5-Spring Web MVC Basics

Author: Vincent Lau

Learning Objectives

- Understand Spring Web MVC's architecture.
- Understand the difference between MVC and RESTful controllers.
- Implement a RESTful controller.
- Understand the @SpringBootApplication annotation and how a Spring Boot application runs.
- Understand the various annotations for implementing a controller.

Overview

MVC Architecture

- MVC stands for Model-View-Controller, and it separates an application into three logical components:
  - ***Model*** - The Model component deals with data-related logic that the user works with. For example, a user object will retrieve user information from the database, manipulate it and update the data back to the database, or use it to render the view.
  - ***View*** - A View represents the presentation of data. The View component includes all UI components, such as text boxes and buttons. Views are created with the collected data from the Model.
  - ***Controller*** - The interface that sits between Model and View to process all business logic. It is the "brain" of the application that controls how data is manipulated in Model and displayed in View.
- MVC separates the business logic and presentation layer from each other, thus achieving *Separation of Concerns*.

What is Spring MVC?

- The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications.
- The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic) while providing a loose coupling between these elements.
  - The ***Model*** encapsulates the application data, and in general they consist of POJOs.
  - The ***View*** is responsible for rendering the model data, and in general it generates HTML output that the client's browser can interpret.
  - The ***Controller*** is responsible for processing user requests, building an appropriate model, and passing it to the view for rendering.

DispatcherServlet

- The Spring Web MVC framework is designed around the ***DispatcherServlet*** that handles all the HTTP requests and responses.
  - After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
  - The ***Controller*** takes the request and calls the appropriate service methods based on the used GET or POST method. The service method will set **model data** based on defined business logic and returns the view **name** to the *DispatcherServlet*.
  - The ***DispatcherServlet*** will take help from *ViewResolver* to pick up the defined view for the request.

    - Once the view is finalized, the *DispatcherServlet* **passes the model data to the view** which is finally rendered on the browser.

RESTful Controllers

- The diagram applies to both typical MVC and RESTful controllers - with some small differences.
    - In the traditional approach, *MVC* applications are **not service-oriented** hence there is a *ViewResolver* that renders final views based on data received from a *Controller*.
    - ***RESTful*** applications are designed to be **service-oriented** and return raw data (JSON typically). Since these applications do not do any view rendering, there is no *ViewResolver* - the *Controller* is generally expected to **send data directly via the HTTP response** (thus skipping Steps 3 and 4 in the diagram).

Setting up the Project

Starting with Spring Initializr

1. Navigate to https://start.spring.io. This service pulls in all the dependencies you need for an application and does most of your setup.
2. Select "**Maven Project**" for Project and **Java** for Language.
3. Click **Dependencies** and select **Spring Web** and **Thymeleaf**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

Maven Dependencies

- The resulting *pom.xml* should look something like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
```

```xml
        <version>2.6.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.bootcamp</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-mvc-demo</name>
    <description>Demo project for Spring MVC</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Creating a Controller

- In Spring's approach to building websites, HTTP requests are handled by a controller.

- You can easily identify the controller by the `@Controller` annotation.
- In the following example, *GreetingController* handles GET requests for `/greeting` by returning the name of a View (in this case greeting).
- A *View* is responsible for rendering the HTML content.

```java
@Controller
public class GreetingController {
    @GetMapping("/greeting")
    public String greeting(@RequestParam(name="name", required=false,
defaultValue="World") String name, Model model) {
        model.addAttribute("name", name);
        return "greeting";
    }
}
```

- The `@GetMapping` annotation ensures that HTTP GET requests to `/greeting` are mapped to the `greeting()` method.
- `@RequestParam` binds the value of the query string parameter *name* into the *name* parameter of the `greeting()` method. This query string parameter is not *required*. If it is absent in the request, the *defaultValue* of World is used.
- The value of the *name* parameter is added to a *Model* object, ultimately making it accessible to the view template.

Creating a View

- The implementation of the method body relies on a view technology (in this case, Thymeleaf) to perform server-side rendering of the HTML.
- Thymeleaf parses the `greeting.html` template and evaluates the `th:text` expression to render the value of the `${name}` parameter that was set in the controller.

```html
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Getting Started: Serving Web Content</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    </head>
    <body>
        <p th:text="'Hello, ' + ${name} + '!'" />
    </body>
</html>
```

Run the Application

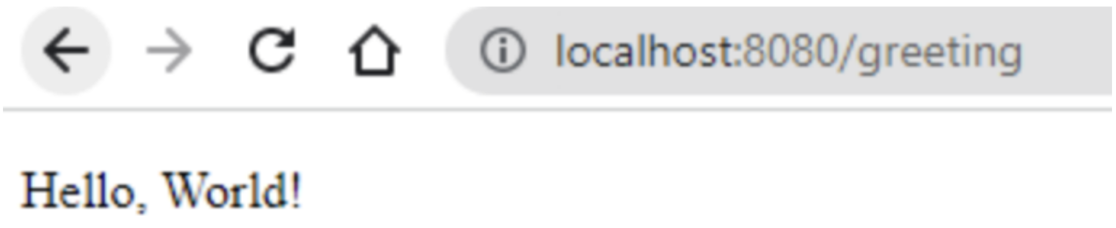
- The Spring Initializr creates an application class for you. In this case, you need not further modify the class provided by the Spring Initializr.

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

- `@SpringBootApplication` is a convenience annotation that adds all of the following:
    - `@Configuration` - Tags the class as a source of bean definitions for the application context.
    - `@EnableAutoConfiguration` - Tells the framework to add beans based on the dependencies on the classpath automatically. For example, if *spring-webmvc* is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a *DispatcherServlet*.
    - `@ComponentScan` - Scans for other configurations and beans in the same package as the *Application* class or deeper in the hierarchy.
- With ***Spring Boot***, we can set up frontend using ***Thymeleaf*** or *JSP* without defining a *ViewResolver* or XML configuration file.
- By adding the ***spring-boot-starter-thymeleaf*** dependency to our *pom.xml*, *Thymeleaf* gets enabled, and no extra configuration is necessary.

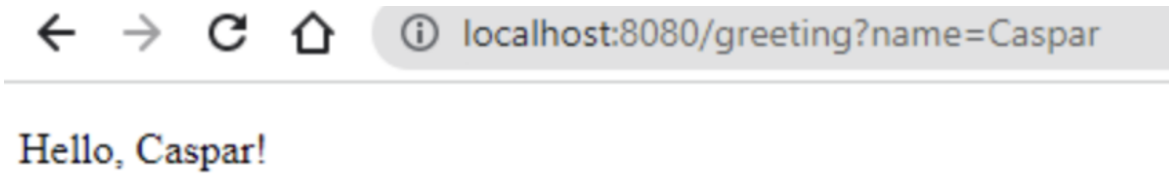
Accessing the Web Page

- When the project runs locally, the *greeting.html* can be accessed at `http://localhost:8080/greeting`.



Hello, World!

- If we specify the *name* parameter in the query string by using `http://localhost:8080/greeting?name=Caspar`, we can see the *name* parameter being picked up by the view which is populated by the model.



Hello, Caspar!

The REST Controller

- The setup for a *Spring RESTful* application is the same as the one for the *MVC* application with the only difference being that there is no *View Resolvers* and no *model mapping.*
- In general, the API will simply return raw data back to the client - *JSON* representations usually - and so the *DispatcherServlet* bypasses the *View Resolvers* and **returns the data right in the HTTP response body**.
- Suppose we have a *GreetingMessage* class:

```
public class GreetingMessage {
    private String message;

    public GreetingMessage(String name) {
        this.message = String.format("Hello %s!", name);
    }

    public String getMessage() {
        return message;
    }
}
```

- Let's look at a simple *RESTful Controller* implementation:

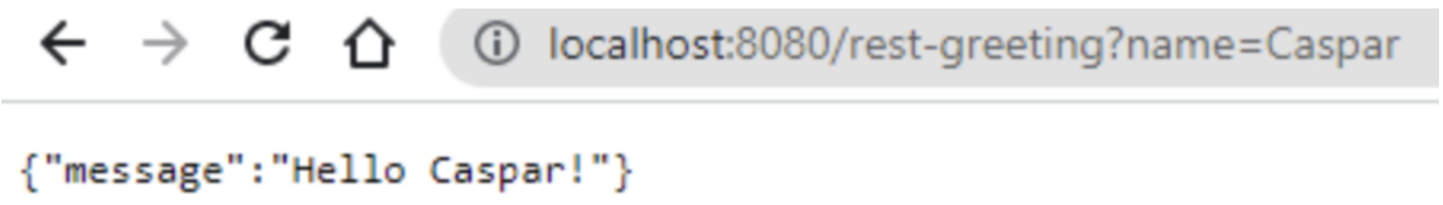
```
@Controller
```

```java
public class GreetingController {

    @ResponseBody
    @GetMapping("/rest-greeting")
    public GreetingMessage getGreetingMessage(@RequestParam(name = "name",
required = false, defaultValue = "World") String name) {
        return new GreetingMessage(name);
    }

}
```

- Note the `@ResponseBody` annotation on the method - which instructs Spring to bypass the *View Resolver* and **essentially writes out the output directly to the HTTP response body**.



```
localhost:8080/rest-greeting?name=Caspar
```

```
{"message":"Hello Caspar!"}
```

- `@GetMapping` is a shortcut of the `@RequestMapping` annotation. To understand more about how the `@RequestMapping` annotation works, read this.

Spring Boot and the @RestController Annotation

- The `@RestController` annotation from Spring Boot is basically a shortcut that saves us from always having to define `@ResponseBody`.

```java
@RestController // @Controller + @ResponseBody
public class GreetingController {

    @GetMapping("/rest-greeting")
    public GreetingMessage getGreetingMessage(@RequestParam(name = "name",
required = false, defaultValue = "World") String name) {
        return new GreetingMessage(name);
    }

}
```

Questions

- How does the Spring MVC architecture work?
- What is the difference between MVC and RESTful controllers?
- What annotations does the `@SpringBootApplication` annotation include?
- What is the use of `@RequestParam` and `@ResponseBody` ?
- What is the difference between `@Controller` and `@RestController` ?