FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

REACTIVE PROGRAMMING IN JAVA

eMag Issue 48 - Feb 2017

InfoQueue

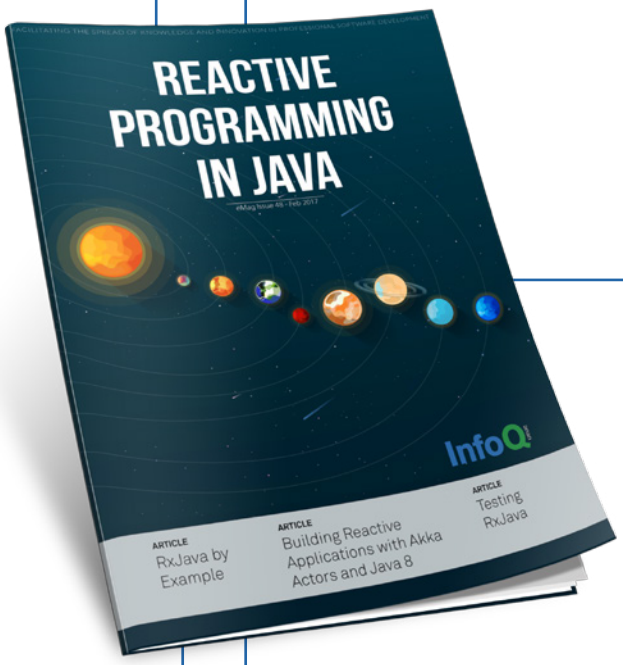
ARTICLE
RxJava by Example

ARTICLE
Building Reactive Applications with Akka Actors and Java 8

ARTICLE
Testing RxJava

RxJava by Example

In the ongoing evolution of paradigms for simplifying concurrency under load, the most promising addition is reactive programming, a specification that provides tools for handling asynchronous streams of data and for managing flow-control, making it easier to reason about overall program design.



Testing RxJava

You are ready to explore reactive opportunities in your code but you are wondering how to test out the reactive idiom in your codebase. In this article Java Champion Andres Almiray provides techniques and tools for testing RxJava.

Reactor by Example

Reactor, like RxJava 2, is a fourth generation reactive library launched by Spring custodian Pivotal. It builds on the Reactive Streams specification, Java 8, and the ReactiveX vocabulary. In this article, we draw a parallel between Reactor and RxJava, and showcase the common elements as well as the differences.

Refactoring to Reactive: Anatomy of a JDBC migration

Reactive programming offers built-in solutions for some of the most difficult challenges in programming, including concurrency management and flow control. So you might ask - how do I get there; can I introduce it in phases? In this article we transform a legacy application to a reactive model using RxJava.

Building Reactive Applications with Akka Actors and Java 8

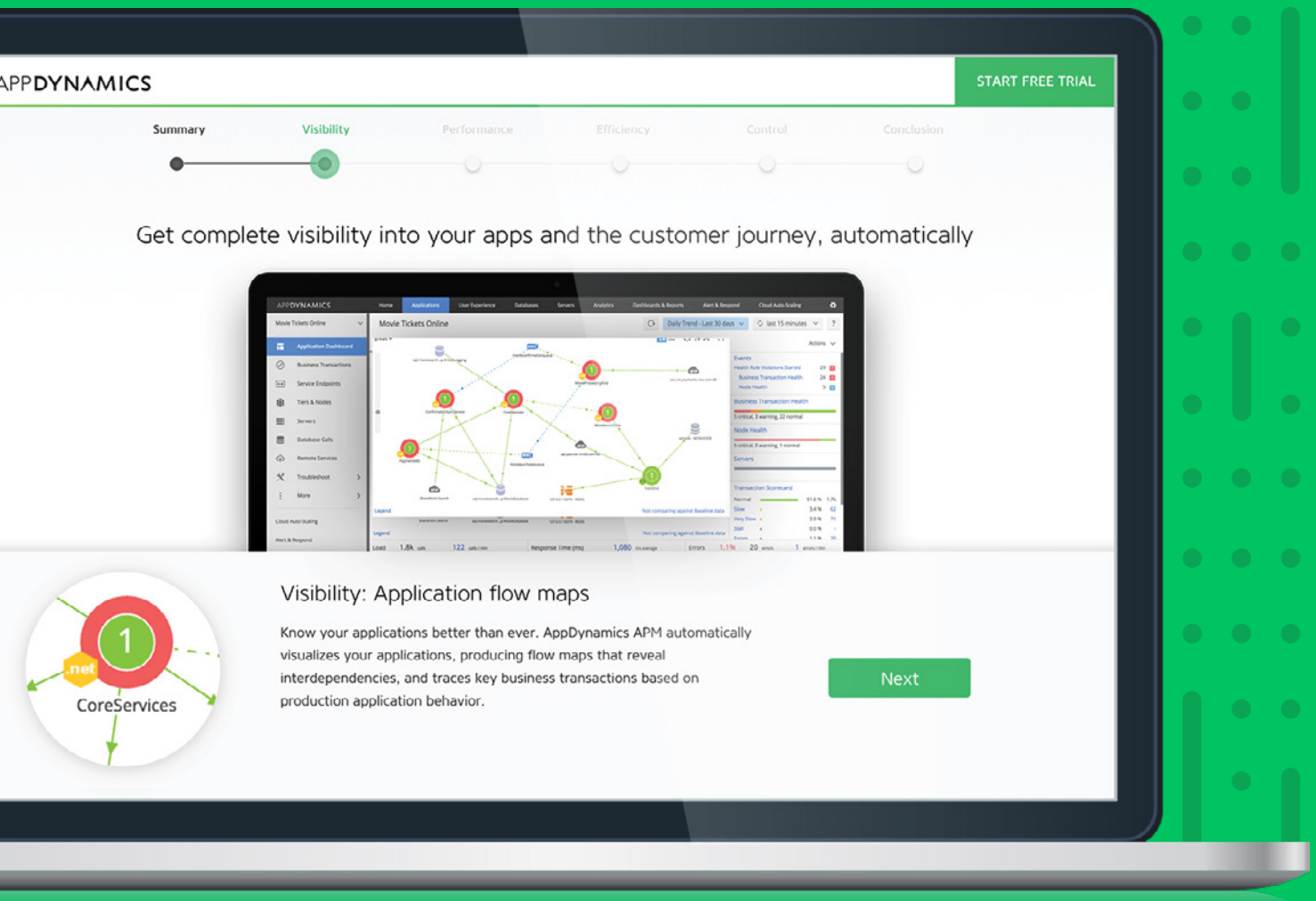
Akka and Java 8 make it possible to create distributed microservice-based systems that just a few years ago were the stuff of dreams. Actor based systems enable developers to create quickly evolving microservice architectures that can elastically scale systems to support huge volumes of data.

APPDYNAMICS

Get complete visibility into your apps and the customer journey, automatically.

Optimizing customer experience with end-to-end visibility and management of even the most complex and distributed applications with auto-generated flow maps, dynamic baselining, and code-level diagnostics.

Take a Tour at **AppDynamics.com**



VICTOR GRAZI

is the Java queue lead at InfoQ. Inducted as an Oracle Java Champion in 2012, Victor works in the financial industry on core platform tools, and as a technical consultant and Java evangelist. He is also a frequent presenter at technical conferences. Victor hosts the "Java Concurrent Animated" and "Bytecode Explorer" open source projects.

A LETTER FROM THE EDITOR

As resource consumption requirements increase and dynamic content generation takes on new dimensions in CPU utilization and data sizes, industry has responded with non-blocking concurrency in the form of reactive programming.

With reactive programming, previously error-prone operations like parking threads and communicating flow control between consumer and provider become trivial.

There is a learning curve, however, and as we travel the road to mastery, we will find ourselves groping for vestiges of traditional programming practices such as testing and refactoring.

For this Reactive Java emag, InfoQ has curated a series of articles to help developers hit the ground running with a comprehensive introduction to the fundamental reactive concepts, followed by a case study/strategy for migrating your project to reactive, some tips and tools for testing reactive, and practical applications using Akka actors.

RxJava by Example - Victor Grazi
We start with a gentle progression of examples designed to help climb the learning curve. The reactive specification provides tools for handling asynchronous streams of data and for managing flow control, making it easier to reason about overall program design.

We start with the basics and ramp up to flow control and backpressure.

Testing RxJava - Andres Almiray
We're ready to explore reactive opportunities in our code but wondering how to test the reactive idiom in our codebase. In this piece, Java Champion Andres Almiray provides techniques and tools for testing RxJava.

Reactor by Example - Simon Basle
Reactor is Pivotal's fourth-generation reactive library. It builds on the Reactive Streams spec, Java 8, and ReactiveX. In this piece, we showcase what Pivotal's Reactor framework brings to the reactive table.

Refactoring to Reactive: Anatomy of a JDBC migration - Nicolae Marasoiu
Reactive programming offers built-in solutions for some of the most difficult challenges in programming, including concurrency management and flow control. We might ask how we get there. Can we introduce it in phases? In this piece, we transform a legacy application to the reactive model using RxJava.

Building Reactive Applications with Akka Actors and Java 8
Akka and Java 8 make it possible to create distributed microservice-based systems that just a few years ago were the stuff of dreams. Actor-based systems allow developers to create quickly evolving microservice architectures that can elastically scale systems to support huge volumes of data.

Reactive programming provides the tools for programming highly scalable systems for microservices, Android applications, and other load and CPU-intensive applications. We hope this emag will help launch your reactive career!

RxJava by Example

by Victor Grazi

In the ongoing evolution of programming paradigms for simplifying concurrency under load, we have seen the adoption of java.util.concurrent, Akka streams, CompletableFuture, and frameworks like Netty. Most recently, reactive programming has been enjoying a burst of popularity thanks to its power and its robust tool set.

Reactive programming is a specification for dealing with asynchronous streams of data, providing tools for transforming and combining streams and for managing flow control, making it easier to reason about your overall program design.

But easy it is not, and there is definitely a learning curve. It reminds the mathematicians among us of the leap from learning standard algebra with its scalar quantities to linear algebra with its vectors, matrices, and tensors, essentially streams of data that are treated as a unit. Unlike objects in traditional programming, the fundamental unit of reactive reasoning is the stream of events. Events can come in the form of objects, data feeds, mouse movements, or even exceptions. The word "exception" expresses the traditional notion of an exceptional handling, as in "this is what is supposed to happen and here are the exceptions." In reactive, exceptions are first-class citizens, treated every bit as such. Since streams are generally asynchronous, it doesn't make sense to throw an exception, so any exception is instead passed as an event in the stream.

This article will consider the fundamentals of reactive program- ▶

KEY TAKEAWAYS

Reactive programming is a specification for dealing with asynchronous streams of data.

Reactive provides tools for transforming and combining streams and for managing flow control.

Marble diagrams provide an interactive canvas for visualizing reactive constructs.

An Observable resembles the Java Streams API but the resemblance is purely superficial.

Attach an Observable to hot streams to attenuate and process asynchronous data feeds.

ming, with a pedagogical eye on internalizing the important concepts.

The first thing to keep in mind is that in reactive everything is a stream. The `Observable` is the fundamental unit that wraps a stream. Streams can contain zero or more events, may or may not complete, and may or may not issue an error. Once a stream completes or issues an error, it is essentially done, although there are tools for retrying or substituting different streams when an exception occurs.

Before you try out some examples, include the RxJava dependencies in your codebase. You can load it from Maven using the dependency: (Code 1)

The `Observable` class has dozens of static factory methods and operators, each in a wide variety of flavors for generating new `Observables`, or for attaching them to processes of interest. Observables are immutable, so operators always produce a new `Observable`. To understand our code examples, let's review the basic `Observable` operators that we'll be using in the code samples later in this article.

`Observable.just` produces an `Observable` that emits a single generic instance, followed by a complete. For example: (Code 2)

That creates a new `Observable` that emits a single event before completing, the string "`Howdy!`"

We can assign that `Observable` to an `Observable` variable. (Code 3)

```
<dependency>
    <groupId>io.reactivex.rxjava2</groupId>
    <artifactId>rxjava</artifactId>
    <version>2.0.5</version>
</dependency>
```

Code 1

```
Observable.just("Howdy!")
```

Code 2

```
Observable<String> hello = Observable.just("Howdy!");
```

Code 3

```
Observable<String> howdy = Observable.just("Howdy!");
howdy.subscribe(System.out::println);
```

Code 4

```
Observable.just("Hello", "World")
          .subscribe(System.out::println);
```

Code 5

```
List<String> words = Arrays.asList(
 “the”,
 “quick”,
 “brown”,
 “fox”,
 “jumped”,
 “over”,
 “the”,
 “lazy”,
 “dog”
);

Observable.just(words)
          .subscribe(System.out::println);
```

Code 6

```
Observable.fromIterable(words)
          .subscribe(System.out::println);
```

Code 7

```
Observable.range(1, 5).subscribe
      (System.out::println);
outputs:
1
2
3
4
5
```

Code 8

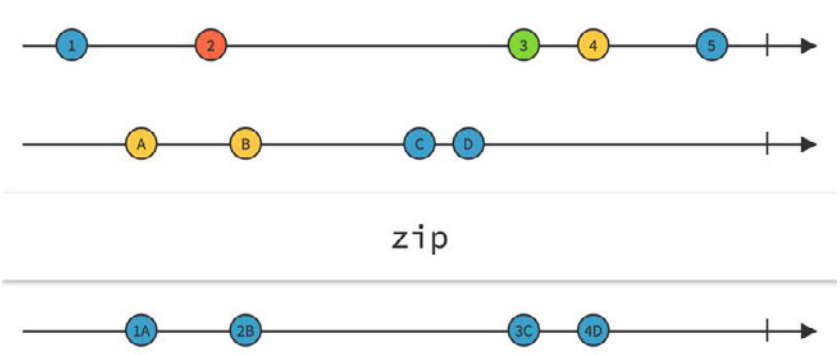


Figure 1

But that by itself won't get us very far, because just like the proverbial tree that falls in a forest, if nobody is around to hear it, it does not make a sound. An `Observable` must have a subscriber to do anything with the events it emits. Thankfully, Java now has lambdas, which allow us to express our observables in a concise declarative style: (Code 4)

That emits a gregarious “Howdy!”

Like all `Observable` methods, the `just` keyword is overloaded and so we can also say: (Code 5)

That puts out:

```
Hello
World
```

`just` is overloaded for up to 10 input parameters. Notice the output is on two separate lines, indicating two separate output events.

Let's try supplying a list and see what happens. (Code 6)

This outputs an abrupt:

```
[the, quick, brown, fox,
      jumped, over, the,
      lazy, dog]
```

We were expecting each word as a separate emission, but we got a single emission consisting of the whole list. To correct that, we invoke the more appropriate `fromIterable` method. (Code 7)

The method converts an iterable to a series of events, one per element. (Note that in RxJava 1 there was a single overloaded `from` method. This has been replaced with several flavors of `from` including `fromIterable` and `fromArray`.)

Executing that provides the more desirable multiline output:

```
the
quick
brown
fox
jumped
over
the
lazy
dog
```

It would be nice to get some numbering on that — again, a job for observables.

Before we code that, let's investigate two operators, `range` and `zip`. `range(i, n)` creates a stream of `n` numbers starting with `i`. (Code 8)

Our problem of adding numbering would be solved if we had a way to combine the range stream with our word stream.

RxMarbles is a great site for smoothing the reactive learning curve, in any language. The site features interactive JavaScript renderings for many of the re- ▶

```
Observable.fromIterable(words)
  .zipWith(Observable.range(1, Integer.MAX_VALUE),
     (string, count)->String.format("%2d. %s", count, string))
  .subscribe(System.out::println);
```

Code 9

```
Observable. fromIterable(words)
  .flatMap(word -> Observable.fromArray(word.split("")))
  .zipWith(Observable.range(1, Integer.MAX_VALUE),
    (string, count) -> String.format("%2d. %s", count, string))
  .subscribe(System.out::println);
```

Code 10

```
Observable.fromIterable(words)
  .flatMap(word -> Observable.fromArray(word.split("")))
  .distinct()
  .zipWith(Observable.range(1, Integer.MAX_VALUE),
    (string, count) -> String.format("%2d. %s", count, string))
  .subscribe(System.out::println);
```

Code 11

active operations. Each uses the common "marbles" reactive idiom to depict one or more source streams and the result stream produced by the operator. Time passes from left to right, and marbles on the timeline represents events. You can click and drag the source marbles to see how they affect the result.

A quick perusal reveals the `zip` operation — just what the doctor ordered. Let's look at the marble diagram to understand it better: (Figure 1)

`zip` combines the elements of the source stream with the elements of a supplied stream, using a pairwise "zip" transformation mapping that you can supply in the form of a lambda. When either of those streams completes, the zipped stream completes, so any remaining events from the other stream would be lost. `zip` accepts up to nine source streams and `zip` operations. There is a corresponding `zipWith` operator that zips a provided stream with the existing stream.

Coming back to our example, we can use range and `zipWith` to prepend our line numbers, using `String.format` as our `zip` transformation. (Code 9)

Which outputs:

```
1. the
2. quick
3. brown
4. fox
5. jumped
6. over
7. the
8. lazy
9. dog
```

Looking good!

Notice that the `zip` and `zipWith` operators stop pulling from all of the streams once any of the streams has completed. That is why we were not intimidated by the `Integer.MAX_VALUE` upper limit.

Now, let's say we want to list not the words but the letters composing those words. This is a job for `flatMap`, which takes the emissions (objects, collections, or arrays) from an `Observable` and maps those elements to individual `Observables`, then flattens the emissions from all of those into a single `Observable`.

For our example, we will use `split` to transform each word into an array of its component characters. We will then `flatMap` those to create a new `Observable` that consists of all of the characters of all of the words. (Code 10)

The output of that is:

```
1. t          ...
2. h         30. l
3. e         31. a
4. q         32. z
5. u         33. y
6. i         34. d
7. c         35. o
8. k         36. g
```

All words are present and accounted for, but there's too much data. We only want the distinct letters so we do this: (Code 11)

That produces:

```
 1. t          14. f
 2. h          15. x
 3. e          16. j
 4. q          17. m
 5. u          18. p
 6. i          19. d
 7. c          20. v
 8. k          21. l
 9. b          22. a
10. r          23. z
11. o          24. y
12. w          25. g
13. n
```

```
.flatMap(word -> Observable.fromIterable(word.split("")))
 .distinct()
 .sorted()
 .zipWith(Observable.range(1, Integer.MAX_VALUE),
   (string, count) -> String.format("%2d. %s", count, string))
 .subscribe(System.out::println);
```

Code 12

```
List<String> words = Arrays.asList(
 "the",
 "quick",
 "brown",
 "fox",
 "jumps",
 "over",
 "the",
 "lazy",
 "dog"
);

Observable.fromIterable(words)
 .flatMap(word -> Observable.fromArray(word.split("")))
 .distinct()
 .sorted()
 .zipWith(Observable.range(1, Integer.MAX_VALUE),
   (string, count) -> String.format("%2d. %s", count, string))
 .subscribe(System.out::println);
```

Code 13

As a child, I was taught that the "quick brown fox" phrase contains every letter in the English alphabet but we see that there are only 25 letters here, not the full 26. Let's sort them to locate the missing one. (Code 12)

That produces:

```
 1. a
 2. b
 3. c
 ...
17. q
18. r
19. t
20. u
21. v
22. w
23. x
24. y
25. z
```

Looks like S is missing. Correcting the "quick brown fox" phrase to the actual one (replacing "jumped" with "jumps") produces the expected output. (Code 13)

That yields:

```
 1. a
 2. b
 3. c
 4. d
 5. e
 6. f
 7. g
 8. h
 9. i
10. j
11. k
12. l
13. m
14. n
15. o
16. p
17. q
18. r
19. s
20. t
21. u
22. v
23. w
24. x
25. y
26. z
```

That's a lot better!

So far, this all looks similar to the Java Streams API introduced in Java 8. The resemblance is strictly coincidental, because reactive adds so much more.

Java Streams and lambda expressions were valuable language additions but they are, in essence, nothing more than ways to iterate collections and produce new collections. They are finite, static, and do not provide for reuse. Even when forked by the Streams `parallel` operator, they go off and do their own fork and join, and only return when done, leaving the program with little control. Reactive, in contrast, introduces the concepts of timing, throttling, and flow control, and they can attach to "infinite" processes that conceivably never end. The output is not a collection but is available for us to deal with, however we require.

Let's take a look at some more marble diagrams to get a better picture.

The `merge` operator merges up to nine source streams into the final output, preserving order. There is no need to worry about race con- ▶

ditions; all events are flattened onto a single thread, including any exception and completion events.

The debounce operator treats all events within a specified time delay as a single event, emitting only the last in each such series: (Figure 2)

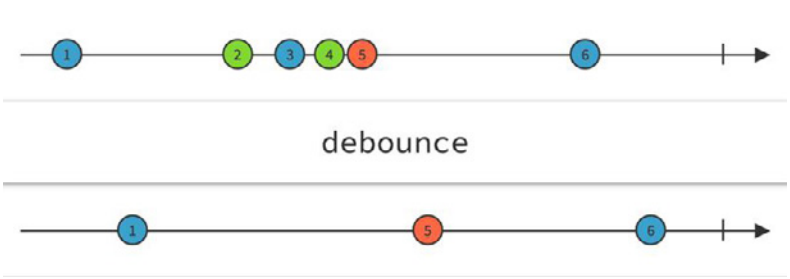


Figure 2

The difference in time between the top 1 and the bottom 1 is the time delay. In the 2, 3, 4, 5 pairs, each element comes more quickly than the time delay from the previous, so they are considered one and debounced away. If we move the top 5 a little bit to the right so that it falls outside that delay window, it starts a new `debounce` window: (Figure 3)

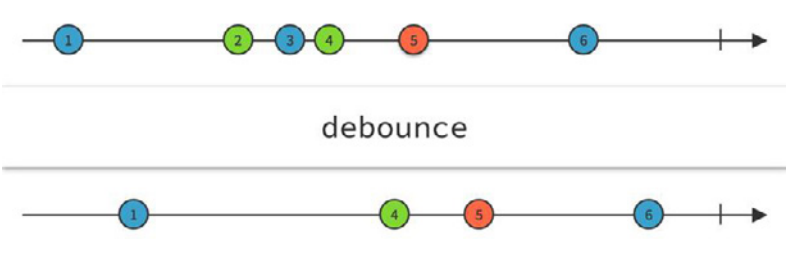


Figure 3

One interesting operator is the dubiously named "ambiguous" operator `amb` and its array incarnation `ambArray`.

`amb` is a conditional operator that selects the first stream to emit from among all its input streams, and sticks with that stream, ignoring all the others. In the following, the second stream is the first to pump, so the result selects that stream and stays with it. (Figure 4)

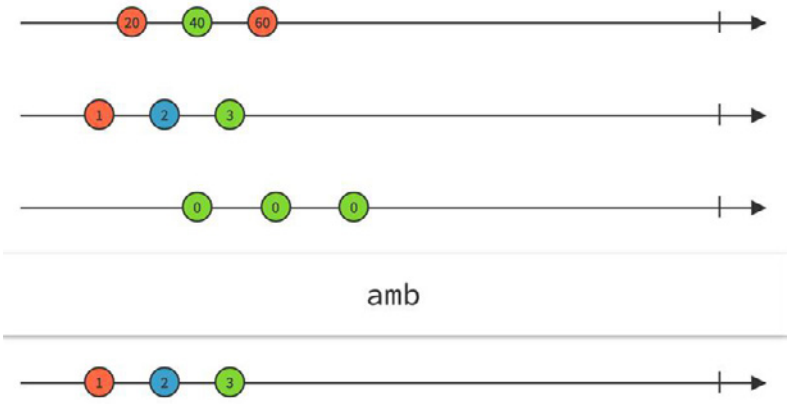


Figure 4

Sliding the 20 in the first stream over to the left makes the top stream the earliest producer, thereby altering the output: (Figure 5)

This is useful, for example, if we have a process that needs to attach to a feed, perhaps reaching to several message topics or, say, to Bloomberg and Reuters, and we don't care which — we just need to get the earliest one and stay with it.

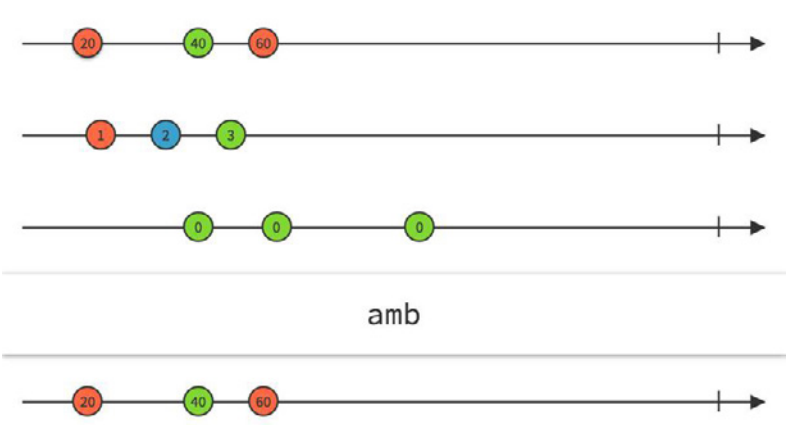


Figure 5

Tick tock

Now we have the tools to combine timed streams to produce a meaningful hybrid. In the next example, we consider a feed that pumps every second during the week but, to save CPU cycles, only pumps every three seconds during the weekend. We can

```
private static boolean isSlowTickTime() {
 return LocalDate.now().getDayOfWeek() == DayOfWeek.SATURDAY ||
        LocalDate.now().getDayOfWeek() == DayOfWeek.SUNDAY;
}
```

Code 14

```
private static long start = System.currentTimeMillis();
public static Boolean isSlowTickTime() {
   return (System.currentTimeMillis() - start) % 30_000 >= 15_000;
}
```

Code 15

```
Observable<Long> fast = Observable.interval(1, TimeUnit.SECONDS);
Observable<Long> slow = Observable.interval(3, TimeUnit.SECONDS);
```

Code 16

```
Observable<Long> clock = Observable.merge(
       slow.filter(tick-> isSlowTickTime()),
       fast.filter(tick-> !isSlowTickTime())
);
```

Code 17

```
clock.subscribe(tick-> System.out.println(new Date()));
```

Code 18

use that hybrid "metronome" to produce market-data ticks at the desired rate.

First let's create a boolean method that checks the current time and returns "true" for weekends and "false" for weekdays. (Code 14)

Those readers following along in an IDE who do not want to wait until next weekend/week to see this work may substitute the following implementation, which ticks fast for 15 seconds and then slow for 15 seconds. (Code 15)

Let's create two `Observables`, `fast` and `slow`, and then apply filtering to schedule and merge them.

We will use the `Observable.interval` operation, which generates a tick every specified number of time units (counting sequential `Longs` beginning with `0`). (Code 16)

`fast` will emit an event every second, `slow` will emit every three seconds. (We will ignore the `Long` value of the event, as we are only interested in the timings.)

Now we can produce our syncopated clock by merging those two `Observables` and applying a filter to each that tells the `fast` stream to tick on the weekdays (or for 15 seconds) and the `slow` one to tick on the weekends (or alternate 15 seconds). (Code 17)

Finally, let's add a subscription to print the time. Launching this will print the system date and time according to our required schedule. (Code 18) ▶

We will also need a keep alive to prevent this from exiting, so add the following to the end of the method (and handle the InterruptedException).

```
Thread.sleep(60_000)
```

Running that produces this:

```
Fri Sep 16 03:08:18 BST 2016
Fri Sep 16 03:08:19 BST 2016
Fri Sep 16 03:08:20 BST 2016
Fri Sep 16 03:08:21 BST 2016
Fri Sep 16 03:08:22 BST 2016
Fri Sep 16 03:08:23 BST 2016
Fri Sep 16 03:08:24 BST 2016
Fri Sep 16 03:08:25 BST 2016
Fri Sep 16 03:08:26 BST 2016
Fri Sep 16 03:08:27 BST 2016
Fri Sep 16 03:08:28 BST 2016
Fri Sep 16 03:08:29 BST 2016
Fri Sep 16 03:08:30 BST 2016
Fri Sep 16 03:08:31 BST 2016
Fri Sep 16 03:08:32 BST 2016
Fri Sep 16 03:08:35 BST 2016
Fri Sep 16 03:08:38 BST 2016
Fri Sep 16 03:08:41 BST 2016
Fri Sep 16 03:08:44 BST 2016
          . . .
```

We can see that the first 15 ticks are a second apart, followed by 15 seconds of ticks that are three seconds apart, in alternation as required.

Attaching to an existing feed

This is all very useful for creating Observables from scratch to pump static data. But how do we attach an Observable to an existing feed so we can take advantage of the reactive flow control and stream manipulation?

RxJava 2 introduced some new classes we should become acquainted with before proceeding.

Cold and hot Observables and Flowables

In previous RxJava versions, Observable was equipped with flow-control methods, even for small streams where it would be irrelevant. To conform to the reactive specification, RxJava 2 removes flow control from the Observable class and introduces the Flowable class, essentially an Observable that provides flow control.

So far, we have discussed cold Observables: they provide static data, although we may still regulate timing. The distinguishing qualities of cold Observables is that they only pump when there is a subscriber, and all subscribers receive the exact set of historical data, regardless when they subscribe. Hot Observables, in contrast, pump regardless of the number of subscribers and generally pump just the latest data to all subscribers (unless some caching strategy is applied). We can convert cold Observables to hot by performing both of the following steps:

1. Call the Observable's publish method to produce a new ConnectableObservable.
2. Call the ConnectableObservable's connect method to start pumping.

This works but does not support flow control. In general, we would prefer to connect to existing long-running feeds with a Flowable, which uses a parallel syntax to an Observable but also provides backpressure controls:

1. Call the Flowable's publish method to produce a new ConnectableFlowable.
2. Call the ConnectableFlowable's connect method to start pumping.

To attach to an existing feed, we could (if we felt so inclined) add a listener to our feed that propagates ticks to subscribers by calling their onNext method on each tick. Our implementation would need to ensure that subscribers are still subscribed or stop pumping to them, and would need to respect backpressure semantics. Thankfully, Flowable's create method performs all that work automatically. For our example, let's assume we have a SomeFeed market-data service that issues price ticks and a SomeListener method that listens for those price ticks as well as lifecycle events. There is an implementation of these on GitHub if you'd like to try it at home.

Our feed accepts a listener, which supports the following API. (Code 19)

Our PriceTick has accessors for date, instrument, and price, and a method for signaling the last tick:

PriceTick
date instrument price
isLast

Let's look at an example that connects an Observable to a live feed using a Flowable. (Code 20)

```
public void priceTick(PriceTick event);
public void error(Throwable throwable);
```

Code 19

```
1    SomeFeed<PriceTick> feed = new SomeFeed<>();
2    Flowable<PriceTick> flowable = Flowable.create(emitter -> {
3        SomeListener listener = new SomeListener() {
4            @Override
5            public void priceTick(PriceTick event) {
6                emitter.onNext(event);
7                if (event.isLast()) {
8                    emitter.onComplete();
9                }
10           }
11
12           @Override
13           public void error(Throwable e) {
14               emitter.onError(e);
15           }
16       };
17       feed.register(listener);
18   }, BackpressureStrategy.BUFFER);
19   flowable.subscribe(System.out::println);
20
```

Code 20

```
21       ConnectableFlowable<PriceTick> hotObservable = flowable.publish();
22       hotObservable.connect();
```

Code 21

```
23       hotObservable.subscribe((priceTick) ->
24         System.out.printf("%s %4s %6.2f%n", priceTick.getDate(),
25           priceTick.getInstrument(), priceTick.getPrice()));
```

Code 22

This is taken almost verbatim from the Flowable Javadoc. The `Flowable` wraps the steps of creating a listener (line 3) and registering to the service (line 17). Subscribers are automatically attached by the `Flowable`. The events generated by the service are delegated to the listener (line 6). Line 18 tells the observer to buffer all notifications until a subscriber consumes them. Other backpressure choices are:

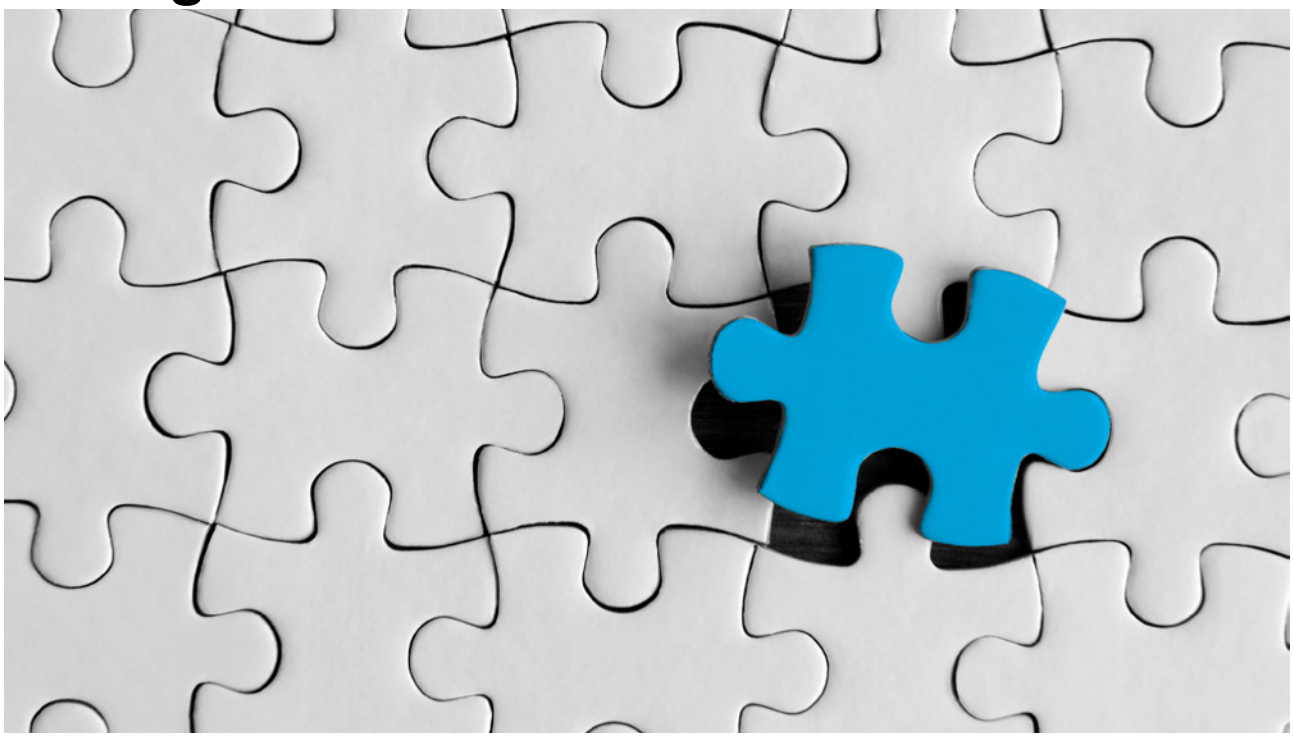
- `BackpressureStrategy.MISSING` applies no backpressure. If the stream can't keep up, it may throw a `MissingBackpressureException` or `IllegalStateException`.
- `BackpressureStrategy.ERROR` emits a `MissingBackpressureException` if the downstream can't keep up.
- `BackpressureStrategy.DROP` drops the incoming `onNext` value if the downstream can't keep up.
- `BackpressureStrategy.LATEST` keeps the latest `onNext` value and overwrites it with newer ones until the downstream can consume it.

All of this produces a cold `Flowable`. As with any cold observable, it would produce no ticks until the first observer subscribes and all subscribers would receive the same set of historical feeds, which is probably not what we want.

To convert this to a hot observable so that all subscribers receive all notifications as they occur in real time, we must call `publish` and `connect`, as described earlier. (Code 21)

Finally, we can subscribe and display our price ticks. (Code 22) ■

Testing RxJava



Andres Almiray is a Java/Groovy developer and Java Champion with more than 17 years of experience in software design and development. He has been involved in web and desktop application development since the early days of Java. He is a true believer in open source and has participated in popular projects like Groovy, JMatter, Asciidoctor, and more. He is a founding member and current project lead of the Griffon framework and is spec lead for JSR 377.

You've read about RxJava; you've played with the samples on the internet, for example in RxJava by Example, and now you have made a commitment to explore reactive opportunities in your own code. But now you are wondering how to test out the new capabilities that you might find in your codebase.

Reactive programming requires a shift in how to reason about a particular problem, because we need to focus not on individual data items but on data flowing as streams of events. These events are often produced and consumed from different threads, and so when writing tests we must keep aware of concurrency concerns. Fortunately RxJava2 provides built-in support for testing Observables and Subscriptions, built right into the core rxjava dependency.

First Steps

Let's revisit the words example from the last article and explore how we can test it out. Let's begin by setting up the base test harness using JUnit as our testing framework: (Code 1)

Our first test will follow a naive approach, given the fact that subscriptions will, by default, run on the calling thread, if no particular Scheduler is supplied. This means we can setup a Subscription and assert its state immediately after the subscription takes place: (Code 2)

Notice that we used an explicit List<String> to accumulate our results, along with a real subscriber. Given the simple nature of this test you may think that

KEY TAKEAWAYS

RxJava includes built-in, test-friendly solutions.

Use TestSubscriber to verify Observables.

Use TestSubscriber to have strict control of time.

The Awaitility library provides additional control of test context.

using an explicit accumulator as such is good enough, but remember that production grade observables may encapsulate errors or produce unexpected events; the simple subscriber plus accumulator combo is not sufficient to cover those cases. But don't fret, RxJava provides a TestObserver type that can be used in such cases. Let's refactor the previous test using this type (Code 3)

TestObserver replaces the custom accumulator, but it also provides some additional behavior. For example it's capable of telling us how many events were received and the data related to each event. It can also assert that the subscription was successfully completed and that no errors appeared during the consumption of the Observable. The current Observable under test does not produce any errors, but as we learned back in RxJava by Example Observables treat exceptions exactly the same as data events. We can simulate an error by concatenating an exception event in the following way (Code 4)

All is well in our limited use case. But actual production code can vary greatly, so let's consider some more complex production cases.

Custom Schedulers

Quite often you'll find cases in production code where Observables are executed on a specific thread, or "scheduler" in Rx par- ▶

```
import io.reactivex.Observable;
import io.reactivex.observers.TestObserver;
import io.reactivex.plugins.RxJavaPlugins;
import io.reactivex.schedulers.Schedulers;
import org.junit.Test;
import java.util.*;
import static java.util.concurrent.TimeUnit.
     SECONDS;
import static org.awaitility.Awaitility.await;
import static org.junit.Assert.assertThat;
import static org.hamcrest.Matchers.*;


public class RxJavaTest {
    private static final List<String> WORDS =
            Arrays.asList(
          "the",
       "quick",
       "brown",
       "fox",
       "jumped",
       "over",
       "the",
       "lazy",
       "dog"
    );
}
```

Code 1

```
@Test
public void testInSameThread() {
   // given:
   List<String> results = new ArrayList<>();
   Observable<String> observable = Observable.
   fromIterable(WORDS)
       .zipWith(Observable.range(1, Integer.MAX_VALUE),
           (string, index) -> String.format("%2d. %s",
   index, string));

   // when:
   observable.subscribe(results::add);

   // then:
   assertThat(results, notNullValue());
   assertThat(results, hasSize(9));
   assertThat(results, hasItem(" 4. fox"));
}
```

Code 2

```
@Test
public void testUsingTestObserver() {
    // given:
    TestObserver<String> observer = new TestObserver<>();
    Observable<String> observable = Observable.fromIterable(WORDS)
        .zipWith(Observable.range(1, Integer.MAX_VALUE),
            (string, index) -> String.format("%2d. %s", index, string));

    // when:
    observable.subscribe(observer);

    // then:
    observer.assertComplete();
    observer.assertNoErrors();
    observer.assertValueCount(9);
    assertThat(observer.values(), hasItem(" 4. fox"));
}
```

Code 3

```
@Test
public void testFailure() {
    // given:
    TestObserver<String> observer = new TestObserver<>();
    Exception exception = new RuntimeException("boom!");

    Observable<String> observable = Observable.fromIterable(WORDS)
        .zipWith(Observable.range(1, Integer.MAX_VALUE),
            (string, index) -> String.format("%2d. %s", index, string))
        .concatWith(Observable.error(exception));

    // when:
    observable.subscribe(observer);

    // then:
    observer.assertError(exception);
    observer.assertNotComplete();
}
```

Code 4

```
@Test
public void testUsingComputationScheduler() {
    // given:
    TestObserver<String> observer = new TestObserver<>();
    Observable<String> observable = Observable.fromIterable(WORDS)
        .zipWith(Observable.range(1, Integer.MAX_VALUE),
            (string, index) -> String.format("%2d. %s", index, string));

    // when:
    observable.subscribeOn(Schedulers.computation())
        .subscribe(observer);

    await().timeout(2, SECONDS)
        .until(observer::valueCount, equalTo(9));

    // then:
    observer.assertComplete();
    observer.assertNoErrors();
    assertThat(observer.values(), hasItem(" 4. fox"));
}
```

Code 5

lance. Many Observable operations take an optional Scheduler parameter as an additional argument. RxJava defines a set of named Schedulers that can be used at any time. Some of these are io and computation, (which are shared threads) and newThread. You can also supply your own custom Scheduler implementation. Let's change the observable code by specifying the computation Scheduler. (Code 5)

You'll quickly discover there's something wrong with this test once you run it. The subscriber performs its assertions on the test thread however the Observable produces values on a background thread (the computation thread). This means the subscriber's assertions may be executed before the Observable has produced all relevant events, resulting in a failing test.

There are a few strategies we can choose to turn the test green:

- Turn the Observable into a blocking Observable.
- Force the test to wait until a certain condition is met.
- Switch the computation scheduler for an immediate one.

We'll cover each strategy starting with the one that requires less effort: turning the Observable into a blocking Observable. This technique works regardless of the Scheduler in use. The assumption is that data is produced in a background thread, causing subscribers to be notified in that same background thread.

What we'd like to do is force all events to be produced and the Observable to complete before the next statement in the test is evaluated. This is done by calling blockingIterable() on the Observable itself. (Code 6)

While this approach may be acceptable for the trivial code we have shown, it may not be practical for actual production code. What if the producer takes a long time to create all the data? This will make the test a slow one, increasing build times. There may be other timing issues as well. Fortunately TestObserver provides a set of methods that can force the test to wait for the termination event. Here's how it can be done: (Code 7)

If that were not enough I'd now like to point your attention to a handy library named Awaitility. Simply put, Awaitility is a DSL that allows you to express expectations about an asynchronous system in a concise and easy to read manner. You can include the Awaitility dependency using Maven: (Code 8)

Or using Gradle:

```
testCompile 'org.
      awaitility:
      awaitility:2.0.0'
```

The entry point of the Awaitility DSL is the org.awaitility.Awaitility.await() method see lines 13-14 in the example below). From there you can define conditions that must be met in order to let the test continue. You may decorate conditions with timeouts and other temporal constraints, for example minimum, maximum or duration range. Revisiting the previous test with Awaitility in tow results in the following code (Code 9)

This version does not change the nature of the Observable in any way, which allows you to test unaltered production code without any modification. This version of the test awaits at most 2 seconds for the Observable to performs its job by checking the state of the subscriber. If everything goes well, the subscriber's state checks out to 9 events before the 2 second timeout elapses.

Awaitility plays nicely with Hamcrest matchers, Java 8 lambdas, and method references, thus resulting in concise and readable conditions. There are also ready made extensions for popular JVM languages such as Groovy and Scala.

The final strategy we'll cover makes use of the extension mechanism that RxJava exposes as part of its API. RxJava defines a series of extension points that enable you to tweak almost every aspect of its default behavior. This extension mechanism effectively allows us to supply tailor made values for a particular RxJava feature. We'll take advantage of this mechanism to let our test inject a specific Scheduler regardless of the one specified by the production code. The behavior we're looking for is encapsulated in the RxJavaPlugins class. Assuming our production code relies on the computation() scheduler we're going to override its default value, returning a Scheduler that makes event processing happen in the same thread as the caller code; this is the Schedulers.trampoline() scheduler. Here's how the test looks now: (Code 10)

The production code is unaware that the computation() scheduler is an immediate one during testing. Please take note that you must reset the hook otherwise the immediate scheduler setting may leak, resulting in broken tests all over the place. The usage of a try/finally block obscures the intention of the test code a bit, but fortunately we can refactor out this behavior using a JUnit rule, making the test slimmer and more readable as a result. Here's one possible implementation for such a rule (Code 11) ▶

```
@Test
public void testUsingBlockingCall() {
    // given:
    Observable<String> observable = Observable.fromIterable(WORDS)
        .zipWith(Observable.range(1, Integer.MAX_VALUE),
            (string, index) -> String.format("%2d. %s", index, string));

    // when:
    Iterable<String> results = observable
        .subscribeOn(Schedulers.computation())
        .blockingIterable();

    // then:
    assertThat(results, notNullValue());
    assertThat(results, iterableWithSize(9));
    assertThat(results, hasItem(" 4. fox"));
}
```

Code 6

```
@Test
public void testUsingComputationScheduler() {
      // given:
      TestObserver<String> observer = new TestObserver<>();
      Observable<String> observable = Observable.fromIterable(WORDS)
            .zipWith(Observable.range(1, Integer.MAX_VALUE),
            (string, index) -> String.format("%2d. %s", index, string));

      // when:
      observable.subscribeOn(Schedulers.computation())
            .subscribe(observer);

      observer.awaitTerminalEvent(2, SECONDS);

      // then:
      observer.assertComplete();
      observer.assertNoErrors();
      assertThat(observer.values(), hasItem(" 4. fox"));
}
```

Code 7

```
<dependency>
    <groupId>org.awaitility</groupId>
    <artifactId>awaitility</artifactId>
    <version>2.0.0</version>
    <scope>test</scope>
</dependency>
```

Code 8

We override two other scheduler producing methods for good measure, making this rule more generic for other testing purposes down the road. Usage of this rule in a new testcase class is straightforward, we simply declare a field with the new type annotated with @Rule, like so (Code 12)

In the end we get the same behavior as before but with less clutter. Let's take a moment to look back what we have accomplished so far:

- Subscribers process data in the same thread as long as there's no specific Scheduler in use. This means we can make assertions on a subscriber right after it subscribes to an Observable.
- TestObserver can accumulate events and provide additional assertions on its state.
- Any Observable can be turned into a blocking Observable, thus enabling us to synchronously wait for events to be produced, regardless of the Scheduler used by the observable.
- RxJava exposes an extension mechanism that enables developers to override its defaults, and inject those right into the production code.

```
@Test
public void testUsingComputationScheduler_awaitility() {
    // given:
    TestObserver<String> observer = new TestObserver<>();
    Observable<String> observable = Observable.fromIterable(WORDS)
       .zipWith(Observable.range(1, Integer.MAX_VALUE),
            (string, index) -> String.format("%2d. %s", index, string));

 // when:
     observable.subscribeOn(Schedulers.computation())
         .subscribe(observer);

     await().timeout(2, SECONDS)
         .until(observer::valueCount, equalTo(9));

     // then:
     observer.assertComplete();
     observer.assertNoErrors();
     assertThat(observer.values(), hasItem(" 4. fox"));
 }
```

Code 9

```
@Test
public void testUsingRxJavaPluginsWithImmediateScheduler() {
    // given:
    RxJavaPlugins.setComputationSchedulerHandler(scheduler -> Schedulers.
            trampoline());
    TestObserver<String> observer = new TestObserver<>();
    Observable<String> observable = Observable.fromIterable(WORDS)
        .zipWith(Observable.range(1, Integer.MAX_VALUE),
          (string, index) -> String.format("%2d. %s", index, string));
     try {

        // when:
         observable.subscribeOn(Schedulers.computation())
            .subscribe(observer);

         // then:
         observer.assertComplete();
         observer.assertNoErrors();
         observer.assertValueCount(9);
         assertThat(observer.values(), hasItem(" 4. fox"));
     } finally {
         RxJavaPlugins.reset();
     }
}
```

Code 10

- Awaitility can be used to test out concurrent code using a DSL.

Each one of these techniques comes in handy in different scenarios, however all them are connected by a common thread (pun intended): the test code waits for the Observable to complete before making assertions on the subscriber's state. What if there was a way to inspect the Observable's behavior as it produces the data? In other words, what if it were possible to programmatically debug the Observable in place? We'll see a technique for doing that next.

Playing with Time

So far we've tested observables and subscriptions in a black box manner. Now we'll have a look at another technique that allows us to manipulate time in such a way that we can pop the hood and look at a subscriber's state while the Observable is still active, in other words, we'll use a white box testing technique. Once again, it's RxJava to the rescue, with its TestScheduler class. This particular Scheduler enables you to specify exactly how time passes inside of it. You can for example advance time by half a second, or make it leap 5 seconds. We'll start by creating an instance of this new Scheduler and then pass it to the test code (Code 13) ▶

```
private static class ImmediateSchedulersRule implements TestRule {
    @Override
    public Statement apply(final Statement base, Description description) {
        return new Statement() {
            @Override
            public void evaluate() throws Throwable {
                RxJavaPlugins.setIoSchedulerHandler(scheduler ->
                    Schedulers.trampoline());
                RxJavaPlugins.setComputationSchedulerHandler(scheduler ->
                    Schedulers.trampoline());
                RxJavaPlugins.setNewThreadSchedulerHandler(scheduler ->
                    Schedulers.trampoline());

                try {
                    base.evaluate();
                } finally {
                    RxJavaPlugins.reset();
                }
            }
        };
    }
}
```

Code 11

```
@Rule
public final ImmediateSchedulersRule schedulers = new ImmediateSchedulersRule();

@Test
public void testUsingImmediateSchedulersRule() {
    // given:
    TestObserver<String> observer = new TestObserver<>();
    Observable<String> observable = Observable.fromIterable(WORDS)
        .zipWith(Observable.range(1, Integer.MAX_VALUE),
            (string, index) -> String.format("%2d. %s", index, string));

    // when:
    observable.subscribeOn(Schedulers.computation())
        .subscribe(observer);

    // then:
    observer.assertComplete();
    observer.assertNoErrors();
    observer.assertValueCount(9);
    assertThat(observer.values(), hasItem(" 4. fox"));
}
```

Code 12

The "production" code changed a little, as we're now using an interval that is tied to the Scheduler to produce the numbering (line 6) instead of a range. This has the side-effect of producing numbers starting with 0 instead of the original 1. Once the Observable and the test scheduler are configured we immediately assert that the subscriber has no values (line 16), and has not completed or generated any errors (line 17). This is a sanity check as the scheduler has not moved at this point and so no values should have been produced by the observable nor received by the subscriber.

Next we move time by one whole second (line 20), this should cause the Observable to produce the first value, and that's exactly what the next set of assertions checks (lines 23-25).

We next advance time to 9 seconds from now. Mind you that this means moving to exactly 9 seconds from the scheduler's start, (and not advancing 9 seconds after already advancing 1 before, which would result in the scheduler looking at 10 seconds after it's start). In other words, advanceTimeBy() moves the scheduler's time relative to its current position, whereas advanceTimeTo()moves the scheduler's time in an absolute fash- ▶

```
@Test
public void testUsingTestScheduler() {
    // given:
    TestScheduler scheduler = new TestScheduler();
    TestObserver<String> observer = new TestObserver<>();
    Observable<Long> tick = Observable.interval(1, SECONDS, scheduler);

    Observable<String> observable = Observable.fromIterable(WORDS)
        .zipWith(tick,
            (string, index) -> String.format("%2d. %s", index, string));

    observable.subscribeOn(scheduler)
        .subscribe(observer);

    // expect:
    observer.assertNoValues();
    observer.assertNotComplete();

    // when:
    scheduler.advanceTimeBy(1, SECONDS);

    // then:
    observer.assertNoErrors();
    observer.assertValueCount(1);
    observer.assertValues(" 0. the");

    // when:
    scheduler.advanceTimeTo(9, SECONDS);
    observer.assertComplete();
    observer.assertNoErrors();
    observer.assertValueCount(9);
}
```

Code 13

```
private static class TestSchedulerRule implements TestRule {
   private final TestScheduler testScheduler = new TestScheduler();

   public TestScheduler getTestScheduler() {
       return testScheduler;
   }

   @Override
   public Statement apply(final Statement base, Description description) {
       return new Statement() {
           @Override
           public void evaluate() throws Throwable {
               RxJavaPlugins.setIoSchedulerHandler(scheduler -> testScheduler);
               RxJavaPlugins.setComputationSchedulerHandler(scheduler ->
                   testScheduler);
               RxJavaPlugins.setNewThreadSchedulerHandler(scheduler ->
                   testScheduler);

               try {
                   base.evaluate();
               } finally {
                   RxJavaPlugins.reset();
               }
           }
       };
   }
}
```

Code 14

```
@Rule
public final TestSchedulerRule testSchedulerRule = new TestSchedulerRule();

@Test
public void testUsingTestSchedulersRule() {
    // given:
    TestObserver<String> observer = new TestObserver<>();

    Observable<String> observable = Observable.fromIterable(WORDS)
        .zipWith(Observable.interval(1, SECONDS),
            (string, index) -> String.format("%2d. %s", index, string));

    observable.subscribeOn(Schedulers.computation())
        .subscribe(observer);

    // expect
    observer.assertNoValues();
    observer.assertNotComplete();

    // when:
    testSchedulerRule.getTestScheduler().advanceTimeBy(1, SECONDS);

    // then:
    observer.assertNoErrors();
    observer.assertValueCount(1);
    observer.assertValues(" 0. the");

    // when:
    testSchedulerRule.getTestScheduler().advanceTimeTo(9, SECONDS);
    observer.assertComplete();
    observer.assertNoErrors();
    observer.assertValueCount(9);
}
```

Code 15

ion. We make another round of assertions (lines 29-31) to ensure the Observable has produced all data and that the subscriber has consumed it all as well. One more thing to note about the usage of TestScheduler is that real time moves immediately, which means our test does not have to wait 9 seconds to complete.

As you can see using this scheduler is quite handy but it requires you to supply the scheduler to the Observable under test. This will not play well with Observables that use a specific scheduler. But wait a second, we saw earlier how we can switch a scheduler without affecting production code using RxJavaPlugins, but this time supplying a TestScheduler instead of an immediate scheduler. We can even go so far as to apply the same technique of a custom JUnit rule, allowing the previous code to be rewritten in a more reusable form. First the new rule: (Code 14)

Followed by the actual test code (in a new testcase class), to use our test rule: (Code 15)

And there you have it. The usage of a TestScheduler injected via RxJavaPlugins allow you to write the test code without altering the composition of the original Observable yet it gives you the means to modify time and make assertions at specific points during the execution of the observable itself. All the techniques showed in this article should give you enough options to test out RxJava enabled code.

The Future

RxJava is one of the first libraries to provide reactive programming capabilities to Java. Version 2.0 has been redesigned to better align its API with the Reactive Streams specification, which provides a standard for asynchronous stream processing with non-blocking back pressure, targeting Java and JavaScript runtimes. I highly encourage you to review the API changes made since version 2.0; you can find a detailed description of these changes at the RxJava wiki.

In terms of testing you'll see that the core types (Observable, Maybe, and Single) now sport a handy method named test() that creates a TestObserver instance for you on the spot. You may now chain method calls on TestObserver and there are some new assertion methods found on this type, as well. ■

Reactor by Example

Simon Basle is a software-development aficionado, especially interested in reactive programming, software design aspects (OOP, design patterns, software architecture), rich clients, what lies beyond code (continuous integration, (D)VCS, best practices), and a bit of computer science in the form of concurrent programming. He works at Pivotal on Reactor.

RxJava recap

Reactor, like RxJava 2, is a fourth-generation reactive library. Spring custodian Pivotal launched Reactor, which builds on the Reactive Streams specification, Java 8, and the ReactiveX vocabulary. Its design is the result of a savant mix of designs and core contributors from previous versions and RxJava.

In the previous articles in this eMag, "RxJava by Example" and "Testing RxJava", you learned about the basics of reactive programming: how data is conceptualized as a stream, the `Observable` class and its various operators, and the factory methods that create `Observables` from static and dynamic sources.

`Observable` is the push source and `Observer` is the simple interface for consuming this source via the act of subscribing. Keep in mind that the contract of an `Observable` is to notify its `Observer` of zero or more data items through `onNext`, optionally followed by either an `onError` or `onComplete` terminating event.

To test an `Observable`, RxJava provides a `TestSubscriber`, which is a special flavor of `Observer` that allows you to assert events in your stream.

This article will draw a parallel between Reactor and what you already learned about RxJava, and showcase the common elements as well as the differences.

Reactor's types

Reactor's two main types are the Flux<T> and Mono<T>. A `Flux` is the equivalent of an RxJava `Observable`, capable of emitting zero or more items, and then optionally either completing or error-ing.

A `Mono`, on the other hand, can emit once at most. It corresponds to both Single and Maybe types on the RxJava side. Thus an asyn- ▶

KEY TAKEAWAYS

Reactor is a reactive-streams library that targets Java 8 and provides an Rx-conforming API.

It uses the same approach and philosophy as RxJava despite some API differences.

It is a fourth-generation reactive library that allows operator fusion, like RxJava 2.

Reactor is a core dependency in the reactive-programming-model support of Spring Framework 5.

chronous task that only wants to signal completion can use a `Mono<Void>`.

This simple distinction between two types makes things easy to grasp while providing meaningful semantics in a reactive API: by just looking at the returned reactive type, you can know if a method is more of a "fire and forget" or "request-response" (`Mono`) kind of thing or if it deals with multiple data items as a stream (`Flux`).

Both `Flux` and `Mono` make use of this semantic by coercing the relevant type when using operators. For instance, calling `single()` on a `Flux<T>` will return a `Mono<T>`, whereas concatenating two `Monos` together using `concatWith` will produce a `Flux`. Similarly, some operators will make no sense on a `Mono` (for example `take(n)`, which produces n > 1 results), whereas other operators will only make sense on a `Mono` (e.g. `or(otherMono)`).

One aspect of the Reactor design philosophy is to keep the API lean, and this separation into two reactive types is a good middle ground between expressiveness and API surface.

Build on Rx, with Reactive Streams at every stage

As expressed in "RxJava by Example", RxJava bears some superficial resemblance to the Java 8 Streams API, in terms of concepts. Reactor looks a lot like RxJava, but this is of course in no way a coincidence. The intention is to provide a Reactive Streams-native library that exposes an Rx-conforming operator API for asynchronous logic composition. So while Reactor is rooted in Reactive Streams, it seeks general API alignment with RxJava where possible.

Reactive libraries and Reactive Streams adoption

Reactive Streams (henceforth abbreviated "RS") is, according the initiative, means "to provide a standard for asynchronous stream processing with non-blocking backpressure." It is a set of textual specifications along with a TCK and four simple interfaces (`Publisher`, `Subscriber`, `Subscription`, and `Processor`) that will be integrated in Java 9.

It mainly deals with the concept of reactive-pull backpressure (more on that later) and how to interoperate between several implementing reactive sources. It doesn't cover operators at all, focusing instead exclusively on the stream's lifecycle.

A key differentiator for Reactor is its RS-first approach. Both `Flux` and `Mono` are `RS Publisher` implementations and conform to reactive-pull backpressure.

In RxJava 1, only a subset of operators support backpressure, and even though RxJava 1 has adapters to RS types, its `Observable` doesn't implement these types directly. That is easily explained by the fact that RxJava 1 predates the RS specification; it served as one of the foundational works during the specification's design.

That means that each time you use these adapters you are left with a `Publisher`, which again doesn't have any operator. In order to do anything useful from there, you'll probably want to go back to an `Observable`, which means using yet another adapter. This visual clutter can be detrimental to readability, especially when an entire framework like

Spring 5 directly builds on top of `Publisher`.

Another RxJava 1 difference to keep in mind when migrating to Reactor or RxJava 2 is that in the RS specification, `null` values are not authorized. This might turn out important if your codebase uses `null` to signal some special cases.

RxJava 2 was developed after the RS specification and thus directly implements `Publisher` in its new `Flowable` type. But instead of focusing exclusively on RS types, RxJava 2 also keeps the legacy RxJava 1 types (`Observable`, `Completable`, and `Single`) and introduces the RxJava's `Optional`, `Maybe`. Although they still provide the semantic differentiation we talked about earlier, these types have the drawback of not implementing RS interfaces. Note that unlike in RxJava 1, `Observable` does not support the backpressure protocol in RxJava 2 (a feature now exclusively reserved to `Flowable`). It has been kept for the purpose of providing a rich and fluent API for cases, such as user-interface event-ing, where backpressure is impractical or impossible. `Completable`, `Single`, and `Maybe` by design have no need for backpressure support; they will offer a rich API and defer any workload until subscribed.

Reactor is once again leaner in this area, sporting `Mono` and `Flux` types that both implement `Publisher` and are backpressure-ready. There's a relatively small overhead for `Mono` to behave as a `Publisher`, but it is mostly offset by other `Mono` optimizations. We'll see in a later section what backpressure means for `Mono`.

An API similar but not equal to RxJava's

The ReactiveX and RxJava vocabularies of operators can be overwhelming at times and some operators have confusing names for historical reasons. Reactor aims to have a more compact API and to deviate in some cases — for example, to choose better names — but overall the two APIs look a lot alike. In fact, the latest iterations of RxJava 2 actually borrow some vocabulary from Reactor, a hint of the ongoing close collaboration between the two projects. Some operators and concepts first appear in one library or the other but often end up in both.

For instance, `Flux` has the same familiar `just` factory method (albeit having only two `just` variants: one element and a vararg). But `from` has been replaced by several explicit variants, most notably `fromIterable`. `Flux` also has all the usual suspects in term of operators: `map`, `merge`, `concat`, `flatMap`, `take`, etc.

One RxJava operator name that Reactor eschewed was the puzzling `amb` operator, which has been replaced with the more appropriately named `firstEmitting`. Additionally, to introduce greater consistency in the API, `toList` has been renamed `collectList`. In fact, all `collectXXX` operators now aggregate values into a specific type of collection but still produce a `Mono` of said collection, while `toXXX` methods are reserved for type conversions that take you out of the reactive world — for example, `toFuture()`.

One more way Reactor can be leaner, this time in terms of class instantiation and resource usage, is fusion: Reactor is capable of merging multiple sequential uses of certain operators (e.g. calling `concatWith` twice) into a single use, only instantiating the operator's inner classes once (which is called "macro-fusion"). That includes some data-source-based optimization that greatly helps `Mono` offset the cost of implementing `Publisher`. It is also capable of sharing resources like inner queues between several compatible operators (called "micro-fusion"). These capabilities make Reactor a fourth-generation reactive library. But that is a topic for a future article.

Let's take a closer look at a few Reactor operators. You will notice the contrast with some of the examples in the earlier articles in our eMag.

A few operator examples

This section contains snippets of code, and you're encouraged to try them and experiment further with Reactor. To that effect, you should open your IDE of choice and create a test project with Reactor as a dependency.To do so in Maven, add the following to the dependencies section of your pom.xml: (Code 1)

Let's play with the examples used in the previous articles in this eMag.

Similarly to how you would create your first `Observable` in RxJava, you can create a `Flux` using the `just(T…)` and `fromIterable(Iterable<T>)` Reactor factory methods. Remember that given a `List`, `just` would only emit the list as one whole, single emission, while `fromIterable` will emit, as its name suggests, each element from the iterable list. (Code 2)

Like in the corresponding RxJava examples, this prints: (Code 3) ▶

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.0.3.RELEASE</version>
</dependency>
To do the same in Gradle, edit the dependencies section to add reactor, similarly to
      this:
dependencies {
    compile “io.projectreactor:reactor-core:3.0.3.RELEASE”
}
```

Code 1

```
public class ReactorSnippets {
  private static List<String> words = Arrays.asList(
        “the”,
        “quick”,
        “brown”,
        “fox”,
        “jumped”,
        “over”,
        “the”,
        “lazy”,
        “dog”
        );

  @Test
  public void simpleCreation() {
     Flux<String> fewWords = Flux.just(“Hello”, “World”);
     Flux<String> manyWords = Flux.fromIterable(words);

     fewWords.subscribe(System.out::println);
     System.out.println();
     manyWords.subscribe(System.out::println);
  }
}
```

Code 2

```
Hello World
the quick brown fox jumped over the lazy dog
```

Code 3

```
@Test
public void findingMissingLetter() {
  Flux<String> manyLetters = Flux
        .fromIterable(words)
        .flatMap(word -> Flux.fromArray(word.split(“”)))
        .distinct()
        .sort()
        .zipWith(Flux.range(1, Integer.MAX_VALUE),
              (string, count) -> String.format(“%2d. %s”, count, string));

  manyLetters.subscribe(System.out::println);
}
```

Code 4

```
@Test
public void restoringMissingLetter() {
  Mono<String> missing = Mono.just("s");
  Flux<String> allLetters = Flux
        .fromIterable(words)
        .flatMap(word -> Flux.fromArray(word.split("")))
        .concatWith(missing)
        .distinct()
        .sort()
        .zipWith(Flux.range(1, Integer.MAX_VALUE),
              (string, count) -> String.format("%2d. %s", count, string));

  allLetters.subscribe(System.out::println);
}
```

Code 5

```
@Test
public void shortCircuit() {
  Flux<String> helloPauseWorld =
              Mono.just("Hello")
                  .concatWith(Mono.just("world")
                  .delaySubscriptionMillis(500));

  helloPauseWorld.subscribe(System.out::println);
}
```

Code 6

```
1. a 2. b ... 18. r 19. t 20. u ... 25. z
```

Code 7

```
1. a 2. b ... 18. r 19. s 20. t ... 26. z
```

Code 8

```
1. a 2. b ... 18. r 19. s 20. t ... 26. z
```

In order to output the individual letters in the "quick brown fox" sentence, you also need `flatMap` (as you did in "RxJava by Example"), but in Reactor, you use `fromArray` instead of `from`. You then want to filter out duplicate letters and sort what's left over using `distinct` and `sort`. Finally, you want to output an index for each distinct letter, which can be done using `zipWith` and `range`: (Code 4)

This helps us notice missing S, as expected: (Code 7)

One way of fixing that is to correct the original words array, but we could also manually add the S value to the `Flux` of letters using `concat`/`concatWith` and a `Mono`: (Code 5)

This adds the missing S just before we filter out duplicates and sort/count the letters: (Code 8)

The previous article noted the resemblance between the Rx vocabulary and the Java Streams API and, in fact, when the data is readily available from memory, Reactor, like Java Streams, acts in simple push mode (see the backpressure section below to understand why). More complex and truly asynchronous snippets wouldn't work with this pattern of just subscribing in the main thread, primarily because control would return to the main thread and then exit the application as soon as the subscription is done. For instance: (Code 6)

This snippet prints "Hello" but fails to print the delayed "world" because the test terminates too early. In snippets and tests where you only sort of write a main class like this, you'll usually want to revert to blocking behavior. To do that, you could create a `CountDownLatch` and call `countDown` in your subscriber (both in `onError` and `onComplete`). But that's not very reactive, is it? And what if you forget to count down, because of error for instance?

A second way to solve that issue is to use one of the operators that revert to the non-reactive world. Specifically, `toIterable` and `toStream` will both produce a blocking instance. Let's use `toStream` in an example: (Code 9)

As you would expect, this prints "Hello", pauses, then prints "world" and terminates. ▶

```
@Test
public void blocks() {
  Flux<String> helloPauseWorld =
    Mono.just("Hello")
        .concatWith(Mono.just("world")
                        .delaySubscriptionMillis(500));

  helloPauseWorld.toStream()
                 .forEach(System.out::println);
}
```

Code 9

```
@Test
public void firstEmitting() {
  Mono<String> a = Mono.just("oops I'm late")
                       .delaySubscriptionMillis(450);
  Flux<String> b = Flux.just("let's get", "the party", "started")
                       .delayMillis(400);

  Flux.firstEmitting(a, b)
      .toIterable()
      .forEach(System.out::println);
}
```

Code 10

```
<dependency>
  <groupId>org.springframework.boot.experimental</groupId>
  <artifactId>spring-boot-starter-web-reactive</artifactId>
</dependency>
```

Code 11

As we mentioned above, Reactor renamed RxJava's `amb()` operator to `firstEmitting`, which more clearly hints at the operator's purpose: selecting the first `Flux` to emit. The following example creates a `Mono`, delays it by 450 ms, and creates a `Flux` that emits its values with a 400 ms pause before each value. When `firstEmitting()` them together, the first value from the `Flux` comes in before the `Mono` value so it is the Flux that ends up played. (Code 10)

This prints each part of the sentence with a 400-ms pause between each section.

At this point, you might wonder, what if you're writing a test for a `Flux` that introduces delays of 4000 ms instead of 400? You don't want to wait four seconds in a unit test! Fortunately, as we'll see in a later section, Reactor comes with powerful testing capabilities that nicely cover this case.

Having sampled how Reactor compares for a few common operators, let's zoom back and have a look at other differentiating aspects of the library.

A Java 8 foundation

Reactor targets Java 8 rather than previous Java versions. This again aligns with the goal of reducing the API surface: RxJava targets Java 6 where there is no `java.util.function` package so you can't take advantage of classes like `Function` or `Consumer`. Specific, additional classes like `Func1`, `Func2`, `Action0`, `Action1`, etc. were required instead. In RxJava 2, these classes mirror `java.util.function` the way Reactor 2 used to do when it still had to support Java 7.

The Reactor API also embraces types introduced in Java 8. Most of the time-related operators will refer to durations (e.g. `timeout`, `interval`, `delay`, etc.) so using the Java 8 `Duration` class is appropriate.

The Java 8 Streams API and `CompletableFuture` can also both be easily converted to a `Flux`/`Mono` and vice versa. But you shouldn't usually convert a `Stream` to a `Flux`. The level of indirection added by `Flux` or `Mono` is a negligible cost when they

decorate more expensive operations like I/O or memory-bound operations, but most of the time a `Stream` doesn't imply that kind of latency and it is perfectly okay to use the streams API directly. Note that you'd use the `Observable` for these use cases in RxJava 2, as the `Observable` has no backpressure and thus becomes a simple push once you've subscribed. But Reactor is based on Java 8, and the Java 8 Streams API is expressive enough for most use cases. Even though you can find `Flux` and `Mono` factories for literal or simple objects, they mostly exist to be combined in higher-level flows. You typically wouldn't want to transform an accessor like "`long getCount()`" into a "`Mono<Long> getCount()`" when migrating an existing codebase to reactive patterns.

The backpressure story

One of the main focuses (if not the main focus) of the RS specification and of Reactor itself is backpressure. The idea of backpressure is that in a push scenario where the producer is quicker than the consumer, there's value in letting the consumer signal the producer to ask it to slow down because it's being overwhelmed. This gives the producer a chance to control its pace rather than having to resort to discarding data (sampling) or, worse, to risk a cascading failure.

You may wonder how backpressure applies to `Mono`: what kind of consumer could possibly be overwhelmed by a single emission? The short answer is that probably none exists. However, there's still a key difference between how a `Mono` works and how a `CompletableFuture` works. The latter is push only: if you have a reference to the `Future`, it means the task that processes an asynchronous result is already executing. On the other hand, a `Flux` or `Mono` with backpressure enables a deferred pull-push interaction:

- "deferred" because nothing happens before the call to `subscribe()`;
- "pull" because at the subscription and request steps, the `Subscriber` will send a signal upstream to the source and essentially pull the next chunk of data; and
- "push" from producer to consumer from there on, within the boundary of the number of requested elements.

For `Mono`, `subscribe()` is the button that you press to say you're ready to receive data. For `Flux`, this button is `request(n)`, which is kind of a generalization of the former.

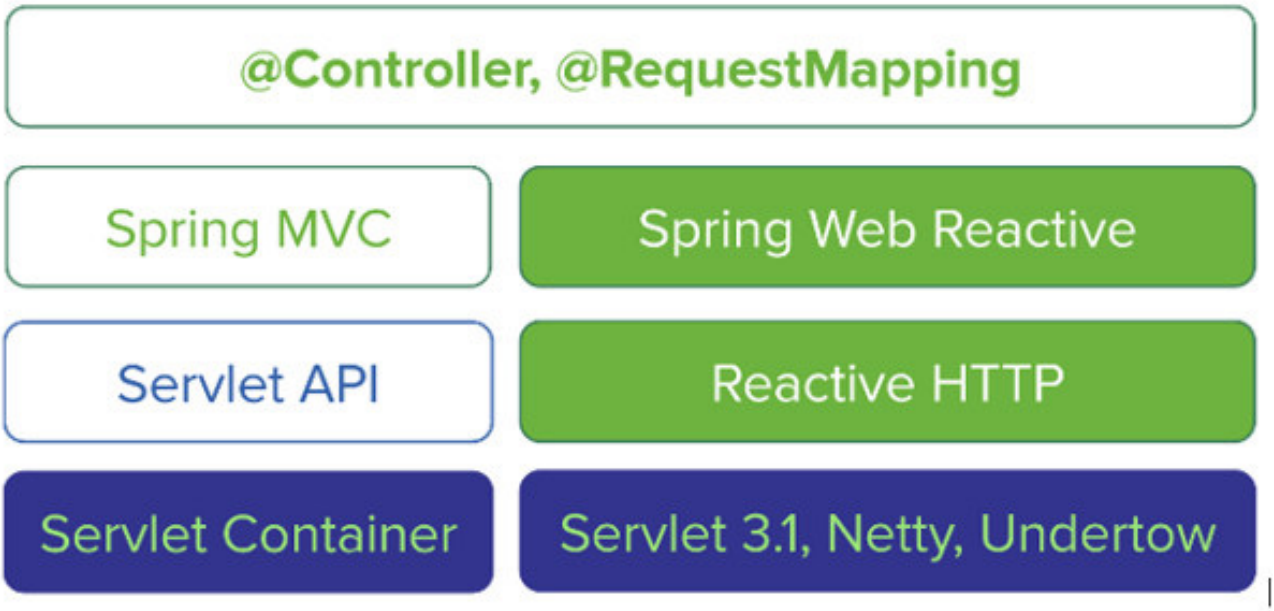
Realizing that `Mono` is a `Publisher` that will usually represent a costly task (in terms of I/O, latency, etc.) is critical to understanding the value of backpressure here: if you don't subscribe, you don't pay the cost of that task. Since `Mono` will often orchestrate with regular back-pressured `Flux` within a reactive chain, possibly combining results from multiple asynchronous sources, the availability of this on-demand subscribe triggering is key for avoiding blocking.

Having backpressure helps to differentiate that last `Mono` use case from another broad use case: asynchronously aggregating data from a `Flux` into a `Mono`. Operators like `reduce` and `hasElement` are capable of consuming each item in the `Flux`, aggregating some form of data about it (respectively, the result of a reduce function and a boolean) and exposing that data as a `Mono`. In that case, the backpressure signaled upstream is `Long.MAX_VALUE`, which lets the upstream work in a fully push fashion.

Another interesting aspect of backpressure is how it naturally limits the amount of objects held in memory by the stream. As a `Publisher`, the source of data is most probably slow (or at least slow-ish) at producing items, so the request from downstream can very well start beyond the number of readily available items. In this case, the whole stream naturally falls into a push pattern by which it notifies the consumer of new items. But when production accelerates, things nicely fall back into a pull model. In both cases, at most `N` data (the `request()` amount) is kept in memory.

You can reason about the memory used by your asynchronous processing by correlating that demand for `N` with the number `W` of kilobytes an item consumes: you can then infer that iut will consume at most `W*N` memory. In fact, Reactor will usually take advantage of knowing `N` to apply optimizations: creating queues bounded accordingly and applying prefetching strategies where it can automatically request 75% of `N` every time it receives that same three-quarters of it.

Finally, Reactor operators will sometimes change the backpressure signal to correlate it with the expectations and semantics they represent. One prime example of this behavior would be `buffer(10)`: for every request of `N` from downstream, that operator would request 10 `N` from upstream, which represents enough data to fill the number of buffers the subscriber is ready to consume. This is called "active ▶



backpressure" and it can be put to good use by developers in order to explicitly tell Reactor how to switch from an input volume to a different output volume, in micro-batching scenarios for instance.

Relation to Spring

Reactor is the reactive foundation for the whole Spring ecosystem, and most notably Spring 5 (through Spring Web Reactive) and Spring Data Kay (which corresponds to Spring Data Commons 2.0).

Having a reactive version for both of these projects is essential in the sense that this allows you to write a web application that is reactive from start to finish: a request comes in, it is asynchronously processed all the way down to and including the database, and results come back asynchronously as well. This allows a Spring application to be efficient with resources, avoiding the usual pattern of dedicating a thread to a request and blocking it for I/O.

So Reactor is going to be used for the internal reactive plumbing of future Spring applications and in the APIs these various Spring components expose. More generally, they'll be able to deal with RS `Publishers`, but most of the time these will happen to be `Flux` or `Mono`, bringing in the rich feature set of Reactor. Of course, you will be able to use any reactive library you choose, as the framework provides hooks for adapting between Reactor types and RxJava types or even simpler RS types.

By the writing of this article, you can already experiment with Spring Web Reactive in Spring Boot by using Spring Boot's `2.0.0.BUILD-SNAPSHOT` and the `spring-boot-starter-web-reactive` dependency (e.g. by generating such a project on start.spring.io). (Code 11)

This lets you write your `@Controller` mostly as usual, but replaces the underlying Spring MVC traditional layer with a reactive one, replacing many of the Spring MVC contracts with reactive non-blocking ones. By default, this reactive layer is based on top of Tomcat 8.5, but you can also elect to use Undertow or Netty.

Although Spring APIs are based on Reactor types, the Spring Web Reactive module lets you use various reactive types for both the request and response:

- `Mono<T>`: As the `@RequestBody`, the request entity `T` is asynchronously deserialized and you can chain your processing to the resulting `Mono` afterward. As the return type, once the `Mono` emits a value, the `T` is serialized asynchronously and sent back to the client. You can combine both approaches by augmenting the request `Mono` and returning that augmented chain as the resulting `Mono`.

- `Flux<T>`: Use this in streaming scenarios (including input streaming when as `@RequestBody` and server-sent events with a `Flux<ServerSentEvent>` return type.`SingleL Observable`: This works for `Mono` and `Flux` respectively but switches to a RxJava implementation.

- `Mono<Void>` as a return type: Request handling

```
@RestController
public class ExampleController {

    private final MyReactiveLibrary reactiveLibrary;

    //Note Spring Boot 4.3+ autowires single constructors now
    public ExampleController(MyReactiveLibrary reactiveLibrary) {
        this.reactiveLibrary = reactiveLibrary;
    }

    @GetMapping("hello/{who}")
    public Mono<String> hello(@PathVariable String who) {
        return Mono.just(who)
                   .map(w -> "Hello " + w + "!");
    }

    @GetMapping("helloDelay/{who}")
    public Mono<String> helloDelay(@PathVariable String who) {
        return reactiveLibrary.withDelay("Hello " + who + "!!", 2);
    }

    @PostMapping("heyMister")
    public Flux<String> hey(@RequestBody Mono<Sir> body) {
        return Mono.just("Hey mister ")
                .concatWith(body
                    .flatMap(sir -> Flux.fromArray(sir.getLastName().split("")))
                    .map(String::toUpperCase)
                    .take(1)
                ).concatWith(Mono.just(". how are you?"));
    }
}
```

Code 12

completes when the `Mono` completes.

- Non-reactive return types (`void` and `T`): These now imply that your controller method is synchronous, but should be non-blocking (short-lived processing). The request handling finishes once the method is executed. The returned `T` is serialized back to the client asynchronously.

Here is a quick example of a plain-text `@Controller` that uses the experimental web reactive module: (Code 12)

The first endpoint takes a path variable, transforms it into a `Mono<String>` and maps that name to a greeting sentence that is returned to the client.

By doing a `GET` on `/hello/Simon` we get "Hello Simon!" as a text response.

The second endpoint is a bit more complicated: it asynchronously receives a serialized `Sir` instance (a class simply made up of a `firstName` and `lastName` attributes) and `flatMaps` it into a stream of the last name's letters. It then takes the first of these letters, `maps` it to upper case and `concatenates` it into a greeting sentence.

So make the following JSON object `POST` to `/heyMister`:

```
{
      "firstName": "Paul",
      "lastName": "tEsT"
}
```

This returns the string "Hello mister T. How are you?"

The reactive aspect of Spring Data is also currently being developed in the Kay release train, which for Spring Data Commons is the 2.0.x branch. There is a first milestone build that you can get by adding the Spring Data Kay-M1 bill of materials (BOM) to your `pom`: (Code 13)

For this simplistic example, just add the `spring-data-commons` dependency in your `pom` (it will take the version from the BOM above): (Code 14)

Reactive support in Spring Data revolves around the new `ReactiveCrudRepository<T, ID>` interface, which extends `Repository<T, ID>`. This interface ▶

```
<dependencyManagement>
  <dependencies>
     <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-releasetrain</artifactId>
        <version>Kay-M1</version>
        <scope>import</scope>
        <type>pom</type>
     </dependency>
  </dependencies>
</dependencyManagement>
```

Code 13

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-commons</artifactId>
</dependency>
```

Code 14

```
@RestController
public class DataExampleController {

   private final ReactiveCrudRepository<Sir, String> reactiveRepository;

   //Note Spring Boot 4.3+ autowires single constructors now
   public DataExampleController(ReactiveCrudRepository<Sir, String> repo) {
      this.reactiveRepository = repo;
   }

   @GetMapping("data/{who}")
   public Mono<ResponseEntity<Sir>> hello(@PathVariable String who) {
      return reactiveRepository.findOne(who)
                  .map(ResponseEntity::ok)
                  .defaultIfEmpty(ResponseEntity.status(404).body(null));
   }
}
```

Code 15

exposes CRUD methods, using Reactor input and return types. There is also an RxJava 1-based version called `RxJava1CrudRepository`. For instance, in the classical blocking `CrudRepository`, retrieving one entity by its `id` would be done using "`T findOne(ID id)`". It becomes "`Mono<T> findOne(ID id)`" and "`Observable<T> findOne(ID id)`" in `ReactiveCrudRepository` and `RxJava1CrudRepository` respectively. There are even variants that take a `Mono`/`Single` as argument, to asynchronously provide the key and compose on that.

Assuming a reactive backing store (or a mock `ReactiveCrudRepository` bean), the following (very naive) controller would be reactive from start to finish: (Code 15)

Notice how the data-repository usage naturally flows into the response path: we asynchronously fetch the entity and wrap it as a `ResponseEntity` using `map`, obtaining a `Mono` we can return right away. If the Spring Data repository cannot find data for this key, it will return an empty `Mono`. We make that explicit by using `defaultIfEmpty` and returning a 404.

Testing Reactor

"Testing RxJava" covered techniques for testing an `Observable`. As we saw, RxJava comes with a `TestScheduler` that you can use with operators that accept a `Scheduler` as a parameter, to manipulate a virtual clock on these operators. It also features a `TestSubscriber` class that can be leveraged to wait for the completion of an `Observable` and to make assertions about every event (number and values for `onNext`, has `onError` triggered, etc.) In RxJava 2, the `TestSubscriber` is an RS `Subscriber`, so you can test Reactor's `Flux` and `Mono` with it!

```
<dependency>
  <groupId>io.projectreactor.addons</groupId>
  <artifactId>reactor-test</artifactId>
  <version>3.0.3.RELEASE</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>3.5.2</version>
  <scope>test</scope>
</dependency>
```

Code 16

```
@Component
public class MyReactiveLibrary {

  public Flux<String> alphabet5(char from) {
     return Flux.range((int) from, 5)
           .map(i -> "" + (char) i.intValue());
  }

  public Mono<String> withDelay(String value, int delaySeconds) {
     return Mono.just(value)
                .delaySubscription(Duration.ofSeconds(delaySeconds));
  }
}
```

Code 17

```
@Test
public void testAlphabet5LimitsToZ() {
  MyReactiveLibrary library = new MyReactiveLibrary();
  StepVerifier.create(library.alphabet5('x'))
        .expectNext("x", "y", "z")
        .expectComplete()
        .verify();
}
```

Code 18

In Reactor, these two broad features are combined into the `StepVerifier` class. It can be found in the add-on module `reactor-test` from the `reactor-addons` repository. You can initialize the `StepVerifier` by creating an instance from any `Publisher`, using the `StepVerifier.create` builder. For virtual time, you can use the `StepVerifier.withVirtualTime` builder, which takes a `Supplier<Publisher>`. This will create a `VirtualTimeScheduler` and enable it as the default scheduler implementation, making obsolete the need to explicitly pass the scheduler to operators. The `StepVerifier` will then, if necessary, configure the `Flux`/`Mono` created within the `Supplier`, turning timed operators into "virtually timed" operators. You can then script stream expectations and time progress: what the next elements should be, should there be an error, should it move forward in time, etc.

Other methods include verifying that data match a given `Predicate` or even consume `onNext` events, allowing you to do more advanced interactions with the value (like using an assertion library). Any `AssertionError` thrown by one of these will be reflected in the final verification result. Finally, call `verify()` to check your expectations; this will truly subscribe to the defined source via `StepVerifier.create` or `StepVerifier.withVirtualTime`.

Let's take a few simple examples and demonstrate how `StepVer-` ▶

```
@Test
public void testAlphabet5LastItemIsAlphabeticalChar() {
  MyReactiveLibrary library = new MyReactiveLibrary();
  StepVerifier.create(library.alphabet5('x'))
              .consumeNextWith(c -> assertThat(c)
                    .as("first is alphabetic").matches("[a-z]"))
              .consumeNextWith(c -> assertThat(c)
                    .as("second is alphabetic").matches("[a-z]"))
              .consumeNextWith(c -> assertThat(c)
                    .as("third is alphabetic").matches("[a-z]"))
              .consumeNextWith(c -> assertThat(c)
                    .as("fourth is alphabetic").matches("[a-z]"))
              .expectComplete()
              .verify();
}
```

Code 19

```
java.lang.AssertionError: expected: onComplete(); actual: onNext({)
and
java.lang.AssertionError: [fourth is alphabetic]
Expecting:
 "{"
to match pattern:
 "[a-z]"
```

Code 20

```
@Test
public void testWithDelay() {
  MyReactiveLibrary library = new MyReactiveLibrary();
  Duration testDuration =
     StepVerifier.withVirtualTime(() -> library.withDelay("foo", 30))
                 .expectSubscription()
                 .thenAwait(Duration.ofSeconds(10))
                 .expectNoEvent(Duration.ofSeconds(10))
                 .thenAwait(Duration.ofSeconds(10))
                 .expectNext("foo")
                 .expectComplete()
                 .verify();
  System.out.println(testDuration.toMillis() + "ms");
}
```

Code 21

ifier works. For these snippets, you'll want to add the following test dependencies to your pom: (Code 16)

First, imagine you have reactive class called MyReactiveLibrary that produces a few instances of Flux that you want to test. (Code 17)

The first method is intended to return five letters of the alphabet following (and including) the given starting letter. The second method returns a Flux that emits a given value after a given delay (in seconds).

The first test to write ensures that calling alphabet5 from X limits the output to X, Y, and Z. With StepVerifier it would go like this: (Code 18)

The second test to run on alphabet5 is that every returned value is an alphabetical character. For that we'd like to use a rich assertion library like AssertJ: (Code 19)

It turns out that both of these tests fail. A look at the StepVerifier output of each case may help you spot the bug. (Code 20)

So it looks like this method doesn't stop at Z but continues emitting characters along the ASCII range. We could fix that by adding a .take(Math.min(5, 'z' - from + 1)), for instance, or using the same Math.min as the second argument to range.

The last test to make involves manipulating virtual time: you'll

```
SomeFeed<PriceTick> feed = new SomeFeed<>();
Flux<PriceTick> flux =
     Flux.create(emitter ->
     {
        SomeListener listener = new SomeListener() {
           @Override
           public void priceTick(PriceTick event) {
              emitter.next(event);
              if (event.isLast()) {
                 emitter.complete();
              }
           }

           @Override
           public void error(Throwable e) {
              emitter.error(e);
           }};
        feed.register(listener);
     }, FluxSink.OverflowStrategy.BUFFER);

ConnectableFlux<PriceTick> hot = flux.publish();
```

Code 22

```
hot.subscribe(priceTick -> System.out.printf("%s %4s %6.2f%n", priceTick
     .getDate(), priceTick.getInstrument(), priceTick.getPrice()));

hot.subscribe(priceTick -> System.out.println(priceTick.getInstrument()));
We then connect to the hot flux and let it run for 5 seconds before our test snippet
     terminates:
hot.connect();
Thread.sleep(5000);
```

Code 23

test the delaying method without actually waiting for the given amount of seconds by using the `withVirtualTime` builder: (Code 21)

This tests a `Flux` that would be delayed by 30 seconds in the scenario of an immediate subscription followed by three periods of 10 seconds in which nothing happens, an `onNext("foo")`, and completion.

The `System.out` output prints the duration of the verification process (which in my latest run was 8 ms).Note that when using the `create` builder instead, the `thenAwait` and `expectNoEvent` methods would still be available but would block for the provided duration.

`StepVerifier` comes with many more methods for describing expectations and asserting the state of a `Publisher` (and if you think about new ones, contributions and feedback are always welcome in the GitHub repository).

Custom hot source

The concepts of hot and cold `Observables` discussed at the end of "RxJava by Example" also apply to Reactor.

To create a custom `Flux`, instead of the RxJava `AsyncEmitter` class, you'd use Reactor's `FluxSink`. This will cover all the asynchronous corner cases and let you focus on emitting your values.

Use `Flux.create` and get a `FluxSink` in the callback that you can use to emit data via `next`. This custom `Flux` can be cold; in order to make it hot, you can use `publish()` and `connect()`. Building on the example from the previous article with a feed of price ticks, we get an almost verbatim translation in Reactor. (Code 22)

Before connecting to the hot `Flux`, why not subscribe twice? One subscription will print the detail of each tick while the other will only print the instrument. (Code 23) ▶

Note that in the example repository, the feed would also terminate on its own if the `isLast()` method of `PriceTick` is changed.

`FluxSink` also lets you check if downstream has cancelled its subscription via `isCancelled()`. You can also get feedback on the outstanding requested amount via `requestedFromDownstream()`, which is useful if you want to simply comply with backpressure. Finally, you can make sure any specific resources your source uses are released upon `Cancellation` via `setCancellation`.

Note that there's a backpressure implication when using `FluxSink`: you must explicitly provide an `OverflowStrategy` to let the operator deal with backpressure. This is equivalent to using `onBackpressureXXX` operators (e.g. `FluxSink.OverflowStrategy.BUFFER` is equivalent to using `.onBackpressureBuffer()`), which kind of overrides any backpressure instructions from downstream.

Conclusion

Reactor builds on the Rx language but targets Java 8 and the RS specification. Concepts you might have learned in RxJava also apply to Reactor, despite a few API differences. If you want to dig deeper into Reactor, try the snippets presented in this article, which are available in a GitHub repository. There is also a Lite Rx API Hands-On workshop that covers more operators and use cases.

You can reach the Reactor team on Gitter and provide feedback there or through the GitHub issues page (and of course, we welcome pull requests as well). ■

Refactoring to Reactive: Anatomy of a JDBC Migration

Nicolae Marasoiu is a passionate software developer with years of experience building high-performance server-side applications for product and outsourcing companies, from start-ups to well-established firms. He enjoys contributing in many areas of product development and inspiring teams in their technical adventures.

Reactive programming is the new kid on the block, offering built-in solutions for some of the most difficult concepts in programming, including concurrency management and flow control. But if we work on an application development team, there's a good chance we're not using reactive and might have questions about it.

This article will transform a real (intentionally simple) legacy application (with a classic setup consisting of a web server and database back end), to a reactive model, striving for a threefold benefit:

1. Working in a functional style will allow us to compose our code and thus reuse it, while rendering it clearer and more readable thanks to a higher level of abstraction, immutability, and fluent style. For example, the `Function<T,Observable<U>>` type is composable, because we can chain together multiple such functions using the `flatMap` operator of the `Observable`.

2. Building a more resilient application allows our server to sustain a greater number of concurrent users than the actual number of available threads, while still allowing the code to execute concurrently instead of queuing in some thread pool.

We will do this by having the application react as soon as it receives each chunk of information from the database rather than having the thread wait for that information. This is a transforma- ▶

KEY TAKEAWAYS

Reactive programming comes with a learning curve and requires experience, practice, and an open mind to fully grok.

It is possible to incrementally introduce reactive programming in any application.

Reactive programming and non-blocking libraries like Netty increase scalability and elasticity and reduce operating and development costs. (See the The Reactive Manifesto for more context.).

Transformation of streams of future values into other streams of future values is a powerful programming paradigm that with practice offers great rewards.

Composability is the hallmark of functional programming; this article explores monadic composition, using the flatMap operator of the Observable.

tion from a pull mechanism (doing blocking I/O) to a push mechanism that continues execution when data becomes available or some event occurs, such as a timeout or an exception.

3. Building a more responsive application allows our browser view to update even as the first bytes start coming back from the database. This is achievable with the streaming Servlet 3.1 API and will become available with Spring Reactor and the associated web extensions on the Spring stack.

The examples in this article may be downloaded from this link.

We want to endow our program with its new reactive makeover in small steps, using incremental refactoring.

Before we start, let's enrich our test suite for the legacy application, because we rely on such tests for the correctness of the refactored versions of the program on its journey to its new functional and reactive form.

Once the tests are done, we can start with the first refactoring step: replacing method return types with `Observable`, `Single`, or `Completable`, all part of RxJava. Specifically, for each method that returns a value of type `T`, we can return `Single<T>`, an `Observable<T>` with exactly one value to emit, or an error. For `List<T>`, we will change the return type to `Observable<T>`, while for each `void` method, we will return `Completable`, which can be converted to `Single<Void>`.

Rather than replace all method return types in all the layers at the same time, let's select one layer and start there. Converting a method that returns `T` into one that returns `Single<T>`, one returning `List<T>` to an `Observable<T>`, or a `void` to a `Completable` does not need to change that method as it is known by its client code: instead, we can retain the method's original signature, but implement it as a delegating method to a new method that contains the implementation to return an `Observable<T>`. The delegating method (with the original method signature) calls `toBlocking` on the `Observable`, to enforce the original synchronous contract, before returning any values. Starting as small as possible is a great migration strategy that helps overcome the learning curve and deliver steady incremental progress. We can apply incremental refactoring with RxJava.

Here is a concrete example. We can see all of the code and its history here, where I take a classical Spring application tkssharma/Spring-JDBC and convert it to RxJava in two ways: using the RxJDBC library (in the rx-jdbc branch) and using the PgAsync library (in pgasync branch).

Let's look at the following methods from the Spring-JDBC project: (Code 1)

Following the above migration strategy, we will retain these

```
List<Student> getStudents();
Student getStudent(int studentId);
```

Code 1

```
public List<Student> getStudents() {
  return getStudentStream().toList().toBlocking().single();
```

Code 2

```
Observable<Student> getStudentStream();
Single<Student> getStudent(int studentId);
```

Code 3

```
public Observable<Student> getStudentStream() {
  List<Student> students = getStudentsViaJDBC();
  return Observable.from(students);
}
```

Code 4

method signatures, but make a small change in their implementation: (Code 2)

We introduced an extra layer: (Code 3)

With the following implementation: (Code 4)

`getStudentsViaJDBC` is the initial implementation.

We have effectively created a new reactive layer while retaining our original non-reactive signature and then replaced the original implementation with a call to our new reactive incarnation. We will make a few further iterations on this data-access layer, and then make the application reactive upwards, toward the controller layer, with the final goal of making it reactive end to end.

The `Observable.toBlocking` method acts as a bridge between the classic and reactive worlds. It is what we need to use to plug reactive code (even if only in API) into code that is classical in the large scale, like a servlet in one end and JDBC at the other. Until those ends are made reactive as well, we need this method.

Of course, at the end of our refactoring, the synchronous `toBlocking` call is anathema to the asynchronous reactive paradigm, so ultimately we will want to remove those from our production code.

Let's say we have a method `List<Student> getStudents()`. We would create a new method `Observable<Student> getStudentStream()` and move the implementation to it, wrapping the resulting list into an `Observable` using `Observable.from(Iterable)`. The original method will call the new one and then `studentStream.toList().toBlocking().single()` to convert it back to `List`. This is inherently blocking, but this is okay since the existing implementation of `getStudentStream` is already blocking at this point.

The biggest obstacle in the reactive learning curve is learning to think in terms of `Observables`. We may find it intuitive for a `List` to become a stream (which is, after all, exactly what an `Observable` is) but applying that notion to individual values is less intuitive. To conceptualize this, consider the traditional Java concept of a `Future`: although it contains a single value, it can be viewed as a particular case of a stream of future values, which happens to contain just a single value or an error if no value can in fact be successfully emitted.

Our first step, wrapping return types in `Observable`, does not change the nature of execution. It is still synchronous, blocking I/O just like JDBC does.

We have completed step one of our refactoring: changing signatures from `List<Student> getStudents()` to `Observable<Student> getStudents()`. Interestingly, even the method `Student getStudent()`, which returns just a single student, is also refactored to `Observable<Student> getStudent()` or potentially to `Single<Student> getStudent()`. Furthermore, even `void` methods are refactored to return `Completable`.

We can further apply the reactive paradigm from the top down, by wrapping large or small parts into a reactive envelope (API) and then further breaking down each part where, for example, we need asynchronicity or non-blocking I/O.

To implement the new signature, instead of returning `studentList`, we return `Observable.just(studentList)`.

At this point, we've introduced `Observable` in a few places but ▶

```
Observable.defer(
   ()->Observable.just(actuallyProduceStudentsViaTripToDatabase()
)).
```

Code 5

```
@RequestMapping(value = "/student.html", method = RequestMethod.GET)
public DeferredResult<ModelAndView> showStudents(Model model) {
   Observable<ModelAndView> observable = studentDAO.getAllStudents()
            .toList()
            .map(students -> {
                ModelAndView modelAndView = new ModelAndView("home");
                modelAndView.addObject("students", students);
                return modelAndView;
            });
   DeferredResult<ModelAndView> deferredResult = new DeferredResult<>();
   observable.subscribe(result -> deferredResult.setResult(result),
                              e -> deferredResult.setErrorResult(e));
   return deferredResult;
}
```

Code 6

```
public Observable<Student> getStudents() {
   Class<String> stringClass = String.class;
   return database
            .select("select id,name from student")
            .getAs(Integer.class, stringClass)
            .map(row->{
                    Student student = new Student();
                    student.setId(row._1());
                    student.setName(String.valueOf(row._2()));
                    return student;
                });
}
```

Code 7

```
<dependency>
   <groupId>com.github.davidmoten</groupId>
   <artifactId>rxjava-jdbc</artifactId>
   <version>0.7.2</version>
</dependency>
```

Code 8

```
public Observable<Student> getStudents() {
       return database
       .queryRows("select id,name from student")
       .map(row -> {
           Student student = new Student();
           int idx = 0;
           student.setId(row.getLong(idx++));
           student.setName(row.getString(idx++));
           return student;
       });
   }
```

Code 9

```
<dependency>
    <groupId>com.github.alaisi.pgasync</groupId>
    <artifactId>postgres-async-driver</artifactId>
    <version>0.9</version>
</dependency>
```

Code 10

besides simply wrapping and unwrapping a list, basically nothing has changed. But this was an important step because it has made our code composable, and we are now ready to make the next move and start using some of the power behind `Observable`, namely lazy evaluation, which is also available in Java streams. Instead of returning `Observable.just(studentList)`, let's return: (Code 5)

Notice that `actuallyProduceStudentsViaTripToDatabase` is the method we started with, which returned the `List<Student>`. By wrapping it with `Observable.defer` or `Observable.fromCallable`, we obtained a lazy `Observable` that only initiates the query to the database at the moment a subscriber subscribes to that data.

At this point, only the data-access layer API has been modified to return `Observable`; the controller methods are as yet unchanged so they must consume (subscribe to) the `Observable` and wait for its results in the same thread — blocking I/O. But our goal is to create end-to-end asynchronous processing, which means that instead of the controller methods returning an already populated `Result` (with data already available for the template rendering), we want to end up with an asynchronous Spring MVC supplied by the `DeferredResult` class with the async object provided by Spring MVC. (Spring plans support for streaming in the upcoming Spring Reactive Web, powered by the Spring Reactor ecosystem.) Using this approach, the controller method returns not a completed `Result` but a promise that when the result becomes available, it will be set on the previously returned `DeferredResult`. If we just modify the controller methods that return `Result` to return `DeferredResult`, that in itself is sufficient to provide a degree of asynchronicity. (Code 6)

We have made an important step toward asynchronicity but, surprisingly, this method is still waiting for the results to come from the database, i.e. it is still blocking. Why is that? You may recall that until now, the `Observable` returned by the access layer executes its `subscribe` from the calling thread so the method, despite using the `DeferredResult` approach, will block until the `Observable` delivers the data, consuming thread resources.

The next step will be to change the `Observable` so that it does not block the current thread on the `subscribe` call. This can be done in two ways: using the native reactive libraries or using `Observable.subscribeOn(scheduler)` and `observeOn(scheduler)`, executing the subscribe method and the observe methods on different schedulers (think of schedulers as thread pools).

The observe methods include `map`, `flatMap`, and `filter` — all of which transform an `Observable` to another `Observable` — as well as methods like `doOnNext`, which executes actions each time a new element is emitted in the stream. This second approach (using `subscribeOn`) is one small, intermediate step toward the goal of fully non-blocking libraries. It simply moves the `subscribe` and `observe` actions to different threads: these actions will still block until the results are available (only they will block other threads), after which they will push the results back to the subscriber, which further pushes them to a `DeferredResult`.

There are libraries that implement RxJava on top of JDBC that use this manner of blocking threads (either the calling thread or other threads, as configured). This approach is currently required for JDBC, since JDBC is a blocking API.

This intermediate step also boosts scalability, allowing us to support a greater number of truly concurrent user operations (or "flows") than the number of available threads.

Here is the `getStudents` implementation using the RxJDBC library: (Code 7)

In order to get the RxJDBC library, add this dependency in the Maven project: (Code 8)

Our third step is to introduce a true reactive library. Several exist, even for relational databases, but you can find more when focusing on a specific database such as PostgreSQL. This is because the database-access library is specific for each low-level protocol of each database. Here we use the postgres-async-driver project, which itself uses RxJava. ▶

```
public class T {
 private Observable<Long> dml(String query, Object... params) {
   return database.begin()
           .flatMap(transaction ->
                   executeDmlWithin(transaction, query, params)
                           .doOnError(e -> transaction.rollback()));
}

private Observable<Long> executeDmlWithin(
       Transaction transaction, String query, Object[] params) {
   return transaction.querySet(query, params)
         .flatMap(resultSet -> {
             Long updatedRowsCount = resultSet.iterator().next().getLong(0);
             return commitAndReturnUpdateCount(transaction, updatedRowsCount);
         });
}

private Observable<Long> commitAndReturnUpdateCount(
       Transaction transaction, Long updatedRowsCount) {
   return transaction.commit()
         .map(__ -> updatedRowsCount);
 }
}
```

Code 11

Here is the `getStudents` implementation again, this time with the PgAsync library. (Code 9)

To use the PgAsync library, import this Maven dependency: (Code 10)

At this moment we have a truly reactive (asynchronous, event-driven, non-blocking) back-end implementation. We also have an end-to-end asynchronous solution that allows us to process more user actions concurrently (at the I/O flows level) than actual threads in the JVM.

Next, let's work on transactions. We will take a scenario where we want to modify data using data-manipulation-language (DML) operations `INSERT` or `UPDATE`. Introducing asynchronicity is complicated even for the simplest transaction consisting of a single DML statement, since we are so used to transactions that block threads. And it's even more complicated for more realistic transactions, which typically contain multiple statements.

Here is how a transaction would look: (Code 11)

This is a single-statement transaction, but it illustrates how we can do transactions in an async reactive API. Transaction `begin`, `commit`, and `rollback` are all monadic functions: they return an `Observable` and can be chained with `flatMap`.

Let's travel through the example above, starting with the signature. The `dml` execution function takes a DML statement like `UPDATE` or `INSERT`, along with any parameters, and schedules it for execution. Note that `db.begin` returns `Observable<Transaction>`. The transaction is not created right away because it involves database I/O. So this is an asynchronous operation such that when execution completes, it returns a `Transaction` object on which SQL queries followed by `commit` or `rollback` can subsequently be called as required.

This `Transaction` object will be passed from Java closure to Java closure, as we see above: first, `transaction` is available as an argument to the `flatMap` operator. There, it is used in three spots:

1. First, it launches the DML statement within the `transaction`. Here, the result of the `querySet` operation that executes the DML is also an `Observable` that holds the result of the DML (generally a `Row` with updated row counts), and is further transformed with `flatMap` to another `Observable`.

2. The second `flatMap` then uses our transaction object to commit the transaction. There, the `transaction` variable is enclosed by a lambda function and is provided as an argument to this second `flatMap`. This is one way we can send data from one part of an async flow to another: using a variable from the lexical scope and using it in a lambda expression creat-

ed at one time but executed at a later time, potentially in a different thread. This is the significance of lambda expressions being Java closures: they enclose variables used in the expressions. We can send data like this using any Java closure, not just lambdas.

3. The third usage of the `transaction` variable is the `doOnError`, where the transaction is rolled back. Again note how the `transaction` variable is passed in three places via the usual Java lexical scoping, even though some pieces of the code will be executed synchronously (as part of the method execution in the calling thread), and others will be executed later, when some events happen — i.e. when a response comes from the database, asynchronously and on different threads. The value `transaction` is available in all these contexts. Ideally, shared values should be immutable, stateless, or thread safe. Java only requires them to be effectively final but this is not enough for non-primitive values.

If successful, the transaction commit result will be translated (mapped) to an update count, which can be used by callers. Here, in order to transmit the number of updated/inserted rows to an outside caller of the transactional method, we cannot capture the result count by using Java closures, since the callee is not in the same lexical scope as the caller. In this case, we need to encapsulate the result in the data type of the resulting `Observable`. If we need to carry multiple results, we can resort to immutable Java classes, arrays, or unmodifiable Java collections.

On error, rollback is called on the transaction. The error then bubbles up the `Observable` chain (not the call stack) unless it is stopped from doing so via specific `Observable` operators that say "when this `Observable` has an error, use this other `Observable` or perhaps try the same one again."This transactional update is a first example of `flatMap` chaining. What we do is pipe multiple steps one to another, in an event-driven manner: when the transaction is started, a query can be issued; when the query result is available, some result parsing and transaction commit can be issued; when the transaction is complete, the result is used to replace the successful commit result (which contains no information) with the result (here, the update count). If the final observable would not have been `Observable<Void>` but `Observable<T>`, we could have packaged `T` with our result `Long` into a data-transfer object.

In the reactive world, we aim to bring a blocking application to a non-blocking state. (A blocking application is one that blocks when performing I/O operations such as opening TCP connections.) Most of the legacy Java APIs for opening sockets, talking to databases (JDBC), and `FileInputStream/FileOutputStream` are all blocking APIs. The same is true about the early implementations of the Servlet API and many other Java constructs.

Over time, things started to adopt non-blocking counterparts; for example, Servlet 3.x integrated a few concepts like async and streaming. But in a typical J2EE application, we would typically find blocking calls, which is not always a bad thing: blocking semantics are easier to understand than explicit async APIs. Some languages like C#, Scala, and Haskell have constructs that transparently generate non-blocking implementations from blocking code: for example, the async high-order functions in C# and Scala. In Java, to my knowledge, the most robust way to perform non-blocking operations is by using reactive streams, RxJava, or non-blocking libraries such as Netty. However, things remain pretty explicit, so the entry barrier can be high. Still, when you need to support more concurrent users than the number of threads or when your application is I/O-bound and you want to minimize costs, then non-blocking will get you an extra order of magnitude in scalability, elasticity, and cost reduction.

When discussing elasticity or robustness, it is helpful to consider the moment when all threads are waiting for I/O. For example, let's assume a modern JVM can support 5,000 threads. This means that when 5,000 calls to various web services are waiting on their respective threads in a blocking application, simply no more user requests can be processed at that stage (they can only queue for later processing by some specialized threads dedicated to queuing). That might be fine in a controlled context such as a corporate intranet, but is certainly not what a start-up needs when 10 times the normal number of users suddenly decide to check out their product in a burst.

One solution to traffic spikes is horizontal scalability, bringing up more servers, but that is not elastic enough and may be expensive. Again, it all depends on the I/O of the application. But even if HTTP is the only potentially slow I/O an Internet service is exposed to, and all other I/O ops occur with internal databases or highly available low-latency services, then HTTP will move bytes ▶

only as fast as, say, a slow client on the other side of the planet.

It is true that this problem is the remit of professional load balancers, but we never know when the most highly available internal or external service will go down or when the most low-latency service actually will only run at near real time and not hard real time and so will respond slowly because of garbage collection. Then, if we're blocking in only parts of your stack, we'll have a blocking bubble, which means that threads will start blocking on the slowest blocking I/O and bring resources to a halt because of, say, a single slow blocking access that is requested by 5% of the traffic and has low business importance.

Hopefully, I have convinced you that making an application non-blocking adds value in many situations, so let's come back to our legacy application. It is blocking in all its layers, HTTP, and database access, so let's start from there. Unless all layers on a vertical (here, those are HTTP and database access) are being made async, the full flow cannot be async.

There is also a difference between async and non-blocking in that while non-blocking implies async (unless we have language constructs), async can always be done for a blocking call by simply moving it to a different thread. This has some of the same issues as the initial blocking solution, but can be a step towards the end goal in an incremental approach. For the HTTP side, we are already partially covered by the current state of the Servlet spec and Spring MVC, which gives us async behavior, but not streaming.

Async implies that when the database finishes responding, processing will kick in. When processing completes, the web layer starts rendering. When the webpage (or the JSON payload) renders, the HTTP layer is called with the full response payload.

The next step would be streaming: when the database tells the processing layer that there's more data, the processing layer accepts it. This acceptance does not necessarily imply that a dedicated thread is being used — for example, NIO or Linux `epoll` would be non-blocking. Here, the idea is that a single thread is querying a hundred thousand connections to the OS to ask whether there is anything new on them. Then the processing layer can do a transformation that outputs more semantic units, like "students". It may be useful at times, like when the data from the database represents only part of a student, to keep the partial info in the processing-layer buffers. When a bulk data fetch from the database has finally obtained all of the data on that student, it can be closed and sent to the upper layer for rendering. In such a pipeline, any component can stream at any granularity: some will just copy bytes from left to right; others will send full student instances or even batches of them; while others, like the `DeferredResult` of MVC Spring, will need the whole result before starting to write an HTTP response.

Let's review the steps of refactoring:

1. Put `Observable` in signatures.
2. Put `observable.subscribeOn` and `observable.observeOn(scheduler)` to move blocking computations (e.g. JDBC calls) to a different thread pool.
3. Use Spring MVC async to make it async.
4. Make the back end non-blocking by using a specialized library for the database that implements a non-blocking alternate implementation.
5. Wrap that non-blocking implementation in RxJava or another reactive framework if it's not already wrapped in one.
6. Use Vert.x to make it streaming.
7. Do the writes.
8. Do the (multi-statement) transactions.
9. Verify error handling.

To run the app, we use the PostgreSQL database. You can install it and set the PostgreSQL user password to "mysecretpassword". Or, more simply, install Docker and run:

```
sudo docker run -d
      -p 5432:5432
      -e POSTGRES_
      PASSWORD=mysecret
      password -d
      postgres
```

We execute student.sql to create the table and insert some sample rows.

Then we install `mvn`, and then deploy the `war` file in Tomcat or Jetty.

The URLto your locally deployed application is here; click on "students".

More on composability

We have spoken a lot about composability and it pays to understand those semantics. In the particular context of reactive pro-

gramming, I will highlight how a kind of composition functions — let's call it "sequencing". That means that we have a pipeline of processing that, given some input, produces some output in a series of steps. Each step can be synchronous (for example, applying some computation to transform a value) or asynchronous (like going to a web service or a database to get some more information). The pipeline can be pulled by its consumers or pushed by its producers.

Let's consider another example. Suppose we are building a non-blocking web server that needs to send data to a back-end system for processing and then return the result in the response. It also needs to authenticate and authorize the user that makes the request. We see a few steps of processing are already emerging in the pipeline of a request that will lead to a state change in the back end (and other systems) and a response to the end user.

In composing such a pipeline, it would be ideal not to need to be aware of the details of any particular step, like whether it is synchronous or asynchronous or how many retries it performs — and, if asynchronous, from which thread pool it takes or which non-blocking framework it uses.

Conversely, when we need to change a processing step from synchronous to asynchronous, we should only need to modify the internal implementation of the monadic function, and nothing outside that.

To illustrate, let's say we want to have a step for validating a JWT token from inside a resource server (an app server). We can do this with a library that checks data on the token payload or with a network call to an identity provider (IdP) to verify even more details, like whether the user is still valid.

Let's defined this monadic function (the return type is a monad, a type having `flatMap` on it).

```
Observable<Boolean>
      isValid(String
      token)
```

We can implement it in memory, a CPU-intensive operation that uses some token-decryption library and validates the signature and other information like expiration date or IDs.

Or we can add a trip to Google, if we use it as an IdP server.

In both cases, the world outside of this function, including the pipeline itself, is not aware of how the `Observable<Boolean>` is implemented beneath the covers. It could only call its subscriber in the same thread, like the in-memory version, in which case it is equivalent to a `boolean isValid(token)` function. Or it might actually perform I/O with Google, parsing the response to reach a `boolean` conclusion. The design is agnostic to the implementation.

We could also wrap such a function inside another one, with the same signature (`String->Observable<Boolean>`), that adds retry mechanisms on top of this validation (which would make sense for a Google trip in case an HTTP request becomes lost in traffic or has a large latency). Or it would add graceful degradation functionality so that if, for example, it cannot access external sites like Google then it could verify signatures with our own library and get on with it.

All these alternative solutions, or decorators, can be added, and each of them would still be a function from `String` to `Observable<Boolean>`.

So we have low coupling, because changing from sync to async and back does not affect the API.

But, unlike Java's `Future`, the `Observable` type is composable: we can call a function that returns a standard `Response` when we learn a token is valid but would return an `ErrorResponse` otherwise.

Let's say we have an `Observable<String>` (which does not imply that we are waiting for multiple tokens — we can wait for just one, which is a form of `Future<String>`). On this token `Observable`, we apply `flatMap` using the `isValid` function and obtain a `boolean observable`. To this, we apply a `flatMap` with a lambda function with an "if" statement: if things are valid, return the `Observable<Response>`, otherwise return another `Observable<ErrorResponse>`.

That could look like this: (Code 12)

You notice that with every `flatMap`, we start with a value of type `Observable<T>` and obtain another `Observable<U>` where `T` and `U` can be the same or different types of parameter. ▶

```
responseObservable = tokenObservable.flatMap(token -> isValid(token)
      .flatMap(valid -> valid? process(request) :
                              Observable.just(new ErrorResponse("invalid"))));
```

Code 12

This composition, then, is an important property: to compose from small components of a certain shape, larger ones of that same shape. But what is that shape?

In the case of the monad, we can model it as a type that has a type parameter `T` and two functions: `flatMap` and `lift`. The latter is easy: it converts an instance of type `T` to an instance of the monad. It is the `Observable.just(value)` or `Option.ofNullable(value)`, to give two examples for two monads.

How about `flatMap`? This is a high-order function that, given an `Observable<T>` instance called "source" and a monadic function `f(T->Observable<U>)`, produces `newObservable = sourceObservable.flatMap(t->f(t))` of type `Observable<U>` and signifies, in the case of `Observable`, that when an element of type `T` is available on source, then the function `f` is called on it, resulting in a new `Observable` for every such element. When result elements start to appear on the resulting `Observables<U>`, they are (also) emitted as part of `newObservable`, in their order of appearance. Why `Observables<U>`? Because if `sourceObservable` emits three elements, then function `f` applied to each of them will generate a total of three `Observables`. These can be merged or concatenated. Merged means that all elements from all three observables are added to `newObservable` output as soon as they emerge. This is what `flatMap` does: merges the three `Observable` results. The alternative is to first wait for all the elements from the first resulting `Observable`, then concatenate it with the second one, which is what `concatMap` does: it concatenates the resulting observables.

This property of the `Observable` type, the ability to generate new `Observable` values with augmented functionality from one `Observable` value with more processing steps and more decorator functionality like retry and fallback mechanisms, is a large part of composability.

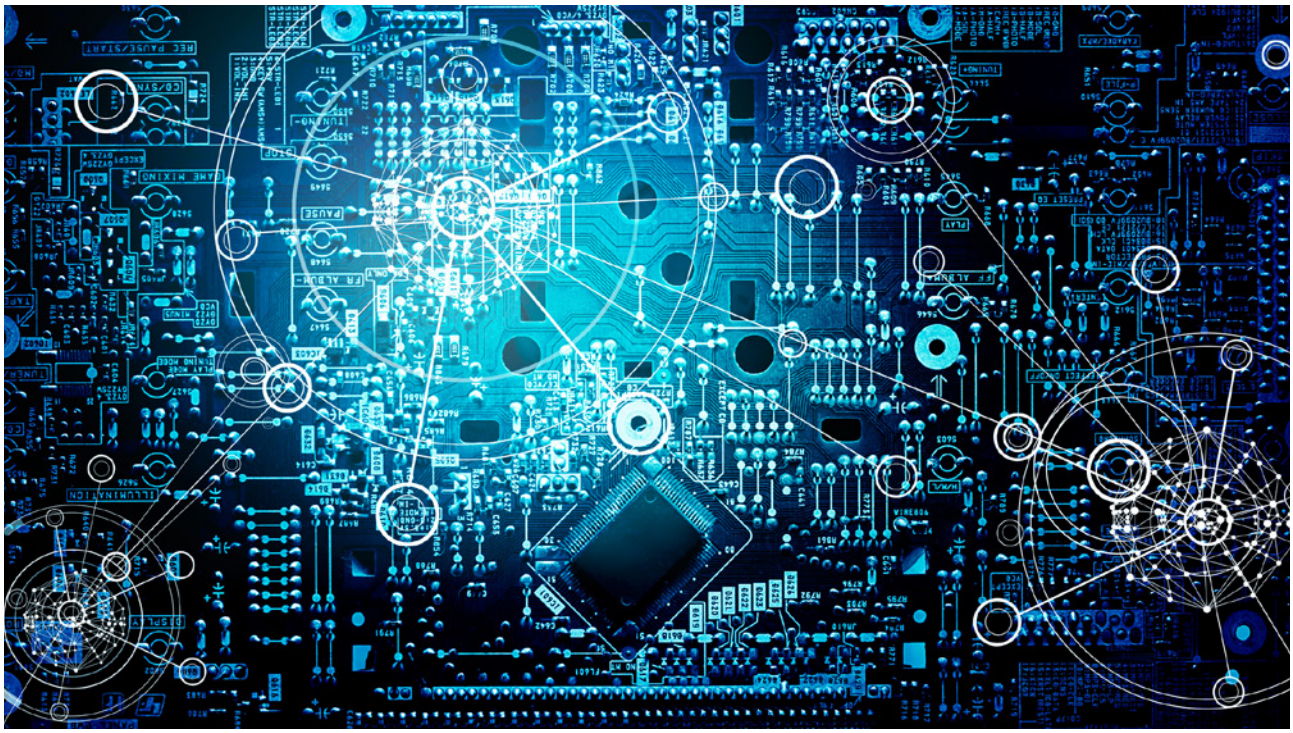
Non-blocking under the covers

At one point, I mentioned that it is possible to have more ongoing flows than available threads by using non-blocking async I/O libraries: you may wonder how that is possible. So let's take a closer look at how libraries like Netty work (Netty is the library used as the non-blocking I/O workhorse by Vert.x and PgAsync).

Java has an API called NIO, which aims at working with multiple connections with fewer threads. It works by making certain OS syscalls under the covers. (In the case of Linux, these can be `epoll` or `poll`.) For example, let's assume we have a thousand open connections. A thread will call an NIO method called `selector.select`, which is a blocking call, and return 10 connections that had queued events like "more data available", "connection closed", and others since the last query. The thread doing the query will typically dispatch the 10 events to other threads so that it can continue polling, so this first thread is an infinite loop that continually queries for events on open connections. It dispatches the 10 events for processing to a thread pool or to an event loop. Netty has an unbound thread pool to process these events. The event processing is CPU-bound (compute intensive). Any I/O would be delegated back to NIO.

A great resource that covers all of these techniques in depth is the classic Reactive Programming with RxJava by Tomasz Nurkiewicz and Ben Christensen. ■

Building Reactive Applications with Akka Actors and Java 8



Markus Eisele is a Java Champion, former Java EE Expert Group member, founder of JavaLand, reputed speaker at Java conferences around the world, and a very well-known figure in the Enterprise Java world. He works as a developer advocate at Lightbend. Find him on Twitter @myfear.

While the term "reactive" has been around for a long time, only recently has the industry recognized it as the de facto way forward in system design, leading to its mainstream adoption. In 2014, Gartner wrote that the three-tier architecture that used to be so popular was beginning to show its age. This has become even clearer with the ongoing modernization efforts driven by enterprise, which has had to start rethinking the way developers have learned to build applications for more than a decade.

Microservices are taking the software industry by storm and the shock waves are shaking the traditional development workflow to the core. We've seen software-design paradigms change and project-management methodologies evolve — but the consequent shift towards new application design and implementation has been forging its way through IT systems with unprecedented momentum. And even if the term "microservices" isn't completely new, our industry is realizing that it's not only about coupling RESTful endpoints and slicing monoliths; the real value lies in better resources consumption and extreme scalability for unpredictable workloads. The traits of the Reactive Manifesto are quickly becoming the new Bible for microservices-based architectures, as they are essentially distributed reactive applications. ▶

KEY TAKEAWAYS

The actor model provides a higher level of abstraction for writing concurrent and distributed systems, which shields the developer from explicit locking and thread management.

The actor model provides the core functionality of reactive systems, defined in the Reactive Manifesto as responsive, resilient, elastic, and message driven.

Akka is an actor-based framework that is easy to implement with the available Java 8 lambda support.

Actors allow developers to design and implement systems that focus more on the core functionality and less on the plumbing.

Actor-based systems are the perfect foundation for quickly evolving microservices architectures.

The actor model in today's applications

Applications must be highly responsive to retain user interest and must evolve quickly to remain relevant to meet the ever-changing needs and expectations of the audience. The technologies available for building applications continue to evolve at a rapid pace; science has evolved, and the emerging requirements cannot rely on yesterday's tools and methodologies. The actor model is one concept that has emerged as an effective tool for building systems that take advantage of the processing power harnessed by multicore, in-memory, clustered environments.

The actor model provides a relatively simple but powerful model for designing and implementing applications that can distribute and share work across all system resources — from threads and cores to clusters of servers and data centers. It provides an effective framework for building applications with high levels of concurrency and for increasing resource efficiency. Importantly, the actor model also has well-defined ways to gracefully handle errors and failures, ensuring a level of resilience that isolates issues and prevents cascading failures and massive downtime.

In the past, building highly concurrent systems typically involved a great deal of low-level wiring and technical programming techniques that were difficult to master. These technical challenges competed for attention with the core business functionality of the system because much of the effort had to be focused on the functional details, requiring a considerable amount of time and effort for plumbing and wiring. When building systems with actors, things are done at a higher level of abstraction because the plumbing and wiring are already built into the actor model. Not only does this liberate us from the gory details of traditional system implementation, it also allows for more focus on core system functionality and innovation.

Actors with Java 8 and Akka

Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM. Akka's "actors" allow you to write concurrent code without having to think about low-level threads and locks. Other tools include Akka Streams and the Akka HTTP layer. Although Akka is written in Scala, there is a Java API, too. The following examples work with Akka 2.4.9 and above — but the Java with lambda support part of Akka has been considered experimental as of its introduction in Akka 2.3.0 so expect it to change until its official release.

The actor is the basic unit of work in Akka. An actor is a container for state and behavior, and can create and supervise child actors. Actors interact with each other through asynchronous messages, which are processed synchronously, one message at a time. This model protects an actor's internal state, makes it thread safe,

```
<dependency>
   <groupId>com.typesafe.akka</groupId>
   <artifactId>akka-actor_2.11</artifactId>
   <version>2.4.9</version>
</dependency>
```

Code 1

```
static class Counter extends AbstractLoggingActor {
    static class Message { }

    private int counter = 0;

    {
      receive(ReceiveBuilder
        .match(Message.class, this::onMessage)
        .build()
      );
    }

    private void onMessage(Message message) {
      counter++;
      log().info("Increased counter " + counter);
    }
}
```

Code 2

```
public static void main(String[] args) {
    ActorSystem system = ActorSystem.create("sample1");
    ActorRef counter = system.actorOf(Counter.props(), "counter");

}
```

Code 3

and implements event-driven behavior that won't block other actors. To get started, you don't need much more than the Akka Maven dependency. (Code 1)

Changing actor state

Just like we exchange text messages on mobile devices, we use messages to invoke an actor. And also like text messages, the messages between actors must be immutable. The most important part of working with an actor is to define the messages it can receive. (The message is also commonly referred to as "protocol" because it defines the point of interaction between actors.) Actors receive messages and react to them. They can send other messages, change their state or behavior, and create other actors.

An actor's initial behavior is defined by implementing the receive() method with the help of a ReceiveBuilder in the default constructor. receive() matches incoming messages and executes related behavior. The behavior for each message is defined using a lambda expression. In the following example, the ReceiveBuilder uses a reference to the interface method onMessage. The onMessage method increases a counter (internal state) and logs an info message via the AbstractLoggingActor.log method. (Code 2)

With the actor in place, we need to start it. This is done via the ActorSystem, which controls the lifecycle of the actors inside it. But first, we need to supply some additional information about how to start the actor. The akka.actor.Props is a configuration object that is used to expose pieces of context-scoped configuration to all pieces of the framework. It is used in creating our actors; it is immutable so it is thread-safe and fully shareable.

```
return Props.
      create(Counter.
      class);
```

The Props object describes the constructor arguments of the actor. It is a good practice to encapsulate it in a factory function that is closer to the actor's constructor. The ActorSystem itself is created in the main method. (Code 3) ▶

```
counter.tell(new Counter.Message(), ActorRef.noSender());
```

Code 4

```
[01/10/2017 10:15:15.400] [sample1-akka.actor.default-dispatcher-4]
[akka://sample1/user/counter] Increased counter 1
```

Code 5

```
static class Alarm extends AbstractLoggingActor {
    // contract
    static class Activity {}
    static class Disable {
      private final String password;
      public Disable(String password) {
        this.password = password;
      }
    }
    static class Enable {
      private final String password;
      public Enable(String password) {
        this.password = password;
      }
    }

    // ...
}
```

Code 6

```
private final String password;
public Alarm(String password) {
      this.password = password;
     // ...
}
```

Code 7

```
public static Props props(String password) {
      return Props.create(Alarm.class, password);
}
```

Code 8

Both ActorSystem ("sample1") and the contained actor "counter" receive names for navigating actor hierarchies (more on these later). Now the ActorRef can send a message to the actor, for example: (Code 4)

Here, the two parameters define the message to be sent and the sender of the message. (As the name implies, noSender indicates that the sender isn't used in this case.) If you run the above example, you get the expected output: (Code 5)

This is a simple example yet it supplies all of our desired thread safety. Messages that are sent to actors from different threads are shielded from concurrency problems since the actor framework serializes the message processing. You can find the complete example online.

Changing actor behavior

You will notice that our simple example modified the actor state but did not change its behavior, and it never sent messages to other actors. Consider the case of a burglar alarm: we can enable and disable it with a password and its sensor detects activity. If a burglar tries to disable the alarm without the correct password, it will sound. As an actor, the alarm can react to three messages: disable, enable with a password (supplied as a payload), and tampering. All of them go into the contract. (Code 6)

The actor gets a preset attribute for the password, which is also passed into the constructor. (Code 7)

The aforementioned akka.actor.Props configuration object also needs to know about the password attribute in order to pass it to the actual constructor when the actor system starts. (Code 8)

```
private final PartialFunction<Object, BoxedUnit> enabled;
private final PartialFunction<Object, BoxedUnit> disabled;
```

Code 9

```
public Alarm(String password) {
      this.password = password;

     enabled = ReceiveBuilder
        .match(Activity.class, this::onActivity)
        .match(Disable.class, this::onDisable)
        .build();

      disabled = ReceiveBuilder
        .match(Enable.class, this::onEnable)
        .build();

      receive(disabled);
    }
}
```

Code 10

```
private void onActivity(Activity ignored) {
    log().warning("oeoeoeoeoe, alarm alarm!!!");
 }
```

Code 11

```
private void onEnable(Enable enable) {
   if (password.equals(enable.password)) {
     log().info("Alarm enable");
     getContext().become(enabled);
   } else {
     log().info("Someone failed to enable the alarm");
   }
}
```

Code 12

The Alarm actor also needs a behavior for each possible message. These behaviors are implementations of the receive method of AbstractActor. The receive method should define a series of match statements (each of type PartialFunction<Object, BoxedUnit>) that define the messages the actor can handle and the implementation of how to process the messages. (Code 9)

If this signature seems frightening, our code can effectively sweep it under the rug by using a ReceiveBuilder that can be used as shown earlier. (Code 10)

Note that the call to receive at the end sets the default behavior to "disabled". All three behaviors are implemented using three existing methods (onActivity, onDisable, and onEnable). The easiest of these methods to work on is onActivity. If there is activity, the alarm logs a string to the console. Note that there is no message payload required for activity so we just give it the name "ignored". (Code 11)

If the actor receives an enable message, the new state will be logged to the console and the state changed to enabled. If the password doesn't match, it logs a short warning. The message payload now contains the password that we can access to validate it. (Code 12)

When it receives a disable message, the actor needs to check the password, log a short message about the changed state, and actually change the state to disabled or log a warning message that the password was wrong. (Code 13)

That completes the actor logic. We are ready to start the actor system and send it a couple of messages. Note that our secret password "cat" is passed as a ▶

```
private void onDisable(Disable disable) {
  if (password.equals(disable.password)) {
    log().info("Alarm disabled");
    getContext().become(disabled);
  } else {
    log().warning("Someone who didn't know the password tried to disable it");
  }
}
```

Code 13

```
ActorSystem system = ActorSystem.create();
    final ActorRef alarm = system.actorOf(Alarm.props("cat"), "alarm");
```

Code 14

```
alarm.tell(new Alarm.Activity(), ActorRef.noSender());
alarm.tell(new Alarm.Enable("dogs"), ActorRef.noSender());
alarm.tell(new Alarm.Enable("cat"), ActorRef.noSender());
alarm.tell(new Alarm.Activity(), ActorRef.noSender());
alarm.tell(new Alarm.Disable("dogs"), ActorRef.noSender());
alarm.tell(new Alarm.Disable("cat"), ActorRef.noSender());
alarm.tell(new Alarm.Activity(), ActorRef.noSender());
```

Code 15

```
[01/10/2017 10:15:15.400] [default-akka.actor.default-dispatcher-4] [akka://default/
     user/alarm] Someone failed to enable the alarm
[01/10/2017 10:15:15.401] [default-akka.actor.default-dispatcher-4] [akka://default/
     user/alarm] Alarm enable
[WARN] [01/10/2017 10:15:15.403] [default-akka.actor.default-dispatcher-4] [akka://
     default/user/alarm] oeoeoeoeoe, alarm alarm!!!
[WARN] [01/10/2017 10:15:15.404] [default-akka.actor.default-dispatcher-4] [akka://
     default/user/alarm] Someone who didn't know the password tried to disable it
[01/10/2017 10:15:15.404] [default-akka.actor.default-dispatcher-4] [akka://
     default/user/alarm] Alarm disabled
```

Code 16

property to the actor system. (Code 14)

Let's send these messages: (Code 15)

They produce the following output: (Code 16)

You can find the complete working example online.

Until now, we've only used individual actors to process messages — but like in a business, actors form natural hierarchies.

Actor hierarchies

Actors may create other actors. When one actor creates another actor, the creator is known as the "supervisor" and the created actor is known as the "worker". Supervisors may create worker actors for many reasons, the most common of which is to delegate work to them.

The supervisor also becomes a caretaker of the workers. Just as a parent watches over his or her children, the supervisor tends to the wellbeing of its workers. If a worker runs into a problem, it suspends itself (which means that it will not process normal messages until resumed) and notifies its supervisor of the failure.

So far, we've created actors and assigned them names. Actor names are used to identify actors in the hierarchy. The actor that generally interacts the most with others is the parent of all user-created actors, the guardian with the path /user. Actors created with the primordial system.actorOf() are direct children of this guardian and when it terminates, all normal actors in the system will shut down as well. In the above alarm example, we

created a user actor with the path /user/alarm.

Since actors are created in a strictly hierarchical fashion, each actor inherits a unique sequence of names by recursively following the supervision links between child and parent towards the root of the actor system. This sequence resembles the enclosing folders in a file system, hence the adopted name "path" to refer to it, although actor hierarchy has some fundamental differences from a file-system hierarchy.

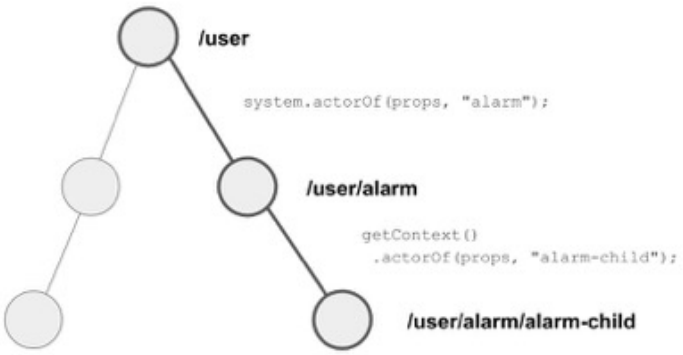


Figure 1

Inside of an actor, you can call `getContext().actorOf(props, "alarm-child")` to create a new actor named alarm-child as a child of the alarm actor . The child lifecycle is bound to the supervising actor, which means that if you stop the "alarm" actor, it will also stop the child. (Figure 1)

This hierarchy also has a significant influence on how actor-based systems handle failures. The quintessential feature of actor systems is that tasks are split up and delegated until they become small enough to be handled in one piece. In doing so, not only is the task itself clearly structured but the resulting actors can be reasoned about in terms of:

- which messages they should process,
- how they should react normally, and
- how failures should be handled.

If one actor does not have the means for dealing with a certain situation, it sends a corresponding failure message to its supervisor, asking for help. The supervisor has four options for reacting to a failure:

- **Resume** the child, keeping its accumulated internal state but ignoring the message that lead to the failure.
- **Restart** the child, clearing out its accumulated internal state by starting a new instance.
- **Stop** the child permanently and send all future messages for the child to the dead-letter box.
- **Escalate** the failure, thereby failing the supervisor itself.

Let's examine this with an example: A NonTrustWorthyChild receives Command messages and increases an internal counter with each message. If the message count is divisible by four, it throws a RuntimeException, which it escalates to the Supervisor. There's nothing really new here, as the command message has no payload. (Code 17)

The Supervisor starts the NonTrustWorthyChild in its constructor and forwards all command messages that it receives directly to the child. (Code 18)

When the Supervisor actor starts, the resulting hierarchy will be /user/supervisor/child. Before this can be done, we need to define the so-called supervision strategy. Akka provides two classes of supervision strategies: OneForOneStrategy and AllForOneStrategy. The former applies the obtained directive only to the failed child whereas the latter applies it to all siblings as well. Normally, you should use the OneForOneStrategy, which is the default if none is explicitly specified. It is defined by overriding the SupervisorStrategy method. (Code 19)

The first parameter defines the maxNrOfRetries, which is the number of times a child actor may restart before the Supervisor stops it (a negative value indicates no limit). The withinTimeRange parameter defines the duration of the window for maxNrOfRetries. As defined above, the strategy is to try 10 times within 10 seconds. The DeciderBuilder works exactly as the ReceiveBuilder to define matches on occurring exceptions and how to react to them. In this case, if there are 10 retries within 10 seconds the Supervisor stops the NonTrustWorthyChild and sends all remaining messages to the dead-letter box.

The actor system is started with the Supervisor actor. (Code 20) ▶

```
public class NonTrustWorthyChild extends AbstractLoggingActor {

  public static class Command {}
  private long messages = 0L;

  {
    receive(ReceiveBuilder
      .match(Command.class, this::onCommand)
      .build()
    );
  }

  private void onCommand(Command c) {
    messages++;
    if (messages % 4 == 0) {
      throw new RuntimeException("Oh no, I got four commands, can't handle more");
    } else {
      log().info("Got a command " + messages);
    }
  }

  public static Props props() {
    return Props.create(NonTrustWorthyChild.class);
  }
}
```

Code 17

```
public class Supervisor extends AbstractLoggingActor {
{
    final ActorRef child = getContext().actorOf(NonTrustWorthyChild.props(), "child");

    receive(ReceiveBuilder
      .matchAny(command -> child.forward(command, getContext()))
      .build()
    );

  }
  //…
}
```

Code 18

```
@Override
public SupervisorStrategy supervisorStrategy() {
   return new OneForOneStrategy(
      10,
      Duration.create(10, TimeUnit.SECONDS),
      DeciderBuilder
          .match(RuntimeException.class, ex -> stop())
          .build()
   );
}
```

Code 19

```
ActorSystem system = ActorSystem.create();
final ActorRef supervisor = system.actorOf(Supervisor.props(), "supervisor");
When the system is up, we start to send 10 command messages to the Supervisor. Note,
      that the message "Command" was defined in the NonTrustWorthyChild.
for (int i = 0; i < 10; i++) {
  supervisor.tell(new NonTrustWorthyChild.Command(), ActorRef.noSender());
}
```

Code 20

```
[01/10/2017 12:33:47.540] [default-akka.actor.default-dispatcher-3] [akka://default/
      user/supervisor/child] Got a command 1
[01/10/2017 12:33:47.540] [default-akka.actor.default-dispatcher-3] [akka://default/
      user/supervisor/child] Got a command 2
[01/10/2017 12:33:47.540] [default-akka.actor.default-dispatcher-3] [akka://default/
      user/supervisor/child] Got a command 3
[01/10/2017 12:33:47.548] [default-akka.actor.default-dispatcher-4] [akka://default/
      user/supervisor] Oh no, I got four commands, I can't handle any more
java.lang.RuntimeException: Oh no, I got four commands, I can't handle any more
      ...
[01/10/2017 12:33:47.556] [default-akka.actor.default-dispatcher-3] [akka://
      default/user/supervisor/child] Message [com.lightbend.akkasample.sample3.
      NonTrustWorthyChild$Command] from Actor[akka://default/deadLetters] to
      Actor[akka://default/user/supervisor/child#-1445437649] was not delivered.
      [1] dead letters encountered.
```

Code 21

The output shows that after four messages, the exception escalates to the Supervisor and the remaining messages are sent to the deadLetters box. If the SupervisorStrategy would have been defined to restart() instead of stop(), the Supervisor would have started a new instance of the NonTrustWorthyChild actor instead. (Code 21)

We can turn off or adjust this logging with configuration settings akka.log-dead-letters and akka.log-dead-letters-during-shutdown.

You can follow up with the complete example online and play around with the SupervisorStrategy.

Summary

With Akka and Java 8, it is now possible to create distributed, microservices-based systems that just a few years ago were the stuff of dreams. Enterprises across all industries now desire the ability to create systems that can evolve at the speed of the business and cater to the whims of users. We can now elastically scale systems that support massive numbers of users and process huge volumes of data. We can now harden systems with a level of resilience that lets us measure downtime not in hours but seconds. Actor-based systems allow us to create quickly evolving microservice architectures that can scale and run without stopping.

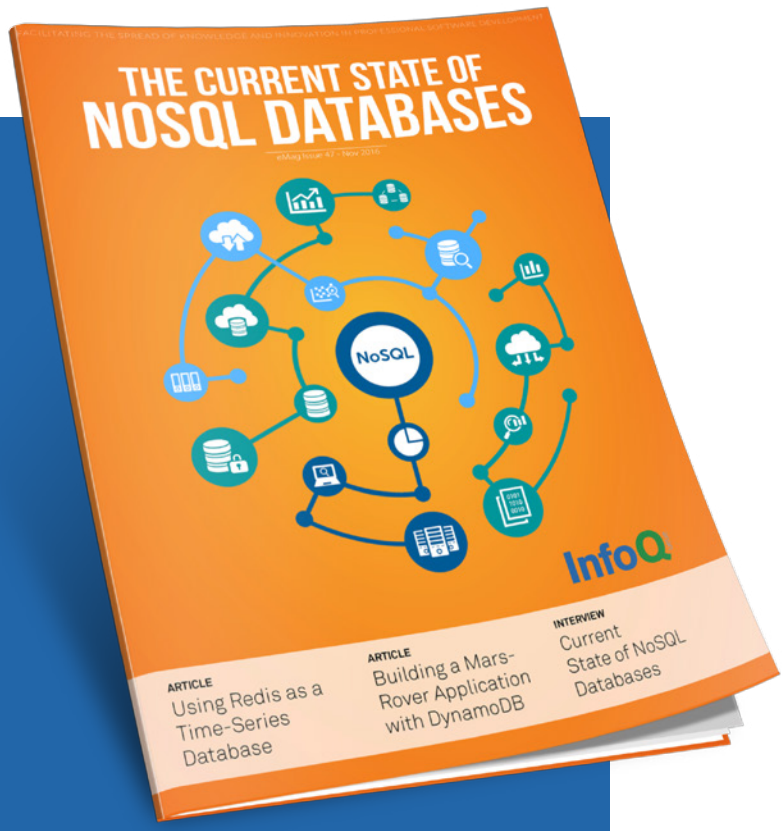
It's the actor model that provides the core functionality of reactive systems, defined in the Reactive Manifesto as responsive, resilient, elastic, and message driven.

The fact that actor systems can scale horizontally, from a single node to clusters with many nodes, provides us with the flexibility to scale our systems for massive load. It is also possible to implement systems with the capability to scale elastically — that is, scale the capacity of systems, either manually or automatically, to adequately support the peaks and valleys of system activity.

With actors and actor systems, failure detection and recovery is an architectural feature, not something that can be patched on later. Out of the box, we get actor-supervision strategies for handling problems with subordinate worker actors up to the actor-system level, with clusters of nodes that actively monitor the state of the cluster — where dealing with failures is baked into the DNA of actors and actor systems. This starts at the most basic level with the asynchronous exchange of messages between actors. If you send me a message, you have to consider the possible outcomes: what do you do when you get the reply you expect… or don't!? This goes all the way towards implementing strategies for handling nodes that leave and join a cluster.

Thinking in terms of actors is in many ways much more intuitive for us when designing systems. The way actors interact is more natural to us since it has, on a simplistic level, much in common with how humans interact. This allows us to design and implement systems in ways that let us focus more on the core functionality of the systems and less on the plumbing. ■

PREVIOUS ISSUES



47

The Current State of NoSQL Databases

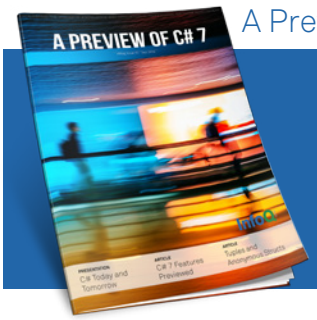
This eMag focuses on the current state of NoSQL databases. It includes articles, a presentation and a virtual panel discussion covering a variety of topics ranging from highly distributed computations, time series databases to what it takes to transition to a NoSQL database solution.

Architectures You've Always Wondered About

46

This eMag takes a look back at five of the most popular presentations from the Architectures You've Always Wondered About track at QCons in New York, London and San Francisco, each presenter adding a new insight into the biggest challenges they face, and how to achieve success. All the companies featured have large, cloud-based, microservice architectures, which probably comes as no surprise.

A Preview of C# 7



45

The C# programming language was first released to the public in 2000. and since that time the language has evolved through 6 releases to add everything from generics to lambda expressions to asynchronous methods and string interpolation. In this eMag we have curated a collection of new and previously content that provides the reader with a solid introduction to C# 7 as it is defined today.

Cloud Lock-In

44

Technology choices are made, and because of a variety of reasons--such as multi-year licensing cost, tightly coupled links to mission-critical systems, long-standing vendor relationships--you feel "locked into" those choices. In this InfoQ eMag, we explore the topic of cloud lock-in from multiple angles and look for the best ways to approach it.