**What We Were Trying to Achieve:**

Our goal was to configure **JSP (JavaServer Pages)** as the **view technology** for rendering dynamic web pages in our **Spring Boot** application. By doing this, we aimed to serve HTML content with dynamic data (like JSP can embed Java code inside HTML) when users access specific URLs.

In short, our setup is part of implementing a **Model-View-Controller (MVC)** pattern using **Spring Boot** and JSP. The controller handles the logic, and JSP files act as the view, which renders dynamic content.

**Step-by-Step Breakdown of What we Did and Why:**

**1. Added JSP Dependency in pom.xml:**

we added the **Tomcat Embed Jasper** dependency in pom.xml:

 **Why this is important**:

* **Tomcat Embed Jasper** is needed because JSP requires a **JSP compiler**, and in Spring Boot, the embedded **Tomcat** server doesn't include the JSP compiler by default.
* **Jasper** is the JSP engine for Tomcat. By adding this dependency, we're enabling Spring Boot's embedded Tomcat to compile and render JSP files.

 **Without it**: If we don't include this dependency, Spring Boot won't be able to process and render our JSP files, resulting in errors when trying to serve JSP pages.

**2. Created the Folder Structure:**

we created the folder structure under src/main/webapp/ and added our JSP files inside /views/.

src/main/webapp/views/home.jsp

src/main/webapp/views/contact.jsp

 **Why this is important**:

* Spring Boot needs to know where to find our JSP files. By default, **JSPs are placed inside the src/main/webapp folder**, which simulates a web application's directory structure in a traditional server deployment.
* The /WEB-INF/views/ folder is commonly used in Java web apps to store JSPs because it hides JSPs from direct access through the browser (meaning they can only be accessed via controllers).

 **What we achieved**: By placing JSPs in src/main/webapp/views/, we provided a location where the Spring Boot application can find and render the JSP files for the web pages.

**3. Configured Prefix and Suffix in application.properties:**

spring.mvc.view.prefix=/views/

spring.mvc.view.suffix=.jsp

 **Why this is important**:

* **Prefix (spring.mvc.view.prefix)**: This defines the **path** where Spring Boot looks for our JSP files. In this case, it's telling Spring Boot to look in the /views/ folder under src/main/webapp/.
* **Suffix (spring.mvc.view.suffix)**: This defines the **file extension** for view files. By setting it to .jsp, we tell Spring Boot that it should expect JSP files.

 **How it works**: When a controller returns a view name (like "home" or "contact"), Spring Boot will automatically look for the file in the folder /views/ and append .jsp. So, for "home", Spring will look for /views/home.jsp.

4. **Created JSP Files (home.jsp and contact.jsp)**:

We created contact.jsp and home.jsp to display content when users visit the /contact and / endpoints, respectively.

**Why this step is important:**

* **JSP (JavaServer Pages)** allows we to write HTML with embedded Java code (though using Java code inside JSP is generally discouraged today in favor of more modern practices). It is primarily used for generating dynamic content for the view layer.
* By creating home.jsp and contact.jsp, we provided the **view layer** for our application, which is rendered when our controller forwards requests to these pages.

**What happens:**

* When a user accesses the /contact URL, the Spring controller returns "contact". Because of the configuration in application.properties, Spring Boot looks in /views/ for a file named contact.jsp and renders it.

5. **Controller Setup (MainController.java)**:

@Controller

public class MainController {

@RequestMapping("/")

public String home() {

System.out.println("This is the home page");

return "home"; // Resolves to /views/home.jsp

}

@RequestMapping("/contact")

public String contact() {

System.out.println("This is the contact page");

return "contact"; // Resolves to /views/contact.jsp

}

}

 **Why this is important**:

* In a **Spring MVC** or **Spring Boot MVC** application, the **controller** handles incoming requests and returns a **view name**. Based on this view name, Spring will look for the corresponding JSP file and render it.
* **@RequestMapping("/")**: This maps the root URL (http://localhost:8080/) to the home() method, which returns the "home" view.
* **@RequestMapping("/contact")**: This maps the /contact URL (http://localhost:8080/contact) to the contact() method, which returns the "contact" view.

 **What we achieved**:

* We created a **controller** that handles different requests and links them to JSP views. When a user accesses /, the home.jsp file is rendered, and when they access /contact, contact.jsp is rendered.

**6. Tomcat Embedded Server (Port 8080):**

We mentioned that we're using **port 8080**. This is the default port for the **embedded Tomcat server** in Spring Boot.

* **Why this is important**:
  + Spring Boot applications come with an embedded server (Tomcat by default), meaning we don't need to manually configure or deploy our application to an external server like we would in traditional Spring MVC setups. Instead, we just run the Spring Boot application, and it starts the Tomcat server on port 8080 (unless configured otherwise).
* **What we achieved**:
  + We made it easier to test and run our application since the embedded Tomcat server runs automatically when we start our Spring Boot application. We can access our application at <http://localhost:8080/>.

**Summary of Each Step:**

1. **JSP Dependency (Tomcat Embed Jasper)**: Enables JSP support in the embedded Tomcat server.
2. **Folder Structure (src/main/webapp/views/)**: The location where JSP files are stored.
3. **Prefix and Suffix (application.properties)**: Configures the folder and file extension for JSP files.
4. **JSP Files (home.jsp, contact.jsp)**: The view files rendered when users access the corresponding URLs.
5. **Controller (MainController.java)**: Handles requests and returns the view names to be resolved to JSP files.
6. **Embedded Tomcat (Port 8080)**: Automatically runs your application and serves requests on localhost:8080.

**Why Was It Easier in Spring Boot Than Spring MVC?**

* **Auto-Configuration**: Spring Boot auto-configures much of the setup that we would otherwise have to do manually in traditional Spring MVC (e.g., configuring the view resolver in xml, setting up the embedded server, etc.).
* **Embedded Server**: Spring Boot provides an embedded Tomcat server, so we don’t have to manually deploy the application to an external server.
* **Fewer Configuration Files**: With Spring Boot, we avoid creating additional xml files (like web.xml, dispatcher-servlet.xml) for configuring the web application. Everything is done using annotations and properties.

**Conclusion:**

we successfully set up a **Spring Boot application** using **JSP** as the view layer. The steps we took were aimed at creating a simple **MVC structure** where:

* The **controller** handles requests.
* The **view (JSP)** is rendered dynamically.
* **Spring Boot** makes this process easier by auto-configuring many parts, such as the embedded server, view resolver, and more.

OUr goal was to create a basic Spring Boot web application that renders **JSP pages** based on user requests.