Building an Application with Spring Boot

Spring Boot如何帮助你加速和促进应用程序开发

This guide provides a sampling of how Spring Boot helps you accelerate and facilitate application development. As you read more Spring Getting Started guides, you will see more use cases for Spring Boot. It 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 you can download either a Maven build file, or a bundled up project as a zip file.

What you'll build

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

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

如何完成本指南

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.

从头开始并完成每个步骤，或者绕过你熟悉的基本设置步骤

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-spring-boot.git`
- cd into `gs-spring-boot/initial`
- Jump ahead to [initial].

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

Learn what you can do with Spring Boot

1、了解你能用Spring Boot做什么

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

Spring Boot提供一种快速构建应用程序的方法。
它通过查看你的类路径和配置的Beans，对你缺少的东西做出合理的假设，并添加它。
使用Spring Boot，你可以更专注于业务功能，更少关注基础架构。

For example:

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

Spring Boot提供的自动配置

These are just a few examples of the automatic configuration Spring Boot provides. At the same time, Spring Boot doesn't get in your way. For example, if Thymeleaf is on your path, Spring Boot adds a `SpringTemplateEngine` to your application context automatically. But if you define your own `SpringTemplateEngine` with your own settings, then Spring Boot won't add one. This leaves you in control with little effort on your part.

自定义配置

> Spring Boot doesn't generate code or make edits to your files. Instead, when you start up your application, Spring Boot dynamically wires up beans and settings and applies them to your application context.
>
> 当你启动应用程序时，Spring Boot会动态地连接Beans和配置，并将应用于你的应用程序上下文。

Create a simple web application 2、创建一个简单的Web应用程序

Now you can create a web controller for a simple web application.

`src/main/java/hello/HelloController.java`

```java
package hello;

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

@RestController
public class HelloController {

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

}
```

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

Create an Application class 3、创建应用入口类

Here you create an `Application` class with the components:

`src/main/java/hello/Application.java`

```java
package hello;

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 {

    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.
- 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.

启动应用程序

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. Web应用程序是100%的纯Java

在启动时运行

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

Run the application 4、运行应用程序

To run the application, execute:

```
./gradlew build && java -jar build/libs/gs-spring-boot-0.1.0.jar
```

If you are using Maven, execute:

```
mvn package && java -jar target/gs-spring-boot-0.1.0.jar
```

You should see some output like this:

```
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.EmbeddedServletContainerAutoConfiguration
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration$DispatcherServletConfiguration
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration$EmbeddedTomcat
org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration
org.springframework.boot.context.embedded.properties.ServerProperties
org.springframework.context.annotation.ConfigurationClassPostProcessor.enhancedConfigurationProcessor
org.springframework.context.annotation.ConfigurationClassPostProcessor.importAwareProcessor
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`.

Check out the service.

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

Add Unit Tests 5、添加单元测试

You will want to add a test for the endpoint you added, and Spring Test already provides some machinery for that, and it's easy to include in your project.

Add this to your build file's list of dependencies:

```
    testCompile("org.springframework.boot:spring-boot-starter-test")
```

If you are using Maven, add this to your list of dependencies:

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

Now write a simple unit test that mocks the servlet request and response through your endpoint:

`src/test/java/hello/HelloControllerTest.java`

```
package hello;

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

import org.junit.Test;
import org.junit.runner.RunWith;
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.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

@RunWith(SpringRunner.class)
@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!")));
    }
}
```

The `MockMvc` comes from Spring Test and allows you, via a set of convenient builder classes, to send HTTP requests into the `DispatcherServlet` and make assertions about the result. Note the use of the `@AutoConfigureMockMvc` together with `@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 using the `@WebMvcTest`. Spring Boot automatically tries to locate the main application class of your application in either case, but you can override it, or narrow it down, if you want to build something different.

As well as mocking the HTTP request cycle we can also use Spring Boot to write a very simple full-stack integration test. For example, instead of (or as well as) the mock test above we could do this:

`src/test/java/hello/HelloControllerIT.java`

```
package hello;

import static org.hamcrest.Matchers.equalTo;
import static org.junit.Assert.assertThat;

import java.net.URL;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.ResponseEntity;
import org.springframework.test.context.junit4.SpringRunner;

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

    @LocalServerPort
    private int port;

    private URL base;

    @Autowired
    private TestRestTemplate template;

    @Before
    public void setUp() throws Exception {
        this.base = new URL("http://localhost:" + port + "/");
    }

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

The embedded server is started up on a random port by virtue of the `webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT` and the actual port is discovered at runtime with the `@LocalServerPort`.

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 out of the box with its actuator module, such as health, audits, beans, and more.

Add this to your build file's list of dependencies:

```
    compile("org.springframework.boot:spring-boot-starter-actuator")
```

If you are using Maven, add this to your list of dependencies:

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

Then restart the app:

```
./gradlew build && java -jar build/libs/gs-spring-boot-0.1.0.jar
```

If you are using Maven, execute:

```
mvn package && java -jar target/gs-spring-boot-0.1.0.jar
```

You will see a new set of RESTful end points added to the application. These are management services provided by Spring Boot.

```
2014-06-03 13:23:28.119  ... : Mapped "{[/error],methods=[],params=[],headers=[],consumes...
2014-06-03 13:23:28.119  ... : Mapped "{[/error],methods=[],params=[],headers=[],consumes...
2014-06-03 13:23:28.136  ... : Mapped URL path [/**] onto handler of type [class org.spri...
2014-06-03 13:23:28.136  ... : Mapped URL path [/webjars/**] onto handler of type [class ...
2014-06-03 13:23:28.440  ... : Mapped "{[/info],methods=[GET],params=[],headers=[],consum...
2014-06-03 13:23:28.441  ... : Mapped "{[/autoconfig],methods=[GET],params=[],headers=[],...
2014-06-03 13:23:28.441  ... : Mapped "{[/mappings],methods=[GET],params=[],headers=[],co...
2014-06-03 13:23:28.442  ... : Mapped "{[/trace],methods=[GET],params=[],headers=[],consu...
2014-06-03 13:23:28.442  ... : Mapped "{[/env/{name:.*}],methods=[GET],params=[],headers=...
2014-06-03 13:23:28.442  ... : Mapped "{[/env],methods=[GET],params=[],headers=[],consume...
2014-06-03 13:23:28.443  ... : Mapped "{[/configprops],methods=[GET],params=[],headers=[]...
2014-06-03 13:23:28.443  ... : Mapped "{[/metrics/{name:.*}],methods=[GET],params=[],head...
2014-06-03 13:23:28.443  ... : Mapped "{[/metrics],methods=[GET],params=[],headers=[],con...
2014-06-03 13:23:28.444  ... : Mapped "{[/health],methods=[GET],params=[],headers=[],cons...
2014-06-03 13:23:28.444  ... : Mapped "{[/dump],methods=[GET],params=[],headers=[],consum...
2014-06-03 13:23:28.445  ... : Mapped "{[/beans],methods=[GET],params=[],headers=[],consu...
```

They include: errors, environment, health, beans, info, metrics, trace, configprops, and dump.

There is also a `/shutdown` endpoint, but it's only visible by default via JMX. To enable it as an HTTP endpoint, add `endpoints.shutdown.enabled=true` to your `application.properties` file.

It's easy to check the health of the app.

```
$ curl localhost:8080/health
{"status":"UP","diskSpace":{"status":"UP","total":397635555328,"free":328389529600,"threshold":10485760}}
```

You can try to invoke shutdown through curl.

```
$ curl -X POST localhost:8080/shutdown
{"timestamp":1401820343710,"error":"Method Not Allowed","status":405,"message":"Request method 'POST' not supported"}
```

Because we didn't enable it, the request is blocked by the virtue of not existing.

For more details about each of these REST points and how you can tune their settings with an `application.properties` file, you can read detailed docs 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 makes it easy to wire beans you may not be aware that you need. And it showed how to turn on convenient management services.

But Spring Boot does yet more. It supports not only traditional WAR file deployments, but also makes it easy to 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, allowing you to build Spring MVC web apps with as little as a single file.

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

```
@RestController
class ThisWillActuallyRun {

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

}
```

It doesn't matter where the file is. You can even fit an application that small inside a single tweet!

Next, install Spring Boot's CLI.

Run it as follows:

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

This assumes you shut down the previous application, to avoid a port collision.

From a different terminal window:

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

Spring Boot does this by dynamically adding key annotations to your code and using Groovy Grape to pull down libraries 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. Checkout Spring Boot's online docs if you want to dig deeper.

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