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 do

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 or later
Gradle 2.3+ or Maven 3.0+
You can also import the code from this guide as well as view the web page directly into Spring Tool Suite (STS) and work your way through it from there.

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 Build with Gradle.
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're finished, you can check your results against the code in `gs-serving-web-content/complete`.

Gradle

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 Gradle and Maven is included here. If you're not familiar with either, refer to Building Java Projects with Gradle or Building Java Projects with 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
```


Page **1** of **7**

```
    └── main
        └── java
            └── hello
```

Create a Gradle build file

Below is the initial Gradle build file.

`build.gradle`

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:1.4.2.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'gs-serving-web-content'
    version =  '0.1.0'
}

repositories {
    mavenCentral()
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    compile("org.springframework.boot:spring-boot-starter-thymeleaf")
    compile("org.springframework.boot:spring-boot-devtools")
    testCompile("junit:junit")
}
```

The Spring Boot gradle 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. You can override any version you wish, but it will default to Boot's chosen set of versions.

Page **2** of **7**

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 is included here. If you're not familiar with Maven, refer to Building Java Projects with 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
http://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>1.4.2.RELEASE</version>
    </parent>

    <dependencies>
        <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>
```


Page **3** of **7**

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

</project>
```

The Spring Boot 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. You can override any version you wish, but it will default to Boot's chosen set of versions.

IDE

- Read how to import this guide straight into Spring Tool Suite.
- Read how to work with this guide in IntelliJ IDEA.

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` annotation. In the following example, the 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:

`src/main/java/hello/GreetingController.java`

```
package hello;

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

@Controller
public class GreetingController {

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


Page **4** of **7**

```
}
```

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

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

> The above example does not specify `GET` vs. `PUT`, `POST`, and so forth, because `@RequestMapping` maps all HTTP operations by default. Use `@RequestMapping(method=GET)` to narrow this mapping.

`@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 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.

Enable hot swapping
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 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 servlet container as the HTTP runtime, instead of deploying to an external instance.

Page **5** of **7**

`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 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 fileinstead.

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

Page **6** of **7**

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` 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 be 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.

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

Page **7** of **7**