Chapter 9. Testing Spring Boot Applications for Increased Production Readiness

This chapter discusses and demonstrates core aspects of testing Spring Boot applications. While the subject of testing has numerous facets, I focus upon the fundamental elements of testing Spring Boot applications that dramatically improve the production readiness of each application. Topics include unit testing, holistic application testing using `@SpringBootTest`, how to write effective unit tests using JUnit, and using Spring Boot testing slices to isolate test subjects and streamline testing.

CODE CHECKOUT CHECKUP

Please check out branch *chapter9begin* from the code repository to begin.

Unit Testing

Unit testing serves as a precursor to other types of application testing for good reason: unit testing enables a developer to find and fix bugs at the earliest possible stages of the develop+deploy cycle and as a result, to fix them at the lowest possible cost.

Simply put, *unit testing* involves validating a defined unit of code isolated to the maximum possible and sensible extent. A test's number of outcomes increases exponentially with size and complexity; reducing the amount of functionality within each unit test makes each one more manageable, thus increasing the likelihood that all likely and/or possible outcomes are considered.

Only once unit testing is implemented successfully and sufficiently should integration testing, UI/UX testing, and so on be added to the mix. Fortunately Spring Boot incorporates features to simplify and streamline unit testing and includes those capabilities in every project built using the Spring Initializr by default, making it easy for developers to get started quickly and "do the right thing".

Introducing @SpringBootTest

So far I’ve primarily focused on the code under *src/main/java* in projects created using the Spring Initializr, beginning with the main application class. In every Initializr-spawned Spring Boot application, however, there is a corresponding *src/***test***/java* directory structure with a single pre-created (but as yet empty) test.

Named to correspond with the main application class as well — for example, if the main app class is named `MyApplication`, the main test class will be `MyApplicationTest` — this default 1:1 correlation helps with both organization and consistency. Within the test class, the Initializr creates a single test method, empty to provide a clean start and so that development begins with a clean build. You can add more test methods, or more typically create additional test classes to parallel other application classes and create 1+ test methods within each.

Normally I would encourage Test Driven Development (TDD) in which tests are written first and code is written to (and only to) make tests pass. Since I firmly believe key aspects of Spring Boot are important to understand prior to the introduction of how Boot handles testing, I trust the reader will indulge my delay in introducing this chapter’s material until foundational topics were addressed.

With that in mind, let’s return to the Aircraft Positions application and write some tests.

NOTES ABOUT TEST COVERAGE

There are many persuasive arguments for every level of unit testing, from minimal to 100% test coverage. I consider myself a pragmatist, balanced on recognizing that too little is, well, too little; and acknowledging the falsity of the idea that “if some is good, more must be better”.

Everything has a cost. With too few tests the cost usually becomes apparent rather quickly: errors or edge cases slip through to production and often cause considerable headaches and unfortunate financial impacts. But writing tests for every accessor and mutator or for every element of exposed library/framework code can add burdensome costs to a project as well, often for very little (or zero) gain. Of course accessors and mutators can change, and of course underlying code can introduce bugs; but how often has that happened within your projects?

For this book and in my usual practice, I adopt a “test enough” mindset, purposely writing tests only for so-called interesting behavior. I generally do not write tests for domain classes, straightforward accessors/mutators, well-established Spring code, or anything else that would seem already to be very stable or (nearly?) foolproof, with a few notable exceptions which I explain at the time; see the earlier comment about interesting behavior. Note also that this assessment should be challenged and revisited in real projects, as software is not static and constantly evolves.

Only you and your organization can determine your risk profile and exposure.

In order to demonstrate the broadest swath of testing features enabled by Spring Boot in the clearest and most concise manner, I return to the JPA version of AircraftPositions and use it as the foundation for this chapter’s focus on testing. There are a few other testing-related topics that offer variations on a theme, complementary to this chapter’s content without being represented within its project; these related topics will be covered in an upcoming chapter.

Important Unit Tests for the Aircraft Positions Application

Within AircraftPositions there is currently only one class with what might be considered interesting behavior. `PositionController` exposes an

API to provide current aircraft positions to the end user directly or via web interface and within that API may perform actions including:

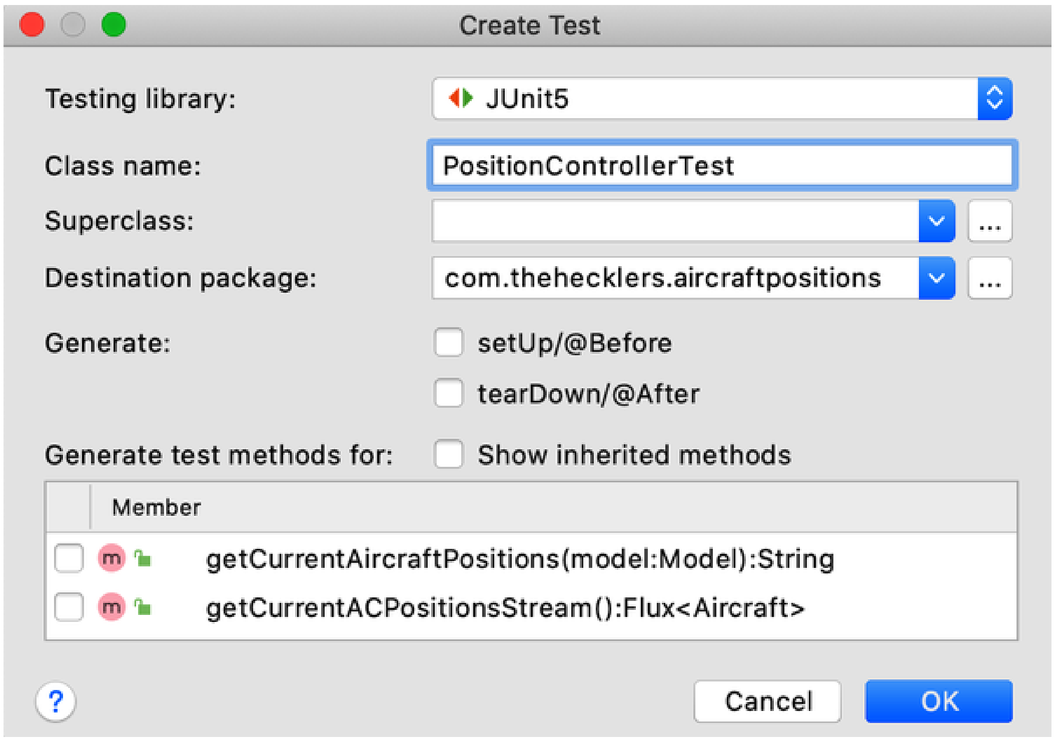
- Fetching current aircraft positions from PlaneFinder
- Storing the positions in a local database
- Retrieving the positions from the local database
- Returning current positions directly or by adding them to the document `Model` for a web page

Ignoring for the moment the fact that this functionality interacts with an external service, it also touches every layer of the application stack from user interface to data storage and retrieval. Recalling that a good testing approach should isolate and test small, cohesive bits of functionality, it's clear that an iterative approach to testing is in order, moving stepwise from the current state of code and no tests toward an eventual endstate of optimized application organization and testing. In this way, it accurately reflects typical production-targeted projects.

NOTE

Since applications in use are never really *done*, neither is testing. As an application's code evolves, tests must also be reviewed and potentially revised, removed, or added to maintain testing effectiveness.

I begin by creating a test class that parallels the `PositionController` class. The mechanism for creating a test class differs between IDEs, and of course it's possible to manually create one as well. Since I primarily use IntelliJ IDEA for development, I use the `CMD+N` keyboard shortcut or click the right mouse button and then "Generate" to open the Generate menu, then select the "Test…" option to create a test class. IntelliJ then presents the popup shown in Figure 9-1.


Create Test

Testing library: JUnit5

Class name: PositionControllerTest

Superclass:

Destination package: com.thehecklers.aircraftpositions

Generate: ☐ setUp/@Before ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

	Member
☐	getCurrentAircraftPositions(model:Model):String
☐	getCurrentACPositionsStream():Flux<Aircraft>

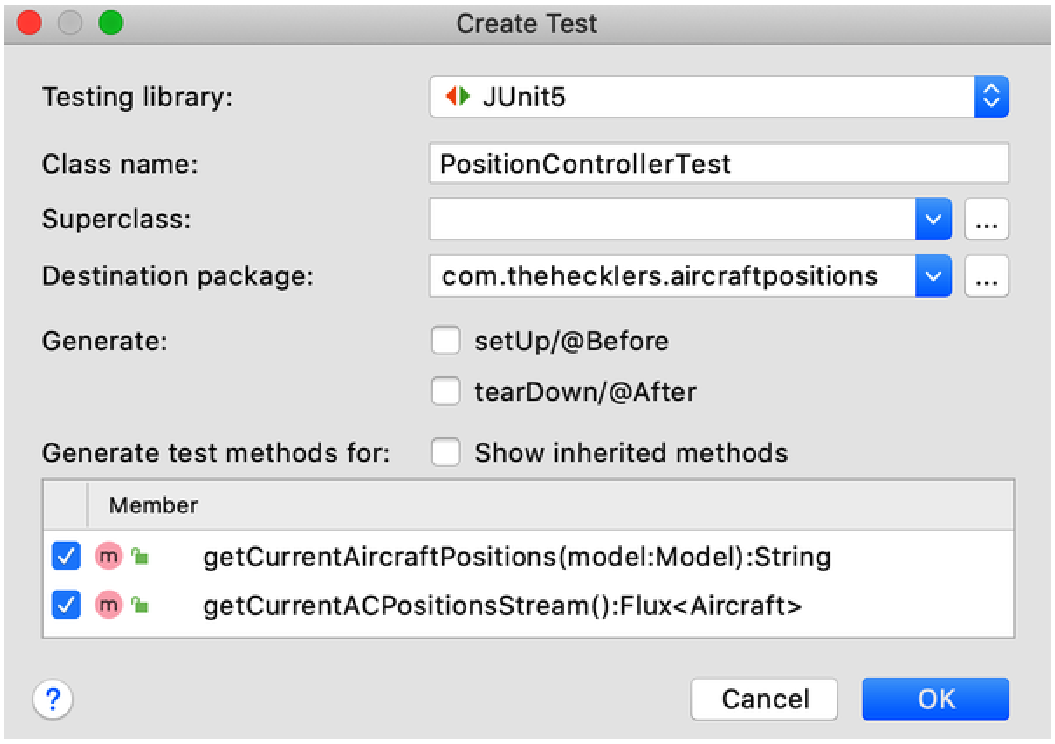
Figure 9-1. Create Test popup initiated from PositionController Class

From the *Create Test* popup, I keep the default “Testing library” option setting of JUnit 5. Since Spring Boot version 2.2 became generally available (GA), JUnit version 5 has been the default for Spring Boot application unit tests. Many other options are supported — including JUnit 3 and 4, Spock, and TestNG among others — but JUnit 5 with its Jupiter engine is a powerful option that offers several capabilities:

- Better testing of Kotlin code (compared to previous versions)
- More efficient once-only instantiation/configuration/cleanup of a test class for all contained tests, using `@BeforeAll` and `@AfterAll` method annotations
- Support for both JUnit 4 and 5 tests (unless JUnit 4 is specifically excluded from dependencies)

JUnit 5’s Jupiter engine is the default, with the vintage engine provided for backward compatibility with JUnit 4 unit tests.

I keep the suggested class name of `PositionControllerTest`, check the boxes to generate `setup/@Before` and `tearDown/@After` methods, and check the box to generate a test method for the `getCurrentAircraftPositions()` method as shown in Figure 9-2.


Create Test

Testing library: JUnit5

Class name: PositionControllerTest

Superclass:

Destination package: com.thehecklers.aircraftpositions

Generate: [ ] setUp/@Before [ ] tearDown/@After

Generate test methods for: [ ] Show inherited methods

	Member
✓	getCurrentAircraftPositions(model:Model):String
✓	getCurrentACPositionsStream():Flux<Aircraft>

Figure 9-2. Create Test popup with Desired Options Selected

Once I click the OK button, IntelliJ creates the `PositionControllerTest` class with the chosen methods and opens it in the IDE, as shown here:

```java
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class PositionControllerTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void getCurrentAircraftPositions() {
    }
}
```

To get a running start on building a test suite after-the-fact, I begin by simply reproducing to the extent possible the existing operation of the `PositionController` method `getCurrentAircraftPositions()`

within the same (literal) context it already successfully runs: the Spring Boot `ApplicationContext`.

NOTES ON APPLICATIONCONTEXT

Every Spring Boot application has an `ApplicationContext` that provides essential context — managing interactions with the environment, application components/beans, passing messages, etc. — and by default, the specific type of `ApplicationContext` required by an application is determined by Spring Boot's autoconfiguration.

When testing, the `@SpringBootTest` class-level annotation supports the `webEnvironment` parameter to allow selecting one of four options:

- `MOCK`
- `RANDOM_PORT`
- `DEFINED_PORT`
- `NONE`

The `MOCK` option is the default. `MOCK` loads a web `ApplicationContext` and leverages a mock web environment (rather than starting an embedded server) if a web environment is on the application's classpath; otherwise, it loads a regular `ApplicationContext` with no web capabilities. `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)` or just `@SpringBootTest` is often accompanied by `@AutoConfigureMockMVC` or `@AutoConfigureWebTestClient` to facilitate mock-based testing of web-based APIs using the corresponding mechanisms.

The `RANDOM_PORT` option loads a web `ApplicationContext` and starts an embedded server to provide an actual web environment exposed on a random available port. `DEFINED_PORT` does the same with one exception: it listens on the port defined in the application's *application.properties* or *application.yml/yaml* file. If no port is defined in those locations, the default port of 8080 is used.

Choosing `NONE` results in the creation of an `ApplicationContext` with no web environment at all, mock or actual. No embedded server is started.

I begin by adding the `@SpringBootTest` annotation at the class level. Since the initial goal is to reproduce as closely as possible the behavior present when the application executes, I specify the option to start an embedded server and have it listen on a random port. To test the web API, I plan to use the `WebTestClient`, which is similar to the `WebClient` used in the application but with a focus on testing:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_POR
@AutoConfigureWebTestClient
```

With only one unit test so far and no setup/teardown yet required, I turn attention to the test method for `getCurrentAircraftPositions()`:

```
@Test
void getCurrentAircraftPositions(@Autowired WebTestClient client) {
    assert client.get()
            .uri("/aircraft")
            .exchange()
            .expectStatus().isOk()
            .expectBody(Iterable.class)
            .returnResult()
            .getResponseBody()
            .iterator()
            .hasNext();
}
```

The first thing of note is that I autowire a `WebTestClient` bean for use within the method. This minimal effort on my part is all that is required to inject a `WebTestClient` bean from the `ApplicationContext`, due to the `@AutoConfigureWebTestClient` annotation I placed at the class level instructing Spring Boot to create and automatically configure a `WebTestClient`.

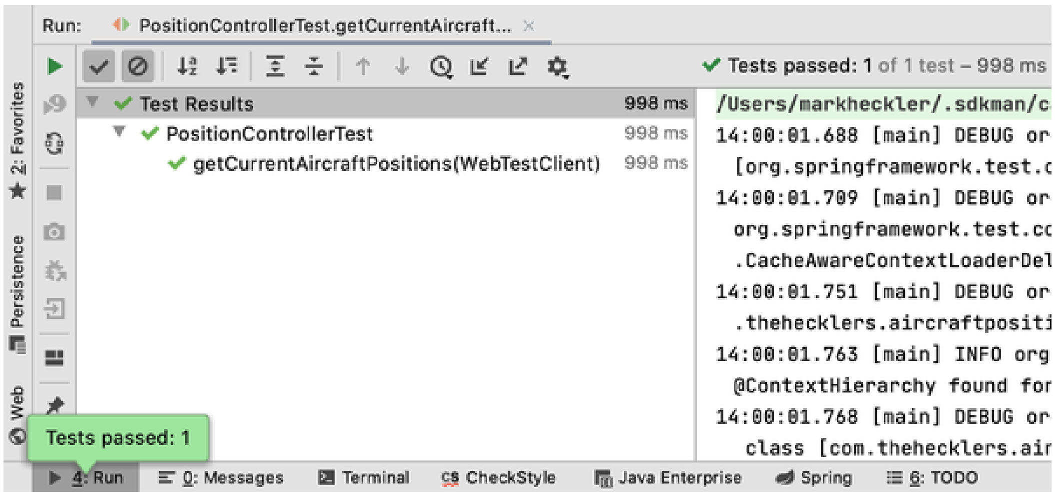
The single statement that is the entirety of the `@Test` method is an assertion that evaluates the expression that immediately follows. For the first iteration of this test, I use Java's `assert` to verify that final result of the chain of operations on the client is a `boolean` true value, thus resulting in a passing test.

The expression itself uses the injected `WebTestClient` bean, issuing a `GET` on the local endpoint */aircraft* serviced by the `PositionController`'s `getCurrentAircraftPositions()` method. Once the request/response exchange takes place, the HTTP status code is checked for a response of "OK" (200), the response body is verified to contain an `Iterable`, and the response is retrieved. Since the response consists of an `Iterable`, I use an `Iterator` to determine if there is at least one value contained within the `Iterable`. If so, the test passes.

WARNING

There are at least a couple small compromises in the current test. First, the test as currently written will fail if the external service that supplies aircraft positions (PlaneFinder) is unavailable, even if all code being tested in AircraftPositions is correct. This means that the test is not testing only the functionality it targets, but much more. Second, the extent of the test is somewhat limited since I test only that an Iterable is returned with 1+ element(s) and perform no examination of the element(s) themselves. This means that returning one element of any kind in an `Iterable`, or valid element(s) with invalid values, will result in a passing test. I'll remedy all of these shortcomings in iterations to follow.

Executing the test provides results similar to those shown in Figure 9-3, indicating that the test passed.



PositionControllerTest.getCurrentAircraft...

✓ Tests passed: 1 of 1 test – 998 ms

```
✓ Test Results                                        998 ms
  ✓ PositionControllerTest                            998 ms
    ✓ getCurrentAircraftPositions(WebTestClient)      998 ms
```

Tests passed: 1

```
/Users/markheckler/.sdkman/c
14:00:01.688 [main] DEBUG or
  [org.springframework.test.c
14:00:01.709 [main] DEBUG or
  org.springframework.test.cc
  .CacheAwareContextLoaderDel
14:00:01.751 [main] DEBUG or
  .thehecklers.aircraftpositi
14:00:01.763 [main] INFO org
  @ContextHierarchy found for
14:00:01.768 [main] DEBUG or
  class [com.thehecklers.air
```

Figure 9-3. First test passed

This is a good start, but even this single test can be improved significantly. Let's clean up this test before further expanding our unit testing mandate.

Refactoring for Better Testing

In the vast majority of cases, loading an entire `ApplicationContext` with embedded server and all capabilities present in the application to run a handful of tests is overkill. As mentioned before, unit tests should be focused and to the extent possible, self-contained. The smaller the surface area and fewer the external dependencies, the more targeted the tests can be. This laserlike focus offers several benefits, including fewer overlooked scenarios/outcomes, greater potential specificity and rigor in testing, more readable and thus understandable tests, and no less importantly speed.

I mentioned earlier that it's counterproductive to write low- and no-value tests, although what that means is dependent upon context. One thing that can discourage developers from adding useful tests, however, is the amount of time it can take to execute the test suite. Once a certain threshold is reached — and such boundary is also context dependent — a developer may hesitate to add to the already significant time burden required to get a clean build. Fortunately Spring Boot has several means to simultaneously increase test quality and decrease test execution times.

If no calls using `WebClient` or `WebTestClient` were required to fulfill the demands of AircraftPosition's API, the next logical step would likely be to remove the `webEnvironment` parameter within the class-level `@SpringBootTest` annotation. This would result in a basic `ApplicationContext` being loaded for the `PositionControllerTest` class's tests using a `MOCK` web environment, reducing the footprint and load time required. Since `WebClient` is a key part of the API and thus `WebTestClient` becomes the best way to test it, I instead replace the `@SpringBootTest` and `@AutoConfigureWebTestClient` class-level annotations with `@WebFluxTest` to streamline the `ApplicationContext` while autoconfiguring and providing access to the `WebTestClient`:

```
@WebFluxTest({PositionController.class})
```

One other thing of note with the `@WebFluxTest` annotation: among other things, it can accept a parameter of `controllers` pointing to an array of `@Controller` bean types to be instantiated for use by the annotated test class. The actual `controllers =` portion can be omitted, as I have, leaving only the array of `@Controller` classes — in this case only the one, `PositionController`.

Revisiting the code to isolate behavior

As mentioned earlier, the code for `PositionController` does several things, including making multiple database calls and directly using `WebClient` to access an external service. In order to better isolate the API from underlying actions so mocking becomes more granular and thus both easier and clearer, I refactor `PositionController` to remove direct definition and use of a `WebClient` and move the entirety of the `getCurrentAircraftPositions()` method's logic to a

`PositionRetriever` class, which is then injected into and used by `PositionController`:

```
import lombok.AllArgsConstructor;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@AllArgsConstructor
@RestController
public class PositionController {
    private final PositionRetriever retriever;

    @GetMapping("/aircraft")
    public Iterable<Aircraft> getCurrentAircraftPositions() {
        return retriever.retrieveAircraftPositions();
    }
}
```

The first mock-ready version of `PositionRetriever` largely consists of the code that had previously been in `PositionController`. The primary goal for this step is to facilitate mocking of the `retrieveAircraftPositions()` method; by removing this logic from the `getCurrentAircraftPositions()` method in `PositionController`, an upstream call can be mocked instead of the web API, thus enabling testing of the `PositionController`:

```
import lombok.AllArgsConstructor;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@AllArgsConstructor
@Component
public class PositionRetriever {
    private final AircraftRepository repository;
    private final WebClient client =
            WebClient.create("http://localhost:7634");

    Iterable<Aircraft> retrieveAircraftPositions() {
        repository.deleteAll();

        client.get()
                .uri("/aircraft")
                .retrieve()
                .bodyToFlux(Aircraft.class)
                .filter(ac -> !ac.getReg().isEmpty())
                .toStream()
```

```
                    .forEach(repository::save);

            return repository.findAll();
        }
    }
```

With these changes to the code, the existing testing can be revised to isolate the Aircraft Positions application's functionality from external services and focus specifically upon the web API by mocking other components/functionality accessed by the web API, thus streamlining and speeding test execution.

Refining the test

Since the focus is on testing the web API, the more logic that isn't an actual web interaction that can be mocked, the better. `PositionController::getCurrentAircraftPositions` now calls on `PositionRetriever` to provide it with current aircraft positions upon request, so `PositionRetriever` is the first component to mock. Mockito's `@MockBean` annotation — Mockito is included automatically with the Spring Boot testing dependency — replaces the `PositionRetriever` bean that normally would be created on application startup with a mocked stand-in, which is then automatically injected:

```
@MockBean
private PositionRetriever retriever;
```

NOTE

Mock beans are automatically reset after each test method is executed.

I then turn my attention to the method that provides aircraft positions, `PositionRetriever::retrieveAircraftPositions`. Since I now inject a `PositionRetriever` mock for testing instead of the real thing, I must provide an implementation for the `retrieveAircraftPositions()` method so that it responds in a predictable and testable manner when it is called by the `PositionController`.

I create a couple aircraft positions to use as sample data for tests within the `PositionControllerTest` class, declaring the `Aircraft` variables at the class level and assigning representative values to them within the `setUp()` method:

```
    private Aircraft ac1, ac2;

    @BeforeEach
    void setUp(ApplicationContext context) {
        // Spring Airlines flight 001 en route, flying STL to SFO,
        //   at 30000' currently over Kansas City
        ac1 = new Aircraft(1L, "SAL001", "sqwk", "N12345", "SAL001",
                "STL-SFO", "LJ", "ct",
                30000, 280, 440, 0, 0,
                39.2979849, -94.71921, 0D, 0D, 0D,
                true, false,
                Instant.now(), Instant.now(), Instant.now());

        // Spring Airlines flight 002 en route, flying SFO to STL,
        //   at 40000' currently over Denver
        ac2 = new Aircraft(2L, "SAL002", "sqwk", "N54321", "SAL002",
                "SFO-STL", "LJ", "ct",
                40000, 65, 440, 0, 0,
                39.8560963, -104.6759263, 0D, 0D, 0D,
                true, false,
                Instant.now(), Instant.now(), Instant.now());
    }
```

NOTE

The number of aircraft positions retrieved in actual operation of the applications under development is nearly always more than one, often significantly more. Bearing that in mind, a sample data set used in testing should *at a minimum* return a number of positions of two. Edge cases involving zero, one, or very large numbers of positions should be considered for additional tests in subsequent iterations for similar production applications.

Now, back to the `retrieveAircraftPositions()` method. Mockito's `when...thenReturn` combination returns a specified response when a specified condition is met. With sample data now defined, I can provide both the condition and the response to return to calls to `PositionRetriever::retrieveAircraftPositions`:

```
@BeforeEach
void setUp(ApplicationContext context) {
    // Aircraft variable assignments omitted for brevity
    ...

    Mockito.when(retriever.retrieveAircraftPositions())
        .thenReturn(List.of(ac1, ac2));
}
```

With the relevant method mocked, it's time to return attention to the unit test located in `PositionControllerTest::getCurrentAircraftPositions`.

Since I've instructed the test instance to load the `PositionController` bean with the class-level annotation `@WebFluxTest(controllers = {PositionController.class})` and have created a mock `PositionRetriever` bean and defined its behavior, I can now refactor the portion of the test that retrieves positions with some certainty of what will be returned:

```
@Test
void getCurrentAircraftPositions(@Autowired WebTestClient client) {
    final Iterable<Aircraft> acPositions = client.get()
            .uri("/aircraft")
            .exchange()
            .expectStatus().isOk()
            .expectBodyList(Aircraft.class)
            .returnResult()
            .getResponseBody();

    // Still need to compare with expected results
}
```

The chain of operators shown should retrieve a `List<Aircraft>` consisting of `ac1` and `ac2`. In order to confirm the correct results, I need to compare `acPositions` — the actual outcome — with that expected outcome. One way of doing so is with a simple comparison such as this:

```
assertEquals(List.of(ac1, ac2), acPositions);
```

This works correctly and the test will pass. I could also have taken things a bit further in this intermediate step by comparing the actual results with results obtained via a mocked call to `AircraftRepository`. Adding the following bits of code to the class, the `setUp()` method, and the `getCurrentAircraftPositions()` test method produces similar (passing) test results:

```
@MockBean
private AircraftRepository repository;

@BeforeEach
void setUp(ApplicationContext context) {
    // Existing setUp code omitted for brevity

    ...

    Mockito.when(repository.findAll()).thenReturn(List.of(ac1, ac2));
}

@Test
void getCurrentAircraftPositions(@Autowired WebTestClient client) {
    // client.get chain of operations omitted for brevity

    ...

    assertEquals(repository.findAll(), acPositions);
}
```

NOTE

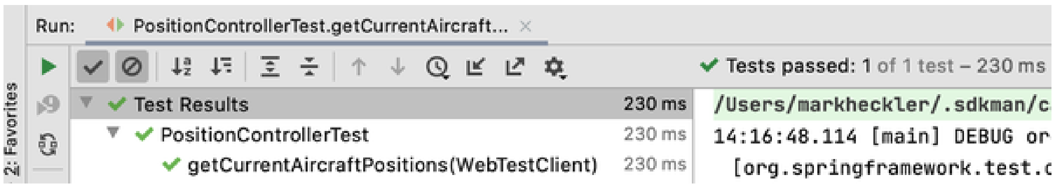
This variant also results in a passing test, but it somewhat contradicts the principle of focused testing, since I now mix the concepts of testing the repository with testing the web API. Since it doesn't actually use the `CrudRepository::findAll` method but simply mocks it, the value of testing it doesn't add any discernible value, either. However, you may encounter tests of this nature at some point, so I thought it worthwhile to show and discuss.

The current working version of `PlaneControllerTest` should now look like this:

```
import org.junit.jupiter.api.BeforeEach;
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.WebFluxTe
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.context.ApplicationContext;
import org.springframework.test.web.reactive.server.WebTestClient;

import java.time.Instant;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
```

```java
@WebFluxTest(controllers = {PositionController.class})
class PositionControllerTest {
    @MockBean
    private PositionRetriever retriever;

    private Aircraft ac1, ac2;

    @BeforeEach
    void setUp(ApplicationContext context) {
        // Spring Airlines flight 001 en route, flying STL to SFO,
        //    at 30000' currently over Kansas City
        ac1 = new Aircraft(1L, "SAL001", "sqwk", "N12345", "SAL001",
                "STL-SFO", "LJ", "ct",
                30000, 280, 440, 0, 0,
                39.2979849, -94.71921, 0D, 0D, 0D,
                true, false,
                Instant.now(), Instant.now(), Instant.now());

        // Spring Airlines flight 002 en route, flying SFO to STL,
        //    at 40000' currently over Denver
        ac2 = new Aircraft(2L, "SAL002", "sqwk", "N54321", "SAL002",
                "SFO-STL", "LJ", "ct",
                40000, 65, 440, 0, 0,
                39.8560963, -104.6759263, 0D, 0D, 0D,
                true, false,
                Instant.now(), Instant.now(), Instant.now());

        Mockito.when(retriever.retrieveAircraftPositions())
            .thenReturn(List.of(ac1, ac2));
    }

    @Test
    void getCurrentAircraftPositions(@Autowired WebTestClient client) {
        final Iterable<Aircraft> acPositions = client.get()
                .uri("/aircraft")
                .exchange()
                .expectStatus().isOk()
                .expectBodyList(Aircraft.class)
                .returnResult()
                .getResponseBody();

        assertEquals(List.of(ac1, ac2), acPositions);
    }
}
```

Running it once again produces a passing test, with results similar to those shown in Figure 9-4.



PositionControllerTest.getCurrentAircraft...

Tests passed: 1 of 1 test – 230 ms

Test Results 230 ms
PositionControllerTest 230 ms
getCurrentAircraftPositions(WebTestClient) 230 ms

```
/Users/markheckler/.sdkman/c
14:16:48.114 [main] DEBUG or
  [org.springframework.test.c
```

Figure 9-4. New, improved test for AircraftRepository::getCurrentAircraftPositions

As the web API required to meet application/user requirements expands, unit tests should be specified first (before creating the actual code to fulfill those requirements) to ensure correct outcomes.

Testing Slices

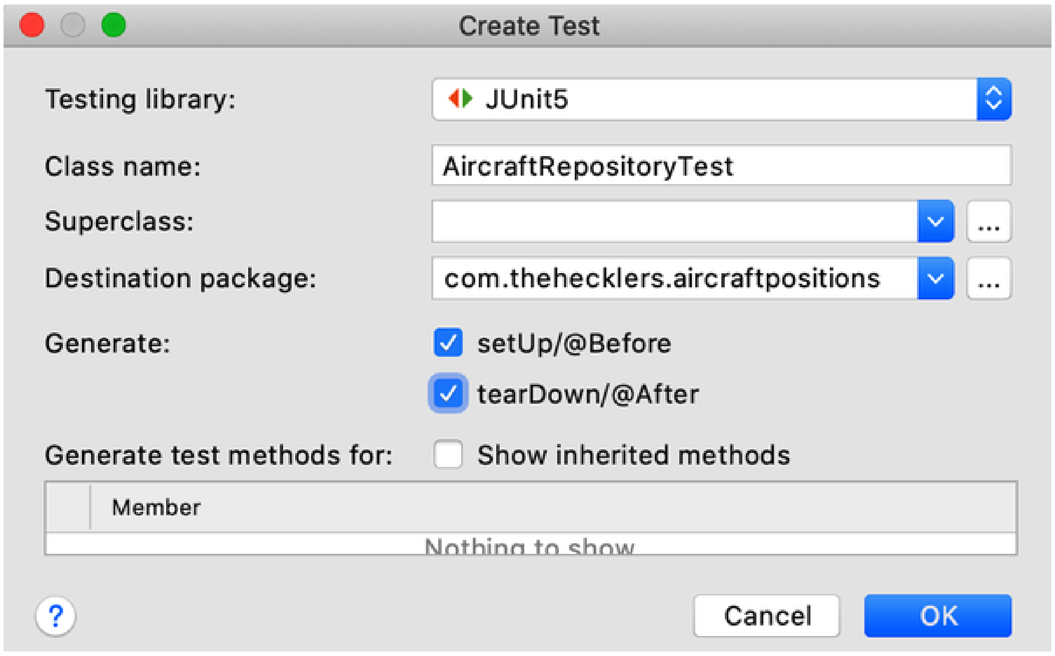
I've already mentioned a few times the importance of focused testing, and Spring has another mechanism that helps developers accomplish that quickly and painlessly: test slices.

Several annotations are built into Spring Boot's testing dependency `spring-boot-starter-test` that automatically configure these slices of functionality. All of these test slice annotations work in similar fashion, loading an `ApplicationContext` and select components that make sense for the specified slice. Examples include:

- `@JsonTest`
- `@WebMvcTest`
- `@WebFluxText` (previously introduced)
- `@DataJpaTest`
- `@JdbcTest`
- `@DataJdbcTest`
- `@JooqTest`
- `@DataMongoTest`
- `@DataNeo4jTest`
- `@DataRedisTest`
- `@DataLdapTest`
- `@RestClientTest`
- `@AutoConfigureRestDocs`
- `@WebServiceClientTest`

During an earlier section leveraging `@WebFluxTest` to exercise and validate the web API, I mentioned testing datastore interactions and excluded doing so from the test, since it was focused on testing web interactions. To better demonstrate data testing and how test slices facilitate targeting specific functionality, I explore that next.

Since the current iteration of Aircraft Positions uses JPA and H2 to store and retrieve current positions, `@DataJpaTest` is a perfect fit. I begin by creating a new class for testing using IntelliJ IDEA, opening the `AircraftRepository` class and using the same approach to create a test class as before: CMD+N, "Test...", leaving JUnit5 as the "Testing Library" and other default values in place, and selecting *setUp/@Before* and *tearDown/@After* options as shown in Figure 9-5.



Testing library: JUnit5
Class name: AircraftRepositoryTest
Superclass:
Destination package: com.thehecklers.aircraftpositions
Generate: [✓ setUp/@Before] [✓ tearDown/@After]
Generate test methods for: Show inherited methods
Member
Nothing to show

Figure 9-5. Create Test popup for AircraftRepository

NOTE

No methods are shown because Spring Data `Repository` beans provide common methods to Spring Boot applications via autoconfiguration. I will add test methods to exercise these as an example next, and if you create custom repository methods, these can (and should be) tested as well.

Clicking the OK button generates the test class `AircraftRepositoryTest`:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;

class AircraftRepositoryTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }
}
```

The first order of business is, of course, to add the test slice annotation `@DataJpaTest` to the `AircraftRepositoryTest` class:

```
@DataJpaTest
class AircraftRepositoryTest {

    ...

}
```

As a result of adding this single annotation, upon execution the test will scan for `@Entity` classes and configure Spring Data JPA repositories — in the Aircraft Positions application, `Aircraft` and `AircraftRepository` respectively. If an embedded database is in the classpath (as H2 is here), the test engine will configure it as well. Typical `@Component` annotated classes are not scanned for bean creation.

In order to test actual repository operations, the repository mustn't be mocked; and since the `@DataJpaTest` annotation loads and configures an `AircraftRepository` bean, there is no need to mock it anyway. I inject the repository bean using `@Autowire` and just as in the `PositionController` test earlier, declare `Aircraft` variables to ultimately serve as test data:

```
@Autowired
private AircraftRepository repository;

private Aircraft ac1, ac2;
```

To setup the proper environment for the tests that will exist within this `AircraftRepositoryTest` class, I create two `Aircraft` objects, assign each to one of the declared member variables, and then save them to the repository within the `setUp()` method using `Repository::saveAll`.

```
@BeforeEach
void setUp() {
    // Spring Airlines flight 001 en route, flying STL to SFO,
    // at 30000' currently over Kansas City
    ac1 = new Aircraft(1L, "SAL001", "sqwk", "N12345", "SAL001",
            "STL-SFO", "LJ", "ct",
            30000, 280, 440, 0, 0,
            39.2979849, -94.71921, 0D, 0D, 0D,
            true, false,
            Instant.now(), Instant.now(), Instant.now());

    // Spring Airlines flight 002 en route, flying SFO to STL,
    // at 40000' currently over Denver
    ac2 = new Aircraft(2L, "SAL002", "sqwk", "N54321", "SAL002",
            "SFO-STL", "LJ", "ct",
            40000, 65, 440, 0, 0,
            39.8560963, -104.6759263, 0D, 0D, 0D,
            true, false,
            Instant.now(), Instant.now(), Instant.now());

    repository.saveAll(List.of(ac1, ac2));
}
```

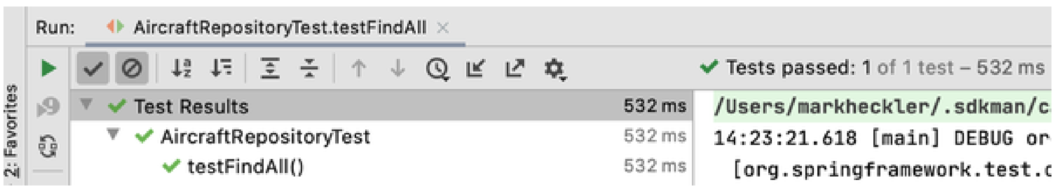
Next, I create a test method to verify that what is returned as a result of executing a `findAll()` on the `AircraftRepository` bean is exactly what should be returned: an `Iterable<Aircraft>` containing the two aircraft positions saved in the test's `setUp()` method:

```
@Test
void testFindAll() {
    assertEquals(List.of(ac1, ac2), repository.findAll());
}
```

NOTE

`List` extends `Collection` which in turn extends `Iterable`.

Running this test provides a passing result that looks something like that shown in Figure 9-6.



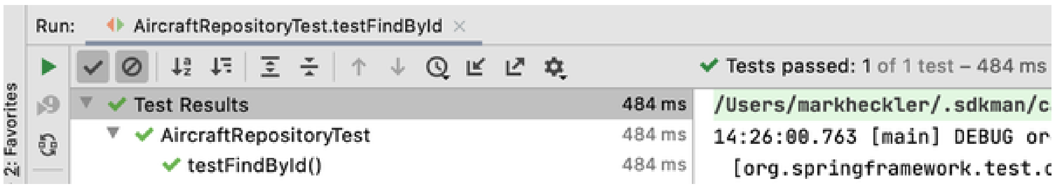
```
Run:   AircraftRepositoryTest.testFindAll ×
                                                     ✓ Tests passed: 1 of 1 test – 532 ms
▼ ✓ Test Results                          532 ms   /Users/markheckler/.sdkman/c
    ▼ ✓ AircraftRepositoryTest            532 ms   14:23:21.618 [main] DEBUG or
          ✓ testFindAll()                 532 ms     [org.springframework.test.c
```

Figure 9-6. Test results for `findAll()`

Similarly, I create a test for the `AircraftRepository` method to find a particular record by its ID field, `findById()`. Since there should be two records stored due to the `Repository::saveAll` method called in the test class's `setUp()`, I query for both and verify the results against expected values.

```
@Test
void testFindById() {
    assertEquals(Optional.of(ac1), repository.findById(ac1.getId()));
    assertEquals(Optional.of(ac2), repository.findById(ac2.getId()));
}
```

Running the `testFindById()` test yields a passing as well, as shown in Figure 9-7.



```
Run:  AircraftRepositoryTest.testFindById ×
                                              ✓ Tests passed: 1 of 1 test – 484 ms
▼ ✓ Test Results                     484 ms   /Users/markheckler/.sdkman/c
  ▼ ✓ AircraftRepositoryTest         484 ms   14:26:00.763 [main] DEBUG or
      ✓ testFindById()               484 ms     [org.springframework.test.c
```

Figure 9-7. Test results for `findById()`

TESTING THE TESTS

When a test passes, most developers assume that their code has been validated. But a passing test can be indicative of one of two possible things:

- The code works
- The test doesn't

As such, I strongly encourage *breaking the test* whenever possible to verify that the test actually is doing what it is supposed to do.

What do I mean by that?

The easiest example is to provide incorrect expected results. If the test suddenly breaks, examine the failure. If it failed in the expected manner, restore the correct functionality and verify the test once again works. However, if the test still passes after providing what should have been failing criteria, correct the test and re-verify. Once it breaks as expected, restore the correct expected results and run once more to confirm that the test is testing the right things in the right way.

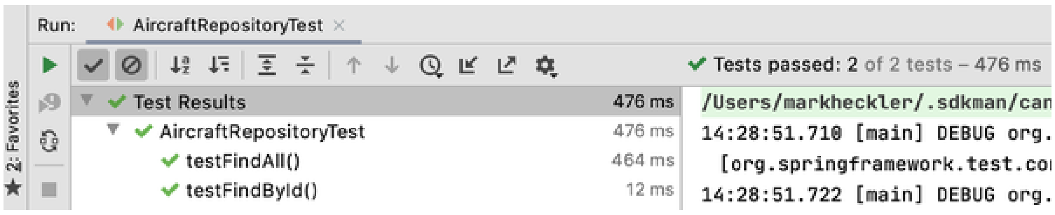
Note that this is not an unusual occurrence, and it's far more gratifying to find bad tests when writing them than when troubleshooting and trying to determine how something slipped through testing to fail in production.

Finally, a bit of cleanup is in order once all tests have run. To the `tearDown()` method I add a single statement to delete all records in the `AircraftRepository`:

```
@AfterEach
void tearDown() {
    repository.deleteAll();
}
```

Note that it really isn't necessary in this case to erase all records from the repository, since it's an in-memory instance of the H2 database that is reinitialized before each test. This is however representative of the type of operation that would typically be placed in a test class's `tearDown()` method.

Executing all tests within `AircraftRepositoryTest` produces passing results similar to those shown in Figure 9-8.



Tests passed: 2 of 2 tests – 476 ms

```
Test Results                476 ms
  AircraftRepositoryTest    476 ms
    testFindAll()           464 ms
    testFindById()           12 ms
```

```
/Users/markheckler/.sdkman/can
14:28:51.710 [main] DEBUG org.
  [org.springframework.test.co
14:28:51.722 [main] DEBUG org.
```

Figure 9-8. Test results for all tests in AircraftRepositoryTest

TEST TIMES: THE NEED FOR SPEED

As mentioned early in this chapter, reducing the scope of each test, along with the number of beans the test engine must load into the `ApplicationContext` to perform each test, improves the speed and fidelity of each test. Fewer unknowns mean a more comprehensive test suite, and faster tests mean that the test suite can do more in less time, working harder to save you headaches down the line.

As a quick measure of time savings seen so far, the initial version of `PositionControllerTest` took 998ms — almost a full second — to load the full `ApplicationContext`, execute the test, and shut down. Doing a bit of refactoring to code and tests improved the application's modularity and honed the focus of the pertinent test, simultaneously reducing test execution time to 230ms — now less than 1/4 of a second. Saving more than 3/4 of a second every time a test is run adds up, and multiplied across several tests and several builds it makes a significant and welcome contribution to development velocity.

Testing is never complete for an application that is still evolving. For the functionality currently present in Aircraft Positions, however, the tests written in this chapter provide a good starting point for code validation and continued expansion as functionality is added to the application.

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter9end* from the code repository.

Summary

This chapter discussed and demonstrated core aspects of testing Spring Boot applications, focusing on the fundamental aspects of testing Spring

Boot applications that most improve the production readiness of each application. Topics covered included unit testing, holistic application testing using `@SpringBootTest`, how to write effective unit tests using JUnit, and using Spring Boot testing slices to isolate test subjects and streamline testing.

The next chapter explores security concepts like Authentication and Authorization. I then demonstrate how to implement forms-based authentication for self-contained applications and for the most demanding requirements, how to leverage OpenID Connect and OAuth2 for maximum security and flexibility, all using Spring Security.