**Spring MVC (Model-View-Controller)**

**1.Introduction**

**Spring MVC** (Model-View-Controller) is a web framework that is part of the Spring Framework and provides an elegant solution for developing web applications in Java. It follows the **MVC architectural pattern** to separate the application's logic, user interface, and data.

In **Spring Boot**, Spring MVC is heavily simplified, offering easy setup with embedded servers, auto-configuration, and pre-defined configurations.

**1.1 Key Concepts of Spring MVC**

**1.1.1 Model-View-Controller Pattern**

* **Model**: Represents the application’s data. It encapsulates the business logic and interacts with the database to manage the data.
  + **In Spring Boot**: The Model works the same, but Spring Boot provides additional support for data management through Spring Data JPA and simplified database configurations using properties.
* **View**: Responsible for displaying the data to the user. It is generally implemented using JSP, Thymeleaf, etc.
  + **In Spring Boot**: Spring Boot supports multiple view technologies like JSP, Thymeleaf, and others by providing default configurations. You just need to add the dependency, and the view technology is ready to be used.
* **Controller**: Handles user input and coordinates between the Model and View components. Controllers are Java classes annotated with @Controller.
  + **In Spring Boot**: Controllers are the same, but Spring Boot’s auto-configuration reduces boilerplate code, such as servlet and web.xml setup, by automatically wiring the application components.

**1.1.2 DispatcherServlet**

* **DispatcherServlet** is the core of Spring MVC. It is the front controller that manages all incoming HTTP requests and dispatches them to the appropriate controller.
  + **In Spring Boot**: The DispatcherServlet is automatically configured and registered by Spring Boot. You don’t need to manually add it in web.xml or provide explicit configuration—Spring Boot does this for you.
* All requests pass through **DispatcherServlet**, which acts as a central hub, managing the entire workflow of a Spring MVC application.
  + **In Spring Boot**: Same process, but the initialization and setup of DispatcherServlet is handled automatically.

**1.1.3 Request Flow in Spring MVC**

1. **A client sends an HTTP request to the application.**
   * **In Spring Boot**: The client’s request is received in the same way, but you don't need to manually configure the servlet mappings as they are automatically set by Spring Boot.
2. **The DispatcherServlet receives the request.**
   * **In Spring Boot**: Spring Boot automatically handles this configuration.
3. **HandlerMapping helps identify the appropriate controller.**
   * **In Spring Boot**: Spring Boot configures the HandlerMapping automatically.
4. **The controller processes the request and prepares the data, interacting with the Model if necessary.**
   * **In Spring Boot**: Controllers are used in the same way, but setup and data handling are simplified with Spring Boot features like @RestController and default JSON support.
5. **The View Resolver locates the appropriate view (e.g., JSP) to render the data.**
   * **In Spring Boot**: Spring Boot provides a default **ViewResolver** based on the dependencies you include (e.g., Thymeleaf or JSP). There is minimal setup required as it is automatically wired based on configurations.
6. **The data is returned to the client as an HTTP response.**
   * **In Spring Boot**: Similar behavior, but response handling can be done via additional convenience annotations like @RestController for automatic JSON response rendering.

**1.1.4 Important Annotations in Spring MVC**

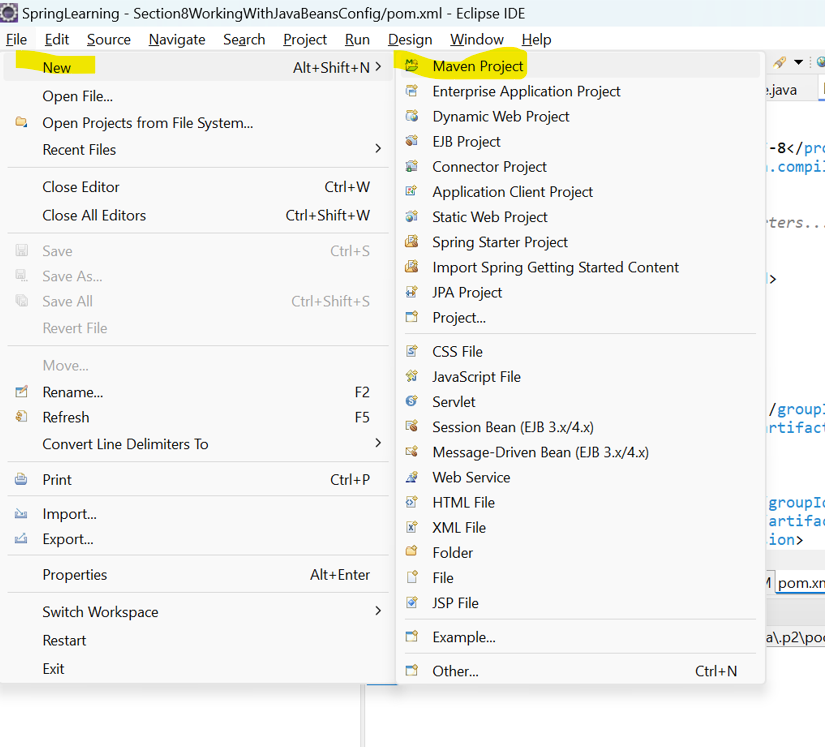
* **@Controller**: Indicates that a class is a Spring MVC controller.
  + **In Spring Boot**: The annotation is used the same way. However, Spring Boot makes setup easier, as it automatically scans and registers controllers in the application context if you use @SpringBootApplication.
* **@RequestMapping**: Maps HTTP requests to handler methods of MVC controllers.
  + Example: @RequestMapping("/home") maps the /home URL to the corresponding method.
  + **In Spring Boot**: Works the same, but Spring Boot simplifies the scanning and registration of mapped handlers, with additional support for @GetMapping, @PostMapping, etc., which provide better clarity for HTTP method types.
* **@RequestParam**: Binds request parameters to method arguments.
  + **In Spring Boot**: Functions the same, and Spring Boot adds convenient JSON parsing via @RequestBody to easily handle RESTful requests.
* **@ModelAttribute**: Binds a model attribute to a method parameter, allowing data from a form to be directly converted to an object.
  + **In Spring Boot**: The behavior is the same, but data binding and validation are simplified with auto-configured features, such as using @Validated and additional validation libraries.
* **@ResponseBody**: Used to directly return data, instead of a view (typically used in REST APIs).
  + **In Spring Boot**: Spring Boot provides @RestController, which combines @Controller and @ResponseBody, making it easier to create RESTful web services.

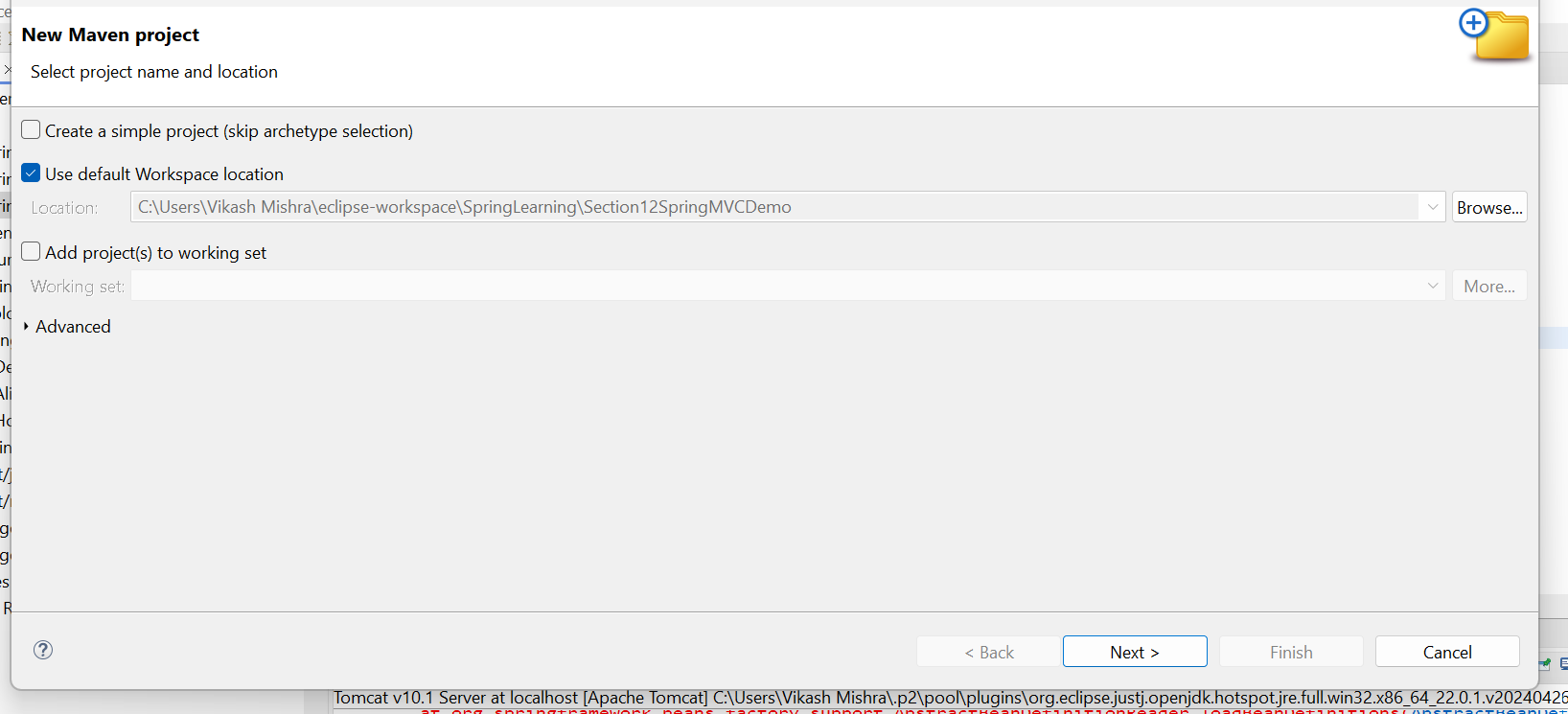
**Summary of Differences**

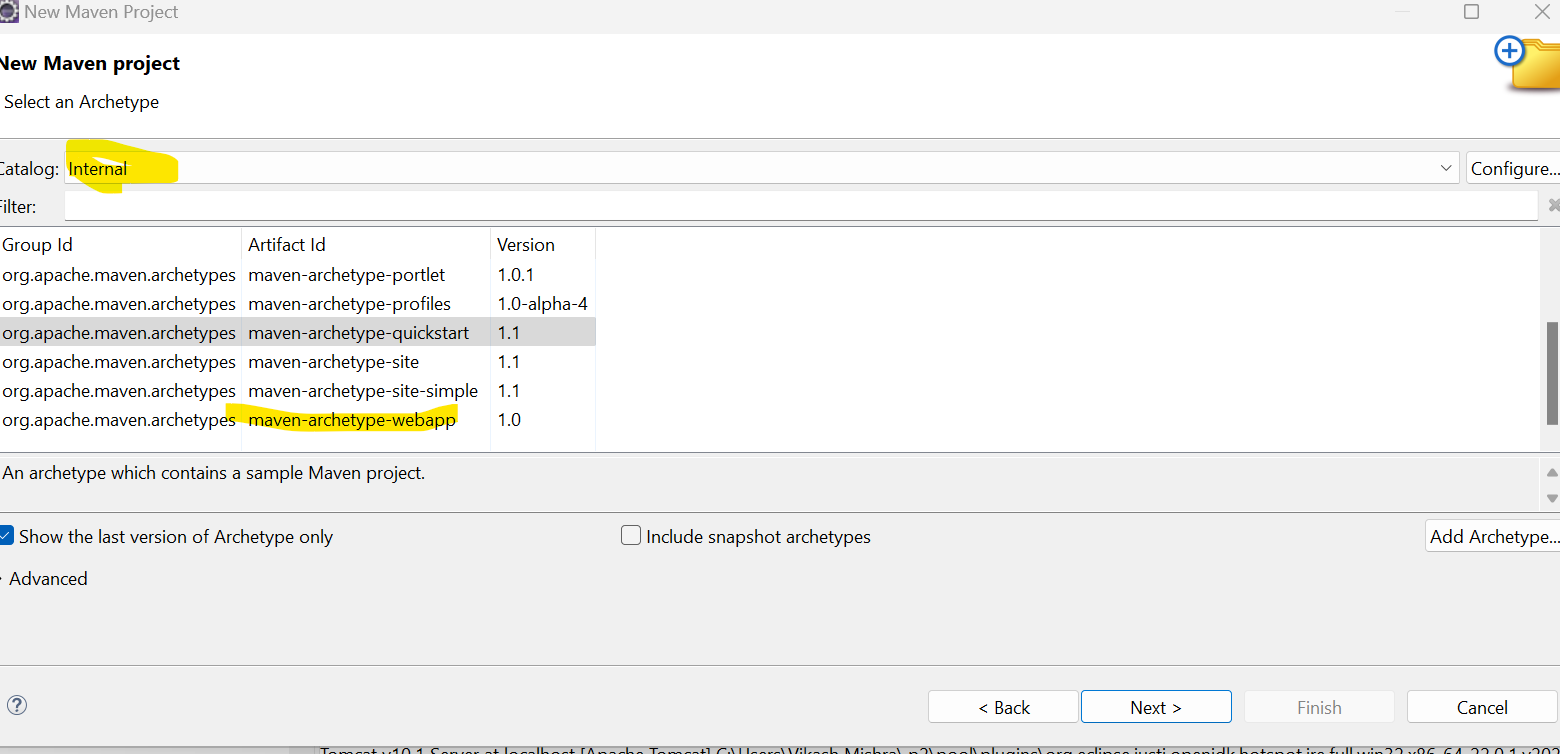
* **Configuration**: In traditional Spring MVC, you need to configure DispatcherServlet, web.xml, and other application components manually, whereas Spring Boot does all the configuration automatically.
* **Setup**: Spring Boot eliminates the need for a web.xml file and allows the use of annotations like @SpringBootApplication for auto-configuration and scanning.
* **Dependencies**: Spring Boot manages most of the dependencies automatically via its **starter dependencies**, reducing boilerplate setup.
* **Embedded Server**: Spring Boot comes with an embedded Tomcat server, allowing you to run your application as a standalone JAR, whereas traditional Spring MVC requires deploying a WAR to an external servlet container.

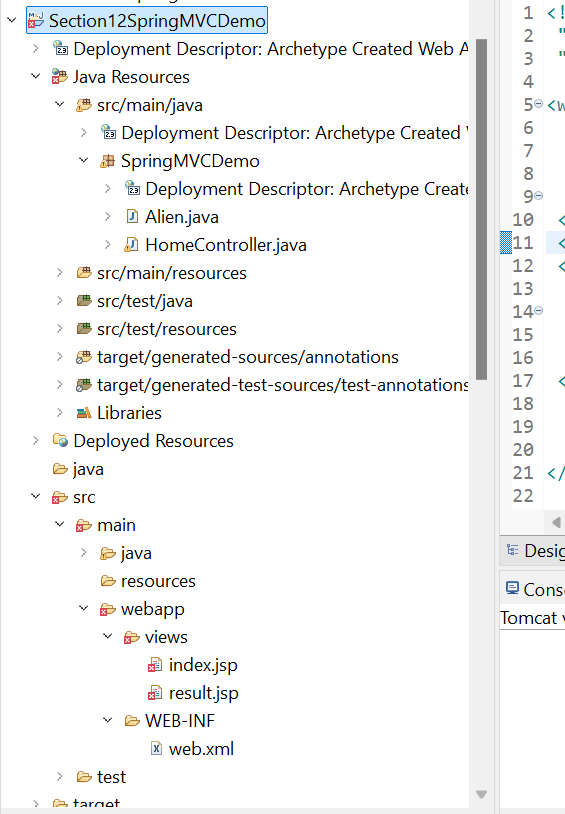
In short, Spring Boot is built on top of the Spring Framework and aims to simplify the development process by providing sensible defaults, reducing boilerplate configurations, and providing an embedded server, while traditional Spring MVC requires more hands-on setup and gives developers complete control over configuration and deployment.

**2. Creating MVC Project**

****

****

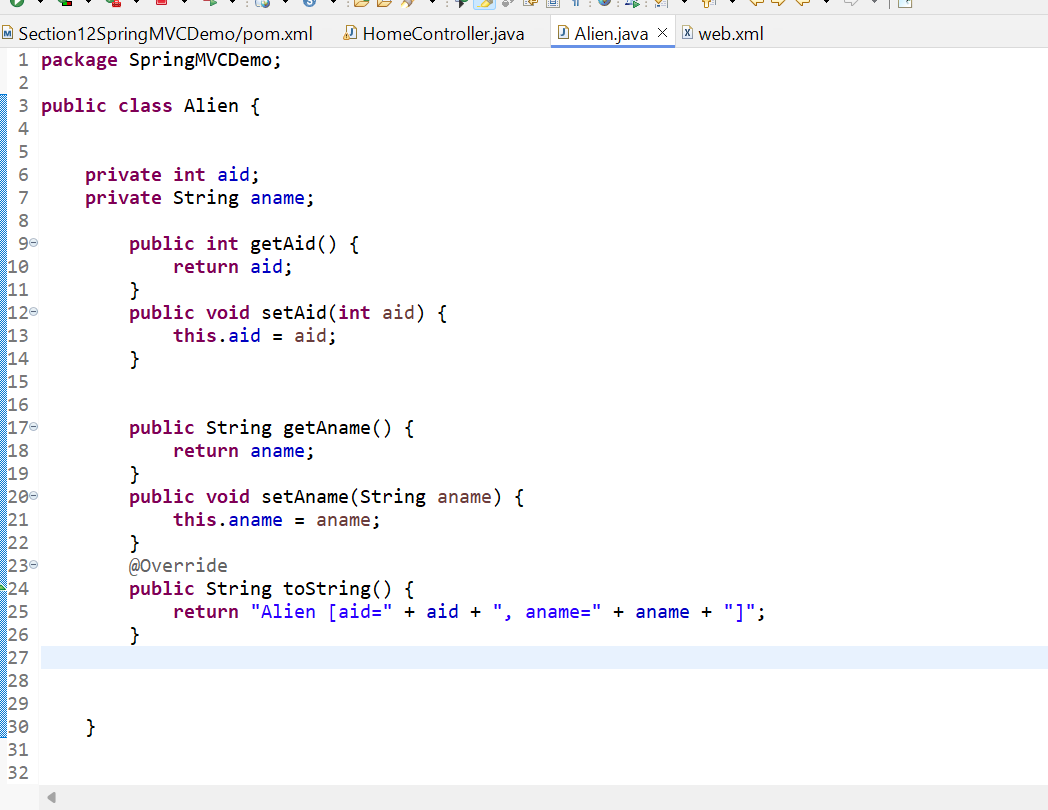
****

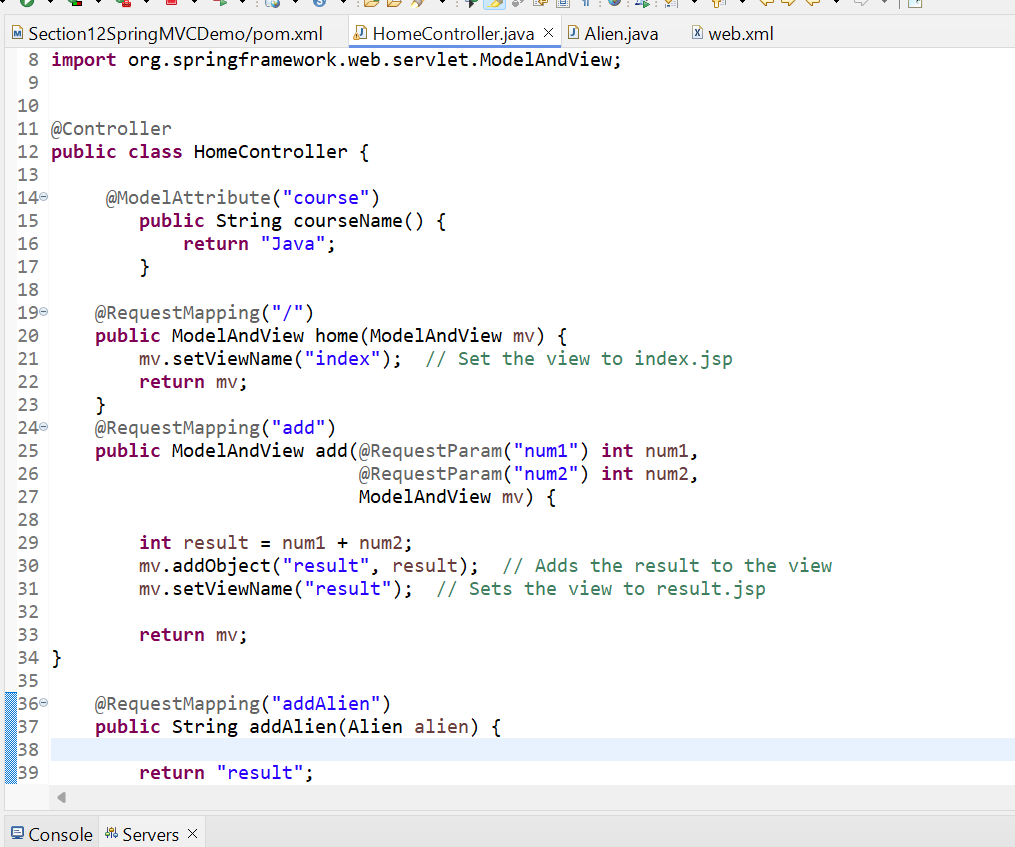
****

**2.1 Add below dependency in POM.xml file**

****

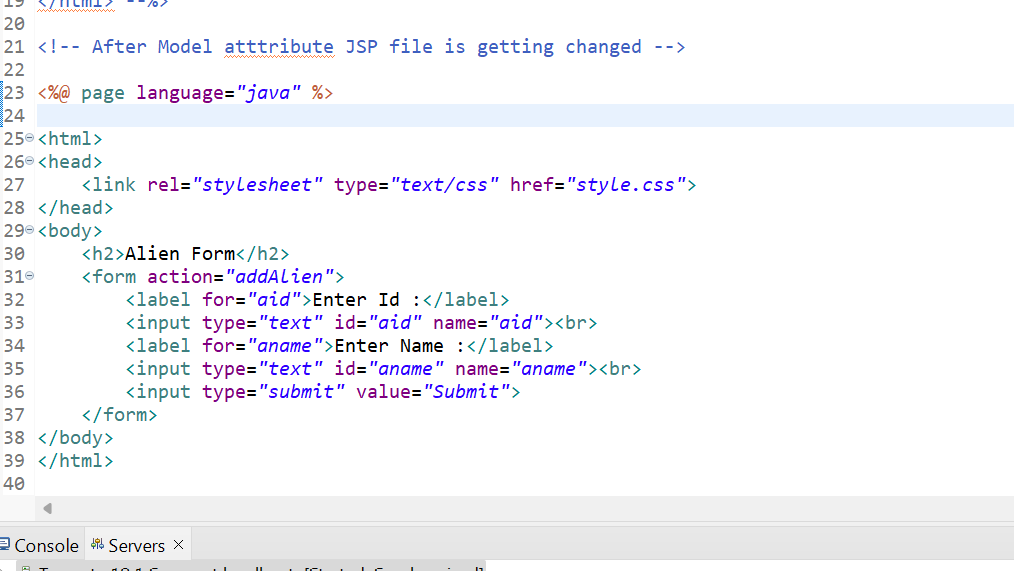
**2.2 Create Alien and homecontroller class**

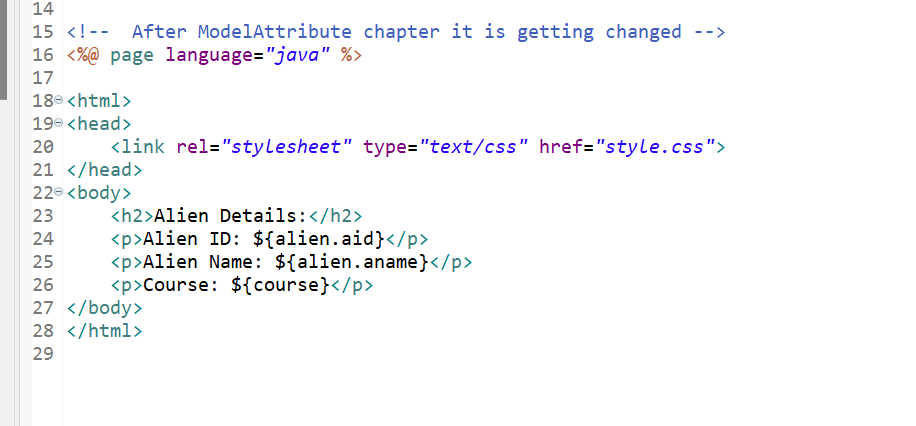
****

****

**2.3 Create Indec.jsp and result.jsp file under below path**

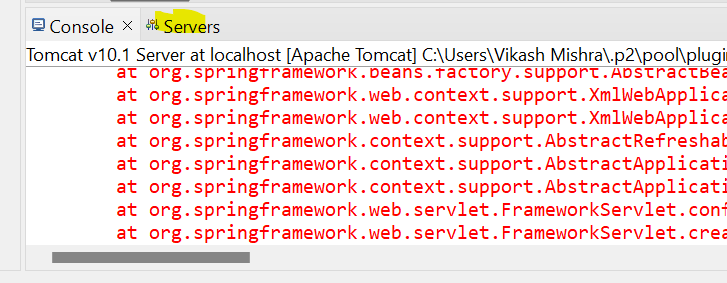
**Src/main/webapp/views**

****

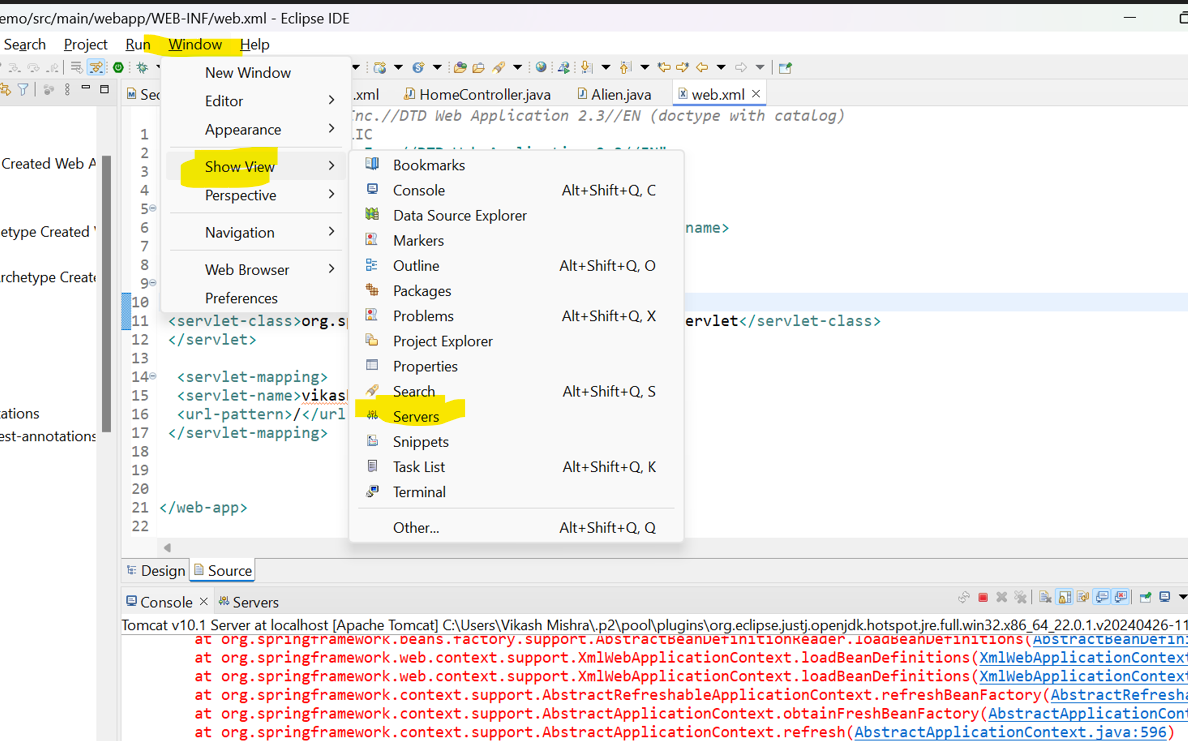
****

**2. Running Tomcat Server in eclipse**

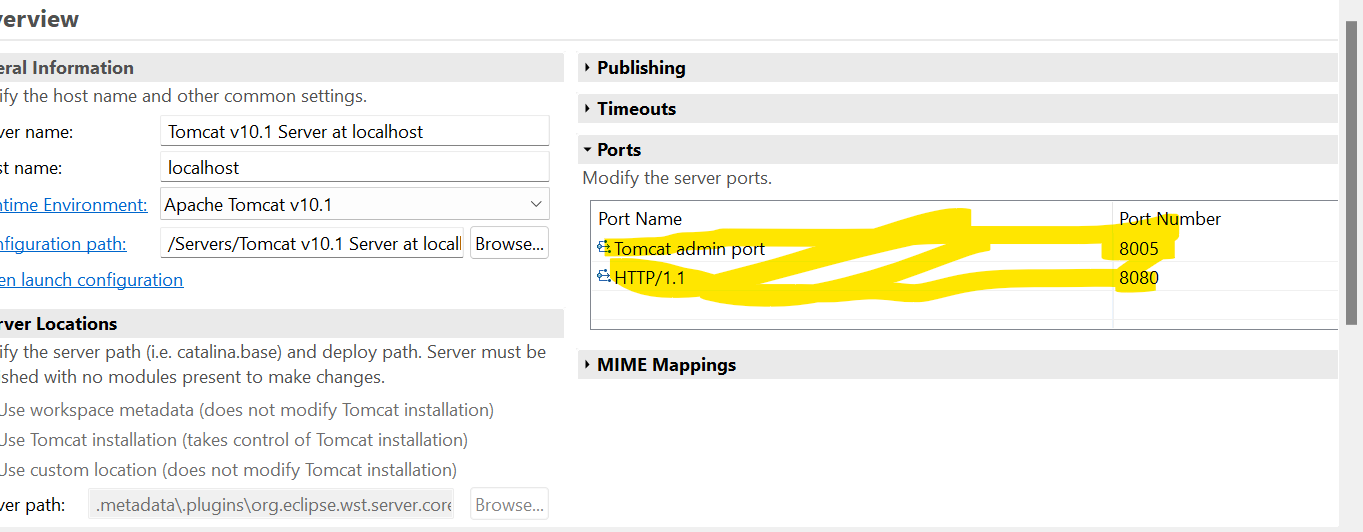
If server is not seen besides of console like below

****

Then follow the below steps

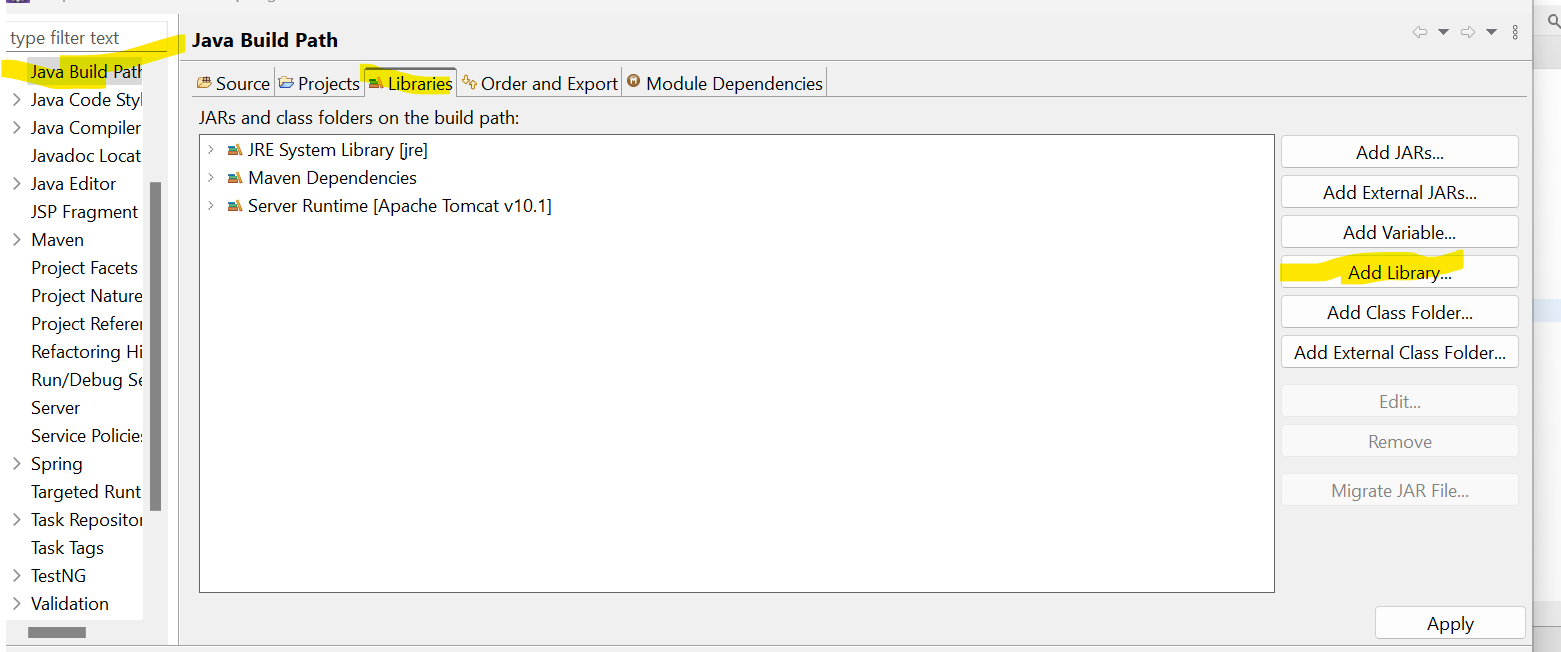
****

Now double click on the server nad make sure for the below port



Now before running tomcat server, we have to make sure jar has been added to check that follow below steps

Right Click on the project >>Build path>>configure build path



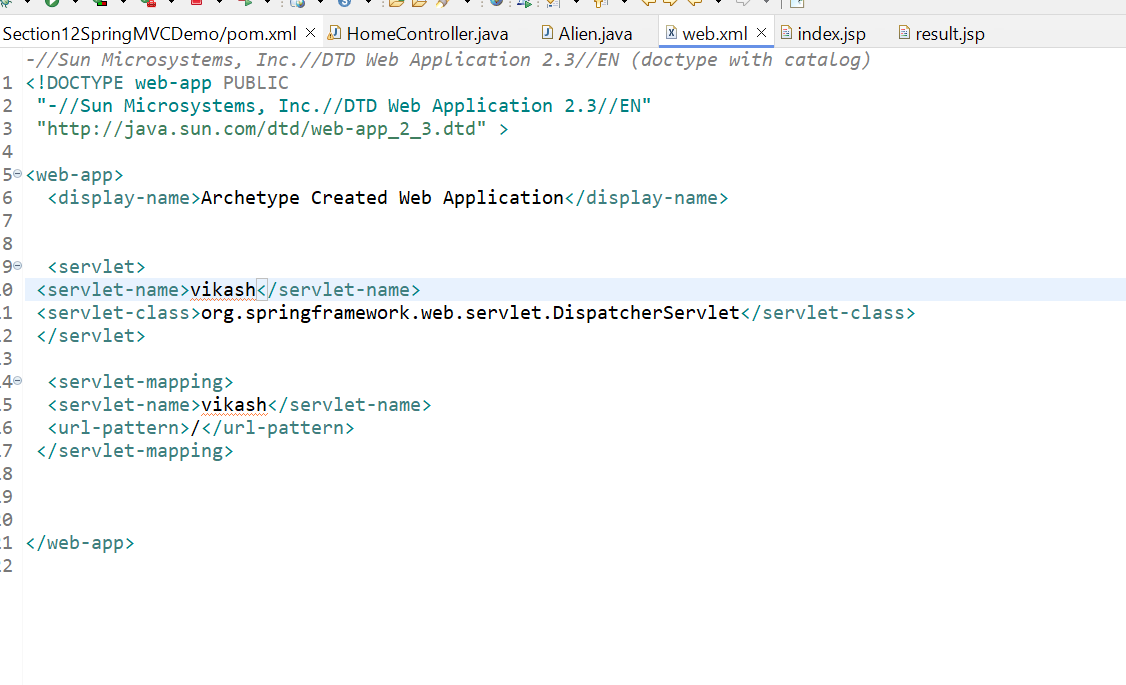
And select your tomcat server

**3. Dispatcher Servlets**

web.xml File Explanation

Your web.xml file serves as the deployment descriptor for your web application and includes configuration for the DispatcherServlet, which is the core component that drives Spring MVC.

Here is the content of your web.xml:



**Key Parts of web.xml**

1. **<servlet>:**

* **<servlet-name>vikash</servlet-name>:** This names the servlet as vikash.
* **<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>:** This is the **DispatcherServlet** provided by **Spring MVC**. It acts as the Front Controller, routing requests to appropriate controllers.

1. **<servlet-mapping>:**

* **<servlet-name>vikash</servlet-name>:** This connects the **DispatcherServlet** named vikash to a URL pattern.
* **<url-pattern>/</url-pattern>:** The / pattern maps all requests to the **DispatcherServlet**, meaning all incoming requests will be processed by this servlet.

**DispatcherServlet Flow in Spring MVC**

The **DispatcherServlet** is the main controller that manages request and response flow in a Spring MVC application. Here is the execution flow in your configuration:

1. **HTTP Request**:
   * A client sends an HTTP request to the server. For example, a request for /index.
2. **DispatcherServlet Initialization**:
   * When the server receives the request, it first passes it through the **DispatcherServlet** (vikash in your case). The **DispatcherServlet** is defined in web.xml and is the entry point for every request.
   * On initialization, the **DispatcherServlet** loads the Spring MVC configuration file. The default configuration file name is vikash-servlet.xml, where vikash matches the <servlet-name> in web.xml. This file contains bean definitions, view resolvers, etc.
3. **Handler Mapping**:
   * The **DispatcherServlet** consults a **HandlerMapping** to find the right controller based on the URL pattern.
   * The **HandlerMapping** is used to determine which controller method should handle the request.
4. **Controller Execution**:
   * The appropriate **Controller** method is invoked. In your case, you have a HomeController that has mapped handler methods, for example, @RequestMapping("/index").
   * The controller processes the request, interacts with the model (if necessary), and prepares the data to be displayed.
5. **View Resolution**:
   * After processing, the **Controller** returns a logical view name (e.g., "index").
   * The **ViewResolver** (configured in the Spring context) maps this logical view name to an actual view file (index.jsp in /views folder).
6. **Render the View**:
   * The view (JSP file) is rendered and the HTML response is sent back to the client.

**Execution Flow Example**

Let’s take an example request flow:

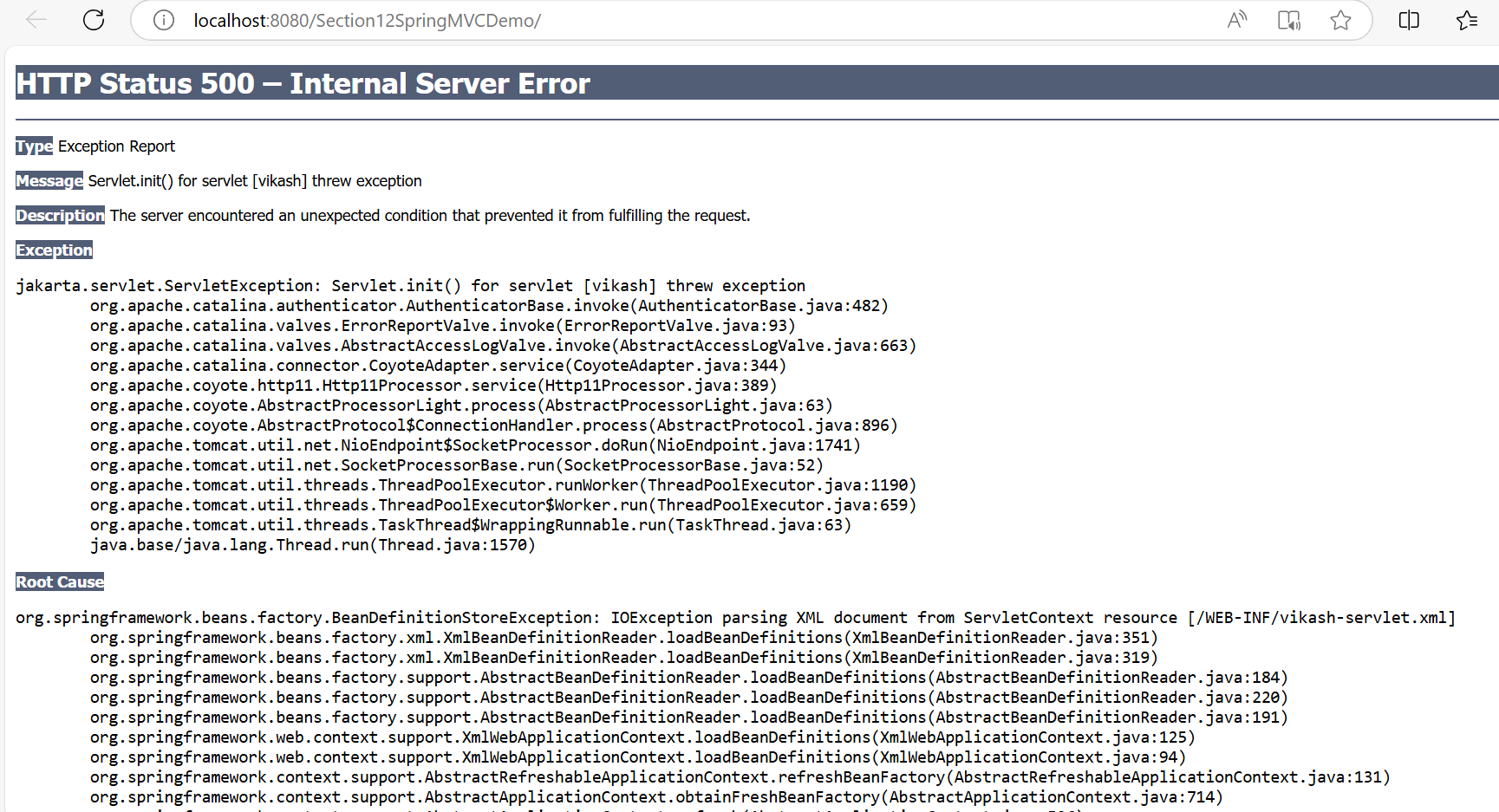
1. **Client Request**:
   * A client requests http://localhost:8080/Section12SpringMVCDemo/index.
2. **DispatcherServlet Receives Request**:
   * The request reaches **DispatcherServlet** (vikash), which is defined in web.xml to handle all incoming requests (/ pattern).
3. **Locate Handler**:
   * **DispatcherServlet** checks for a controller that can handle the /index request.
   * It finds a method in HomeController annotated with @RequestMapping("/index").
4. **Controller Processing**:
   * The HomeController processes the request and prepares the data.
   * It returns a logical view name like "index".
5. **View Resolution**:
   * The **ViewResolver** maps "index" to /views/index.jsp (configured in the Spring context).
6. **Response**:
   * **DispatcherServlet** forwards the request to the JSP view.
   * The **JSP** page (index.jsp) is rendered and returned to the client as an HTML response.

**Summary**

* The **DispatcherServlet** is configured in web.xml as vikash, acting as the front controller for all requests.
* It receives requests, finds appropriate controller methods, and renders the correct view.
* The **Controller** handles business logic, while the **ViewResolver** determines which JSP should be returned.

If you have any specific part you'd like me to elaborate further on, such as configuring vikash-servlet.xml, creating a controller, or adding more views, please let me know!

But when you run the project it will throw 500 error



Now we need to add vikash-servlet.xml file at src/main/webapp/WEB-INF/vikash-servlet.xml

**Detailed Breakdown**

1. **Namespaces and Schema Definitions**:
   * The <beans> tag contains multiple XML namespace definitions, which are necessary for configuring various Spring components.
   * The xmlns:mvc and xmlns:context are used to enable Spring MVC and context-related features.
2. **Component Scanning (<ctx:component-scan>)**:
   * <ctx:component-scan base-package="com.telusko"/> enables Spring to automatically detect Spring components like @Controller, @Service, @Repository, etc.
   * **base-package="com.telusko"** means Spring will scan this package and its sub-packages for components.
   * If you have controllers, services, etc., make sure they are located under this package or adjust the package name accordingly.
3. **Annotation-driven Configuration (<ctx:annotation-config/>)**:
   * This enables the processing of Spring annotations like @Autowired, @Qualifier, etc.
   * Note: Since you are working with Spring MVC, it is recommended to use <mvc:annotation-driven/> instead of <ctx:annotation-config/>. This allows handling things like @RequestMapping, data binding, and JSON conversions.
4. **View Resolver (InternalResourceViewResolver)**:
   * This bean is used to resolve the logical view names returned by your controllers to actual JSP files.
   * <property name="prefix" value="/views/"/>: This specifies that all views will be located in the /views/ directory inside webapp.
   * <property name="suffix" value=".jsp"/>: This means that when a controller returns "index", Spring MVC will look for /views/index.jsp.

