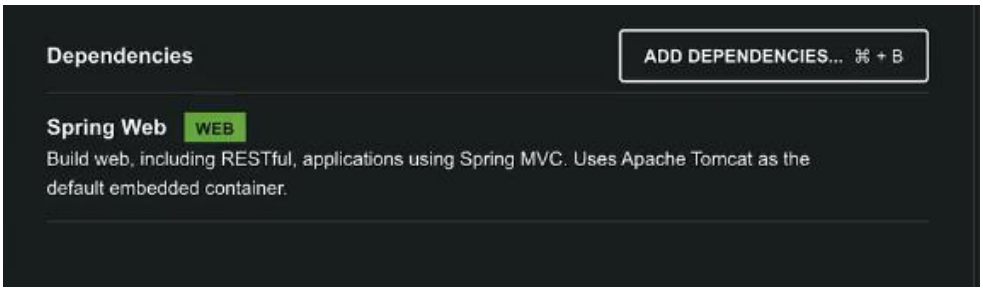
- Follows a standard directory layout (src/main/java, src/main/resources).
- Includes dependencies like Spring Web and Embedded Tomcat.

Create a Spring Boot project.

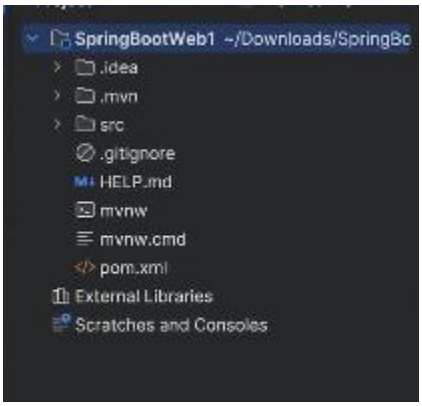
1. Go to start.spring.io or Eclipse with STS support



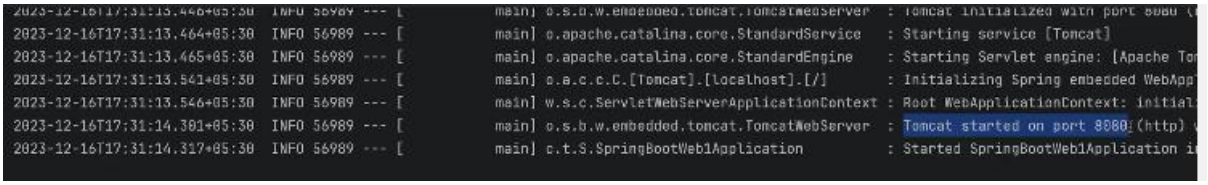
2. Add dependencies



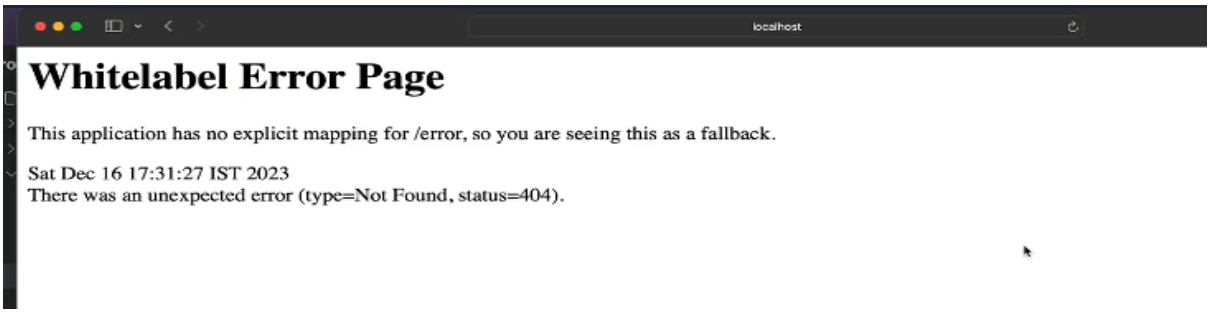
3. Project structure



4. Run as a Spring Boot project.



5. In Browser



This error message means try to get the resource on a specified path but not match with any mapping.

Create a JSP Page

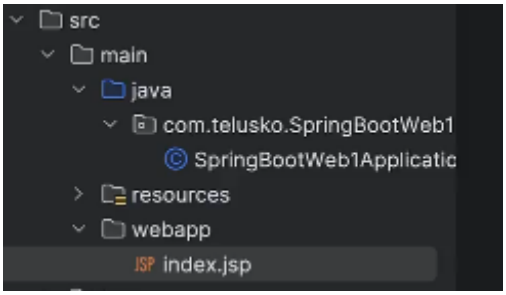
JSP (Java Server Pages): A technology that allows embedding Java code into HTML for dynamic web content.

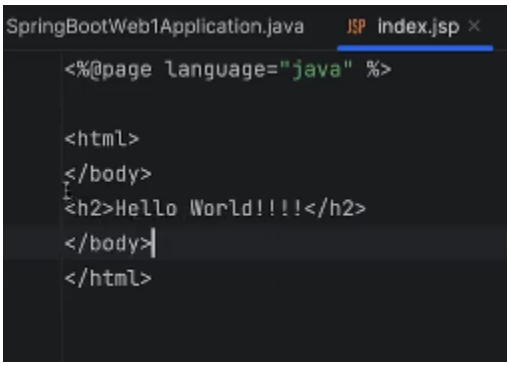
Controller:

- **@Controller Annotation**: Indicates a class as a controller to handle web requests.

Inside the web app, create view pages.

index.jsp





Still not working

- ⇨ In MVC the JSP will be called by controller not by client.
- ⇨ We need to create a controller we will see in the next lecture.

Creating a Controller

1. Create a Simple class HomeController.
   ⇨ Specify annotation **@Controller**
2. We need a method also.
   ⇨ home() method

```
@Controller
public class HomeController{
  public String home(){
   return "index.jsp";
  }
}
```

3. Run
   ⇨ But still not working (not able to find index.jsp or method not called)
   ⇨ We find the home() method not running.
4. Need mapping (learn in next lecture)

Request Mapping

Spring MVC View Resolver:

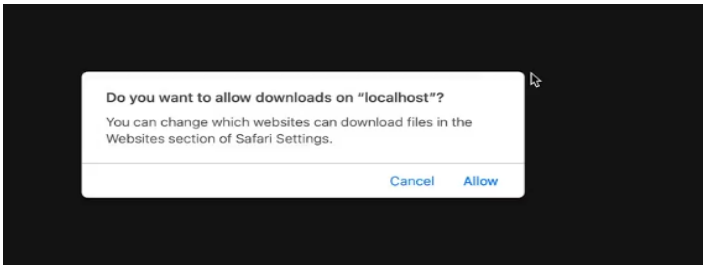
- Configures paths to locate JSP files using spring.mvc.view.prefix and spring.mvc.view.suffix in application.properties.

Tomcat Jasper Dependency: Enables JSP support in Spring Boot.

1. If you want to map (use annotation called @RequestMapping("/")

```
@Controller
public class HomeController{
   @RequestMapping("/")
   public String home(){
     return "index.jsp";
   }
}
```

2. Run

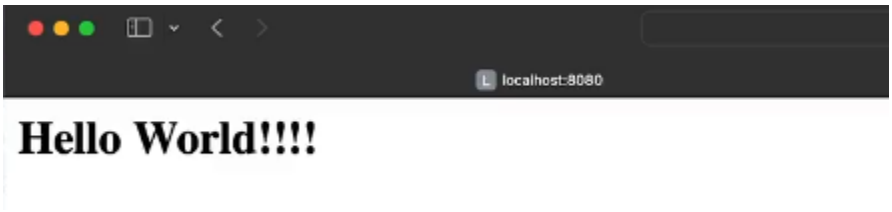


   ⇨ You get something to download.

3. Why isn't it working?
   ⇨ By default, Spring Boot does not support JSP.
   ⇨ Need dependency MVN Repository

As per the Tomcat version, use the Tomcat Jasper version.

4. Run again.

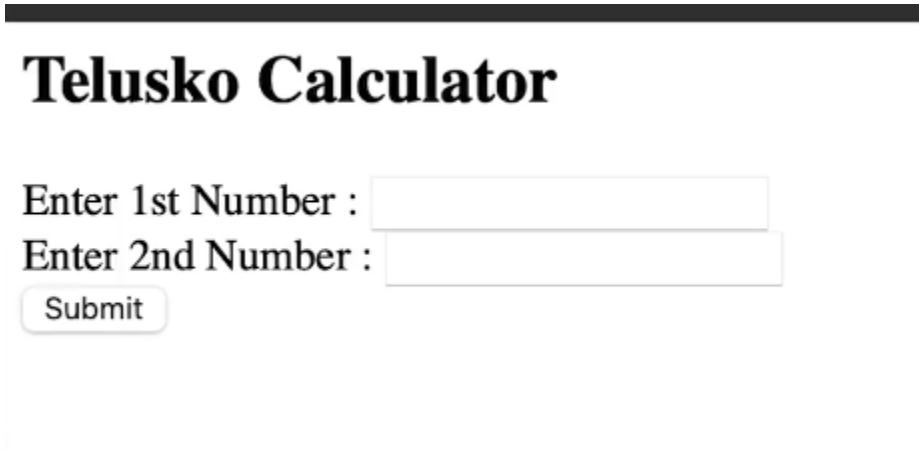


Sending data to Controller

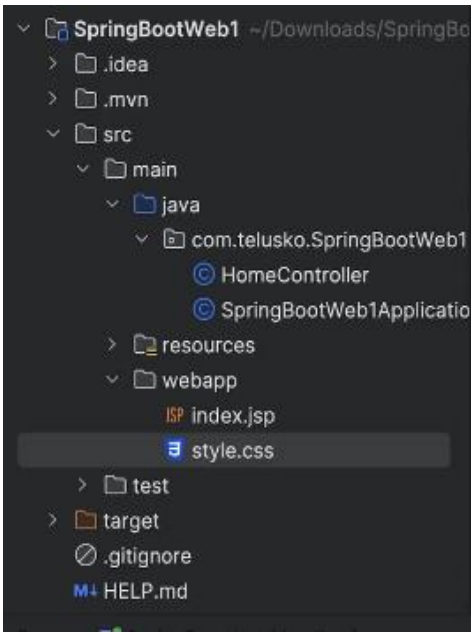
1. We need a form. (index.jsp)

```
<%@page language="java" %>
<html>
<body>
   <h2>Telusko Calculator</h2>
   <form action="add">
      <label for="num1">Enter 1st Number :</label>
      <input type="text" id="num1" name="num1"><br>
      <label for="num2">Enter 2nd Number :</label>
      <input type="text" id="num2" name="num2"><br>
      <input type="submit" value="Submit">
   </form>
</body>
</html>
```

2. Run

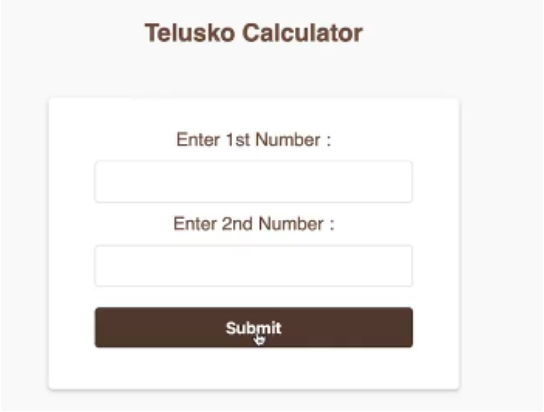


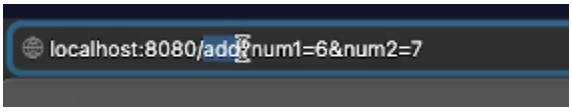
3. Add CSS file (style.css)



Add a link of CSS in JSP

```
<head>
<link rel="stylesheet" type="text/css" href="style.css">
</head>
```





4. Get data in the controller to process a request for calculating the addition of two numbers (we can see in the next lecture).

Accepting Data the Servlet Way

HttpServletRequest:

- getParameter("name"): Retrieves form data sent in the request.

Steps:

1. Create an add() method inside HomeController.
2. Create result.jsp page
3. Add mapping (@RequestMapping("add"))

```
@RequestMapping("add")
public String add(){
  return "result.jsp";
}
```

4. We need the Dispatcher Servlet to map requests.
5. We need to accept value inside the add() method.
   - ⇨ Using the HttpServletRequest object, we can get the value.
   - ⇨ int num1 = Integer.parseInt(req.getParameter("num1"));
   - ⇨ int num2 = Integer.parseInt(req.getParameter("num2"));

```
@RequestMapping("add")
public String add(HttpServletRequest req) {
   int num1 = Integer.parseInt(req.getParameter("num1"));
   int num2 = Integer.parseInt(req.getParameter("num2"));
   int result = num1 + num2;
   System.out.println(result);
   return "result.jsp";
}
```

Display Data on Result Page

HttpSession: Stores data that is available throughout a user's session.

JSTL (JavaServer Pages Standard Tag Library):

- Used for accessing attributes in JSP files. Example: ${result}.

We have a concept of session, so for a particular session, whatever you add in session.

Available for a particular session.

1. Using the session object, add the result value inside the session.
   ⇨ session.setAttribute("result",result);

```
@RequestMapping("add")
public String add(HttpServletRequest req, HttpSession session) {
    int num1 = Integer.parseInt(req.getParameter("num1"));
    int num2 = Integer.parseInt(req.getParameter("num2"));
    int result = num1 + num2;
    session.setAttribute("result", result);
    return "result.jsp";
}
```

2. Inside JSP we directly use the session object.
   <%= session.getAttribute("result") %>

```
<%@ page language="java" %>

<html>
    <head>
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body>
        <h2>Result is: <%= session.getAttribute("result") %></h2>
    </body>
</html>
```

3. Output

⇨ We can also use JSTL ${result}

```
<%@ page language="java" %>

<html>
   <head>
      <link rel="stylesheet" type="text/css" href="style.css">
   </head>
   <body>
      <h2>Result is: ${result}</h2>
   </body>
</html>
```

RequestParam

@RequestParam:

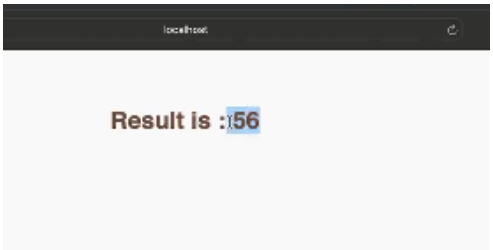
- Maps query parameters or form data to method arguments. Example:

```
public String add(@RequestParam("num1") int num1) {...}
```

1. From the query parameter, we can take it.
   ⇨ Using annotation @RequestParam()

```
@RequestMapping("add")
public String add(int num1, int num2, HttpSession session) {
    int result = num1 + num2 + 1;
    session.setAttribute("result", result);
    return "result.jsp";
}
```

   ⇨ By default, inside the add method, the signatures work as add(@RequestParam("num1") int num1, @RequestParam("num2") int num2, HttpSession session).



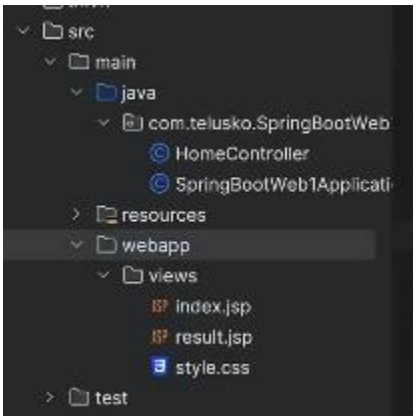
Model Object

Model:

- Used to pass data from the controller to the view using model.addAttribute("key", value).

```
@RequestMapping("add")
public String add(@RequestParam("num1") int num1, @RequestParam("num2")
int num2, Model model) {
    int result = num1 + num2;

    model.addAttribute("result", result);

    return "result.jsp";
}
```

Que: After making the change to the location of view:



Is it working?

⇨ No, it's not working.

Setting Prefix and Suffix

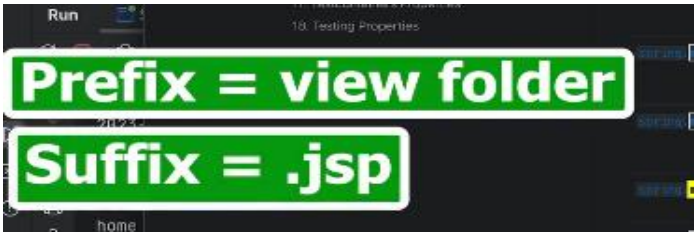
View Resolver:

- Defines how Spring locates view files using prefix (/views/) and suffix (.jsp).

Now we have *viewresolver* to resolve and identify where the views file is and with which name it is associated.

```
@RequestMapping("add")
public String add(@RequestParam("num1") int num1, @RequestParam("num2") int
num2, Model model) {
    int result = num1 + num2;

    model.addAttribute(attributeName: "result", result);

    return "result";
}
```

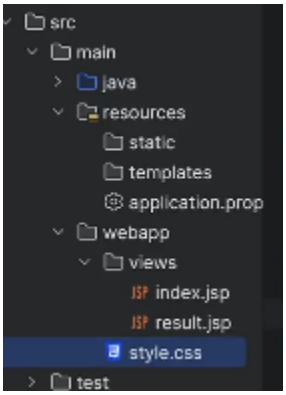
Configure **application.properties**



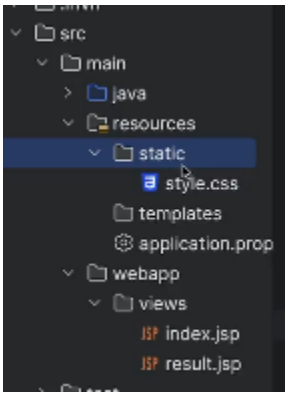

```
spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp
```

ModelAndView

We see CSS not working in the previous lecture; just put style.css in the parent for i.e webapp



Or What can you do? You can put the CSS file in the static folder.



ModelAndView:

The ModelAndView class does it slightly differently. In this, we will combine the view and data in a single object either with the help of the constructor or set by the **setViewName()** method of this class. So that with ModelAndView we can return the view and model in a single object.

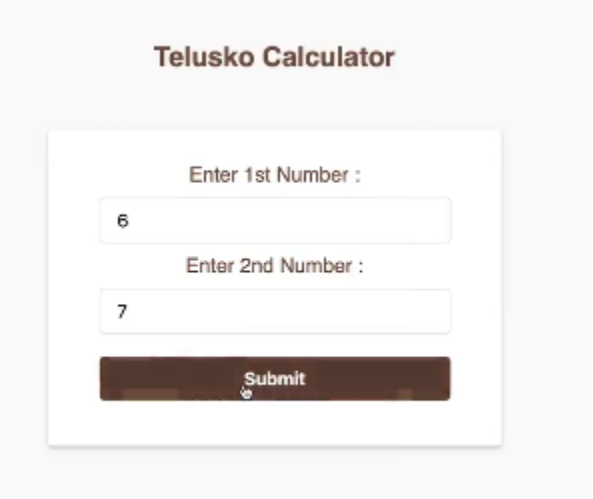
use of **ModelAndView object**

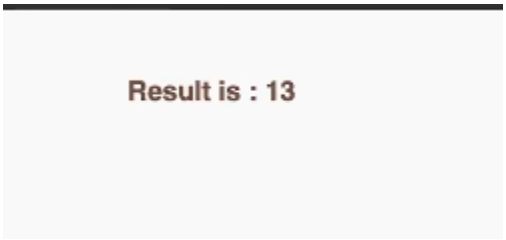
```
mv.addObject("result",result);
mv.setViewName("result");
```

Code:

```
@RequestMapping("add")
public ModelAndView add(@RequestParam("num1") int num1, @RequestParam("num2") int
num2, ModelAndView mv) {
    int result = num1 + num2;

    mv.addObject("result", result);
    mv.setViewName("result");

    return mv;
}
```

Output:





Need for ModelAttribute

@ModelAttribute:

- Binds form data to an object and adds it to the model. Example:

```
public String addAlien(@ModelAttribute Alien alien) { ... }
```

1. Change index.jsp

```
<h2>Telusko Calculator</h2>

<form action="addAlien">
   <label for="aid">Enter Id : </label>
   <input type="text" id="aid" name="aid"><br>
   <label for="aname">Enter Name :</label>
   <input type="text" id="name" name="name"><br>
   <input type="submit" value="Submit">
</form>
```

2. Create Controller for addAlien
   Before using ModelAttribute, we are using the RequestParam annotation.

```
@RequestMapping("addAlien")
public ModelAndView addAlien(@RequestParam("aid") int aid,
@RequestParam("aname") String aname, ModelAndView mv) {
   int result = num1 + num2;

   mv.addObject(attributeName: "result", result);
   mv.setViewName("result");

   return mv;
}
```

3. Create an Alien class

```
public class Alien {
    private int aid;
    private String aname;

   //Setters & Getters
}
```

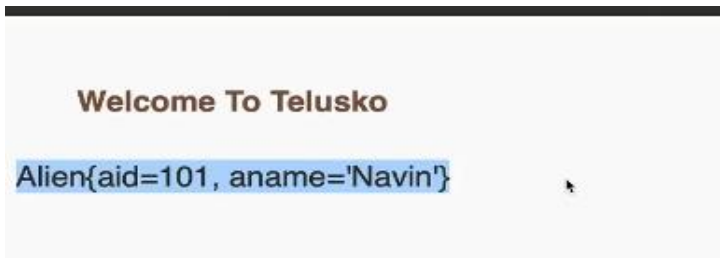
4. Make some changes in the addAlien controller.

```
@RequestMapping("addAlien")
public ModelAndView addAlien(@RequestParam("aid") int aid, @RequestParam("aname")
String aname, ModelAndView mv) {
    Alien alien = new Alien();
    alien.setAid(aid);
    alien.setAname(aname);

    mv.addObject("alien", alien);
    mv.setViewName("result");

    return mv;
}
```

5. Create an result.jsp

```
<%@page language="java" %>

<html>
   <head>
      <link rel="stylesheet" type="text/css" href="style.css">
   </head>

   <body>
      <h2>Welcome To Telusko</h2>

      <p>${alien}</p>
   </body>
</html>
```

Output:



- ⇨ A better way is to take the whole property value into a single Alien object. (We can see this in the next lecture.).
- ⇨ Inside HomeController, add method as courseName.

```
@ModelAttribute("course")
public String courseName() {
    return "Java";
}
```

Using ModelAttribute

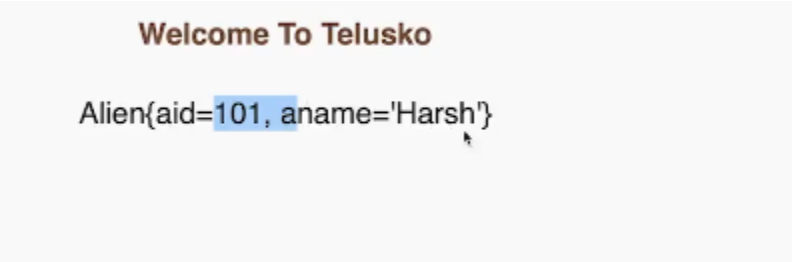
Advanced Usage:

- Custom names can be assigned to objects using @ModelAttribute("customName").

Now, using @ModelAttribute, we can simplify things in a single annotation.

```
@RequestMapping("addAlien")
public String addAlien(@ModelAttribute Alien alien) {
    return "result";
}
```

Output:



Suppose you want to use a different name as alien1.

```
@RequestMapping("addAlien")
public String addAlien(@ModelAttribute("alien") Alien alien) {
    return "result";
}
```

By default, @ModelAttribute is provided.

```
@RequestMapping("addAlien")
public String addAlien(Alien alien) {
    return "result";
}
```

Suppose you want to use course value inside view,

```
<body>
   <h2>Welcome To Telusko</h2>
   <p>${alien}</p>
   <p>Welcome to the ${course} World</p>
</body>
```

- ⇨ You need to separately declare the course.
- ⇨ Create the method courseName inside HomeController.

```
@ModelAttribute("course")
public String courseName() {
   return "Java";
}
```