ALL GUIDES

Serving Web Content with Spring MVC

This guide walks you through the process of creating a "Hello, World" web site with Spring.

Get the Code
Go To Repo

What You Will Build

You will build an application that has a static home page and that will also accept HTTP GET requests at `http://localhost:8080/greeting`.

It will respond with a web page that displays HTML. The body of the HTML will contain a greeting: "Hello, World!"

You can customize the greeting with an optional `name` parameter in the query string. The URL might then be `http://localhost:8080/greeting?name=User`.

The `name` parameter value overrides the default value of `World` and is reflected in the response by the content changing to "Hello, User!"

What You Need

- About 15 minutes
- A favorite text editor or IDE
- Java 17 or later
- Gradle 7.5+ or Maven 3.5+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA
  - VSCode

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Starting with Spring Initializr.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-serving-web-content.git`
- cd into `gs-serving-web-content/initial`
- Jump ahead to Create a Web Controller.

When you finish, you can check your results against the code in `gs-serving-web-content/complete`.

Starting with Spring Initializr

You can use this pre-initialized project and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

To manually initialize the project:

1. Navigate to https://start.spring.io. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring Web**, **Thymeleaf**, and **Spring Boot DevTools**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

> If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

> You can also fork the project from Github and open it in your IDE or other editor.

Create a Web Controller

In Spring's approach to building web sites, 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. The following listing (from `src/main/java/com/example/servingwebcontent/GreetingController.java`) shows the controller:

```
package com.example.servingwebcontent;

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 GreetingController {

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

}
```

This controller is concise and simple, but there is plenty going on. We break it down step by step.

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.

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.The following listing (from `src/main/resources/templates/greeting.html`) shows the `greeting.html` template:

```
<!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>
```

> Make sure you have Thymeleaf on your classpath (artifact co-ordinates: `org.springframework.boot:spring-boot-starter-thymeleaf`). It is already there in the "initial" and "complete" samples in Github.

Spring Boot Devtools

A common feature of developing web applications is coding a change, restarting your application, and refreshing the browser to view the change. This entire process can eat up a lot of time. To speed up this refresh cycle, Spring Boot offers with a handy module known as spring-boot-devtools. Spring Boot Devtools:

- Enables hot swapping.
- Switches template engines to disable caching.
- Enables LiveReload to automatically refresh the browser.
- Other reasonable defaults based on development instead of production.

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. The following listing (from `src/main/java/com/example/servingwebcontent/ServingWebContentApplication.java`) shows the application class:

```
package com.example.servingwebcontent;

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

@SpringBootApplication
public class ServingWebContentApplication {

    public static void main(String[] args) {
        SpringApplication.run(ServingWebContentApplication.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 Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. 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`: Tells Spring to look for other components, configurations, and services in the `com/example` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there was not a single line of XML? There is no `web.xml` file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

```
java -jar build/libs/gs-serving-web-content-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-serving-web-content-0.1.0.jar
```

> The steps described here create a runnable JAR. You can also build a classic WAR file.

Logging output is displayed. The application should be up and running within a few seconds.

Test the Application

Now that the web site is running, visit `http://localhost:8080/greeting`, where you should see "Hello, World!"

Provide a `name` query string parameter by visiting `http://localhost:8080/greeting?name=User`. Notice how the message changes from "Hello, World!" to "Hello, User!".

This change demonstrates that the `@RequestParam` arrangement in `GreetingController` is working as expected. The `name` parameter has been given a default value of `World`, but it can be explicitly overridden through the query string.

Add a Home Page

Static resources, including HTML and JavaScript and CSS, can be served from your Spring Boot application by dropping them into the right place in the source code. By default, Spring Boot serves static content from resources in the classpath at `/static` (or `/public`). The `index.html` resource is special because, if it exists, it is used as a "welcome page", which means it is served up as the root resource (that is, at `http://localhost:8080/`). As a result, you need to create the following file (which you can find in `src/main/resources/static/index.html`):

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <p>Get your greeting <a href="/greeting">here</a></p>
</body>
</html>
```

When you restart the application, you will see the HTML at `http://localhost:8080/`.

Summary

Congratulations! You have just developed a web page by using Spring.

See Also

The following guides may also be helpful:

- Building an Application with Spring Boot
- Accessing Data with GemFire
- Accessing Data with JPA
- Accessing Data with MongoDB
- Accessing data with MySQL
- Testing the Web Layer
- Building a RESTful Web Service

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.

Get the Spring newsletter
Stay connected with the Spring newsletter
SUBSCRIBE