Chapter 8. Reactive Programming with Project Reactor and Spring WebFlux

This chapter introduces reactive programming, discusses its origins and reasons for being, and demonstrates how Spring is leading the development and advancement of numerous tools and technologies that make it one of the best possible solutions for numerous use cases. More specifically, I demonstrate how to use Spring Boot and Project Reactor to drive database access using SQL and NoSQL databases, integrate reactive types with view technologies like Thymeleaf, and take interprocess communication to unexpected new levels with RSocket.

CODE CHECKOUT CHECKUP

Please check out branch *chapter8begin* from the code repository to begin.

Introduction to Reactive Programming

While a full treatise on reactive programming could—and has, and will—consume an entire book, it's critical to understand why it's such an important concept in the first place.

In a typical service, a thread is created for each request to be handled. Each thread requires resources, and as such, the number of threads that an application can manage is limited. As a somewhat simplified example, if an app can service 200 threads, that application can accept requests from up to 200 discrete clients at once, but no more; any additional attempts to connect to the service must wait for a thread to become available.

Performance for the 200 connected clients may or may not be satisfactory, depending on a number of factors. What is uncontestable is that for the client application making concurrent request number 201 and up, response time may be dramatically worse due to blocking by the service while it waits for an available thread. This hard stop in scalability can go from nonissue to crisis without warning and with no simple solution, and workarounds like the traditional "throw more instances at the problem" introduce both pressure relief and new problems to solve. Reactive programming was created to address this scalability crisis.

The Reactive Manifesto states that reactive systems are:

- Responsive

- Resilient
- Elastic
- Message driven

In a nutshell, the four key points of reactive systems as listed combine to create (at the macro level) a maximally available, scalable, and performant system requiring the fewest resources possible to do the job effectively.

Speaking at a systems level, i.e., several applications/services working together to fulfill various use cases, we might notice that most of the challenges involve communication between applications: one app responding to another, app/service availability when requests arrive, the ability of a service to scale out or in to adjust to demand, one service notifying other interested services of updated/available information, etc. Addressing the potential pitfalls of interapplication interactions can go a long way toward mitigating and/or solving the scalability issues referenced earlier.

This very observation of communication being the greatest potential source of issues and consequently the greatest opportunity for their resolution led to the Reactive Streams initiative. The Reactive Streams (RS) initiative focuses on the interactions among services—the Streams, if you will—and includes four key elements:

- The Application Programming Interface (API)
- The specification
- Examples for implementations
- The Technology Compatibility Kit (TCK)

The API consists of only four interfaces:

- `Publisher` : Creator of things
- `Subscriber` : Consumer of things
- `Subscription` : Contract between Publisher and Subscriber
- `Processor` : Incorporates both Subscriber and Publisher in order to receive, transform, and dispatch things

This lean simplicity is key, as is the fact that the API consists solely of *interfaces* and not *implementations*. This allows for various interoperable implementations across different platforms, languages, and programming models.

The textual specification details expected and/or required behavior for API implementations. For example:

```
If a Publisher fails it MUST signal an onError.
```

Examples for implementations are useful aids for implementors, providing reference code for use when creating a particular RS implementation.

Perhaps the most critical piece is the Technology Compatibility Kit. The TCK enables implementors to verify and demonstrate the level of compatibility—and any current shortcomings—with their RS implementation (or someone else's). Knowledge is power, and identifying anything that doesn't work in full compliance with the specification can speed resolution while providing a warning to current library consumers until the shortcoming is resolved.

NOTES ABOUT REACTIVE STREAMS, ASYNCHRONICITY, AND BACKPRESSURE

Reactive Streams build on a foundation of asynchronous communication and processing, as stated clearly in the purpose statement located in the very first paragraph of the Reactive Streams information site:

```
Reactive Streams is an initiative to provide a standard for asynchronous
stream processing with nonblocking back pressure. This encompasses efforts
aimed at runtime environments (JVM and JavaScript) as well as network
protocols.
```

At the risk of oversimplification, it may help to think in this way of the various concepts and components that compose Reactive Streams:

Asynchronicity is achieved when the application doesn't stop the world while one thing takes place. As an example, when Service A requests information from Service B, Service A doesn't postpone processing all subsequent instructions until receiving a response, idling (and thus wasting) precious compute resources while awaiting its answer from B; instead, Service A continues with other tasks until notified that a response has arrived.

As opposed to synchronous processing, in which tasks are executed sequentially, each one beginning only after the prior task completes, asynchronous processing can involve starting a task, then jumping to another if the original task can be performed in the background (or can await a notification of readiness/completion) and tasks can be performed concurrently. This enables fuller use of resources, since CPU time that would have been spent at idle or near-idle levels can be used for actual processing instead of just *waiting*. It also can—in some cases—improve performance.

Performance gains are not universal when adopting an asynchronous model of any kind. It's practically impossible to exceed the performance provided by a blocking, synchronous communication and processing model when only two services are interacting directly with only one exchange at a time. This is easily demonstrated: if Service B has only a single client, Service A, and Service A makes only a single request at a time and *blocks* all other activity, dedicating all resources to awaiting the response from Service B, this dedicated processing and connection results in the best possible performance for the two applications' interactions, all other circumstances being equal. This scenario and other similar ones are exceedingly rare, but they are possible.

Asynchronous processing adds minimal overhead due to implementation mechanisms like an event loop, in which a service "listens" for responses to pending requests. As a result, for scenarios involving very limited interapplication connections, performance may be slightly less than an exchange involving synchronous communications and processing. This outcome flips quickly as connections increase and thread exhaustion becomes a reality. Unlike with synchronous processing, resources aren't simply obligated and idled with asynchronous processing; they're repurposed and used, resulting in both increased resource utilization and application scalability.

Reactive Streams goes beyond asynchronous processing, adding nonblocking backpressure. This introduces flow control and robustness to interapplication communication.

Without backpressure, Service A can request information from Service B with no means of protecting against an overwhelming response. For example, if Service B returns one million objects/records, Service A dutifully tries to ingest all of them, likely buckling under the load. If Service A doesn't have sufficient compute and network resources to handle the staggering influx of information, the app can and will crash. At that point, asynchronicity isn't a factor, as app resources are wholly consumed trying (and failing) to keep up with the deluge. This is where backpressure demonstrates its value.

The concept of nonblocking backpressure simply means that Service A has a way to inform Service B of its capacity to handle response(s). Rather than just saying "give me everything," Service A requests a number of objects from Service B, processes them, and requests more when it is ready and able to process them. Originating in the field of fluid dynamics, the term *backpressure* represents a way of controlling flow from the point of origin by applying pressure back through the conduit against the supplier's flow of material(s). In Reactive Streams, backpressure provides Service A a way to manage the pace of incoming responses, setting and adjusting it in real time if circumstances change.

Although various workarounds have been created (with varying degrees of complexity, scope, and success) to implement means of backpressure within nonreactive systems, the declarative programming model favored by Reactive Streams makes the integration of asynchronicity and backpressure largely transparent and seamless to the developer.

Project Reactor

Although there are several available Reactive Streams implementations for the JVM, Project Reactor is among the most active, advanced, and performant. Reactor has been adopted by and provides the essential underpinnings for numerous mission-critical projects worldwide, including libraries, APIs, and applications developed and deployed by small organizations and global tech titans alike. Adding to this impressive momentum of development and adoption is the fact that Reactor provides the foundation for Spring's WebFlux reactive web capabilities, Spring Data's reactive database access for several open source and commercial databases, and interapplication communication, allowing for the creation of end-to-end reactive pipelines from top of stack to bottom and laterally as well. It's a 100% solution.

Why is this important?

From top of stack to bottom, from end user to lowest-tier computing resource, each interaction provides a potential sticking point. If interactions between the user's browser and the backend application are nonblocking but the app has to wait for a blocking interaction with the database, the result is a blocking system. The same goes with interapplication communication; if the user's browser communicates with backend Service A but Service A blocks waiting for a response from Service B, what has the user gained? Probably very little, and possibly nothing at all.

Developers usually can see the vast potential a switch to Reactive Streams offers them and their systems. A counterweight to that is the change in mindset that, combined with the relative newness of reactive (vs. imperative) programming constructs and tooling, can require adjustment and a bit more work from developers to harness, at least in the short term. This is still an easy decision to make as long as the effort required is clearly exceeded by the scalability benefits and the breadth and depth of Reactive Stream's application within overall systems. Having reactive pipelines throughout the entirety of a system's applications is a force multiplier on both counts.

Project Reactor's implementation of Reactive Streams is clean and simple, building on concepts with which Java and Spring developers are already well acquainted. Resembling Java 8+'s Stream API, Reactor is best utilized via declarative, chained operators, often with lambdas. Compared to more procedural, imperative code, it first feels somewhat different and then fairly elegant. Familiarity with `Stream` speeds the acclimatization and appreciation.

Reactor takes the concept of a Reactive Streams `Publisher` and specializes it, providing constructs similar to imperative Java in the process. Rather than using a common `Publisher` for everything in which a Reactive Stream—think of it as an on-demand, dynamic `Iterable`—is required, Project Reactor defines two types of `Publisher`:

`Mono` :: emits 0 or 1 element `Flux` :: emits 0 to *n* elements, a defined number or boundless

This aligns brilliantly with imperative constructs. For example, in standard Java, a method may return an object of type T or an `Iterable<T>`.

Using Reactor, that same method would return a `Mono<T>` or a `Flux<T>` —one object or potentially many, or in the case of the reactive code, a `Publisher` of those objects.

Reactor also fits very naturally into Spring's opinions. Depending on the use case, converting from blocking to nonblocking code can be as simple as changing a project dependency and a few method return values as shown previously. This chapter's examples demonstrate how to do exactly that, along with extending outward—up, down, and laterally—to go from a single reactive application to a reactive system, including reactive database access, for maximum benefit.

Tomcat versus Netty

In the imperative world of Spring Boot, Tomcat is the default servlet engine used for web applications, although even at that level, developers have options like Jetty and Undertow that can be used as drop-in replacements. Tomcat makes a great deal of sense as a default, though, as it is established, proven, and performant, and Spring team developers have contributed (and still do contribute) to refining and evolving Tomcat's codebase. It's a superb servlet engine for Boot applications.

That said, numerous iterations of the servlet specification have been intrinsically synchronous with no async capabilities. Servlet 3.0 began to address this with asynchronous request processing but still only supported traditional blocking I/O. Version 3.1 of the spec added nonblocking I/O, making it suitable for asynchronous, and thus also reactive, applications.

Spring WebFlux is the name for Spring's reactive counterpart to Spring WebMVC (package name), usually referred to simply as Spring MVC. Spring WebFlux is built on Reactor and uses Netty as the default network engine, just as Spring MVC uses Tomcat to listen for and service requests. Netty is a proven and performant asynchronous engine, and Spring team developers also contribute to Netty to tightly integrate Reactor and keep Netty on the cutting edge of features and performance.

Just as with Tomcat, though, you have options. Any Servlet 3.1–compatible engine can be used with Spring WebFlux applications, should your mission or organization require it. Netty is the category leader for a reason, however, and for the vast majority of use cases, it is the best choice.

Reactive Data Access

As mentioned previously, the ultimate goal for ultimate scalability and optimal systemwide throughput is a fully end-to-end reactive implementation. At the lowest level, this rests on database access.

Years of effort have gone into designing databases in ways to minimize contention and system performance blockages. Even with this impressive work, there are areas that remain problematic in many database engines and drivers, among them means for performing operations without blocking the requesting application(s) and sophisticated flow control/backpressure mechanisms.

Paging constructs have been used to address both of these constraints, but they are imperfect solutions. Using an imperative model with paging typically requires a query to be issued for each page with a different range and/or constraints. This requires a new request and new response each time instead of the continuation that is possible with a `Flux`. An analogy is scooping up one cup of water at a time from a basin (imperative approach) versus simply turning on the tap to refill the cup. Rather than a "go get, bring back" imperative operation, the water is waiting to flow in the reactive scenario.

R2DBC with H2

In the existing version of PlaneFinder, I use the Java Persistence API (JPA) and the H2 database to store (in an in-memory instance of H2) aircraft positions retrieved from my local device that monitors in-range aircraft. JPA was built on an imperative specification and is thus inherently blocking. Seeing the need for a nonblocking reactive means of interacting with SQL databases, several industry leaders and luminaries joined forces to create and evolve the Reactive Relational Database Connectivity (R2DBC) project.

Like JPA, R2DBC is an open specification that can be used, along with the Service Provider Interface (SPI) it provides, by vendors or other interested parties to create drivers for relational databases and client libraries for downstream developers. Unlike JPA, R2DBC builds on Project Reactor's implementation of Reactive Streams and is fully reactive and nonblocking.

Updating PlaneFinder

As with most complex systems, we don't (currently) control all aspects and nodes of the entire distributed system. Also like most complex systems, the more completely a paradigm is embraced, the more can be gained from it. I start this "journey to reactive" as close to the point of origin of the communication chain as possible: in the PlaneFinder service.

Refactoring PlaneFinder to use Reactive Streams `Publisher` types, e.g., `Mono` and `Flux`, is the first step. I'll stay with the existing H2 database, but in order to "reactiv-ate" it, I need to remove the JPA project dependency and replace it with R2DBC libraries. I update PlaneFinder's *pom.xml* Maven build file as follows:

```
<!--    Comment out or remove this      -->
<!--<dependency>-->
<!--    <groupId>org.springframework.boot</groupId>-->
<!--    <artifactId>spring-boot-starter-data-jpa</artifactId>-->
<!--</dependency>-->

<!--    Add this                        -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-r2dbc</artifactId>
</dependency>

<!--    Add this too                -->
<dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

The `PlaneRepository` interface must be updated to extend the `ReactiveCrudRepository` interface instead of its blocking counterpart `CrudRepository`. This simple update is shown here:

```
public interface PlaneRepository
    extends ReactiveCrudRepository<Aircraft, String> {}
```

The change to `PlaneRepository` ripples outward, which leads naturally to the next stop, the `PlaneFinderService` class, where the `getAircraft()` method returns the result of `PlaneRepository::saveAll` when aircraft are found and of the `saveSamplePositions()` method otherwise. Replacing the value returned, a blocking `Iterable<Aircraft>`, with `Flux<Aircraft>` for the `getAircraft()` and `saveSamplePositions()` methods again correctly specifies the method return value.

```
public Flux<Aircraft> getAircraft() {
    ...
}

private Flux<Aircraft> saveSamplePositions() {
    ...
}
```

Since the `PlaneController` class's method `getCurrentAircraft()` calls `PlaneFinderService::getAircraft`, it now returns a `Flux<Aircraft>`. This necessitates a change to the signature for `PlaneController::getCurrentAircraft` as well:

```
public Flux<Aircraft> getCurrentAircraft() throws IOException {
    ...
}
```

Using H2 with JPA is a fairly mature affair; the specifications involved, along with the relevant APIs and libraries, have been under development for roughly a decade. R2DBC is a relatively recent development, and while support is expanding apace, a few features present in the Spring Data JPA's support for H2 have yet to be implemented. This doesn't pose much of an increased burden but is something to keep in mind when choosing to use a relational database—in this case, H2—reactively.

Currently, to use H2 with R2DBC, it is necessary to create and configure a `ConnectionFactoryInitializer` bean for use by the application. Configuration requires only two steps in reality:

- Setting the connection factory to the (already autoconfigured) `ConnectionFactory` bean, injected as a parameter
- Configuring the database "populator" to execute one or more scripts to initialize or reinitialize the database as desired/required

Recall that when using Spring Data JPA with H2, an associated `@Entity` class is used to create a corresponding table within the H2 database. This step is completed manually when using H2 with R2DBC using a standard SQL DDL (Data Definition Language) script.

```
DROP TABLE IF EXISTS aircraft;

CREATE TABLE aircraft (id BIGINT auto_increment primary key,
callsign VARCHAR(7), squawk VARCHAR(4), reg VARCHAR(8), flightno VARCHAR(10),
route VARCHAR(30), type VARCHAR(4), category VARCHAR(2),
altitude INT, heading INT, speed INT, vert_rate INT, selected_altitude INT,
lat DOUBLE, lon DOUBLE, barometer DOUBLE, polar_distance DOUBLE,
polar_bearing DOUBLE, is_adsb BOOLEAN, is_on_ground BOOLEAN,
last_seen_time TIMESTAMP, pos_update_time TIMESTAMP, bds40_seen_time TIMESTAMP
```

NOTE

This is an additional step, but it isn't without precedent. Many SQL databases require this step when used with Spring Data JPA as well; H2 was an exception to the rule.

Next up is the code for the `DbConxInit`, or Database Connection Initializer, class. The required bean-creation method is the first one—`initializer()`—that produces the needed `ConnectionFactoryInitializer` bean. The second method produces a `CommandLineRunner` bean that is executed once the class is configured.

`CommandLineRunner` is a functional interface with a single abstract method, `run()`. As such, I provide a lambda as its implementation, populating (and then listing) the contents of the `PlaneRepository` with a single `Aircraft`. Currently I have the `@Bean` annotation for the `init()` method commented out, so the method is never called, the `CommandLineRunner` bean is never produced, and the sample record is never stored:

```java
import io.r2dbc.spi.ConnectionFactory;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.r2dbc.connection.init.ConnectionFactoryInitializer;
import org.springframework.r2dbc.connection.init.ResourceDatabasePopulator;

@Configuration
public class DbConxInit {
    @Bean
    public ConnectionFactoryInitializer
            initializer(@Qualifier("connectionFactory")
            ConnectionFactory connectionFactory) {
        ConnectionFactoryInitializer initializer =
            new ConnectionFactoryInitializer();
        initializer.setConnectionFactory(connectionFactory);
        initializer.setDatabasePopulator(
            new ResourceDatabasePopulator(new ClassPathResource("schema.sql"))
        );
        return initializer;
    }

//    @Bean // Uncomment @Bean annotation to add sample data
    public CommandLineRunner init(PlaneRepository repo) {
        return args -> {
            repo.save(new Aircraft("SAL001", "N12345", "SAL001", "LJ",
                    30000, 30, 300,
                    38.7209228, -90.4107416))
                .thenMany(repo.findAll())
                    .subscribe(System.out::println);
        };
    }
}
```

The `CommandLineRunner` lambda merits some explanation.

The structure itself is a typical lambda of `x -> { <code to execute here> }`, but the code contained within has a couple of interesting Reactive Streams–specific features.

The first declared operation is `repo::save`, which saves the content provided—in this case, a new `Aircraft` object—and returns a `Mono<Aircraft>`. It's possible to simply `subscribe()` to this result and log/print it for verification. But a good habit to adopt is to save all desired sample data, then query the repository to produce all records. Doing so allows for full verification of the final state of the table at that point in time and should result in all records being displayed.

Recall, though, that reactive code doesn't block, so how can one be certain that all previous operations have completed prior to proceeding? In this case, how can we be sure all records are saved before trying to retrieve all records? Within Project Reactor there are operators that await the completion signal, then proceed with the next function in the chain. The `then()` operator waits for a `Mono` as input, then accepts another `Mono` to play going forward. The `thenMany()` operator shown in the previous example awaits the completion of any upstream `Publisher` and plays a new `Flux` going forward. In the `init` method that produces the `CommandLineRunner` bean, `repo.findAll()` produces a `Flux<Aircraft>`, filling the bill as expected.

Finally, I subscribe to the `Flux<Aircraft>` output from `repo.findAll()` and print the results to the console. It isn't necessary to log the results, and in fact a plain `subscribe()` fulfills the requirement to start the flow of data. But why is it necessary to subscribe?

With few exceptions, `Reactive Streams Publisher`s are *cold publishers*, meaning they perform no work and consume no resources if they have no subscriber(s). This maximizes efficiency and thus scalability and makes perfect sense, but it also provides a common trap for those new to reactive programming. If you aren't returning a `Publisher` to calling code for subscription and use there, be sure to add a `subscribe()` to it to activate the `Publisher` or chain of operations that results in one.

GOING DECLARATIVE

I often refer to nonreactive code as *blocking*, which makes sense in most cases, as code (with a few notable exceptions) executes sequentially; one line of code begins after the previous one finishes. Reactive code doesn't block, though—unless it calls blocking code, which I address in an upcoming chapter—and as a result, sequential lines of code don't provide any delineation between instructions whatsoever. This can be a bit jarring, especially for developers coming from a sequential execution background or who still work in one much or all of the time, which is most of us.

Most blocking code is imperative code, in which we specify *how* to do something. Using a *for* loop as an example, we do the following:

- Declare a variable and assign an initial value.
- Check against an outer boundary.
- Perform some instructions.
- Adjust the variable's value.
- Repeat the loop beginning with the value check.

While there are very useful declarative constructs in blocking code—perhaps the best-known and -loved is the Java Stream API—these declarative morsels add zest to the meal but together still compose a relatively small portion of it. Not so with reactive programming.

Because of the name Reactive Streams, you might form associations with Java's `Stream`. Although the two are not connected, the declarative approach used in `java.util.Stream` fits perfectly with Reactive Streams as well: declaring outcomes via a chain of functions that operate by passing output from one to the next as immutable results. This adds structure, both visually and logically, to reactive code.

Finally, some changes to the domain class `Aircraft` are required due to differences in JPA and R2DBC and their supporting H2 code. The `@Entity` notation used by JPA is no longer required, and the `@GeneratedValue` annotation for the primary key-associated member variable `id` is now similarly unnecessary. Removing both of these and their associated import statements are the only required changes when migrating from PlaneFinder from JPA to R2DBC using H2.

To accommodate the `CommandLineRunner` bean shown earlier (should sample data be desired) and its field-limited constructor call, I create an additional constructor in `Aircraft` to match. Note that this is required only if you wish to create an `Aircraft` instance without providing all parameters as required by the constructor Lombok based on the `@AllArgsConstructor` annotation. Note that I call the all-args constructor from this limited-args constructor:

```
    public Aircraft(String callsign, String reg, String flightno, String type,
                    int altitude, int heading, int speed,
                    double lat, double lon) {

        this(null, callsign, "sqwk", reg, flightno, "route", type, "ct",
                altitude, heading, speed, 0, 0,
                lat, lon, 0D, 0D, 0D,
                false, true,
                Instant.now(), Instant.now(), Instant.now());
    }
```

With that, it’s time to verify our work.

After starting the PlaneFinder application from within the IDE, I return to HTTPie in a terminal window to test the updated code:

```
mheckler-a01 :: OReilly/code » http -b :7634/aircraft
[
    {
        "altitude": 37000,
        "barometer": 0.0,
        "bds40_seen_time": null,
        "callsign": "EDV5123",
        "category": "A3",
        "flightno": "DL5123",
        "heading": 131,
        "id": 1,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-09-19T21:40:56Z",
        "lat": 38.461505,
        "lon": -89.896606,
        "polar_bearing": 156.187542,
        "polar_distance": 32.208164,
        "pos_update_time": "2020-09-19T21:40:56Z",
        "reg": "N582CA",
        "route": "DSM-ATL",
        "selected_altitude": 0,
        "speed": 474,
        "squawk": "3644",
        "type": "CRJ9",
        "vert_rate": -64
    },
    {
        "altitude": 38000,
        "barometer": 0.0,
        "bds40_seen_time": null,
        "callsign": null,
        "category": "A4",
        "flightno": "FX3711",
        "heading": 260,
        "id": 2,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-09-19T21:40:57Z",
        "lat": 39.348558,
        "lon": -90.330383,
        "polar_bearing": 342.006425,
        "polar_distance": 24.839372,
        "pos_update_time": "2020-09-19T21:39:50Z",
        "reg": "N924FD",
        "route": "IND-PHX",
        "selected_altitude": 0,
        "speed": 424,
        "squawk": null,
        "type": "B752",
        "vert_rate": 0
    },
    {
        "altitude": 35000,
        "barometer": 1012.8,
        "bds40_seen_time": "2020-09-19T21:41:11Z",
        "callsign": "JIA5304",
        "category": "A3",
        "flightno": "AA5304",
        "heading": 112,
        "id": 3,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-09-19T21:41:12Z",
        "lat": 38.759811,
        "lon": -90.173632,
        "polar_bearing": 179.833023,
        "polar_distance": 11.568717,
        "pos_update_time": "2020-09-19T21:41:11Z",
        "reg": "N563NN",
        "route": "CLT-RAP-CLT",
        "selected_altitude": 35008,
        "speed": 521,
        "squawk": "6506",
        "type": "CRJ9",
        "vert_rate": 0
    }
]
```

Confirming that the refactored, reactive PlaneFinder works properly, we can now turn our attention to the Aircraft Positions application.

Updating the Aircraft Positions application

Currently the *aircraft-positions* project uses Spring Data JPA and H2, just as PlaneFinder did when it was a blocking application. While I could update Aircraft Positions to use R2DBC and H2 just as PlaneFinder now does, this required refactoring of the *aircraft-positions* project offers the perfect opportunity to explore other reactive database solutions.

MongoDB is often at the forefront of database innovation, and indeed it was one of the first database providers of any kind to develop fully reactive drivers for use with its namesake database. Developing applications using Spring Data and MongoDB is nearly frictionless, reflecting the maturity of its reactive streams support. For the reactive refactoring of Aircraft Positions, MongoDB is a natural choice.

Some changes to the build file, *pom.xml* in this case, are in order. First I remove the unnecessary dependencies for Spring MVC, Spring Data JPA, and H2:

- `spring-boot-starter-web`
- `spring-boot-starter-data-jpa`
- `h2`

Next I add the following dependencies for the reactive version going forward:

- `spring-boot-starter-data-mongodb-reactive`
- `de.flapdoodle.embed.mongo`
- `reactor-test`

NOTE

`spring-boot-starter-webflux` was already a dependency due to `WebClient`, so it wasn't necessary to add it.

As in Chapter 6, I will make use of the embedded MongoDB for this example. Since the embedded MongoDB is typically used only for testing, it usually includes a scope of "test"; since I use this during application execution, I omit or remove that scoping qualifier from the build file. The updated Maven *pom.xml* dependencies look like this:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
```

```
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>de.flapdoodle.embed</groupId>
            <artifactId>de.flapdoodle.embed.mongo</artifactId>
        </dependency>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

A quick refresh to the dependencies either via command line or the IDE and we're ready to refactor.

I begin again with the very simple change to the `AircraftRepository` interface, changing it to extend `ReactiveCrudRepository` instead of the blocking `CrudRepository`:

```
public interface AircraftRepository extends ReactiveCrudRepository<Aircraft, I
```

Updating the `PositionController` class is a fairly small chore, since `WebClient` already converses using Reactive Streams `Publisher` types. I define a local variable `Flux<Aircraft> aircraftFlux`, then chain the requisite declarative operations to clear the repository of previously retrieved aircraft positions, retrieve new positions, convert them to instances of the `Aircraft` class, filter out positions without a listed aircraft registration number, and save them to the embedded MongoDB repository. I then add the `aircraftFlux` variable to the `Model` for use in the user-facing web UI and return the name of the Thymeleaf template for rendering:

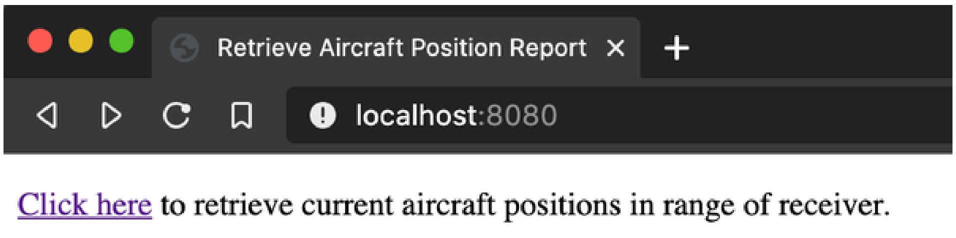
```
@RequiredArgsConstructor
@Controller
public class PositionController {
    @NonNull
    private final AircraftRepository repository;
    private WebClient client
        = WebClient.create("http://localhost:7634/aircraft");

    @GetMapping("/aircraft")
    public String getCurrentAircraftPositions(Model model) {
        Flux<Aircraft> aircraftFlux = repository.deleteAll()
                .thenMany(client.get()
                        .retrieve()
                        .bodyToFlux(Aircraft.class)
                        .filter(plane -> !plane.getReg().isEmpty())
                        .flatMap(repository::save));

        model.addAttribute("currentPositions", aircraftFlux);
        return "positions";
    }
}
```

Finally, a few small changes are required for the domain class `Aircraft` itself. The class-level `@Entity` annotation is JPA-specific; the corresponding annotation used by MongoDB is `@Document`, indicating that instances of a class are to be stored as documents within the database. Additionally, the `@Id` annotation used previously referenced `javax.persistence.Id`, which disappears without the JPA dependency. Replacing `import javax.persistence.Id;` with `import org.springframework.data.annotation.Id;` retains the table identifier context for use with MongoDB. The class file in its entirety is shown for reference:

```
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.time.Instant;

@Document
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Aircraft {
    @Id
    private Long id;
    private String callsign, squawk, reg, flightno, route, type, category;

    private int altitude, heading, speed;
    @JsonProperty("vert_rate")
    private int vertRate;
    @JsonProperty("selected_altitude")
    private int selectedAltitude;
```

```
        private double lat, lon, barometer;
        @JsonProperty("polar_distance")
        private double polarDistance;
        @JsonProperty("polar_bearing")
        private double polarBearing;

        @JsonProperty("is_adsb")
        private boolean isADSB;
        @JsonProperty("is_on_ground")
        private boolean isOnGround;

        @JsonProperty("last_seen_time")
        private Instant lastSeenTime;
        @JsonProperty("pos_update_time")
        private Instant posUpdateTime;
        @JsonProperty("bds40_seen_time")
        private Instant bds40SeenTime;
    }
```

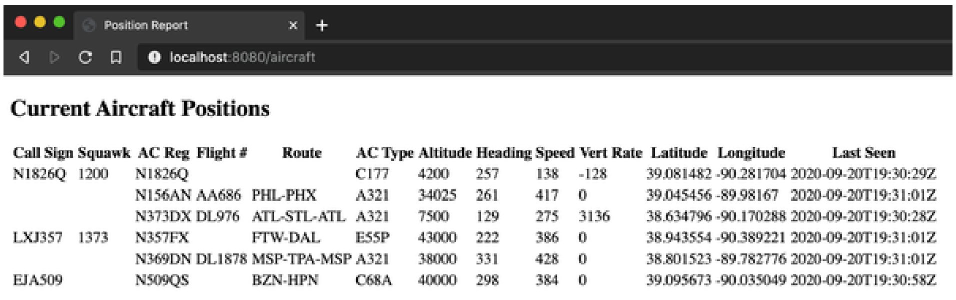
Running both the PlaneFinder and Aircraft Positions applications, I return to a browser tab and type *http://localhost:8080* into the address bar and load it, resulting in the page shown in Figure 8-1.



Click here to retrieve current aircraft positions in range of receiver.

Figure 8-1. The Aircraft Positions application landing page, *index.html*

Clicking on the *Click here* link loads the `Aircraft Positions` report page, as shown in Figure 8-2.



Current Aircraft Positions

Call Sign	Squawk	AC Reg	Flight #	Route	AC Type	Altitude	Heading	Speed	Vert Rate	Latitude	Longitude	Last Seen
N1826Q	1200	N1826Q			C177	4200	257	138	-128	39.081482	-90.281704	2020-09-20T19:30:29Z
		N156AN	AA686	PHL-PHX	A321	34025	261	417	0	39.045456	-89.98167	2020-09-20T19:31:01Z
		N373DX	DL976	ATL-STL-ATL	A321	7500	129	275	3136	38.634796	-90.170288	2020-09-20T19:30:28Z
LXJ357	1373	N357FX		FTW-DAL	E55P	43000	222	386	0	38.943554	-90.389221	2020-09-20T19:31:01Z
		N369DN	DL1878	MSP-TPA-MSP	A321	38000	331	428	0	38.801523	-89.782776	2020-09-20T19:31:01Z
EJA509		N509QS		BZN-HPN	C68A	40000	298	384	0	39.095673	-90.035049	2020-09-20T19:30:58Z

Figure 8-2. The Aircraft Positions report page

With each periodic refresh, the page will requery PlaneFinder and update the report with current data on demand as before, with one very key difference: the multiple aircraft positions that are supplied to the *positions.html* Thymeleaf template for display are no longer a fully formed, blocking `List` but rather a Reactive Streams `Publisher`, specifically of type `Flux`. The next section addresses this further, but for now, it's important to realize that this content negotiation/accommodation occurs with no effort required from the developer.

Reactive Thymeleaf

As mentioned in Chapter 7, the vast majority of frontend web applications are now being developed using HTML and JavaScript. This doesn't alter the existence of a number of production applications that use view technologies/templating to fulfill their objectives; neither does it imply that said technologies don't continue to satisfy a range of requirements simply and effectively. This being the case, it's important for template engines and languages to adapt to circumstances in which Reactive Streams are also brought to bear on a problem.

Thymeleaf approaches RS support at three different levels, allowing developers to settle on the one that best fits their requirements. As mentioned earlier, it's possible to convert backend processing to leverage Reactive Streams and let Reactor feed Thymeleaf values supplied by a `Publisher`—like a `Mono` or `Flux`—instead of `Object<T>` and `Iterable<T>`. This doesn't result in a reactive frontend, but if the concern is primarily conversion of backend logic to use Reactive Streams to eliminate blocking and implement flow control among services, this is a frictionless on-ramp to deploying a supporting user-facing application with the least possible effort.

Thymeleaf also supports chunked and data-driven modes in support of Spring WebFlux, both involving the use of Server Sent Events and some JavaScript code to accomplish the feed of data to the browser. While both of these modalities are entirely valid, the increased amount of JavaScript required to achieve the desired outcome may tip the scales away from templating+HTML+JavaScript and toward 100% HTML+JavaScript frontend logic. This decision is heavily dependent on requirements, of course, and should be left to the developer(s) tasked with creating and supporting said functionality.

In the preceding section, I demonstrated how to migrate the backend functionality to RS constructs and how Spring Boot uses Reactor+Thymeleaf to maintain functionality in the front end, helping ease conversions of blocking systems of applications while minimizing downtime. This is sufficient to satisfy the current use case, allowing us to examine ways to further improve backend functionality before returning (in an upcoming chapter) to expanding frontend capabilities.

RSocket for Fully Reactive Interprocess

Communication

Already in this chapter I've laid the groundwork for interprocess communication using Reactive Streams between separate applications. While the distributed system created does indeed use reactive constructs, the system has yet to reach its potential. Crossing the network boundary using higher-level HTTP-based transports imposes limitations due to the request-response model, and even upgrading to WebSocket alone doesn't address all of them. RSocket was created to eliminate interprocess communication shortfalls flexibly and powerfully.

What Is RSocket?

The result of a collaboration among several industry leaders and cutting-edge innovators, RSocket is a blazing-fast binary protocol that can be used over TCP, WebSocket, and Aeron transport mechanisms. RSocket supports four asynchronous interaction models:

- Request-response
- Request-stream
- Fire & forget
- Request channel (bidirectional stream)

RSocket builds on the reactive streams paradigm and Project Reactor, enabling fully interconnected systems of applications while providing mechanisms that increase flexibility and resilience. Once a connection is made between two apps/services, distinctions of client versus server disappear and the two are effectively peers. Any of the four interaction models can be initiated by either party and accommodate all use cases:

- A 1:1 interaction in which one party issues a request and receives a response from the other party
- A 1:N interaction in which one party issues a request and receives a stream of responses from the other party
- A 1:0 interaction in which one party issues a request
- A fully bidirectional channel in which both parties can send requests, responses, or data streams of any kind unbidden

As you can see, RSocket is incredibly flexible. Being a binary protocol with a performance focus, it is also fast. On top of that, RSocket is resilient, making it possible for a dropped connection to be reestablished and communications to automatically resume where they left off. And since RSocket is built on Reactor, developers who use RSocket can truly consider separate applications as a fully integrated system, since the network boundary no longer imposes any limitations on flow control.

Spring Boot, with its legendary autoconfiguration, arguably provides the fastest, most developer-friendly way for Java and Kotlin developers to use RSocket.

Putting RSocket to Work

Currently both the PlaneFinder and Aircraft Positions applications use HTTP-based transports to communicate. Converting both Spring Boot apps to use RSocket is the obvious next step forward.

Migrating PlaneFinder to RSocket

First, I add the RSocket dependency to the PlaneFinder build file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-rsocket</artifactId>
</dependency>
```

After a quick Maven re-import, it's off to refactor the code.

For the time being, I'll leave the existing endpoint of */aircraft* intact and add an RSocket endpoint to `PlaneController`. In order to place both REST endpoints and RSocket endpoints in the same class, I decouple the functionality built into the `@RestController` annotation into its component parts: `@Controller` and `@ResponseBody`.

Replacing the class-level `@RestController` annotation with `@Controller` means that for any REST endpoints from which we wish to return objects directly as JSON—such as the existing */aircraft* endpoint associated with the `getCurrentAircraft()` method—it is necessary to add `@ResponseBody` to the method. The advantage to this seeming step back is that RSocket endpoints can then be defined in the same `@Controller` class as REST endpoints, keeping points of ingress and egress for PlaneFinder in one, and only one, location:

```
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import reactor.core.publisher.Flux;

import java.io.IOException;
import java.time.Duration;

@Controller
public class PlaneController {
    private final PlaneFinderService pfService;
    public PlaneController(PlaneFinderService pfService) {
        this.pfService = pfService;
    }

    @ResponseBody
    @GetMapping("/aircraft")
    public Flux<Aircraft> getCurrentAircraft() throws IOException {
        return pfService.getAircraft();
    }

    @MessageMapping("acstream")
    public Flux<Aircraft> getCurrentACStream() throws IOException {
        return pfService.getAircraft().concatWith(
                Flux.interval(Duration.ofSeconds(1))
                        .flatMap(l -> pfService.getAircraft()));
    }
}
```

To create a repeating stream of aircraft positions sent initially and at subsequent one-second intervals, I create the `getCurrentACStream()` method and annotate it as an RSocket endpoint with `@MessageMapping`. Note that since RSocket mappings don't build upon a root path as HTTP addresses/endpoints do, no forward slash (/) is required in the mapping.

With the endpoint and servicing method defined, the next step is to designate a port for RSocket to listen for connection requests. I do so in PlaneFinder's *application.properties* file, adding a property value for `spring.rsocket.server.port` to the existing one for the HTTP-based `server.port`:

```
server.port=7634
spring.rsocket.server.port=7635
```

The presence of this single RSocket server port assignment is sufficient for Spring Boot to configure the containing application as an RSocket server, creating all necessary beans and performing all of the requisite configuration. Recall that while one of the two applications involved in an RSocket connection must act initially as a server, once the connection is established the distinction between client (the app that initiates a connection) and server (the app that listens for a connection) evaporates.

With those few changes, PlaneFinder is now RSocket ready. Simply start the application to ready it for connection requests.

Migrating Aircraft Positions to RSocket

Once again, the first step in adding RSocket is to add the RSocket dependency to the build file—in this case, for the Aircraft Positions application:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-rsocket</artifactId>
</dependency>
```

Don't forget to re-import and thus activate changes with Maven for the project prior to continuing. Now, on to the code.

Similarly to how I did with PlaneFinder, I refactor the `PositionController` class to create a single point for all ingress/egress. Replacing the class-level `@RestController` annotation with `@Controller` allows for the inclusion of RSocket endpoints along with the HTTP-based (but template-driven, in this case) endpoint that activates the *positions.html* Thymeleaf template.

To enable Aircraft Positions to act as an RSocket client, I create an `RSocketRequester` by autowiring via constructor injection an `RSocketRequester.Builder` bean. The `RSocketRequester.Builder` bean is automatically created by Spring Boot as a result of adding the RSocket dependency to the project. Within the constructor, I use the builder to create a TCP connection (in this case) to PlaneFinder's RSocket server via the builder's `tcp()` method.

NOTE

Since I need to inject a bean (`RSocketRequester.Builder`) used to create an instance of a different object (`RSocketRequester`), I must create a constructor. Since I now have a constructor, I removed the class-level `@RequiredArgsConstructor` and member variable-level `@NonNull` Lombok annotations and simply add `AircraftRepository` to the constructor I wrote as well. Either way, Spring Boot autowires the bean, and it is assigned to the `repository` member variable.

To verify the RSocket connection is working properly and data is flowing, I create an HTTP-based endpoint */acstream*, specify it will return a stream of Server Sent Events (SSE) as a result, and with the `@ResponseBody` annotation indicate that the response will comprise JSON-formatted objects directly. Using the `RSocketRequester` member variable initialized in the constructor, I specify the `route` to match the RSocket endpoint defined in PlaneFinder, send some `data` (optional; I don't pass any useful data in this particular request), and retrieve the `Flux` of `Aircraft` returned from PlaneFinder:

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

@Controller
public class PositionController {
    private final AircraftRepository repository;
    private final RSocketRequester requester;
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");

    public PositionController(AircraftRepository repository,
                              RSocketRequester.Builder builder) {
        this.repository = repository;
        this.requester = builder.tcp("localhost", 7635);
    }

    // HTTP endpoint, HTTP requester (previously created)
    @GetMapping("/aircraft")
    public String getCurrentAircraftPositions(Model model) {
        Flux<Aircraft> aircraftFlux = repository.deleteAll()
                .thenMany(client.get()
                        .retrieve()
                        .bodyToFlux(Aircraft.class)
                        .filter(plane -> !plane.getReg().isEmpty())
                        .flatMap(repository::save));

        model.addAttribute("currentPositions", aircraftFlux);
        return "positions";
    }

    // HTTP endpoint, RSocket client endpoint
    @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);
    }
}
```

To verify the RSocket connection is viable and PlaneFinder is feeding data to the Aircraft Positions application, I start Aircraft Positions and return to the terminal and HTTPie, adding the *-S* flag to the command to process the data as a stream, as it arrives, rather than wait for a response body completion. An example of the results follows, edited for brevity:

```
mheckler-a01 :: ~ » http -S :8080/acstream
HTTP/1.1 200 OK
Content-Type: text/event-stream;charset=UTF-8
transfer-encoding: chunked
```

```
data:{"id":1,"callsign":"RPA3427","squawk":"0526","reg":"N723YX","flightno":
"UA3427","route":"IAD-MCI","type":"E75L","category":"A3","altitude":36000,
"heading":290,"speed":403,"lat":39.183929,"lon":-90.72259,"barometer":0.0,
"vert_rate":64,"selected_altitude":0,"polar_distance":29.06486,
"polar_bearing":297.519943,"is_adsb":true,"is_on_ground":false,
"last_seen_time":"2020-09-20T23:58:51Z",
"pos_update_time":"2020-09-20T23:58:49Z","bds40_seen_time":null}

data:{"id":2,"callsign":"EDG76","squawk":"3354","reg":"N776RB","flightno":"",
"route":"TEB-VNY","type":"GLF5","category":"A3","altitude":43000,"heading":256
"speed":419,"lat":38.884918,"lon":-90.363026,"barometer":0.0,"vert_rate":64,
"selected_altitude":0,"polar_distance":9.699159,"polar_bearing":244.237695,
"is_adsb":true,"is_on_ground":false,"last_seen_time":"2020-09-20T23:59:22Z",
"pos_update_time":"2020-09-20T23:59:14Z","bds40_seen_time":null}

data:{"id":3,"callsign":"EJM604","squawk":"3144","reg":"N604SD","flightno":"",
"route":"ENW-HOU","type":"C56X","category":"A2","altitude":38000,"heading":201
"speed":387,"lat":38.627464,"lon":-90.01416,"barometer":0.0,"vert_rate":-64,
"selected_altitude":0,"polar_distance":20.898095,"polar_bearing":158.9935,
"is_adsb":true,"is_on_ground":false,"last_seen_time":"2020-09-20T23:59:19Z",
"pos_update_time":"2020-09-20T23:59:19Z","bds40_seen_time":null}
```

This confirms that data is flowing from PlaneFinder to Aircraft Positions via Reactive Streams over an RSocket connection using the *request-stream* model. All systems go.

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter8end* from the code repository.

Summary

Reactive programming gives developers a way to make better use of resources, and in an increasingly distributed world of interconnected systems, the master key to scalability involves extending scaling mechanisms beyond application boundaries and into the communication channels. The Reactive Streams initiative, and in particular Project Reactor, serves as a powerful, performant, and flexible foundation for maximizing system-wide scalability.

In this chapter, I introduced reactive programming and demonstrated how Spring is leading the development and advancement of numerous tools and technologies. I explained blocking and nonblocking communication and the engines that provide those capabilities, e.g., Tomcat, Netty, and others.

Next, I demonstrated how to enable reactive database access to SQL and NoSQL databases by refactoring the PlaneFinder and Aircraft Positions applications to use Spring WebFlux/Project Reactor. Reactive Relational Database Connectivity (R2DBC) provides a reactive replacement for the Java Persistence API (JPA) and works with several SQL databases; MongoDB and other NoSQL databases provide drop-in reactive drivers that work seamlessly with Spring Data and Spring Boot.

This chapter also discussed options for frontend integration of reactive types and demonstrated how Thymeleaf provides a limited migration path if your applications are still using generated view technologies. Additional options will be considered in future chapters.

Finally, I demonstrated how to take interprocess communication to unexpected new levels with RSocket. Doing so via Spring Boot's RSocket support and autoconfiguration provides the fast path to performance, scalability, resilience, and developer productivity.

In the next chapter, I'll dig into testing: how Spring Boot enables better, faster, and easier testing practices, how to create effective unit tests, and how to hone and focus testing to speed the build-and-test cycle.