While building SpringMVC/SpringBootMVC applications we need to use the following annotations while developing handler classes =>

To make java class as SpringBean + handler/controller class

**a. @Controller**

* This is used on a **Java class**.
* It tells Spring:

“Hey Spring, treat this class as a **Controller (handler class)** which can handle web requests.”

* By default, when you mark a class with @Controller, Spring will create an object of this class and keep it inside the **Spring container** (it becomes a **Spring Bean**).

So:  
👉 @Controller = Make this class a **Spring-managed handler class**.

**b. @RequestMapping**

* This is used on a **method** inside the controller.
* It tells Spring:

“When a request with a particular **URL (path)** and **HTTP method (GET/POST)** comes, call this Java method to handle it.”

**Note:**

* Browser can send only GET (default) and POST requests directly.
* Other request types like PUT, DELETE are usually sent by tools/APIs (not from normal HTML forms).

**Example in Code**

@Controller

public class LoginController {

// This method handles GET request for "/login"

@RequestMapping(value="/login", method=RequestMethod.GET)

public String login(String username, String password) {

// 1. We can write logic to check if username & password are valid

// Example: if(username.equals("admin") && password.equals("1234")) ...

// 2. Or we can send the request to a service class to do the validation.

// 3. Finally, we return the name of the view (JSP/HTML/Thymeleaf page) to show result.

return "loginSuccess"; // means it will open loginSuccess.jsp or loginSuccess.html

}

}

**🔷 Developing Spring Web MVC Application (Spring Boot)**

**✅ 1. spring-boot-starter-web dependency**

* This dependency is added in pom.xml.
* It provides all the required libraries to build a **Spring Web MVC** application.
* It includes:
  + Embedded Tomcat Server
  + Spring MVC modules
  + JSON support
  + DispatcherServlet, etc.

**✅ 2. @Controller Annotation**

* This annotation is used on a class.
* It tells Spring that the class is a **Controller** (also called Request Handler).
* It will handle **web requests** (like URL calls from the browser).

**✅ 3. @RequestMapping Annotation**

* This is used to **map a URL pattern to a controller method**.
* When a client sends a request to a specific URL, the corresponding method is executed.
* Example:

@RequestMapping("/welcome")

public String showWelcomePage() { }

**✅ 4. Returning ModelAndView Object**

* Controller method returns a ModelAndView object.
* This object contains:
  + a) **Model data** → in the form of key-value pairs
  + b) **Logical view name** → without extension

**✅ 5. Accessing Model Data in View File**

* The data sent using Model (in the form of key-value)  
  can be **accessed in the view file** using the key.
* Example in JSP:

<h1>Hello ${username}</h1>

(Here username is the key used in Model)

**✅ 6. @GetMapping Annotation**

* Shortcut for @RequestMapping(method = RequestMethod.GET)
* Used to bind a controller method to a **HTTP GET request**.
* Commonly used to **display data or forms**.

**✅ 7. Model Interface**

* Provided by Spring.
* It is used to **store data as key-value pairs**.
* This data is passed from **Controller to View (UI)**.
* View file uses this data to display on the screen.

Client Request → @Controller method → prepares data in Model → returns ModelAndView → View displays data

Eg: SpringWebMVC

### 🔹 @RequestMapping

* This is used to map (or connect) a method in your controller to a **web request**.
* By default, it **handles GET requests** (unless you specify another type).
* It is used like this:

@RequestMapping("/hello")

public String sayHello() {

return "hello";

}

* In the above example, if a user types http://localhost:8080/hello in the browser, the sayHello() method will be called.

### 🔹 From Spring 4.3 onward, we have ****shortcut annotations**** for different types of HTTP requests:

#### ✅ @GetMapping

* Used when we want to handle a **GET request**.
* GET is used to **fetch data** (like loading a webpage or getting details).
* Example:

@GetMapping("/students")

public List<Student> getAllStudents() {

return studentService.getStudents();

}

* This method will run when the browser or client makes a GET request to /students.

#### ✅ @PostMapping

* Used when we want to handle a **POST request**.
* POST is used to **send data to the server** (like submitting a form or saving new data).
* Example:

@PostMapping("/students")

public String saveStudent(@RequestBody Student student) {

studentService.save(student);

return "Student saved";

}

* This method runs when data is **sent** (posted) to /students.

**✅ @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