# ✅ Notes: Special Parameter Types in Handler Methods (Industry Recommended with Examples)

### ✅ 1️⃣ Model

👉 Purpose:  
Used to send data from controller to view (JSP page).

#### ✅ Example:

@GetMapping("/showMessage")

public String showMessage(Model model) {

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

return "display"; // Returns display.jsp

}

📋 What Happens:

* Data added to model becomes available in JSP as ${msg}.
* Example usage in display.jsp:

<h1>${msg}</h1>

### ✅ 2️⃣ ModelMap

👉 Purpose:  
Works similar to Model, stores attributes as key-value pairs.

#### ✅ Example:

@GetMapping("/showUser")

public String showUser(ModelMap modelMap) {

modelMap.addAttribute("username", "Pavan");

return "userInfo"; // Returns userInfo.jsp

}

📋 What Happens:

* Data added to modelMap becomes available in JSP as ${username}.

### ✅ 3️⃣ HttpSession

👉 Purpose:  
Holds session data for a specific user (like login info).

#### ✅ Example:

@GetMapping("/sessionExample")

public String sessionExample(HttpSession session, Model model) {

session.setAttribute("sessionId", session.getId());

model.addAttribute("sessionInfo", "Your session ID is: " + session.getId());

return "sessionDisplay"; // Returns sessionDisplay.jsp

}

📋 What Happens:

* Session ID is available in JSP as ${sessionInfo}.
* Session attributes stay available across multiple requests by the same user.

### ✅ 4️⃣ HttpServletRequest

👉 Purpose:  
Used to access HTTP request-specific data, like headers or query parameters.

#### ✅ Example:

@GetMapping("/clientInfo")

public String clientInfo(HttpServletRequest request, Model model) {

String clientIP = request.getRemoteAddr();

model.addAttribute("clientIP", clientIP);

return "clientDisplay"; // Returns clientDisplay.jsp

}

📋 What Happens:

* Client IP address available in JSP as ${clientIP}.

### ✅ 5️⃣ HttpServletResponse

👉 Purpose:  
Used to manipulate the HTTP response directly (e.g., setting headers, sending files).

#### ✅ Example (File Download):

@GetMapping("/downloadFile")

public void downloadFile(HttpServletResponse response) throws IOException {

response.setContentType("application/octet-stream");

response.setHeader("Content-Disposition", "attachment; filename=\"example.txt\"");

PrintWriter out = response.getWriter();

out.println("This is a sample file content.");

}

📋 What Happens:

* Browser directly downloads a file named example.txt with the provided content.

### ✅ 6️⃣ What NOT to Use as Parameters

| **Wrong Practice** | **Why Not Allowed** |
| --- | --- |
| ServletContext | Global object → Should be injected using @Autowired if needed |
| ServletConfig | Global object → Should be injected using @Autowired if needed |
| Model as Return Type | Bad practice → No control of Logical View Name (LVN) and wastes shared memory |

### ✅ Final Simple Thought

👉 Industry-Recommended Approach:

* Use Model, ModelMap, HttpSession, HttpServletRequest, and HttpServletResponse as method parameters when needed.
* Never pass ServletContext or ServletConfig as handler method parameters.
* Keep the code clean, predictable, and easy to maintain.

**✅ @PutMapping – Used for updating existing data**

🟡 **What it means:**  
This method will run when a **PUT request** is made to the server.  
But remember: Browsers like Chrome or Firefox **can’t send PUT requests directly from the address bar** (they only support GET there).  
So, PUT requests are usually sent using:

* A form (with JavaScript)
* Tools like **Postman**
* **Axios/Fetch** in frontend frameworks (React, Angular, etc.)

🟢 **Example Code:**

@PutMapping("/students/{id}")

public String updateStudent(@PathVariable int id, @RequestBody Student student) {

studentService.update(id, student);

return "Student updated successfully";

}

🟢 **How it works in real-time:**

1. A **client (frontend)** sends a PUT request to URL /students/10.
2. The @PutMapping method is triggered with:
   * id = 10
   * Updated student data in request body (e.g., name, email)
3. The method updates the student info in the database.

**✅ @DeleteMapping – Used for deleting data**

🟡 **What it means:**  
This method is called when a **DELETE request** is made.  
Just like PUT, browsers **don’t send DELETE requests from the address bar**, so it’s used via:

* Postman or tools like curl
* JavaScript/React/Angular HTTP clients

🟢 **Example Code:**

@DeleteMapping("/students/{id}")

public String deleteStudent(@PathVariable int id) {

studentService.delete(id);

return "Student deleted successfully";

}

🟢 **How it works in real-time:**

1. A client app or Postman sends a DELETE request to /students/10.
2. Spring calls this method with id = 10.
3. It deletes the student with ID 10 from the database.

**❌ Can a browser send a POST request using the address bar?**

**No**, a browser **cannot send a POST request directly from the address bar**. Here's why:

**🔍 Explanation:**

* When you **type a URL in the browser's address bar and press Enter**, it always sends a **GET request**.
* The address bar **does not allow** sending a **POST**, **PUT**, or **DELETE** request.
* To send a **POST request**, you need to use:
  + An **HTML <form>** with method="post"
  + JavaScript (e.g., fetch(), axios)
  + Tools like **Postman**, **curl**, etc.

**✅ Example:**

<form action="/students" method="post">

<input name="name" value="John" />

<button type="submit">Submit</button>

</form>

When you click "Submit", the browser will send a **POST** request to /students.

**✅ Updated Summary Table**

| **Annotation** | **Sent via Browser Address Bar?** | **Usually Triggered By** | **Common Use Case** |
| --- | --- | --- | --- |
| @GetMapping | ✅ Yes | Typing URL in browser | View or fetch data |
| @PostMapping | ❌ No | HTML form, JavaScript, Postman | Submit new data |
| @PutMapping | ❌ No | JavaScript, Postman, REST client | Update existing data |
| @DeleteMapping | ❌ No | JavaScript, Postman, REST client | Delete existing data |

Eg: SpringWebMVCMultipleControllers

**✅ Can we create multiple controllers in Spring Web MVC?**

**Yes**, we can create **multiple controller classes** in a Spring Web MVC application.

**🔸 What is a Controller?**

A **Controller** is a special Java class that handles **web requests** (like clicking a button, submitting a form, or visiting a URL).

In Spring MVC, we mark it using:

@Controller

public class MyController {

// request handling methods

}

**✅ Why use multiple controllers?**

Using multiple controllers helps us to:

* Organize code **cleanly** (one controller per module/feature)
* Make **code easier to read** and **maintain**
* Avoid putting too many methods in one class

📌 **Example**:

* StudentController – for student-related requests
* CourseController – for course-related requests

**✅ One controller can have many request methods**

Inside a single controller class, we can write **multiple methods**, and each method can handle **different requests**.

📌 Example:

@Controller

@RequestMapping("/student")

public class StudentController {

@GetMapping("/add")

public String showAddForm() {

return "add-student";

}

@PostMapping("/save")

public String saveStudent(Student student) {

// save logic here

return "success";

}

@GetMapping("/list")

public String listStudents(Model model) {

// list logic here

return "students-list";

}

}

✅ All these methods are inside **one controller class** and each handles a different **URL path**.

**✅ Using class-level URL mapping**

We can set a **common path** for all methods inside a controller using @RequestMapping at **class level**.

📌 Example:

@RequestMapping("/student")

public class StudentController {

// Now all methods in this class will start with "/student"

}

Then method-level mappings will be added to it:

@GetMapping("/add") → Full path becomes → /student/add

@GetMapping("/list") → Full path becomes → /student/list

**✅ Each method must have a unique URL pattern**

Inside a controller class, you **cannot have two methods handling the same path and same HTTP method** (like two @GetMapping("/list")). This will confuse Spring.

🛑 **Wrong Example**:

@GetMapping("/list")

public String list1() { }

@GetMapping("/list")

public String list2() { } // ❌ Not allowed, same URL and HTTP method

✅ Instead, make each method **unique**:

@GetMapping("/list")

public String listStudents() { }

@GetMapping("/details")

public String showDetails() { }

**✅ Different controllers can also have class-level mappings**

Each controller class can map to a different base path using @RequestMapping.

📌 Example:

@Controller

@RequestMapping("/student")

public class StudentController {

// Handles student-related requests

}

@Controller

@RequestMapping("/course")

public class CourseController {

// Handles course-related requests

}

✅ This helps in keeping all student URLs under /student/\*, and all course URLs under /course/\*.

**🔚 Final Notes (Key Points Recap)**

| **Concept** | **Explanation** |
| --- | --- |
| Multiple Controllers | Yes, you can create many controller classes. |
| One Controller – Many Methods | One class can have many request handler methods. |
| Class-Level Mapping | Use @RequestMapping at class level to group URLs. |
| Unique Method URLs | Method-level URLs must be unique within the same controller. |
| Clean Structure | Split features into separate controllers for better code management. |

**🔹 What is an Embedded Container?**

* Spring Boot applications come with a **built-in (embedded)** web server.
* The most commonly used embedded server is **Apache Tomcat** (by default).
* When we create and run a Spring Boot web app, the embedded server **starts automatically** with our application.
* This means we **don’t need to install Tomcat or any other web server separately**.
* Once we stop the app, the **embedded server also stops**.

🧠 **Think of it like this**: Spring Boot packs a mini Tomcat server inside your project — so it’s portable and can run anywhere.

**🔸 Can I run multiple Spring Boot web apps in one embedded container?**

**❌ No, it's not possible.**

Why?

* Because each Spring Boot application comes with **its own embedded container**.
* The embedded server is **part of the application**, not shared.
* So if you create two Spring Boot web apps:
  + Each one will have **its own copy of the embedded Tomcat**.
  + Each one will run **independently** on different ports.

📝 Example:

| **Application** | **Port** | **Embedded Server** |
| --- | --- | --- |
| App1 | 8080 | Tomcat |
| App2 | 8081 | Tomcat |

You cannot combine both apps into a **single embedded container**, because each app is **self-contained**.

**🔸 Can I run multiple web apps on an external server?**

**✅ Yes, this is possible.**

How?

* You install a **standalone (external)** Tomcat server.
* You can deploy multiple WAR files (web applications) into this single external Tomcat.
* Each web application is **separated by a project name**, also called the **context path**.
* This way, Tomcat knows which app to call based on the URL.

📝 Example:

If two apps are deployed in the same external server:

* App1 with context path /shop
* App2 with context path /admin

Then the URLs will be:

http://localhost:8080/shop/

http://localhost:8080/admin/

The context path helps the external server **differentiate between applications**.

**🔸 What is a Context Path?**

* A **context path** is a name given to your application in the URL.
* It helps you **identify your app** when it's hosted on a server (especially an external one).
* In Spring Boot, **context path is optional** because by default, it runs on the root (/) path.

🧠 **Without context path**:

http://localhost:8080/hello

🧠 **With context path**:

If you set the context path as /myproject:

http://localhost:8080/myproject/hello

**🔧 How to Set Context Path in Spring Boot?**

You can set it in your application.properties file:

server.servlet.context-path=/myproject

Or, if you're using application.yml:

server:

servlet:

context-path: /myproject

This tells Spring Boot to **prefix every controller path** with /myproject.

**🔄 Embedded Server vs External Server Comparison**

| **Feature** | **Embedded Server** | **External Server (e.g., Tomcat)** |
| --- | --- | --- |
| Provided by Spring Boot | ✅ Yes | ❌ No, you install it manually |
| Shared between apps | ❌ No (1 app per server) | ✅ Yes (can deploy many apps) |
| Requires WAR packaging | ❌ No (can use JAR) | ✅ Yes (usually requires WAR) |
| Context path default | / (root) | Must set unique context for each app |
| Deployment process | Run the app directly (java -jar) | Deploy WAR file in /webapps folder |

**💡 Final Summary (Simple Takeaway)**

1. Spring Boot runs with an **embedded server** (like Tomcat) — **1 app per server only**.
2. You **cannot** run multiple Spring Boot apps in one embedded container.
3. You **can run multiple apps** in an external Tomcat — using **context paths** to separate them.
4. Spring Boot apps run on root path by default, but you can **set a custom path** using server.servlet.context-path.

Eg: SpringWebMVCContextPath