## 8️⃣ Embedded vs External Tomcat Behavior

### ✅ What Happens:

When using SpringBoot MVC, we can run the app in two different ways:

1. ✅ Embedded Tomcat → Runs inside the application as a JAR.
2. ✅ External Tomcat → Runs as a WAR deployed to an external server.

👉 Both behave a little differently, especially when it comes to **Context Path** and **Index.jsp file usage**.

### ✅ Comparison Table

| **Feature** | **Embedded Tomcat** | **External Tomcat** |
| --- | --- | --- |
| ✅ Context Path | Taken from application.properties. | Defaults to project name as context path (unless configured). |
| ✅ Index.jsp as Welcome File | Does NOT serve index.jsp automatically as default page. | Serves index.jsp automatically as default welcome file. |

### ✅ Example Scenario

#### Embedded Tomcat Example:

* Configuration in application.properties:

server.servlet.context-path=/MyApp

spring.mvc.view.prefix=/WEB-INF/pages/

spring.mvc.view.suffix=.jsp

server.port=8080

* Accessing URL:

http://localhost:8080/MyApp/index.jsp

* Important Point:  
  If you just type http://localhost:8080/MyApp/, the embedded Tomcat will NOT serve index.jsp by default → You must specify the full path.

#### External Tomcat Example:

* No context path in application.properties →  
  Automatically, context path becomes the project name (e.g., MyApp).
* Accessing URL:

http://localhost:8080/MyApp/

* Important Point:  
  External Tomcat will automatically serve index.jsp as the default welcome file.

### ✅ Why Is This Important?

| **Case** | **Simple Explanation** |
| --- | --- |
| Embedded Tomcat | You have more control → Explicitly configure everything (context path, view prefix/suffix). Prevents ambiguity. |
| External Tomcat | Behaves like a traditional web server → Default context path is project name, and index.jsp works as default welcome page. |

👉 Helps developers avoid confusion when switching between standalone apps and deployed WAR files.

### ✅ Simple Summary of Example

| **Step** | **Embedded Tomcat** | **External Tomcat** |
| --- | --- | --- |
| 1. | Context Path | From application.properties (e.g., /MyApp). |
| 2. | Index.jsp | Not served automatically → Must type full URL. |

## 9️⃣ Class Requirements Based on App Type

### ✅ What Happens:

Depending on whether the SpringBoot MVC app is a **Standalone App (JAR)** or a **Deployable App (WAR)**, the required classes are different.

| **Type of App** | **Required Class** | **Reason (Simple Explanation)** |
| --- | --- | --- |
| ✅ Standalone App | No need for ServletInitializer class | Because SpringApplication.run() automatically creates everything (Embedded Tomcat, IOC Container, DispatcherServlet). |
| ✅ Deployable App | Requires ServletInitializer class | Helps create the IOC Container and register DispatcherServlet when deployed as a WAR in an external server. |

### ✅ Example Scenario – How It Works

#### 👉 Standalone App Example:

Main class looks like this:

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

❗ Important Tip:  
No need for ServletInitializer class at all.  
If it exists, comment it out →

// public class ServletInitializer extends SpringBootServletInitializer { ... }

Reason:  
Spring Boot handles everything by itself using SpringApplication.run().

#### 👉 Deployable WAR Example:

Create a class like this:

public class ServletInitializer extends SpringBootServletInitializer {

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

return application.sources(MyApp.class);

}

}

❗ Important Tip:  
Comment out the main() method →

// public static void main(String[] args) { ... }

Reason:  
External Tomcat server does NOT use the main method. Instead, ServletInitializer helps to initialize the app (IOC + DispatcherServlet).

### ✅ Why Is This Important?

| **Case** | **Simple Explanation** |
| --- | --- |
| Standalone App | Simple → Just run the app using main method → Spring Boot starts embedded Tomcat, IOC Container, DispatcherServlet automatically. |
| Deployable App | Needs extra setup → ServletInitializer ensures the external server knows how to initialize the app. Otherwise, app won’t start properly. |

👉 Avoid mistakes:

* Don’t have both main() and ServletInitializer active at the same time.
* Choose the right setup depending on how you deploy the app.

### ✅ Simple Summary of Example

| **Type** | **Action Needed** | **Simple Explanation** |
| --- | --- | --- |
| Standalone | Add main() method only | Run app as JAR → Everything handled automatically. |
| Deployable WAR | Add ServletInitializer class only | Tomcat uses this class to create IOC and DispatcherServlet dynamically. |
| Tip | Comment out the class you don’t need based on deployment type | Prevents confusion and deployment errors. |

While building SpringMVC/SpringBootMVC applications we need to use the following annotations while developing handler classes =>

To make java class as SpringBean + handler/controller class

**a. @Controller**

* This is used on a **Java class**.
* It tells Spring:

“Hey Spring, treat this class as a **Controller (handler class)** which can handle web requests.”

* By default, when you mark a class with @Controller, Spring will create an object of this class and keep it inside the **Spring container** (it becomes a **Spring Bean**).

So:  
👉 @Controller = Make this class a **Spring-managed handler class**.

**b. @RequestMapping**

* This is used on a **method** inside the controller.
* It tells Spring:

“When a request with a particular **URL (path)** and **HTTP method (GET/POST)** comes, call this Java method to handle it.”

**Note:**

* Browser can send only GET (default) and POST requests directly.
* Other request types like PUT, DELETE are usually sent by tools/APIs (not from normal HTML forms).

**Example in Code**

@Controller

public class LoginController {

// This method handles GET request for "/login"

@RequestMapping(value="/login", method=RequestMethod.GET)

public String login(String username, String password) {

// 1. We can write logic to check if username & password are valid

// Example: if(username.equals("admin") && password.equals("1234")) ...

// 2. Or we can send the request to a service class to do the validation.

// 3. Finally, we return the name of the view (JSP/HTML/Thymeleaf page) to show result.

return "loginSuccess"; // means it will open loginSuccess.jsp or loginSuccess.html

}

}

**✅ How to Display Home Page Without Typing index.jsp**

**✅ What Happens:**

Normally, when you run a SpringBoot MVC app using embedded Tomcat (Standalone Web App), it does NOT automatically serve index.jsp as the welcome page.

👉 To avoid manually typing:

http://localhost:9999/FirstApp/index.jsp

And make the home page display by just using:

http://localhost:9999/FirstApp/

We create a special **handler method** that returns the Logical View Name (LVN) of the home page.

**✅ Example Scenario – How It Works**

👉 Create a Controller class like this:

@Controller

public class ShowHomeController {

@RequestMapping(value = "/", method = RequestMethod.GET)

public String showHomePage() {

return "home"; // Maps to /WEB-INF/pages/home.jsp

}

}

| **Part** | **Simple Explanation** |
| --- | --- |
| @RequestMapping(value = "/", method = RequestMethod.GET) | Listens to URL path / (root of app). |
| return "home"; | Logical View Name → The view resolver maps this to /WEB-INF/pages/home.jsp. |

**✅ Why Is This Important?**

| **Problem Without Handler** | **Simple Explanation** |
| --- | --- |
| Typing index.jsp | Not automatic in embedded Tomcat → Users would have to type full URL. |
| Public JSP | Bad practice → Exposing JSP directly is insecure. |

👉 By using a Controller method for /, you:  
✔ Handle home page request in a clean way.  
✔ Keep all views in private area (WEB-INF/pages/) → Secure.  
✔ Avoid direct access to JSP files.

**✅ Example Detailed Note**

1. User types this URL in browser:

http://localhost:9999/FirstApp/

1. DispatcherServlet intercepts request and finds handler method mapped to /.
2. showHomePage() method is called → Returns "home".
3. ViewResolver converts "home" → Physical JSP file:

/WEB-INF/pages/home.jsp

1. Result:  
   The home page is displayed →

Welcome to SpringMVC

**✅ Simple Summary of Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Create Controller Method | Maps request path / to showHomePage(). |
| 2. | Return LVN | Returns "home". |
| 3. | ViewResolver Maps LVN | Finds /WEB-INF/pages/home.jsp. |
| 4. | Home Page Displayed | Shows HTML content → “Welcome to SpringMVC”. |

👉 This way, the home page appears automatically without typing index.jsp, and we follow best practices by keeping JSP files in the private area.

**Eg:** WelcomePageExample

**🔷 Developing Spring Web MVC Application (Spring Boot)**

**✅ 1. spring-boot-starter-web dependency**

* This dependency is added in pom.xml.
* It provides all the required libraries to build a **Spring Web MVC** application.
* It includes:
  + Embedded Tomcat Server
  + Spring MVC modules
  + JSON support
  + DispatcherServlet, etc.

**✅ 2. @Controller Annotation**

* This annotation is used on a class.
* It tells Spring that the class is a **Controller** (also called Request Handler).
* It will handle **web requests** (like URL calls from the browser).

**✅ 3. @RequestMapping Annotation**

* This is used to **map a URL pattern to a controller method**.
* When a client sends a request to a specific URL, the corresponding method is executed.
* Example:

@RequestMapping("/welcome")

public String showWelcomePage() { }

**✅ 4. Returning ModelAndView Object**

* Controller method returns a ModelAndView object.
* This object contains:
  + a) **Model data** → in the form of key-value pairs
  + b) **Logical view name** → without extension

**✅ 5. Accessing Model Data in View File**

* The data sent using Model (in the form of key-value)  
  can be **accessed in the view file** using the key.
* Example in JSP:

<h1>Hello ${username}</h1>

(Here username is the key used in Model)

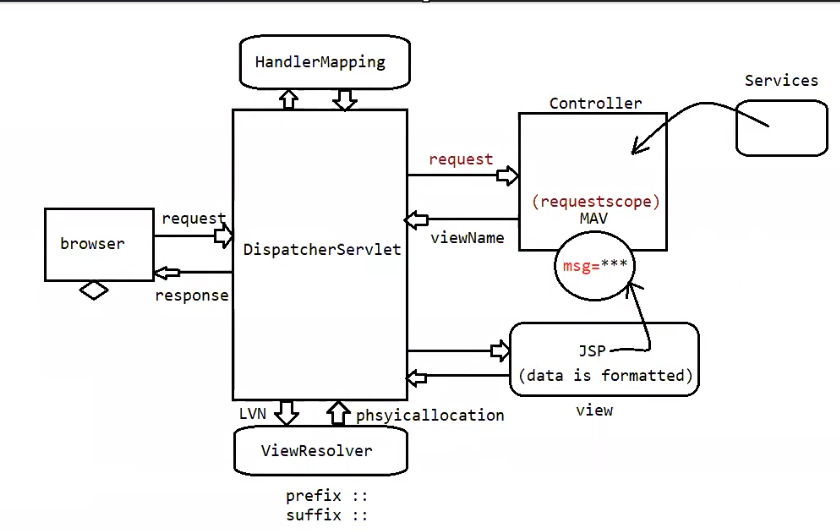
**✅ 6. @GetMapping Annotation**

* Shortcut for @RequestMapping(method = RequestMethod.GET)
* Used to bind a controller method to a **HTTP GET request**.
* Commonly used to **display data or forms**.

**✅ 7. Model Interface**

* Provided by Spring.
* It is used to **store data as key-value pairs**.
* This data is passed from **Controller to View (UI)**.
* View file uses this data to display on the screen.

Client Request → @Controller method → prepares data in Model → returns ModelAndView → View displays data



Eg: SpringWebMVC

Eg:SpringBootSimpleNavigationExample

### ✅ How Handler Methods Work in Spring MVC

When a **handler method** is called by the **DispatcherServlet**, it can return two types of data to tell the framework what to do next:

#### 1️⃣ Returning a **String**

* If the handler method returns a **String**, that String is treated as the **Logical View Name (LVN)**.
* Example:
* @RequestMapping(value="/", method = RequestMethod.GET)
* public String showHome() {
* return "home";
* }
  + Here, "home" is the **Logical View Name (LVN)**.
  + The **DispatcherServlet** uses this LVN and sends it to **ViewResolver**.
  + ViewResolver converts it into the actual JSP file location:

/WEB-INF/pages/home.jsp

* No data (model attributes) is sent unless we use a **Model parameter** or similar approach.

#### 2️⃣ Returning a **ModelAndView Object**

* This is an older way (legacy approach), but still supported.
* The method returns a **ModelAndView object** that contains:
  + **Model Data** (key-value pairs).
  + **Logical View Name (LVN)**.
* Example:
* @RequestMapping(value="/wish", method = RequestMethod.GET)
* public ModelAndView generateMessage() {
* String msg = service.generateWishMessage();
* ModelAndView mav = new ModelAndView();
* mav.addObject("wmg", msg); // Store data in Model
* mav.setViewName("display"); // Logical view name
* return mav;
* }
* What happens internally:
  + DispatcherServlet gets the **ModelAndView object**.
  + It puts the model data (wmg → message text) into **request scope**.
  + It sends the LVN "display" to the **ViewResolver**.
  + ViewResolver converts LVN to full JSP path:

/WEB-INF/pages/display.jsp

### ✅ Why Prefer String Return Type Over ModelAndView

👉 **1. Simpler and Clearer:**  
When we return a **String** from a handler method, it just tells the framework the name of the view (like a JSP page) to show.  
We don’t need to create a big object like **ModelAndView**.  
This makes the code easy to read and understand.

#### ✅ Example Code for Step 1:

@RequestMapping(value="/home", method = RequestMethod.GET)

public String homePage() {

return "home"; // Just return the view name "home"

}

👉 **2. Easier to Pass Data:**  
We can use the **Model** parameter in the method to send data to the view.  
There is no need to use extra steps or objects.

#### ✅ Example:

@RequestMapping(value="/welcome", method = RequestMethod.GET)

public String welcomePage(Model model) {

model.addAttribute("msg", "Welcome to our site!");

return "welcome";

}

🔍 **What Happens Here:**

* We pass a message "Welcome to our site!" using model.addAttribute().
* We return the view name "welcome".
* The framework shows the welcome.jsp page with the message.

👉 **3. Industry Practice:**  
In modern applications, developers prefer using **String return type + Model argument** because:  
✔ It is simple.  
✔ Easy to maintain.  
✔ Focuses only on what is needed (view name and data).

#### ✅ application.properties Settings

server.port=9999

spring.mvc.view.prefix=/WEB-INF/pages/

spring.mvc.view.suffix=.jsp

server.servlet.context-path=/SecondApp

* **server.port=9999** → Application will run on port 9999.
* **spring.mvc.view.prefix=/WEB-INF/pages/** → All views are in this folder.
* **spring.mvc.view.suffix=.jsp** → Every LVN will be converted to a .jsp file.
* **server.servlet.context-path=/SecondApp** → The base URL will be:

http://localhost:9999/SecondApp

### ✅ Simple Summary

| **Return Type** | **What is Sent to DispatcherServlet** | **Usage** |
| --- | --- | --- |
| String | LVN only | Modern, simpler way. |
| ModelAndView | LVN + Model Data | Legacy way. Useful when you want to combine model and view in one object. |

👉 Prefer **String + Model attributes** in modern applications, but **ModelAndView** is still supported for older apps.