**✅ 1. With View (Using Presentation File like JSP)**

This is a very common way used in web applications with UI (like JSP files). Here's how it works step-by-step:

**🔁 When a client sends a request:**

1. The request URL (like /hello) comes to the Spring Boot application.
2. The @Controller class has a method (like display()).
3. This method processes the request.
4. It puts some **data into the Model object**.
5. It returns the name of the JSP file (like "index").
6. The data is then displayed in the JSP using ${key} format.

**📌 Model Object**

* The **Model** is like a container (or bag) to **carry data** from the controller to the view.
* Data is added using model.addAttribute("key", "value")
* The JSP file can then **access this key** using ${key} and show the value on screen.

**✅ Example Code**

**👨‍💻 Controller**

@Controller

public class HelloController {

@GetMapping("/hello")

public String display(Model model) {

model.addAttribute("message", "Welcome to Spring MVC!");

return "index"; // points to /WEB-INF/views/index.jsp

}

}

**🧾 JSP File (index.jsp)**

jsp

CopyEdit

<body>

<h1>${message}</h1> <!-- will print "Welcome to Spring MVC!" -->

</body>

**✅ Summary (With View):**

| **Step** | **What Happens** |
| --- | --- |
| 1 | Browser sends request to URL |
| 2 | Spring matches it to a method in controller |
| 3 | Controller method puts data into Model |
| 4 | Returns the name of the JSP page (view) |
| 5 | JSP uses ${key} to show the data on UI |

**✅ 2. Raw Response (Just an Outline for Now)**

**What is Raw Response?**

* It means sending back **pure data** instead of showing a UI.
* This is common in **APIs**, where data is returned in **JSON format**.

**Example:**

**👨‍💻 Controller with Raw Response**

@RestController

public class HelloRestController {

@GetMapping("/greet")

public String greet() {

return "Hello from Spring Boot!"; // plain text (raw)

}

}

This will directly return:

Hello from Spring Boot!

No JSP. No View. Just text or data sent to browser/postman.

**When will you learn this?**

* When your sir starts **RESTful web services**, you'll learn:
  + @RestController
  + JSON
  + @ResponseBody
  + APIs for frontend/backend

**🔚 Final Thoughts**

| **Method** | **Type** | **Uses JSP View?** | **Uses Model?** | **Best For** |
| --- | --- | --- | --- | --- |
| With View | Traditional | ✅ Yes | ✅ Yes | Web UI |
| Raw Response | REST API | ❌ No | ❌ No (or limited) | Mobile apps, APIs, frontend JS frameworks |

**📝 Notes: Why Use Model in Spring MVC Controller Instead of Just Passing Key-Value Pairs**

**✅ What is Model in Spring MVC?**

* Model is an interface provided by Spring.
* It is used to **send data** from the **Controller** to the **JSP/View** page.
* It works like a container that holds **attributes (data)** that we want to show in the view.

**🔄 Example (Using Model)**

@Controller

public class MyController {

@RequestMapping("/show")

public String showData(Model model) {

model.addAttribute("name", "Pavan");

model.addAttribute("age", 25);

return "myPage"; // this goes to myPage.jsp

}

}

In JSP:

Name: ${name}

Age: ${age}

**❓ Why use Model instead of passing raw data or key-value pairs manually?**

**💬 First, let's understand what "passing key-value pairs manually" means**

You might imagine doing something like this:

public String showData() {

String name = "Pavan";

int age = 25;

return "myPage.jsp"; // trying to go to JSP

}

But here’s the **problem**:  
There’s **no way to send name and age to the JSP** because:

* You only return a view name (myPage.jsp)
* But you’re **not telling Spring what data to send to the view**

So even if name and age exist inside this method, the JSP will **not receive them**.

**💡 How Spring MVC works under the hood**

When you return a view like this:

return "myPage";

Spring does two things:

1. Uses the ViewResolver to find the actual JSP file (/WEB-INF/views/myPage.jsp)
2. Looks at the Model object to find any data that needs to be shown on the JSP

**✅ That’s why we need Model**

Using Model lets you **formally add data** that should be sent to the view.

public String showData(Model model) {

model.addAttribute("name", "Pavan");

model.addAttribute("age", 25);

return "myPage";

}

Now Spring knows:

* View to render: myPage.jsp
* Data to show: name = Pavan, age = 25

In the JSP file, you can now easily write:

Name: ${name}

Age: ${age}

Behind the scenes:

* ${name} → calls model.getAttribute("name")
* The value "Pavan" is printed

**🔴 Why you can’t pass raw data directly without Model**

* Spring MVC follows a **framework pattern**: Controller → Model → View.
* It needs a **standard way** to pass data, and that’s what Model provides.
* You can't just return random values (like strings or numbers) and expect Spring to "guess" where and how to show them on a JSP.

**✅ Think of Model like a delivery box 📦**

Imagine:

* Controller = Restaurant chef 🍳
* Model = Food delivery box 📦
* View = Customer table 🍽️ (JSP)

If the chef cooks the food (data) but doesn’t put it in the box (Model), how will the food reach the table (View)?  
**Model** is the only standard way to "transport" that data.

**✅ Why Use a Model Class (JavaBean/POJO) and Pass Its Reference to JSP?**

**🔍 What We’re Comparing:**

| **Option 1** | **Option 2** |
| --- | --- |
| Add individual values to model using addAttribute("name", "Pavan") | Create an object like Student s = new Student(...) and do model.addAttribute("student", s) |
| Pass key-value pairs one by one | Pass one object with all the data inside |

**✅ Advantages of Using a Model Class Object**

**1. ✅ Groups Related Data Together (Object-Oriented Design)**

Instead of passing:

model.addAttribute("id", 101);

model.addAttribute("name", "Pavan");

model.addAttribute("marks", 90);

We do:

Student s = new Student();

s.setId(101);

s.setName("Pavan");

s.setMarks(90);

model.addAttribute("student", s);

📌 Now all related data is stored in **one object**, which is easier to manage, pass, and understand.

**2. ✅ Reduces Code Repetition**

Imagine a form with 10 fields — you would need 10 addAttribute() calls if you used individual values.  
But with a model object, you only need **one line**:

model.addAttribute("student", studentObject);

**3. ✅ Easier to Work With Forms and Binding**

Spring MVC supports automatic form data binding to model objects. Example:

In your form (JSP):

<form:form modelAttribute="student">

<form:input path="name" />

<form:input path="marks" />

</form:form>

When the form is submitted, Spring will automatically bind the form data into a Student object — this only works **if you use a model class**, not individual attributes.

**4. ✅ Cleaner JSP Access**

In JSP:

${student.name} // behind the scenes → student.getName()

${student.marks} // → student.getMarks()

This is more readable and structured than having:

${name}

${marks}

${id}

You always know all values are coming from student.

**5. ✅ Scales Well in Larger Applications**

In real-world apps, you deal with complex data like:

* List of students
* Student with nested objects (Address, Marks, etc.)
* Form submissions and validations

Model objects help you manage this easily. You can also reuse the same model class across many pages.

**6. ✅ Supports Framework Features Like Validation, Binding, etc.**

* You can use annotations like @Valid, @NotNull, etc., on your model class fields.
* Spring can automatically validate input and bind form values to the object.

**🔁 Summary**

| **Reason** | **Benefit** |
| --- | --- |
| Groups data logically | One object holds all related fields |
| Reduces code | Fewer addAttribute() calls |
| Enables data binding | Works with forms easily |
| Cleaner JSP syntax | Use ${student.name} instead of many separate keys |
| Easier maintenance | Change field once in class instead of everywhere |
| Supports validations | Use JSR-303 annotations (@NotNull, etc.) |

**✅ Real Example (Controller + Model + JSP)**

**Controller:**

@RequestMapping("/show")

public String showPage(Model model) {

Student s = new Student(101, "Pavan", 90);

model.addAttribute("student", s);

return "studentPage";

}

**Student.java (Model class):**

public class Student {

private int id;

private String name;

private int marks;

// constructor, getters, setters

}

**studentPage.jsp:**

<h2>Student Info</h2>

ID: ${student.id} <br/>

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

Marks: ${student.marks}

**✅ Advantages of Using Model**

**1. Clean separation between logic and view**

* Controller focuses only on processing and preparing data.
* View (JSP) handles displaying data.
* Model acts as a bridge between them.

**2. Stores multiple values in a structured way**

* You can store multiple values (strings, objects, collections) using model.addAttribute(key, value).
* It helps in passing complex data like Java objects to the view easily.

**3. Supports JavaBeans-style object binding**

* You can pass a full object (like Student, Course, etc.) to JSP.
* In JSP, you can access fields using ${student.name}, ${student.marks}, etc.
* No need to extract each field separately in the controller.

**4. Better integration with Spring’s internal mechanisms**

* Model works with ViewResolver to decide which JSP or page to load.
* It also works smoothly with form data and binding in Spring.

**5. Scales well for real projects**

* In larger apps, sending lots of individual values manually becomes hard.
* Model keeps things clean, reusable, and easy to maintain.

**🧠 Extra Learning Points (Very Useful to Know)**

**✅ Difference between Model, ModelMap, and ModelAndView**

| **Term** | **Description** |
| --- | --- |
| Model | Simple interface to pass data to the view. |
| ModelMap | Like a map, allows flexible handling of attributes. |
| ModelAndView | Combines both model data + view name in one object. |

✅ Best Practice: Use Model or ModelMap when you want cleaner code.

**✅ Common Mistake to Avoid**

* If you don't add your object or value to the model, the JSP **won’t be able to access it**.
* Just creating a Java object in the controller is not enough — you must pass it to the model using model.addAttribute().

**✅ JSP EL Syntax Recap**

* ${key} → Calls getKey() method internally if object is passed.
* Example: ${student.name} → internally calls student.getName().

**✅ Model Stores Data in Request Scope by Default**

* Data stored in Model is **only available for that request** (doesn't stay in memory after page reload).
* If you want to keep data for longer (like across pages), you can use **Session** or **ModelMap** + @SessionAttributes.

**✅ Summary**

| **Point** | **Why We Use Model** |
| --- | --- |
| 1 | To send data from Controller to View in a clean way |
| 2 | Helps pass both simple and complex data types (objects) |
| 3 | Works with JSP EL automatically |
| 4 | Scales well in real projects |
| 5 | Makes code readable and maintainable |

Eg: SpringWebMVCModelClassPOJO