



# Spring Boot IN ACTION

Craig Walls

FOREWORD BY Andrew Glover

SAMPLE CHAPTER

 MANNING



# ***Spring Boot in Action***

by Craig Walls

## **Chapter 4**

Copyright 2016 Manning Publications

## *brief content*

---

- 1 ■ Bootstarting Spring 1
- 2 ■ Developing your first Spring Boot application 23
- 3 ■ Customizing configuration 49
- 4 ■ Testing with Spring Boot 76
- 5 ■ Getting Groovy with the Spring Boot CLI 92
- 6 ■ Applying Grails in Spring Boot 107
- 7 ■ Taking a peek inside with the Actuator 124
- 8 ■ Deploying Spring Boot applications 160
  
- appendix A ■ Spring Boot Developer Tools 181
- appendix B ■ Spring Boot starters 188
- appendix C ■ Configuration properties 195
- appendix D ■ Spring Boot dependencies 232

# Testing with Spring Boot



## ***This chapter covers***

- Integration testing
- Testing apps in a server
- Spring Boot's test utilities

It's been said that if you don't know where you're going, any road will get you there. But with software development, if you don't know where you're going, you'll likely end up with a buggy application that nobody can use.

The best way to know for sure where you're going when writing applications is to write tests that assert the desired behavior of an application. If those tests fail, you know you have some work to do. If they pass, then you've arrived (at least until you think of some more tests that you can write).

Whether you write tests first or after the code has already been written, it's important that you write tests to not only verify the accuracy of your code, but to also to make sure it does everything you expect it to. Tests are also a great safeguard to make sure that things don't break as your application continues to evolve.

When it comes to writing unit tests, Spring is generally out of the picture. Loose coupling and interface-driven design, which Spring encourages, makes it really easy to write unit tests. But Spring isn't necessarily involved in those unit tests.

Integration tests, on the other hand, require some help from Spring. If Spring is responsible for configuring and wiring up the components in your production application, then Spring should also be responsible for configuring and wiring up those components in your tests.

Spring's `SpringJUnit4ClassRunner` helps load a Spring application context in JUnit-based application tests. Spring Boot builds on Spring's integration testing support by enabling auto-configuration and web server startup when testing Spring Boot applications. It also offers a handful of useful testing utilities.

In this chapter, we'll look at all of the ways that Spring Boot supports integration testing. We'll start by looking at how to test with a fully Spring Boot-enabled application context.

## 4.1 Integration testing auto-configuration

At the core of everything that the Spring Framework does, its most essential task is to wire together all of the components that make up an application. It does this by reading a wiring specification (whether it be XML, Java-based, Groovy-based, or otherwise), instantiating beans in an application context, and injecting beans into other beans that depend on them.

When integration testing a Spring application, it's important to let Spring wire up the beans that are the target of the test the same way it wires up those beans when the application is running in production. Sure, you might be able to manually instantiate the components and inject them into each other, but for any substantially big application, that can be an arduous task. Moreover, Spring offers additional facilities such as component-scanning, autowiring, and declarative aspects such as caching, transactions, and security. Given all that would be required to recreate what Spring does, it's generally best to let Spring do the heavy lifting, even in an integration test.

Spring has offered excellent support for integration testing since version 1.1.1. Since Spring 2.5, integration testing support has been offered in the form of `SpringJUnit4ClassRunner`, a JUnit class runner that loads a Spring application context for use in a JUnit test and enables autowiring of beans into the test class.

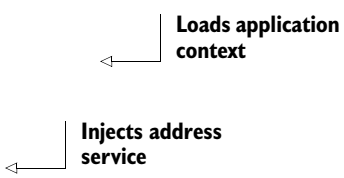
For example, consider the following listing, which shows a very basic Spring integration test.

### Listing 4.1 Integration testing Spring with `SpringJUnit4ClassRunner`

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    classes=AddressBookConfiguration.class)
public class AddressServiceTests {

    @Autowired
    private AddressService addressService;

    @Test
```



```

public void testService() {
    Address address = addressService.findByName("Sheman");
    assertEquals("P", address.getFirstName());
    assertEquals("Sherman", address.getLastName());
    assertEquals("42 Wallaby Way", address.getAddressLine1());
    assertEquals("Sydney", address.getCity());
    assertEquals("New South Wales", address.getState());
    assertEquals("2000", address.getPostCode());
}
}

```

← Tests address service

As you can see, `AddressServiceTests` is annotated with both `@RunWith` and `@ContextConfiguration`. `@RunWith` is given `SpringJUnit4ClassRunner.class` to enable Spring integration testing.<sup>1</sup> Meanwhile, `@ContextConfiguration` specifies how to load the application context. Here we're asking it to load the Spring application context given the specification defined in `AddressBookConfiguration`.

In addition to loading the application context, `SpringJUnit4ClassRunner` also makes it possible to inject beans from the application context into the test itself via autowiring. Because this test is targeting an `AddressService` bean, it is autowired into the test. Finally, the `testService()` method makes calls to the address service and verifies the results.

Although `@ContextConfiguration` does a great job of loading the Spring application context, it doesn't load it with the full Spring Boot treatment. Spring Boot applications are ultimately loaded by `SpringApplication`, either explicitly (as in listing 2.1) or using `SpringBootServletInitializer` (which we'll look at in chapter 8). `SpringApplication` not only loads the application context, but also enables logging, the loading of external properties (application.properties or application.yml), and other features of Spring Boot. If you're using `@ContextConfiguration`, you won't get those features.

To get those features back in your integration tests, you can swap out `@ContextConfiguration` for Spring Boot's `@SpringApplicationConfiguration`:

```

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
    classes=AddressBookConfiguration.class)
public class AddressServiceTests {
    ...
}

```

The use of `@SpringApplicationConfiguration` is largely identical to `@ContextConfiguration`. But unlike `@ContextConfiguration`, `@SpringApplicationConfiguration` loads the Spring application context using `SpringApplication` the same way and with the same treatment it would get if it was being loaded in a production application. This includes the loading of external properties and Spring Boot logging.

<sup>1</sup> As of Spring 4.2, you can optionally use `SpringClassRule` and `SpringMethodRule` as JUnit rule-based alternatives to `SpringJUnit4ClassRunner`.

Suffice it to say that, for the most part, `@SpringApplicationConfiguration` replaces `@ContextConfiguration` when writing tests for Spring Boot applications. We'll certainly use `@SpringApplicationConfiguration` throughout this chapter as we write tests for our Spring Boot application, including tests that target the web front end of the application.

Speaking of web testing, that's what we're going to do next.

## 4.2 Testing web applications

One of the nice things about Spring MVC is that it promotes a programming model around plain old Java objects (POJOs) that are annotated to declare how they should process web requests. This programming model is not only simple, it enables you to treat controllers just as you would any other component in your application. You might even be tempted to write tests against your controller that test them as POJOs.

For instance, consider the `addToReadingList()` method from `ReadingListController`:

```
@RequestMapping(method=RequestMethod.POST)
public String addToReadingList(Book book) {
    book.setReader(reader);
    readingListRepository.save(book);
    return "redirect:/readingList";
}
```

If you were to disregard the `@RequestMapping` method, you'd be left with a rather basic Java method. It wouldn't take much to imagine a test that provides a mock implementation of `ReadingListRepository`, calls `addToReadingList()` directly, and asserts the return value and verifies the call to the repository's `save()` method.

The problem with such a test is that it only tests the method itself. While that's better than no test at all, it fails to test that the method handles a POST request to `/readingList`. It also fails to test that form fields are properly bound to the `Book` parameter. And although you could assert that the returned `String` contains a certain value, it would be impossible to test definitively that the request is, in fact, redirected to `/readingList` after the method is finished.

To properly test a web application, you need a way to throw actual HTTP requests at it and assert that it processes those requests correctly. Fortunately, there are two options available to Spring Boot application developers that make those kinds of tests possible:

- *Spring Mock MVC*—Enables controllers to be tested in a mocked approximation of a servlet container without actually starting an application server
- *Web integration tests*—Actually starts the application in an embedded servlet container (such as Tomcat or Jetty), enabling tests that exercise the application in a real application server

Each of these kinds of tests has its share of pros and cons. Obviously, starting a server will result in a slower test than mocking a servlet container. But there's no doubt that

server-based tests are closer to the real-world environment that they'll be running in when deployed to production.

We're going to start by looking at how you can test a web application using Spring's Mock MVC test framework. Then, in section 4.3, you'll see how to write tests against an application that's actually running in an application server.

#### 4.2.1 *Mocking Spring MVC*

Since Spring 3.2, the Spring Framework has had a very useful facility for testing web applications by mocking Spring MVC. This makes it possible to perform HTTP requests against a controller without running the controller within an actual servlet container. Instead, Spring's Mock MVC framework mocks enough of Spring MVC to make it almost as though the application is running within a servlet container ... but it's not.

To set up a Mock MVC in your test, you can use `MockMvcBuilders`. This class offers two static methods:

- `standaloneSetup()`—Builds a Mock MVC to serve one or more manually created and configured controllers
- `webApplicationContextSetup()`—Builds a Mock MVC using a Spring application context, which presumably includes one or more configured controllers

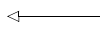
The primary difference between these two options is that `standaloneSetup()` expects you to manually instantiate and inject the controllers you want to test, whereas `webApplicationContextSetup()` works from an instance of `WebApplicationContext`, which itself was probably loaded by Spring. The former is slightly more akin to a unit test in that you'll likely only use it for very focused tests around a single controller. The latter, however, lets Spring load your controllers as well as their dependencies for a full-blown integration test.

For our purposes, we're going to use `webApplicationContextSetup()` so that we can test the `ReadingListController` as it has been instantiated and injected from the application context that Spring Boot has auto-configured.

The `webApplicationContextSetup()` takes a `WebApplicationContext` as an argument. Therefore, we'll need to annotate the test class with `@WebAppConfiguration` and use `@Autowired` to inject the `WebApplicationContext` into the test as an instance variable. The following listing shows the starting point for our Mock MVC tests.

#### Listing 4.2 Creating a Mock MVC for integration testing controllers

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
    classes = ReadingListApplication.class)
@WebAppConfiguration
public class MockMvcWebTests {
```



Enables web context testing



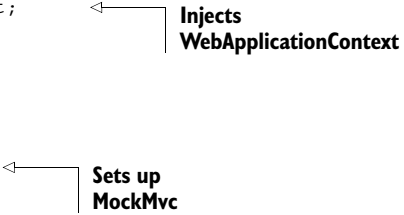
```

@Autowired
private WebApplicationContext webContext;

private MockMvc mockMvc;

@Before
public void setupMockMvc() {
    mockMvc = MockMvcBuilders
        .webAppContextSetup(webContext)
        .build();
}

```



Injects **WebApplicationContext**

Sets up **MockMvc**

The `@WebAppConfiguration` annotation declares that the application context created by `SpringJUnit4ClassRunner` should be a `WebApplicationContext` (as opposed to a basic non-web `ApplicationContext`).

The `setupMockMvc()` method is annotated with JUnit's `@Before`, indicating that it should be executed before any test methods. It passes the injected `WebApplicationContext` into the `webAppContextSetup()` method and then calls `build()` to produce a `MockMvc` instance, which is assigned to an instance variable for test methods to use.

Now that we have a `MockMvc`, we're ready to write some test methods. Let's start with a simple test method that performs an HTTP GET request against `/readingList` and asserts that the model and view meet our expectations. The following `homePage()` test method does what we need:

```

@Test
public void homePage() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get("/readingList"))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.view().name("readingList"))
        .andExpect(MockMvcResultMatchers.model().attributeExists("books"))
        .andExpect(MockMvcResultMatchers.model().attribute("books",
            Matchers.is(Matchers.empty())));
}

```

As you can see, a lot of static methods are being used in this test method, including static methods from Spring's `MockMvcRequestBuilders` and `MockMvcResultMatchers`, as well as from the Hamcrest library's `Matchers`. Before we dive into the details of this test method, let's add a few static imports so that the code is easier to read:

```

import static org.hamcrest.Matchers.*;
import static org.springframework.test.web.servlet.request.
    ➤ MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.
    ➤ MockMvcResultMatchers.*;

```

With those static imports in place, the test method can be rewritten like this:

```

@Test
public void homePage() throws Exception {
    mockMvc.perform(get("/readingList"))
        .andExpect(status().isOk())
        .andExpect(view().name("readingList"))
        .andExpect(model().attributeExists("books"))
        .andExpect(model().attribute("books", isEmpty()));
}

```

Now the test method almost reads naturally. First it performs a GET request against `/readingList`. Then it expects that the request is successful (`isOk()` asserts an HTTP 200 response code) and that the view has a logical name of `readingList`. It also asserts that the model contains an attribute named `books`, but that attribute is an empty collection. It's all very straightforward.

The main thing to note here is that at no time is the application deployed to a web server. Instead it's run within a mocked out Spring MVC, just capable enough to handle the HTTP requests we throw at it via the `MockMvc` instance.

Pretty cool, huh?

Let's try one more test method. This time we'll make it a bit more interesting by actually sending an HTTP POST request to post a new book. We should expect that after the POST request is handled, the request will be redirected back to `/readingList` and that the `books` attribute in the model will contain the newly added book. The following listing shows how we can use Spring's Mock MVC to do this kind of test.

#### Listing 4.3 Testing the post of a new book

```

@Test
public void postBook() throws Exception {
    mockMvc.perform(post("/readingList")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("title", "BOOK TITLE")
        .param("author", "BOOK AUTHOR")
        .param("isbn", "1234567890")
        .param("description", "DESCRIPTION")
        .andExpect(status().is3xxRedirection())
        .andExpect(header().string("Location", "/readingList")));
}

Book expectedBook = new Book();
expectedBook.setId(1L);
expectedBook.setReader("craig");
expectedBook.setTitle("BOOK TITLE");
expectedBook.setAuthor("BOOK AUTHOR");
expectedBook.setIsbn("1234567890");
expectedBook.setDescription("DESCRIPTION");

mockMvc.perform(get("/readingList"))
    .andExpect(status().isOk())
    .andExpect(view().name("readingList"));

```

Performs POST request

Sets up expected book

Performs GET request

```

        .andExpect(model().attributeExists("books"))
        .andExpect(model().attribute("books", hasSize(1)))
        .andExpect(model().attribute("books",
            contains(samePropertyValuesAs(expectedBook))));
    }

```

Obviously, the test in listing 4.3 is a bit more involved. It's actually two tests in one method. The first part posts the book and asserts the results from that request. The second part performs a fresh GET request against the home page and asserts that the newly created book is in the model.

When posting the book, we must make sure we set the content type to “application/x-www-form-urlencoded” (with `MediaType.APPLICATION_FORM_URLENCODED`) as that will be the content type that a browser will send when the book is posted in the running application. We then use the `MockMvcRequestBuilders`'s `param()` method to set the fields that simulate the form being submitted. Once the request has been performed, we assert that the response is a redirect to `/readingList`.

Assuming that much of the test method passes, we move on to part two. First, we set up a `Book` object that contains the expected values. We'll use this to compare with the value that's in the model after fetching the home page.

Then we perform a GET request for `/readingList`. For the most part, this is no different than how we tested the home page before, except that instead of an empty collection in the model, we're checking that it has one item, and that the item is the same as the expected `Book` we created. If so, then our controller seems to be doing its job of saving a book when one is posted to it.

So far, these tests have assumed an unsecured application, much like the one we wrote in chapter 2. But what if we want to test a secured application, such as the one from chapter 3?

#### 4.2.2 Testing web security

Spring Security offers support for testing secured web applications easily. In order to take advantage of it, you must add Spring Security's test module to your build. The following `testCompile` dependency in Gradle is all you need:

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

Or if you're using Maven, add the following `<dependency>` to your build:

```

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-test</artifactId>
  <scope>test</scope>
</dependency>

```

With Spring Security's test module in your application's classpath, you just need to apply the Spring Security configurer when creating the `MockMvc` instance:

```

@Before
public void setupMockMvc() {
    mockMvc = MockMvcBuilders
        .webApplicationContextSetup(webContext)
        .apply(springSecurity())
        .build();
}

```

The `springSecurity()` method returns a Mock MVC configurer that enables Spring Security for Mock MVC. By simply applying it as shown here, Spring Security will be in play on all requests performed through `MockMvc`. The specific security configuration will depend on how you've configured Spring Security (or how Spring Boot has auto-configured Spring Security). In the case of the reading-list application, it's the same security configuration we created in `SecurityConfig.java` in chapter 3.

**THE SPRINGSECURITY() METHOD** `springSecurity()` is a static method of `SecurityMockMvcConfigurers`, which I've statically imported for readability's sake.

With Spring Security enabled, we can no longer simply request the home page and expect an HTTP 200 response. If the request isn't authenticated, we should expect a redirect to the login page:

```

@Test
public void homePage_unauthenticatedUser() throws Exception {
    mockMvc.perform(get("/"))
        .andExpect(status().is3xxRedirection())
        .andExpect(header().string("Location",
            "http://localhost/login"));
}

```

But how can we perform an authenticated request? Spring Security offers two annotations that can help:

- **@WithMockUser**—Loads the security context with a `UserDetails` using the given username, password, and authorization
- **@WithUserDetails**—Loads the security context by looking up a `UserDetails` object for the given username

In both cases, Spring Security's security context is loaded with a `UserDetails` object that is to be used for the duration of the annotated test method. The `@WithMockUser` annotation is the most basic of the two. It allows you to explicitly declare a `UserDetails` to be loaded into the security context:

```

@Test
@WithMockUser(username="craig",
    password="password",
    roles="READER")
public void homePage_authenticatedUser() throws Exception {
    ...
}

```

As you can see, `@WithMockUser` bypasses the normal lookup of a `UserDetails` object and instead creates one with the values specified. For simple tests, this may be fine. But for our test, we need a `Reader` (which implements `UserDetails`) instead of the generic `UserDetails` that `@WithMockUser` creates. For that, we'll need `@WithUserDetails`.

The `@WithUserDetails` annotation uses the configured `UserDetailsService` to load the `UserDetails` object. As you'll recall from chapter 3, we configured a `UserDetailsService` bean that looks up and returns a `Reader` object for a given username. That's perfect! So we'll annotate our test method with `@WithUserDetails`, as shown in the following listing.

#### Listing 4.4 Testing a secured method with user authentication

```
@Test
@WithUserDetails("craig")
public void homePage_authenticatedUser() throws Exception {

    Reader expectedReader = new Reader();
    expectedReader.setUsername("craig");
    expectedReader.setPassword("password");
    expectedReader.setFullname("Craig Walls");

    mockMvc.perform(get("/"))
        .andExpect(status().isOk())
        .andExpect(view().name("readingList"))
        .andExpect(model().attribute("reader",
            samePropertyValuesAs(expectedReader)))
        .andExpect(model().attribute("books", hasSize(0)))
}
```

Uses "craig" user

Sets up expected Reader

Performs GET request

In listing 4.4, we use `@WithUserDetails` to declare that the "craig" user should be loaded into the security context for the duration of this test method. Knowing that the `Reader` will be placed into the model, the method starts by creating an expected `Reader` object that it can compare with the model later in the test. Then it performs the GET request and asserts the view name and model contents, including the model attribute with the name "reader".

Once again, no servlet container is started up to run these tests. Spring's Mock MVC takes the place of an actual servlet container. The benefit of this approach is that the test methods run faster because they don't have to wait for the server to start. Moreover, there's no need to fire up a web browser to post the form, so the test is simpler and faster.

On the other hand, it's not a complete test. It's better than simply calling the controller methods directly, but it doesn't truly exercise the application in a web browser and verify the rendered view. To do that, we'll need to start a real web server and hit it with a real web browser. Let's see how Spring Boot can help us start a real web server for our tests.

### 4.3 Testing a running application

When it comes to testing web applications, nothing beats the real thing. Firing up the application in a real server and hitting it with a real web browser is far more indicative of how it will behave in the hands of users than poking at it with a mock testing engine.

But real tests in real servers with real web browsers can be tricky. Although there are build-time plugins for deploying applications in Tomcat or Jetty, they are clunky to set up. Moreover, it's nearly impossible to run any one of a suite of many such tests in isolation or without starting up your build tool.

Spring Boot, however, has a solution. Because Spring Boot already supports running embedded servlet containers such as Tomcat or Jetty as part of the running application, it stands to reason that the same mechanism could be used to start up the application along with its embedded servlet container for the duration of a test.

That's exactly what Spring Boot's `@WebIntegrationTest` annotation does. By annotating a test class with `@WebIntegrationTest`, you declare that you want Spring Boot to not only create an application context for your test, but also to start an embedded servlet container. Once the application is running along with the embedded container, you can issue real HTTP requests against it and make assertions against the results.

For example, consider the simple web test in listing 4.5, which uses `@WebIntegrationTest` to start the application along with a server and uses Spring's `RestTemplate` to perform HTTP requests against the application.

**Listing 4.5** Testing a web application in-server

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
    classes=ReadingListApplication.class)
@WebIntegrationTest
public class SimpleWebTest {

    @Test(expected=HttpClientErrorException.class)
    public void pageNotFound() {
        try {
            RestTemplate rest = new RestTemplate();
            rest.getForObject(
                "http://localhost:8080/bogusPage", String.class);
            fail("Should result in HTTP 404");
        } catch (HttpClientErrorException e) {
            assertEquals(HttpStatus.NOT_FOUND, e.getStatusCode());
            throw e;
        }
    }
}
```

Runs test in server

Performs GET request

Asserts HTTP 404 (not found) response

Although this is a very simple test, it sufficiently demonstrates how to use the `@WebIntegrationTest` to start the application with a server. The actual server that's

started will be determined in the same way it would be if we were running the application at the command line. By default, it starts Tomcat listening on port 8080. Optionally, however, it could start Jetty or Undertow if either of those is in the classpath.

The body of the test method is written assuming that the application is running and listening on port 8080. It uses Spring's `RestTemplate` to make a request for a non-existent page and asserts that the response from the server is an HTTP 404 (not found). The test will fail if any other response is returned.

#### 4.3.1 Starting the server on a random port

As mentioned, the default behavior is to start the server listening on port 8080. That's fine for running a single test at a time on a machine where no other server is already listening on port 8080. But if you're like me, you've probably *always* got something listening on port 8080 on your local machine. In that case, the test would fail because the server wouldn't start due to the port collision. There must be a better way.

Fortunately, it's easy enough to ask Spring Boot to start up the server on a randomly selected port. One way is to set the `server.port` property to 0 to ask Spring Boot to select a random available port. `@WebIntegrationTest` accepts an array of `String` for its `value` attribute. Each entry in the array is expected to be a name/value pair, in the form `name=value`, to set properties for use in the test. To set `server.port` you can use `@WebIntegrationTest` like this:

```
@WebIntegrationTest(value={"server.port=0"})
```

Or, because there's only one property being set, it can take a simpler form:

```
@WebIntegrationTest("server.port=0")
```

Setting properties via the `value` attribute is handy in the general sense, but `@WebIntegrationTest` also offers a `randomPort` attribute for a more expressive way of asking the server to be started on a random port. You can ask for a random port by setting `randomPort` to `true`:

```
@WebIntegrationTest(randomPort=true)
```

Now that we have the server starting on a random port, we need to be sure we use the correct port when making web requests. At the moment, the `getForObject()` method is hard-coded with port 8080 in its URL. If the port is randomly chosen, how can we construct the request to use the right port?

First we'll need to inject the chosen port as an instance variable. To make this convenient, Spring Boot sets a property with the name `local.server.port` to the value of the chosen port. All we need to do is use Spring's `@Value` to inject that property:

```
@Value("${local.server.port}")  
private int port;
```

Now that we have the port, we just need to make a slight change to the `getForObject()` call to use it:

```
rest.getForObject(
    "http://localhost:{port}/bogusPage", String.class, port);
```

Here we've traded the hardcoded 8080 for a `{port}` placeholder in the URL. By passing the `port` property as the last parameter in the `getForObject()` call, we can be assured that the placeholder will be replaced with whatever value was injected into `port`.

### 4.3.2 Testing HTML pages with Selenium

`RestTemplate` is fine for simple requests and it's perfect for testing REST endpoints. But even though it can be used to make requests against URLs that return HTML pages, it's not very convenient for asserting the contents of the page or performing operations on the page itself. At best, you'll be able to assert the precise content of the resulting HTML (which will result in fragile tests). But you won't easily be able to assert selected content on the page or perform operations such as clicking links or submitting forms.

A better choice for testing HTML applications is Selenium ([www.seleniumhq.org](http://www.seleniumhq.org)). Selenium does more than just perform requests and fetch the results for you to verify. Selenium actually fires up a web browser and executes your test within the context of the browser. It's as close as you can possibly get to performing the tests manually with your own hands. But unlike manual testing, Selenium tests are automated and repeatable.

To test our reading list application using Selenium, let's write a test that fetches the home page, fills out the form for a new book, posts the form, and then finally asserts that the landing page includes the newly added book.

First we'll need to add Selenium to the build as a test dependency:

```
testCompile("org.seleniumhq.selenium:selenium-java:2.45.0")
```

Now we can write the test class. The following listing shows a basic template for a Selenium test that uses Spring Boot's `@WebIntegrationTest`.

#### Listing 4.6 A template for Selenium testing with Spring Boot

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
    classes=ReadingListApplication.class)
@WebIntegrationTest(randomPort=true)
public class ServerWebTests {

    private static FirefoxDriver browser;

    @Value("${local.server.port}")
    private int port;
```

Starts on a random port

Injects the port



```

@BeforeClass
public static void openBrowser() {
    browser = new FirefoxDriver();
    browser.manage().timeouts()
        .implicitlyWait(10, TimeUnit.SECONDS);
}

@AfterClass
public static void closeBrowser() {
    browser.quit();
}
}

```

**Sets up Firefox driver**

**Shuts down browser**

As with the simpler web test we wrote earlier, this class is annotated with `@WebIntegrationTest` and sets `randomPort` to `true` so that the application will be started and run with a server listening on a random port. And, as before, that port is injected into the `port` property so that we can use it to construct URLs to the running application.

The static `openBrowser()` method creates a new instance of `FirefoxDriver`, which will open a Firefox browser (it will need to be installed on the machine running the test). When we write our test method, we'll perform browser operations through the `FirefoxDriver` instance. The `FirefoxDriver` is also configured to wait up to 10 seconds when looking for any elements on the page (in case those elements are slow to load).

After the test has completed, we'll need to shut down the Firefox browser. Therefore, `closeBrowser()` calls `quit()` on the `FirefoxDriver` instance to bring it down.

**PICK YOUR BROWSER** Although we're testing with Firefox, Selenium also provides drivers for several other browsers, including Internet Explorer, Google's Chrome, and Apple's Safari. Not only can you use other browsers, it's probably a good idea to write your tests to use any and all browsers you want to support.

Now we can write our test method. As a reminder, we want to load the home page, fill in and submit the form, and then assert that we land on a page that includes our newly added book in the list. The following listing shows how to do this with Selenium.

#### Listing 4.7 Testing the reading-list application with Selenium

```

@Test
public void addBookToEmptyList() {
    String baseUrl = "http://localhost:" + port;

    browser.get(baseUrl);

    assertEquals("You have no books in your book list",
        browser.findElementByTagName("div").getText());

    browser.findElementByName("title")
}

```

**Fetches the home page**

**Asserts an empty book list**

```

        .sendKeys("BOOK TITLE");
browser.findElementByName("author")
        .sendKeys("BOOK AUTHOR");
browser.findElementByName("isbn")
        .sendKeys("1234567890");
browser.findElementByName("description")
        .sendKeys("DESCRIPTION");
browser.findElementByTagName("form")
        .submit();
    }

    WebElement dl =
        browser.findElementByCssSelector("dt.bookHeadline");
    assertEquals("BOOK TITLE by BOOK AUTHOR (ISBN: 1234567890)",
        dl.getText());
    WebElement dt =
        browser.findElementByCssSelector("dd.bookDescription");
    assertEquals("DESCRIPTION", dt.getText());
}

```

**Fills in and submits form**

**Asserts new book in list**

The very first thing that the test method does is use the `FirefoxDriver` to perform a GET request for the reading list's home page. It then looks for a `<div>` element on the page and asserts that its text indicates that no books are in the list.

The next several lines look for the fields in the form and use the driver's `sendKeys()` method to simulate keystroke events on those field elements (essentially filling in those fields with the given values). Finally, it looks for the `<form>` element and submits it.

After the form submission is processed, the browser should land on a page with the new book in the list. So the final few lines look for the `<dt>` and `<dd>` elements in that list and assert that they contain the data that the test submitted in the form.

When you run this test, you'll see the browser pop up and load the reading-list application. If you pay close attention, you'll see the form filled out, as if by a ghost. But it's no spectre using your application—it's the test.

The main thing to notice about this test is that `@WebIntegrationTest` was able to start up the application and server for us so that Selenium could start poking at it with a web browser. But what's especially interesting about how this works is that you can use the test facilities of your IDE to run as many or as few of these tests as you want, without having to rely on some plugin in your application's build to start a server for you.

If testing with Selenium is something that you think you'll find useful, you should check out *Selenium WebDriver in Practice* by Yujun Liang and Alex Collins (<http://manning.com/liang/>), which goes into far more details about testing with Selenium.

## 4.4 Summary

Testing is an important part of developing quality software. Without a good suite of tests, you'll never know for sure if your application is doing what it's expected to do.

For unit tests, which focus on a single component or a method of a component, Spring isn't really necessary. The benefits and techniques promoted by Spring—loose

coupling, dependency injection, and interface-driven design—make writing unit tests easy. But Spring doesn't need to be directly involved in unit tests.

Integration-testing multiple components, however, begs for help from Spring. In fact, if Spring is responsible for wiring those components up at runtime, then Spring should also be responsible for wiring them up in integration tests.

The Spring Framework provides integration-testing support in the form of a JUnit class runner that loads a Spring application context and enables beans from the context to be injected into a test. Spring Boot builds upon Spring integration-testing support with a configuration loader that loads the application context in the same way as Spring Boot itself, including support for externalized properties and Spring Boot logging.

Spring Boot also enables in-container testing of web applications, making it possible to fire up your application to be served by the same container that it will be served by when running in production. This gives your tests the closest thing to a real-world environment for verifying the behavior of the application.

At this point we've built a rather complete (albeit simple) application that leverages Spring Boot starters and auto-configuration to handle the grunt work so that we can focus on writing our application. And we've also seen how to take advantage of Spring Boot's support for testing the application. Coming up in the next couple of chapters, we're going to take a slightly different tangent and explore the ways that Groovy can make developing Spring Boot applications even easier. We'll start in the next chapter by looking at a few features from the Grails framework that have made their way into Spring Boot.

# Spring Boot IN ACTION

Craig Walls



**T**he Spring Framework simplifies enterprise Java development, but it does require lots of tedious configuration work. Spring Boot radically streamlines spinning up a Spring application. You get automatic configuration and a model with established conventions for build-time and runtime dependencies. You also get a handy command-line interface you can use to write scripts in Groovy. Developers who use Spring Boot often say that they can't imagine going back to hand configuring their applications.

**Spring Boot in Action** is a developer-focused guide to writing applications using Spring Boot. In it, you'll learn how to bypass configuration steps so you can focus on your application's behavior. Spring expert Craig Walls uses interesting and practical examples to teach you both how to use the default settings effectively and how to override and customize Spring Boot for your unique environment. Along the way, you'll pick up insights from Craig's years of Spring development experience.

## What's Inside

- Develop Spring apps more efficiently
- Minimal to no configuration
- Runtime metrics with the Actuator
- Covers Spring Boot 1.3

Written for readers familiar with the Spring Framework.

**Craig Walls** is a software developer, author of the popular book *Spring in Action, Fourth Edition*, and a frequent speaker at conferences.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit [manning.com/books/spring-boot-in-action](http://manning.com/books/spring-boot-in-action)

“Easy to digest and fun to read.”

—From the Foreword by Andrew Glover, Netflix

“The evolution of Spring continues, and this guide helps maximize its potential.”

—Michael A. Angelo  
ThreatConnect

“A lucid, real-world treatment of a valuable toolset. The practical examples help bring agility and simplicity to application construction.”

—Eric Kramer  
Research Institute at Nationwide  
Children's Hospital

“Easy-to-follow, comprehensive, awesome!”

—Furkan Kamaci, Alcatel-Lucent

