SpringBoot Web Application

Web Applications in the Spring Framework Ecosystem

- Spring is an **umbrella framework** consisting of multiple projects.
- Several Spring projects are designed specifically for building **web applications**.
- For web development:
  - **Spring Web** can be used.
  - **Spring MVC** can be used when not working with Spring Boot.

Why Web Applications Are Needed

- Most modern systems rely on **web applications**.
- Although **mobile applications** exist, the majority of them also rely on a **web backend**.
- The same backend typically serves:
  - Browser-based desktop clients
  - Mobile application clients
- The focus is on building the **server-side backend** that supports these clients.

Client–Server Interaction and Dynamic Content

- Clients can be:
  - Desktop browsers
  - Mobile applications
- A client usually displays a **UI layout**, while the **data** comes from the server.
- Static content:
  - Same for all users
  - Can be built using **HTML** and **CSS**
- Dynamic content:
  - Varies per user
  - Requires **backend programming**
- Java is used to generate and provide this dynamic data.

Data Exchange Using JSON

- Modern applications focus on transferring **data**, not full HTML pages.
- Backend servers typically send **JSON data** to clients.
- Clients may also send data back to the server in **JSON format**.
- This approach is common for:
  - Web applications built with **React**, **Angular**, or **Vanilla JavaScript**
  - Mobile applications

Example of Server Data Usage

- Mobile applications may request data such as:
  - Live sports scores
  - User-specific information
- The server generates this data, which may come from:
  - A database
  - Another external server
- The server responds with data that the client displays.

Introduction to Servlets

- Servlets are Java components used on the **server side**.
- The term servlet comes from **serve + let**.
- Responsibilities of a servlet:
  - Accept client requests
  - Process requests
  - Send responses back to clients

Servlet Containers (Web Containers)

- Servlets cannot be run like normal Java programs on the JVM.
- They require a special runtime environment called a **Servlet Container** or **Web Container**.
- The container provides capabilities such as:
  - Receiving requests from the internet
  - Managing servlet lifecycle
  - Sending responses to clients

Apache Tomcat as a Servlet Container

- **Apache Tomcat** is a lightweight and commonly used servlet container.
- It allows developers to:
  - Deploy and run servlets
  - Handle HTTP requests and responses

Relationship Between Spring and Servlets

- Even when using **Spring Web** or **Spring MVC**, servlets are used internally.
- Spring simplifies servlet-based development by:
  - Abstracting low-level servlet code
  - Providing easier configuration and development patterns
- Spring still relies on servlets **behind the scenes**.

Approaches to Building Web Applications

- Two main approaches are mentioned:
  1. **Servlet-based approach**
  2. **Reactive approach**
- The focus of this module is on the **servlet-based approach**.

Learning Path for Web Development with Spring

- Development will begin by:
  - Understanding **servlets first**
- Then progress towards:
  - Using **Spring** and its web-related projects
- This approach helps build a strong foundation before using Spring abstractions.

Web Applications in the Spring Framework Ecosystem

- Spring is an **umbrella framework** consisting of multiple projects.
- Several Spring projects are designed specifically for building **web applications**.
- For web development:
  - **Spring Web** can be used.

- **Spring MVC** can be used when not working with Spring Boot.

Why Web Applications Are Needed

- Most modern systems rely on **web applications**.
- Although **mobile applications** exist, the majority of them also rely on a **web backend**.
- The same backend typically serves:
  - Browser-based desktop clients
  - Mobile application clients
- The focus is on building the **server-side backend** that supports these clients.

Client–Server Interaction and Dynamic Content

- Clients can be:
  - Desktop browsers
  - Mobile applications
- A client usually displays a **UI layout**, while the **data** comes from the server.
- Static content:
  - Same for all users
  - Can be built using **HTML** and **CSS**
- Dynamic content:
  - Varies per user
  - Requires **backend programming**
- Java is used to generate and provide this dynamic data.

Data Exchange Using JSON

- Modern applications focus on transferring **data**, not full HTML pages.
- Backend servers typically send **JSON data** to clients.
- Clients may also send data back to the server in **JSON format**.
- This approach is common for:
  - Web applications built with **React**, **Angular**, or **Vanilla JavaScript**
  - Mobile applications

Example of Server Data Usage

- Mobile applications may request data such as:

    - Live sports scores
    - User-specific information
- The server generates this data, which may come from:
    - A database
    - Another external server
- The server responds with data that the client displays.

Introduction to Servlets

- Servlets are Java components used on the **server side**.
- The term servlet comes from **serve + let**.
- Responsibilities of a servlet:
    - Accept client requests
    - Process requests
    - Send responses back to clients

Servlet Containers (Web Containers)

- Servlets cannot be run like normal Java programs on the JVM.
- They require a special runtime environment called a **Servlet Container** or **Web Container**.
- The container provides capabilities such as:
    - Receiving requests from the internet
    - Managing servlet lifecycle
    - Sending responses to clients

Apache Tomcat as a Servlet Container

- **Apache Tomcat** is a lightweight and commonly used servlet container.
- It allows developers to:
    - Deploy and run servlets
    - Handle HTTP requests and responses

Relationship Between Spring and Servlets

- Even when using **Spring Web** or **Spring MVC**, servlets are used internally.
- Spring simplifies servlet-based development by:
    - Abstracting low-level servlet code
    - Providing easier configuration and development patterns

- Spring still relies on servlets **behind the scenes**.

Approaches to Building Web Applications

- Two main approaches are mentioned:
    1. **Servlet-based approach**
    2. **Reactive approach**
- The focus of this module is on the **servlet-based approach**.

Learning Path for Web Development with Spring

- Development will begin by:
    - Understanding **servlets first**
- Then progress towards:
    - Using **Spring** and its web-related projects
- This approach helps build a strong foundation before using Spring abstractions.

Working with Servlets and Tomcat

Why Servlets Need a Container

- Servlets cannot run on their own; they require a **Servlet Container**.
- **Apache Tomcat** is the most commonly used servlet container.
- The container is responsible for:
    - Managing servlet lifecycle
    - Handling HTTP requests and responses
    - Providing servlet APIs

Packaging Java Applications

- **Console applications** are packaged as .jar files.
- **Web applications** are traditionally packaged as .war (Web ARchive) files.
- A .war file is deployed to a servlet container like Tomcat.

Using External Tomcat

- Steps involved:
    - Download Apache Tomcat from the official website
    - Unzip the distribution

        - Place the .war file inside the webapps folder
        - Start the server using scripts from the bin directory
            - startup.sh to start
            - shutdown.sh to stop
    - This approach gives more control and features, useful for deep servlet-based applications.

Embedded Tomcat Concept

- Instead of installing Tomcat separately, it can be **embedded inside the project**.
- When the project runs, Tomcat starts automatically.
- Advantages:
    - No external server configuration
    - Easier project setup
    - Faster development and testing
- Since the focus is on **Spring**, embedded Tomcat is sufficient for learning servlets.

Creating a Servlet Project (Maven)

- Create a **Maven project** from the IDE
- Example project details:
    - Project name: servlet-example
    - Java version: **Java 17**
    - Group ID: com.telescope
- Default Maven project includes JUnit dependency.

Required Dependencies for Servlets

Servlet support is **not part of the JDK**, so additional dependencies are required.

1. Servlet API

- Provides core servlet classes and interfaces
- After Java EE, servlet APIs moved to **Jakarta Servlet**
- Used only at compile time (provided by the container at runtime)

2. Embedded Tomcat

- Allows running Tomcat directly from the application
- Required to start and manage the servlet container internally

---

Dependency Summary

- **Jakarta Servlet API**: Enables servlet development
- **Tomcat Embedded Core**: Runs the servlet container within the application
- Dependencies are managed using **Maven Repository**

Creating and Running Your First Servlet (with Embedded Tomcat)

Now that our project structure is ready, let's create a **Servlet** and understand how it actually works behind the scenes.

---

1. Creating a Servlet Class

In Java, whenever we want to add a new feature, we create a **class**. A servlet is no different.

- Right-click on the project → **New → Class**
- Name it something meaningful, for example: HelloServlet

  Using Servlet at the end of the class name is a good convention. It immediately tells other developers that this class is a servlet.

---

2. Making a Class a Servlet

Just naming a class HelloServlet does **not** make it a servlet.

To give servlet behavior to a class, we must:

- **Extend HttpServlet**

HttpServlet provides built-in support for:

- Handling HTTP requests
- Sending HTTP responses

```
public class DemoServlet extends HttpServlet {
}
```

---

3. The service() Method

A servlet needs a method that executes **whenever a request comes in**.

That method is called **service()**.

```
@Override
public void service(HttpServletRequest request, HttpServletResponse response) {
   System.out.println("Service method called");
}
```

Why service()?

- It is automatically invoked by Tomcat
- It receives two important objects:
    - HttpServletRequest → data coming from the client
    - HttpServletResponse → data sent back to the client

> For now, we're just printing a message to verify that the servlet is being called.

---

4. Why Running main() Is Not Enough

If you run the default main() method, you'll only see:

Hello World

You **won't** see Service method called.

Why?

- Servlets do **not** run like normal Java classes
- A servlet runs **only when an HTTP request is sent**
- That request must be handled by a **Servlet Container** (Tomcat)

---

5. Starting Embedded Tomcat

Since we are using **embedded Tomcat**, we must explicitly start it in our main() method.

```
Tomcat tomcat = new Tomcat();
tomcat.start();
tomcat.getServer().await();
```

Explanation

- new Tomcat() → creates a Tomcat server instance
- start() → starts the server
- getServer().await() → keeps Tomcat running

Without await(), Tomcat starts and immediately shuts down.

———

6. Verifying Tomcat Is Running

When Tomcat starts successfully, you'll see messages like:

```
Starting service [Tomcat]
Starting ProtocolHandler [http-nio-8080]
```

At this point:

- Tomcat is running
- Port **8080** is active

However, if you open the browser and visit:

http://localhost:8080

You'll still see an error.

———

7. Why the Servlet Is Still Not Called

Even though Tomcat is running, it does **not know**:

- Which URL should trigger which servlet

In real websites, URLs look like:

/domain/page

Similarly, we want:

http://localhost:8080/hello

To trigger our HelloServlet.

To make this happen, we need **URL mapping** between:

- /hello → HelloServlet

———

8. What's Next?

So far, we have:

- Created a servlet
- Started embedded Tomcat
- Kept the server running

❌ Missing piece: **Servlet URL mapping**

Servlet URL Mapping with Embedded Tomcat

Overview of Servlet Mapping Approaches

- **XML-based configuration (web.xml)**
  - Earlier approach used when deploying on **external Tomcat**.
  - Mapping defined by specifying:
    - URL pattern
    - Servlet class
  - Tomcat uses this mapping to decide which servlet to execute for a given request.
- **Annotation-based configuration (@WebServlet)**
  - Uses the @WebServlet annotation on top of the servlet class.
  - Example usage: mapping a /hello request to a specific servlet.

        - Works automatically when using **external Tomcat**.
- **Embedded Tomcat limitation**
        - When using **embedded Tomcat**, annotation-based mapping is not automatically picked up.
        - Mapping must be configured **programmatically**.

Creating Servlet Mapping Programmatically

Step 1: Obtain the Context Object

- The **Context** represents the web application configuration inside Tomcat.
- It is retrieved using the Tomcat instance.
- Parameters:
        - Application context path: empty string ("") for default application
        - Directory structure: null since no external directory is created

```
Context context = tomcat.addContext("", null);
```

Step 2: Register the Servlet with Tomcat

- Use Tomcat.addServlet() to register the servlet.
- Required parameters:
        1. Context object
        2. Servlet name (logical name)
        3. Servlet instance (object)

```
Tomcat.addServlet(context, "DemoServlet", new DemoServlet());
```

- The servlet name is an identifier and does not have to match the class name, but it must be consistent.

Step 3: Map URL Pattern to the Servlet

- Use context.addServletMappingDecoded() to map a URL to the servlet.
- Parameters:
        1. URL pattern (e.g., /hello)
        2. Servlet name (same name used during registration)

```
context.addServletMappingDecoded("/hello", "DemoServlet");
```

- This mapping ensures that requests to /hello invoke the DemoServlet.

Starting the Embedded Tomcat Server

- Set the port number (default is 8080, can be changed if needed).
- Start the Tomcat server.
- Keep the server running using await().

```
tomcat.setPort(8081); // Optional: change port if required
tomcat.start();
tomcat.getServer().await();
```

- Without await(), the server would start and immediately stop.

Result of the Mapping

- Accessing http://localhost:8081/hello:
  - Successfully triggers the servlet
  - Executes the service() method
  - Confirms servlet mapping is working
- Browser output:
  - Blank page (no response body yet)
- Console output:
  - Displays Service Method called

Notes on Port Configuration

- Default Tomcat port: **8080**
- Can be changed using tomcat.setPort(portNumber)
- Useful when another service is already using port 8080

Servlet Class Note

- Annotation shown below is **not used** with embedded Tomcat mapping:

```
//@WebServlet("/hello") // Used only with external Tomcat
public class DemoServlet extends HttpServlet {
```

}

Next Step

- Sending data back to the client using HttpServletResponse
- Writing output (e.g., “Hello World”) to the browser response

Sending Response from a Servlet

- Servlets send data back to the client using the **HttpServletResponse** object.
- By default, the response body is empty unless data is explicitly written to it.
- To write data:
  - Obtain a writer using response.getWriter().
  - Use print or println to write content.
- Writing to the response is conceptually similar to writing on paper using a pen:
  - Response object → paper
  - Writer → pen

Writing Plain Text Response

- Example approach:
  - Call response.getWriter().println("Hello World").
  - This sends text data to the client.
- Writing to the response may throw **IOException**, which must be handled or declared.

Using PrintWriter Explicitly

- response.getWriter() returns a **PrintWriter** object.
- Assigning it to a variable improves readability and familiarity:
  - Similar to System.out.println, but output goes to the client instead of the console.
- Output destination depends on where the PrintWriter is obtained from.

Sending HTML Content

- Servlets can send HTML content directly in the response.
- HTML tags can be written using the writer.
- By default, HTML is treated as plain text unless specified otherwise.

Setting Content Type

- To ensure HTML is rendered correctly, the response content type must be set:
    - response.setContentType("text/html")
- This informs the client that the response contains HTML, not plain text.

HTTP Request Methods in Servlets

- HTTP defines multiple request methods:
    - **GET** – Retrieve data
    - **POST** – Send data to the server
    - **PUT** – Update existing data
    - **DELETE** – Remove data
- Browser URL access triggers a **GET** request by default.

Handling Request Types with Servlet Methods

- Instead of using the generic service() method, specific methods are preferred:
    - doGet() – Handles GET requests
    - doPost() – Handles POST requests
- Using method-specific handlers improves clarity and aligns with HTTP semantics.

Servlets, JSP, and MVC Architecture

Web Applications in Java

- A **web application** acts as a backend system that:
    - Accepts requests from clients (browser, mobile app, another server)
    - Processes those requests
    - Sends responses back to the client
- In Java, the backend is typically built using **Servlets**.
- The frontend is **not** built in Java. It is usually created using:
    - HTML, CSS, JavaScript (Vanilla JS)
    - Frontend frameworks like **React**
    - Mobile applications
    - Other backend servers consuming APIs

The backend's main responsibility is to **return data** to the client.

Problem with Servlets Returning HTML

- A servlet can:
    - Accept a request
    - Talk to the database
    - Fetch data
    - Return the response
- Returning **raw data** is simple, but:
    - Browsers expect **HTML** for meaningful display
- If a servlet returns HTML directly:
    - HTML tags must be written inside Java code
    - This makes code bulky, unreadable, and hard to maintain

This leads to the need for a **better solution**.

JSP (Java Server Pages)

- **JSP** is a view technology used to generate HTML pages
- Instead of writing HTML inside Java:
    - You write an **HTML page**
    - Embed Java code inside it where required
- JSP allows:
    - Clean separation of UI and backend logic
    - Easier maintenance of frontend pages
- JSP is **not the only view technology**, but it is a commonly used one in Java-based web apps

---

Servlet + JSP Together

- In a real application, both are used together:
    - **Servlet**:
    - Accepts client requests
    - Performs processing
    - Fetches data from the database
    - **JSP**:
    - Receives data from the servlet
    - Generates a beautiful HTML page
    - Sends it back to the client
- The servlet sends **data**, not HTML, to the JSP

Importance of Objects (Model)

- Java is **object-oriented**
- Data should be represented as **objects**, not scattered variables

Example:

- Client requests student data (e.g., ID = 101)
- Data fetched from database should be stored in a **Student object**

These objects represent the **Model** of the application

MVC Architecture (Model–View–Controller)

MVC is a **design pattern** used across web applications (not Java-specific).

Components:

1. **Controller**
   - Accepts client requests
   - Decides what action to take
   - Communicates with database or services
2. **Model**
   - Holds application data
   - Represented using Java objects
   - Typically POJO classes
3. **View**
   - Responsible for UI
   - Displays data to the client
   - Implemented using JSP or other view technologies

Flow:

1. Client sends request
2. Request goes to **Controller** (Servlet)
3. Controller processes request and prepares **Model**
4. Controller sends model data to **View** (JSP)
5. View renders HTML and sends response to client

POJO (Plain Old Java Object)

- A **POJO** is a simple Java class:

- No special inheritance
- No framework dependency
- Contains fields, getters, setters

Example:

- Student class with id, name, age

POJOs are used as **Models** in MVC

Mapping MVC to What We Learned

- **Controller** → Servlets
- **View** → JSP (and other view technologies)
- **Model** → POJO classes

How JSP Works Internally

- Tomcat is:
  - A web server
  - A **servlet container**
- Tomcat can only execute **Servlets**
- So how does JSP work?

Behind the scenes:

- JSP is automatically converted into a **Servlet**
- That servlet is then compiled and executed by Tomcat

As a developer:

- You focus on HTML + embedded Java
- The JSP-to-Servlet conversion is handled internally

Spring Boot Web Application – MVC Setup (Step-by-Step)

1) Why Spring Boot for Web MVC

- Two ways to build Spring MVC:
  - **Spring Framework (traditional)** → lots of manual configuration

        - **Spring Boot** → auto-configuration, faster setup ✅
- Goal: build an **MVC-based web application** with minimal configuration.

---

2) Creating a Spring Boot Project

Recommended approach: start.spring.io

Project configuration:
- Project: **Maven**
- Language: **Java**
- Spring Boot version: **3.2.x**
- Group: com.telescope
- Artifact: spring-boot-web-1
- Packaging: **JAR**
- Java version: **21**

---

3) Why JAR Instead of WAR?
- Traditional web apps:
    - Deploy **WAR** to an **external Tomcat**
- Spring Boot approach:
    - Uses **embedded Tomcat**
    - Application runs as a **standalone JAR**
    - No external server setup required

➡️ Embedded Tomcat starts automatically when the app runs.

---

4) Dependencies Required

Add only one dependency:
- **Spring Web (spring-boot-starter-web)**

This provides:
- Embedded Tomcat
- Spring MVC
- DispatcherServlet
- REST & web support

(Reactive Web / WebFlux is a different model and not used here)

---

5) Project Structure After Generation

Key files:

- pom.xml → dependency management
- Application.java → main entry point (@SpringBootApplication)
- Embedded Tomcat visible under **External Libraries**

---

6) Running the Application

Steps:

1. Open the project in IntelliJ
2. Run the main class (Application.java)
3. Observe logs:
    - Embedded **Tomcat started on port 8080**

---

7) Accessing the Application

- URL: http://localhost:8080

Initial behavior:

- ❌ Before running → *Cannot connect to server*
- ❌ After running → **404 Not Found**

➡️ Reason: No controller / home page defined yet

---

8) Key Takeaway So Far

- Spring Boot app is running successfully
- Server (Tomcat) is up
- No request handler exists yet

Creating a Homepage in Spring Boot Using JSP

Understanding the 404 Error on Home Page

- When accessing http://localhost:8080, the browser requests the **home page**.
- The server responds with **404 Not Found**, which means:
    - The server is running.
    - There is **no resource mapped** to handle the request for /.
- In a Spring Boot web application, a home page is not available by default unless explicitly configured.

Using JSP as a View Technology

- **JSP (JavaServer Pages)** is used to return a full-fledged HTML page to the client.
- JSP allows:
    - Writing complete HTML pages.
    - Embedding Java code inside HTML.
- JSP is a **view technology** in the MVC pattern.

Creating the JSP Home Page

Required Directory Structure

- To use JSP in a Spring Boot application:
    - Create a directory named **webapp**.
    - Place JSP files inside this directory.
- Spring looks for JSP pages inside the webapp folder.

Creating index.jsp

- Inside the webapp directory, create a file named **index.jsp**.
- index.jsp is commonly used as the home page.

JSP Page Configuration

- A JSP page must declare that it supports Java.
- This is done using a **page directive** at the top of the file.

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

- This directive tells the server:

        - The language used is Java.
        - The content type is HTML.
        - Character encoding is UTF-8.

Sample JSP Home Page Code

```
<!DOCTYPE html>
<html>
<head>
    <title>Simple JSP Homepage</title>
</head>
<body>
    <h1>Welcome to My Simple JSP Homepage!</h1>
    <p>This is a basic JavaServer Pages example.</p>
    <p>Current date and time: <%= new java.util.Date() %></p>
</body>
</html>
```

- HTML tags are used to structure the page.
- Java code can be embedded using <%= %>.
- The example displays:
    - A heading.
    - A paragraph.
    - The current date and time using Java.

Why the JSP Page Still Returns 404

- Even after creating index.jsp, accessing / still returns **404**.
- Reason:
    - In **MVC architecture**, the client request does **not** directly access the JSP.
    - JSP is a **view**, not a controller.
- The request flow is:
    - Client → Controller → View (JSP)

Missing Component: Controller

- There is currently **no controller** in the application.
- Without a controller:
    - No request mapping exists.

        - JSP pages cannot be invoked.
- JSP pages are meant to be returned **by a controller**, not accessed directly by the client.

Spring Boot Application Entry Point

```
@SpringBootApplication
public class Springbootwebapp1Application {

    public static void main(String[] args) {

SpringApplication.run(Springbootwebapp1Application.class,
args);
    }
}
```

- The application starts successfully.
- Embedded **Apache Tomcat** runs on port **8080**.
- Only one dependency is used:
    - **Spring Web (spring-boot-starter-web)**
- No controller is defined yet, which is why requests fail.

Key Takeaways

- JSP is a **view technology** and cannot handle requests directly.
- The client request must first go to a **controller**.
- A controller is required to:
    - Accept the request.
    - Decide which JSP page to return.
- Creating a controller in Spring Boot will be covered next.

Title: Creating a Controller in Spring Boot

- In Spring Boot, you **don't need to manually create servlets**.
- Instead, you create a **controller class** that handles HTTP requests and delegates to the appropriate view or logic.

- Steps to create a controller:
    1. Create a **plain Java class** in your Spring Boot package.
    2. Annotate it with @Controller (stereotype annotation) to make it a **Spring-managed object**.

```
import org.springframework.stereotype.Controller;

@Controller
public class HelloController {
    // controller logic here
}
```

- Behind the scenes:
    - Spring converts the controller into a **servlet**.
    - The servlet runs on **embedded Tomcat**.
- Key takeaway: **Developers only create controllers**, not servlets.

Title: Adding Methods in a Controller

- Just like servlets have doGet() and doPost(), controllers require methods to handle specific requests.
- Example:
    - A method to call a JSP home page:

```
public String homePage() {
    return "index.jsp"; // returns the view
}
```

- **Responsibilities of this method:**
    - Handles a specific request (e.g., the home page).
    - Returns the **view name** (JSP, Thymeleaf, etc.).
    - Spring automatically resolves the view and sends it to the client.

Title: Mapping URLs to Controller Methods

- Controllers **do not automatically know** which URL they respond to.
- Each method must be mapped to a specific URL using **request mapping** (covered in the next step/video).
- Importance of mapping:
    - Ensures that when a client requests a URL, the **correct controller**

method is called.
    - Example scenario:
        - /home → returns index.jsp
        - /students → returns list of students
- Mapping is similar to **URL-to-servlet mapping in traditional servlets**.

Title: Observing Controller Behavior

- Debugging controller calls:
    - Add a System.out.println() inside the method to check if it is executed.

```
public String homePage() {
    System.out.println("Home method called");
    return "index.jsp";
}
```

    - If the console doesn't print, the **method is not mapped** or the URL is not specified.

- Key insight:
    - Without proper URL mapping, **the method cannot be accessed**, even if the controller exists.

Summary Notes

- **Controller** = Spring-managed class that handles requests (replaces manual servlet creation).
- **Method inside controller** = Handles specific requests and returns the view name.
- **Mapping** = Associates a method with a specific URL for client access.
- **Behind the scenes** = Spring converts controller → servlet → runs on Tomcat.
- Always ensure:
    1. Controller annotated with @Controller.
    2. Methods return **view names** (String).
    3. Methods are mapped to URLs using @RequestMapping or similar annotations.

Code Example (Current State)

```
package com.springbootwebapp.springbootwebapp_1;

import org.springframework.stereotype.Controller;

@Controller
public class HelloController {

    // Method responsible for calling the index.jsp view
    public String homePage() {
        return "index.jsp"; // returns the view
    }

}
```

This sets the stage for the **next step**: adding **URL mapping** (@RequestMapping or @GetMapping) to make the controller method accessible from a browser.

Title: Mapping Requests in Spring Boot Controllers

- Spring Boot controllers require **URL-to-method mapping** to handle HTTP requests.
- Use **annotations** instead of XML configuration for mapping.
- **@RequestMapping Annotation**:
  - Maps a URL path to a controller method.
  - Example:

```
@Controller
public class HelloController {

    @RequestMapping("/")  // maps the root URL to this method
    public String homePage() {
        System.out.println("Homepage called");
```

```
        return "index.jsp"; // returns the JSP view
    }
}
```

- You can map any path:
    - /home → home page
    - /students → student page

- Other specialized annotations:
    - @GetMapping → for GET requests
    - @PostMapping → for POST requests
    - More mapping annotations exist for PUT, DELETE, etc.

Title: JSP Behavior in Spring Boot

- After mapping a URL to a method, Spring Boot attempts to render the **JSP view**.
- Common issue:
    - JSP content sometimes triggers a **file download instead of rendering**.
    - Cause: Spring Boot **does not support JSP out-of-the-box**; JSP must be converted into a servlet.
- Resolution: Add **Tomcat Jasper dependency** to enable JSP compilation.

Title: Adding Tomcat Jasper Dependency

- **Purpose:** Converts JSP pages into servlets at runtime so Spring Boot can render them.
- Maven dependency example:

```
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>11.0.15</version>
    <scope>compile</scope>
</dependency>
```

- After adding:
    1. Reload Maven dependencies in your IDE.
    2. Restart the Spring Boot application.
    3. JSP pages should render correctly instead of downloading.

Title: Key Points and Observations

- Once @RequestMapping is applied:
  - The **controller method is called**.
  - The mapped view (index.jsp) is returned.
- Console debugging:
  - Add System.out.println() inside the method to verify execution.

System.out.println("Homepage called");

- Important insight:
  - Modern applications rarely use JSP in production.
  - Typically, Spring Boot **returns data** (JSON/REST APIs) to front-end frameworks like React, Angular, or mobile apps.

Title: Complete Controller Example with JSP Mapping

```
package com.springbootwebapp.springbootwebapp_1;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloController {

    @RequestMapping("/") // maps the root URL to this method
    public String homePage() {
        System.out.println("Homepage called"); // verify method
execution
        return "index.jsp"; // returns the JSP view
    }
}
```

- JSP page (index.jsp) is placed inside:

```
src/main/webapp/index.jsp
```

- The page includes:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Simple JSP Homepage</title>
</head>
<body>
    <h1>Welcome to My Simple JSP Homepage!</h1>
    <p>Current date and time: <%= new java.util.Date() %></p>
</body>
</html>
```

✅ **Summary**

- @RequestMapping maps URL requests to controller methods.
- Spring Boot requires **Tomcat Jasper** for JSP rendering.
- Controller method returns **view name** (JSP page) as a string.
- Console prints can be used to verify the controller method is invoked.
- JSP is largely for learning; modern apps use **front-end frameworks** and return **data (JSON)** instead of full pages.

Here's a **comprehensive set of notes** for your latest transcript covering **handling form input and sending requests to Spring Boot controllers**:

Title: Introducing Dynamic Operations via Spring Boot

- Objective: Move from a static "Hello World" page to a **dynamic web application** where the server performs operations based on client input.
- Use case example: **Adding two numbers**.
- Process flow:
    1. Client submits input via a form.

2. Server performs computation/logic.
3. Server returns results to the client.

- Key concept: **Backend interaction with user input**; not focusing on frontend design, but on server-side processing.

———

Title: Creating a JSP Form for Input

- **Form requirements:**
  - Two text fields for number input (num1, num2).
  - A dropdown to select the operation (add, subtract, multiply, divide).
  - A submit button to send the request to the server.
- Example JSP form (index.jsp or calculator.jsp):

```
<form action="calculate" method="post">
  <label for="num1">Enter first number:</label>
  <input type="number" id="num1" name="num1" required>

  <label for="num2">Enter second number:</label>
  <input type="number" id="num2" name="num2" required>

  <label for="operation">Select operation:</label>
  <select id="operation" name="operation" required>
    <option value="add">Addition (+)</option>
    <option value="subtract">Subtraction (-)</option>
    <option value="multiply">Multiplication (*)</option>
    <option value="divide">Division (/)</option>
  </select>

  <button type="submit">Calculate</button>
</form>
```

- The form sends a **POST request** to the /calculate endpoint.

———

Title: Styling the JSP Page with CSS

- A simple CSS file can enhance page readability and basic UI aesthetics.
- style.css example highlights:
  - Modern font stack for readability.
  - Gradient background.
  - Centered layout using flexbox.
  - Styled form inputs, labels, and buttons.
  - Responsive design for smaller screens.

```
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    background: linear-gradient(135deg, #f5f7fa, #c3cfe2);
    display: flex;
    justify-content: center;
    align-items: center;
    min-height: 100vh;
}

.container {
    max-width: 500px;
    padding: 20px;
    background-color: #fff;
    border-radius: 12px;
    box-shadow: 0 8px 20px rgba(0,0,0,0.1);
}

input, select, button {
    width: 100%;
    padding: 12px;
    margin-bottom: 20px;
    border-radius: 8px;
}
```

- CSS is linked in JSP via:

```
<link rel="stylesheet" type="text/css" href="style.css">
```

Title: Form Submission Flow

- When the user clicks **Calculate**, the browser sends a **POST request** to /calculate:
  - URL example: localhost:8080/calculate?num1=6&num2=7&operation=add
  - Server receives **parameters num1, num2, operation**.
- **404 Error Scenario:** Occurs when:
  - No controller is mapped to /calculate.
  - JSP alone cannot handle the request; controller is required.

Title: Next Step – Creating the Calculation Controller

- Need a **Spring Boot controller** method to:
    1. Accept POST request at /calculate.
    2. Read input parameters from the form (num1, num2, operation).
    3. Perform the selected operation.
    4. Return the result (as JSP view or JSON data).
- Key point: Each form submission must be mapped to a **controller method**, similar to URL-to-method mapping with @RequestMapping.

Title: Key Concepts Illustrated

- **Client-Server Interaction:**
    - Client submits data → Server processes → Server returns response.
- **Server-side computation** is separate from frontend logic.
- **Controllers** in Spring Boot handle business logic, not JSP pages.
- **URL mapping** links form action paths to controller methods.
- **POST vs GET:**
    - method="post" ensures data is sent in the body.
    - Can also use GET for simple parameter passing, though POST is preferred for operations.

✅ **Summary**

- JSP pages can include forms to send input to Spring Boot controllers.
- CSS improves readability and user experience but is optional.
- Form submission generates a request to the server.
- The server must have a **mapped controller method** to handle requests; otherwise, a 404 error occurs.
- Next step: Implement a **/calculate controller method** to process the inputs and return results.

Here's a **comprehensive set of notes** for this transcript section on **handling form submission and reading input in Spring Boot**:

Title: Handling Multiple Requests in a Single Controller

- **Spring allows multiple methods (requests) in one controller class:**
  - Example: HomeController can handle /home and /calculate requests.
- **Logical separation of controllers:**
  - UserController → manage users (add, update, delete)
  - ProductController → manage products
  - OrderController → manage orders
- In small applications, multiple request methods can coexist in the same controller.

Title: Creating a Request Handler for Form Submission

- To handle form input, create a new method in the controller:

```
@RequestMapping("/calculate")
public String resultPage(HttpServletRequest request) {
   int num1 = Integer.parseInt(request.getParameter("num1"));
   int num2 = Integer.parseInt(request.getParameter("num2"));
   int result = num1 + num2; // example operation
   System.out.println("Result: " + result);
   return "result.jsp";
}
```

- **Explanation:**
  - @RequestMapping("/calculate") → maps this method to the /calculate URL.
  - Method parameter HttpServletRequest request → Spring injects the request object automatically.
  - request.getParameter("num1") → retrieves value from the form input with name num1.
  - Input values are returned as strings → convert to integers using Integer.parseInt().
  - Result can be processed and printed to console or passed to JSP page.

Title: DispatcherServlet Handling in Spring Boot

- In traditional servlets, URL mapping required web.xml configuration.

- In Spring Boot:
  - **DispatcherServlet** automatically maps incoming requests to the appropriate controller method based on annotations.
  - Example flow:
    1. Client sends request to /calculate.
    2. DispatcherServlet finds controller method annotated with @RequestMapping("/calculate").
    3. Controller method executes logic and returns a view (e.g., result.jsp).

Title: Accepting Form Input in Spring Boot

- Two ways to retrieve data from a client:
  1. **Servlet way:** Use HttpServletRequest.
  2. **Spring way:** Use @RequestParam (not covered yet in this transcript, will be discussed later).
- **Servlet approach example:**

```
int num1 = Integer.parseInt(request.getParameter("num1"));
int num2 = Integer.parseInt(request.getParameter("num2"));
```

- All client data resides in the request object.

Title: Dynamic Form Action Based on Operation

- **Form setup in JSP (index.jsp):**
  - Inputs for num1 and num2.
  - Dropdown for operation (add, subtract, multiply, divide).
  - Submit button sends request to a URL dynamically based on selected operation.

```
<form id="calcForm" method="get">
  ...
</form>

<script>
  const operationSelect = document.getElementById('operation');
  const form = document.getElementById('calcForm');

  // Set initial action
  form.action = '/' + operationSelect.value;
```

```
    // Update action on change
    operationSelect.addEventListener('change', function() {
        form.action = '/' + this.value;
    });
</script>
```

- This allows the same form to submit to multiple URLs depending on user selection.

Title: Result Page (result.jsp)

- A simple JSP page to display output from server-side computation:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Calculation Result</title>
</head>
<body>
    <h1>Result Page</h1>
    <p>Result is: <%= request.getAttribute("result") %></p>
</body>
</html>
```

- Currently, the result is printed to the **console**, but can be passed to JSP via:

```
request.setAttribute("result", result);
```

Title: Key Concepts Illustrated

- **Controller Method Mapping:** @RequestMapping("/path") maps a URL to a controller method.
- **Multiple Requests in One Controller:** Multiple methods can coexist logically in a single controller.
- **Handling Client Input:** Use HttpServletRequest to get form parameters in a servlet-style approach.
- **DispatcherServlet:** Manages request-to-controller mapping

automatically.

- **Dynamic Form Submission:** JavaScript can modify the form action URL based on user input.

✅ **Summary**

- Spring Boot controllers can handle multiple URLs within the same class.
- HttpServletRequest allows retrieving form data sent from the client.
- DispatcherServlet automatically routes requests to controller methods.
- Results can be printed on the console or sent to JSP pages using request.setAttribute().
- Form actions can dynamically change using JavaScript to support multiple operations.

Title: Mapping Requests to Controller Methods in Spring

- **Spring MVC Controllers** allow multiple request-handling methods in a single class.
  - Example: A HomeController can handle /home, /add, /subtract, /multiply, /divide.
- **Logical grouping of controllers:**
  - UserController → operations on users
  - ProductController → operations on products
  - OrderController → operations on orders
- Use annotations to map URLs to methods:
  - @RequestMapping("/path") → maps any HTTP method (GET, POST, etc.)
  - @GetMapping("/path") → maps only GET requests
  - @PostMapping("/path") → maps only POST requests

Example:

```
@RequestMapping("/")
public String homePage() {
    return "index.jsp";
}
```

Title: Handling Form Submissions

- Form submissions from JSP can be handled by Spring controllers.
- Form inputs are retrieved via HttpServletRequest (servlet-style approach):

```
@RequestMapping("/add")
public String resultPage(HttpServletRequest request,
HttpSession session) {
    int num1 = Integer.parseInt(request.getParameter("num1"));
    int num2 = Integer.parseInt(request.getParameter("num2"));
    int result = num1 + num2;

    // Save data in session to display in result.jsp
    session.setAttribute("operation", "Addition");
    session.setAttribute("result", result);
    session.setAttribute("error", null);

    return "result.jsp";
}
```

- **Flow:**
    1. User enters numbers in index.jsp.
    2. User selects operation (add, subtract, multiply, divide).
    3. Form submission triggers the mapped controller method.
    4. Method processes data and stores results in the HttpSession.
    5. Result page (result.jsp) retrieves the data from the session.

Title: Passing Data Between JSP Pages

- **Using Session:**
    - HttpSession is used to maintain data across multiple pages.
    - Set attributes in controller:

```
session.setAttribute("result", result);
session.setAttribute("operation", "Addition");
```

- Retrieve attributes in JSP:

```
<%= session.getAttribute("result") %>
```

- **Using JSTL (JavaServer Pages Standard Tag Library):**
  - Provides a cleaner syntax than scriptlets:

```
<c:if test="${not empty error}">
    <p style="color: red;"><c:out value="${error}" /></p>
</c:if>
<c:if test="${empty error}">
    <p><c:out value="${num1}" /> <c:out value="${operation}" />
<c:out value="${num2}" /> = <c:out value="${result}" /></p>
</c:if>
```

- JSTL automatically looks for attributes in session and request.

Title: Dynamic Form Actions Based on Operation

- In index.jsp, the form action is dynamically set based on the selected operation:

```
<form id="calcForm" method="get">
    ...
</form>

<script>
    const operationSelect =
document.getElementById('operation');
    const form = document.getElementById('calcForm');

    // Set initial action
    form.action = '/' + operationSelect.value;

    // Update action on change
    operationSelect.addEventListener('change', function() {
        form.action = '/' + this.value;
    });
</script>
```

- Allows a single form to submit to multiple URLs: /add, /subtract, /multiply, /divide.

Title: Controller Methods for Calculator Operations

- **Addition:**

```
@RequestMapping("/add")
public String add(HttpServletRequest request, HttpSession
session) {
    int num1 = Integer.parseInt(request.getParameter("num1"));
    int num2 = Integer.parseInt(request.getParameter("num2"));
    int result = num1 + num2;
    session.setAttribute("num1", num1);
    session.setAttribute("num2", num2);
    session.setAttribute("operation", "Addition");
    session.setAttribute("result", result);
    return "result.jsp";
}
```

- **Subtraction:**

```
@GetMapping("/subtract")
public String subtract(HttpServletRequest request, HttpSession session) { ... }
```

- **Multiplication:**

```
@GetMapping("/multiply")
public String multiply(HttpServletRequest request, HttpSession session) { ... }
```

- **Division with error handling:**

```
@GetMapping("/divide")
public String divide(HttpServletRequest request, HttpSession
session) {
    int num1 = Integer.parseInt(request.getParameter("num1"));
    int num2 = Integer.parseInt(request.getParameter("num2"));

    if (num2 == 0) {
        session.setAttribute("error", "Division by zero is not
allowed.");
        session.setAttribute("result", null);
    } else {
        double result = (double) num1 / num2;
        session.setAttribute("num1", num1);
        session.setAttribute("num2", num2);
        session.setAttribute("operation", "Division");
        session.setAttribute("result", result);
        session.setAttribute("error", null);
```

```
    }
    return "result.jsp";
}
```

Title: JSP Pages for Calculator

- **index.jsp** – displays form and allows user input:
  - Uses HTML form fields for num1, num2, and operation selection.
  - Includes JavaScript to dynamically update form action.
- **result.jsp** – displays calculation results:
  - Can use scriptlets:

```
<%= session.getAttribute("result") %>
```

- Or JSTL (recommended):

```
<c:out value="${result}" />
```

- Displays errors if any (e.g., division by zero).

Title: Required Dependencies for JSP and JSTL Support

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webmvc</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>11.0.15</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>jakarta.servlet.jsp.jstl</groupId>
    <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
    <version>3.0.0</version>
</dependency>
<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>jakarta.servlet.jsp.jstl</artifactId>
```

```
    <version>3.0.0</version>
</dependency>
```

- **Spring Boot Starter WebMVC** → provides controller and DispatcherServlet support.
- **Tomcat Jasper** → enables JSP compilation.
- **JSTL API + Implementation** → enables tag-based JSP syntax.

Title: Key Concepts Illustrated

- **DispatcherServlet:** automatically maps URLs to controller methods.
- **Session Management:** HttpSession maintains state between JSP pages.
- **Servlet-style Request Handling:** HttpServletRequest retrieves client parameters.
- **JSP Integration:** session attributes or JSTL tags are used to render dynamic content.
- **Dynamic Form Submission:** JavaScript allows one form to submit to multiple URLs based on operation type.

Title: Simplifying Spring Controllers with @RequestParam

- **Problem with the servlet-style approach:**
    - Previously, we used HttpServletRequest to manually extract query parameters:

```
int num1 = Integer.parseInt(request.getParameter("num1"));
int num2 = Integer.parseInt(request.getParameter("num2"));
```

    - Similarly, HttpSession was used to store results for the JSP page.
    - This approach is verbose and less readable.

Title: Direct Parameter Binding in Spring

- Spring allows you to bind query parameters directly to method arguments using **@RequestParam**.
- Instead of using HttpServletRequest, you can declare parameters directly in the method:

```
@RequestMapping("/add")
public String resultPage(@RequestParam("num1") int val1,
                         @RequestParam("num2") int val2,
                         HttpSession session) {
    int result = val1 + val2;
    session.setAttribute("operation", "Addition");
    session.setAttribute("result", result);
    session.setAttribute("error", null);
    return "result.jsp";
}
```

- **How it works:**
  - Query parameters from the URL are automatically mapped to method arguments.
  - Example URL:

```
http://localhost:8080/add?num1=7&num2=8
```

  - num1 → val1, num2 → val2

- **Benefits:**
  - Eliminates manual parsing with Integer.parseInt().
  - Reduces boilerplate code.
  - Improves readability and maintainability.

———

Title: Handling Different Variable Names

- By default, Spring maps query parameter names to method argument names.
- If you want the method variable name to differ from the query parameter, use **@RequestParam("paramName")**:

```
@RequestParam("num1") int val1
```

- Ensures correct mapping even if method variable names differ from the

URL parameters.

- **Important:** If the names don't match and @RequestParam is not used, Spring will throw a **500 Internal Server Error**.

Title: Key Notes on Query Parameter Mapping

1. **Direct Mapping** simplifies controller methods:
   - No need for HttpServletRequest.getParameter().
   - Automatically converts query parameters to appropriate types (int, double, String, etc.).
2. **Optional Parameters:**
   - By default, @RequestParam is **required**.
   - Can be made optional:

```
@RequestParam(value = "num1", required = false, defaultValue = "0") int val1
```

3. **Session Handling:**
   - Session can still be used for storing results, but later can be simplified further using Spring's Model or RedirectAttributes.

Title: Updated Controller Example Using @RequestParam

```
@RequestMapping("/add")
public String resultPage(@RequestParam("num1") int val1,
                         @RequestParam("num2") int val2,
                         HttpSession session) {
    int result = val1 + val2;
    session.setAttribute("operation", "Addition");
    session.setAttribute("result", result);
    session.setAttribute("error", null); // Clear previous errors
    return "result.jsp";
}
```

- Method now directly receives num1 and num2 from the URL.
- Reduces boilerplate, improves clarity, and keeps session logic intact.

✅ **Key Takeaways:**

- Use **@RequestParam** for query parameters instead of

HttpServletRequest.

- Allows **type-safe, clean, and readable controllers**.
- Ensures **correct mapping even with variable name differences**.
- Lays the foundation to further simplify code by potentially removing HttpSession in favor of Spring's Model.

Title: Replacing HttpSession with Model in Spring MVC

- Previously, data was transferred from the controller to the JSP using **HttpSession**.
- In **Spring MVC**, data transfer between the **Controller** and **View** should be handled using the **Model**.
- MVC recap:
  - **Controller**: Accepts user requests.
  - **View**: Sends the response to the client (JSP in this case).
  - **Model**: Transfers data between the controller and the view.
- Instead of using:

```
session.setAttribute("result", result);
```

Spring provides a **Model object**.

Title: Using the Model Object to Transfer Data

- The **Model** is an interface introduced in **Spring 2.5**.
- It is used to pass data from the controller to the view.
- The Model object is injected directly into the controller method.
- Key method:
  - addAttribute() is used instead of setAttribute().
  - Multiple attributes can be added to the model.
- Example usage:

```
model.addAttribute("result", result);
```

- Once added, these attributes are accessible in the JSP view.

Title: Controller Refactoring – Removing HttpSession

- The HttpSession object is completely removed.
- The controller now relies only on:
  - @RequestParam for input
  - Model for output data
- Updated controller method:

```
@RequestMapping("add")
public String resultPage(@RequestParam("num1") int val1,
@RequestParam("num2") int val2, Model model) {;
    int result = val1 + val2;
    model.addAttribute("num1", val1);
    model.addAttribute("num2", val2);
    model.addAttribute("operation", "Addition");
    model.addAttribute("result", result);
    model.addAttribute("error", null); // Clear any previous
error
    return "result"; // removing .jsp extension and moving the
view pages to views directory
}
```

- Outcome:
  - HttpServletRequest is removed.
  - HttpSession is removed.
  - Data is passed cleanly using the **Model**.

———

Title: Verifying Model-Based Data Transfer

- After restarting the application:
  - Values are entered in the browser.
  - The result is displayed correctly.
- This confirms that:
  - The **Model object successfully replaces HttpSession**.
  - Data transfer between controller and JSP works as expected.

———

Title: Removing .jsp Extension from Controller Return Values

- Previously, controllers returned view names like:

return "result.jsp";

- This tightly couples controllers to JSP technology.
- Spring MVC supports multiple view technologies:
  - **JSP**
  - **Freemarker**
  - **Thymeleaf**
- Including .jsp in controllers makes future view changes harder.

Title: Moving View Files and Using Logical View Names

- View files are moved to a dedicated **views directory**.
- Controllers now return **logical view names** instead of file names.
- Example:

return "result";

- Benefits:
  - No hardcoded view extensions.
  - Easier to switch view technologies.
  - Cleaner and more maintainable controller code.

Title: Result of View Changes and Observed Issue

- After:
  - Moving JSP files
  - Removing .jsp extensions
- The application fails to locate the home page.
- Reason:
  - Views are now in a different folder.
  - The extension is no longer specified.
- This introduces a **view resolution problem**.

Title: Next Step – View Resolution Configuration

- The issue of:
  - Missing home page
  - Views not being resolved correctly

- Will be addressed in the **next video**.
- The solution will involve configuring how Spring maps logical view names to actual view files.

Key Takeaways

- **Model** replaces **HttpSession** for controller-to-view data transfer.
- addAttribute() is used to pass multiple values.
- Controllers are now:
  - Cleaner
  - Framework-aligned
  - Less tightly coupled
- Removing .jsp from return statements improves flexibility.
- View resolution must be configured when changing view locations or naming conventions.

Title: Handling Static Resources (CSS) in Spring MVC

- CSS files are **static resources** used only for styling.
- Static resources **should not be moved with JSP files**.
- JSP files represent **views**, while CSS files are **public static content**.

Key Points:

- JSP files were moved to a different folder (views).
- CSS was also moved accidentally, causing styling issues.
- Solution:
  - Keep CSS files in one of the following locations:
    - **webapp directory**
    - **resources/static directory**

Static Folder Usage:

- Spring Boot provides a default **static folder** under resources.
- Any static files such as:
  - CSS
  - Images

- JavaScript

can be placed here.

Benefit:

- Clean project structure:
  - views → JSP files
  - static → CSS, images, JS

Title: Model Object – Passing Data Between Controller and View

- The **Model** object is used to transfer data from:
  - Controller → JSP (View)
- Model contains **only data**, not view information.

How It Works:

- Controller creates a Model object.
- Data is added using:

```
model.addAttribute("result", result);
```

- JSP fetches the data using the attribute name.

Key Characteristics:

- Used only for **data transfer**
- View name is returned separately as a String
- Example return:

```
return "result";
```

Title: Limitation of Using Model Alone

- When using Model:
  - Data and view name are handled **separately**
- Controller:
  - Adds data to Model
  - Returns a view name as a String

Drawback:

- Two responsibilities:

- Managing data
- Returning view name

Title: Introduction to ModelAndView

- **ModelAndView** combines:
  - Model (data)
  - View (view name)
- Allows returning **one single object** instead of two separate elements.

Purpose:

- Encapsulates:
  - Data attributes
  - View name
- Makes controller code more structured and meaningful.

Title: Using ModelAndView in Controller

Replacing Model:

- Instead of:

Model model

- Use:

ModelAndView modelAndView

Adding Data:

modelAndView.addObject("result", result);

Setting View Name:

modelAndView.setViewName("result");

Returning ModelAndView:

return modelAndView;

Title: Error Encountered Without Setting View Name

- After switching to ModelAndView:
  - Application returned **404 error**
- Reason:
  - View name was not set in ModelAndView

Fix:

- Explicitly set the view name:

```
modelAndView.setViewName("result");
```

Title: Final Working Flow with ModelAndView

- Controller:
  - Adds multiple attributes using addObject()
  - Sets the view name
  - Returns ModelAndView
- View Resolver:
  - Resolves logical view name
  - Maps it to the JSP file
- JSP:
  - Displays the result successfully

Title: Model vs ModelAndView – When to Use What

Use Model when:

- Only passing data
- Returning view name separately as a String

Use ModelAndView when:

- Passing data **and** view name together
- Want a single return object
- Cleaner MVC structure

Title: Final Controller Example Using ModelAndView

```
@RequestMapping("add")
public ModelAndView resultPage(
        @RequestParam("num1") long val1,
        @RequestParam("num2") long val2,
        ModelAndView modelAndView) {

    try {
        modelAndView.addObject("num1", val1);
        modelAndView.addObject("num2", val2);

        long result = val1 + val2;
        modelAndView.addObject("result", result);
        modelAndView.addObject("operation", "Addition");
        modelAndView.addObject("error", null);

        modelAndView.setViewName("result");

    } catch (Exception e) {
        modelAndView.addObject("error", "Invalid input: " + e.getMessage());
    }

    return modelAndView;
}
```

Key Takeaways

- Static resources should be placed in **static or webapp**, not inside views.
- **Model** → data only
- **ModelAndView** → data + view
- Always set the view name when using ModelAndView.
- Both approaches are valid; choice depends on design preference.

Title: Introduction to the Need for @ModelAttribute

- Until now, data transfer between **Controller** and **View** has been handled using:
  - Model
  - ModelAndView
- These approaches work well when handling **simple data** (e.g., two numbers).
- A limitation appears when dealing with **real-world entities** that contain multiple fields.

Title: Understanding the Use Case for Entities

- In real applications, data often represents **entities** such as:
  - Laptop
  - House
  - Office
  - Human roles (Student, Teacher, Engineer, Doctor)
- An entity represents a **real-world object** and is typically stored in a database.
- Example entity introduced: **Alien** (used as a conceptual example).

Title: Alien as an Entity (POJO)

- The **Alien** entity represents a programmer.
- It contains two fields:
  - aid (Alien ID – integer)
  - aname (Alien Name – string)

Alien Class Structure:

- Private fields
- Getters and setters
- toString() method for displaying object data

```
public class Alien {
    private int aid;
    private String aname;

    public int getAid() {
        return aid;
    }

    public void setAid(int aid) {
        this.aid = aid;
```

```
    }

    public String getAname() {
        return aname;
    }

    public void setAname(String aname) {
        this.aname = aname;
    }

    @Override
    public String toString() {
        return "Alien [aid=" + aid + ", aname=" + aname + "]";
    }
}
```

Title: Alien Input Form (JSP View)

- A new JSP page is created to collect Alien details.
- The form sends data to the server using a request path.
- Fields included:
  - Alien ID
  - Alien Name

```
<form action="addAlien" method="post">
    <label for="aid">Alien ID:</label>
    <input type="number" id="aid" name="aid" required>

    <label for="aname">Alien Name:</label>
    <input type="text" id="aname" name="aname" required>

    <button type="submit">Add Alien</button>
</form>
```

Title: Controller Handling Alien Requests Using @RequestParam

- A new controller mapping is created to handle Alien-related requests.
- Data is accepted **field by field** using @RequestParam.

```
@RequestMapping("addAlien")
```

```
public ModelAndView addAlien(@RequestParam("aid") int aid,
                             @RequestParam("aname") String
aname,
                             ModelAndView modelAndView) {

    Alien alien = new Alien();
    alien.setAid(aid);
    alien.setAname(aname);

    modelAndView.addObject("alien", alien);
    modelAndView.setViewName("alienresult");
    return modelAndView;
}
```

Title: Displaying Alien Data in the Result View

- The result JSP prints the Alien object.
- The object is displayed using Expression Language (EL).

```
<p>${alien}</p>
```

- Output is generated using the toString() method of the Alien class.

Title: Problem with Using Multiple @RequestParam

- Each field in the entity requires a separate @RequestParam.
- This approach becomes inefficient when:
  - The entity has many fields (e.g., 10 or more attributes).
- Controller method signatures become lengthy and harder to maintain.

Title: Motivation for @ModelAttribute

- Instead of accepting individual fields:
  - Accept the **entire entity object** directly.
- This allows Spring to:
  - Automatically bind request parameters to object fields.
- Reduces repetitive @RequestParam usage.
- Leads to cleaner and more scalable controller code.

Title: Summary – Why @ModelAttribute Is Needed

- @RequestParam works well for simple inputs.
- For entity-based data:
  - Accepting an object is more efficient.
- @ModelAttribute enables:
  - Automatic data binding
  - Cleaner controller methods
  - Better scalability for complex forms
- This sets the foundation for handling real-world objects in Spring MVC.

Using @ModelAttribute (Data Binding Made Easy)

Problem with @RequestParam

- Using multiple @RequestParam becomes messy when forms grow.
- You end up manually mapping request values to objects.

Solution: @ModelAttribute

- Spring **automatically creates the object** and **binds form data** to it.
- No need to manually instantiate or set values.

Controller Example

```
@RequestMapping("addAlien")
public String addAlien(@ModelAttribute Alien alien) {
    return "alienresult";
}
```

Key Points

- Spring creates the Alien object automatically.
- Form field names **must match** entity field names.
- The object is automatically added to the model.

Is @ModelAttribute Mandatory?

❌ No — it is **optional**.

```
public String addAlien(Alien alien) {
```

```
    return "alienresult";
}
```

- Works as long as attribute name matches (alien).

Using a Custom Attribute Name

public String addAlien(@ModelAttribute("newAlien") Alien alien)

- JSP must reference ${newAlien} instead of ${alien}.
- Required only when you want a **different name**.

@ModelAttribute at Method Level

Why Use It?

- To add **common or global data** to views.
- Useful for headers, labels, course names, user info, etc.

Example

```
@ModelAttribute("course")
public String courseName() {
    return "Java";
}
```

JSP Usage

```
<p>Welcome to the ${course} world</p>
```

Behavior

- Runs **before every request** in the controller.
- Automatically adds course to the model.
- Ideal for dynamic but shared values.

When to Use @ModelAttribute

✅ **Use it when:**

- Binding form data to an object
- Passing common data to multiple views
- You want clean, scalable controllers

❌ **Skip it when:**

- Attribute name matches object name
- You don’t need custom naming

Summary

- @ModelAttribute simplifies data binding
- Optional for method parameters
- Mandatory only for custom names
- Powerful at **method level** for shared data

This is why @ModelAttribute is widely used in Spring MVC 🚀