

**✅ Simple Explanation of How Spring MVC Works (Using the Diagram)**

**1️⃣ Client Sends Request**

* The **client (browser)** sends an HTTP **request** to the application.

Example:

http://localhost:9999/SecondApp/welcome

**2️⃣ DispatcherServlet Receives the Request**

* The **DispatcherServlet** is the front controller.
* It receives the **request** first.
* Its job is to figure out what to do next.

**3️⃣ HandlerMapping Finds the Right Handler Method**

* DispatcherServlet asks **HandlerMapping**:

"Which handler method should I call for this request URI and request mode (GET/POST)?"

* **HandlerMapping** looks for the matching method based on the request URI and HTTP method.
* Once found, it returns the method to the DispatcherServlet.

**4️⃣ Handler Method Executes and Provides Data**

* The selected **handler method** (inside a controller) executes.
* It creates some **data to send to the view**.  
  Example:
* model.addAttribute("message", "Welcome to My App");
* return "home"; // Logical View Name (LVN)
* The data is stored in something called **BindingAwareModelMap**, which acts as a container.
  + Think of it like a simple key → value storage, where:
    - Key = "message"
    - Value = "Welcome to My App"
* The data lives in **request scope** → It is available only for this request.

**5️⃣ DispatcherServlet Sends LVN to ViewResolver**

* The handler method returns a **Logical View Name (LVN)** like "home".
* The **DispatcherServlet** sends this LVN to **ViewResolver**.

**6️⃣ ViewResolver Converts LVN to Actual JSP Path**

* ViewResolver takes LVN "home" and converts it into the real JSP file location using the prefix and suffix from application.properties.

Example:

* + LVN: "home" → /WEB-INF/pages/home.jsp

**7️⃣ View Renders the Data**

* The **View (.jsp file)** takes the data from **BindingAwareModelMap (request scope)** and displays it.

Example in JSP:

Message: ${message}

Will display:

Message: Welcome to My App

**8️⃣ Response Sent to Client**

* The final HTML page generated by JSP is sent back by the **DispatcherServlet** as the **response** to the client (browser).

**✅ Simple Summary of Data Flow**

| **Step** | **What Happens** |
| --- | --- |
| 1 | Client sends HTTP request to DispatcherServlet |
| 2 | DispatcherServlet asks HandlerMapping → Finds correct controller method |
| 3 | Handler method executes → Adds data to BindingAwareModelMap + returns LVN |
| 4 | DispatcherServlet sends LVN to ViewResolver |
| 5 | ViewResolver resolves LVN to actual JSP location |
| 6 | View (.jsp) reads data from request scope (BindingAwareModelMap) and renders HTML |
| 7 | DispatcherServlet sends final HTML response back to client |

**✅ Key Concepts in Simple Words**

* ✅ **BindingAwareModelMap** → Simple place where key → value data is stored (like a small box).
* ✅ **Logical View Name (LVN)** → A simple name like "home" that points to a JSP page.
* ✅ **ViewResolver** → Converts LVN into real JSP path.
* ✅ **HandlerMapping** → Decides which controller method should handle the request.

**✅ 1️⃣ Using Model Interface**

👉 Example:

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

public String generateMessage(Model model) {

System.out.println("SharedMemory object class name is :: " + model.getClass().getName());

String msg = service.generateWishMessage();

model.addAttribute("wmg", msg); // Add data to model

return "display"; // View: /WEB-INF/pages/display.jsp

}

* ✅ What happens here:
  + A special object called **Model** is provided by Spring.
  + We use model.addAttribute("wmg", msg) to add data into a simple key → value map.
  + The key is "wmg" and the value is the wish message string.

**✅ 2️⃣ Using Map<String, Object>**

👉 Example:

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

public String generateMessage(Map<String, Object> model) {

System.out.println("SharedMemory object class name is :: " + model.getClass().getName());

String msg = service.generateWishMessage();

model.put("wmg", msg); // Same as Model but using Map directly

return "display"; // View: /WEB-INF/pages/display.jsp

}

* ✅ What happens here:
  + Instead of Model, we use a normal **Map<String, Object>**.
  + We use model.put("wmg", msg) to store data.
  + Functionally, this works exactly the same as Model.

✅ 3️⃣ Using **void Return Type**

### ✅ When You Use Void as Return Type:

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

public void generateMessage(Model model) {

String msg = service.generateWishMessage();

model.addAttribute("wmg", msg);

// No return type (void)

}

### ✅ How Spring Decides Which View to Use (Behind the Scenes)

* Since the handler method returns **void**, there is no explicit view name (Logical View Name, LVN) provided.
* ✅ Internally, Spring uses a special component called  
  ➔ **RequestToViewNameTranslator**

#### 🔧 What does it do?

* It takes the incoming **request URI** (e.g., /wish) and automatically converts it into the Logical View Name.
* Example rule:
* Incoming request: /wish
* → Remove leading slash → "wish"
* → ViewResolver looks for /WEB-INF/pages/wish.jsp

### ✅ Example Full Flow

1. Client makes request:
2. GET http://localhost:9999/SecondApp/wish
3. Handler method executes:
4. public void generateMessage(Model model) {
5. model.addAttribute("wmg", "Good Morning User");
6. // No return type
7. }
8. Spring calls **RequestToViewNameTranslator**.
   * It converts /wish → "wish" (LVN).
9. ViewResolver resolves "wish" →
10. /WEB-INF/pages/wish.jsp
11. Model data ("wmg" → "Good Morning User") is available in **request scope**.
12. Final rendered JSP page is returned to the client.

### ✅ Important Notes

* ⚠ This behavior happens only if you don’t return anything (void).
* ✅ It is useful for very simple situations where the view name is the same as the URL path.
* ❌ Not recommended in industry because:
  + It is implicit and less readable.
  + Hard to know which view will be returned just by looking at the method.
  + For clarity, it is better to return a **String (LVN)** explicitly.

**✅ ✅ Most Common Industry Practice →**

**✅ Return Type: String + Model Parameter**

**✔ Why It Is Preferred:**

1. ✅ **Clean and Simple Code**
   * Easy to read and maintain.
   * Clear separation of concerns.

Example Industry-Style Code:

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

public String generateMessage(Model model) {

String msg = service.generateWishMessage();

model.addAttribute("wmg", msg);

return "display"; // Logical View Name (LVN)

}

1. ✅ **Better Testing Support**
   * Easy to test the returned view name and model attributes in unit tests.
2. ✅ **Good Support for Spring Features**
   * Works smoothly with @ModelAttribute, form data binding, validation (@Valid), RedirectAttributes, etc.
3. ✅ **Explicit and Predictable**
   * Returning String makes the LVN explicit.
   * Easy to understand exactly what view will be rendered.

**⚠ Avoid in Industry (Unless Special Case):**

| **Approach** | **Why Not Preferred** |
| --- | --- |
| ModelAndView | More verbose and mixes data + view in one object → Harder to maintain. Mostly legacy code. |
| void return type | Not explicit → LVN is auto-generated → Not good for clarity and maintainability. Only used for very simple cases where explicit LVN is not needed. |
| Map<String, Object> | Works, but not commonly used because Model interface is more semantically clear in MVC context. |

**✅ Behind the Scenes**

* ✅ **BindingAwareModelMap** is a special internal object used by Spring.
  + Acts as a shared container (SharedMemory) for storing key → value pairs.
  + Lives in **request scope** → Exists only for that particular HTTP request.
* ✅ When handler methods receive a **Model, Map<String, Object>, or ModelMap** parameter, they all point to the same shared object.
* ✅ The data we put in the model (like "wmg" → msg) is available to the view (JSP).

**✅ Important Note About InternalResourceViewResolver**

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

spring.mvc.view.suffix=.jsp

* ✅ This means all JSP views are expected to be in the same folder /WEB-INF/pages/ and all use .jsp.
* ✅ If you want to use different technologies (like Thymeleaf, FreeMarker, etc.), or store views in different locations:
  + You need to use **Multiple View Resolvers**.

**✅ Simple Summary Table**

| **Method** | **How It Works** | **Example LVN** |
| --- | --- | --- |
| Model | Use model.addAttribute() | Return "display" → /WEB-INF/pages/display.jsp |
| Map<String, Object> | Use model.put() | Same as Model approach |
| void return type | Uses RequestToViewNameTranslator → LVN becomes method path | Request /wish → LVN "wish" → /WEB-INF/pages/wish.jsp |

## ✅ Sending Response Directly to Browser Without Using DispatcherServlet

Normally, Spring MVC follows this flow:

1. Handler method returns a **view name (String)**.
2. DispatcherServlet sends that view name to ViewResolver.
3. ViewResolver finds the JSP.
4. Data in **Model (BindingAwareModelMap)** is available in the JSP.

### ✅ But What If We Want to Send Response Directly?

👉 Example – Returning HTML directly to browser using **HttpServletResponse**:

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

public String generateWishMessage(HttpServletResponse response) throws Exception {

String msg = service.generateWishMessage();

PrintWriter out = response.getWriter();

response.setContentType("text/html");

out.println("<b>WishMessage is :: " + msg + "</b>");

return null; // No view name is used

}

Or better with void return type:

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

public void generateWishMessage(HttpServletResponse response) throws Exception {

String msg = service.generateWishMessage();

PrintWriter out = response.getWriter();

response.setContentType("text/html");

out.println("<b>WishMessage is :: " + msg + "</b>");

// No return value needed

}

#### ✅ Why Is This Useful?

* ✅ Useful when sending custom content directly to the browser (like custom HTML or file download).

Example:

* + File Download (CSV, PDF, etc.)
  + Direct HTML generation in special cases.
* ❗ This skips DispatcherServlet → ViewResolver → JSP flow.

## ✅ Why Not Use Model as Return Type?

👉 Example – Bad Practice:

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

public Model generateWishMessage() {

String msg = service.generateWishMessage();

Model model = new ExtendedModelMap(); // Manually created Model

model.addAttribute("wmg", msg);

return model;

}

#### ⚠ Why This Is Bad Practice:

a. ❌ We should not manually create Model objects in handler methods.

b. ❌ The **BindingAwareModelMap** is already created internally by Spring → creating another object wastes memory.

c. ❌ We have no control over Logical View Name (LVN):

* Since only Model is returned, Spring will automatically take request URI as LVN.
* This is not clear and makes the behavior unpredictable.

## ✅ Why Is It Good to Take Model or Map as a Parameter?

👉 Example – Good Practice (Recommended Way):

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

public String generateWishMessage(Map<String, Object> model) {

String msg = service.generateWishMessage();

model.put("wmg", msg);

return "display"; // Clear Logical View Name (LVN)

}

#### ✔ Why This Is Good Practice:

a. ✅ We have full control over which view to return (clear LVN).

b. ✅ The shared memory (BindingAwareModelMap) is created and managed by DispatcherServlet automatically.

c. ✅ We do not manually create Model objects → No resource waste.

d. ✅ The Model or Map parameter is directly linked to that shared memory → Clean, non-invasive, and efficient.

## ✅ Simple Summary Table

| **Approach** | **Why It’s Good or Bad** |
| --- | --- |
| Return Model object | ❌ Bad – Waste of resources + No control over view name |
| Return void + use HttpServletResponse | ✅ Useful for file download or direct response |
| Return String + Model/Map as parameter | ✅ Best practice – Clear LVN, efficient, easy to maintain |
| Use Map<String, Object> | ✅ Valid – Makes the method “non-invasive” (just adding data) |
| Use Model parameter | ✅ Recommended – Semantically correct and clearer |

## ✅ Final Simple Thought

👉 Always prefer this style in industry projects:

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

public String generateWishMessage(Model model) {

String msg = service.generateWishMessage();

model.addAttribute("wmg", msg);

return "display"; // Clear and explicit view

}

👉 Use void + HttpServletResponse when you want to send direct data (like file download).

⚠ Never use Model as return type in handler methods → It’s bad practice.

### 🔹 @RequestMapping

* This is used to map (or connect) a method in your controller to a **web request**.
* By default, it **handles GET requests** (unless you specify another type).
* It is used like this:

@RequestMapping("/hello")

public String sayHello() {

return "hello";

}

* In the above example, if a user types http://localhost:8080/hello in the browser, the sayHello() method will be called.

## ✅ Important Points to Remember When Defining Request Paths in Controller Methods

### ✅ 1️⃣ Request Path Must Start with "/"

👉 Example (Correct Way):

@RequestMapping(value = "/report") // Correct: starts with "/"

public String showReport(Map<String, Object> map) {

// Handler logic

return "displayReport";

}

❌ Incorrect Example (Bad Practice):

@RequestMapping(value = "report") // ❌ Wrong! Missing leading slash

public String showReport(Map<String, Object> map) {

// This may not work as expected

}

✅ Always start the request path with /.

### ✅ 2️⃣ Request Path Is Case Sensitive

This means:

* /REPORT and /report are treated as completely different paths by Spring.

#### ✅ Example Controller:

@Controller

public class TestController {

@RequestMapping(value = "/REPORT")

public String showReport(Map<String, Object> map) {

// Handles request when URL ends with "/REPORT"

return "displayReport";

}

@RequestMapping(value = "/report")

public String showReport1(Map<String, Object> map) {

// Handles request when URL ends with "/report"

return "displayReport1";

}

}

#### ✅ Example Usage

| **Request URL** | **Method Called** |
| --- | --- |
| http://localhost:9999/ThirdApp/REPORT | showReport() |
| http://localhost:9999/ThirdApp/report | showReport1() |

👉 Notice that /REPORT and /report are NOT the same because of case sensitivity.

### ✅ Simple Summary Table

| **Rule** | **Why Important** |
| --- | --- |
| Request Path must start with / | It avoids incorrect URL mapping or handler method not being called. |
| Request Path is Case Sensitive | /report and /REPORT are treated differently → They map to different handler methods. |

## ✅ Final Simple Thought

👉 Always define the request path carefully:

* Start with /.
* Be mindful of case sensitivity.

✅ This ensures that the right handler method is called based on the exact URL the user send