## **Form Data Binding with** @ModelAttribute

### **1. What is a Model Class (Java Bean)?**

* A **Model class** is just a normal Java class (also called JavaBean).
* It is used to **hold form data** entered by the user in the UI page (like register.jsp).
* To make it work for form binding:
  1. Count the number of form fields → create the same number of properties in the Model class.
  2. The **form field names** and the **property names** in the Model class must **match exactly**.
  3. Add **getter and setter methods** for every property.
  4. In the controller’s handler method, use @ModelAttribute with the Model class type.

### **2. Example: Employee Registration**

#### **register.jsp**

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

<table align="center">

<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>

## **3. Model Class (Employee.java)**

package in.orcas.model;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data // generates getters, setters, toString, equals, hashCode

@NoArgsConstructor // generates a 0-parameter constructor

@AllArgsConstructor // generates a parameterized constructor

public class Employee {

private int eno;

private String ename;

private String eadd = "hyd"; // default value

private float salary;

}

👉 **Explanation**:

* @Data → gives you all getters, setters, toString(), equals(), hashCode().
* @NoArgsConstructor → generates 0-argument constructor (needed for Spring binding).
* @AllArgsConstructor → generates full constructor (handy if you want to manually create objects later).

## 4. **Controller Class (EmployeeController.java)**

package in.orcas.controller;

import in.orcas.model.Employee;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

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

@Controller

public class EmployeeController {

@PostMapping("/emp\_register")

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

// emp object automatically holds form data

System.out.println("Employee object received: " + emp);

// Add employee object to Model to forward it to the result page

model.addAttribute("employee", emp);

return "result"; // forwards to result.jsp

}

}

👉 **Explanation**:

* The handler method accepts:
  + @ModelAttribute("emp") Employee emp → for automatic form binding.
  + Model model → industry-standard parameter for sending data to the view.
* After binding, the filled Employee object is added to the Model so the JSP page can use it.

**Internal Operations of Form Binding (@ModelAttribute)**

**📝 Internal Operations of Form Binding in Spring MVC**

**1. End User Action**

* User opens browser → hits URL:
* http://localhost:8080/register
* Browser sends a **GET request** to the server.

**2. DispatcherServlet (for GET request)**

* DispatcherServlet receives the request.
* It checks the handler mappings and finds:
* @GetMapping("/register")
* public String showRegisterForm(Model model)
* It creates a new **Employee** object (new Employee()), adds it to the model with name employee.
* DispatcherServlet forwards the request to **register.jsp**.

**3. Form Page (register.jsp)**

* In register.jsp, we have:
* <form:form method="POST" modelAttribute="employee" action="emp\_register">
* The Spring <form:form> tag binds the input fields (eno, ename, eadd, salary) with the properties of the Employee model object.
* User fills data in text boxes and clicks **Submit**.
* Browser sends a **POST request** to /emp\_register with all form data as request parameters.

**4. DispatcherServlet (for POST request)**

* DispatcherServlet traps this POST request.
* It looks up the handler mapping and finds:
* @PostMapping("/emp\_register")
* public String processRegisterForm(@ModelAttribute("employee") Employee emp, Model model)
* Now, it sees that the handler method has an argument:
* @ModelAttribute("employee") Employee emp

→ This signals that **Form Binding** (a.k.a. **Data Binding**) must be performed.

**5. Creating & Binding the Model Object**

1. DispatcherServlet creates a new **Employee** object (if not already in model).
2. Employee emp = new Employee();
3. It calls request.getParameter("eno"), request.getParameter("ename"), etc. to read submitted values.
4. It converts those String values into the correct property types (like int, float) using **PropertyEditors** (or in modern Spring, ConversionService).
5. It calls the **setter methods** on the Employee object:
6. emp.setEno(...);
7. emp.setEname(...);
8. emp.setEadd(...);
9. emp.setSalary(...);

→ Now the emp object contains all the form data.

**6. Preparing to Call the Handler Method**

* DispatcherServlet creates the other required objects (like BindingAwareModelMap).
* It calls the controller handler method:
* processRegisterForm(emp, model);
* Inside the method, you can use emp (with populated data) or send it to a service class.
* You can also add it to the model for the view.

**7. Rendering the Result Page**

* Handler method returns "result".
* DispatcherServlet uses InternalResourceViewResolver to resolve it as:
* /WEB-INF/views/result.jsp
* It forwards the request to result.jsp, along with the model (containing emp).
* In JSP, you can display data using ${emp.eno}, ${emp.ename}, etc.

**🔑 Key Points**

* **GET request** → only used to land on the form page.
* **POST request** → used to bind data from form to model object.
* DispatcherServlet does all heavy lifting: creating object, reading request parameters, converting types, and calling setters.
* Binding happens automatically as long as **form field names** and **model class property names** match.

⚡ Now, your flow is clean:

* /register (GET) → show form.
* /emp\_register (POST) → bind data → show result.

Eg: DataBindingWithModelAttribute

**📝 Bi-Directional Binding (Two-Way Binding) with Example**

**🔹 Step 1: User Opens the Form (GET Request)**

👉 **Direction: Controller ➝ JSP (Object → Form fields)**

**Controller method:**

@GetMapping("/register")

public String showForm(Model model) {

// 1. Create Employee object with some default data

Employee emp = new Employee();

emp.setEname("Sachin");

emp.setSalary(50000f);

// 2. Add object to Model

model.addAttribute("employee", emp);

// 3. Send to register.jsp

return "register";

}

**register.jsp:**

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

<html>

<body>

<h2>Employee Registration Form</h2>

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

Employee No: <form:input path="eno"/> <br>

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

Salary: <form:input path="salary"/> <br>

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

</form:form>

</body>

</html>

**What happens here:**

1. Controller places employee object in model.
2. DispatcherServlet forwards to register.jsp.
3. Spring Tag Library (form:input) calls **getter methods**:
   * getEname() → returns "Sachin" → textbox is pre-filled.
   * getSalary() → returns 50000 → textbox is pre-filled.

👉 **Direction 1 (Controller → JSP):**  
Java Object → JSP form fields (via getters).

**🔹 Step 2: User Submits the Form (POST Request)**

👉 **Direction: JSP ➝ Controller (Form fields → Object)**

**Controller method:**

@PostMapping("/save")

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

// Employee object is automatically filled with submitted values

model.addAttribute("emp", emp);

return "result";

}

**result.jsp:**

<html>

<body>

<h2>Employee Registered Successfully</h2>

Employee No: ${emp.eno} <br>

Name: ${emp.ename} <br>

Salary: ${emp.salary} <br>

</body>

</html>

**What happens here:**

1. User types new values in register.jsp form:
2. eno=101
3. ename=Rahul
4. salary=60000
5. Browser sends these values as request parameters.
6. DispatcherServlet creates a new empty Employee object.
7. Calls setters:
   * setEno(101)
   * setEname("Rahul")
   * setSalary(60000)
8. Filled object is given to controller method as parameter emp.
9. Controller adds it to model → result.jsp displays the values.

👉 **Direction 2 (JSP → Controller):**  
Form fields → Java Object (via setters).

**🔑 Bi-Directional Binding = Two Directions Together**

1. **First Direction (GET):** Controller → JSP (object data shown in form using getters).
2. **Second Direction (POST):** JSP → Controller (form data stored back into object using setters).

This cycle is **two-way**, so it’s called **Bi-Directional Binding**.  
It is very useful for **edit/update forms**, because:

* Old values are shown automatically (GET).
* Updated values are sent back automatically (POST).

**🔹 Unidirectional Binding in Spring MVC**

👉 **Definition**:  
Unidirectional Binding means data flows **only in one direction**:  
**From the View (JSP form) → Controller (Model object)**.  
The controller only receives the submitted values, but it doesn’t send anything back to the JSP form.

**📝 Example**

**1. register.jsp (View → form page)**

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

<table align="center">

<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>

**2. Employee.java (Model class)**

package in.orcas.model;

import lombok.Data;

@Data

public class Employee {

private int eno;

private String ename;

private String eadd;

private float salary;

}

**3. EmployeeController.java (Controller)**

package in.orcas.controller;

import org.springframework.stereotype.Controller;

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

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

import in.orcas.model.Employee;

@Controller

public class EmployeeController {

// Only handles POST data (unidirectional)

@PostMapping("/emp\_register")

public String registerEmployee(@ModelAttribute("emp") Employee emp) {

// Here emp object already has form data filled

System.out.println("Form Data Received: " + emp);

return "result"; // forwards to result.jsp

}

}

**4. result.jsp**

<h2>Employee Registered Successfully!</h2>

<p>Thank you for submitting your details.</p>

**🔄 Flow of Unidirectional Binding**

1. User opens register.jsp.
2. Fills in employee details.
3. Clicks **Register** → Browser sends a **POST request** to /emp\_register.
4. **DispatcherServlet** catches the request.
5. It sees @ModelAttribute("emp") Employee emp in the method signature.
6. Creates a new Employee object (emp).
7. Reads request parameters (eno, ename, eadd, salary) → fills into emp using **setters**.
8. Calls registerEmployee() with the ready emp object.
9. Controller prints or processes the data.
10. Returns "result" → DispatcherServlet forwards to /WEB-INF/views/result.jsp.
11. Result page is shown.

**🚦 Key Point (Direction)**

* **Data goes only one way:**  
  👉 register.jsp → EmployeeController  
  ❌ Controller does not send data back to register.jsp.
* That’s why it is **Unidirectional Binding**.

✅ **Comparison with Bidirectional Binding**

* **Unidirectional:** form data → controller only (like registration form submission).
* **Bidirectional:** controller can also pre-fill form fields and send data → view (like editing an employee where old values are shown in form).