<https://spring.io/guides/gs/serving-web-content/>

**Serving Web Content with Spring MVC**

This guide walks you through the process of creating a "hello world" web site with Spring.

**What you’ll build**

You’ll build an application that has a static home page, and also will accept HTTP GET requests at:

http://localhost:8080/greeting

and respond with a web page displaying HTML. The body of the HTML contains a greeting:

"Hello, World!"

You can customize the greeting with an optional name parameter in the query string:

http://localhost:8080/greeting?name=User

The name parameter value overrides the default value of "World" and is reflected in the response:

"Hello, User!"

**What you’ll need**

* About 15 minutes
* A favorite text editor or IDE
* [JDK 1.8](http://www.oracle.com/technetwork/java/javase/downloads/index.html) or later
* [Gradle 4+](http://www.gradle.org/downloads) or [Maven 3.2+](https://maven.apache.org/download.cgi)
* You can also import the code straight into your IDE:
  + [Spring Tool Suite (STS)](https://spring.io/guides/gs/sts)
  + [IntelliJ IDEA](https://spring.io/guides/gs/intellij-idea/)

**How to complete this guide**

Like most Spring [Getting Started guides](https://spring.io/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 [Build with Gradle](https://spring.io/guides/gs/serving-web-content/#scratch).

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

* [Download](https://github.com/spring-guides/gs-serving-web-content/archive/master.zip) and unzip the source repository for this guide, or clone it using [Git](https://spring.io/understanding/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](https://spring.io/guides/gs/serving-web-content/#initial).

**When you’re finished**, you can check your results against the code in gs-serving-web-content/complete.

**Build with Gradle**

**Build with Maven**

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with [Maven](https://maven.apache.org/) is included here. If you’re not familiar with Maven, refer to [Building Java Projects with Maven](https://spring.io/guides/gs/maven).

**Create the directory structure**

In a project directory of your choosing, create the following subdirectory structure; for example, with mkdir -p src/main/java/hello on \*nix systems:

└── src

└── main

└── java

└── hello

pom.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>

<groupId>org.springframework</groupId>

<artifactId>gs-serving-web-content</artifactId>

<version>0.1.0</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.6.RELEASE</version>

</parent>

<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-devtools</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

<properties>

<java.version>1.8</java.version>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

The [Spring Boot Maven plugin](https://docs.spring.io/spring-boot/docs/current/maven-plugin) provides many convenient features:

* It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.
* It searches for the public static void main() method to flag as a runnable class.
* It provides a built-in dependency resolver that sets the version number to match [Spring Boot dependencies](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-dependencies/pom.xml). You can override any version you wish, but it will default to Boot’s chosen set of versions.

**Build with your IDE**

**Create a web controller**

In Spring’s approach to building web sites, HTTP requests are handled by a controller. You can easily identify these requests by the [@Controller](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Controller.html) annotation. In the following example, the GreetingController handles GET requests for /greeting by returning the name of a [View](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/View.html), in this case, "greeting". A View is responsible for rendering the HTML content:

src/main/java/hello/GreetingController.java

package hello;

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’s plenty going on. Let’s break it down step by step.

The @GetMapping annotation ensures that HTTP GET requests to /greeting are mapped to the greeting() method.

[@RequestParam](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestParam.html) 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](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/ui/Model.html) object, ultimately making it accessible to the view template.

The implementation of the method body relies on a [view technology](https://spring.io/understanding/view-templates), in this case [Thymeleaf](http://www.thymeleaf.org/doc/tutorials/2.1/thymeleafspring.html), to perform server-side rendering of the HTML. Thymeleaf parses the greeting.html template below and evaluates the th:text expression to render the value of the ${name} parameter that was set in the controller.

src/main/resources/templates/greeting.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>

**Developing web apps**

A common feature of developing web apps is coding a change, restarting your app, and refreshing the browser to view the change. This entire process can eat up a lot of time. To speed up the cycle of things, Spring Boot comes with a handy module known as [spring-boot-devtools](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools).

* Enable [hot swapping](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-hotswapping)
* Switches template engines to disable caching
* Enables LiveReload to refresh browser automatically
* Other reasonable defaults based on development instead of production

**Make the application executable**

Although it is possible to package this service as a traditional [WAR](https://spring.io/understanding/WAR) file for deployment to an external application server, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a good old Java main() method. Along the way, you use Spring’s support for embedding the [Tomcat](https://spring.io/understanding/Tomcat) servlet container as the HTTP runtime, instead of deploying to an external instance.

src/main/java/hello/Application.java

package hello;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.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.
* Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This 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 hello package, allowing it to find the controllers.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application. Did you notice that there wasn’t a single line of XML? No **web.xml** file either. This web application is 100% pure Java and you didn’t 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. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This 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 are using Gradle, you can run the application using ./gradlew bootRun. Or you can build the JAR file using ./gradlew build. Then you can run the JAR file:

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

If you are using Maven, you can run the application using ./mvnw spring-boot:run. Or you can build the JAR file with ./mvnw clean package. Then you can run the JAR file:

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

|  |  |
| --- | --- |
|  | The procedure above will create a runnable JAR. You can also opt to [build a classic WAR file](https://spring.io/guides/gs/convert-jar-to-war/)instead. |

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

**Test the App**

Now that the web site is running, visit <http://localhost:8080/greeting>, where you see:

"Hello, World!"

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

"Hello, User!"

This change demonstrates that the [@RequestParam](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestParam.html) arrangement in GreetingController is working as expected. The name parameter has been given a default value of "World", but can always be explicitly overridden through the query string.

**Add a Home Page**

Static resources, like HTML or JavaScript or CSS, can easily be served from your Spring Boot application just 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 it is used as a "welcome page" if it exists, which means it will be served up as the root resource, i.e. at <http://localhost:8080/> in our example. So create this file:

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>

and when you restart the app you will see the HTML at <http://localhost:8080/>.

**Summary**

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

**See Also**

The following guides may also be helpful:

* [Building an Application with Spring Boot](https://spring.io/guides/gs/spring-boot/)
* [Accessing Data with GemFire](https://spring.io/guides/gs/accessing-data-gemfire/)
* [Accessing Data with JPA](https://spring.io/guides/gs/accessing-data-jpa/)
* [Accessing Data with MongoDB](https://spring.io/guides/gs/accessing-data-mongodb/)
* [Accessing data with MySQL](https://spring.io/guides/gs/accessing-data-mysql/)
* [Testing the Web Layer](https://spring.io/guides/gs/testing-web/)
* [Building a RESTful Web Service](https://spring.io/guides/gs/rest-service/)

Want to write a new guide or contribute to an existing one? Check out our [contribution guidelines](https://github.com/spring-guides/getting-started-guides/wiki).

|  |  |
| --- | --- |
|  | All guides are released with an ASLv2 license for the code, and an [Attribution, NoDerivatives creative commons license](https://creativecommons.org/licenses/by-nd/3.0/) |