

# Spring MVC/ API Building

## Introduction to Spring MVC/ MVC Architecture

Spring MVC (Model-View-Controller) is a framework for building web applications. It separates the application into three interconnected components:

1. **Model:** Represents the application's data and business logic.
2. **View:** Displays the data (user interface).
3. **Controller:** Manages the application flow by handling user requests and responding appropriately.

### Key Features of Spring MVC:

- Clear separation of concerns.
- Support for RESTful APIs.
- Integration with other Spring modules.

## MVC Ki Simple Story

Imagine a **restaurant**:

- **Model (Chef):**

Data ka asli baap! Yeh chef kitchen mein kaam karta hai, khana banata hai, aur fresh dishes (data) ready karta hai. Bas aapko kya chahiye yeh bataana padta hai (input).

- **View (Waiter/Presentation):**

Chef ke haathon se bana khana (data) ko plate mein sajake customer (user) ke saamne laata hai. Yeh sirf dikhane ka kaam karta hai, khana (data) ke taste pe control nahi hai.

- **Controller (Manager):**

Manager aapki order (request) sunta hai aur usko chef (Model) aur waiter (View) ke beech set kar deta hai.

Manager decide karta hai ki kaunsi order kahan jayegi aur kaise wapas customer ko milegi.

## Spring MVC Ki Dukaan

### 1. User (Customer):

- Yeh banda browser kholta hai aur "Menu mein kya hai?" poochta hai.

### 2. DispatcherServlet (Receptionist):

- Aapki request pehle yahan aayegi. Yeh decide karega ki aapki request kahan forward karni hai.

### 3. Controller (Manager):

- Yeh aapki request Model (Chef) tak bhejta hai aur chef se output lekar waiter ko deta hai.

### 4. Model (Chef):

- Data ka kaam karta hai: database se uthaata hai, process karta hai, aur fresh output Controller ko deta hai.

### 5. View (Waiter):

- Yeh user ke liye data ko style karke dikhata hai (HTML, JSON, JSP, etc.).

### 6. Response (Khana):

- Aakhir mein user ko result milta hai, ekdum properly serve kiya hua.

## Role of Dispatcher Servlet

The `DispatcherServlet` acts as the front controller in Spring MVC. It routes incoming requests to appropriate controllers and resolves views.

### Flow:

1. Receives HTTP requests.
2. Delegates requests to handler mappings, controllers, and views.
3. Sends the HTTP response.

**Code Example** (web.xml configuration for DispatcherServlet):

```

<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

```

## Setting Up Spring MVC

1. **Dependencies:** Add the required dependencies to `pom.xml` for Maven.

```

<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.12</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

```

1. **Spring Configuration:** Create a `SpringConfig.java` class for bean definitions.

```

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.example")

```

```

public class SpringConfig {
    @Bean
    public InternalResourceViewResolver viewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}

```

1. **Controller:** Define controllers to handle requests.

## Controllers

Controllers handle the logic for processing user input and returning the appropriate view or data.

### Example:

```

@Controller
public class HomeController {
    @GetMapping("/")
    public String home() {
        return "index"; // Returns view (index.jsp)
    }
}

```

## Handling Requests

### @Controller and @RestController

- `@Controller`: Used for returning views (HTML, JSP).
- `@RestController`: A combination of `@Controller` and `@ResponseBody`, used for REST APIs returning JSON/XML.

### Example:

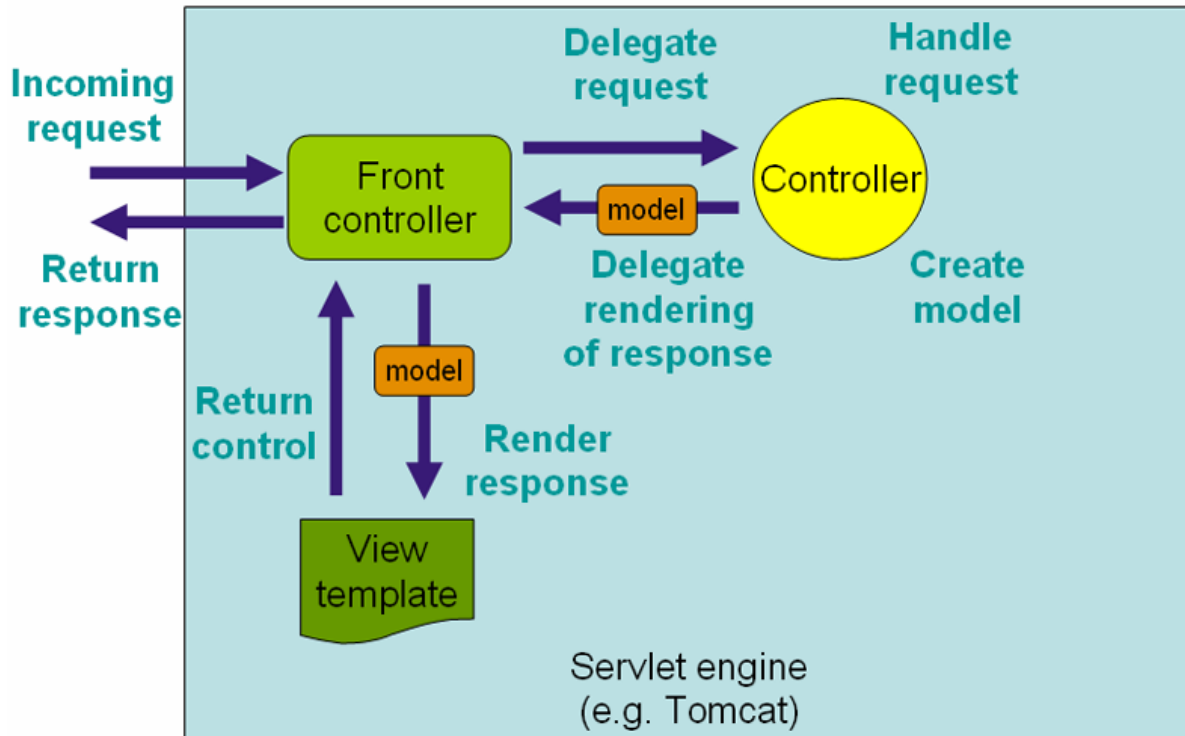
```
@RestController
@RequestMapping("/api")
public class ApiController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!"; // Returns raw data (JSON,
XML, etc.)
    }
}
```

## The MVC Design Pattern

- **MVC** is a design pattern that divides an application into three interconnected components:
  - **Model:** Represents the data and the business logic of the application. It interacts with the database and other data sources.
  - **View:** Represents the presentation layer. It's responsible for displaying the data provided by the model to the user.
  - **Controller:** Acts as an intermediary between the Model and the View. It handles user input, processes it, and updates the Model.

## Role of `DispatcherServlet` in Spring MVC

The `DispatcherServlet` is a crucial component in the Spring MVC framework, acting as the front controller in the MVC architecture. It is responsible for handling all incoming HTTP requests and dispatching them to the appropriate components within the Spring MVC framework.



## Step-by-Step Guide to Creating a Spring MVC Project with Manual Configuration

This guide will walk you through setting up a Spring MVC project from scratch using manual configurations (without Spring Boot). We'll start by setting up the project, configuring the necessary components, and creating a simple request handling mechanism.

### 1. Setting Up the Project

#### a. Create a Maven Project:

1. Open your IDE (like IntelliJ IDEA, Eclipse, or STS) and create a new Maven project.
2. Set the Group ID (e.g., `com.example`) and Artifact ID (e.g., `SpringMVCDemo`).
3. Choose the Maven project template if prompted.

#### b. Update the `pom.xml` File:

Add the necessary dependencies for Spring MVC:

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.
0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>SpringMVCDemo</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <!-- Spring MVC -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.3.13</version>
    </dependency>

    <!-- Servlet API -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>4.0.1</version>
      <scope>provided</scope>
    </dependency>

    <!-- JSP support -->
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>javax.servlet.jsp-api</artifactId>
      <version>2.3.3</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>

```

```

        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.3</version>
                <configuration>
                    <failOnMissingWebXml>>false</failOnMissi
ngWebXml>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>

```

- **Explanation:**

- We include `spring-webmvc` for Spring MVC.
- The `javax.servlet-api` and `javax.servlet.jsp-api` dependencies are added to support servlets and JSPs, respectively.

### c. Directory Structure:

Ensure your project directory structure looks like this:

```

bashCopy code
src/
├── main/
│   ├── java/
│   │   └── com/example/SpringMVCDemo/
│   │       ├── config/
│   │       └── controller/
│   ├── resources/
│   └── webapp/
│       ├── WEB-INF/
│       │   ├── web.xml
│       │   └── views/
│       │       └── hello.jsp

```



└─ test/

## 2. Configure `web.xml`

`web.xml` is the deployment descriptor for your application. Here we configure the `DispatcherServlet`.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                               http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
          version="3.1">

    <display-name>SpringMVCDemo</display-name>

    <!-- DispatcherServlet Configuration -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

```

<!-- Character Encoding Filter -->
<filter>
    <filter-name>encodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>

```

- **Explanation:**

- The `DispatcherServlet` is configured with the name `dispatcher`.
- The `contextConfigLocation` parameter points to the `spring-servlet.xml` file for Spring configuration.
- The URL pattern `/` maps all requests to the `DispatcherServlet`.
- A `CharacterEncodingFilter` is added to ensure UTF-8 encoding for all requests and responses.

### 3. Create Spring Configuration File: `spring-servlet.xml`

This file configures Spring MVC components such as view resolvers and component scanning.

```

<beans xmlns="http://www.springframework.org/schema/beans"

```

```

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instan
e"
        xmlns:context="http://www.springframework.org/schem
a/context"
        xmlns:mvc="http://www.springframework.org/schema/mv
c"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spri
ng-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/sp
ring-context.xsd
            http://www.springframework.org/schema/mvc
            http://www.springframework.org/schema/mvc/spring
-mvc.xsd">

        <!-- Enable annotation-driven configuration -->
        <mvc:annotation-driven/>

        <!-- Scan for components (controllers, services, etc.)
-->
        <context:component-scan base-package="com.example.Sprin
gMVCDemo"/>

        <!-- View Resolver to map view names to JSP files in th
e WEB-INF/views directory -->
        <bean class="org.springframework.web.servlet.view.Inter
nalResourceViewResolver">
            <property name="prefix" value="/WEB-INF/views/">
            <property name="suffix" value=".jsp"/>
        </bean>

    </beans>

```

- **Explanation:**

- `mvc:annotation-driven` enables Spring MVC annotations such as `@Controller` and `@RequestMapping`.
- `context:component-scan` tells Spring to scan the specified package for components (like controllers).
- `InternalResourceViewResolver` maps view names returned by controllers to JSP files in the `WEB-INF/views` directory.

## 4. Create a Simple Controller

Create a controller to handle requests and return a view.

### a. Create the Controller Class:

```
javaCopy code
package com.example.SpringMVCDemo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HelloController {

    @GetMapping("/hello")
    public String sayHello(@RequestParam(name="name", required=false, defaultValue="World") String name, Model model)
    {
        model.addAttribute("name", name);
        return "hello"; // Logical view name
    }
}
```

### • Explanation:

- The `HelloController` class is annotated with `@Controller`, making it a Spring MVC controller.

- The `@GetMapping("/hello")` annotation maps HTTP GET requests to `/hello` to this method.
- The method takes a `name` parameter from the request and adds it to the model, which is passed to the view.
- It returns `"hello"`, which is the logical name of the view.

## b. Create the View (JSP):

Create a JSP file `hello.jsp` under `src/main/webapp/WEB-INF/views/`.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix
="c" %>
<!DOCTYPE html>
<html>
<head>
    <title>Hello Page</title>
</head>
<body>
    <h1>Hello, ${name}!</h1>
</body>
</html>
```

### • Explanation:

- This JSP file displays a simple message using the `name` attribute passed from the controller.

## 5. Deploy and Run the Application

## Step-by-Step Guide to Creating a Spring MVC Project with Java Configuration

This guide will walk you through setting up a Spring MVC project from scratch using Java-based configuration, without any XML configuration.

# 1. Setting Up the Project

## a. Create a Maven Project:

1. Open your IDE (like IntelliJ IDEA, Eclipse, or STS) and create a new Maven project.
2. Set the Group ID (e.g., `com.example`) and Artifact ID (e.g., `SpringMVCJavaConfigDemo`).
3. Choose the Maven project template if prompted.

## b. Update the `pom.xml` File:

Add the necessary dependencies for Spring MVC:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>SpringMVCJavaConfigDemo</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <!-- Spring MVC -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.3.13</version>
    </dependency>

    <!-- Servlet API -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>4.0.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

```

        </dependency>

        <!-- JSP support -->
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>javax.servlet.jsp-api</artifactId>
            <version>2.3.3</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.3</version>
                <configuration>
                    <failOnMissingWebXml>>false</failOnMissi
ngWebXml>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>

```

- **Explanation:**

- We include `spring-webmvc` for Spring MVC.
- The `javax.servlet-api` and `javax.servlet.jsp-api` dependencies are added to support servlets and JSPs, respectively.

### c. Directory Structure:

Ensure your project directory structure looks like this:

```

bashCopy code
src/
├─ main/

```

```

|   |─ java/
|   |   └─ com/example/SpringMVCJavaConfigDemo/
|   |       └─ config/
|   |           └─ controller/
|   └─ resources/
|   └─ webapp/
|       └─ WEB-INF/
|           └─ views/
|               └─ hello.jsp
└─ test/

```

## 2. Java-Based Configuration

### a. Create the `WebInitializer` Class:

This class will replace the `web.xml` and configure the `DispatcherServlet`.

```

javaCopy code
package com.example.SpringMVCJavaConfigDemo.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null; // No root configuration in this case
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { WebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {

```



```

        return new String[] { "/" };
    }
}

```

- **Explanation:**

- `WebInitializer` extends `AbstractAnnotationConfigDispatcherServletInitializer`, which helps register the `DispatcherServlet`.
- `getRootConfigClasses()` is for root application context (e.g., services, repositories), which we don't use here.
- `getServletConfigClasses()` returns the configuration class (`WebConfig`), which will contain our Spring MVC configuration.
- `getServletMappings()` maps the `DispatcherServlet` to the root URL (`/`).

**b. Create the `WebConfig` Class:**

This class will replace `spring-servlet.xml` and configure Spring MVC.

```

javaCopy code
package com.example.SpringMVCJavaConfigDemo.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.example.SpringMVCJavaConfigDemo")
public class WebConfig implements WebMvcConfigurer {

```

```

    @Override
    public void configureViewResolvers(ViewResolverRegistry
registry) {
        registry.jsp("/WEB-INF/views/", ".jsp");
    }
}

```

- **Explanation:**

- `@Configuration` indicates that this class is a configuration class.
- `@EnableWebMvc` enables Spring MVC features like `@Controller` annotations.
- `@ComponentScan` scans the specified base package for Spring components (controllers, etc.).
- `WebMvcConfigurer` provides callback methods to customize the default configuration.
- `configureViewResolvers` configures the view resolver to map view names to JSP files in the `WEB-INF/views` directory.

### 3. Create a Simple Controller

Create a controller to handle requests and return a view.

#### a. Create the Controller Class:

```

javaCopy code
package com.example.SpringMVCJavaConfigDemo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HelloController {

    @GetMapping("/hello")
    public String sayHello(@RequestParam(name="name", requi

```

```

red=false, defaultValue="World") String name, Model model)
{
    model.addAttribute("name", name);
    return "hello"; // Logical view name
}
}

```

- **Explanation:**

- The `HelloController` class is annotated with `@Controller`, making it a Spring MVC controller.
- The `@GetMapping("/hello")` annotation maps HTTP GET requests to `/hello` to this method.
- The method takes a `name` parameter from the request and adds it to the model, which is passed to the view.
- It returns `"hello"`, which is the logical name of the view.

## b. Create the View (JSP):

Create a JSP file `hello.jsp` under `src/main/webapp/WEB-INF/views/`.

```

jspCopy code
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix
="c" %>
<!DOCTYPE html>
<html>
<head>
    <title>Hello Page</title>
</head>
<body>
    <h1>Hello, ${name}!</h1>
</body>
</html>

```

- **Explanation:**

- This JSP file displays a simple message using the `name` attribute passed from the controller.

## 4. Deploy and Run the Application

### a. Build the Project:

- Use Maven to build the project by running `mvn clean install`.

### b. Deploy the WAR file:

- Deploy the generated WAR file ( `SpringMVCJavaConfigDemo.war` ) to a servlet container like Apache Tomcat.

### c. Access the Application:

- Start your servlet container and navigate to `http://localhost:8080/SpringMVCJavaConfigDemo/hello?name=YourName` in your browser.
  - You should see a message like "Hello, YourName!" displayed on the page.
- 

## How Spring Boot Configure Spring MVC Automatically

Spring Boot automates the process of setting up a Spring MVC application by providing a set of conventions and auto-configurations that eliminate the need for most of the manual configurations. Here's how Spring Boot simplifies and automates the setup process:

### 1. No `web.xml` Required

- **Spring Boot Auto-Configuration:** Spring Boot automatically sets up the `DispatcherServlet` and maps it to the root URL ( `/` ) without requiring a `web.xml` file or a `WebInitializer` class.
- **Embedded Server:** Spring Boot includes an embedded servlet container (like Tomcat, Jetty, or Undertow), so you don't need to deploy the application to an external server. You can run your application as a simple Java application.

### 2. No XML Configuration Files

- **Java-Based Configuration by Default:** Spring Boot uses Java-based configuration by default. The `@SpringBootApplication` annotation is a convenience annotation that combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`.

```
@SpringBootApplication
public class SpringBootMvcApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootMvcApplication.class, args);
    }
}
```

- `@EnableAutoConfiguration`: This annotation tells Spring Boot to automatically configure your application based on the dependencies you have added. For example, if `spring-webmvc` is on the classpath, Spring Boot automatically configures Spring MVC.
- `@ComponentScan`: This annotation enables component scanning, which automatically detects and registers beans, including `@Controller` classes.

### 3. Auto-Configuration of Spring MVC Components

- **DispatcherServlet:** Spring Boot automatically registers a `DispatcherServlet` with the default URL mapping (`/`).
- **View Resolvers:** If you have JSPs, Thymeleaf, or other view templates in your project, Spring Boot automatically configures the necessary view resolvers.
- **Static Resources:** Spring Boot automatically serves static resources like HTML, CSS, and JavaScript from the `/static`, `/public`, `/resources`, or `/META-INF/resources` directories without additional configuration.

### 4. Embedded Server and Simplified Deployment

- **Run as a Standalone Application:** Spring Boot packages your application as an executable JAR with an embedded server, allowing you to run it directly with `java -jar`.

```
java -jar target/myapp.jar
```

- **No External Server Setup:** There's no need to set up an external server like Tomcat. Spring Boot handles that internally, making the deployment process much simpler.

## 5. Simplified Dependency Management

- **Starters:** Spring Boot provides "starter" dependencies that aggregate commonly used libraries, reducing the need to manually manage dependencies. For example, adding `spring-boot-starter-web` brings in all dependencies needed for a Spring MVC web application, including Spring MVC itself, an embedded Tomcat server, and Jackson for JSON processing.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

## 6. Integrated Development Features

- **Spring Boot DevTools:** Provides additional features to enhance the development experience, such as automatic restart, live reload, and configuration for faster development.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <scope>runtime</scope>
</dependency>
```

## 7. Automatic Configuration of Common Use Cases

- **Error Handling:** Spring Boot provides default error pages and JSON error responses for REST APIs.
- **Embedded Database Support:** If you include an embedded database dependency like H2, Spring Boot will automatically configure it.
- **Security:** If `spring-boot-starter-security` is added, Spring Boot configures basic authentication by default.

## 8. Production-Ready Features

- **Monitoring and Management:** Spring Boot provides production-ready features out of the box, such as health checks, metrics, and application monitoring via the `spring-boot-starter-actuator`.
- **Configuration Management:** Spring Boot supports various ways to configure your application, such as via `application.properties` or `application.yml` files, environment variables, and command-line arguments.

### 1. Create a Spring Boot Project:

You can use Spring Initializr (<https://start.spring.io>) to generate a Spring Boot project with the necessary dependencies:

- **Dependencies:** `Spring Web` (which includes Spring MVC).

### 2. The Main Application Class:

```
javaCopy code
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootMvcApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootMvcApplication.class, args);
    }
}
```

### 3. A Simple Controller:

```
javaCopy code
package com.example.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HelloController {

    @GetMapping("/hello")
    public String sayHello(@RequestParam(name="name", required=false, defaultValue="World") String name, Model model)
    {
        model.addAttribute("name", name);
        return "hello";
    }
}
```

### 4. Create the View (JSP or Thymeleaf):

- **JSP:** `src/main/webapp/WEB-INF/jsp/hello.jsp`

```
<html>
<body>
    <h1>Hello, ${name}!</h1>
</body>
</html>
```

### 5. Run the Application:

Simply run the `SpringBootMvcApplication` class as a Java application. The application will start on `http://localhost:8080`, and you can access the hello



endpoint at `http://localhost:8080/hello?name=YourName` .

## 1. `@Controller`

- **Purpose:**

- The `@Controller` annotation is used to define a controller in Spring MVC. It is a specialization of the `@Component` annotation and indicates that the class serves the role of a controller in the MVC pattern.
- The controller processes incoming web requests, prepares a model, and returns a view.

- **Example:**

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class MyController {

    @GetMapping("/welcome")
    public String welcome(Model model) {
        model.addAttribute("message", "Welcome to Spring MVC!");
        return "welcome"; // This refers to a view named "welcome"
    }
}
```

- Here, `MyController` is a Spring MVC controller, and the `welcome` method handles GET requests to `/welcome` .

## 2. `@RequestMapping`

- **Purpose:**

- The `@RequestMapping` annotation is used to map web requests to specific handler methods in a controller class.
- It can be applied at both the class and method levels to specify the URL patterns and HTTP methods.

- **Example:**

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

@Controller
@RequestMapping("/api")
public class ApiController {

    @RequestMapping(value = "/greet", method = RequestMethod.GET)
    public String greet() {
        return "greeting"; // This refers to a view named "greeting"
    }
}
```

- `@RequestMapping("/api")` at the class level defines a base URL for all methods in the class.
- `@RequestMapping(value = "/greet", method = RequestMethod.GET)` maps GET requests to `/api/greet`.

### 3. `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`

- **Purpose:**

- These are specialized variants of `@RequestMapping` for specific HTTP methods: GET, POST, PUT, DELETE. They simplify the declaration of request mappings.

- **Example:**

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    public String getUser(@PathVariable("id") Long id) {
        return "Fetching user with ID: " + id;
    }

    @PostMapping
    public String createUser() {
        return "Creating a new user";
    }
}
```

- `@GetMapping("/{id}")` maps GET requests to `/users/{id}`.
- `@PostMapping` maps POST requests to `/users`.

#### 4. `@PathVariable`

- **Purpose:**

- `@PathVariable` is used to extract values from the URI. It binds a method parameter to a URI template variable.

- **Example:**

```
import org.springframework.web.bind.annotation.GetMapping
```

```

g;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/orders")
public class OrderController {

    @GetMapping("/{orderId}")
    public String getOrder(@PathVariable("orderId") String orderId) {
        return "Order ID: " + orderId;
    }
}

```

- In this example, `@PathVariable("orderId")` binds the `orderId` parameter to the value of `{orderId}` in the URI.

## 5. `@RequestParam`

- **Purpose:**
  - `@RequestParam` is used to bind a method parameter to a query parameter in the request.
- **Example:**

```

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/search")

```

```
public class SearchController {

    @GetMapping
    public String search(@RequestParam("q") String query) {
        return "Searching for: " + query;
    }
}
```

- Here, `@RequestParam("q")` binds the query parameter `q` to the `query` method parameter.
- If you access `/search?q=Spring`, the method will return `Searching for: Spring`.

## 6. `@RequestBody`

- **Purpose:**

- `@RequestBody` is used to bind the HTTP request body to a method parameter. It is typically used in POST, PUT, or DELETE methods.

- **Example:**

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/products")
public class ProductController {

    @PostMapping
    public String addProduct(@RequestBody Product product) {
        return "Product added: " + product.getName();
    }
}
```

```
}  
}
```

- In this example, the JSON body of a POST request is bound to the `Product` object using `@RequestBody`.

## 7. `@ResponseBody`

- **Purpose:**

- `@ResponseBody` is used to indicate that the return value of a method should be bound to the web response body. It is commonly used in RESTful web services.

- **Example:**

```
javaCopy code  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.ResponseBody;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
@RequestMapping("/status")  
public class StatusController {  
  
    @GetMapping  
    @ResponseBody  
    public String status() {  
        return "Application is running";  
    }  
}
```

- `@ResponseBody` indicates that the method's return value will be written directly to the HTTP response body as plain text or JSON.

## 8. `@RestController`

- **Purpose:**

- `@RestController` is a convenience annotation that combines `@Controller` and `@ResponseBody`. It is used for building RESTful web services.

- **Example:**

```
javaCopy code
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class ApiRestController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello from REST API";
    }
}
```

- With `@RestController`, there's no need to annotate each method with `@ResponseBody`; it's implied for all methods in the class.

## 9. `@RequestHeader`

- **Purpose:**

- `@RequestHeader` is used to bind a method parameter to a request header value.

- **Example:**

```
javaCopy code
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
```

```

ader;
import org.springframework.web.bind.annotation.RestController
oller;

@RestController
@RequestMapping("/headers")
public class HeaderController {

    @GetMapping
    public String getHeader(@RequestHeader("User-Agent")
String userAgent) {
        return "User-Agent: " + userAgent;
    }
}

```

- In this example, `@RequestHeader("User-Agent")` binds the `User-Agent` header value to the `userAgent` method parameter.

## 10. `@ModelAttribute`

- **Purpose:**

- `@ModelAttribute` is used to bind a method parameter or method return value to a named model attribute and then expose it to a web view.

- **Example:**

```

javaCopy code
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

@Controller
@RequestMapping("/user")

```



```

public class UserController {

    @GetMapping("/form")
    public String userForm(Model model) {
        model.addAttribute("user", new User());
        return "userForm";
    }

    @PostMapping("/save")
    public String saveUser(@ModelAttribute User user) {
        // Save user to database
        return "redirect:/user/success";
    }
}

```

- In this example, `@ModelAttribute` binds the form data to a `User` object and adds it to the model.

## 11. `@SessionAttributes`

- **Purpose:**

- `@SessionAttributes` is used to store model attributes in the session. This is typically used to maintain the state of form-backing objects across multiple requests.

- **Example:**

```

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;

@Controller

```

```

@RequestMapping("/checkout")
@SessionAttributes("cart")
public class CheckoutController {

    @ModelAttribute("cart")
    public Cart setupCart() {
        return new Cart();
    }

    @RequestMapping("/review")
    public String reviewCart(@ModelAttribute("cart") Cart cart, Model model) {
        // Process cart
        return "reviewCart";
    }
}

```

- `@SessionAttributes("cart")` indicates that the `cart` attribute should be stored in the session and remain available across multiple requests.

## 12. `@ExceptionHandler`

- **Purpose:**

- `@ExceptionHandler` is used to define a method that handles exceptions thrown by request-handling methods in the same controller or across the application if used in a `@ControllerAdvice`.

- **Example:**

```

import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.http.HttpStatus;

```

```

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String handleResourceNotFoundException() {
        return "error/404";
    }
}

```

- In this example, if a `ResourceNotFoundException` is thrown, the method returns a view called `error/404` and sets the HTTP status to `404`.

## Data Binding and Validation

For validation add `validation` dependency .

Data binding is the process of binding the incoming JSON (or other formats) data from HTTP requests to Java objects. This is typically done using the `@RequestBody` annotation.

### Model Class with Jakarta Validation Annotations

Here's how to define a model class using `jakarta.validation` annotations:

```

package com.example.demo.model;

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.Size;

public class User {

    @NotEmpty(message = "Name is required")
    @Size(min = 2, max = 30, message = "Name must be between 2 and 30 characters")
    private String name;

    @NotEmpty(message = "Email is required")

```

```

    @Email(message = "Email should be valid")
    private String email;

    @NotEmpty(message = "Password is required")
    @Size(min = 8, message = "Password must be at least 8 c
haracters")
    private String password;

    // Getters and Setters
    public String getName() {
        return name;
    }

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}

```

- **Explanation:**

- The model class `User` uses annotations from `jakarta.validation.constraints` for data validation.

- The fields are validated to ensure that they meet specific constraints, such as not being empty, having a valid email format, and adhering to specific size limits.

## Controller Class with Validation

Here's how to create a controller that uses Jakarta Validation to validate incoming data:

```
javaCopy code
package com.example.demo.controller;

import com.example.demo.model.User;
import jakarta.validation.Valid;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;

@RestController
public class UserController {

    @PostMapping("/users")
    public ResponseEntity<Object> createUser(@Valid @RequestBody User user, BindingResult result) {
        if (result.hasErrors()) {
            Map<String, String> errors = new HashMap<>();
            result.getFieldErrors().forEach(error -> errors.put(error.getField(), error.getDefaultMessage()));
            return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
        }
    }
}
```

```

        // Logic to save the user to the database

        return new ResponseEntity<>("User created successfully", HttpStatus.OK);
    }
}

```

- **Explanation:**

- `@Valid` is from `jakarta.validation` and is used to trigger validation on the `User` object.
- `@RequestBody` binds the incoming JSON data to the `User` object.
- If validation fails, the errors are captured and returned as a `BAD_REQUEST` response.

## Global Exception Handling with Jakarta Validation

Create a global exception handler that will handle validation errors and other exceptions using Jakarta Validation:

```

package com.example.demo.exception;

import jakarta.validation.ConstraintViolationException;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;

import java.util.HashMap;
import java.util.Map;

```

```

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ResponseEntity<Map<String, String>> handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });
        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(ConstraintViolationException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ResponseEntity<Map<String, String>> handleConstraintViolationExceptions(ConstraintViolationException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getConstraintViolations().forEach(violation -> {
            String fieldName = violation.getPropertyPath().toString();
            String errorMessage = violation.getMessage();
            errors.put(fieldName, errorMessage);
        });
        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }

    // Additional exception handlers can be added here for

```

```
other exception types
}
```

- **Explanation:**

- The global exception handler now uses `MethodArgumentNotValidException` to handle validation errors when using `@Valid`.
- `ConstraintViolationException` handles cases where individual constraint violations are caught (such as when validating a single field).

## Custom Validator with Jakarta Validation

Here's how to create and use a custom validator with Jakarta Validation:

### a. Define the Custom Annotation

```
javaCopy code
package com.example.demo.validation;

import jakarta.validation.Constraint;
import jakarta.validation.Payload;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PasswordValidator.class)
public @interface ValidPassword {

    String message() default "Password must contain at least one uppercase letter, one lowercase letter, and one digit";

    Class<?>[] groups() default {};
```



```

        Class<? extends Payload>[] payload() default {};
    }

```

## b. Create the Validator Implementation

```

package com.example.demo.validation;

import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class PasswordValidator implements ConstraintValidator<ValidPassword, String> {

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if (value == null) {
            return false;
        }
        return value.matches("^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d).+$");
    }
}

```

```

package com.example.demo.validation;

import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class PasswordValidator implements ConstraintValidator<ValidPassword, String> {

    private String message;

    @Override

```

```

    public void initialize(ValidPassword constraintAnnotation) {
        // Access the message from the annotation
        this.message = constraintAnnotation.message();
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if (value == null || !value.matches("^(?=.*[a-z])(?=.*[A-Z]).{8,}$")) {
            // Customize the error message in the validation context
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate(message).addConstraintViolation();
            return false;
        }
        return true;
    }
}

```

### c. Apply the Custom Validator

```

javaCopy code
package com.example.demo.model;

import com.example.demo.validation.ValidPassword;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.Size;

public class User {

    @NotEmpty(message = "Name is required")
    @Size(min = 2, max = 30, message = "Name must be between 2 and 30 characters")
    private String name;

    @NotEmpty(message = "Email is required")
    @Email(message = "Email should be valid")
    private String email;
}

```

```

@ValidPassword
private String password;

// Getters and Setters
public String getName() {
    return name;
}

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

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}
}

```

## How to create custom image validator.

### Define the Custom Annotation

First, we'll create a custom annotation named `ValidImage`:

```

javaCopy code
package com.example.demo.validation;

```

```

import jakarta.validation.Constraint;
import jakarta.validation.Payload;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = ImageValidator.class)
public @interface ValidImage {

    String message() default "Invalid image file";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}

```

- **Explanation:**

- `@Target({ ElementType.FIELD, ElementType.PARAMETER })` : This annotation can be applied to fields or method parameters.
- `@Retention(RetentionPolicy.RUNTIME)` : The annotation will be available at runtime.
- `@Constraint(validatedBy = ImageValidator.class)` : Specifies that the validation logic is implemented in the `ImageValidator` class.
- `message` : The default error message that will be returned if validation fails.

## 2. Implement the `ConstraintValidator`

Next, we implement the `ConstraintValidator` to define the logic for validating an image file.

```

javaCopy code
package com.example.demo.validation;

import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;
import org.springframework.web.multipart.MultipartFile;

import java.util.Arrays;
import java.util.List;

public class ImageValidator implements ConstraintValidator<ValidImage, MultipartFile> {

    private static final List<String> ALLOWED_CONTENT_TYPES
= Arrays.asList("image/jpeg", "image/png", "image/gif");

    @Override
    public boolean isValid(MultipartFile value, ConstraintValidatorContext context) {
        if (value == null || value.isEmpty()) {
            return true; // Consider null or empty file as
valid. Handle required validation separately.
        }

        // Check if the file's content type is in the list
of allowed image content types
        return ALLOWED_CONTENT_TYPES.contains(value.getCont
entType());
    }
}

```

- **Explanation:**

- The `ImageValidator` implements `ConstraintValidator<ValidImage, MultipartFile>`, where `MultipartFile` is the type of the file being validated.
- `ALLOWED_CONTENT_TYPES` contains the MIME types that are considered valid images ( `image/jpeg` , `image/png` , `image/gif` ).

- The `isValid` method checks whether the uploaded file's content type matches one of the allowed types.

### 3. Apply the Custom Annotation to a Model Class

Now, you can use the `@ValidImage` annotation in your model class to validate image uploads.

```
javaCopy code
package com.example.demo.model;

import com.example.demo.validation.ValidImage;
import org.springframework.web.multipart.MultipartFile;

public class ImageUploadRequest {

    @ValidImage(message = "Please upload a valid image file
(JPEG, PNG, GIF)")
    private MultipartFile image;

    // Getters and Setters
    public MultipartFile getImage() {
        return image;
    }

    public void setImage(MultipartFile image) {
        this.image = image;
    }
}
```

- **Explanation:**

- `@ValidImage` is applied to the `MultipartFile` field to validate that the uploaded file is a valid image.
- The `message` attribute allows customization of the error message.

### 4. Create a Controller to Handle Image Uploads

Here's an example of a controller that handles image uploads and uses the `ImageUploadRequest` model:

```
javaCopy code
package com.example.demo.controller;

import com.example.demo.model.ImageUploadRequest;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.multipart.MultipartFile;

import jakarta.validation.Valid;
import java.util.HashMap;
import java.util.Map;

@RestController
public class ImageUploadController {

    @PostMapping("/uploadImage")
    public ResponseEntity<Object> uploadImage(@Valid @ModelAttribute ImageUploadRequest request, BindingResult result)
    {
        if (result.hasErrors()) {
            Map<String, String> errors = new HashMap<>();
            result.getFieldErrors().forEach(error -> errors.put(error.getField(), error.getDefaultMessage()));
            return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
        }

        // Process the valid image file (e.g., save to disk
```

```
or database)
    MultipartFile image = request.getImage();

    return new ResponseEntity<>("Image uploaded success
fully", HttpStatus.OK);
}
}
```

## Interceptors in Spring MVC

Interceptors in Spring MVC are used to perform operations before and after the execution of a controller action. They can be used for various cross-cutting concerns like logging, authentication, and request processing. Interceptors in Spring MVC are similar to filters in the servlet specification but are more powerful and flexible.

### 1. What is a Spring MVC Interceptor?

A Spring MVC Interceptor is a component that allows you to intercept HTTP requests before they reach the controller and/or after the controller has processed them but before the response is sent to the client. Interceptors can also intercept the execution of view rendering.

Interceptors in Spring MVC are typically implemented by creating a class that implements the `HandlerInterceptor` interface.

### 2. Key Methods in `HandlerInterceptor`

The `HandlerInterceptor` interface has three key methods:

1. `preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) :`
  - Called before the actual controller method is invoked.
  - Returns `true` to continue the execution chain, or `false` to stop it.
2. `postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) :`
  - Called after the controller method has been invoked but before the view is rendered.



- Allows modification of the `ModelAndView` object or adding additional data to the model.
3. `afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)`:
- Called after the view has been rendered and the request is complete.
  - Useful for resource cleanup or logging after request processing is finished.

### 3. Creating an Interceptor

Let's create a simple logging interceptor that logs request details before and after a controller is executed.

#### a. Implement the Interceptor

```
package com.example.demo.interceptor;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

public class LoggingInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        System.out.println("Pre Handle method is Calling: " + request.getRequestURI());
        return true; // Continue with the next interceptor or the controller method
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView mode
```

```

    ModelAndView) throws Exception {
        System.out.println("Post Handle method is Calling:
" + request.getRequestURI());
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
    HttpServletResponse response, Object handler, Exception ex)
    throws Exception {
        System.out.println("Request and Response is complet
ed: " + request.getRequestURI());
    }
}

```

- **Explanation:**

- `preHandle` : Logs the request URI before the controller method is executed.
- `postHandle` : Logs after the controller method has executed but before the view is rendered.
- `afterCompletion` : Logs after the request has been fully processed and the response is completed.

## b. Register the Interceptor

You need to register the interceptor with Spring MVC so that it can be invoked for incoming requests. This is done by implementing `WebMvcConfigurer` and overriding the `addInterceptors` method.

```

javaCopy code
package com.example.demo.config;

import com.example.demo.interceptor.LoggingInterceptor;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.Web

```

```

bMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoggingInterceptor())
            .addPathPatterns("/**") // Intercept all requests
            .excludePathPatterns("/api/ignore/**"); // Exclude certain paths
    }
}

```

- **Explanation:**

- `addInterceptor(new LoggingInterceptor())` : Registers the `LoggingInterceptor` with the application.
- `addPathPatterns("/**")` : Specifies that the interceptor should apply to all incoming requests.
- `excludePathPatterns("/api/ignore/**")` : Excludes specific paths from being intercepted.

## 4. Common Use Cases for Interceptors

Interceptors are useful for implementing common cross-cutting concerns:

1. **Logging:** Log request and response data for debugging or auditing purposes.
2. **Authentication/Authorization:** Check if a user is authenticated or authorized to access a specific resource.
3. **Performance Monitoring:** Measure the time taken to process a request.
4. **Request Modification:** Modify the request or response before it reaches the controller or after it leaves the controller.
5. **Resource Management:** Open and close resources

how to register in latest spring boot using filterchain

ChatGPT

In the latest versions of Spring Boot (particularly with Spring Security and Spring Web), the registration of interceptors and filters has evolved, particularly with the introduction of `SecurityFilterChain` and the use of `Filter` for handling security-related tasks. However, if you want to register a custom interceptor in the context of a Spring Boot application, the approach remains straightforward.

## Registering an Interceptor with Spring Boot using `WebMvcConfigurer`

If you are looking to register an interceptor (not a security filter), you would typically do this using `WebMvcConfigurer`, as shown earlier. This approach remains valid even in the latest versions of Spring Boot.

## Registering Filters in Spring Boot with `SecurityFilterChain`

If you want to work with security filters or modify the security filter chain, you can use `SecurityFilterChain` introduced with Spring Security.

### 1. Registering an Interceptor Using `WebMvcConfigurer`

This approach is used to handle HTTP requests at the MVC level, allowing you to add custom interceptors:

1. **Create the Interceptor:** (same as before)
2. **Register the Interceptor:**
  - In the latest Spring Boot, you still use `WebMvcConfigurer` to register your interceptors.

```
javaCopy code
package com.example.demo.config;

import com.example.demo.interceptor.LoggingInterceptor;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.Web
```

```

WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoggingInterceptor())
            .addPathPatterns("/**")
            .excludePathPatterns("/api/ignore/**");
    }
}

```

This approach is still valid in the latest Spring Boot versions.

## 2. Registering a Custom Filter Using `SecurityFilterChain`

If you are specifically dealing with security filters (e.g., for authentication or authorization), Spring Security uses the `SecurityFilterChain` configuration.

### 1. Create a Custom Filter:

- A custom filter can be created by implementing the `Filter` interface.

```

// javaCopy code
package com.example.demo.filter;

import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;

```

```

public class CustomSecurityFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // Initialization logic if needed
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        // Custom security logic here (e.g., logging, authentication checks)
        System.out.println("Custom Security Filter is applied on URI: " + httpRequest.getRequestURI());

        // Continue the filter chain
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {
        // Cleanup logic if needed
    }
}

```

## 1. Register the Custom Filter Using `SecurityFilterChain`:

- The `SecurityFilterChain` bean allows you to customize the security filter chain.

```

javaCopy code
package com.example.demo.config;

import com.example.demo.filter.CustomSecurityFilter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            // Add your custom filter before/after specific filters
            .addFilterBefore(new CustomSecurityFilter(), UsernamePasswordAuthenticationFilter.class)
            // Add your security configurations
            .authorizeHttpRequests(authorizeRequests ->
                authorizeRequests
                    .requestMatchers("/public/**").permitAll()
                    .anyRequest().authenticated()
            )
            .formLogin()
            .and()
            .httpBasic();

        return http.build();
    }
}

```

```
}  
}
```

- **Explanation:**

- `SecurityFilterChain` configures the security filter chain for your application.
- `addFilterBefore` or `addFilterAfter` allows you to insert your custom filter into the security filter chain at the desired position.
- In this example, the `CustomSecurityFilter` is added before the `UsernamePasswordAuthenticationFilter`.

## File Upload

File upload in Spring Boot can be handled easily using Spring MVC's `MultipartFile` interface. This allows you to upload files from clients to your server, where you can process or store them.

### 1. Setting Up a Spring Boot Project

Ensure your Spring Boot project includes the necessary dependencies. The `spring-boot-starter-web` dependency is usually sufficient because it includes support for file uploads.

```
<dependencies>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
  </dependency>  
</dependencies>
```

### 2. Configuring File Upload Settings



By default, Spring Boot sets up multipart support with sensible defaults. However, you can customize these settings in the `application.properties` or `application.yml` file.

**Example in `application.properties` :**

```
spring.servlet.multipart.max-file-size=2MB
spring.servlet.multipart.max-request-size=2MB
```

- `spring.servlet.multipart.max-file-size` : The maximum size for a single file.
- `spring.servlet.multipart.max-request-size` : The maximum size for the entire multipart request.

### 3. Creating a File Upload Controller

Let's create a controller that handles file uploads.

```
package com.example.demo.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

import java.io.File;
import java.io.IOException;

@RestController
public class FileUploadController {

    private static final String UPLOAD_DIR = "uploads/";
```

```

    @PostMapping("/upload")
    public ResponseEntity<String> uploadFile(@RequestParam
("file") MultipartFile file) {
        if (file.isEmpty()) {
            return new ResponseEntity<>("Please select a fi
le to upload.", HttpStatus.BAD_REQUEST);
        }

        try {
            // Get the file and save it to the local filesy
stem

            String fileName = file.getOriginalFilename();
            File destinationFile = new File(UPLOAD_DIR + fi
leName);

            file.transferTo(destinationFile);

            return new ResponseEntity<>("File uploaded succ
essfully: " + fileName, HttpStatus.OK);
        } catch (IOException e) {
            e.printStackTrace();
            return new ResponseEntity<>("Failed to upload t
he file: " + e.getMessage(), HttpStatus.INTERNAL_SERVER_ERR
OR);
        }
    }
}

```

- **Explanation:**

- `@RequestParam("file") MultipartFile file` : Binds the uploaded file to a `MultipartFile` object.
- `file.isEmpty()` : Checks if a file was uploaded.
- `file.transferTo(destinationFile)` : Saves the uploaded file to the specified location on the server.

## 4. Creating the Upload Directory

Ensure that the directory where you want to save uploaded files exists. In this example, the directory is `uploads/` in the root of your project. You can create it manually or programmatically.

```
import javax.annotation.PostConstruct;

@PostConstruct
public void init() {
    File uploadDir = new File(UPLOAD_DIR);
    if (!uploadDir.exists()) {
        uploadDir.mkdir();
    }
}
```

```
package com.example.demo.config;

import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;
import java.io.File;

@Configuration
public class FileUploadConfig {

    private static final String UPLOAD_DIR = "uploads/";

    @PostConstruct
    public void init() {
        File uploadDir = new File(UPLOAD_DIR);
        if (!uploadDir.exists()) {
            boolean created = uploadDir.mkdir();
            if (created) {
                System.out.println("Upload directory created:");
            } else {
                System.err.println("Failed to create upload d
```

```

        }
    }
}

public static String getUploadDir() {
    return UPLOAD_DIR;
}
}

```

To handle multiple file uploads in a single request, modify the controller method as follows:

```

javaCopy code
@PostMapping("/uploadMultiple")
public ResponseEntity<String> uploadMultipleFiles(@RequestP
aram("files") MultipartFile[] files) {
    StringBuilder resultMessage = new StringBuilder();

    for (MultipartFile file : files) {
        if (file.isEmpty()) {
            resultMessage.append("Failed to upload empty fi
le.\n");
            continue;
        }

        try {
            String fileName = file.getOriginalFilename();
            File destinationFile = new File(UPLOAD_DIR + fi
leName);

            file.transferTo(destinationFile);
            resultMessage.append("File uploaded successfull
y: ").append(fileName).append("\n");
        } catch (IOException e) {
            e.printStackTrace();
            resultMessage.append("Failed to upload file:
").append(file.getOriginalFilename()).append("\n");
        }
    }
}

```

```

        return new ResponseEntity<>(resultMessage.toString(), H
ttpStatus.OK);
    }

```

- **Explanation:**

- `@RequestParam("files") MultipartFile[] files` : Binds multiple uploaded files to an array of `MultipartFile` objects.
- The method iterates over each file and processes them as before.

## 7. Handling Errors and Validation

You can add validation and error handling to ensure the uploaded files meet your requirements (e.g., file type, file size).

### Example: Validating File Type and Size:

```

@PostMapping("/upload")
public ResponseEntity<String> uploadFile(@RequestParam("fil
e") MultipartFile file) {
    if (file.isEmpty()) {
        return new ResponseEntity<>("Please select a file t
o upload.", HttpStatus.BAD_REQUEST);
    }

    // Validate file type
    String contentType = file.getContentType();
    if (!contentType.equals("image/jpeg") && !contentType.e
quals("image/png")) {
        return new ResponseEntity<>("Only JPEG and PNG file
s are allowed.", HttpStatus.BAD_REQUEST);
    }

    // Validate file size
    if (file.getSize() > 2 * 1024 * 1024) { // 2MB
        return new ResponseEntity<>("File size should be le
ss than 2MB.", HttpStatus.BAD_REQUEST);
    }
}

```

```

    }

    try {
        // Save the file
        String fileName = file.getOriginalFilename();
        File destinationFile = new File(UPLOAD_DIR + fileName);

        file.transferTo(destinationFile);

        return new ResponseEntity<>("File uploaded successfully: " + fileName, HttpStatus.OK);
    } catch (IOException e) {
        e.printStackTrace();
        return new ResponseEntity<>("Failed to upload the file: " + e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

## Rest APIS

- **REST (Representational State Transfer):** An architectural style for designing networked applications, relying on a stateless, client-server communication protocol—usually HTTP.
- It's a way for different software applications to communicate with each other over the internet.
- **Resources:** Everything in a RESTful API is considered a resource. Each resource is identified by a URI (Uniform Resource Identifier).

### HTTP Methods:

REST APIs commonly use HTTP methods to interact with resources:

- **GET:** Retrieve data from the server (e.g., fetch a list of users).
- **POST:** Create a new resource on the server (e.g., create a new user).
- **PUT:** Update an existing resource on the server (e.g., update user details).

- **DELETE:** Remove a resource from the server (e.g., delete a user).

## Designing RESTful APIs

- **Resource Naming Conventions:**

- Use nouns to represent resources, not verbs (e.g., `/users` rather than `/getUsers` ).
- Use plural names for resources (e.g., `/users` instead of `/user` ).
- Keep URLs simple and intuitive.

- **Versioning:**

- Version your API using the URL (e.g., `/v1/users` ) or using headers.
- This helps in managing updates and maintaining backward compatibility.

- **Statelessness:**

- Each API call should contain all the information the server needs to fulfill the request.
- The server does not store any session information between requests.

- **HTTP Status Codes:**

- Use appropriate status codes to indicate the result of the API request:
  - **200 OK:** Request succeeded.
  - **201 Created:** Resource successfully created.
  - **204 No Content:** Request succeeded, but no content to return.
  - **400 Bad Request:** The request was invalid.
  - **401 Unauthorized:** Authentication is required and has failed or not been provided.
  - **403 Forbidden:** The request was valid, but the server is refusing action.
  - **404 Not Found:** The requested resource was not found.
  - **500 Internal Server Error:** A generic error message for server issues.

## Example of a REST API

Let's consider a simple example: a REST API for managing a collection of books in a library.

### 1. Getting a List of Books (GET Request):

#### Request:

```
GET /api/books
```

#### Response:

```
[
  {
    "id": 1,
    "title": "The Great Gatsby",
    "author": "F. Scott Fitzgerald",
    "published_year": 1925
  },
  {
    "id": 2,
    "title": "To Kill a Mockingbird",
    "author": "Harper Lee",
    "published_year": 1960
  }
]
```

This request fetches a list of all the books available in the library.

### 2. Getting a Specific Book (GET Request):

#### Request:

```
GET /api/books/1
```



### Response:

```
{
  "id": 1,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "published_year": 1925
}
```

This request fetches the details of the book with ID 1.

### 3. Creating a New Book (POST Request):

#### Request:

```
POST /api/books
Content-Type: application/json

{
  "title": "1984",
  "author": "George Orwell",
  "published_year": 1949
}
```

#### Response:

```
{
  "id": 3,
  "title": "1984",
  "author": "George Orwell",
  "published_year": 1949
}
```

```
}
```

This request creates a new book in the library and returns the details of the newly created book.

#### 4. Updating an Existing Book (PUT Request):

##### Request:

```
PUT /api/books/1
Content-Type: application/json

{
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "published_year": 1926
}
```

##### Response:

```
jsonCopy code
{
  "id": 1,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "published_year": 1926
}
```

This request updates the details of the book with ID 1.

#### 5. Deleting a Book (DELETE Request):

##### Request:

```
DELETE /api/books/1
```

### Response:

```
{
  "message": "Book deleted successfully"
}
```

This request deletes the book with ID 1 from the library.

## Understanding Nested Resources

- **Categories and Products:** In this context, a category can have multiple products, and each product belongs to a category. This creates a parent-child relationship where `Category` is the parent resource and `Product` is the child resource.
- **Nested URLs:** These represent the relationship between the parent and child resources. For example, `/categories/{categoryId}/products` refers to all products within a specific category.

## 2. Designing the API Endpoints

### 1. Retrieve All Categories

#### Request:

```
GET /api/categories
```

#### Response:

```
[
  {
    "id": 1,
```

```
        "name": "Electronics"
    },
    {
        "id": 2,
        "name": "Books"
    }
]
```

This endpoint retrieves a list of all categories.

## 2. Retrieve All Products in a Specific Category

**Request:**

```
GET /api/categories/1/products
```

**Response:**

```
[
  {
    "id": 101,
    "name": "Smartphone",
    "price": 699.99
  },
  {
    "id": 102,
    "name": "Laptop",
    "price": 1299.99
  }
]
```

This endpoint retrieves all products under the category with ID 1.

## 3. Retrieve a Specific Product in a Category

**Request:**

```
GET /api/categories/1/products/101
```

**Response:**

```
jsonCopy code
{
  "id": 101,
  "name": "Smartphone",
  "price": 699.99,
  "description": "Latest model smartphone with advanced features."
}
```

This endpoint retrieves the product with ID 101 within category 1.

## 4. Create a New Product in a Specific Category

**Request:**

```
POST /api/categories/1/products
Content-Type: application/json

{
  "name": "Tablet",
  "price": 499.99,
  "description": "A powerful tablet with a sleek design."
}
```

**Response:**

```
{
```

```
"id": 103,  
"name": "Tablet",  
"price": 499.99,  
"description": "A powerful tablet with a sleek design."  
}
```

This endpoint creates a new product under category 1.

## 5. Update a Product in a Specific Category

### Request:

```
PUT /api/categories/1/products/103  
Content-Type: application/json  
  
{  
  "name": "Tablet",  
  "price": 549.99,  
  "description": "Updated tablet with enhanced features."  
}
```

### Response:

```
{  
  "id": 103,  
  "name": "Tablet",  
  "price": 549.99,  
  "description": "Updated tablet with enhanced features."  
}
```

This endpoint updates the product with ID 103 within category 1.

## 6. Delete a Product in a Specific Category

### Request:

```
DELETE /api/categories/1/products/103
```

**Response:**

```
{  
  "message": "Product deleted successfully"  
}
```

This endpoint deletes the product with ID 103 within category 1.

Tradition way to create web app: