CHAPTER 8

Spring Boot Testing

In this chapter we are going to talk about how Spring Boot uses the power of the Spring Testing Framework to facilitate development by providing powerful tools for creating unit and integration testing with ease. In previous chapters we have done some testing of our two applications, but we haven't covered other important features of the Spring Boot Testing Framework, so let's start talking about the Spring Testing Framework, which is the basis for Spring Boot testing.

Spring Testing Framework

One of the main ideas of the Spring Framework is to encourage developers to create simple and loosely coupled classes and to program to interfaces, making the software more robust and extensible. The Spring Framework provides the tools for making unit and integration testing easy (actually, you don't need Spring to test the functionality of your system if you really program to interfaces); in other words, your application should be testable using either the JUnit or TestNG test engine with objects (by simple instantiation using the `new` operator)—without Spring or any other container.

The Spring Framework has several testing packages that help create unit and integration testing for applications. It offers unit testing by providing several mock objects (`Environment`, `PropertySource`, JNDI, Servlet; Reactive: `ServerHttpRequest` and `ServerHttpResponse` test utilities) that help you to test your code in isolation.

One of the most commonly used testing features of the Spring Framework is integration testing. Its primary's goals are

- Managing the Spring IoC container caching between test execution
- Transaction management
- Dependency injection of test fixture instances
- Spring-specific base classes

F. Gutierrez, *Pro Spring Boot 3*, https://doi.org/10.1007/978-1-4842-9294-5_8

The Spring Framework provides an easy way to do testing by integrating the `ApplicationContext` in the tests. The Spring testing module offers several ways to use the `ApplicationContext`, programmatically and through annotations:

- `BootstrapWith`: A class-level annotation to configure how the Spring `TestContext` Framework is bootstrapped.
- `@ContextConfiguration`: Defines a class-level metadata to determine how to load and configure an `ApplicationContext` for integration tests. This is a must-have annotation for your classes because that's where the `ApplicationContext` loads all your bean definitions.
- `@WebAppConfiguration`: A class-level annotation to declare that the `ApplicationContext` that loads for an integration test should be a `WebApplicationContext`.
- `@ActiveProfile`: A class-level annotation to declare which bean definition profile(s) should be active when loading an `ApplicationContext` for an integration test.
- `@TestPropertySource`: A class-level annotation to configure the locations of properties files and inline properties to be added to the set of `PropertySources` in the `Environment` for an `ApplicationContext` loaded for an integration test.
- `@DirtiesContext`: Indicates that the underlying Spring `ApplicationContext` has been dirtied during the execution of a test (modified or corrupted, for example, by changing the state of a singleton bean) and should be closed.

The Spring Framework offers many more annotations, including `@TestExecutionListeners`, `@Commit`, `@Rollback`, `@ BeforeTransaction`, `@AfterTransaction`, `@Sql`, `@SqlGroup`, `@SqlConfig`, `@Timed`, `@Repeat`, `@IfProfileValue`, and so forth.

As you can see, there are a lot of choices when you test with the Spring Framework. Normally, you always use the `@RunWith` annotation that wires up all the test framework goodies. For example, the following code shows how you can do unit/integration testing using just Spring:

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})

@ActiveProfiles("dev")
@Transactional
public class UsersTests {
      @Test
      public void userPersistenceTest(){
//...
}
}
```

Now, let's explore the Spring Boot set of features that enable you to create better unit, integration, and isolation tests (per layer) with ease.

Spring Boot Testing Framework

Spring Boot uses the power of the Spring Testing Framework by enhancing and adding new annotations and features that make testing easier for developers.

If you want to start using all the testing features provided by Spring Boot, you only need to add the `spring-boot-starter-test` dependency with scope test to your application. This dependency is already in place if you used the Spring Initializr (https://start.spring.io) to create your projects.

The `spring-boot-starter-test` dependency provides several test frameworks that play very well with all the Spring Boot testing features: Junit 5, AssertJ, Hamcrest, Mockito, JSONassert, JsonPath, and the Spring Test and Spring Boot Test utilities and integration support for Spring Boot applications. If you are using a different test framework, it likely will play very nicely with the Spring Boot Test module; you only need to include those dependencies manually.

Spring Boot provides the `@SpringBootTest` annotation that simplifies the way you can test Spring apps. Normally, with Spring testing, you are required to add several annotations to test a particular feature or functionality of your app, but not in Spring Boot. The `@SpringBootTest` annotation has parameters that are useful when testing a web app with the parameter `webEnvironment` which accepts values such as a `RANDOM_PORT` or `DEFINED_PORT`, `MOCK`, and `NONE`. The `@SpringBootTest` annotation also defines `properties` (properties that you want to test for different values), `args` (for arguments

you normally passed in the command line, such `@SpringBootTest(args = "--app.name=Users")`), `classes` (a list of classes in which you declared your beans using the `@Configuration` annotation), and `useMainMethod`, with values such as `ALWAYS` and `WHEN_AVAILABLE` (meaning that it will use the `main` method of the application to the set the `ApplicationContext` to run the tests).

In the following sections I'll show you some of the main test classes from the two different projects that we have been developing so far.

Note In this chapter we are going to use the code in the folder `08-testing`. You have complete access to the code at the Apress website: https://www.apress.com/gp/services/source-code.

Testing Web Apps with a Mock Environment

The `@SpringBootTest` annotation by default uses a mock environment, meaning that it doesn't start the server and it's ready for testing web endpoints. To use this feature, we need to use the `MockMvc` class along with the `@AutoConfigurationMockMvc` annotation as a marker for the test class. This annotation will configure the `MockMvc` class and all its dependencies, such as filters, security (if any), and so forth. If we were using just Spring (no Spring Boot), we could use the `MockMvc` class, but we would need to do some extra steps because it relies on the `WebApplicationContext`. Fortunately, with Spring Boot, we only need to inject it with the `@Autowired` annotation.

Let's look at the Users App project in the `test` folder. The name of the class is `UserMockMvcTests`. See Listing 8-1.

Listing 8-1. src/test/java/apress/com/users/UserMockMvcTests.java

```
package com.apress.users;

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.context.ActiveProfiles;
```

```
import org.springframework.test.web.servlet.MockMvc;

import static org.hamcrest.Matchers.hasItem;
import static org.hamcrest.Matchers.hasSize;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("mockMvc")
public class UserMockMvcTests {

    @Autowired
    MockMvc mockMvc;

    @Test
    void createUserTests() throws Exception {
        String location = mockMvc.perform(post("/users")
                        .contentType("application/json")
                        .content("""
                        {
                            "email": "dummy@email.com",
                            "name": "Dummy",
                            "password": "aw2sOmeR!",
                            "gravatarUrl": "https://www.gravatar.com/
                            avatar/fb651279f4712e209991e05610dfb03a?d
                            =wavatar",
                            "userRole": ["USER"],
                            "active": true
                        }
                       """))
                .andExpect(status().isCreated())
                .andExpect(header().exists("Location"))
                .andReturn().getResponse().getHeader("Location");
```

```
        mockMvc.perform(get(location))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.email").exists())
                .andExpect(jsonPath("$.active").value(true));
    }

    @Test
    void getAllUsersTests() throws Exception {
        mockMvc.perform(get("/users"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$[0].name").value("Dummy"))
                .andExpect(jsonPath("$..active").value(hasItem(true)))
                .andExpect(jsonPath("$[*]").value(hasSize(1)));
    }
}
```

Let's analyze the `UserMockMvcTests` class:

- `@SpringBootTest`: This annotation is essential for testing with Spring Boot. It provides the `SpringBootContextLoader` as the default `ContextLoader`. It searches for all the `@Configuration` classes, it allows custom `Environment` properties, and it can register either a `TestRestTemplate` (which we have been using in previous chapters when running our tests) or a `WebTestClient` (for Reactive apps, such as MongoDB). As I've already mentioned, it also accepts parameters where you can add the type of web environment—by default, `MOCK`—arguments, and properties.
- `@AutoConfigureMockMvc`: This annotation creates the `MockMvc` bean that can be injected to perform all the server-side testing, among other features, such as filters, security, etc.
- `@ActiveProfiles`: This annotation helps to run only the defined beans under the name set (in this case `mockMvc`). This is helpful when you have a lot of testing and you require certain behavior. This annotation will activate the `mockMvc` profile. Normally, in every configuration class, you can add the `@Profile({"<profile-name>"})` annotation to specify which beans are needed for such profile(s).

- `MockMvc`: This bean class is the entry point for all the server-side testing, and it supports request builders and result matchers that can be combined with different test libraries such as Mockito, Hamcrest, and AssertJ, among others.
- `MockMvcRequestBuilders`: The `MockMvc` class has a `perform` method that accepts a request builder, and the `MockMvcRequestBuilders` class provides a fluent API where you can perform the `get`, `post`, `put`, etc. These request builders accept the URI in the form of a path (it's not necessary to add the complete URL), and you can build upon whatever request you need, like adding headers, content type, content, etc., and this is because it returns a `MockHttpServletRequestBuilder` class that brings a lot of customization for the request.
- `Hamcrest` and `MockMvcResultMatchers`: After calling the `perform` method (from `MockMvc`), you have access to a `ResultsActions` interface where you can add all the expectations on the result of the executed request; it includes the `andExpect(ResultMatcher)`, `andExpectAll(ResultMatcher...)`, `andDo(ResultHandler)`, and `andReturn()` methods. The `andReturn()` method returns a `MvcResult` interface that is also a fluent API where you can get anything that brings the request (content, headers, etc.). And because some of these methods require a `ResultMatcher` interface, you can find implementation using the Hamcrest library (such as `contains`, `hasSize`, etc.). In our tests, we are using the `jsonPath` result matchers that allow us to interact with the JSON response.

It's important to mention that these tests need something extra that allows them to use the repositories, and we are going to talk about that in the following sections.

Using Mocking and Spying Beans

Spring Boot testing includes two annotations that allow you to mock Spring beans, which is helpful when some of these beans depend on external services, such a third-party REST endpoints, database connections, and so forth. If one of these services is unavailable, these annotations can help. Let's first look at the `UserMockBeanTests` class. See Listing 8-2.

Listing 8-2. src/test/java/apress/com/users/UserMockBeanTests.java

```
package com.apress.users;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
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.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.web.servlet.MockMvc;

import static org.mockito.Mockito.*;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("mockBean")
public class UserMockBeanTests {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    UserRepository userRepository;

    @Test
    void saveUsers() throws Exception {
        var user = UserBuilder.createUser()
                    .withName("Dummy")
                    .withEmail("dummy@email.com")
```

```
                .active()
                .withRoles(UserRole.USER)
                .withPassword("aw3sOm3R!")
                .build();

        when(userRepository.save(any())).thenReturn(user);

        mockMvc.perform(post("/users")
            .contentType(MediaType.APPLICATION_JSON)
            .content("""
    {
                    "email": "dummy@email.com",
                    "name": "Dummy",
                    "password": "aw2sOmeR!",
                    "gravatarUrl": "https://www.gravatar.com/
                    avatar/fb651279f4712e209991e05610dfb03a?d
                    =wavatar",
                    "userRole": ["USER"],
                    "active": true
                }
            """))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.name").value(user.getName()))
            .andExpect(jsonPath("$.email").value(user.getEmail()))
            .andExpect(jsonPath("$.userRole").isArray())
            .andExpect(jsonPath("$.userRole[0]").value("USER"));

        verify(userRepository, times(1)).save(Mockito.any(User.class));
    }
}
```

The `UserMockBeanTests class` includes the following:

- `@MockBean`: This annotation mocks the object marked, enabling you to add the necessary behavior and outcome for that object. It replaces the object implementation, so it can be easily customized.

To use this bean, you need to use a framework such as Mockito. In our test in Listing 8-2, we are mocking the `when() UserRepository`, meaning that we don't need any connection or anything related to the database; basically, we are mocking its behavior.

- `Mockito.*`: The Mockito library help us to add the behavior to our mock bean (in this case, the `UserRepository`) with the `thenReturn()` methods. Then we can verify the behavior after we performed the call with the `verify()` method. Note that the Mockito library has a lot of useful fluent APIs. In our test, we are specifying that when the `UserRepository` uses the method `save` (with any value), it will return the user we specified, then at the end we `verify` that our mock bean was call once when saving the `User` object.

Remember, the `@MockBean` will replace the actual bean with a mock implementation that is easy to modify to get the desired behavior.

Next, let's go to the `UserSpyBeanTests` class, shown in Listing 8-3.

Listing 8-3. src/test/java/apress/com/users/UserSpyBeanTests.java

```
package com.apress.users;

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.boot.test.mock.mockito.SpyBean;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Arrays;
import java.util.List;

import static org.mockito.Mockito.doReturn;
import static org.mockito.Mockito.verify;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
```

```
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("spyBean")
public class UserSpyBeanTests {

    @Autowired
    private MockMvc mockMvc;

    @SpyBean
    private UserRepository userRepository;

    @Test
    public void testGetAllUsers() throws Exception {

        List<User> mockUsers = Arrays.asList(
                UserBuilder.createUser()
                        .withName("Ximena")
                        .withEmail("ximena@email.com")
                        .build(),
                UserBuilder.createUser()
                        .withName("Norma")
                        .withEmail("norma@email.com")
                        .build()
        );
        doReturn(mockUsers).when(userRepository).findAll();

        mockMvc.perform(get("/users"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$[0].name").value("Ximena"))
                .andExpect(jsonPath("$[1].name").value("Norma"));

        verify(userRepository).findAll();
    }
}
```

Let's analyze the `UserSpyBeanTests` class:

> `@SpyBean`: This annotation creates a spy bean. A spy bean is similar to a mock object, but it retains the original behavior of the real bean. It allows you to intercept and verify method invocations and allows you to specify specific behavior for selected methods while keeping the rest of the methods intact. In our example in Listing 8-3 we are using the Mockito library to use the `doReturn()` method when the `UserRepository` instance is used, and when we call the `findAll()` method. At the end, we are verifying that the actual `findAll()` method was called.

Testcontainers

Testcontainers (`https://testcontainers.com/`) is an open source framework that allows you to run some services in a docker container through unit testing frameworks. Spring Boot now has the capability to use Testcontainers with ease, without any configuration; you simply include two dependencies in your Maven or Gradle build files: `org.springframework.boot:spring-boot-testcontainers` and `org.testcontainers:junit-jupiter` (with `scope test` for Maven or `testImplementation` for Gradle).

Previous chapters demonstrated the use of the `spring-boot-docker-compose` feature, which enables to read a `docker-compose.yaml` file provided by you (the developer) and allows you to run your app by creating the necessary environment for your app. Recall that this feature works only when running the app, not when executing the tests. So, the solution is Testcontainers!

With Testcontainers, Spring Boot introduces the following annotations:

- `@Testcontainers`: Starts and stops the containers; this annotation looks for every `@Container` annotation defined.
- `@Container`: Sets up all the necessary configuration that is required for the Testcontainers framework to initialize.

- `@ServiceConnection`: Takes care of creating the default connection details to be used with your application. And in this case, it will depend on which technology you will use. For example, in this chapter we are using Postgres for the Users App project and MongoDB for the My Retro App project, so it will be necessary to include the necessary Testcontainers dependency, `org.testcontainers:postgresql` or `org.testcontainers:mongodb`, respectively.

You will find out that with Testcontainers in Spring Boot, normally it will take some time to pull down the image (if it's not there) and start the testing. For this behavior, the Spring Boot team also created the `@RestartScope` annotation that allows you to recreate the container (keep it running) when your app is restarted if you are using Spring Boot Dev Tools. One of the main features of Dev Tools is that you can use it in your favorite IDE and it will restart your app in any new file modification when it's saved. The normal steps without Dev Tools are to modify your app, save it, and then either stop or restart the app; with Spring Boot Dev Tools, you don't need to do this, because it automatically restarts the app when the file is saved.

So, let's look at how to use Testcontainers with Spring Boot. The following snippet shows you how to use the `@Testcontainers`, `@Container`, and `@ServiceConnection` annotations and how to enable the PostgreSQL container:

```
@SpringBootTest
@Testcontainers
public class UserTests {

    @Container
    @ServiceConnection
    static PostgreSQLContainer<?> postgreSQLContainer =
       new PostgreSQLContainer<>("postgres:latest");

    // Your test here ...
}
```

In the following sections, we'll see this as part of the test.

Spring Boot Testing Slices

So far, we have seen integration testing because normally we require the whole server to do such tests, in some other chapters we used this a lot, and we use the `TestRestTemplate` to perform some calls to the web application using the `SpringBootTest.WebEnvironment.RANDOM_PORT` to get the port and such.

Spring Boot offers a way to test only the layers we need to test instead of testing the whole environment. So, suppose we only need to parse the JSON response and verify that the serialization was done correctly, the values of the field match, and so forth. With Spring Boot, we don't need the whole environment for that. The Spring Boot slice testing feature allows us to test every layer separately, including the controller layer, the data layer, and the domain layer.

- `@WebMvcTest`

  Purpose: Tests Spring MVC controllers in isolation.

  Includes: Web layer components (controllers, filters, view resolvers, etc.)

  Excludes: Service layer, repository layer, and other non-web components.

  Suitable For: Testing the behavior of controllers, request mappings, validation, and rendering views (if applicable).

- `@DataJpaTest`

  Purpose: Tests Spring Data JPA repositories in isolation.

  Includes: JPA repositories, entity classes, and related configuration.

  Excludes: Service layer, web layer, and other non-JPA components.

  Suitable For: Testing repository CRUD operations, queries, and custom repository methods.

- `@JdbcTest`

  Purpose: Tests data access code that uses plain JDBC (without Spring Data JPA).

  Includes: DataSource configuration, JDBC templates, and SQL scripts.

  Excludes: JPA repositories, web layer, and other non-JDBC components.

  Suitable For: Testing low-level JDBC operations, SQL scripts, and data access logic not using JPA.

- `@JsonTest`

  Purpose: Tests JSON serialization and deserialization.

  Includes: Jackson or Gson configurations and custom serializers/deserializers.

  Excludes: Web layer, data access layers, and other non-JSON components.

  Suitable For: Verifying correct serialization and deserialization of your objects to and from JSON.

- `@RestClientTest`

  Purpose: Tests Spring's RestTemplate or WebClient based REST clients.

  Includes: REST client beans, error handling configurations, and related components.

  Excludes: Web layer (server-side), data access layers, and other non-REST client components.

  Suitable For: Verifying REST client configurations, request building, response handling, and error scenarios.

- `@DataMongoTest`

  Purpose: Tests Spring Data MongoDB repositories in isolation.

  Includes: MongoDB repositories, entity classes, and related configuration.

  Excludes: Web layer, JPA components, and other non-MongoDB components.

  Suitable For: Testing repository operations with MongoDB, queries, and custom repository methods.

- `@SpringBootTest` (Special Case)

  Purpose: Not strictly a "slice," but provides a way to test the full application context.

  Includes: All Spring beans and configurations.

  Suitable For: End-to-end tests or integration tests that need the complete application context.

Let's start with the @JsonTest annotation.

@JsonTest

`@JsonTest` is an annotation that allows you to test your JSON serialization and deserialization. This annotation also auto-configures the right mapper support if it finds any of these libraries in your classpath: Jackson, Gson, or Jsonb.

Listing 8-4 shows the `UserJsonTest` class.

Listing 8-4. src/test/java/apress/com/users/UserJsonTest.java

```
package com.apress.users;

import jakarta.validation.ConstraintViolationException;
import jakarta.validation.Validation;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.json.JsonTest;
import org.springframework.boot.test.json.JacksonTester;
import org.springframework.boot.test.json.JsonContent;
```

```
import java.io.IOException;

import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.assertThatExceptionOfType;
import static org.junit.jupiter.api.Assertions.assertThrows;

@JsonTest
public class UserJsonTests {

    @Autowired
    private JacksonTester<User> jacksonTester;

    @Test
    void serializeUserJsonTest() throws IOException{
        User user = UserBuilder.createUser(Validation.
        buildDefaultValidatorFactory().getValidator())
                .withEmail("dummy@email.com")
                .withPassword("aw2sOme")
                .withName("Dummy")
                .withRoles(UserRole.USER)
                .active().build();

        JsonContent<User> json =  jacksonTester.write(user);

        assertThat(json).extractingJsonPathValue("$.email").
        isEqualTo("dummy@email.com");
        assertThat(json).extractingJsonPathArrayValue("$.userRole").size().
        isEqualTo(1);
        assertThat(json).extractingJsonPathBooleanValue("$.active").
        isTrue();
        assertThat(json).extractingJsonPathValue("$.gravatarUrl").
        isNotNull();
        assertThat(json).extractingJsonPathValue("$.gravatarUrl").
        isEqualTo(UserGravatar.getGravatarUrlFromEmail(user.getEmail()));
    }
```

```
    @Test
    void serializeUserJsonFileTest() throws IOException{
        User user = UserBuilder.createUser(Validation.
        buildDefaultValidatorFactory().getValidator())
                .withEmail("dummy@email.com")
                .withPassword("aw2sOme")
                .withName("Dummy")
                .withRoles(UserRole.USER)
                .active().build();

        System.out.println(user);

        JsonContent<User> json =  jacksonTester.write(user);

                    assertThat(json).isEqualToJson("user.json");

    }

    @Test
    void deserializeUserJsonTest() throws Exception{
        String userJson = """
                {
                  "email": "dummy@email.com",
                  "name": "Dummy",
                  "password": "aw2sOme",
                  "userRole": ["USER"],
                  "active": true
                }
                """;
        User user = this.jacksonTester.parseObject(userJson);

        assertThat(user.getEmail()).isEqualTo("dummy@email.com");
        assertThat(user.getPassword()).isEqualTo("aw2sOme");
        assertThat(user.isActive()).isTrue();

    }

    @Test
    void userValidationTest(){
        assertThatExceptionOfType(ConstraintViolationException.class)
```

```
            .isThrownBy( () -> UserBuilder.createUser(Validation.
            buildDefaultValidatorFactory().getValidator())
                    .withEmail("dummy@email.com")
                    .withName("Dummy")
                    .withRoles(UserRole.USER)
                    .active().build());

        // Junit 5
        Exception exception = assertThrows(ConstraintViolationException.
        class, () -> {
            UserBuilder.createUser(Validation.
            buildDefaultValidatorFactory().getValidator())
                    .withName("Dummy")
                    .withRoles(UserRole.USER)
                    .active().build();
        });

        String expectedMessage = "email: Email can not be empty";
        assertThat(exception.getMessage()).contains(expectedMessage);

    }
}
```

The `UserJsonTest` class includes the following:

- `@JsonTest`: This annotation marks this class as only JSON serialization. It auto-configures the test based on what is in the classpath on what dependency library are you using, either Jackson (comes with `spring-boot-starter-web` as default), Jsonb, or Gson.
- `JacksonTester`: This class takes care of the *serialization* and *deserialization* of the domain class. It uses the Jackson library (the default) and uses the `ObjectMapper` class to do the serialization.
- `JsonContent`: This class includes the content from a JSON tester, which is useful to get the values from the serialization.

- `UserBuilder`: This class receives a `Validation` class that helps to inspect and validate the values of fields marked with `@NotBlank` or `@NotNull` annotations.
- `Validation`: This class belongs to the Jakarta validation package that helps to review and validate the fields marked with annotations such as `@NotBlank`, `@NoNull`, etc.
- `assertThat`: We are using the AssertJ package to use the assertations for our serialization and deserialization in this class.
- `assertThatExceptionOfType`: This class helps to identify the type of any error or exception thrown during testing. In Listing 8-4, we are creating a `User` class that doesn't comply with the `@NotBlank` annotation, and because of that, this class identifies it as a `ConstraintViolationException` exception.
- `assertThrows`: And we can do the same with this `assertThrows` call. Different way to accomplish the same asserts.

As you can see, doing tests about your domain gets simpler that even, more waiting for some response or a particular service, you can test your own domain schemas by doing serialization and deserialization of your classes. Also, note that we are allowed to test against JSON files, in this case the `user.json` file. This file must be placed in the `src/test/resources/com/apress/users` folder in order to be picked up by the test framework.

@WebMvcTest

If we can do stand-alone domain/schemas tests, can we do them over our web controllers? Yes, we can! We can use the `@WebMvcTest` annotation to test only the web endpoint—our controllers. The `@WebMvcTest` annotation auto-configures all the Spring MVC infrastructure, but it will be limited just to the `@RestController`, annotated classes, `Filter` interface implementations, and all other web-related classes, and by default it sets the `webEnvironment` to `MOCK`, which means that this annotation inherits from the `@AutoConfigureMockMvc`, which in turn means that you can use the `MockMvc` class.

Let's look at the `UserControllerTests` class. See Listing 8-5.

Listing 8-5. src/main/java/apress/com/users/UserControllerTests.java

```
package com.apress.users;

import com.fasterxml.jackson.databind.ObjectMapper;
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.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Arrays;
import java.util.Optional;

import static org.mockito.Mockito.doNothing;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.
MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@WebMvcTest(controllers = { UsersController.class })
public class UserControllerTests {

  @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserRepository userRepository;

    @Test
    void getAllUsersTest() throws Exception {
        when(userRepository.findAll()).thenReturn(Arrays.asList(
                UserBuilder.createUser()
```

```
                .withName("Ximena")
                .withEmail("ximena@email.com")
                .active()
                .withRoles(UserRole.USER, UserRole.ADMIN)
                .withPassword("aw3sOm3R!")
                .build(),
        UserBuilder.createUser()
                .withName("Norma")
                .withEmail("norma@email.com")
                .active()
                .withRoles(UserRole.USER)
                .withPassword("aw3sOm3R!")
                .build()
    ));

    mockMvc.perform(get("/users"))
            .andDo(print())
            .andExpect(status().isOk())
  .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$[0].active").value(true));
}

@Test
void newUserTest() throws Exception {
    User user = UserBuilder.createUser()
            .withName("Dummy")
            .withEmail("dummy@email.com")
            .active()
            .withRoles(UserRole.USER, UserRole.ADMIN)
            .withPassword("aw3sOm3R!")
            .build();
    when(userRepository.save(user)).thenReturn(user);

    mockMvc.perform(post("/users")
                .content(toJson(user))
                .contentType(MediaType.APPLICATION_JSON))
            .andDo(print())
```

```
            .andExpect(status().isCreated())
            .andExpect(content().contentType(MediaType.
            APPLICATION_JSON))
            .andExpect(jsonPath("$.email").value("dummy@email.com"));
}

@Test
void findUserByEmailTest() throws Exception {
    User user = UserBuilder.createUser()
            .withName("Dummy")
            .withEmail("dummy@email.com")
            .active()
            .withRoles(UserRole.USER, UserRole.ADMIN)
            .withPassword("aw3sOm3R!")
            .build();
    when(userRepository.findById(user.getEmail())).thenReturn(Optional.
    of(user));

    mockMvc.perform(get("/users/{email}",user.getEmail())
            .contentType(MediaType.APPLICATION_JSON))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.
            APPLICATION_JSON))
            .andExpect(jsonPath("$.email").value("dummy@email.com"));
}

@Test
void deleteUserByEmailTest() throws Exception{
    User user = UserBuilder.createUser()
            .withEmail("dummy@email.com")
            .build();
    doNothing().when(userRepository).deleteById(user.getEmail());

    mockMvc.perform(delete("/users/{email}",user.getEmail()))
            .andExpect(status().isNoContent());

}
```

```
    private static String toJson(final Object obj) {
        try {
            return new ObjectMapper().writeValueAsString(obj);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

The `UserControllerTests` class includes the following:

- `@WebMvcvTest`: This annotation inherits the `@AutoConfiguraMockMvc` annotation, so it's ready to use the `MockMvc` class, which also means that the `webEnvironment` is a `MOCK`. Thanks to this, we can test only our web controllers without starting up the complete server to do some testing. You can declare more controllers in this annotation.
- `MockMvc`: As you already know, this class is the entry point for the server side for the Spring MVC test support. In this case, it allows us to perform HTTP method requests to our controllers.
- `@MockBean`: This annotation will mock the bean behavior. This is helpful when you don't want to wait for an external service to be ready, because you can mock the behavior. In this case, we are mocking the `UserRepository`.
- `Mockito.*`: We are using the `when().thenReturn` to prepare our call and then perform the calls and expect the results. Also, we are using the `doNothing().when()` calls. As you can see, the Mockito library offer a nice fluent API that you apply for these scenarios.

Before you continue to the next section, take time to review the `@WebMvcTest` annotation and `UserControllerTests` class in depth.

@DataJpaTest

With this annotation, you can test everything about the JPA technology. It auto-configures all the repositories and the needed entities for you to test without starting a web server or any other dependency; in other words, `@DataJpaTest` is dedicated to testing the data layer. It also sets the `spring.jpa.show-sql` property to `true` so that you can see what the queries are

when the tests are been executed. Let's look at the `UserJpaRepositoryTests` class. See Listing 8-6.

Listing 8-6. src/test/java/apress/com/users/UserJpaRepositoryTests.java

```
package com.apress.users;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jdbc.
AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.testcontainers.service.connection.
ServiceConnection;
import org.springframework.context.annotation.Import;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

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

@Import({UserConfiguration.class})
@Testcontainers
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@DataJpaTest
public class UserJpaRepositoryTests {

    @Container
    @ServiceConnection
    static PostgreSQLContainer<?> postgreSQLContainer = new PostgreSQL
    Container<>("postgres:latest");

    @Autowired
    UserRepository userRepository;

    @Test
    void findAllTest(){
        var expectedUsers = userRepository.findAll();
        assertThat(expectedUsers).isNotEmpty();
        assertThat(expectedUsers).isInstanceOf(Iterable.class);
```

```
        assertThat(expectedUsers).element(0).isInstanceOf(User.class);
        assertThat(expectedUsers).element(0).matches( user -> user.
        isActive());
    }

    @Test
    void saveTest(){
        var dummyUser =  UserBuilder.createUser()
                .withName("Dummy")
                .withEmail("dummy@email.com")
                .active()
                .withRoles(UserRole.INFO)
                .withPassword("aw3sOm3R!")
                .build();
        var expectedUser = userRepository.save(dummyUser);
        assertThat(expectedUser).isNotNull();
        assertThat(expectedUser).isInstanceOf(User.class);
        assertThat(expectedUser).hasNoNullFieldsOrProperties();
        assertThat(expectedUser.isActive()).isTrue();
    }

    @Test
    void findByIdTest(){
        var expectedUser = userRepository.findById("norma@email.com");

        assertThat(expectedUser).isNotNull();
        assertThat(expectedUser.get()).isInstanceOf(User.class);
        assertThat(expectedUser.get().isActive()).isTrue();
        assertThat(expectedUser.get().getName()).isEqualTo("Norma");
    }

    @Test
    void deleteByIdTest(){
        var expectedUser = userRepository.findById("ximena@email.com");
        assertThat(expectedUser).isNotNull();
        assertThat(expectedUser.get()).isInstanceOf(User.class);
        assertThat(expectedUser.get().isActive()).isTrue();
        assertThat(expectedUser.get().getName()).isEqualTo("Ximena");
```

```
        userRepository.deleteById("ximena@email.com");

        expectedUser = userRepository.findById("ximena@email.com");
        assertThat(expectedUser).isNotNull();
        assertThat(expectedUser).isEmpty();

    }
}
```

The `UserJpaRepositoryTests` class includes the following:

- `@DataJpaTest`: This annotation auto-configures everything related to the JPA, from the repositories to the `EntityManager` (required as part of the JPA implementation), and even inherits from the `@Transactional` annotation, meaning that you are making sure your tests are completely transactional. With this annotation, you don't need any web layer to perform any of this action against your data persistence.
- `@Import`: One of the cool features of Spring is that you can import specific configurations with this annotation. In this case, we are importing the `UserConfiguration` where we are adding some users.
- `@Testcontainers`: As previously described, this annotation starts and stops the containers; this annotation looks for every `@Container` annotation defined. In this case we are marking this class as `Testcontainers`, and this will look for any `@Container` annotation and set up the environment for running the container image specified.
- `@AutoConfigureTestDatabase`: When testing the data layer, normally it is better to use an in-memory database, because it's fast and efficient; but sometimes you are required to do testing over a real database, and in this case the `@AutoConfigureTestDatabase` annotation will not use the embedded auto-configuration, but instead will follow what you specified with the `@Container` and `@ServerConnection`.

- `@Container`, `@ServerConnection`, `PostgreSQLContainer`: You already know these annotations. All these annotations will auto-configure the start and stop of the PostgreSQL container, the `DataSource` interface that required connection parameters such as `username`, `password`, `url`, `dialect`, `driverClass`, etc.
- `Assertions.*`: In this class we are using AssertJ, which comes with a fluent API to do assertions.

Spring Boot not only has support for the JPA, but also has the following annotations (and many more) that follow the same pattern, isolate your tests based on the technology you need to test for:

- `@JdbcTest`: For tests related to `JdbcTemplate` programming, covered in Chapters 4 and 5
- `@DataJdbcTest`: For tests on the Spring Data repositories
- `@JooqTest`: For all the jOOQ-related tests
- `@DataNeo4jTest`: For Spring Data Neo4J–related tests
- `@DataRedisTest`: For all Spring Data Redis–related tests
- `@DataLdapTest`: For LDAP tests

@WebFluxTest

If you have any `Reactive` web app with Spring Boot WebFlux, then you can test it using the `@WebFluxTest` annotation, which will auto-configure all the web-related beans and annotations for a WebFlux application. It auto-configures the `WebTestClient` interface to perform any exchange/request to the WebFlux endpoints, and it plays well with the `@MockBean` annotation for any services or repositories.

Let's look at the `RetroBoardWebFluxTests` class. See Listing 8-7.

Listing 8-7. src/test/java/apress/com/myretro/ RetroBoardWebFluxTests.java

```
package com.apress.myretro;

import com.apress.myretro.board.Card;
import com.apress.myretro.board.CardType;
import com.apress.myretro.board.RetroBoard;
```

```
import com.apress.myretro.service.RetroBoardService;
import com.apress.myretro.web.RetroBoardController;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.
WebFluxTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.test.web.reactive.server.WebTestClient;
import org.springframework.web.reactive.function.BodyInserters;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Arrays;
import java.util.UUID;

@WebFluxTest(controllers = {RetroBoardController.class})
public class RetroBoardWebFluxTests {

    @MockBean
    RetroBoardService retroBoardService;

    @Autowired
    private WebTestClient webClient;

    @Test
    void getAllRetroBoardTest(){

        Mockito.when(retroBoardService.findAll()).thenReturn(Flux.just(
                new RetroBoard(UUID.randomUUID(),"Simple Retro",
                Arrays.asList(
                        new Card(UUID.randomUUID(),"Happy to be here",
                        CardType.HAPPY),
                        new Card(UUID.randomUUID(),"Meetings everywhere",
                        CardType.SAD),
                        new Card(UUID.randomUUID(),"Vacations?",
                        CardType.MEH),
```

```
                        new Card(UUID.randomUUID(),"Awesome Discounts",
                        CardType.HAPPY),
                        new Card(UUID.randomUUID(),"Missed my train",
                        CardType.SAD)
                ))
        ));

        webClient.get()
                .uri("/retros")
                .accept(MediaType.APPLICATION_JSON)
                .exchange()
                .expectStatus().isOk()
                .expectBody().jsonPath("$[0].name").isEqualTo("Simple
                Retro");

        Mockito.verify(retroBoardService,Mockito.times(1)).findAll();
    }

    @Test
    void findRetroBoardByIdTest(){
        UUID uuid = UUID.randomUUID();
        Mockito.when(retroBoardService.findById(uuid)).
        thenReturn(Mono.just(
                new RetroBoard(uuid,"Simple Retro", Arrays.asList(
                        new Card(UUID.randomUUID(),"Happy to be here",
                        CardType.HAPPY),
                        new Card(UUID.randomUUID(),"Meetings everywhere",
                        CardType.SAD),
                        new Card(UUID.randomUUID(),"Vacations?",
                        CardType.MEH),
                        new Card(UUID.randomUUID(),"Awesome Discounts",
                        CardType.HAPPY),
                        new Card(UUID.randomUUID(),"Missed my train",
                        CardType.SAD)
                ))
        ));
```

```
    webClient.get()
            .uri("/retros/{uuid}",uuid.toString())
            .header(HttpHeaders.ACCEPT, MediaType.APPLICATION_
            JSON_VALUE)
            .exchange()
            .expectStatus().isOk()
            .expectBody(RetroBoard.class);

    Mockito.verify(retroBoardService,Mockito.times(1)).findById(uuid);
}

@Test
void saveRetroBoardTest(){
    RetroBoard retroBoard = new RetroBoard();
    retroBoard.setName("Simple Retro");

    Mockito.when(retroBoardService.save(retroBoard))
            .thenReturn(Mono.just(retroBoard));

    webClient.post()
            .uri("/retros")
            .contentType(MediaType.APPLICATION_JSON)
            .body(BodyInserters.fromValue(retroBoard))
            .exchange()
            .expectStatus().isOk();

    Mockito.verify(retroBoardService,Mockito.times(1)).
    save(retroBoard);
}

@Test
void deleteRetroBoardTest(){
    UUID uuid = UUID.randomUUID();
    Mockito.when(retroBoardService.delete(uuid)).thenReturn(Mono.
    empty());

    webClient.delete()
            .uri("/retros/{uuid}",uuid.toString())
```

```
                .exchange()
                .expectStatus().isOk();

        Mockito.verify(retroBoardService,Mockito.times(1)).delete(uuid);
    }

}
```

The `RetroBoardWebFluxTests` class includes the following:

- `@WebFluxTests`: This annotation configures everything related to a WebFlux application, looking for `@Controller`, `Filter`, etc. And you can add the controller you want to test. In this case, it is the `RetroBoardController` class. See that this test isolates from the data layer, only testing the web layer.
- `WebTestClient`: Recall from Chapter 7 that this class is used for testing HTTP and WebFlux endpoints and returns mocks of the response, making it easier to do the assertions and test our classes.
- `Mockito.*`: Again, we are using the Mockito library not only to prepare our call but also to verify that the call was executed the times we said we did.

@DataMongoTest

Similar to its support for SQL testing, Spring Boot provides testing support for NoSQL databases, such as the `@DataMongoTest` annotation to test only the MongoDB data layer.

Let's look at the `RetroBoardMongoTests` class. See Listing 8-8.

Listing 8-8. src/main/java/apress/com/myretro/RetroBoardMongoTests.java

```
package com.apress.myretro;

import com.apress.myretro.board.RetroBoard;
import com.apress.myretro.persistence.RetroBoardRepository;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.
DataMongoTest;
```

```
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;
import org.testcontainers.containers.MongoDBContainer;
import org.testcontainers.junit.jupiter.Container;
import reactor.core.publisher.Mono;
import reactor.test.StepVerifier;

import java.util.UUID;

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

@ActiveProfiles("mongoTest")
@DataMongoTest
public class RetroBoardMongoTests {

    @Container
    static MongoDBContainer mongoDBContainer = new MongoDBContainer("mongo
    :latest");

    static {
        mongoDBContainer.start();
    }

    @DynamicPropertySource
    static void setProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.data.mongodb.uri", mongoDBContainer::getRepli
        caSetUrl);
    }

    @Autowired
    RetroBoardRepository retroBoardRepository;

    @Test
    void saveRetroTest(){
        var name = "Spring Boot 3 Retro";
        RetroBoard retroBoard = new RetroBoard();
        retroBoard.setId(UUID.randomUUID());
        retroBoard.setName(name);
```

```
        var retroBoardResult = this.retroBoardRepository.
        insert(retroBoard).block();

        assertThat(retroBoardResult).isNotNull();
        assertThat(retroBoardResult.getId()).isInstanceOf(UUID.class);
        assertThat(retroBoardResult.getName()).isEqualTo(name);
    }

    @Test
    void findRetroBoardById(){
        RetroBoard retroBoard = new RetroBoard();
        retroBoard.setId(UUID.randomUUID());
        retroBoard.setName("Migration Retro");

        var retroBoardResult = this.retroBoardRepository.
        insert(retroBoard).block();
        assertThat(retroBoardResult).isNotNull();
        assertThat(retroBoardResult.getId()).isInstanceOf(UUID.class);

        Mono<RetroBoard> retroBoardMono = this.retroBoardRepository.
        findById(retroBoardResult.getId());

        StepVerifier
                .create(retroBoardMono)
                .assertNext( retro -> {
                    assertThat(retro).isNotNull();
                    assertThat(retro).isInstanceOf(RetroBoard.class);
                    assertThat(retro.getName()).isEqualTo("Migration
                    Retro");
                })
                .expectComplete()
                .verify();
    }
}
```

The `RetroBoardMongoTests` class includes the following:

- `@DataMongoTest`: This annotation auto-configures all the mongo layer, where it should find all related to Mongo data layer, from domain classes to repositories.
- `@ActiveProfiles`: You know this annotation will set the profile for a `mongoTest`. Useful to isolate tests.
- `@Container`: You also know this annotation, which will start and stop the container. In this case we statically declared the MongoDB container and omitted the `@Testcontainers` annotation, simply to demonstrate another way to achieve the same result.
- `@DynamcPropertiesSource`: With the `@ServiceConnection` annotation (introduced earlier in the chapter), the test will set up with the right mongo connection properties (the defaults), but in this case we are using another approach, a dynamic way to set the connection properties using the `@DynamicPropertiesSource` annotation. With this annotation, you can override all the defaults.

Using Testcontainers to Run Your Spring Boot Applications

So far, we have been using the `spring-boot-docker-compose` dependency to run the apps and the `spring-boot-testcontainers` dependency only for testing. With Testcontainers, Spring Boot provides a way to run your app without the need to use the `docker-compose`, directly from a container. How? It's very easy to do. Open the `RetroBoardTestConfiguration` class. See Listing 8-9.

Listing 8-9. src/main/java/apress/com/myretro/RetroBoardTestConfiguration.java

```
package com.apress.myretro;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.devtools.restart.RestartScope;
import org.springframework.boot.testcontainers.service.connection.
ServiceConnection;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.testcontainers.containers.MongoDBContainer;

@Profile({"!mongoTest"})
@Configuration
public class RetroBoardTestConfiguration {

        @Bean
        @RestartScope
        @ServiceConnection
        public MongoDBContainer mongoDBContainer(){
                return new MongoDBContainer("mongo:latest");
        }

        public static void main(String[] args) {
                SpringApplication.from(MyretroApplication::main).run(args);
        }
}
```

Let's analyze the `RetroBoardTestConfiguration` class:

- `@RestartScope`: Every time run your application the container resttart, and one way to avoid this situation is to use the `spring-boot-devtools` dependency, then using the @RestartScope will prevent of a container to restart for every test method you have in your Tests. The time required to run the tests will be shorter! because you don't need to wait for the container to restart.

- `@ServiceConnection`: As you already know, this annotation sets up all the connection parameters from the container you are trying to run, so you don't need to worry about these parameters.
- `SpringApplication.from`: This is very important! Notice that we are using a main method (yes, another entry point for our application), and in this case, instead of using `SpringApplication.run()` method, we are using the **`SpringApplication.from()`** method that is pointing to our main class (in this case, **`MyretroApplication::main`**).

Summary

In this chapter you learned all about the unit and integration testing support in Spring Boot. You discovered that Spring Boot testing is based on the Spring Framework Testing, and that Spring Boot auto-configuration helps to configure a lot of the testing.

You also learned that Spring Boot testing includes slice testing, which enables you to test all the layers in isolation and by technology—from your domain classes with the `@JsonTest annotation`, to the web controllers with `@WebMvcTest`, all the way to the data layer with `@DataJpaTest` annotations.

This chapter also introduced you to Testcontainers and how it can help not only for testing but also for running your application. You saw many ways to do testing thanks to the Spring Boot Testing Framework that offers the choice of AssertJ, Mockito, Hamcrest, and many more libraries for your tests.