**✅ Data Binding and Data Rendering in Spring MVC**

**🎯 Data Binding**

**Definition:**  
Data Binding is the process where input values from a view component (such as form input fields or request parameters) are automatically mapped and written into a Java class object (called a Model class object) when a request is sent to the controller.

* Data Binding means converting the **form input data** (sent by the user in an HTTP request) into a **Java object** automatically.
* Instead of manually extracting each parameter using request.getParameter(), Spring does this automatically for us.
* This helps in writing clean and simple code in the Controller.

**✅ 2️ Why Use @ModelAttribute in the Controller?**

@PostMapping("/register")

public String registerUser(@ModelAttribute User user, Model model) { ... }

* @ModelAttribute User user tells Spring:

👉 "Please take all form data fields from the request, and automatically fill the matching properties of the User object."

* Example:
  + Form inputs: name=Pavan, email=pavan@example.com, age=30
  + Spring creates a User object like:
  + User user = new User();
  + user.setName("Pavan");
  + user.setEmail("pavan@example.com");
  + user.setAge(30);
* ✅ This makes our code very simple and clean.

**✅ 3️ Step-by-Step Flow**

| **Step** | **What Happens** |
| --- | --- |
| ✅ 1 | User visits /register URL. |
| ✅ 2 | Controller's showForm() method is called, which returns the logical view name "register". |
| ✅ 3 | Spring shows register.jsp form to the user. |
| ✅ 4 | User fills the form and clicks "Register". |
| ✅ 5 | Form sends a POST request to /data with form fields as request parameters. |
| ✅ 6 | Spring detects @ModelAttribute User user, so it automatically creates a User object and sets its fields with the form data. |
| ✅ 7 | Controller now has a fully populated User object. We can use user.getName(), user.getEmail(), etc. |
| ✅ 8 | Controller passes the User object back to the view (success.jsp) using model.addAttribute("user", user). |
| ✅ 9 | In success.jsp, data is displayed using ${user.name}, ${user.email}, ${user.age}. |
| ✅ 10 | The user sees a confirmation page showing their registered data. |

**✅ 4️ Why Is Model Important?**

* The Model object acts like a **shared memory area** between the Controller and the View.
* Data placed in model.addAttribute() is available in the JSP view.

Example:

model.addAttribute("user", user);

* Now, in JSP, we can access user’s fields like:

Name: ${user.name}

**✅ 5️ Why Use Lombok?**

* Without Lombok, we would need to write getters, setters, and a toString() method manually in the User class.
* Lombok’s @Data does this automatically.

Before Lombok:

public class User {

private String name;

private String email;

private int age;

public String getName() { return name; }

public void setName(String name) { this.name = name; }

// ... similar for email and age

}

With Lombok:

@Data

public class User {

private String name;

private String email;

private int age;

}

✅ This makes code shorter and easier to read.

**✅ 6️ Simple Summary of the Example**

* **Data Binding** automatically fills a Java object from form fields.
* @ModelAttribute tells Spring to create and populate the object.
* The **Model object** holds data that the view needs.
* The **View (JSP)** shows the final result.
* **Lombok** avoids writing repetitive code like getters/setters.

Eg: DataBinding

**✅ Data Rendering**

**Definition:**  
Data Rendering is the process where data generated or retrieved by the controller (after executing business logic) is passed to the view component. This is done through the shared memory object called BindingAwareModelMap, typically accessed via the Model or ModelMap parameter.

**👉 Example Scenario:**

* Controller retrieves a list of products from the database.
* Controller puts this data into the Model.
* Data is rendered and displayed in the view (e.g., JSP).

@Controller

public class ProductController {

@GetMapping("/products")

public String showProducts(Model model) {

List<String> productList = List.of("Laptop", "Phone", "Tablet");

model.addAttribute("products", productList); // Data Rendering

return "productListPage";

}

}

In the view (JSP):

<ul>

<c:forEach var="product" items="${products}">

<li>${product}</li>

</c:forEach>

</ul>

**✅ Types of Data Passed in Data Rendering**

| **No.** | **Data Type** | **Example** |
| --- | --- | --- |
| a. | Simple Values | model.addAttribute("username", "Pavan"); |
| b. | Array/Collections | model.addAttribute("colors", new String[]{"Red", "Green", "Blue"}); |
| c. | Collection of Model/Business Objects | model.addAttribute("userList", List<User>); |
| d. | Single Model/Business Object | model.addAttribute("user", new User(...)); |

**✅ Important Note**

* **Model (Interface)** → Comes from **Spring Framework**.
* **Model (Business Object)** → Refers to your **Entity Class / Business Object (BO)**.

**✅ Simple Summary**

* **Data Binding** → Converts incoming form/request data → Java Model Object (for easy manipulation in the controller).
* **Data Rendering** → Sends data from Controller → View Component (JSP) via Model or BindingAwareModelMap.

Eg: DataRendering

### Program Flow: How the Application Works

**Purpose**: The DataRendering application is a web app built with Spring Boot. It starts a server, handles HTTP requests, and shows a webpage (/products) with a table listing products (ID, Name, Price). The data is hardcoded in the controller and rendered using a JSP file.

#### Step 1: Starting the Application

* **File**: DataRenderingApplication.java
* **What Happens**:
  + This is the main entry point of the program, located in the in.orcas package.
  + The @SpringBootApplication annotation tells Spring Boot to set up the app, including:
    - Scanning for components (like controllers) in in.orcas and its subpackages (in.orcas.model, in.orcas.controller).
    - Configuring an embedded web server (Tomcat, thanks to spring-boot-starter-web).
    - Loading settings from application.properties.
  + The main method runs SpringApplication.run(DataRenderingApplication.class, args), which:
    - Starts the Spring Boot application.
    - Initializes the Spring context (a container managing all components).
    - Launches the Tomcat server on http://localhost:8080 (default port).
* **Flow Detail**:
  + Spring Boot scans for @Controller classes (like ProductController) to handle web requests.
  + It reads application.properties to configure the view resolver for JSP files.
  + It uses dependencies from pom.xml to ensure JSP and JSTL work.
* **Status**: This file is correct—no mistakes here. It sets up the foundation perfectly.

#### Step 2: Setting Up Dependencies (Maven Build)

* **File**: pom.xml
* **What Happens**:
  + This file tells Maven (the build tool) what libraries your project needs and how to build it.
  + Key dependencies include:
    - spring-boot-starter-web: Provides Spring MVC for web requests and Tomcat for the server.
    - tomcat-embed-jasper: Enables JSP support in the embedded Tomcat server.
    - jakarta.servlet.jsp.jstl-api and org.glassfish.web:jakarta.servlet.jsp.jstl: Provide JSTL for dynamic JSP rendering.
    - lombok: Reduces boilerplate code in Product.java (e.g., auto-generates getters/setters).
    - spring-boot-devtools (optional): Auto-restarts the app during development.
    - spring-boot-starter-test: For testing (not used in runtime flow).
  + The <parent> tag (spring-boot-starter-parent:3.5.5) ensures all dependencies are compatible with Spring Boot 3.x and Java 17.
  + The spring-boot-maven-plugin packages the app into a runnable JAR.
* **Flow Detail**:
  + Maven downloads these libraries from Maven Central when you run mvn clean install or mvn spring-boot:run.
  + These libraries are added to the classpath, enabling JSP rendering, JSTL tags, and Spring MVC features.
* **Mistake You Made**:
  + **Invalid JSTL Dependency**: You initially used:

xml

<dependency>

<groupId>org.glassfish.jstl</groupId>

<artifactId>jakarta.servlet.jsp.jstl</artifactId>

<version>1.2.7</version>

</dependency>

* + - **Why Wrong**: The groupId org.glassfish.jstl doesn’t exist, and version 1.2.7 is for the old javax namespace (used in Spring Boot 2.x). Spring Boot 3.x uses Jakarta EE 9+, requiring org.glassfish.web:jakarta.servlet.jsp.jstl and the API dependency.
    - **Impact**: Maven failed to download the library, causing build errors like "Could not find artifact" and "Missing artifact," stopping the project from compiling or running.
  + **Fix**:
    - Replaced with:

xml

<dependency>

<groupId>jakarta.servlet.jsp.jstl</groupId>

<artifactId>jakarta.servlet.jsp.jstl-api</artifactId>

</dependency>

<dependency>

<groupId>org.glassfish.web</groupId>

<artifactId>jakarta.servlet.jsp.jstl</artifactId>

</dependency>

* + - Omitted versions (Spring Boot’s parent manages them for compatibility).
    - Ran mvn clean install to download the correct JARs and clear the broken cache (~/.m2/repository/org/glassfish/jstl/).
* **Lesson**: Always verify dependency coordinates on Maven Central (search.mvnrepository.com) and ensure they match your Spring Boot version’s Jakarta EE requirements.

#### Step 3: Defining the Product Model

* **File**: Product.java
* **What Happens**:
  + Located in in.orcas.model, this class defines a Product with fields: id (integer), name (string), price (double).
  + Lombok annotations (@Data, @NoArgsConstructor, @AllArgsConstructor) generate:
    - Getters/setters (e.g., getId(), setName()).
    - A no-arg constructor (new Product()).
    - A constructor with all fields (new Product(1, "Laptop", 75000)).
    - toString(), equals(), and hashCode() methods.
  + This class represents the data structure for products displayed on the webpage.
* **Flow Detail**:
  + The controller creates Product objects and passes them to the JSP.
  + JSP accesses fields via getters (e.g., ${product.id} calls getId()).
* **Status**: No mistakes here—clean and correct. Lombok works as expected with pom.xml’s provided scope.

#### Step 4: Handling HTTP Requests (Controller Logic)

* **File**: ProductController.java
* **What Happens**:
  + Located in in.orcas.controller, this class is marked @Controller, telling Spring it handles web requests.
  + The @GetMapping("/products") method showProducts responds to GET requests at http://localhost:8080/products.
  + Inside showProducts:
    - Creates an ArrayList<Product> and adds three hardcoded products:
      * Product(1, "Laptop", 75000)
      * Product(2, "Phone", 15000)
      * Product(3, "Tablet", 25000)
    - Adds the list to the Model with key "products" (accessible in JSP as ${products}).
    - Returns "productList", instructing Spring to render productList.jsp.
* **Flow Detail**:
  + When a user visits /products, Spring calls showProducts.
  + The Model passes the product list to the JSP view.
  + Spring uses application.properties to resolve "productList" to a JSP file.
* **Status**: No mistakes—logic is perfect and aligns with the model and JSP.

#### Step 5: Configuring View Resolution

* **File**: application.properties
* **What Happens**:
  + Located in src/main/resources, this file configures the app:
    - spring.application.name=DataRendering: Names the app (used in logs, not critical for flow).
    - spring.mvc.view.prefix=/WEB-INF/jsp/: Tells Spring to look for JSP files in src/main/webapp/WEB-INF/jsp/.
    - spring.mvc.view.suffix=.jsp: Appends .jsp to view names (e.g., "productList" becomes productList.jsp).
  + Spring’s view resolver combines these to locate src/main/webapp/WEB-INF/jsp/productList.jsp.
* **Flow Detail**:
  + When ProductController returns "productList", Spring resolves it to /WEB-INF/jsp/productList.jsp.
  + The JSP is processed by Tomcat’s JSP engine (enabled by tomcat-embed-jasper).
* **Mistake You Made**:
  + **Potential File Placement Issue**: If productList.jsp was not in src/main/webapp/WEB-INF/jsp/ (e.g., in WEB-INF/views/), Spring wouldn’t find it, causing a 404 or Whitelabel Error Page.
    - **Why Wrong**: The view resolver’s path must exactly match the JSP’s location.
    - **Impact**: The webpage wouldn’t render, even if everything else worked.
  + **Fix**:
    - Ensure productList.jsp is in src/main/webapp/WEB-INF/jsp/.
    - Alternatively, update spring.mvc.view.prefix to match the actual folder (e.g., /WEB-INF/views/ if that’s where the JSP is).
* **Lesson**: Verify JSP file paths against application.properties. Test early by accessing the endpoint to catch 404s.

#### Step 6: Rendering the Webpage

* **File**: productList.jsp
* **What Happens**:
  + Located in src/main/webapp/WEB-INF/jsp/productList.jsp, this JSP generates the HTML webpage.
  + Key components:
    - <!DOCTYPE html>: Declares an HTML5 page.
    - <table>: Creates a table with headers (ID, Name, Price).
    - <c:forEach var="product" items="${products}">: Loops over the products list from the controller.
    - ${product.id}, ${product.name}, ${product.price}: Displays each product’s fields using getters.
  + The JSP is processed by Tomcat, which:
    - Uses JSTL (jakarta.tags.core) to handle the <c:forEach> loop.
    - Converts the JSP into a servlet, merging dynamic data (${products}) with static HTML.
    - Sends the final HTML to the browser.
* **Flow Detail**:
  + User visits http://localhost:8080/products.
  + Controller passes List<Product> to the JSP via the Model.
  + JSP generates HTML, showing a table with three products:

text

Available Products

ID Name Price

1 Laptop 75000

2 Phone 15000

3 Tablet 25000

* **Mistake You Made**:
  + **Incorrect Taglib URI**: You used:

jsp

<%@ taglib prefix="c" uri="jakarta.servlet.jsp.jstl.core" %>

* + - **Why Wrong**: The URI jakarta.servlet.jsp.jstl.core is invalid. The correct URI for JSTL core tags in Jakarta EE 9+ (Spring Boot 3.x) is jakarta.tags.core.
    - **Impact**: Caused a runtime JasperException (e.g., "The absolute uri cannot be resolved"), preventing the JSP from rendering. This likely felt like a syntax error (e.g., the "comma" you mentioned) if the IDE/logs were vague.
  + **Fix**:
    - Changed to:

jsp

<%@ taglib prefix="c" uri="jakarta.tags.core" %>

* + - Ensured pom.xml has the correct JSTL dependencies to support this URI.
* **Lesson**: Use the exact taglib URI from the Jakarta EE spec (jakarta.tags.core for core tags). Test JSP rendering early to catch runtime errors.

#### Step 7: Browser Display

* **What Happens**:
  + The browser receives the HTML from productList.jsp.
  + It displays a webpage with a heading ("Available Products") and a table listing the three products.
  + The user sees the data rendered dynamically from the controller’s hardcoded list.
* **Flow Detail**:
  + The entire flow—starting the server, handling the request, resolving the JSP, and rendering HTML—takes milliseconds.
  + Any errors (e.g., 404, 500) would appear here if earlier steps failed.

**tomcat-embed-jasper** → Compiles and runs JSP files in embedded Tomcat. Handles basic JSP syntax like ${} and scriptlets.

 **jakarta.servlet.jsp.jstl-api** → Think of it as the **interface/contract**. It defines *what* JSTL tags like <c:forEach> should do, but doesn’t actually do it. Used at **compile-time**.

 **org.glassfish.web:jakarta.servlet.jsp.jstl** → This is the **implementation**. It contains the actual code that executes the JSTL tags at **runtime**.

**Why both are needed** → API alone allows compilation, GlassFish provides execution, and Jasper handles JSP compilation. All three together make dynamic JSP pages with JSTL work correctly.

**Data Rendering in Spring Boot MVC**

* **Meaning**: Data rendering is about **sending data from the controller to the view (like JSP, Thymeleaf, etc.)**.
* **How it works**:
  1. A request comes to the **DispatcherServlet**.
  2. The controller’s handler method prepares the data.
  3. This data is stored in a **shared memory object** called BindingAwareModelMap.
  4. The view (UI page) uses this data to display results to the user.
* **In short**:
  1. Controller → View = **Data Rendering**

**Data Binding in Spring Boot MVC**

* **Meaning**: Data binding is about **taking values entered by the user in the view (form or hyperlink params) and giving them to the controller’s handler methods**.
* **In short**:
  + View → Controller = **Data Binding**

**Difference Between the Two**

* **Data Rendering** = Controller → View
* **Data Binding** = View → Controller

**Ways of Data Binding**

1. **Form Data Binding (a.k.a. Request Wrapping / Form Binding)**
   * When a user submits a form, the values can be directly bound to a Java object (Model/Command object).
   * Annotation used: @ModelAttribute.
   * Example: If a form has fields like name, email, and age, Spring automatically sets these values into the matching fields of a Java object (say User).
   * **Self-explanatory point**: This avoids manually reading request parameters one by one; Spring “wraps” the form data into an object for us.
2. **Request Parameter Binding (a.k.a. RequestParam Binding)**
   * When extra values are passed as request parameters (like ?id=101&status=active in a hyperlink), we can bind them directly to method parameters in the controller.
   * Annotation used: @RequestParam.
   * Example:
   * public String getEmployee(@RequestParam("id") int empId) { ... }

Here, the id parameter from the request URL is automatically given to the empId variable.

* + **Self-explanatory point**: This is useful for simple values that don’t need a whole object, like IDs, flags, or search filters.

✅ **Final Recap**:

 **Data Rendering**: Controller sends data to View → uses BindingAwareModelMap.

 **Data Binding**: View sends data to Controller → done via @ModelAttribute or @RequestParam.