### 5️⃣ ****ViewResolver****

* This component **takes the view name** from ModelAndView and **finds the actual view file** (like a JSP file).
* Example: If the controller returns studentList, the ViewResolver may convert it to WEB-INF/views/studentList.jsp.

✅ It works like a **translator** that finds the real path of the view page.

## ✅ What is ViewResolver in Spring Web MVC?

### 🔷 1. ****What is ViewResolver?****

* **ViewResolver** is a **predefined component** in Spring Web MVC.
* It helps the application **find the correct view file** (like JSP, HTML).
* It maps a **logical view name** (given by the Controller) to the **actual view file** (like success.jsp).

✅ **Key Point:**  
**Controller gives just a name**, and **ViewResolver finds the actual page.**

### 🔷 2. ****Why Do We Need ViewResolver?****

* In your controller, you don’t need to write full file names like "/WEB-INF/views/success.jsp".
* You just return a simple name like "success".
* **ViewResolver will automatically add the path and file extension.**

✅ **Example:**  
If controller returns "success"  
➡️ ViewResolver will convert it to /WEB-INF/views/success.jsp

### 🔷 3. ****What Does ViewResolver Do Exactly?****

* It **adds prefix (folder path)** and **suffix (file extension)** to the view name.
* Then it checks if that file exists and sends it to the DispatcherServlet.

✅ **Example Configuration (in XML):**

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/views/" />

<property name="suffix" value=".jsp" />

</bean>

* This means:
  + "success" → /WEB-INF/views/success.jsp
  + "home" → /WEB-INF/views/home.jsp

## ✅ ViewResolver in Annotation-Based Approach

### 🔷 1. ****Is there an annotation for ViewResolver?****

* ❌ There is **no direct annotation like @ViewResolver**.
* ✅ But in annotation-based approach, we use **Java Configuration Class** (with @Configuration) to define and register a **ViewResolver bean**.

### 🔷 2. ****How do we configure ViewResolver without XML?****

* In annotation-based config, we **write a Java class** instead of an XML file.
* We **use @Configuration and @Bean annotations** to declare the InternalResourceViewResolver.

✅ Example:

@Configuration

public class WebMvcConfig implements WebMvcConfigurer {

@Bean

public InternalResourceViewResolver viewResolver() {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

return resolver;

}

}

### 🔷 3. ****Simple Explanation of the Code Above:****

| **Line** | **Meaning** |
| --- | --- |
| @Configuration | Tells Spring: This class contains configuration code |
| public class WebMvcConfig | Java class to hold view resolver and other web-related setup |
| @Bean | Tells Spring: Create a bean for this method’s return type |
| InternalResourceViewResolver | This object helps in locating JSP files |
| setPrefix("/WEB-INF/views/") | Views are inside this folder |
| setSuffix(".jsp") | All views are JSP files (like home.jsp, welcome.jsp) |

### 🔷 4. ****Where Do We Put This Configuration Class?****

* Place this class in a package that gets **scanned** by Spring.
* If using Spring Boot, it gets picked up automatically.

### 🔷 Final Summary:

| **Topic** | **Details** |
| --- | --- |
| **ViewResolver with XML** | Used in older, XML-based Spring MVC projects |
| **ViewResolver with Java** | ✅ Used in real-time, annotation-based Spring MVC |
| Direct Annotation? | ❌ No @ViewResolver — we use @Bean inside @Configuration class |
| Advantage | No need for XML files; configuration is in pure Java and easier to manage |

### 🔷 4. ****Types of ViewResolvers in Spring****

There are many built-in ViewResolver classes in Spring:

### 🔷 Commonly Used ViewResolver Classes in Spring MVC

| **ViewResolver Class** | **Description** | **Real-Time Usage** |
| --- | --- | --- |
| **InternalResourceViewResolver** | ✅ Most commonly used in real-time projects. It resolves **JSP files** located inside the project using prefix and suffix. | ✅ **Used the most** |
| **XmlViewResolver** | Takes view names and maps them to actual views using an **external XML file**. | ❌ **Rarely used** in modern apps |
| **UrlBasedViewResolver** | Supports **multiple view types** (JSP, PDF, Excel, etc.) using URL patterns. | 🔶 Used in **advanced or multi-view** apps |
| **BeanNameViewResolver** | Resolves view by **matching bean name** in the config file or code. | ❌ **Rarely used** |

✅ **Key Point:**  
We usually use **InternalResourceViewResolver** in basic web apps (JSP-based).

### 🔷 5. ****When Does ViewResolver Get Involved?****

1. **Controller** returns ModelAndView with a **logical view name**.
2. **DispatcherServlet** gives that name to the **ViewResolver**.
3. ViewResolver finds the **actual JSP/HTML file** using prefix and suffix.
4. DispatcherServlet renders that page to the **user**.

✅ **Key Point:**  
ViewResolver works **after the controller finishes** its job, to show the correct page.

### 6️⃣ ****View****

* This is the **final HTML or JSP page** shown to the user.
* It uses the **model data** to display dynamic content.

✅ It’s like the **result screen** that the user sees in the browser.

## 🔷 **View (in Spring Web MVC)**

### ✅ What is a View?

* A **View** is the final output page that the **user sees** in the browser.
* It shows the **data returned by the controller**, in a format that is easy to understand.
* It is the **presentation layer** in the **MVC design pattern**.

### ✅ Where does the View come in?

| **Flow Step** | **Explanation** |
| --- | --- |
| Controller processes data | After business logic, Controller returns data |
| Controller returns view name | Controller gives a **logical view name** like "welcome" |
| ViewResolver steps in | It finds the actual **view file** (like welcome.jsp) |
| View is rendered | Final output is shown to the user |

### ✅ Technologies used as View in Spring MVC:

| **View Type** | **Description** | **Real-Time Use** |
| --- | --- | --- |
| **JSP** | ✅ Most commonly used in Spring Web MVC | ⭐ Yes |
| Thymeleaf | Used in Spring Boot projects, supports HTML5 | 🔶 Sometimes |
| PDF View | Used for generating downloadable PDFs | 🔶 Special cases |
| Excel View | Used for exporting Excel files | 🔶 Special cases |
| FreeMarker | Template engine alternative to JSP | ❌ Rarely used |

### ✅ Example of a View File (JSP):

<!-- File: /WEB-INF/views/welcome.jsp -->

<html>

<body>

<h2>Welcome, ${userName}!</h2>

</body>

</html>

| **Part** | **Explanation** |
| --- | --- |
| ${userName} | Dynamic data passed from Controller (Model) |
| welcome.jsp | File shown to user when view name is "welcome" |

### ✅ How is data sent to the View?

* The **Controller** uses a Model or ModelAndView to **send data**.
* This data is accessed using **${}** syntax in JSP.

### ✅ Folder structure (common in real-time projects):

/WEB-INF/

└── views/

└── welcome.jsp

└── dashboard.jsp

| **Folder/File** | **Purpose** |
| --- | --- |
| /WEB-INF/views/ | Stores all JSP view files |
| welcome.jsp | View file matched to logical view name "welcome" |

### ✅ Final Summary:

| **Concept** | **Description** |
| --- | --- |
| View | Final output shown to user |
| View file | JSP, Thymeleaf, etc. |
| Comes from | Controller returns logical name → ViewResolver → View file |
| Most used | ✅ JSP (with InternalResourceViewResolver) |
| View format | Uses HTML + dynamic data (${}) from controller |

## ✅ How SpringMVC/SpringBootMVC Handles Private JSPs

* In Spring, JSPs are kept in WEB-INF/pages/.
* Since browser cannot access this directly, **DispatcherServlet + ViewResolver** is used to forward request internally.

### Example Configuration (application.properties)

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

spring.mvc.view.suffix = .jsp

So:

* prefix = folder location of JSPs (/WEB-INF/pages/)
* suffix = file extension (.jsp)
* **LVN (Logical View Name)** = controller’s return value (e.g., "login")

➡️ Internally, Spring will map:

LVN "login"

↓

/WEB-INF/pages/login.jsp

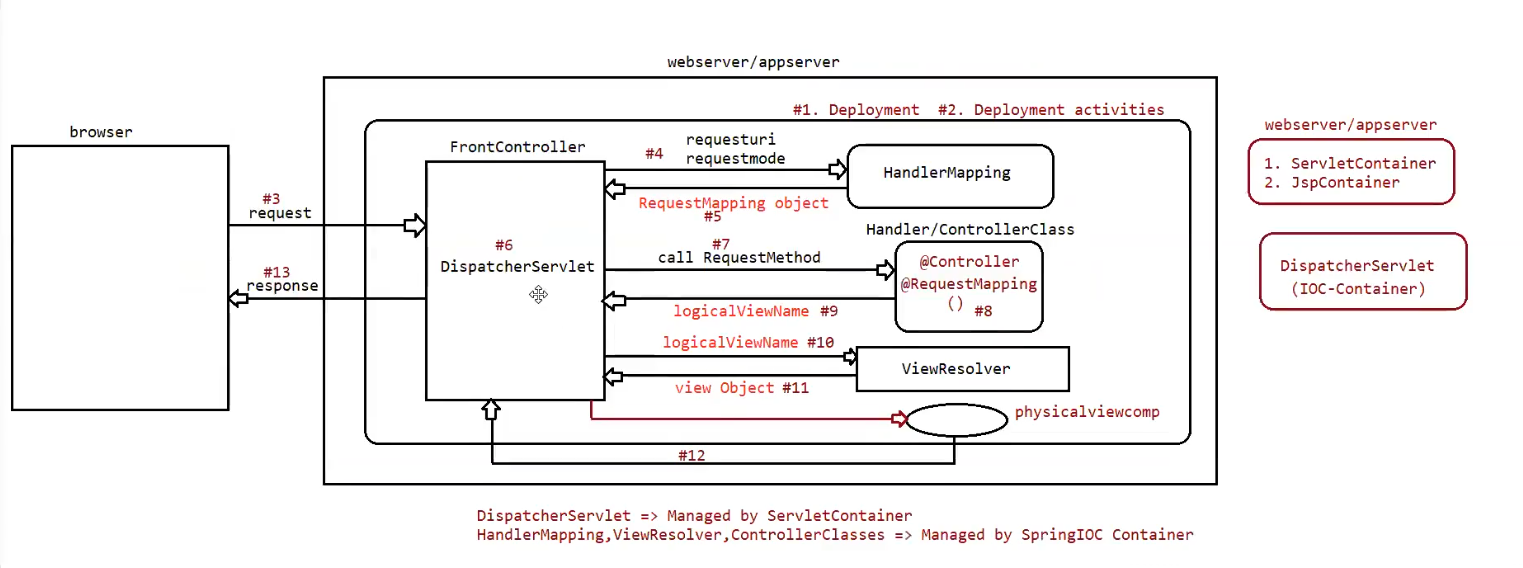
But the **browser will only see /login** (not the physical JSP path).

## ✅ Quick Analogy

Think of JSP in **private area** as a **backstage in a theater 🎭**:

* Audience (browser) never directly enters backstage.
* They only see the final show (rendered HTML).
* Backstage (JSP) is controlled by stage manager (DispatcherServlet).

## 🔁 Full Flow Summary (Step-by-Step):

****

## 🔄 SpringMVC / SpringBootMVC Request Handling Flow

### 1. ****Deployment****

* Programmer deploys the application (WAR/JAR) into a **web server / servlet container**.
* During deployment:
  + **IoC container** is created.
  + **DispatcherServlet** is registered.
  + Spring pre-instantiates all **singleton beans** like Controllers, Service, DAO, ViewResolvers, HandlerMapping etc.
  + Dependencies are injected automatically.

### 2. ****Request Handling****

1. Browser sends an HTTP request (ex: /login).
2. **DispatcherServlet** (Front Controller) traps the request.
   * It applies common services like logging, auditing, security checks, etc.
3. DispatcherServlet consults **HandlerMapping**.
   * Finds which Controller & which method should handle this request (@RequestMapping info).
4. DispatcherServlet gets the **Controller bean** from IoC container.
   * Prepares necessary method arguments (like @RequestParam, @PathVariable, Model, Session, etc.).
5. DispatcherServlet invokes the **handler method** of the Controller.
6. Controller method:
   * Performs logic directly **or** calls **Service/DAO** for business logic & DB operations.
   * Stores result in a **scope object** (usually request).
   * Returns **LVN (Logical View Name)** back to DispatcherServlet.

### 3. ****Response Handling****

1. DispatcherServlet gives LVN to **ViewResolver**.
2. ViewResolver maps LVN → actual **physical view** (like JSP, Thymeleaf, FreeMarker, etc.).
   * Returns a **View object** with physical location.
3. DispatcherServlet forwards request & model data to the **view**.
   * View formats the data (presentation logic).
4. DispatcherServlet sends the **final HTML response** back to browser.

## 🔗 Simplified Diagram Flow

Browser → DispatcherServlet (Front Controller)

↓

HandlerMapping → Finds Controller & Method

↓

Controller → (Service → DAO → DB if needed)

↓

Returns LVN (Logical View Name)

↓

DispatcherServlet → ViewResolver

↓

Physical View (JSP/HTML/Thymeleaf)

↓

DispatcherServlet → Browser Response

# 🔹 Why Do We Need Servlet Configuration?

A servlet is just a **Java class** that extends HttpServlet.  
If you don’t configure it, it’s like writing a letter but never telling the post office where to deliver it.

Servlet configuration is basically a **communication channel** between:

* The **developer** (who wrote the servlet)
* The **container** (Tomcat/Jetty/GlassFish that runs the servlet)
* The **outside world** (browsers sending requests like /login, /products)

Let’s break down the full need:

## 1. **Telling the Container “This is a Servlet”**

* The container manages many things in a web app: JSPs, HTML files, static images, servlets, filters, listeners, etc.
* If you just put a Java class in your project, the container won’t treat it as a servlet.
* You must **configure** it so the container knows:
  + This class is a servlet.
  + It should be created and managed in a special way (lifecycle).

Without this, your servlet is just another .class file, ignored by the container.

## 2. **Mapping URLs to Servlets**

* Every web request comes with a URL: /login, /register, /dashboard.
* The container receives the request, but it doesn’t know **which servlet should handle it**.
* Configuration maps a URL (or pattern) → Servlet.  
  Example:
  + /login → LoginServlet
  + /register → RegisterServlet
  + /app/\* → AppServlet

👉 Without mapping, typing /login in the browser would give a 404 (not found), because the container doesn’t know where to send it.

## 3. **Passing Extra Information (Init Parameters)**

* Often, servlets need configuration values like:
  + Database URL
  + Maximum number of users allowed
  + Admin email address
* Hardcoding these inside Java code is bad (you’d have to recompile every time something changes).
* Instead, configuration lets you pass these as **init parameters**.
* The servlet can then read them at runtime.

👉 This makes applications more **flexible** and easier to maintain.

## 4. **Controlling Lifecycle**

* By default, a servlet is created only when the first request comes in.
* But sometimes you want it **ready as soon as the application starts** (e.g., a servlet that loads cache data).
* Configuration lets you say:
  + load-on-startup = 1 → Load immediately.
  + (no value) → Load lazily, when first requested.

This is crucial for performance tuning.

## 5. **Handling Multiple URLs with One Servlet**

* Sometimes one servlet should handle multiple request patterns.
* Example: \*.do or /app/\*.
* Configuration lets you specify multiple mappings easily.

This is exactly what happens in **FrontController**:

* One servlet (FrontController) is mapped to catch all requests like \*.do or /.
* Then it decides internally which handler should process the request.

## 6. **Error Handling**

* Configuration can say:
  + If 404 (Not Found) happens → show error404.jsp.
  + If 500 (Internal Error) happens → show error500.jsp.
* This makes the app user-friendly, instead of showing raw errors.

## 7. **Flexibility Across Approaches**

Servlet configuration is also important because **different styles of applications use different configuration methods**:

* **XML (Declarative)** → Used in traditional apps.
* **Annotations** → Used in modern apps for user-defined servlets.
* **Programmatic** → Used in frameworks (like Spring Boot) where XML isn’t allowed.

No matter the method, the **need** is always the same: to connect the servlet class with the container and requests.

# 🔹 In Very Simple Words

Servlet configuration is needed because:

1. The container must know which classes are servlets.
2. Requests (URLs) must be mapped to the right servlet.
3. Servlets may need extra info (init params).
4. We must control when servlets are created (startup vs. first request).
5. We may want one servlet to handle many URLs.
6. We want friendly error handling.
7. It gives flexibility depending on whether you use XML, annotations, or programmatic style.

👉 Without configuration, a servlet is just a plain Java class, and the container cannot use it in the web application.

Would you like me to now **connect this with the FrontController servlet in MVC2** (showing why its configuration is super important)?

## 🔹 3 Ways of Servlet Configuration

### 1. ****Declarative Approach (XML-based)****

* We declare servlet details in **web.xml** file.
* Mostly used when XML configuration is allowed.
* Good for **predefined servlets** (like DispatcherServlet).
* Example use: XML-driven **Spring MVC apps**.

👉 Example snippet:

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

### 2. ****Annotation Driven Approach****

* Use **@WebServlet** annotation directly above the servlet class.
* Mostly for **user-defined servlets**.
* Cleaner and avoids web.xml.
* Example use: Model1, MVC1, MVC2 apps with custom FrontController.

👉 Example snippet:

@WebServlet("/login")

public class LoginServlet extends HttpServlet {

protected void doGet(HttpServletRequest req, HttpServletResponse resp) {

resp.getWriter().write("Login Page");

}

}

### 3. ****Programmatic Approach (Code-based)****

* Configure servlet using **ServletContext.addServlet()** inside Java code.
* Very useful when **XML configuration is not allowed** (like in Spring Boot).
* Mostly used for **predefined servlets** such as DispatcherServlet.
* Example use: 100% code-driven Spring MVC apps, Spring Boot MVC apps.

👉 Example snippet:

public class MyAppInitializer implements ServletContextListener {

@Override

public void contextInitialized(ServletContextEvent sce) {

ServletContext context = sce.getServletContext();

ServletRegistration.Dynamic dispatcher =

context.addServlet("dispatcher", new DispatcherServlet());

dispatcher.addMapping("/");

}

}

# 🔹 Different Approaches to Develop Spring MVC Applications

### 1. ****Declarative Configuration (XML Driven)****

* Here everything is configured inside web.xml and other XML files.
* Example: web.xml tells the container:
  + "Hey, there is a servlet called DispatcherServlet."
  + "Map all requests like \*.do or /app/\* to this servlet."
* Then, extra XML files (applicationContext.xml, spring-servlet.xml) hold bean definitions, controller mappings, and view resolvers.

👉 This was the **oldest style**, very verbose, but still clear and explicit.

### 2. ****Annotation + XML Driven Configuration****

* In this style, part of the configuration is in XML, part is in Java annotations.
* Example:
  + DispatcherServlet is still declared in web.xml.
  + Controllers, Services, DAOs can use annotations like @Controller, @Service, @Repository.
  + Component scanning (<context:component-scan>) in XML tells Spring:
    - "Look in this package for annotated classes and register them as beans."

👉 This reduced XML size and allowed developers to use annotations for business logic.

### 3. ****100% Code-Driven Configuration (Java Config)****

* Here, **no XML at all**.
* The configuration is done using pure Java classes with annotations like @Configuration, @EnableWebMvc, @Bean.
* DispatcherServlet is registered **programmatically** using ServletContext.addServlet().
* All beans, controllers, view resolvers are declared in Java config classes.

👉 This is more modern, flexible, and easier to maintain compared to XML.

### 4. ****Spring Boot MVC Applications****

* The easiest and most modern approach.
* Spring Boot does the configuration for you (auto-configuration).
* You don’t declare DispatcherServlet anywhere — Spring Boot automatically registers it **programmatically** behind the scenes.
* You only focus on writing controllers (@RestController, @Controller) and application logic.

👉 This is the most used style in industry today because it saves time and reduces boilerplate code.

# 🔹 Common Notes

* In **1st and 2nd approaches**: DispatcherServlet is declared in web.xml (Declarative approach).
* In **3rd and 4th approaches**: DispatcherServlet is registered using **Programmatic approach** (no web.xml).
* The DispatcherServlet **does not** have @WebServlet annotation inside its source code.
  + It always needs external configuration (XML or Programmatic).
* No matter which approach is used, **all Spring MVC / Spring Boot MVC applications follow MVC2 architecture**.
* DispatcherServlet works as the **FrontController** design pattern.

✅ So in very simple words:  
Different approaches only change **how we configure things** (XML vs Annotation vs Java Code vs Boot Auto Config).  
But inside, the **architecture (MVC2)** and the **FrontController (DispatcherServlet)** concept remain the same.