Testing the Web Layer

Get the Code
Go To Repo

This guide walks you through the process of creating a Spring application and then testing it with JUnit.

What You Will Build

You will 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 only the web layer by using Spring's `MockMvc`.

What You Need

- About 15 minutes
- A favorite text editor or IDE
- Java 1.8 or later
- Gradle 7.5+ or Maven 3.5+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA
  - VSCode

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

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

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 Application

Create a new controller for your Spring application. The following listing (from `src/main/java/com/example/testingweb/HomeController.java`) shows how to do so:

COPY

```
package com.example.testingweb;

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 preceding example does not specify `GET` versus `PUT`, `POST`, and so forth. By default `@RequestMapping` maps all HTTP operations. You can use `@GetMapping` or `@RequestMapping(method=GET)` to narrow this mapping.

Run the Application

The Spring Initializr creates an application class (a class with a `main()` method) for you. For this guide, you need not modify this class. The following listing (from `src/main/java/com/example/testingweb/TestingWebApplication.java`) shows the application class that the Spring Initializr created:

COPY

```
package com.example.testingweb;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TestingWebApplication {

	public static void main(String[] args) {
		SpringApplication.run(TestingWebApplication.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.
- `@EnableWebMvc`: Flags the application as a web application and activates key behaviors, such as setting up a `DispatcherServlet`. Spring Boot adds it automatically when it sees `spring-webmvc` on the classpath.
- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the package where your annotated `TestingWebApplication` class resides (`com.example.testingweb`), letting it find the `com.example.testingweb.HelloController`.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there is 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. Spring Boot handles all of that for you.

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. You can load the home page at `http://localhost:8080`. However, to give yourself more confidence that the application works when you make changes, you want to automate the testing.

> Spring Boot assumes you plan to test your application, so it adds the necessary dependencies to your build file (`build.gradle` or `pom.xml`).

The first thing you can do is write a simple sanity check test that will fail if the application context cannot start. The following listing (from `src/test/java/com/example/testingweb/TestingWebApplicationTest.java`) shows how to do so:

COPY

```
package com.example.testingweb;

import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class TestingWebApplicationTests {

	@Test
	void contextLoads() {
	}

}
```

The `@SpringBootTest` annotation tells Spring Boot to 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 (by running `./mvnw test` or `./gradlew test`), and it should pass. To convince yourself that the context is creating your controller, you could add an assertion, as the following example (from `src/test/java/com/example/testingweb/SmokeTest.java`) shows:

COPY

```
package com.example.testingweb;

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

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class SmokeTest {

	@Autowired
	private HomeController controller;

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

Spring interprets the `@Autowired` annotation, and the controller is injected before the test methods are run. We use AssertJ (which provides `assertThat()` and other methods) to express the test assertions.

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

It is nice to have a sanity check, but you should also write some tests that assert the behavior of your application. To do that, you could start the application and listen for a connection (as it would do in production) and then send an HTTP request and assert the response. The following listing (from `src/test/java/com/example/testingweb/HttpRequestTest.java`) shows how to do so:

COPY

```
package com.example.testingweb;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
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.beans.factory.annotation.Value;
import org.springframework.boot.test.web.server.LocalServerPort;

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

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

	@LocalServerPort
	private int port;

	@Autowired
	private TestRestTemplate restTemplate;

	@Test
	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 automatically provided a `TestRestTemplate` for you. All you have to do is add `@Autowired` to it.

Another useful approach is to not start the server at all but to test only the layer below that, where Spring handles the incoming HTTP request and hands it off to your controller. That way, almost all of the full stack is used, and your code will be called in exactly the same way as if it were processing a real HTTP request but without the cost of starting the server. To do that, use Spring's `MockMvc` and ask for that to be injected for you by using the `@AutoConfigureMockMvc` annotation on the test case. The following listing (from `src/test/java/com/example/testingweb/TestingWebApplicationTest.java`) shows how to do so:

COPY

```
package com.example.testingweb;

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.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.test.web.servlet.MockMvc;

@SpringBootTest
@AutoConfigureMockMvc
class TestingWebApplicationTest {

	@Autowired
	private MockMvc mockMvc;

	@Test
	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 the tests to only the web layer by using `@WebMvcTest`, as the following listing (from `src/test/java/com/example/testingweb/WebLayerTest.java`) shows:

COPY

```
@WebMvcTest
include::complete/src/test/java/com/example/testingweb/WebLayerTest.java
```

The test assertion is the same as in the previous case. However, in this test, Spring Boot instantiates only the web layer rather than the whole context. In an application with multiple controllers, you can even ask for only one to be instantiated by using, for example, `@WebMvcTest(HomeController.class)`.

So far, our `HomeController` is simple and has no dependencies. We could make it more realistic by introducing an extra component to store the greeting (perhaps in a new controller). The following example (from `src/main/java/com/example/testingweb/GreetingController.java`) shows how to do so:

COPY

```
package com.example.testingweb;

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();
	}

}
```

Then create a greeting service, as the following listing (from `src/main/java/com/example/testingweb/GreetingService.java`) shows:

COPY

```
package com.example.testingweb;

import org.springframework.stereotype.Service;

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

Spring automatically injects the service dependency into the controller (because of the constructor signature). The following listing (from `src/test/java/com/example/testingweb/WebMockTest.java`) shows how to test this controller with `@WebMvcTest`:

COPY

```
package com.example.testingweb;

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.jupiter.api.Test;

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.web.servlet.MockMvc;

@WebMvcTest(GreetingController.class)
class WebMockTest {

	@Autowired
	private MockMvc mockMvc;

	@MockBean
	private GreetingService service;

	@Test
	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 do not do so, the application context cannot start), and we set its expectations using `Mockito`.

Summary

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

See Also

The following guides may also be helpful:

- Building an Application with Spring Boot
- Serving Web Content with Spring MVC

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