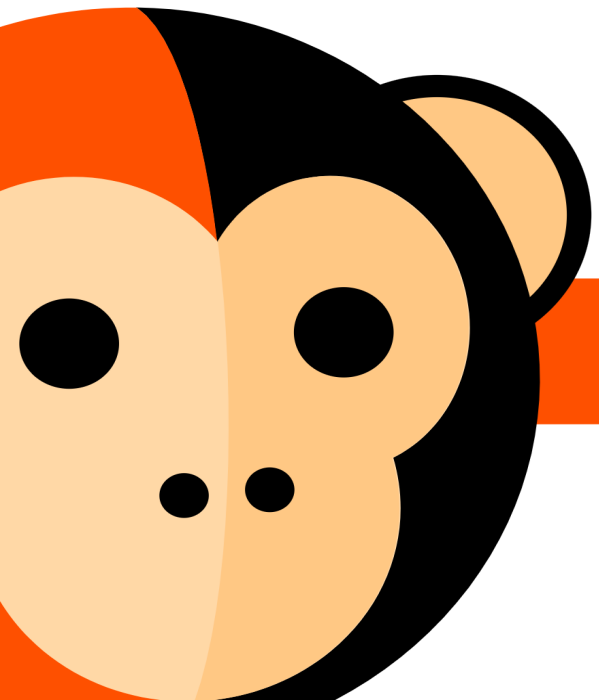
Full Reactive Stack

with Spring Boot 2, Spring WebFlux, Reactive MongoDB, and Angular



Moisés Macero García

ThePracticalDeveloper.com

Full Reactive Stack with Spring Boot, WebFlux and MongoDB

Moisés Macero

This book is for sale at http://leanpub.com/full-reactive

This version was published on 2020-06-15

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

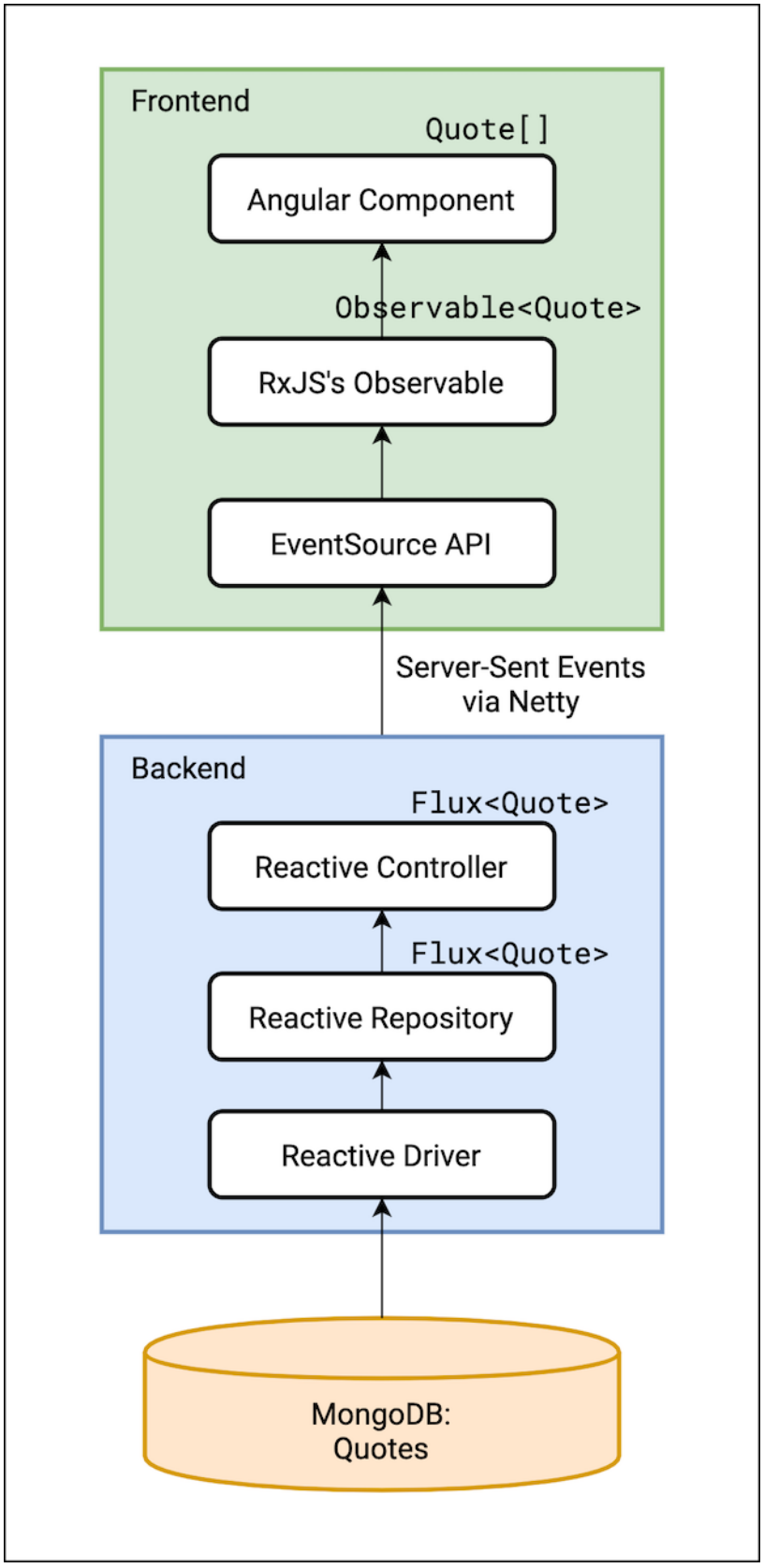
© 2018 - 2020 Moisés Macero

Contents

Chapter 1: The Reactive Web

This guide focuses on the capabilities of Spring WebFlux and Spring Boot 2 to create a **Reactive Web Application**, supported by a code example that you can build step by step.

To avoid dumb, non-realistic examples where Spring is also the client of the Reactive API, you will complete the stack with a client application in **Angular 9**. To make it reactive, you'll use Server-Sent Events (SSE) to communicate the backend with the frontend. See the figure below for a quick view of the stack we'll build.



Full Reactive Stack - Quick View

This guide has four main chapters and an appendix:

1. **The Reactive Web**. This chapter guides you through the main concepts of a reactive approach and compares it to the classic web interfaces.
2. **The backend**. Covers the creation of the Spring Boot application using the required dependencies and the development of the backend layers, paying special attention to the Controllers with WebFlux and the MongoDB reactive driver.
3. **The frontend**. Goes through the development of an Angular application that consumes the reactive endpoint created on the server's side.
4. **Conclusions**. This part completes the guide, summarizing what you've built and comparing the Reactive Web approach with the Classic Web approach. It gives some insights on performance, testability, etc.
5. **Run the app with Docker**. This appendix helps you build and run the full stack application in Docker.

All the source code (spring-boot, Angular, Docker) is available on GitHub: Full-Reactive Stack repository[1]. If you find it useful, please give it a star!

The goal of this Guide is that you can **learn Reactive Web development by building an application from scratch** (or navigating through the code if you prefer that). The chosen technologies are:

- Spring Boot 2.3.
- Spring WebFlux (also Spring Web to compare).
- Spring Data JPA Reactive Repositories
- Angular 9
- RxJS
- EventSource API.

Reactive Web Patterns

Reactive patterns and non-blocking programming techniques are widely extended technologies nowadays. However, when we focus on the Web layer and exposed APIs, there is still a wide majority of applications that use blocking strategies.

To try to clarify a bit better how this new Reactive Web approach differs from other ones, let's compare it with two existing techniques:

- The **Servlet 3.0** specification introduced asynchronous support. That means you can optimize the usage of container threads using `Callable` and Spring's `DeferredResult` as a response from controllers. However, from the client's perspective, **they're still blocking calls**. Besides, the imperative way of using the API is far from being developer-friendly, so they are seldom used.

[1] https://github.com/mechero/full-reactive-stack

- From the web client's point of view, you can rely on asynchronous patterns as well. This is the most popular way of performing a request to the server, using *promises*. Clients usually perform requests and wait for the response in different threads so the main flow doesn't block. The response arrives in the background and then a callback function processes the data and has the ability to, for example, update the HTML's DOM. Again, even being asynchronous, **the additional thread is also blocked**, waiting for a response that might take a long time to complete and receiving all the data at once.

In a Reactive Web approach, threads don't block until all the data is available. Instead, every layer is capable of providing data as soon as they process it. Therefore, web clients can start doing their part sooner, so we improve the user's experience. It's also more efficient since the different layers can process information in a continuous stream instead of being idle until a whole chunk of data comes.

WebFlux and Project Reactor

Spring 5 provides a non-blocking, reactive alternative to the standard Spring Web MVC approach: **Spring WebFlux**. It is based on the Project Reactor[2] (also part of the Spring family), which follows the Reactive Streams Specification[3]. That means WebFlux follows the standard concepts of Reactive Programming: *Publishers, Subscribers, Observables, etc.*

We'll cover WebFlux and Project Reactor in more detail in the next chapter.

Reactive Web: Advantages

So WebFlux implements a reactive, non-blocking API. That sounds pretty cool but, what does that mean exactly? Why should we care about non-blocking web calls?

The first reason is *slow Internet connections*. Having a slow connection is annoying, but it gets even worse with classic blocking HTTP calls. Let's say you're searching for the best mobile phones on your favorite online store app. You enter the query terms and press 'Search'. Your Internet connection is slow, so you need to wait for 15 seconds and stare at a spinner or a blank page. Then, the full first page with the results pops up, all at once. That happens because the web server doesn't care about your slow connection, so it's responding with a -let's say- 300 kilobytes response of full JSON with summaries, descriptions, and prices of the first 50 items that match your criteria. On your side -*the client's side*-, the problem is related to classic web client approaches that use blocking HTTP calls (even in different threads), that wait until the full HTTP response is received to process the data, and then render the page. A better alternative would be that the server, instead of returning the 50 items together, would send to the client the items one by one as soon as they're found. The client could then adapt its interface to react to these individual item pushes, and render them as they come. Note

[2] https://projectreactor.io/
[3] https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.1/README.md#specification

that switching only one of these two sides of the communication wouldn't fix this situation: both would need to use a different approach.

Something similar to the slow connection issue would happen if the server is busy. Let's imagine that many users are searching for products on that online store at the same time, and the database's capacity is not ready for that. The database becomes slower, and the queries take a few seconds to fetch 50 results. Now, the bottleneck in the client-server interaction is not the web interface but the database connection. However, the result is the same. The user has to wait until the query is fully completed to get the list of products. A better outcome could be achieved if the database connection, instead of returning query results at once (blocking), would open streams with the database clients (another backend layer) and return results as it's finding them.

With a reactive web interface, the user would see three items on the screen per second, instead of all at once after fifteen seconds. That's a first non-blocking advantage: faster availability of the data, which in our example means **better user experience**. We as developers should aim for that and move away from blocking calls. According to a study from Google[4], up to 53% of mobile users abandon a site if it takes more than 3 seconds to load.

In a full-reactive stack scenario, the database retrieves results as soon as they're available. Same for the Web Interface (let's say HTTP). The Business logic layer should be prepared to process items in a reactive way too, e.g. by using reactive libraries. Finally, to close the loop and benefit from a full-reactive stack, the client's side should be able to process the data as soon as it arrives, following a *subscriber pattern*.

There is an extra advantage that reactive web interfaces share with other asynchronous approaches: **a better thread usage**. In a non-blocking system, you don't have threads waiting to be completed. Instead, the server *parks* the client thread quickly so there are fewer server threads occupied. Web clients will be notified when new data becomes available, following a Publish-Subscribe pattern. This advantage is very relevant if you have a server application that may perform slowly sometimes, thus accumulating a lot of blocked threads and eventually not being able to process more.

Is Non-Blocking the same as Reactive?

It depends on whom you ask that question. 'Reactive' is a generic concept describing that you're acting triggered by a previous action. You could argue that programming techniques that follow a non-blocking approach are *reactive programming* styles because you normally use a callback function, that reacts to the result. In that broad scope of the 'reactive' word, using a `CompletableFuture`, or any other tool based on *promises* with callbacks, would be considered *reactive programming*. However, people normally refer to these libraries and APIs as *asynchronous* programming.

It's more common to talk about reactive programming when our code uses one of the existing frameworks that follow the Reactive Streams spec[5]. Some popular reactive frameworks are ReactiveX[6]

[4] https://www.soasta.com/blog/google-mobile-web-performance-study/
[5] https://github.com/reactive-streams/reactive-streams-jvm/blob/master/README.md
[6] http://reactivex.io/

(RxJS, RxJava, etc.), Akka[7], and Project Reactor[8] (the one used by WebFlux). Java 9 also introduced the reactive streams specification with the Flow API[9] so the implementors can choose the Java API standards for their libraries. However, that hasn't worked so far and most of the reactive libraries still use their custom classes. In any case, all these frameworks implement (with some slight differences in naming) patterns like *Observables*, *Publishers*, and *Subscribers*.

Additionally, these reactive frameworks adopt the **backpressure** concept. The main idea of backpressure is that the subscriber has the control of the stream, so the consumer can *signal the publisher to stop producing data* instead of having to accumulate it in buffers. Project Reactor implements backpressure, as we'll see and demonstrate later in this guide.

The application

To showcase the reactive capabilities, you'll create a client web application that receives quotes from the book *Don Quixote* (yes, you can tell I'm Spanish). Instead of just asking for the quotes using a standard blocking call, you'll open a Server-Sent Events channel[10]. If you know about *WebSockets*, you can see this as a similar technology where the communication is unidirectional (server to client). The backend application will send these events (quotes) once there is a subscriber (our client application). This part is supported by WebFlux returning a `Flux<Quote>` object (don't worry for now about the terms, we'll cover that). On the Angular app's end, you'll model the connection using an `EventSource` object, mapped to an RxJS's `Observable<Queue>`.

The Angular component subscribes to the `Observable`, adds the new element to an array, and re-renders the UI.

While we build the Full Reactive Stack, we'll create a classic blocking approach too. We'll use it to compare both strategies from the implementation and runtime perspectives.

We'll dive into the implementation details in the rest of the chapters of this guide.

[7] https://akka.io/
[8] https://projectreactor.io/
[9] https://community.oracle.com/docs/DOC-1006738
[10] https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events