ALL GUIDES

Building an Application with Spring Boot

This guide provides a sampling of how Spring Boot helps you accelerate application development. As you read more Spring Getting Started guides, you will see more use cases for Spring Boot. This guide is meant to give you a quick taste of Spring Boot. If you want to create your own Spring Boot-based project, visit Spring Initializr, fill in your project details, pick your options, and download a bundled up project as a zip file.

What You Will build

You will build a simple web application with Spring Boot and add some useful services to it.

What You Need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 4+ or Maven 3.2+
- You can also import the code straight into your IDE:
    - Spring Tool Suite (STS)
    - IntelliJ IDEA

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-spring-boot.git`
- cd into `gs-spring-boot/initial`
- Jump ahead to Create a Simple Web Application.

When you finish, you can check your results against the code in `gs-spring-boot/complete`.

Learn What You Can Do with Spring Boot

Spring Boot offers a fast way to build applications. It looks at your classpath and at the beans you have configured, makes reasonable assumptions about what you are missing, and adds those items. With Spring Boot, you can focus more on business features and less on infrastructure.

The following examples show what Spring Boot can do for you:

- Is Spring MVC on the classpath? There are several specific beans you almost always need, and Spring Boot adds them automatically. A Spring MVC application also needs a servlet container, so Spring Boot automatically configures embedded Tomcat.
- Is Jetty on the classpath? If so, you probably do NOT want Tomcat but instead want embedded Jetty. Spring Boot handles that for you.
- Is Thymeleaf on the classpath? If so, there are a few beans that must always be added to your application context. Spring Boot adds them for you.

These are just a few examples of the automatic configuration Spring Boot provides. At the same time, Spring Boot does not get in your way. For example, if Thymeleaf is on your path, Spring Boot automatically adds a `SpringTemplateEngine` to your application context. But if

you define your own `SpringTemplateEngine` with your own settings, Spring Boot does not add one. This leaves you in control with little effort on your part.

> Spring Boot does not generate code or make edits to your files. Instead, when you start your application, Spring Boot dynamically wires up beans and settings and applies them to your application context.

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**.
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 Simple Web Application

Now you can create a web controller for a simple web application, as the following listing (from `src/main/java/com/example/springboot/HelloController.java`) shows:

```
package com.example.springboot;

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

@RestController
public class HelloController {

        @GetMapping("/")
        public String index() {
                return "Greetings from Spring Boot!";
        }

}
```

COPY

The class is flagged as a `@RestController`, meaning it is ready for use by Spring MVC to handle web requests. `@GetMapping` maps `/` to the `index()` method. When invoked from a browser or by using curl on the command line, the method returns pure text. That is because `@RestController` combines `@Controller` and `@ResponseBody`, two annotations that results in web requests returning data rather than a view.

Create an Application class

The Spring Initializr creates a simple application class for you. However, in this case, it is too simple. You need to modify the application class to match the following listing (from `src/main/java/com/example/springboot/Application.java`):

```
package com.example.springboot;

import java.util.Arrays;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class Application {
```

COPY

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

	@Bean
	public CommandLineRunner commandLineRunner(ApplicationContext ctx) {
		return args -> {

			System.out.println("Let's inspect the beans provided by
Spring Boot:");

			String[] beanNames = ctx.getBeanDefinitionNames();
			Arrays.sort(beanNames);
			for (String beanName : beanNames) {
				System.out.println(beanName);
			}

		};
	}

}
```

`@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.

There is also a `CommandLineRunner` method marked as a `@Bean`, and this runs on start up. It retrieves all the beans that were created by your application or that were automatically added by Spring Boot. It sorts them and prints them out.

Run the Application

To run the application, run the following command in a terminal window (in the `complete`) directory:

```
./gradlew bootRun
```

If you use Maven, run the following command in a terminal window (in the `complete`) directory:

```
./mvnw spring-boot:run
```

You should see output similar to the following:

```
Let's inspect the beans provided by Spring Boot:
application
beanNameHandlerMapping
defaultServletHandlerMapping
dispatcherServlet
embeddedServletContainerCustomizerBeanPostProcessor
handlerExceptionResolver
helloController
httpRequestHandlerAdapter
messageSource
mvcContentNegotiationManager
mvcConversionService
mvcValidator
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfigurati
on
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfigurati
on$DispatcherServletConfiguration
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfigurati
on$EmbeddedTomcat
org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration
org.springframework.boot.context.embedded.properties.ServerProperties
org.springframework.context.annotation.ConfigurationClassPostProcessor.enhancedCon
figurationProcessor
org.springframework.context.annotation.ConfigurationClassPostProcessor.importAware
Processor
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.annotation.internalRequiredAnnotationProcessor
org.springframework.web.servlet.config.annotation.DelegatingWebMvcConfiguration
propertySourcesBinder
propertySourcesPlaceholderConfigurer
```

```
requestMappingHandlerAdapter
requestMappingHandlerMapping
resourceHandlerMapping
simpleControllerHandlerAdapter
tomcatEmbeddedServletContainerFactory
viewControllerHandlerMapping
```

You can clearly see `org.springframework.boot.autoconfigure` beans. There is also a `tomcatEmbeddedServletContainerFactory`.

Now run the service with curl (in a separate terminal window), by running the following command (shown with its output):

```
$ curl localhost:8080
Greetings from Spring Boot!
```

COPY

Add Unit Tests

You will want to add a test for the endpoint you added, and Spring Test provides some machinery for that.

If you use Gradle, add the following dependency to your `build.gradle` file:

```
testImplementation('org.springframework.boot:spring-boot-starter-test')
```

COPY

If you use Maven, add the following to your `pom.xml` file:

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

COPY

Now write a simple unit test that mocks the servlet request and response through your endpoint, as the following listing (from `src/test/java/com/example/springboot/HelloControllerTest.java`) shows:

```
package com.example.springboot;

import static org.hamcrest.Matchers.equalTo;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
```

COPY

```
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

@SpringBootTest
@AutoConfigureMockMvc
public class HelloControllerTest {

        @Autowired
        private MockMvc mvc;

        @Test
        public void getHello() throws Exception {

mvc.perform(MockMvcRequestBuilders.get("/").accept(MediaType.APPLICATION_JSON))
                                .andExpect(status().isOk())
                                .andExpect(content().string(equalTo("Greetings
from Spring Boot!")));
        }
}
```

`MockMvc` comes from Spring Test and lets you, through a set of convenient builder classes, send HTTP requests into the `DispatcherServlet` and make assertions about the result. Note the use of `@AutoConfigureMockMvc` and `@SpringBootTest` to inject a `MockMvc` instance. Having used `@SpringBootTest`, we are asking for the whole application context to be created. An alternative would be to ask Spring Boot to create only the web layers of the context by using `@WebMvcTest`. In either case, Spring Boot automatically tries to locate the main application class of your application, but you can override it or narrow it down if you want to build something different.

As well as mocking the HTTP request cycle, you can also use Spring Boot to write a simple full-stack integration test. For example, instead of (or as well as) the mock test shown earlier, we could create the following test (from `src/test/java/com/example/springboot/HelloControllerIT.java`):

```
package com.example.springboot;

import org.junit.jupiter.api.Test;
```

COPY

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.ResponseEntity;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerIT {

        @Autowired
        private TestRestTemplate template;

    @Test
    public void getHello() throws Exception {
        ResponseEntity<String> response = template.getForEntity("/",
String.class);
        assertThat(response.getBody()).isEqualTo("Greetings from Spring Boot!");
    }
}
```

The embedded server starts on a random port because of `webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT`, and the actual port is configured automatically in the base URL for the `TestRestTemplate`.

Add Production-grade Services

If you are building a web site for your business, you probably need to add some management services. Spring Boot provides several such services (such as health, audits, beans, and more) with its actuator module.

If you use Gradle, add the following dependency to your `build.gradle` file:

```
implementation 'org.springframework.boot:spring-boot-starter-actuator'
```

COPY

If you use Maven, add the following dependency to your `pom.xml` file:

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

COPY

Then restart the application. If you use Gradle, run the following command in a terminal window (in the `complete` directory):

```
./gradlew bootRun
```

If you use Maven, run the following command in a terminal window (in the `complete` directory):

```
./mvnw spring-boot:run
```

You should see that a new set of RESTful end points have been added to the application. These are management services provided by Spring Boot. The following listing shows typical output:

```
management.endpoint.configprops-
org.springframework.boot.actuate.autoconfigure.context.properties.ConfigurationPropertiesReportEndpointProperties
management.endpoint.env-
org.springframework.boot.actuate.autoconfigure.env.EnvironmentEndpointProperties
management.endpoint.health-
org.springframework.boot.actuate.autoconfigure.health.HealthEndpointProperties
management.endpoint.logfile-
org.springframework.boot.actuate.autoconfigure.logging.LogFileWebEndpointProperties
management.endpoints.jmx-
org.springframework.boot.actuate.autoconfigure.endpoint.jmx.JmxEndpointProperties
management.endpoints.web-
org.springframework.boot.actuate.autoconfigure.endpoint.web.WebEndpointProperties
management.endpoints.web.cors-
org.springframework.boot.actuate.autoconfigure.endpoint.web.CorsEndpointProperties
management.health.diskspace-
org.springframework.boot.actuate.autoconfigure.system.DiskSpaceHealthIndicatorProperties
management.info-
org.springframework.boot.actuate.autoconfigure.info.InfoContributorProperties
management.metrics-
org.springframework.boot.actuate.autoconfigure.metrics.MetricsProperties
management.metrics.export.simple-
org.springframework.boot.actuate.autoconfigure.metrics.export.simple.SimpleProperties
management.server-
org.springframework.boot.actuate.autoconfigure.web.server.ManagementServerProperties
```

The actuator exposes the following:

- actuator/health
- actuator

> There is also an `/actuator/shutdown` endpoint, but, by default, it is visible only through JMX. To enable it as an HTTP endpoint, add `management.endpoint.shutdown.enabled=true` to your `application.properties` file and expose it with `management.endpoints.web.exposure.include=health,info,shutdown`. However, you probably should not enable the shutdown endpoint for a publicly available application.

You can check the health of the application by running the following command:

```
$ curl localhost:8080/actuator/health
{"status":"UP"}
```

COPY

You can try also to invoke shutdown through curl, to see what happens when you have not added the necessary line (shown in the preceding note) to `application.properties`:

```
$ curl -X POST localhost:8080/actuator/shutdown
{"timestamp":1401820343710,"error":"Not
Found","status":404,"message":"","path":"/actuator/shutdown"}
```

COPY

Because we did not enable it, the requested endpoint is not available (because the endpoint does not exist).

For more details about each of these REST endpoints and how you can tune their settings with an `application.properties` file (in `src/main/resources`), see the the documentation about the endpoints.

View Spring Boot's Starters

You have seen some of Spring Boot's "starters". You can see them all here in source code.

JAR Support and Groovy Support

The last example showed how Spring Boot lets you wire beans that you may not be aware you need. It also showed how to turn on convenient management services.

However, Spring Boot does more than that. It supports not only traditional WAR file deployments but also lets you put together executable JARs, thanks to Spring Boot's loader

module. The various guides demonstrate this dual support through the `spring-boot-gradle-plugin` and `spring-boot-maven-plugin`.

On top of that, Spring Boot also has Groovy support, letting you build Spring MVC web applications with as little as a single file.

Create a new file called `app.groovy` and put the following code in it:

```
@RestController
class ThisWillActuallyRun {

    @GetMapping("/")
    String home() {
        return "Hello, World!"
    }

}
```

COPY

> It does not matter where the file is. You can even fit an application that small inside a single tweet!

Next, install Spring Boot's CLI.

Run the Groovy application by running the following command:

```
$ spring run app.groovy
```

COPY

> Shut down the previous application, to avoid a port collision.

From a different terminal window, run the following curl command (shown with its output):

```
$ curl localhost:8080
Hello, World!
```

COPY

Spring Boot does this by dynamically adding key annotations to your code and using Groovy Grape to pull down the libraries that are needed to make the app run.

Summary

Congratulations! You built a simple web application with Spring Boot and learned how it can ramp up your development pace. You also turned on some handy production services. This is only a small sampling of what Spring Boot can do. See Spring Boot's online docs for much more information.

See Also

The following guides may also be helpful:

- Securing a Web Application
- Serving Web Content with Spring MVC

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