**📘 Notes: Handler Methods for Form Operations in Spring MVC**

**General Rule**

Generally, we take **two handler methods** in a controller class when working with a form:

1. **First handler method (GET mode)** → To **launch the form page**.
2. **Second handler method (POST mode)** → To **process the submitted form data**.

**Case 1: Same Request Path (Recommended)**

* Both handler methods share the **same URL path**, but they are separated by **different request modes** (GET and POST).
* This is recommended in industry because it keeps the URL **clean and consistent**.
* In this case, the <form action="..."> attribute becomes **optional** — Spring automatically knows where to submit.

**Step 1: GET Handler → Launch the Form Page**

* This handler prepares the **empty form** (or pre-filled values if required).
* The method should return the JSP page name (register.jsp).
* Example:

// Step 1: Launch the form (GET mode)

@GetMapping("/register")

public String showRegisterForm(Model model) {

// Create an empty Employee object to bind with form fields

model.addAttribute("employee", new Employee());

return "register"; // This will forward to register.jsp

}

➡️ Here, the URL pattern is /register.  
➡️ The **Employee object** is passed into the form to hold input values.

**Step 2: POST Handler → Process the Form Submission**

* This handler collects the **submitted form data**.
* Using @ModelAttribute, the form fields are automatically mapped to the Employee object.
* Example:

// Step 2: Process the form (POST mode)

@PostMapping("/register")

public String processRegisterForm(@ModelAttribute("employee") Employee employee, Model model) {

// Add the submitted employee object into the model

model.addAttribute("empData", employee);

return "result"; // Forward to result.jsp to display submitted data

}

➡️ Same URL pattern /register is used.  
➡️ Spring distinguishes between **GET** and **POST** automatically.  
➡️ After form submission, data goes to **result.jsp**.

**register.jsp (Form Page)**

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>

<html>

<head><title>Register Employee</title></head>

<body>

<h2>Employee Registration</h2>

<form:form modelAttribute="employee" method="post">

Name: <form:input path="name"/><br/>

Age: <form:input path="age"/><br/>

Dept: <form:input path="department"/><br/>

<input type="submit" value="Register"/>

</form:form>

</body>

</html>

**result.jsp (Result Page)**

<html>

<head><title>Result Page</title></head>

<body>

<h2>Employee Submitted Data</h2>

Name: ${empData.name}<br/>

Age: ${empData.age}<br/>

Dept: ${empData.department}<br/>

</body>

</html>

**Case 2: Different Request Paths (Optional)**

* You can also take **different paths** for the GET and POST handler methods.
* Example:
  + GET → /showRegister → to display form.
  + POST → /processRegister → to handle submission.

This is valid, but **less recommended** because it makes the URL design longer and less consistent.

**Extra Notes (Elaboration)**

1. If you take the **same URL**:
   * Cleaner design.
   * <form action> is optional.
   * Spring automatically distinguishes between GET and POST.
2. If you take **different URLs**:
   * <form action="..."> becomes **mandatory**.
   * Slightly more flexible, but not as clean in industry practice.
3. **Form Prefill (Bidirectional Binding)**:
   * If you pass a model object with existing values in Step 1, Spring tags (<form:input>) will automatically display them in the form.
   * Example: showing a default department name, or editing an existing employee.

✅ **Conclusion**:

* In real-time projects, developers **prefer the same URL with GET + POST separation** because it is neat and avoids unnecessary repetition.

**✅ Step 3: Writing <form> without action (when same URL is used for GET and POST)**

**Explanation**

* In real-time projects, we usually **use the same URL path** for both launching the form (GET) and processing the form (POST).
* If the URL path is the same, then we don’t need to explicitly mention the action attribute in the <form> tag.
* Why? Because:
  + When the form page is loaded using the GET handler, Spring already knows which URL was used.
  + When the user presses **Submit**, the form will automatically send the request to the **same URL** but with **POST mode**.
* Hence, action attribute is **optional** in this case.

👉 **If GET and POST have different URLs**

* Then you **must use action** in <form> so that the request goes to the correct POST handler.
* That’s why the **recommended industry practice is to use the same URL** for both GET and POST. It reduces confusion and avoids mistakes.

**Example – JSP (register.jsp)**

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<body>

<h2>Employee Registration</h2>

<!-- No action needed because GET & POST have same URL -->

<form:form method="POST" modelAttribute="emp">

<table>

<tr><td>Employee Number:</td> <td><form:input path="eno"/></td></tr>

<tr><td>Employee Name:</td> <td><form:input path="ename"/></td></tr>

<tr><td>Employee Address:</td> <td><form:input path="eadd"/></td></tr>

<tr><td>Employee Salary:</td> <td><form:input path="salary"/></td></tr>

<tr><td colspan="2"><input type="submit" value="Register"/></td></tr>

</table>

</form:form>

</body>

</html>

**Flow**

1. **User opens** → http://localhost:8080/orcas/emp\_register (GET request).  
   → Controller returns register.jsp.
2. **User fills details** in form.
3. **User presses Submit** → Form automatically goes to **same /emp\_register**.
4. This time request is POST → handled by POST method in controller.

**Notes**

* ✅ Same URL for GET and POST → action optional (preferred in real projects).
* ❌ Different URLs for GET and POST → action mandatory.
* This improves **code readability** and avoids **unnecessary boilerplate in JSP**.

**✅ Step 4: Initial Data in Form Page**

**Explanation**

* When form is launched via **GET handler**, sometimes you want to pre-fill form fields with **default values**.
* These values come from the **Model object** (e.g., Employee).
* If the JSP uses **Spring Form Tags (<form:input>, <form:select>)**, then Spring will automatically display the values set in the object.

👉 **Why is this useful?**

* Saves time for user (e.g., default location, default department).
* Guides user with recommended values.

**Example – Controller (GET handler with initial data)**

@GetMapping("/emp\_register")

public String showForm(Model model) {

Employee emp = new Employee();

emp.setEadd("Hyderabad"); // Pre-filled Address

emp.setSalary(10000.0f); // Pre-filled Salary

model.addAttribute("emp", emp);

return "register"; // JSP page name

}

**Example – JSP (register.jsp)**

<form:form method="POST" modelAttribute="emp">

<table>

<tr><td>Employee Number:</td> <td><form:input path="eno"/></td></tr>

<tr><td>Employee Name:</td> <td><form:input path="ename"/></td></tr>

<tr><td>Employee Address:</td> <td><form:input path="eadd"/></td></tr>

<tr><td>Employee Salary:</td> <td><form:input path="salary"/></td></tr>

<tr><td colspan="2"><input type="submit" value="Register"/></td></tr>

</table>

</form:form>

**What happens here?**

* When page opens:
  + **Employee Address field** already shows → Hyderabad.
  + **Employee Salary field** already shows → 10000.0.
* User can **accept** or **overwrite** these defaults.

**Notes**

* Works **only with Spring form tags**, not with normal HTML <input>.
* Industry best practice: pre-fill only **non-sensitive values** (never passwords or secret info).

**✅ Step 5: Processing Submitted Data**

**Explanation**

* When form is submitted → Spring MVC automatically binds all input fields to the **Employee object** using its **setters**.
* This happens because of @ModelAttribute("emp").
* The POST handler receives this populated object.

👉 Controller can then:

1. **Send data to service layer** → save in database.
2. **Pass data back to JSP** → show confirmation message.

**Example – Controller (POST handler)**

@PostMapping("/emp\_register")

public String processForm(@ModelAttribute("emp") Employee emp, Model model) {

// At this point, emp object contains submitted form data

System.out.println(emp); // Debugging

// Option 1: Save in DB via Service layer

// employeeService.saveEmployee(emp);

// Option 2: Show result page directly

model.addAttribute("empData", emp);

return "result"; // JSP page name

}

**Example – JSP (result.jsp)**

<html>

<body>

<h2>Employee Registration Successful</h2>

<p>Employee No: ${empData.eno}</p>

<p>Employee Name: ${empData.ename}</p>

<p>Employee Address: ${empData.eadd}</p>

<p>Employee Salary: ${empData.salary}</p>

</body>

</html>

**Flow**

1. User submits form → POST request sent to /emp\_register.
2. Spring binds data → Employee emp object is created with user’s input.
3. Controller processes → sends empData back to model.
4. JSP result.jsp displays success message with all details.

**Notes**

* @ModelAttribute does automatic binding — **no need to manually call setters**.
* If form field names don’t match object properties → binding will fail (industry devs always ensure naming consistency).
* This step shows how **UI → Controller → Service → DB** cycle is completed.

⚡ **Final Recap**

* **Step 1:** Launch Form → GET handler → JSP.
* **Step 2:** Fill Form → Bound to model.
* **Step 3:** Submit → No action needed if same URL (recommended).
* **Step 4:** Default values can be pre-filled from Model.
* **Step 5:** POST handler processes → DB save or JSP result.

## ✅ HTML Forms (Default Behavior)

* When you design a form using **pure HTML <form> + <input>**, Spring MVC will only support **One-Way Binding**:
  + **Form → Model Object** (data flows from UI to Controller).
* That means:
  + You can **submit data** from the form into your Model object (via setters).
  + But you **cannot pre-fill** form fields directly from the Model object.

👉 Example – HTML Form (register.html)

<html>

<body>

<h2>Employee Registration</h2>

<!-- This is plain HTML form -->

<form action="/emp\_register" method="POST">

<table>

<tr><td>Employee Number:</td> <td><input type="text" name="eno"/></td></tr>

<tr><td>Employee Name:</td> <td><input type="text" name="ename"/></td></tr>

<tr><td>Employee Address:</td> <td><input type="text" name="eadd"/></td></tr>

<tr><td>Employee Salary:</td> <td><input type="text" name="salary"/></td></tr>

<tr><td colspan="2"><input type="submit" value="Register"/></td></tr>

</table>

</form>

</body>

</html>

👉 What happens here?

* On submit → Data goes **Form → Controller → Model object**.
* But if Controller sets default values like Hyderabad or 10000.0 → They will **not appear in the HTML form automatically**.
* You’d need to manually write ${emp.eadd} in value="" attribute, which is extra work.

📌 So plain HTML forms = **Unidirectional (One-Way) Binding only**.

**✅ Spring MVC Form Tag Library (Two-Way Binding)**

Spring provides **JSP tag libraries** that make binding easier:

1. **Generic Tag Library**

<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>

* Provides general Spring tags (like spring:message for i18n).

1. **Form Tag Library**

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

* Provides tags like <form:form>, <form:input>, <form:select>, etc.
* Enables **Two-Way Binding**:
  + **Form → Model object** (when user submits).
  + **Model object → Form** (when form loads with default values).

**Example – Spring Form Tag (register.jsp)**

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<html>

<body>

<h2>Employee Registration</h2>

<!-- Two-Way Binding is enabled here -->

<form:form method="POST" modelAttribute="emp">

<table>

<tr><td>Employee Number:</td> <td><form:input path="eno"/></td></tr>

<tr><td>Employee Name:</td> <td><form:input path="ename"/></td></tr>

<tr><td>Employee Address:</td> <td><form:input path="eadd"/></td></tr>

<tr><td>Employee Salary:</td> <td><form:input path="salary"/></td></tr>

<tr><td colspan="2"><input type="submit" value="Register"/></td></tr>

</table>

</form:form>

</body>

</html>

**Example – Controller (Pre-fill data)**

@GetMapping("/emp\_register")

public String showForm(Model model) {

Employee emp = new Employee();

emp.setEadd("Hyderabad"); // Default value

emp.setSalary(10000.0f); // Default value

model.addAttribute("emp", emp);

return "register"; // JSP

}

**Flow with Spring Form Tags**

1. User opens form (GET request).
   * Spring puts emp.eadd = Hyderabad and emp.salary = 10000.0 **into form fields automatically**.
2. User changes or keeps values and submits.
3. Data comes back **into Employee object** (POST handler).

👉 This is **Bidirectional Binding**:

* Controller → Form (default values displayed).
* Form → Controller (submitted values captured).

**🔹 @RequestParam — quick idea (simple)**

@RequestParam maps a single HTTP request parameter (from query string or form data) to a controller method parameter.

Example URL:

http://localhost:9999/DataBindingApp/data?sno=10&sname=Ravi

Here sno=10 and sname=Ravi are request parameters. @RequestParam reads them and gives them to your method.

**✅ Basic usage (recommended pattern)**

package in.orcas.controller;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.ui.Model;

@Controller

public class DataBindingController {

@GetMapping("/data")

public String bindData(Model model,

@RequestParam("sno") Integer sno,

@RequestParam("sname") String sname) {

model.addAttribute("sno", sno);

model.addAttribute("sname", sname);

return "show\_data";

}

}

* Spring will take sno and sname from the request and convert types automatically.
* If values are present and convertible → method receives them.

**🚩 Your three cases explained (step-by-step)**

**Case 1: http://.../data?sno=10 with**

@GetMapping("/data")

public String bindData(Model model,

@RequestParam Integer sno,

@RequestParam String sname) { ... }

**What happens**

* sno=10 is present → bound to sno.
* sname is **missing** -> but @RequestParam by default is required=true.
* Spring throws MissingServletRequestParameterException (a 400/405 type error in console).

**Result**

* You get an exception and no normal page rendering.
* **Reason:** required request param sname is not provided.

**Fixes:** either make sname optional or provide a default value (see below).

**Case 2: http://.../data?sno=10 but we want sname optional**

Two common ways to allow missing sname:

**A. required=false**

@GetMapping("/data")

public String bindData(Model model,

@RequestParam Integer sno,

@RequestParam(required = false) String sname) {

// if sname missing -> sname == null

model.addAttribute("sname", sname);

return "show\_data";

}

* If sname is missing, sname becomes null. Controller can handle null safely.

**B. Use Optional<T>** (Java 8+)

@GetMapping("/data")

public String bindData(Model model,

@RequestParam Integer sno,

@RequestParam Optional<String> snameOpt) {

String sname = snameOpt.orElse(null);

model.addAttribute("sname", sname);

return "show\_data";

}

* snameOpt.isPresent() tells you whether it came or not.

**Case 3: want a fallback value when sname missing**

@GetMapping("/data")

public String bindData(Model model,

@RequestParam Integer sno,

@RequestParam(defaultValue = "sachin") String sname) {

// if sname missing (or empty) -> sname == "sachin"

model.addAttribute("sname", sname);

return "show\_data";

}

* **Behavior**: defaultValue is used **when the parameter is not present or is empty**.
* defaultValue implicitly sets required=false for you.

**Output for** /data?sno=10 → sno = 10, sname = "sachin".

**⚠️ Common pitfalls and rules (simple)**

1. **Default required=true**
   * @RequestParam without required=false means the param must be present. If it’s missing, Spring throws an exception.
2. **Primitive vs wrapper types**
   * Use wrapper types (Integer, Long, Float) for optional params.
   * If you use primitive int and the param is missing, you can run into errors — prefer Integer or supply defaultValue.
3. **Type conversion errors**
   * If request has sno=abc but your method expects Integer, Spring throws a TypeMismatch error. Validate/convert before using, or catch/handle binding exceptions.
4. **Empty value handling**
   * ?sname= (present but empty) is treated as empty string. If you used defaultValue, Spring will apply it for empty values too.
5. **Name mismatch**
   * If your parameter name differs from query name, use @RequestParam("qname") String myParam.
6. **Binding repeated params**
   * For ?id=1&id=2&id=3 you can accept @RequestParam List<Integer> id or @RequestParam Integer[] id.
7. **Read all params**
   * @RequestParam Map<String,String> params — gets all query/form params as a map. Useful for dynamic handling.

**🔸 More examples (realistic)**

**A. Optional string + default**

@GetMapping("/data")

public String bindData(Model model,

@RequestParam("sno") Integer sno,

@RequestParam(name="sname", required=false) String sname) {

if (sname == null) sname = "guest"; // application fallback

model.addAttribute("sno", sno);

model.addAttribute("sname", sname);

return "show\_data";

}

**B. Using Map to capture everything**

@GetMapping("/data")

public String bindData(@RequestParam Map<String,String> allParams, Model model) {

// allParams contains keys and values from query string/form

model.addAttribute("params", allParams);

return "show\_data";

}

**C. Multiple values**

URL: /data?tag=java&tag=spring

@GetMapping("/data")

public String bindData(@RequestParam List<String> tag, Model model) {

// tag = ["java", "spring"]

model.addAttribute("tags", tag);

return "show\_data";

}

**🔑 Practical recommendations (industry style)**

* Use @RequestParam for small/simple inputs: ids, flags, page, sort, filters.
* For many fields (forms with many inputs), prefer binding to a model object with @ModelAttribute.
* Use wrapper types (Integer, Long) if param can be absent.
* Prefer defaultValue when you want a safe fallback and want to avoid null checks.
* Use required=false or Optional<T> when presence of param is optional and null is acceptable.
* Always validate & sanitize the incoming values (especially when using them for DB operations).

**🧾 Summary table (very short)**

| **Use case** | **Annotation** | **Result if missing** |
| --- | --- | --- |
| Required param | @RequestParam String p | Exception if missing |
| Optional param | @RequestParam(required=false) String p | p == null |
| Optional with fallback | @RequestParam(defaultValue="x") String p | p == "x" when missing/empty |
| Multiple values | @RequestParam List<Integer> ids | Accepts repeated params |
| Capture all | @RequestParam Map<String,String> all | Full params map |