Chapter 12. Going Deeper with Reactive

As previously discussed, reactive programming gives developers a way to make better use of resources in distributed systems, even extending powerful scaling mechanisms across application boundaries and into the communication channels. For developers with experience exclusively with mainstream Java development practices—often called *imperative* Java due to its explicit and sequential logic versus the more declarative approach generally used in reactive programming, although this label, like most, is imperfect—these reactive capabilities may bear some undesired costs. In addition to the expected learning curve, which Spring helps flatten considerably due to parallel and complementary WebMVC and WebFlux implementations, there are also relative limitations in tooling, its maturity, and established practices for essential activities like testing, troubleshooting, and debugging.

While it is true that reactive Java development is in its infancy relative to its imperative cousin, the fact that they are family has allowed a much faster development and maturation of useful tooling and processes. As mentioned, Spring builds similarly on established imperative expertise within its development and community to condense decades of evolution into production-ready components available *now*.

This chapter introduces and explains the current state of the art in testing and diagnosing/debugging issues you might encounter as you begin to deploy reactive Spring Boot applications and demonstrates how to put WebFlux/Reactor to work for you, even before—and to help—you go to production.

CODE CHECKOUT CHECKUP

Please check out branch *chapter12begin* from the code repository to begin.

When Reactive?

Reactive programming, and in particular those applications focusing on reactive streams, enables system-wide scaling that is difficult to match using other means available at this point in time. However, not all applications need to perform at the far reaches of end-to-end scalability, or they may already be performing (or are expected to perform) admirably with relatively predictable loads over impressive time frames. Imperative

apps have long fulfilled production demands for organizations globally, and they will not simply cease to do so because a new option arrives.

While reactive programming is unquestionably exciting in terms of the possibilities it offers, the Spring team clearly states that reactive code will not replace all imperative code for the foreseeable future, if ever. As stated in the Spring Framework reference documentation for Spring WebFlux:

> *If you have a large team, keep in mind the steep learning curve in the shift to non-blocking, functional, and declarative programming. A practical way to start without a full switch is to use the reactive WebClient. Beyond that, start small and measure the benefits. We expect that, for a wide range of applications, the shift is unnecessary. If you are unsure what benefits to look for, start by learning about how non-blocking I/O works (for example, concurrency on single-threaded Node.js) and its effects.*
>
> —Spring Framework Reference Documentation

In short, adopting reactive programming and Spring WebFlux is a choice—a great choice that provides perhaps the best way to accomplish certain requirements—but still a choice to make after careful consideration of the relevant requirements and demands for the system in question. Reactive or not, Spring Boot provides unsurpassed options to develop business-critical software to handle all of your production workloads.

Testing Reactive Applications

In order to better focus on the key concepts of testing reactive Spring Boot applications, I take several steps to tighten the scope of code under consideration. Like zooming in on a subject you wish to photograph, other project code is still present but is not on the critical path for the information in this section.

ADDITIONAL NOTES ON TESTING

I covered testing and, to some degree, my testing philosophy in Chapter 9. To delve more deeply into the aspects of testing covered in this chapter, I must share more of my thinking in order for steps taken here to be clear. Since this book is focused primarily on Spring Boot and only secondarily on related topics, I have attempted (and will continue to attempt) to find the "just enough" amount of additional information needed to provide context without unnecessary elaboration. As the reader might imagine, such a balancing point is impossible to find because it differs by reader, but I hope to get as close as humanly possible.

Testing informs code structure. When done as true Test Driven Development (TDD), this structural guidance occurs from the very beginning of application development. Fleshing out tests once code is in place—as I have done in several chapters of this book in order to place full emphasis on the Spring Boot concepts to be shared rather than applicable test harnesses—can result in greater code refactoring efforts to better isolate and decouple behavior to test specific components and outcomes. This may feel disruptive, but it typically results in better code with cleaner boundaries, making it both more testable and more robust.

This chapter's code is no exception. In order to isolate and test properly for desired behavior, some refactoring of existing, working code is in order. It doesn't take long, and the end results are provably better.

For this section I'll zero in specifically on testing externally those APIs that expose reactive streams publishers—`Flux`, `Mono`, and `Publisher` types that could be either `Flux` or `Mono`—instead of the typical blocking `Iterable` or `Object` types. I begin with the class within `AircraftPositions` that provides the external APIs: `PositionController`.

TIP

If you haven't already checked out the Chapter 12 code as indicated at the beginning of this chapter, please do so now.

But First, Refactoring

While the code within `PositionController` does work, it is a bit of a testing muddle. The first order of business is to provide a cleaner separation of concerns, and I begin by removing the code to create an `RSocketRequester` object to an `@Configuration` class that will create it as a Spring bean, accessible anywhere within the application:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.rsocket.RSocketRequester;

@Configuration
public class RSocketRequesterConfig {
    @Bean
    RSocketRequester requester(RSocketRequester.Builder builder) {
        return builder.tcp("localhost", 7635);
    }
}
```

This streamlines the constructor for `PositionController`, placing the work for creating the `RSocketRequester` where it belongs and well outside of a controller class. To use the `RSocketRequester` bean in `PositionController`, I simply autowire it in using Spring Boot's constructor injection:

```
public PositionController(AircraftRepository repository,
                          RSocketRequester requester) {
    this.repository = repository;
    this.requester = requester;
}
```

NOTE

Testing the RSocket connection would require integration testing. While this section focuses on unit testing and not integration testing, it is still essential to decouple the construction of the `RSocketRequester` from `PositionController` in order to isolate and properly unit test `PositionController`.

There is another source of logic that falls well outside of controller functionality that remains, this time involving the acquisition, then the storing and retrieving, of aircraft positions using the `AircraftRepository` bean. Typically when complex logic unrelated to a particular class finds its way into that class, it's best to extract it, as I did for the `RSocketRequester` bean. To relocate this somewhat complex and unrelated code outside of `PositionController`, I create a `PositionService` class and define it as a `@Service` bean available throughout the application. The `@Service` annotation is simply a more visually specific description of the oft-used `@Component` annotation:

```
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
```

```
public class PositionService {
    private final AircraftRepository repo;
    private WebClient client = WebClient.create(
        "http://localhost:7634/aircraft");

    public PositionService(AircraftRepository repo) {
        this.repo = repo;
    }

    public Flux<Aircraft> getAllAircraft() {
        return repo.deleteAll()
                .thenMany(client.get()
                        .retrieve()
                        .bodyToFlux(Aircraft.class)
                        .filter(plane -> !plane.getReg().isEmpty()))
                .flatMap(repo::save)
                .thenMany(repo.findAll());
    }

    public Mono<Aircraft> getAircraftById(Long id) {
        return repo.findById(id);
    }

    public Flux<Aircraft> getAircraftByReg(String reg) {
        return repo.findAircraftByReg(reg);
    }
}
```

NOTE

Currently there is no `findAircraftByReg()` method defined within `AircraftRepository`. I address that prior to creating tests.

Although more work could be done (especially with regard to the `WebClient` member variable), it is sufficient for now to remove the complex logic shown in `PositionService::getAllAircraft` from its former home within `PositionController::getCurrentAircraftPositions` and inject the `PositionService` bean into the controller for its use, resulting in a much cleaner and focused controller class:

```
import org.springframework.http.MediaType;
import org.springframework.messaging.rsocket.RSocketRequester;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import reactor.core.publisher.Flux;

@Controller
public class PositionController {
```

```
    private final PositionService service;
    private final RSocketRequester requester;

    public PositionController(PositionService service,
            RSocketRequester requester) {
        this.service = service;
        this.requester = requester;
    }

    @GetMapping("/aircraft")
    public String getCurrentAircraftPositions(Model model) {
        model.addAttribute("currentPositions", service.getAllAircraft());

        return "positions";
    }

    @ResponseBody
    @GetMapping(value = "/acstream", produces =
        MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<Aircraft> getCurrentACPositionsStream() {
        return requester.route("acstream")
                .data("Requesting aircraft positions")
                .retrieveFlux(Aircraft.class);
    }
}
```

Reviewing the existing `PositionController` endpoints shows that they feed a Thymeleaf template (`public String getCurrentAircraftPositions(Model model)`) or require an external `RSocket` connection (`public Flux<Aircraft> getCurrentACPositionsStream()`). In order to isolate and test the Aircraft Positions application's ability to provide an external API, I need to expand the currently defined endpoints. I add two more endpoints mapped to */acpos* and */acpos/search* to create a basic, but flexible, API leveraging the methods I created within `PositionService`.

I first create a method to retrieve and return as JSON all positions of aircraft currently within range of our PlaneFinder service-enabled device. The `getCurrentACPositions()` method calls `PositionService::getAllAircraft` just as its counterpart `getCurrentAircraftPositions(Model model)`, but it returns JSON object values instead of adding them to the domain object model and redirecting to the template engine for display of an HTML page.

Next, I create a method for searching current aircraft positions by the unique position record identifier and by the aircraft registration number. The record (technically document, since this version of `Aircraft Positions` uses MongoDB) identifier is the database unique ID among the stored positions last retrieved from PlaneFinder. It is useful for retrieving a specific position record; but more useful from an aircraft per-

spective is the ability to search for an aircraft's unique registration number.

Interestingly, PlaneFinder may report a small number of positions reported by a single aircraft when queried. This is due to near-constant position reports being sent from aircraft in flight. What this means for us is that when searching by an aircraft's unique registration number within currently reported positions, we may actually retrieve 1+ position reports for that flight.

There are various ways to write a search mechanism with flexibility to accept different search criteria of different types returning different numbers of potential results, but I chose to incorporate all options within a single method:

```
@ResponseBody
@GetMapping("/acpos/search")
public Publisher<Aircraft>
        searchForACPosition(@RequestParam Map<String, String> searchParams) {

    if (!searchParams.isEmpty()) {
        Map.Entry<String, String> setToSearch =
                searchParams.entrySet().iterator().next();

        if (setToSearch.getKey().equalsIgnoreCase("id")) {
            return service.getAircraftById(Long.valueOf(setToSearch.getValue()
        } else {
            return service.getAircraftByReg(setToSearch.getValue());
        }
    } else {
        return Mono.empty();
    }
}
```

NOTES ON SEARCHFORACPOSITION'S DESIGN AND IMPLEMENTATION DECISIONS

First, `@ResponseBody` is necessary because I chose to combine REST endpoints with template-driving endpoints in the same `Controller` class. As mentioned previously, the `@RestController` meta-annotation includes both `@Controller` and `@ResponseBody` to indicate `Object` values are returned directly, versus via an HTML page's Domain Object Model (DOM). Since `PositionController` is annotated with only `@Controller`, it is necessary to add `@ResponseBody` to any methods I desire to return `Object` values directly.

Next, the `@RequestParam` annotation allows for the user to provide zero or more request parameters by appending a question mark (?) to an endpoint's mapping and specifying parameters in the format `key=value`, separated by a comma. In this example, I made the conscious choice to check only the first parameter (if any exist) for a key named "id"; if the request includes an `id` parameter, it is used to request the aircraft position document by its database ID. If the parameter is not `id`, I default to a search of aircraft registration numbers within the currently reported positions.

There are several tacit assumptions here that I might not make in a production system, including a default search for registrations, the intentional discard of any subsequent search parameters, and more. I leave these as a future exercise for myself and the reader.

One thing of note from the method signature: I return a `Publisher`, not specifically a `Flux` or `Mono`. This is necessary based on my decision to integrate search options into a single method and the fact that while searching for a position document within the database by the database ID will return no more than one match, searching by an aircraft's registration may yield multiple closely grouped position reports. Specifying a return value of `Publisher` for the method allows me to return either `Mono` or `Flux`, since both are specializations of `Publisher`.

Finally, if no search parameters are supplied by the user, I return an empty `Mono` using `Mono.empty()`. Your requirements may dictate the same outcome, or you may choose (or be required) to return a different result, such as all aircraft positions. Whatever the design decision, the Principle of Least Astonishment should inform the outcome.

The final (for now) version of the `PositionController` class should look something like this:

```
import org.reactivestreams.Publisher;
import org.springframework.http.MediaType;
import org.springframework.messaging.rsocket.RSocketRequester;
import org.springframework.stereotype.Controller;
```

```
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Map;

@Controller
public class PositionController {
    private final PositionService service;
    private final RSocketRequester requester;

    public PositionController(PositionService service,
            RSocketRequester requester) {
        this.service = service;
        this.requester = requester;
    }

    @GetMapping("/aircraft")
    public String getCurrentAircraftPositions(Model model) {
        model.addAttribute("currentPositions", service.getAllAircraft());

        return "positions";
    }

    @ResponseBody
    @GetMapping("/acpos")
    public Flux<Aircraft> getCurrentACPositions() {
        return service.getAllAircraft();
    }

    @ResponseBody
    @GetMapping("/acpos/search")
    public Publisher<Aircraft> searchForACPosition(@RequestParam Map<String,
            String> searchParams) {

        if (!searchParams.isEmpty()) {
            Map.Entry<String, String> setToSearch =
                searchParams.entrySet().iterator().next();

            if (setToSearch.getKey().equalsIgnoreCase("id")) {
                return service.getAircraftById(Long.valueOf
                    (setToSearch.getValue()));
            } else {
                return service.getAircraftByReg(setToSearch.getValue());
            }
        } else {
            return Mono.empty();
        }
    }

    @ResponseBody
    @GetMapping(value = "/acstream", produces =
```

```
                MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<Aircraft> getCurrentACPositionsStream() {
        return requester.route("acstream")
                .data("Requesting aircraft positions")
                .retrieveFlux(Aircraft.class);
    }
}
```

Next, I return to the `PositionService` class. As mentioned earlier, its `public Flux<Aircraft> getAircraftByReg(String reg)` method references a currently undefined method in `AircraftRepository`. To fix that, I add a `Flux<Aircraft> findAircraftByReg(String reg)` method to the `AircraftRepository` interface definition:

```
import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import reactor.core.publisher.Flux;

public interface AircraftRepository extends
        ReactiveCrudRepository<Aircraft, Long> {
    Flux<Aircraft> findAircraftByReg(String reg);
}
```

This interesting bit of code, this single method signature, demonstrates the powerful Spring Data concept of query derivation using a set of widely applicable conventions: operators like `find`, `search`, or `get`, the specified type of objects stored/retrieved/managed (in this case `Aircraft`), and member variable names like `reg`. By declaring a method signature with parameters+types and return type using the method naming conventions mentioned, Spring Data can build the method implementation for you.

If you want or need to provide more specifics or hints, it's also possible to annotate the method signature with `@Query` and supply desired or required details. That isn't necessary for this case, as stating we wish to search aircraft positions by registration number and return 0+ values in a reactive streams `Flux` is ample information for Spring Data to create the implementation.

Returning to `PositionService`, the IDE now happily reports `repo.findAircraftByReg(reg)` as a valid method call.

NOTE

Another design decision I made for this example was to have both `getAircraftByXxx` methods query the current position documents. This may be considered to assume some position documents exist in the database or that the user isn't concerned with a fresh retrieval if the database doesn't already contain any positions within. Your requirements may drive a different choice, such as verifying some positions are present prior to searching and if not executing a fresh retrieval with a call to `getAllAircraft`.

And Now, the Testing

In the earlier chapter on testing, standard `Object` types were used to test expected outcomes. I did use `WebClient` and `WebTestClient`, but only as the tool of choice for interacting with all HTTP-based endpoints, regardless of whether they returned reactive streams publisher types or not. Now, it's time to properly test those reactive streams semantics.

Using the existing `PositionControllerTest` class as a starting point, I retool it to accommodate the new reactive endpoints exposed by its counterpart class `PositionController`. Here are the class-level details:

```
@WebFluxTest(controllers = {PositionController.class})
class PositionControllerTest {
    @Autowired
    private WebTestClient client;

    @MockBean
    private PositionService service;
    @MockBean
    private RSocketRequester requester;

    private Aircraft ac1, ac2, ac3;

    ...

}
```

First, I make the class-level annotation `@WebFluxTest(controllers = {PositionController.class})`. I still use the reactive `WebTestClient` and wish to restrict the scope of this test class to WebFlux capabilities, so loading a full Spring Boot application context is unnecessary and wasteful of time and resources.

Second, I autowire a `WebTestClient` bean. In the earlier chapter on testing, I directly injected the `WebTestClient` bean into a single test method, but since it now will be needed in multiple methods, it makes more sense to create a member variable from which to reference it.

Third, I create mock beans using Mockito's `@MockBean` annotation. I mock the `RSocketRequester` bean simply because `PositionController`—which we request (and need) to be loaded in the class-level annotation—requires a bean, real or mocked, of an `RSocketRequester`. I mock the `PositionService` bean in order to mock and use its behavior within this class's tests. Mocking `PositionService` allows me to assure its proper behavior, exercise a consumer of its outputs (`PositionController`), and compare the actual results with known expected results.

Finally, I create three `Aircraft` instances for use in the contained tests.

Prior to executing a JUnit `@Test` method, a method annotated with `@BeforeEach` is run to configure the scenario and expected results. This is the `setUp()` method I use to prepare the testing environment before each test method:

```
@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());

    // Spring Airlines flight 002 en route, flying SFO to STL,
    // at 40000' currently just past DEN
    ac3 = new Aircraft(3L, "SAL002", "sqwk", "N54321", "SAL002",
            "SFO-STL", "LJ", "ct",
            40000, 65, 440, 0, 0,
            39.8412964, -105.0048267, 0D, 0D, 0D,
            true, false,
            Instant.now(), Instant.now(), Instant.now());


    Mockito.when(service.getAllAircraft()).thenReturn(Flux.just(ac1, ac2, ac3));
    Mockito.when(service.getAircraftById(1L)).thenReturn(Mono.just(ac1));
    Mockito.when(service.getAircraftById(2L)).thenReturn(Mono.just(ac2));
    Mockito.when(service.getAircraftById(3L)).thenReturn(Mono.just(ac3));
    Mockito.when(service.getAircraftByReg("N12345"))
```

```
            .thenReturn(Flux.just(ac1));
        Mockito.when(service.getAircraftByReg("N54321"))
            .thenReturn(Flux.just(ac2, ac3));
    }
```

I assign an aircraft position for the aircraft with registration N12345 to the `ac1` member variable. For `ac2` and `ac3`, I assign positions very close to each other for the same aircraft, N54321, simulating a frequent case of closely updated position reports arriving from PlaneFinder.

The last several lines of the `setUp()` method define the behavior that the `PositionService` mock bean will provide when its methods are called in various ways. Similar to the method mocks in the earlier chapter on testing, the only difference of import is the types of return values; since the actual `PositionService` methods return Reactor `Publisher` types of `Flux` and `Mono`, so must the mock methods.

Test for retrieving all aircraft positions

Finally, I create a method to test the `PositionController` method `getCurrentACPositions()`:

```
    @Test
    void getCurrentACPositions() {
        StepVerifier.create(client.get()
                .uri("/acpos")
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .returnResult(Aircraft.class)
                .getResponseBody())
            .expectNext(ac1)
            .expectNext(ac2)
            .expectNext(ac3)
            .verifyComplete();
    }
```

Testing reactive streams applications can bring myriad challenges to what is often considered a pretty mundane (if prone to omission) effort of setting expected results, obtaining actual results, and comparing the two to determine test success or failure. Though multiple results *can* be obtained in an effectively instantaneous manner, just as with a blocking type of `Iterable`, reactive streams `Publishers` don't wait for a complete result set prior to returning it as a single unit. From a machine perspective, it's the difference between receiving one group of five all at once (for example) or receiving five results very quickly, but individually.

The core of Reactor's testing tools is the `StepVerifier` and its utility methods. `StepVerifier` subscribes to a `Publisher` and, as the name implies, enables the developer to consider results obtained as discrete

values and verify each one. In the test for `getCurrentACPositions`, I perform the following actions:

- Create a `StepVerifier`.
- Supply it a `Flux` produced by the following steps:
  - Use the `WebTestClient` bean.
  - Access the `PositionController::getCurrentACPositions` method mapped to the */acpos* endpoint.
  - Initiate the `exchange()`.
  - Verify a response status of `200 OK`.
  - Verify the response header has a content type of "application/json".
  - Return the result items as instances of the `Aircraft` class.
  - GET the response.
- Evaluate the actual first value against the expected first value `ac1`.
- Evaluate the actual second value against the expected second value `ac2`.
- Evaluate the actual third value against the expected third value `ac3`.
- Verify all actions and receipt of `Publisher` completion signal.

This is quite an exhaustive evaluation of expected behavior, including conditions and values returned. Running the test results in output similar to the following (trimmed to fit page):

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.0)

: Starting PositionControllerTest on mheckler-a01.vmware.com with PID 21211
: No active profile set, falling back to default profiles: default
: Started PositionControllerTest in 2.19 seconds (JVM running for 2.879)

Process finished with exit code 0
```

Run from the IDE, the result will look similar to that shown in Figure 12-1.

Figure 12-1. Successful test

Testing Aircraft Positions search capabilities

Testing the search functionality within `PositionController::searchForACPosition` requires a minimum of two separate tests due to the ability to handle searches for aircraft positions by database document ID and aircraft registration numbers.

To test searching by database document identifier, I create the following unit test:

```
@Test
void searchForACPositionById() {
    StepVerifier.create(client.get()
            .uri("/acpos/search?id=1")
            .exchange()
            .expectStatus().isOk()
            .expectHeader().contentType(MediaType.APPLICATION_JSON)
            .returnResult(Aircraft.class)
            .getResponseBody())
        .expectNext(ac1)
        .verifyComplete();
}
```

This is similar to the unit test for all aircraft positions. There are two notable exceptions:

- The specified URI references the search endpoint and includes the search parameter `id=1` to retrieve `ac1`.
- The expected result is only `ac1`, as indicated in the `expectNext(ac1)` chained operation.

To test searching for aircraft positions by an aircraft registration number, I create the following unit test, using a registration that I've mocked to include two corresponding position documents:

```
@Test
void searchForACPositionByReg() {
    StepVerifier.create(client.get()
            .uri("/acpos/search?reg=N54321")
            .exchange()
            .expectStatus().isOk()
            .expectHeader().contentType(MediaType.APPLICATION_JSON)
            .returnResult(Aircraft.class)
            .getResponseBody())
        .expectNext(ac2)
        .expectNext(ac3)
        .verifyComplete();
}
```

The differences between this test and the previous one are minimal:

- The URI includes the search parameter `reg=N54321` and should result in the return of both `ac2` and `ac3`, both of which contain reported positions for the aircraft with registration number N54321.
- Expected results are verified to be `ac2` and `ac3` with the `expectNext(ac2)` and `expectNext(ac3)` chained operations.

The final state of the `PositionControllerTest` class is shown in the following listing:

```
import org.junit.jupiter.api.AfterEach;
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.WebFluxTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.messaging.rsocket.RSocketRequester;
import org.springframework.test.web.reactive.server.WebTestClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import reactor.test.StepVerifier;

import java.time.Instant;

@WebFluxTest(controllers = {PositionController.class})
class PositionControllerTest {
    @Autowired
    private WebTestClient client;

    @MockBean
    private PositionService service;
    @MockBean
    private RSocketRequester requester;

    private Aircraft ac1, ac2, ac3;

    @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());

        // Spring Airlines flight 002 en route, flying SFO to STL, at 40000'
        // currently just past DEN
        ac3 = new Aircraft(3L, "SAL002", "sqwk", "N54321", "SAL002",
                "SFO-STL", "LJ", "ct",
                40000, 65, 440, 0, 0,
                39.8412964, -105.0048267, 0D, 0D, 0D,
                true, false,
                Instant.now(), Instant.now(), Instant.now());


        Mockito.when(service.getAllAircraft())
                .thenReturn(Flux.just(ac1, ac2, ac3));
        Mockito.when(service.getAircraftById(1L))
                .thenReturn(Mono.just(ac1));
        Mockito.when(service.getAircraftById(2L))
                .thenReturn(Mono.just(ac2));
        Mockito.when(service.getAircraftById(3L))
                .thenReturn(Mono.just(ac3));
        Mockito.when(service.getAircraftByReg("N12345"))
                .thenReturn(Flux.just(ac1));
        Mockito.when(service.getAircraftByReg("N54321"))
                .thenReturn(Flux.just(ac2, ac3));
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void getCurrentACPositions() {
        StepVerifier.create(client.get()
                .uri("/acpos")
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .returnResult(Aircraft.class)
                .getResponseBody())
            .expectNext(ac1)
            .expectNext(ac2)
            .expectNext(ac3)
            .verifyComplete();
    }

    @Test
    void searchForACPositionById() {
        StepVerifier.create(client.get()
                .uri("/acpos/search?id=1")
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .returnResult(Aircraft.class)
                .getResponseBody())
```

```
                .expectNext(ac1)
                .verifyComplete();
    }

    @Test
    void searchForACPositionByReg() {
        StepVerifier.create(client.get()
                .uri("/acpos/search?reg=N54321")
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .returnResult(Aircraft.class)
                .getResponseBody())
                .expectNext(ac2)
                .expectNext(ac3)
                .verifyComplete();
    }
}
```

Executing all tests within the `PositionControllerTest` class provides the gratifying results shown in Figure 12-2.

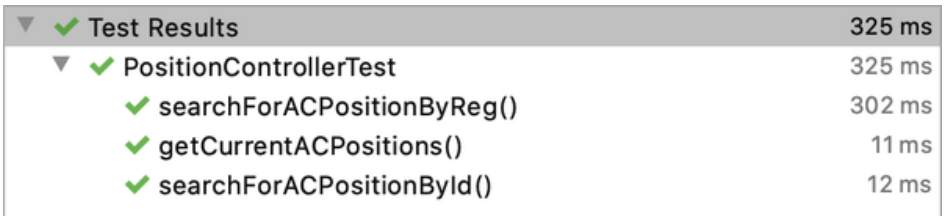

Figure 12-2. Successful execution of all unit tests

NOTE

`StepVerifier` enables more testing possibilities, a few of which have been hinted at in this section. Of particular interest is the `StepVerifier::withVirtualTime` method that enables tests of publishers that emit values sporadically to be compressed, producing results instantaneously that might ordinarily be spaced over extensive periods of time. `StepVerifier::withVirtualTime` accepts a `Supplier<Publisher>` instead of a `Publisher` directly, but otherwise the mechanics of its use are quite similar.

These are essential elements of testing reactive Spring Boot applications. But what happens when you encounter issues in production? What tools does Reactor offer for identification and resolution of issues when your app goes live?

Diagnosing and Debugging Reactive Applications

When things go sideways in typical Java applications, there is usually a stacktrace. A useful (if sometimes voluminous) stacktrace can be produced by imperative code for several reasons, but at a high level, two factors enable this helpful information to be collected and shown:

- Sequential execution of code that typically dictates how to do something (imperative)
- Execution of that sequential code occurs within a single thread

There are exceptions to every rule, but generally speaking, this is the common combination that allows for the capture of steps executed sequentially up to the time an error was encountered: everything happens one step at a time in a single swimlane. It may not leverage full system resources as effectively, and it generally doesn't, but it makes isolating and resolving issues a much simpler affair.

Enter reactive streams. Project Reactor and other reactive streams implementations use schedulers to manage and use those other threads. Resources that would typically have remained idle or underutilized can be put to work to enable reactive applications to scale far beyond their blocking counterparts. I would refer you to the Reactor Core documentation for more details regarding `Schedulers` and the options available for controlling how they can be used and tuned, but suffice it to say for now that Reactor does a fine job handling scheduling automatically in the vast majority of circumstances.

This does highlight one challenge with producing a meaningful execution trace for a reactive Spring Boot (or any reactive) application, however. One can't expect to simply follow a single thread's activity and produce a meaningful sequential list of code executed.

Compounding the difficulty of tracing execution due to this thread-hopping optimizing feature is that reactive programming separates code *assembly* from code *execution*. As mentioned in Chapter 8, in most cases for most `Publisher` types, nothing happens until you *subscribe*.

Simply put, it's unlikely that you will ever see a production failure that points to an issue with the code where you declaratively assembled the `Publisher` (whether `Flux` or `Mono`) pipeline of operations. Failures nearly universally occur at the point the pipeline becomes active: producing, processing, and passing values to a `Subscriber`.

This distancing between code assembly and execution and Reactor's ability to utilize multiple threads to complete a chain of operations necessi-

tates better tooling to effectively troubleshoot errors that surface at runtime. Fortunately, Reactor provides several excellent options.

Hooks.onOperatorDebug()

This is not to imply that troubleshooting reactive applications using existing stacktrace results is impossible, only that it could be significantly improved upon. As with most things, the proof is in the code—or in this case, the logged, post failure output.

To simulate a failure in a reactive `Publisher` chain of operators, I revisit the `PositionControllerTest` class and change one line of code in the `setUp()` method run before each test's execution:

```
Mockito.when(service.getAllAircraft()).thenReturn(Flux.just(ac1, ac2, ac3));
```

I replace the properly operating `Flux` produced by the mock `getAllAircraft()` method with one that includes an error in the resultant stream of values:

```
Mockito.when(service.getAllAircraft()).thenReturn(
        Flux.just(ac1, ac2, ac3)
                .concatWith(Flux.error(new Throwable("Bad position report")))
);
```

Next, I execute the test for `getCurrentACPositions()` to see the results of our intentional `Flux` sabotage (wrapped to fit page):

```
500 Server Error for HTTP GET "/acpos"

java.lang.Throwable: Bad position report
        at com.thehecklers.aircraftpositions.PositionControllerTest
        .setUp(PositionControllerTest.java:59) ~[test-classes/:na]
        Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Error has been observed at the following site(s):
        |_ checkpoint ⇢ Handler com.thehecklers.aircraftpositions
        .PositionController
        #getCurrentACPositions() [DispatcherHandler]
        |_ checkpoint ⇢ HTTP GET "/acpos" [ExceptionHandlingWebHandler]
Stack trace:
                at com.thehecklers.aircraftpositions.PositionControllerTest
        .setUp(PositionControllerTest.java:59) ~[test-classes/:na]
                at java.base/jdk.internal.reflect.NativeMethodAccessorImpl
        .invoke0(Native Method) ~[na:na]
                at java.base/jdk.internal.reflect.NativeMethodAccessorImpl
        .invoke(NativeMethodAccessorImpl.java:62) ~[na:na]
                at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl
        .invoke(DelegatingMethodAccessorImpl.java:43) ~[na:na]
                at java.base/java.lang.reflect.Method
        .invoke(Method.java:564) ~[na:na]
```

```
        at org.junit.platform.commons.util.ReflectionUtils
.invokeMethod(ReflectionUtils.java:686)
~[junit-platform-commons-1.6.2.jar:1.6.2]
        at org.junit.jupiter.engine.execution.MethodInvocation
.proceed(MethodInvocation.java:60)
        ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution.InvocationInterceptorChai
$ValidatingInvocation.proceed(InvocationInterceptorChain.java:131)
~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.extension.TimeoutExtension
.intercept(TimeoutExtension.java:149)
~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.extension.TimeoutExtension
.interceptLifecycleMethod(TimeoutExtension.java:126)
~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.extension.TimeoutExtension
.interceptBeforeEachMethod(TimeoutExtension.java:76)
~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution
.ExecutableInvoker$ReflectiveInterceptorCall.lambda$ofVoidMethod
  $0(ExecutableInvoker.java:115)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution.ExecutableInvoker
.lambda$invoke$0(ExecutableInvoker.java:105)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution.InvocationInterceptorChai
$InterceptedInvocation.proceed(InvocationInterceptorChain.java:106)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution.InvocationInterceptorChai
.proceed(InvocationInterceptorChain.java:64)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution.InvocationInterceptorChai
.chainAndInvoke(InvocationInterceptorChain.java:45)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution.InvocationInterceptorChai
.invoke(InvocationInterceptorChain.java:37)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution.ExecutableInvoker
.invoke(ExecutableInvoker.java:104)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.execution.ExecutableInvoker
.invoke(ExecutableInvoker.java:98)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.descriptor.ClassBasedTestDescriptor
.invokeMethodInExtensionContext(ClassBasedTestDescriptor.java:481)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.descriptor.ClassBasedTestDescriptor
.lambda$synthesizeBeforeEachMethodAdapter
  $18(ClassBasedTestDescriptor.java:466)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
.lambda$invokeBeforeEachMethods$2(TestMethodTestDescriptor.java:169)
  ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
        at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
.lambda$invokeBeforeMethodsOrCallbacksUntilExceptionOccurs
```

```
        $5(TestMethodTestDescriptor.java:197)
        ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
            at org.junit.platform.engine.support.hierarchical.ThrowableColl
    .execute(ThrowableCollector.java:73)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
    .invokeBeforeMethodsOrCallbacksUntilExceptionOccurs
        (TestMethodTestDescriptor.java:197)
        ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
            at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
    .invokeBeforeEachMethods(TestMethodTestDescriptor.java:166)
        ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
            at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
    .execute(TestMethodTestDescriptor.java:133)
        ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
            at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
    .execute(TestMethodTestDescriptor.java:71)
        ~[junit-jupiter-engine-5.6.2.jar:5.6.2]
            at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$5(NodeTestTask.java:135)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.ThrowableColl
    .execute(ThrowableCollector.java:73)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$7(NodeTestTask.java:125)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.Node
    .around(Node.java:135) ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$8(NodeTestTask.java:123)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.ThrowableColl
    .execute(ThrowableCollector.java:73)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .executeRecursively(NodeTestTask.java:122)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .execute(NodeTestTask.java:80)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at java.base/java.util.ArrayList.forEach(ArrayList.java:1510) ~
            at org.junit.platform.engine.support.hierarchical
    .SameThreadHierarchicalTestExecutorService
        .invokeAll(SameThreadHierarchicalTestExecutorService.java:38)
            ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$5(NodeTestTask.java:139)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.ThrowableColl
    .execute(ThrowableCollector.java:73)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
            at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$7(NodeTestTask.java:125)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
```

```
        at org.junit.platform.engine.support.hierarchical.Node
.around(Node.java:135) ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
.lambda$executeRecursively$8(NodeTestTask.java:123)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.ThrowableColl
.execute(ThrowableCollector.java:73)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
.executeRecursively(NodeTestTask.java:122)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
.execute(NodeTestTask.java:80)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at java.base/java.util.ArrayList.forEach(ArrayList.java:1510) ~
        at org.junit.platform.engine.support.hierarchical
.SameThreadHierarchicalTestExecutorService
    .invokeAll(SameThreadHierarchicalTestExecutorService.java:38)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
.lambda$executeRecursively$5(NodeTestTask.java:139)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.ThrowableColl
.execute(ThrowableCollector.java:73)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
.lambda$executeRecursively$7(NodeTestTask.java:125)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.Node
.around(Node.java:135) ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
.lambda$executeRecursively$8(NodeTestTask.java:123)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.ThrowableColl
.execute(ThrowableCollector.java:73)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
.executeRecursively(NodeTestTask.java:122)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
.execute(NodeTestTask.java:80)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical
.SameThreadHierarchicalTestExecutorService
    .submit(SameThreadHierarchicalTestExecutorService.java:32)
        ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical
.HierarchicalTestExecutor.execute(HierarchicalTestExecutor.java:57)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.engine.support.hierarchical
.HierarchicalTestEngine.execute(HierarchicalTestEngine.java:51)
    ~[junit-platform-engine-1.6.2.jar:1.6.2]
        at org.junit.platform.launcher.core.DefaultLauncher
.execute(DefaultLauncher.java:248)
    ~[junit-platform-launcher-1.6.2.jar:1.6.2]
```

```
                at org.junit.platform.launcher.core.DefaultLauncher
        .lambda$execute$5(DefaultLauncher.java:211)
            ~[junit-platform-launcher-1.6.2.jar:1.6.2]
                at org.junit.platform.launcher.core.DefaultLauncher
        .withInterceptedStreams(DefaultLauncher.java:226)
            ~[junit-platform-launcher-1.6.2.jar:1.6.2]
                at org.junit.platform.launcher.core.DefaultLauncher
        .execute(DefaultLauncher.java:199)
            ~[junit-platform-launcher-1.6.2.jar:1.6.2]
                at org.junit.platform.launcher.core.DefaultLauncher
        .execute(DefaultLauncher.java:132)
            ~[junit-platform-launcher-1.6.2.jar:1.6.2]
                at com.intellij.junit5.JUnit5IdeaTestRunner
        .startRunnerWithArgs(JUnit5IdeaTestRunner.java:69)
            ~[junit5-rt.jar:na]
                at com.intellij.rt.junit.IdeaTestRunner$Repeater
        .startRunnerWithArgs(IdeaTestRunner.java:33)
            ~[junit-rt.jar:na]
                at com.intellij.rt.junit.JUnitStarter
        .prepareStreamsAndStart(JUnitStarter.java:230)
            ~[junit-rt.jar:na]
                at com.intellij.rt.junit.JUnitStarter
        .main(JUnitStarter.java:58) ~[junit-rt.jar:na]


java.lang.AssertionError: Status expected:<200 OK>
    but was:<500 INTERNAL_SERVER_ERROR>

> GET /acpos
> WebTestClient-Request-Id: [1]

No content

< 500 INTERNAL_SERVER_ERROR Internal Server Error
< Content-Type: [application/json]
< Content-Length: [142]

{"timestamp":"2020-11-09T15:41:12.516+00:00","path":"/acpos","status":500,
        "error":"Internal Server Error","message":"","requestId":"699a523c"}

        at org.springframework.test.web.reactive.server.ExchangeResult
    .assertWithDiagnostics(ExchangeResult.java:209)
        at org.springframework.test.web.reactive.server.StatusAssertions
    .assertStatusAndReturn(StatusAssertions.java:227)
        at org.springframework.test.web.reactive.server.StatusAssertions
    .isOk(StatusAssertions.java:67)
        at com.thehecklers.aircraftpositions.PositionControllerTest
    .getCurrentACPositions(PositionControllerTest.java:90)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl
    .invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl
    .invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl
    .invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
```

```
        at org.junit.platform.commons.util.ReflectionUtils
    .invokeMethod(ReflectionUtils.java:686)
        at org.junit.jupiter.engine.execution.MethodInvocation
    .proceed(MethodInvocation.java:60)
        at org.junit.jupiter.engine.execution.InvocationInterceptorChain
    $ValidatingInvocation.proceed(InvocationInterceptorChain.java:131)
        at org.junit.jupiter.engine.extension.TimeoutExtension
    .intercept(TimeoutExtension.java:149)
        at org.junit.jupiter.engine.extension.TimeoutExtension
    .interceptTestableMethod(TimeoutExtension.java:140)
        at org.junit.jupiter.engine.extension.TimeoutExtension
    .interceptTestMethod(TimeoutExtension.java:84)
        at org.junit.jupiter.engine.execution.ExecutableInvoker
    $ReflectiveInterceptorCall
        .lambda$ofVoidMethod$0(ExecutableInvoker.java:115)
        at org.junit.jupiter.engine.execution.ExecutableInvoker
    .lambda$invoke$0(ExecutableInvoker.java:105)
        at org.junit.jupiter.engine.execution.InvocationInterceptorChain
    $InterceptedInvocation.proceed(InvocationInterceptorChain.java:106)
        at org.junit.jupiter.engine.execution.InvocationInterceptorChain
    .proceed(InvocationInterceptorChain.java:64)
        at org.junit.jupiter.engine.execution.InvocationInterceptorChain
    .chainAndInvoke(InvocationInterceptorChain.java:45)
        at org.junit.jupiter.engine.execution.InvocationInterceptorChain
    .invoke(InvocationInterceptorChain.java:37)
        at org.junit.jupiter.engine.execution.ExecutableInvoker
    .invoke(ExecutableInvoker.java:104)
        at org.junit.jupiter.engine.execution.ExecutableInvoker
    .invoke(ExecutableInvoker.java:98)
        at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
    .lambda$invokeTestMethod$6(TestMethodTestDescriptor.java:212)
        at org.junit.platform.engine.support.hierarchical.ThrowableCollector
    .execute(ThrowableCollector.java:73)
        at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
    .invokeTestMethod(TestMethodTestDescriptor.java:208)
        at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
    .execute(TestMethodTestDescriptor.java:137)
        at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor
    .execute(TestMethodTestDescriptor.java:71)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$5(NodeTestTask.java:135)
        at org.junit.platform.engine.support.hierarchical.ThrowableCollector
    .execute(ThrowableCollector.java:73)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$7(NodeTestTask.java:125)
        at org.junit.platform.engine.support.hierarchical.Node.around(Node.java
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$8(NodeTestTask.java:123)
        at org.junit.platform.engine.support.hierarchical.ThrowableCollector
    .execute(ThrowableCollector.java:73)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .executeRecursively(NodeTestTask.java:122)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .execute(NodeTestTask.java:80)
        at java.base/java.util.ArrayList.forEach(ArrayList.java:1510)
```

```
        at org.junit.platform.engine.support.hierarchical
    .SameThreadHierarchicalTestExecutorService
        .invokeAll(SameThreadHierarchicalTestExecutorService.java:38)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$5(NodeTestTask.java:139)
        at org.junit.platform.engine.support.hierarchical.ThrowableCollector
    .execute(ThrowableCollector.java:73)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$7(NodeTestTask.java:125)
        at org.junit.platform.engine.support.hierarchical.Node.around(Node.java
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$8(NodeTestTask.java:123)
        at org.junit.platform.engine.support.hierarchical.ThrowableCollector
    .execute(ThrowableCollector.java:73)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .executeRecursively(NodeTestTask.java:122)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .execute(NodeTestTask.java:80)
        at java.base/java.util.ArrayList.forEach(ArrayList.java:1510)
        at org.junit.platform.engine.support.hierarchical
    .SameThreadHierarchicalTestExecutorService
        .invokeAll(SameThreadHierarchicalTestExecutorService.java:38)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$5(NodeTestTask.java:139)
        at org.junit.platform.engine.support.hierarchical.ThrowableCollector
    .execute(ThrowableCollector.java:73)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$7(NodeTestTask.java:125)
        at org.junit.platform.engine.support.hierarchical.Node.around(Node.java
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .lambda$executeRecursively$8(NodeTestTask.java:123)
        at org.junit.platform.engine.support.hierarchical.ThrowableCollector
    .execute(ThrowableCollector.java:73)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .executeRecursively(NodeTestTask.java:122)
        at org.junit.platform.engine.support.hierarchical.NodeTestTask
    .execute(NodeTestTask.java:80)
        at org.junit.platform.engine.support.hierarchical
    .SameThreadHierarchicalTestExecutorService
        .submit(SameThreadHierarchicalTestExecutorService.java:32)
        at org.junit.platform.engine.support.hierarchical.HierarchicalTestExecu
    .execute(HierarchicalTestExecutor.java:57)
        at org.junit.platform.engine.support.hierarchical.HierarchicalTestEngi
    .execute(HierarchicalTestEngine.java:51)
        at org.junit.platform.launcher.core.DefaultLauncher
    .execute(DefaultLauncher.java:248)
        at org.junit.platform.launcher.core.DefaultLauncher
    .lambda$execute$5(DefaultLauncher.java:211)
        at org.junit.platform.launcher.core.DefaultLauncher
    .withInterceptedStreams(DefaultLauncher.java:226)
        at org.junit.platform.launcher.core.DefaultLauncher
    .execute(DefaultLauncher.java:199)
        at org.junit.platform.launcher.core.DefaultLauncher
    .execute(DefaultLauncher.java:132)
        at com.intellij.junit5.JUnit5IdeaTestRunner
```

```
        .startRunnerWithArgs(JUnit5IdeaTestRunner.java:69)
            at com.intellij.rt.junit.IdeaTestRunner$Repeater
        .startRunnerWithArgs(IdeaTestRunner.java:33)
            at com.intellij.rt.junit.JUnitStarter
        .prepareStreamsAndStart(JUnitStarter.java:230)
            at com.intellij.rt.junit.JUnitStarter
        .main(JUnitStarter.java:58)
    Caused by: java.lang.AssertionError: Status expected:<200 OK>
            but was:<500 INTERNAL_SERVER_ERROR>
            at org.springframework.test.util.AssertionErrors
        .fail(AssertionErrors.java:59)
            at org.springframework.test.util.AssertionErrors
        .assertEquals(AssertionErrors.java:122)
            at org.springframework.test.web.reactive.server.StatusAssertions
        .lambda$assertStatusAndReturn$4(StatusAssertions.java:227)
            at org.springframework.test.web.reactive.server.ExchangeResult
        .assertWithDiagnostics(ExchangeResult.java:206)
            ... 66 more
```

As you can see, the volume of information for a single bad value is quite difficult to digest. Useful information is present, but it's overwhelmed by excessive, less-helpful data.

NOTE

I reluctantly but deliberately included the full output resulting from the preceding `Flux` error to show how difficult it can be to navigate the usual output when a `Publisher` encounters an error and to contrast it with how dramatically available tools reduce the noise and boost the signal of key information. Getting to the core of the problem reduces frustration in development, but it is absolutely critical when troubleshooting business-critical applications in production.

Project Reactor includes configurable life cycle callbacks called *hooks*, available via its `Hooks` class. One operator that is particularly useful for increasing the signal to noise ratio when things go awry is `onOperatorDebug()`.

Calling `Hooks.onOperatorDebug()` prior to instantiation of the failing `Publisher` enables assembly-time instrumentation of all subsequent instances of type `Publisher` (and subtypes). In order to ensure capture of the necessary information at the necessary time(s), the call is usually placed in the application's main method as follows:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import reactor.core.publisher.Hooks;

@SpringBootApplication
public class AircraftPositionsApplication {
```

```
        public static void main(String[] args) {
                Hooks.onOperatorDebug();
                SpringApplication.run(AircraftPositionsApplication.class, args
        }

}
```

Since I am demonstrating this capability from a test class, I instead insert `Hooks.onOperatorDebug();` on the line immediately preceding assembly of the intentionally failing `Publisher`:

```
Hooks.onOperatorDebug();
Mockito.when(service.getAllAircraft()).thenReturn(
        Flux.just(ac1, ac2, ac3)
                .concatWith(Flux.error(new Throwable("Bad position report")))
);
```

This single addition doesn't eliminate the somewhat voluminous stacktrace—there are still rare occasions in which any additional bit of data provided can be helpful—but for the vast majority of cases, the tree summary added to the log by `onOperatorDebug()` as a backtrace results in faster issue identification and resolution. The backtrace summary for the same error I introduced in the `getCurrentACPositions()` test is shown in Figure 12-3 in order to preserve full details and formatting.

```
java.lang.Throwable: Bad position report
    at com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:67) ~[test-classes/:na]
    Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Assembly trace from producer [reactor.core.publisher.FluxError] :
    reactor.core.publisher.Flux.error(Flux.java:871)
    com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:68)
Error has been observed at the following site(s):
    |_        Flux.error ⇢ at com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:68)
    |_    Flux.concatWith ⇢ at com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:68)
    |_                    ⇢ at com.thehecklers.aircraftpositions.PositionService$MockitoMock$883678645.getAllAircraft(null:-1)
    |_         Flux.from ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:178)
    |_   Flux.collectList ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:179)
    |_   Flux.collectList ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:179)
    |_          Mono.map ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:188)
    |_          Mono.map ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:188)
    |_         Mono.flux ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:181)
    |_         Mono.flux ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:181)
    |_        Flux.from ⇢ at org.springframework.http.server.reactive.ChannelSendOperator.<init>(ChannelSendOperator.java:57)
    |_    Mono.doOnError ⇢ at org.springframework.http.server.reactive.AbstractServerHttpResponse.writeWith(AbstractServerHttpResponse.java:221)
    |_                   ⇢ at org.springframework.http.codec.EncoderHttpMessageWriter.write(EncoderHttpMessageWriter.java:203)
    |_                   ⇢ at org.springframework.web.reactive.result.method.annotation.AbstractMessageWriterResultHandler.writeBody
(AbstractMessageWriterResultHandler.java:104)
    |_                   ⇢ at org.springframework.web.reactive.result.method.annotation.ResponseBodyResultHandler.handleResult(ResponseBodyResultHandler.java:86)
    |_        checkpoint ⇢ Handler com.thehecklers.aircraftpositions.PositionController#getCurrentACPositions() [DispatcherHandler]
    |_      Mono.flatMap ⇢ at org.springframework.web.reactive.DispatcherHandler.lambda$handleResult$5(DispatcherHandler.java:172)
    |_ Mono.onErrorResume ⇢ at org.springframework.web.reactive.DispatcherHandler.handleResult(DispatcherHandler.java:171)
    |_      Mono.flatMap ⇢ at org.springframework.web.reactive.DispatcherHandler.handle(DispatcherHandler.java:147)
    |_                   ⇢ at org.springframework.web.server.handler.DefaultWebFilterChain.lambda$filter$0(DefaultWebFilterChain.java:120)
    |_         Mono.defer ⇢ at org.springframework.web.server.handler.DefaultWebFilterChain.filter(DefaultWebFilterChain.java:119)
    |_         Mono.defer ⇢ at org.springframework.web.server.handler.DefaultWebFilterChain.filter(DefaultWebFilterChain.java:119)
    |_                   ⇢ at org.springframework.web.server.handler.FilteringWebHandler.handle(FilteringWebHandler.java:59)
    |_                   ⇢ at org.springframework.web.server.handler.WebHandlerDecorator.handle(WebHandlerDecorator.java:56)
    |_         Mono.error ⇢ at org.springframework.web.server.handler.ExceptionHandlingWebHandler$CheckpointInsertingHandler.handle(ExceptionHandlingWebHandler
.java:98)
    |_         Mono.error ⇢ at org.springframework.web.server.handler.ExceptionHandlingWebHandler$CheckpointInsertingHandler.handle(ExceptionHandlingWebHandler
.java:98)
    |_        checkpoint ⇢ HTTP GET "/acpos" [ExceptionHandlingWebHandler]
    |_                   ⇢ at org.springframework.web.server.handler.ExceptionHandlingWebHandler.lambda$handle$0(ExceptionHandlingWebHandler.java:77)
    |_ Mono.onErrorResume ⇢ at org.springframework.web.server.handler.ExceptionHandlingWebHandler.handle(ExceptionHandlingWebHandler.java:77)
    |_ Mono.onErrorResume ⇢ at org.springframework.web.server.handler.ExceptionHandlingWebHandler.handle(ExceptionHandlingWebHandler.java:77)
```

Figure 12-3. Debugging backtrace

At the top of the tree is the incriminating evidence: a `Flux` error has been introduced using `concatWith` on line 68 of `PositionControllerTest.java`. Thanks to `Hooks.onOperatorDebug()`, the time it took to identify this issue and its specific location has been reduced from several minutes (or more) to a few seconds.

Instrumenting all assembly instructions for all subsequent `Publisher` occurrences doesn't come without a cost, however; using hooks to instrument your code is relatively runtime-expensive, as debug mode is global and impacts every chained operator of every reactive streams `Publisher` executed once enabled. Let's consider another alternative.

Checkpoints

Rather than populate every possible backtrace of every possible `Publisher`, one can set checkpoints near key operators to assist with troubleshooting. Inserting a `checkpoint()` operator into the chain works like enabling a hook but only for that segment of that chain of operators.

There are three variants of checkpoints:

- Standard checkpoints that include a backtrace
- Light checkpoints that accept a descriptive `String` parameter and do not include backtrace
- Standard checkpoints with backtrace that also accept a descriptive `String` parameter

Let's see them in action.

First, I remove the `Hooks.onOperatorDebug()` statement before the mocked method for `PositionService::getAllAircraft` in the `setUp() method within `PositionControllerTest`:

```
//Hooks.onOperatorDebug();      Comment out or remove
Mockito.when(service.getAllAircraft()).thenReturn(
    Flux.just(ac1, ac2, ac3)
        .checkpoint()
        .concatWith(Flux.error(new Throwable("Bad position report")))
        .checkpoint()
);
```

Rerunning the test for `getCurrentACPositions()` produces the results shown in Figure 12-4.

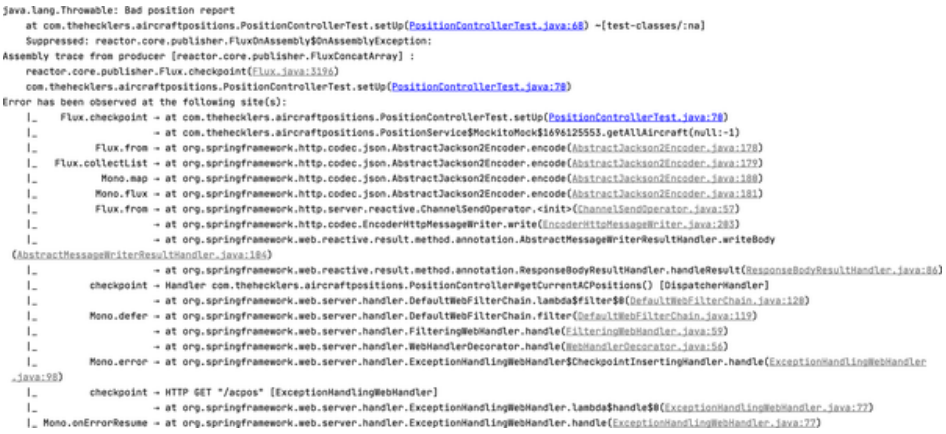

Figure 12-4. Standard checkpoint output

The checkpoint at the top of the list directs us to the problematic operator: the one immediately preceding the triggered checkpoint. Note that backtrace information is still being collected, as the checkpoint reflects the actual source code file and specific line number for the checkpoint I inserted on line 64 of the `PositionControllerTest` class.

Switching to lightweight checkpoints replaces the collection of backtrace information with a useful `String` description specified by the developer. While backtrace collection for standard checkpoints is limited in scope, it still requires resources beyond the simple storage of a `String`. If done with sufficient detail, light checkpoints provide the same utility in locating problematic operators. Updating the code to leverage light checkpoints is a simple matter:

```
//Hooks.onOperatorDebug();      Comment out or remove
Mockito.when(service.getAllAircraft()).thenReturn(
    Flux.just(ac1, ac2, ac3)
        .checkpoint("All Aircraft: after all good positions reported")
        .concatWith(Flux.error(new Throwable("Bad position report")))
        .checkpoint("All Aircraft: after appending bad position report")
);
```

Re-running the `getCurrentACPositions()` test produces the results shown in Figure 12-5.

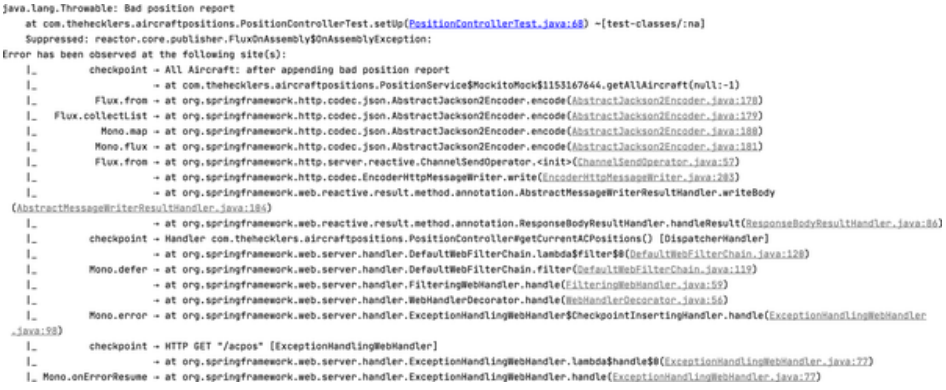

Figure 12-5. Light checkpoint output

Although file and line number coordinates are no longer present in the top-listed checkpoint, its clear description makes it easy to find the problem operator in the `Flux` assembly.

Occasionally there will be a requirement to employ an extremely complex chain of operators to build a `Publisher`. In those circumstances, it may be useful to include both a description and full backtrace information for troubleshooting. To demonstrate a very limited example, I refactor the mock method used for `PositionService::getAllAircraft` once more as follows:

```
//Hooks.onOperatorDebug();      Comment out or remove
Mockito.when(service.getAllAircraft()).thenReturn(
    Flux.just(ac1, ac2, ac3)
        .checkpoint("All Aircraft: after all good positions reported", true)
        .concatWith(Flux.error(new Throwable("Bad position report")))
        .checkpoint("All Aircraft: after appending bad position report", true)
);
```

Running the `getCurrentACPositions()` test once again results in the output shown in Figure 12-6.

```
java.lang.Throwable: Bad position report
    at com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:68) ~[test-classes/:na]
    Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Assembly trace from producer [reactor.core.publisher.FluxConcatArray], described as [All Aircraft: after appending bad position report] :
    reactor.core.publisher.Flux.checkpoint(Flux.java:3261)
    com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:78)
Error has been observed at the following site(s):
    |_    Flux.checkpoint ⇢ at com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:78)
    |_                    ⇢ at com.thehecklers.aircraftpositions.PositionService$MockitoMock$1050388294.getAllAircraft(null:-1)
    |_          Flux.from ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:178)
    |_   Flux.collectList ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:179)
    |_           Mono.map ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:180)
    |_          Mono.flux ⇢ at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:181)
    |_          Flux.from ⇢ at org.springframework.http.server.reactive.ChannelSendOperator.<init>(ChannelSendOperator.java:57)
    |_                    ⇢ at org.springframework.http.codec.EncoderHttpMessageWriter.write(EncoderHttpMessageWriter.java:203)
    |_                    ⇢ at org.springframework.web.reactive.result.method.annotation.AbstractMessageWriterResultHandler.writeBody
  (AbstractMessageWriterResultHandler.java:104)
    |_                    ⇢ at org.springframework.web.reactive.result.method.annotation.ResponseBodyResultHandler.handleResult(ResponseBodyResultHandler.java:86)
    |_         checkpoint ⇢ Handler com.thehecklers.aircraftpositions.PositionController#getCurrentACPositions() [DispatcherHandler]
    |_                    ⇢ at org.springframework.web.server.handler.DefaultWebFilterChain.lambda$filter$0(DefaultWebFilterChain.java:120)
    |_         Mono.defer ⇢ at org.springframework.web.server.handler.DefaultWebFilterChain.filter(DefaultWebFilterChain.java:119)
    |_                    ⇢ at org.springframework.web.server.handler.FilteringWebHandler.handle(FilteringWebHandler.java:59)
    |_                    ⇢ at org.springframework.web.server.handler.WebHandlerDecorator.handle(WebHandlerDecorator.java:56)
    |_         Mono.error ⇢ at org.springframework.web.server.handler.ExceptionHandlingWebHandler$CheckpointInsertingHandler.handle(ExceptionHandlingWebHandler
  .java:98)
    |_         checkpoint ⇢ HTTP GET "/acpos" [ExceptionHandlingWebHandler]
    |_                    ⇢ at org.springframework.web.server.handler.ExceptionHandlingWebHandler.lambda$handle$0(ExceptionHandlingWebHandler.java:77)
    |_ Mono.onErrorResume ⇢ at org.springframework.web.server.handler.ExceptionHandlingWebHandler.handle(ExceptionHandlingWebHandler.java:77)
Stack trace:
        at com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:68) ~[test-classes/:na] <35 internal calls>
        at java.base/java.util.ArrayList.forEach(ArrayList.java:1510) ~[na:na] <9 internal calls>
        at java.base/java.util.ArrayList.forEach(ArrayList.java:1510) ~[na:na] <23 internal calls>
```

Figure 12-6. Standard checkpoint with description output

ReactorDebugAgent.init()

There is a way to realize the benefits of full backtracing for all `Publishers` within an application—like that produced using hooks—without the performance penalties imposed by enabling debugging using those same hooks.

Within the Reactor project is a library called `reactor-tools` that includes a separate Java agent used to instrument a containing application's code. `reactor-tools` adds debugging information to the application and attaches to the running application (of which it is a dependency) to track and trace execution of every subsequent `Publisher`, providing the same kind of detailed backtrace information as hooks with nearly zero

performance impact. As such, there are few if any downsides and numerous upsides to running reactive applications in production with `ReactorDebugAgent` enabled.

As a separate library, `reactor-tools` must be manually added to an application's build file. For the Aircraft Positions application's Maven *pom.xml*, I add the following entry:

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-tools</artifactId>
</dependency>
```

After saving the updated *pom.xml*, I refresh/reimport the dependencies to gain access to the `ReactorDebugAgent` within the project.

Like `Hooks.onOperatorDebug()`, the `ReactorDebugAgent` is typically initialized in the application's main method prior to running the app. Since I will be demonstrating this within a test that doesn't load the full application context, I insert the initialization call just as I did `Hooks.onOperatorDebug()`, immediately before constructing the `Flux` used to demonstrate a runtime execution error. I also remove the now-unnecessary calls to `checkpoint()`:

```
//Hooks.onOperatorDebug();
ReactorDebugAgent.init();        // Add this line
Mockito.when(service.getAllAircraft()).thenReturn(
        Flux.just(ac1, ac2, ac3)
                .concatWith(Flux.error(new Throwable("Bad position report")))
);
```

Returning once again to the `getCurrentACPositions()` test, I run it and am treated to the summary tree output shown in Figure 12-7, which is similar to that provided by `Hooks.onOperatorDebug()` but without runtime penalty:

```
java.lang.Throwable: Bad position report
    at com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:67) ~[test-classes/:na]
    Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Error has been observed at the following site(s):
    |_            → at com.thehecklers.aircraftpositions.PositionService$MockitoMock$563984815.getAllAircraft(null:-1)
    |_      Flux.from → at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:178)
    |_ Flux.collectList → at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:179)
    |_       Mono.map → at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:188)
    |_      Mono.flux → at org.springframework.http.codec.json.AbstractJackson2Encoder.encode(AbstractJackson2Encoder.java:181)
    |_      Flux.from → at org.springframework.http.server.reactive.ChannelSendOperator.<init>(ChannelSendOperator.java:57)
    |_                → at org.springframework.http.codec.EncoderHttpMessageWriter.write(EncoderHttpMessageWriter.java:203)
    |_                → at org.springframework.web.reactive.result.method.annotation.AbstractMessageWriterResultHandler.writeBody
(AbstractMessageWriterResultHandler.java:104)
    |_                → at org.springframework.web.reactive.result.method.annotation.ResponseBodyResultHandler.handleResult(ResponseBodyResultHandler.java:84)
    |_     checkpoint → Handler com.thehecklers.aircraftpositions.PositionController#getCurrentACPositions() [DispatcherHandler]
    |_                → at org.springframework.web.server.handler.DefaultWebFilterChain.lambda$filter$0(DefaultWebFilterChain.java:120)
    |_     Mono.defer → at org.springframework.web.server.handler.DefaultWebFilterChain.filter(DefaultWebFilterChain.java:119)
    |_                → at org.springframework.web.server.handler.FilteringWebHandler.handle(FilteringWebHandler.java:59)
    |_                → at org.springframework.web.server.handler.WebHandlerDecorator.handle(WebHandlerDecorator.java:56)
    |_     Mono.error → at org.springframework.web.server.handler.ExceptionHandlingWebHandler$CheckpointInsertingHandler.handle(ExceptionHandlingWebHandler
.java:98)
    |_     checkpoint → HTTP GET "/acpos" [ExceptionHandlingWebHandler]
    |_                → at org.springframework.web.server.handler.ExceptionHandlingWebHandler.lambda$handle$0(ExceptionHandlingWebHandler.java:77)
    |_ Mono.onErrorResume → at org.springframework.web.server.handler.ExceptionHandlingWebHandler.handle(ExceptionHandlingWebHandler.java:77)
Stack trace:
        at com.thehecklers.aircraftpositions.PositionControllerTest.setUp(PositionControllerTest.java:67) ~[test-classes/:na] <35 internal calls>
        at java.base/java.util.ArrayList.forEach(ArrayList.java:1510) ~[na:na] <9 internal calls>
        at java.base/java.util.ArrayList.forEach(ArrayList.java:1510) ~[na:na] <23 internal calls>
```

Figure 12-7. ReactorDebugAgent output resulting from `Flux` error

Other tools are available that don't directly help test or debug reactive applications but that nevertheless help to improve application quality. One example is BlockHound, which, although outside the scope of this chapter, can be a useful tool for determining if blocking calls are hidden within your application's code or its dependencies. And, of course, these and other tools are evolving and maturing rapidly to provide numerous ways to level up your reactive applications and systems.

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter12end* from the code repository.

Summary

Reactive programming gives developers a way to make better use of resources in distributed systems, even extending powerful scaling mechanisms across application boundaries and into the communication channels. For developers with experience exclusively with mainstream Java development practices—often called *imperative* Java due to its explicit and sequential logic versus the more declarative approach generally used in reactive programming—these reactive capabilities may bear some undesired costs. In addition to the expected learning curve, which Spring helps flatten considerably due to parallel and complementary WebMVC and WebFlux implementations, there are also relative limitations in tooling, its maturity, and established practices for essential activities like testing, troubleshooting, and debugging.

While it is true that reactive Java development is in its infancy relative to its imperative cousin, the fact that they are family has allowed a much faster development and maturation of useful tooling and processes. As mentioned, Spring builds similarly on established imperative expertise within its development and community to condense decades of evolution into production-ready components available *now*.

In this chapter, I introduced and elaborated on the current state of the art in testing and diagnosing/debugging issues you might encounter as you begin to deploy reactive Spring Boot applications. I then demonstrated how to put WebFlux/Reactor to work for you before and in production to test and troubleshoot reactive applications in various ways, showing relative advantages of each option available. You have a wealth of tools at your disposal even now, and the outlook is only getting better.

In this book, I had to choose which of the innumerable "best parts" of Spring Boot to cover in order to provide what I hope to be the best possible way to get up and running with Spring Boot. There is so much more,

and I only wish I could have doubled (or trebled) the scope of the book to do so. Thank you for accompanying me on this journey; I hope to share more in future. Best to you in your continued Spring Boot adventures.