Testing the Web Layer

This guide walks you through the process of creating a Spring application.

What you'll build

You'll build a simple Spring application and test it with JUnit. You probably already know how to write and run unit tests of the individual classes in your application, so for this guide we will concentrate on using Spring Test and Spring Boot features to test the interactions between Spring and your code. You will start with a simple test that the application context loads successfully, and continue on to test just the web layer using Spring's `MockMvc`.

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-testing-web.git`
- cd into `gs-testing-web/initial`
- Jump ahead to Create a simple application.

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

Create a simple application

Create a new controller for your Spring application:

`src/main/java/hello/HomeController.java`

```
package hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
```

```
@Controller
public class HomeController {

    @RequestMapping("/")
    public @ResponseBody String greeting() {
        return "Hello World";
    }

}
```

> 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.

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.

`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 `HelloController`.

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-testing-web-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-testing-web-0.1.0.jar
```

> The procedure above will create a runnable JAR. You can also opt to build a classic WAR file instead.

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

Test the application

Now that the application is running, you can test it. If it is running you can load the home page on http://localhost:8080. But to give yourself more confidence that the application is working when you make changes, you want to automate the testing.

The first thing you can do is write a simple sanity check test that will fail if the application context cannot start. First add Spring Test as a dependency to your pom.xml, in the test scope. If you are using Maven:

`pom.xml`

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

or if you are using Gradle:

`build.gradle`

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

Then create a test case with the `@RunWith` and `@SpringBootTest` annotations and an empty test method:

`src/test/java/hello/ApplicationTest.java`

```
package hello;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationTest {

    @Test
    public void contextLoads() throws Exception {
    }

}
```

The `@SpringBootTest` annotation tells Spring Boot to go and look for a main configuration class (one with `@SpringBootApplication` for instance), and use that to start a Spring application context. You can run this test in your IDE or on the command line (`mvn test` or `gradle test`) and it should pass. To convince yourself that the context is creating your controller you could add an assertion:

`src/test/java/hello/SmokeTest.java`

```
package hello;

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

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SmokeTest {

    @Autowired
    private HomeController controller;

    @Test
    public void contexLoads() throws Exception {
        assertThat(controller).isNotNull();
    }
}
```

The `@Autowired` annotation is interpreted by the Spring and the controller is injected before the test methods

are run. We are using AssertJ (`assertThat()` etc.) to express the test assertions.

> A nice feature of the Spring Test support is that the application context is cached in between tests, so if you have multiple methods in a test case, or multiple test cases with the same configuration, they only incur the cost of starting the application once. You can control the cache using the `@DirtiesContext` annotation.

It's nice to have a sanity check like that, but we should also write some tests that assert the behaviour of our application. To do that we could start the application up and listen for a connection like it would do in production, and then send an HTTP request and assert the response.

`src/test/java/hello/HttpRequestTest.java`

```
package hello;

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

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.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

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

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void greetingShouldReturnDefaultMessage() throws Exception {
        assertThat(this.restTemplate.getForObject("http://localhost:" + port + "/",
                String.class)).contains("Hello World");
    }
}
```

Note the use of `webEnvironment=RANDOM_PORT` to start the server with a random port (useful to avoid conflicts in test environments), and the injection of the port with `@LocalServerPort`. Also note that Spring Boot has provided a `TestRestTemplate` for you automatically, and all you have to do is `@Autowired` it.

Another useful approach is to not start the server at all, but test only the layer below that, where Spring handles the incoming HTTP request and hands it off to your controller. That way, almost the full stack is used, and your code will be called exactly the same way as if it was processing a real HTTP request, but without the cost of

starting the server. To do that we will use Spring's `MockMvc`, and we can ask for that to be injected for us by using the `@AutoConfigureMockMvc` annotation on the test case:

`src/test/java/hello/ApplicationTest.java`

```
package hello;

import static org.hamcrest.Matchers.containsString;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
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.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class ApplicationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void shouldReturnDefaultMessage() throws Exception {
        this.mockMvc.perform(get("/")).andDo(print()).andExpect(status().isOk())
                .andExpect(content().string(containsString("Hello World")));
    }
}
```

In this test, the full Spring application context is started, but without the server. We can narrow down the tests to just the web layer by using `@WebMvcTest`:

`src/test/java/hello/WebLayerTest.java`

```
@RunWith(SpringRunner.class)
@WebMvcTest
public class WebLayerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void shouldReturnDefaultMessage() throws Exception {
        this.mockMvc.perform(get("/")).andDo(print()).andExpect(status().isOk())
                .andExpect(content().string(containsString("Hello World")));
    }
```

```
}
```

The test assertion is the same as in the previous case, but here Spring Boot is only instantiating the web layer, not the whole context. In an application with multiple controllers you can even ask for just one to be instantiated, using, for example `@WebMvcTest(HomeController.class)`

So far, our `HomeController` is very simple and has no dependencies. We could make it more realistic by introducing an extra component to store the greeting. E.g. in a new controller:

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

```
package hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;


@Controller
public class GreetingController {

    private final GreetingService service;

    public GreetingController(GreetingService service) {
        this.service = service;
    }

    @RequestMapping("/greeting")
    public @ResponseBody String greeting() {
        return service.greet();
    }

}
```

and then

`src/main/java/hello/GreetingService.java`

```
package hello;

import org.springframework.stereotype.Service;

@Service
public class GreetingService {
    public String greet() {
        return "Hello World";
    }
}
```

The service dependency will be automatically injected by Spring into the controller (because of the constructor signature). To test this controller with `@WebMvcTest` you can do this

`src/test/java/hello/WebMockTest.java`

```
package hello;

import static org.hamcrest.Matchers.containsString;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
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.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

@RunWith(SpringRunner.class)
@WebMvcTest(GreetingController.class)
public class WebMockTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private GreetingService service;

    @Test
    public void greetingShouldReturnMessageFromService() throws Exception {
        when(service.greet()).thenReturn("Hello Mock");
        this.mockMvc.perform(get("/greeting")).andDo(print()).andExpect(status().isOk())
                .andExpect(content().string(containsString("Hello Mock")));
    }
}
```

We use `@MockBean` to create and inject a mock for the `GreetingService` (if you don't do this the application context cannot start), and we set its expectations using `Mockito`.

Summary

Congratulations! You've just developed a Spring application and tested it with JUnit and Spring `MockMvc` using Spring Boot to isolate the web layer and load a special application context.

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