### 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

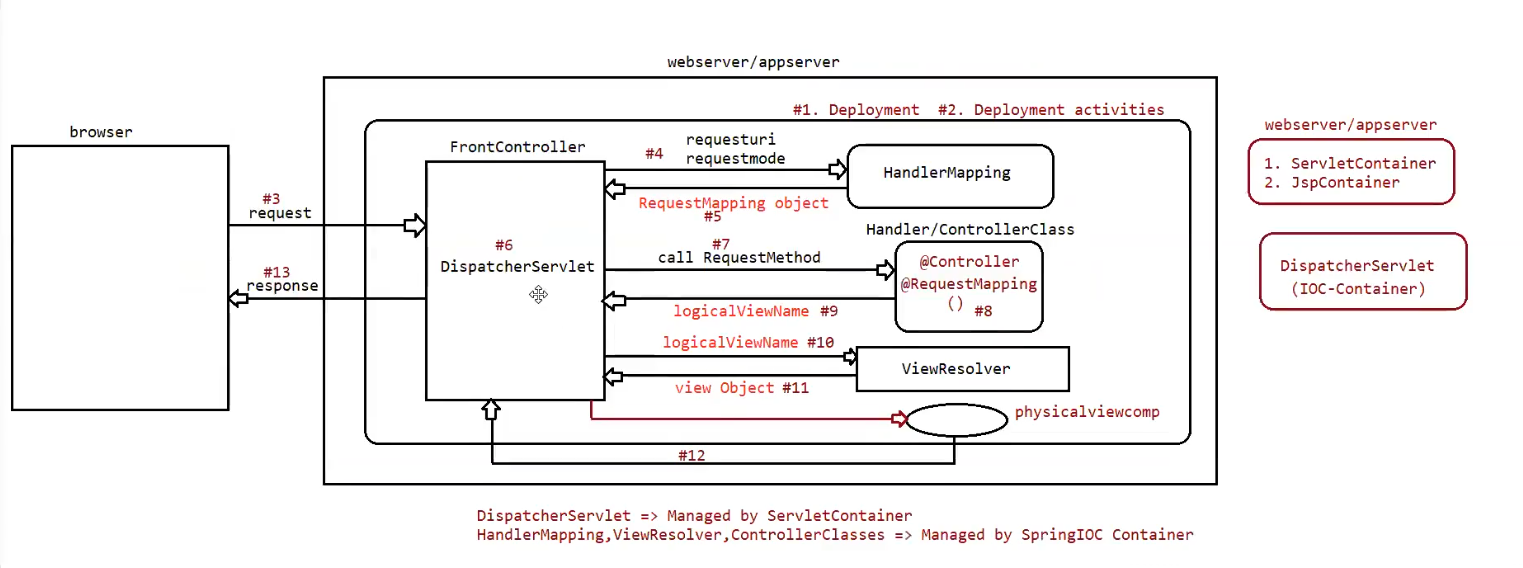
But the **browser will only see /login** (not the physical JSP path).

## ✅ 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

## 1️⃣ Deployment of SpringBoot MVC App

✅ **What Happens:**

* The programmer writes the SpringBoot MVC application.
* Packages it as a **JAR file** (standalone) or **WAR file** (for external web server).
* Runs the app using:

SpringApplication.run(StudentApp.class, args);

* The app becomes ready to handle incoming requests.

✅ **Example Scenario:**

@SpringBootApplication

public class StudentApp {

public static void main(String[] args) {

SpringApplication.run(StudentApp.class, args);

}

}

This code starts an embedded Tomcat server automatically.

✅ **Example Detailed Note:**  
At this moment:

* Tomcat server starts running → listens on port 8080.
* Application becomes available at:

http://localhost:8080

✅ **Simple Summary of Step 1 Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Runs main() | SpringBoot auto-starts embedded Tomcat. |
| 2. | App becomes ready | Now listens for requests on localhost:8080 |

## 2️⃣ Startup Process

✅ **What Happens:**

* SpringBoot performs startup tasks automatically:  
  → Creates **IOC Container** (where Spring keeps all beans).  
  → Registers **DispatcherServlet** (Front Controller).  
  → Creates singleton beans: Controller, Service, DAO, ViewResolver.  
  → Performs **Dependency Injection** automatically.

✅ **Example Scenario:**

@Controller

public class StudentController {

@Autowired

private StudentService studentService;

}

SpringBoot creates both StudentController and StudentService and connects them automatically.

✅ **Example Detailed Note:**  
During startup:

IOC Container creates:

- StudentController instance

- StudentService instance

It injects studentService into studentController automatically.

👉 No need for manual object creation like new StudentService().

✅ **Simple Summary of Step 2 Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | IOC creates beans | Spring creates StudentController and StudentService automatically. |
| 2. | Performs Dependency Injection | Connects StudentService inside StudentController without extra code. |

## 3️⃣ Browser Sends Request

✅ **What Happens:**

* User opens browser and types URL:

http://localhost:8080/students

* A **GET HTTP Request** is sent to SpringBoot app.

✅ **Example Scenario:**

GET /students HTTP/1.1

Host: localhost:8080

✅ **Example Detailed Note:**  
When user hits Enter in browser:

* Browser sends request →  
  DispatcherServlet is ready to receive it.

👉 This is the first interaction between user and the app.

✅ **Simple Summary of Step 3 Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Browser makes GET request | Sends "GET /students" to app |
| 2. | DispatcherServlet receives it | Ready to process request |

## 4️⃣ DispatcherServlet Receives Request

✅ **What Happens:**

* DispatcherServlet catches the incoming HTTP request first.
* Applies system-level services:  
  → Logging  
  → Auditing  
  → Tracking sessions

✅ **Example Scenario:**

INFO: Received GET /students request at 12:00 PM

✅ **Example Detailed Note:**  
Before processing the request:

* Logs record when and which URL was accessed.
* Sessions may be created/updated.
* Helps developers monitor request flow.

✅ **Simple Summary of Step 4 Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Logs incoming request | Records "Received GET /students" |
| 2. | Tracks user session | Useful for debugging and auditing |

## 5️⃣ Finding the Handler Method

✅ **What Happens:**

* DispatcherServlet asks **HandlerMapping**:  
  “Which Controller method handles this URL?”
* It uses @RequestMapping annotations to match URL.

✅ **Example Scenario:**

@RequestMapping("/students")

public String getStudents(Model model) {

// Matched by HandlerMapping

}

✅ **Example Detailed Note:**  
Using reflection:

* Spring matches /students to getStudents() method.
* Finds method automatically → No extra programmer work needed.

✅ **Simple Summary of Step 5 Example**

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Match URL to method | /students → getStudents(model) |
| 2. | Reflection used | Spring finds method by itself |

### 6️⃣ ****Prepare Controller Method Call****

#### ✅ What Happens:

* The **DispatcherServlet** already knows the correct Controller method to call (from step 5).
* Now, it needs to:
  1. Get the **Controller object** (the actual instance of StudentController) from the IOC Container.
  2. Prepare the method arguments that the handler method needs.  
     These arguments could be:  
     → A Model object (to store data that will be sent to the view).  
     → Request parameters (if user submitted form data like ?name=John).  
     → Path variables (like /students/1).  
     → HTTP request objects.

#### ✅ Example Scenario:

The controller method looks like this:

@RequestMapping("/students")

public String getStudents(Model model) {

List<Student> students = studentService.getAllStudents();

model.addAttribute("studentList", students);

return "student-list";

}

#### ✅ What Arguments Does It Need?

* Model model → An object to hold data that will be sent to the view later.

#### ✅ What the DispatcherServlet Does in Detail:

* Step 1:  
  It asks the IOC Container →  
  "Give me the StudentController bean instance."

Example result:

StudentController studentController = applicationContext.getBean(StudentController.class);

* Step 2:  
  It prepares the method arguments.  
  Since the method needs a Model object, DispatcherServlet creates it:
* Model model = new ExtendedModelMap();

👉 This object is empty at first but will store data later.

#### ✅ Why Is This Important?

* Without preparing these arguments, the Controller method could not run properly.
* The **Model object acts like a bag**, where the controller can store the data (like list of students) so that the view (JSP) can later read it.

#### 📚 Example Detailed Note:

Before calling:

studentController.getStudents(model);

At this moment:

* model is an empty object → {}.
* After the method runs:
* model.addAttribute("studentList", students);

→ Now the model looks like:

{

"studentList": [Student{name='John'}, Student{name='Jane'}]

}

Later, this model will be available to the view.

### ✅ Simple Summary of Step 6 Example

| **Step** | **Example Action** | **Simple Explanation** |
| --- | --- | --- |
| 1. | Get Controller bean | Spring provides StudentController object automatically. |
| 2. | Create Arguments | Prepares Model model = new ExtendedModelMap(). |
| 3. | Call Controller method | studentController.getStudents(model) → Now the model holds data. |