Baeldung

Spring Reactive Guide

1: Intro To Reactor Core

2. Debugging Reactive Streams in Java

3. Guide to Spring 5 WebFlux

4. Introduction to the Functional Web Framework in Spring 5

5. Spring 5 WebClient

6. Spring WebClient vs. RestTemplate

7. Spring WebClient Requests With Parameters

8. Handling Errors in Spring WebFlux

9. Spring Security 5 for Reactive Applications

10. Concurrency in Spring WebFlux

1: Intro To Reactor Core

1. Introduction

Reactor Core is a Java 8 library that implements the reactive programming model. It's built on top of the Reactive Streams specification, which is a standard for building reactive applications.

From a background of non-reactive Java development, going reactive can be quite a steep learning curve. This becomes more challenging when comparing it to the Java 8 *Stream* API, as they could be mistaken for being the same high-level abstractions.

In this chapter, we'll attempt to demystify this paradigm. We'll take small steps through Reactor until we've illustrated how to compose reactive code, laying the foundation for more advanced chapters to come in a later series.

Before we look at Reactor, we should look at the Reactive Streams Specification. This is what Reactor implements, and it lays the groundwork for the library.

Essentially, Reactive Streams is a specification for asynchronous stream processing.

In other words, it's a system where lots of events are being produced and consumed asynchronously. Think about a stream of thousands of stock updates per second coming into a financial application, and for it to have to respond to those updates in a timely manner.

So one of the main goals is to address the problem of backpressure. If we have a producer that's emitting events to a consumer faster than it can process them, then eventually the consumer will be overwhelmed with events, running out of system resources.

Backpressure means that our consumer should be able to tell the producer how much data to send in order to prevent this from happening, which is precisely what's laid out in the specification.

3. Maven Dependencies

Before we get started, let's add our Maven dependencies:

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.4.16</version>
</dependency>
```

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.6</version>
</dependency>
```

We'll also add Logback as a dependency because we'll be logging the output of the Reactor in order to better understand the flow of data.

4. Producing a Stream of Data

In order for an application to be reactive, the first thing it must be able to do is produce a stream of data.

This could be something like the stock update example that we gave earlier. Without this data, we wouldn't have anything to react to, which is why this is a logical first step.

Reactive Core gives us two data types that enable us to do this.

4.1. Flux

The first way of doing this is with *Flux*. It's a stream that can emit 0..n elements. Let's try creating a simple one:

```
1. Flux<Integer> just = Flux.just(1, 2, 3, 4);
```

In this case, we have a static stream of four elements.

4.2. Mono

The second way of doing this is with a *Mono*, which is a stream of 0..1 elements. Let's try instantiating one:

```
1. Mono<Integer> just = Mono.just(1);
```

This looks and behaves almost exactly the same as the *Flux*, only this time we're limited to no more than one element.

4.3. Why Not Only Flux?

Before experimenting further, it's worth highlighting why we have these two data types.

First, it should be noted that both *Flux* and *Mono* are implementations of the Reactive Streams *Publisher* interface. Both classes are compliant with the specification, and we could use this interface in their place:

```
1. Publisher<String> just = Mono.just("foo");
```

But really, knowing this cardinality is useful because a few operations only make sense for one of the two types, and it can be more expressive (imagine *findOne()* in a repository).

Now that we have a high-level overview of how to produce a stream of data, we need to subscribe to it in order for it to emit the elements.

5.1. Collecting Elements

Let's use the *subscribe()* method to collect all the elements in a stream:

```
1.  List<Integer> elements = new ArrayList<>();
2.
3.  Flux.just(1, 2, 3, 4)
4.    .log()
5.    .subscribe(elements::add);
6.
7.  assertThat(elements).containsExactly(1, 2, 3, 4);
```

The data won't start flowing until we subscribe. Notice that we added some logging as well, which will be helpful when we look at what's happening behind the scenes.

5.2. The Flow of Elements

With logging in place, we can use it to visualize how the data is flowing through our stream:

```
1.  20:25:19.550 [main] INFO  reactor.Flux.Array.1 - |
2.  onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
3.  20:25:19.553 [main] INFO  reactor.Flux.Array.1 - |
4.  request(unbounded)
5.  20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onNext(1)
6.  20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onNext(2)
7.  20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onNext(3)
8.  20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onNext(4)
9.  20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onComplete()
```

5.2. The Flow of Elements

First of all, everything is running on the main thread. We won't go into any details about this now, as we'll be taking a further look at concurrency later on in this chapter.

It does make things simple though, as we can deal with everything in order.

Now let's go through the sequence that we logged one by one:

1. *onSubscribe()* – This is called when we subscribe to our stream.

2. request(unbounded) – When we call *subscribe*, behind the scenes we're creating a *Subscription*. This subscription requests elements from the stream. In this case, it defaults to *unbounded*, meaning it requests every single element available.

3. onNext() – This is called on every single element.

4. onComplete() – This is called last, after receiving the last element. There's actually an *onError()* as well, which would be called if there's an exception, but in this case, there isn't.

This is the flow laid out in the *Subscriber* interface as part of the Reactive Streams Specification. In reality, this is what's been instantiated behind the scenes in our call to *onSubscribe()*.

It's a useful method, but to better understand what's happening, let's provide a *Subscriber* interface directly:

```
Flux.just(1, 2, 3, 4)
  .log()
  .subscribe(new Subscriber<Integer>() {
    @Override
    public void onSubscribe(Subscription s) {
      s.request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(Integer integer) {
      elements.add(integer);
    }

    @Override
    public void onError(Throwable t) {}

    @Override
    public void onComplete() {}
});
```

We can see that each possible stage in the above flow maps to a method in the *Subscriber* implementation.

It just happens that *Flux* has provided us with a helper method to reduce this verbosity.

5.3. Comparison to Java 8 Streams

It still might appear that we have something synonymous to a Java 8 Stream doing collect:

```
List<Integer> collected = Stream.of(1, 2, 3, 4)
  .collect(toList());
```

Only we don't.

The core difference is that Reactive is a push model, whereas the Java 8 *Streams* are a pull model. **In a reactive approach, events are *pushed* to the subscribers as they come in.**

The next thing to notice is that a *Streams* terminal operator is just that, a terminal, pulling all the data and returning a result.

With Reactive, we could have an infinite stream coming in from an external resource, with multiple subscribers attached and removed on an ad hoc basis.

We can also do things like combine streams, throttle streams, and apply backpressure, which we'll cover next.

The next thing we should consider is backpressure. In our example, the subscriber is telling the producer to push every single element at once. This could end up becoming overwhelming for the subscriber, consuming all of its resources.

Backpressure is when a downstream can tell an upstream to send it less data in order to prevent it from being overwhelmed.

We can modify our *Subscriber* implementation to apply backpressure. Let's tell the upstream to only send two elements at a time by using *request()*:

```
1.  Flux.just(1, 2, 3, 4)
2.    .log()
3.    .subscribe(new Subscriber<Integer>() {
4.      private Subscription s;
5.      int onNextAmount;
6.
7.      @Override
8.      public void onSubscribe(Subscription s) {
9.          this.s = s;
10.         s.request(2);
11.     }
12.
13.     @Override
14.     public void onNext(Integer integer) {
15.         elements.add(integer);
16.         onNextAmount++;
17.         if (onNextAmount % 2 == 0) {
18.             s.request(2);
19.         }
20.     }
21.
22.     @Override
23.     public void onError(Throwable t) {}
24.
25.     @Override
26.     public void onComplete() {}
27. });
```

Now if we run our code again, we'll see the request(2) is called, followed by two onNext() calls, and then request(2) again:

```
23:31:15.395 [main] INFO  reactor.Flux.Array.1 - |
onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
23:31:15.397 [main] INFO  reactor.Flux.Array.1 - | request(2)
23:31:15.397 [main] INFO  reactor.Flux.Array.1 - | onNext(1)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | onNext(2)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | request(2)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | onNext(3)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | onNext(4)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | request(2)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | onComplete()
```

Essentially, this is reactive pull backpressure. We're requesting the upstream to only push a certain amount of elements, and only when we're ready.

If we imagine we're being streamed tweets from Twitter, it would then be up to the upstream to decide what to do. If tweets were coming in, but there were no requests from the downstream, then the upstream could drop items, store them in a buffer, or use some other strategy.

We can also perform operations on the data in our stream, responding to events as we see fit.

7.1. Mapping Data in a Stream

A simple operation that we can perform is applying a transformation. In this case, we'll just double all the numbers in our stream:

```
1.  Flux.just(1, 2, 3, 4)
2.    .log()
3.    .map(i -> i * 2)
4.    .subscribe(elements::add);
```

map() will be applied when *onNext()* is called.

7.2. Combining Two Streams

We can then make things more interesting by combining another stream with this one. Let's try this by using the *zip()* function:

```
1.  Flux.just(1, 2, 3, 4)
2.    .log()
3.    .map(i -> i * 2)
4.    .zipWith(Flux.range(0, Integer.MAX_VALUE),
5.      (one, two) -> String.format("First Flux: %d, Second Flux: %d",
6.  one, two))
7.    .subscribe(elements::add);
8.
9.  assertThat(elements).containsExactly(
10.   "First Flux: 2, Second Flux: 0",
11.   "First Flux: 4, Second Flux: 1",
12.   "First Flux: 6, Second Flux: 2",
13.   "First Flux: 8, Second Flux: 3");
```

Here we're creating another *Flux* that keeps incrementing by one, and streaming it together with our original one. We can see how these work together by inspecting the logs:

```
20:04:38.064 [main] INFO  reactor.Flux.Array.1 - |
onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
20:04:38.065 [main] INFO  reactor.Flux.Array.1 - | onNext(1)
20:04:38.066 [main] INFO  reactor.Flux.Range.2 - |
onSubscribe([Synchronous Fuseable] FluxRange.RangeSubscription)
20:04:38.066 [main] INFO  reactor.Flux.Range.2 - | onNext(0)
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | onNext(2)
20:04:38.067 [main] INFO  reactor.Flux.Range.2 - | onNext(1)
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | onNext(3)
20:04:38.067 [main] INFO  reactor.Flux.Range.2 - | onNext(2)
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | onNext(4)
20:04:38.067 [main] INFO  reactor.Flux.Range.2 - | onNext(3)
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | onComplete()
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | cancel()
20:04:38.067 [main] INFO  reactor.Flux.Range.2 - | cancel()
```

Note that we now have one subscription per *Flux*. The *onNext()* calls are also alternated, so the index of each element in the stream will match when we apply the *zip()* function.

Thus far, we've focused primarily on cold streams. These are static, fixed-length streams that are easy to deal with. A more realistic use case for reactive might be something that happens infinitely.

For example, we could have a stream of mouse movements that constantly needs to be reacted to, or a Twitter feed. These types of streams are called hot streams, as they are always running and can be subscribed to at any point in time, missing the start of the data.

8.1. Creating a *ConnectableFlux*

One way to create a hot stream is by converting a cold stream into one. Let's create a *Flux* that lasts forever, outputting the results to the console, which will simulate an infinite stream of data coming from an external resource:

```
ConnectableFlux<Object> publish = Flux.create(fluxSink -> {
    while(true) {
        fluxSink.next(System.currentTimeMillis());
    }
})
  .publish();
```

By calling *publish()*, we're given a *ConnectableFlux*. This means that calling *subscribe()* won't cause it to start emitting, allowing us to add multiple subscriptions:

```
publish.subscribe(System.out::println);
publish.subscribe(System.out::println);
```

If we try running this code, nothing will happen. It's not until we call connect() that the *Flux* will start emitting:

```
publish.connect();
```

9. Concurrency

All of our above examples have currently run on the main thread. However, we can control which thread our code runs on if we want. The Scheduler interface provides an abstraction around asynchronous code, for which many implementations are provided for us. Let's try subscribing to a different thread than main:

9. Concurrency

```
Flux.just(1, 2, 3, 4)
  .log()
  .map(i -> i * 2)
  .subscribeOn(Schedulers.parallel())
  .subscribe(elements::add);
```

The *Parallel* scheduler will cause our subscription to be run on a different thread, which we can prove by looking at the logs. We can see the first entry comes from the main thread, and the Flux is running in another thread called *parallel-1:*

```
20:03:27.505 [main] DEBUG reactor.util.Loggers$LoggerFactory -
Using Slf4j logging framework
20:03:27.529 [parallel-1] INFO  reactor.Flux.Array.1 - |
onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - |
request(unbounded)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onNext(1)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onNext(2)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onNext(3)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onNext(4)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - |
onComplete()
```

Concurrency gets more interesting than this, and it'll be worth exploring it in another chapter.

10. Conclusion

In this chapter, we've given a high-level, end-to-end overview of Reactive Core. We explained how we can publish and subscribe to streams, apply backpressure, operate on streams, and also handle data asynchronously. This should hopefully lay a foundation for us to write reactive applications.

Later chapters in this series will cover more advanced concurrency and other reactive concepts. There's also another chapter covering Reactor with Spring.

The source code for our application is available over on GitHub.

2. Debugging Reactive Streams in Java

Debugging reactive streams is probably one of the main challenges we'll have to face once we start using these data structures.

Keeping in mind that Reactive Streams have been gaining popularity over the last few years, it's a good idea to know how we can carry out this task efficiently.

Let's start by setting up a project using a reactive stack to see why this is often troublesome.

We want to simulate a real-case scenario, where several asynchronous processes are running, and we've introduced some defects in the code that will eventually trigger exceptions.

To understand the big picture, we'll mention that our application will be consuming and processing streams of simple *Foo* objects, which contain only an *id*, a *formattedName*, and a *quantity* field.

2.1. Analyzing the Log Output

Now let's examine a snippet and the output it generates when an unhandled error shows up:

```
1.  public void processFoo(Flux<Foo> flux) {
2.      flux.map(FooNameHelper::concatFooName)
3.        .map(FooNameHelper::substringFooName)
4.        .map(FooReporter::reportResult)
5.        .subscribe();
6.  }
7.
8.  public void processFooInAnotherScenario(Flux<Foo> flux) {
9.      flux.map(FooNameHelper::substringFooName)
10.       .map(FooQuantityHelper::divideFooQuantity)
11.       .subscribe();
12. }
```

After running our application for a few seconds, we can see that it's logging exceptions from time to time.

If we take a close look at one of the errors, we'll find something similar to this:

```
Caused by: java.lang.StringIndexOutOfBoundsException: String index
out of range: 15
    at j.l.String.substring(String.java:1963)
    at com.baeldung.debugging.consumer.service.FooNameHelper
      .lambda$1(FooNameHelper.java:38)
    at r.c.p.FluxMap$MapSubscriber.onNext(FluxMap.java:100)
    at r.c.p.FluxMap$MapSubscriber.onNext(FluxMap.java:114)
    at r.c.p.FluxConcatMap$ConcatMapImmediate.
innerNext(FluxConcatMap.java:275)
    at r.c.p.FluxConcatMap$ConcatMapInner.onNext(FluxConcatMap.
java:849)
    at r.c.p.Operators$MonoSubscriber.complete(Operators.
java:1476)
    at r.c.p.MonoDelayUntil$DelayUntilCoordinator.
signal(MonoDelayUntil.java:211)
    at r.c.p.MonoDelayUntil$DelayUntilTrigger.
onComplete(MonoDelayUntil.java:290)
    at r.c.p.MonoDelay$MonoDelayRunnable.run(MonoDelay.java:118)
    at r.c.s.SchedulerTask.call(SchedulerTask.java:50)
    at r.c.s.SchedulerTask.call(SchedulerTask.java:27)
    at j.u.c.FutureTask.run(FutureTask.java:266)
    at j.u.c.ScheduledThreadPoolExecutor$ScheduledFutureTask
      .access$201(ScheduledThreadPoolExecutor.java:180)
    at j.u.c.ScheduledThreadPoolExecutor$ScheduledFutureTask
      .run(ScheduledThreadPoolExecutor.java:293)
    at j.u.c.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.
java:1149)
    at j.u.c.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.
java:624)
    at j.l.Thread.run(Thread.java:748)
```

Based on the root cause, and noting the *FooNameHelper* class mentioned in the stack trace, we can imagine that on some occasions, our *Foo* objects are being processed with a *formattedName* value that's shorter than expected.

Of course, this is just a simplified case, and the solution seems rather obvious.

But let's imagine this was a real-case scenario, where the exception itself doesn't help us solve the issue without some context information.

Was the exception triggered as part of the *processFoo*, or of the *processFooInAnotherScenario* method?

Did other previous steps affect the *formattedName* field before arriving at this stage?

The log entry won't help us answer these questions.

To make things worse, sometimes the exception isn't even thrown from within our functionality.

For example, imagine we rely on a reactive repository to persist our *Foo* objects. If an error arises at that point, we might not even have a clue where to begin to debug our code.

We need tools to debug reactive streams efficiently.

One option to figure out what's going on with our application is to start a debugging session using our favorite IDE.

We'll have to set up a couple of conditional breakpoints and analyze the flow of data when each step in the stream gets executed.

However, this might be a cumbersome task, especially when we've got a lot of reactive processes running and sharing resources.

Additionally, there are many circumstances where we can't start a debugging session for security reasons.

4. Logging Information With the doOnErrorMethod or Using the Subscribe Parameter

Sometimes, **we can add useful context information by providing a *Consumer* as a second parameter of the *subscribe* method:**

```
1.  public void processFoo(Flux<Foo> flux) {
2.
3.
4.      // ...
5.
6.
7.      flux.subscribe(foo -> {
8.          logger.debug("Finished processing Foo with Id {}", foo.
9.  getId());
10.     }, error -> {
11.         logger.error(
12.           "The following error happened on processFoo method!",
13.            error);
14.     });
15. }
```

Note: It's worth mentioning that if we don't need to carry out further processing on the *subscribe* method, we can chain the *doOnError* function on our publisher:

```
1.  flux.doOnError(error -> {
2.      logger.error("The following error happened on processFoo
3.  method!", error);
4.  }).subscribe();
```

Now we'll have some guidance on where the error might be coming from, even though we still don't have much information about the actual element that generated the exception.

The Reactor library provides a Hooks class that lets us configure the behavior of *Flux* and *Mono* operators.

By simply adding the following statement, our application will instrument the calls to the publishers' methods, wrap the construction of the operator, and capture a stack trace:

```
Hooks.onOperatorDebug();
```

After the debug mode is activated, our exception logs will include some helpful information:

```
16:06:35.334 [parallel-1] ERROR c.b.d.consumer.service.FooService
  - The following error happened on processFoo method!
java.lang.StringIndexOutOfBoundsException: String index out of
range: 15
    at j.l.String.substring(String.java:1963)
    at c.d.b.c.s.FooNameHelper.lambda$1(FooNameHelper.java:38)
    ...
    at j.l.Thread.run(Thread.java:748)
    Suppressed: r.c.p.FluxOnAssembly$OnAssemblyException:
Assembly trace from producer [reactor.core.publisher.
FluxMapFuseable] :
    reactor.core.publisher.Flux.map(Flux.java:5653)
    c.d.b.c.s.FooNameHelper.substringFooName(FooNameHelper.
java:32)
    c.d.b.c.s.FooService.processFoo(FooService.java:24)
    c.d.b.c.c.ChronJobs.consumeInfiniteFlux(ChronJobs.java:46)
    o.s.s.s.ScheduledMethodRunnable.run(ScheduledMethodRunnable.
java:84)
    o.s.s.s.DelegatingErrorHandlingRunnable
      .run(DelegatingErrorHandlingRunnable.java:54)
    o.u.c.Executors$RunnableAdapter.call(Executors.java:511)
    o.u.c.FutureTask.runAndReset(FutureTask.java:308)
Error has been observed by the following operator(s):
    |_    Flux.map - c.d.b.c.s.FooNameHelper
            .substringFooName(FooNameHelper.java:32)
    |_    Flux.map - c.d.b.c.s.FooReporter.
reportResult(FooReporter.java:15)
```

As we can see, the first section remains relatively the same, but the following sections provide information about:

1. The assembly trace of the publisher — here we can confirm that the error was first generated in the *processFoo* method.

2. The operators that observed the error after it was first triggered, with the user class where they were chained.

Note: In this example, mainly to see this clearly, we're adding the operations on different classes.

We can toggle the debug mode on or off at any time, but it won't affect *Flux* and *Mono* objects that have already been instantiated.

5.1. Executing Operators on Different Threads

One other aspect to keep in mind is that the assembly trace is generated properly even if there are different threads operating on the stream.

Let's have a look at the following example:

```
1. public void processFoo(Flux<Foo> flux) {
2.     flux.publishOn(Schedulers.newSingle("foo-thread"))
3.         // ...
4.        .publishOn(Schedulers.newSingle("bar-thread"))
5.        .map(FooReporter::reportResult)
6.        .subscribeOn(Schedulers.newSingle("starter-thread"))
7.        .subscribe();
8. }
```

Now if we check the logs, we'll appreciate that in this case, the first section might change a little bit, but the last two remain fairly the same.

The first part is the thread stack trace; therefore, it'll show only the operations carried out by a particular thread.

As we've seen, that's not the most important section when we're debugging the application, so this change is acceptable.

Instrumenting and generating a stack trace in every single reactive process is costly.

Thus, **we should implement the former approach only in critical cases**.

Reactor also provides a way to enable the debug mode on single crucial processes, which is less memory-consuming.

We're referring to the *checkpoint* operator:

```
public void processFoo(Flux<Foo> flux) {

    // ...


    flux.checkpoint("Observed error on processFoo", true)
      .subscribe();
}
```

Note that in this manner, the assembly trace will be logged at the checkpoint stage:

```
Caused by: java.lang.StringIndexOutOfBoundsException: String
index out of range: 15
     ...
Assembly trace from producer [reactor.core.publisher.FluxMap],
  described as [Observed error on processFoo] :
    r.c.p.Flux.checkpoint(Flux.java:3096)
    c.b.d.c.s.FooService.processFoo(FooService.java:26)
    c.b.d.c.c.ChronJobs.consumeInfiniteFlux(ChronJobs.java:46)
    o.s.s.s.ScheduledMethodRunnable.run(ScheduledMethodRunnable.
java:84)
    o.s.s.s.DelegatingErrorHandlingRunnable.
run(DelegatingErrorHandlingRunnable.java:54)
    j.u.c.Executors$RunnableAdapter.call(Executors.java:511)
    j.u.c.FutureTask.runAndReset(FutureTask.java:308)
Error has been observed by the following operator(s):
    |_    Flux.checkpoint - c.b.d.c.s.FooService.
processFoo(FooService.java:26)
```

We should implement the *checkpoint* method towards the end of the reactive chain.

Otherwise, the operator won't be able to observe errors occurring downstream.

Also, we'll note that the library offers an overloaded method. We can avoid:

- specifying a description for the observed error if we use the no-args option

- generating a filled stack trace (which is the most costly operation) by providing just the custom description

Finally, Reactor publishers offer one more method that could potentially come in handy in some cases. **By calling the *log* method in our reactive chain, the application will log each element in the flow with the state that it has at that stage.**

Let's try it out in our example:

```
1.  public void processFoo(Flux<Foo> flux) {
2.      flux.map(FooNameHelper::concatFooName)
3.        .map(FooNameHelper::substringFooName)
4.        .log();
5.        .map(FooReporter::reportResult)
6.        .doOnError(error -> {
7.          logger.error("The following error happened on processFoo
8.  method!", error);
9.        })
10.       .subscribe();
11. }
```

And check the logs:

```
1.  INFO  reactor.Flux.OnAssembly.1 - onSubscribe(FluxMap.MapSubscriber)
2.  INFO  reactor.Flux.OnAssembly.1 - request(unbounded)
3.  INFO  reactor.Flux.OnAssembly.1 - onNext(Foo(id=0, formattedName=theFo,
4.  quantity=8))
5.  INFO  reactor.Flux.OnAssembly.1 - onNext(Foo(id=1, formattedName=theFo,
6.  quantity=3))
7.  INFO  reactor.Flux.OnAssembly.1 - onNext(Foo(id=2, formattedName=theFo,
8.  quantity=5))
9.  INFO  reactor.Flux.OnAssembly.1 - onNext(Foo(id=3, formattedName=theFo,
10. quantity=6))
11. INFO  reactor.Flux.OnAssembly.1 - onNext(Foo(id=4, formattedName=theFo,
12. quantity=6))
13. INFO  reactor.Flux.OnAssembly.1 - cancel()
    ERROR c.b.d.consumer.service.FooService
      - The following error happened on processFoo method!
    ...
```

We can easily see the state of each *Foo* object at this stage, and how the framework cancels the flow when an exception happens,

Of course, this approach is also costly, and we'll have to use it in moderation.

8. Conclusion

We can spend a lot of our time and effort troubleshooting problems if we don't know the tools and mechanisms to debug our application properly.

This is especially true if we're not used to handling reactive and asynchronous data structures, and we need extra help to figure out how things work.

As always, the full example is available over on the GitHub repo.

3. Guide to Spring 5 WebFlux

Spring 5 includes Spring WebFlux, which provides reactive programming support for web applications.

In this chapter, we'll create a small reactive REST application using the reactive web components *RestController* and *WebClient*.

We'll also look at how to secure our reactive endpoints using Spring Security.

Spring WebFlux internally uses *Project Reactor* and its publisher implementations, *Flux* and *Mono*.

The new framework supports two programming models:

- Annotation-based reactive components
- Functional routing and handling

We'll focus on the annotation-based reactive components, as we already explored the functional style, routing and handling, in another chapter.

Let's start with the *spring-boot-starter-webflux* dependency, which pulls in all other required dependencies:

- spring-boot and spring-boot-starter for basic Spring Boot application setup
- spring-webflux framework
- reactor-core that we need for reactive streams and also reactor-netty

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <version>2.6.4</version>
</dependency>
```

The latest spring-boot-starter-webflux can be downloaded from Maven Central.

Now we'll build a very simple reactive REST *EmployeeManagement* application using Spring WebFlux:

- Use a simple domain model – Employee with an id and a name field
- Build a REST API with a RestController to publish Employee resources as a single resource and as a collection
- Build a client with WebClient to retrieve the same resource
- Create a secured reactive endpoint using WebFlux and Spring Security

Spring WebFlux supports annotation-based configurations in the same way as the Spring Web MVC framework.

To begin with, on the server, we'll create an annotated controller that publishes a reactive stream of the *Employee resource*.

Let's create our annotated *EmployeeController*:

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {


    private final EmployeeRepository employeeRepository;

    // constructor...
}
```

EmployeeRepository can be any data repository that supports non-blocking reactive streams.

5.1. Single Resource

Then we'll create an endpoint in our controller that publishes a single *Employee resource:*

```
@GetMapping("/{id}")
private Mono<Employee> getEmployeeById(@PathVariable String id) {
    return employeeRepository.findEmployeeById(id);
}
```

We'll wrap a single *Employee* resource in a *Mono* because we return at most one employee.

5.2. Collection Resource

We'll also add an endpoint that publishes the collection resource of all *Employees*:

```
1. @GetMapping
2. private Flux<Employee> getAllEmployees() {
3.     return employeeRepository.findAllEmployees();
4. }
```

For the collection resource, we'll use a *Flux* of type *Employee* since that's the publisher for 0..n elements.

WebClient, introduced in Spring 5, is a non-blocking client with support for reactive streams.

We can use *WebClient* to create a client to retrieve data from the endpoints provided by the *EmployeeController*.

Let's create a simple *EmployeeWebClient*:

```
1.  public class EmployeeWebClient {
2.
3.
4.      WebClient client = WebClient.create("http://localhost:8080");
5.
6.
7.      // ...
8.  }
```

Here we created a *WebClient* using its factory method, create. It'll point to localhost:*8080*, so we can use relative URLs for calls made by this client instance.

6.1. Retrieving a Single Resource

To retrieve a single resource of type *Mono* from endpoint */employee/{id}*:

```
1.  Mono<Employee> employeeMono = client.get()
2.    .uri("/employees/{id}", "1")
3.    .retrieve()
4.    .bodyToMono(Employee.class);
5.
6.
7.  employeeMono.subscribe(System.out::println);
```

6.2. Retrieving a Collection Resource

Similarly, to retrieve a collection resource of type *Flux* from endpoint */employees*:

```
1. Flux<Employee> employeeFlux = client.get()
2.   .uri("/employees")
3.   .retrieve()
4.   .bodyToFlux(Employee.class);
5.
6. employeeFlux.subscribe(System.out::println);
```

We also have a detailed chapter on setting up and working with WebClient.

We can use Spring Security to secure our reactive endpoints.

Let's suppose we have a new endpoint in our *EmployeeController*. This endpoint updates *Employee* details and sends back the updated *Employee*.

Since this allows users to change existing employees, we want to restrict this endpoint to *ADMIN* role users only.

As such, we'll add a new method to our *EmployeeController*:

```
@PostMapping("/update")
private Mono<Employee> updateEmployee(@RequestBody Employee
employee) {
    return employeeRepository.updateEmployee(employee);
}
```

Now, to restrict access to this method, we'll create *SecurityConfig* and define some path-based rules to allow only ADMIN users:

```
@EnableWebFluxSecurity
public class EmployeeWebSecurityConfig {


    // ...


    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(
      ServerHttpSecurity http) {
        http.csrf().disable()
          .authorizeExchange()
          .pathMatchers(HttpMethod.POST, "/employees/update").
hasRole("ADMIN")
          .pathMatchers("/**").permitAll()
          .and()
          .httpBasic();
        return http.build();
    }
}
```

This configuration will restrict access to the endpoint */employees/update*. Therefore, only users with a role *ADMIN* will be able to access this endpoint and update an existing *Employee*.

Finally, the annotation *@EnableWebFluxSecurity* adds Spring Security WebFlux support with some default configurations.

For more information, we also have a detailed chapter on configuring and working with Spring WebFlux security.

In this chapter, we explored how to create and work with reactive web components as supported by the Spring WebFlux framework. As an example, we built a small Reactive REST application.

Then we learned how to use *RestController* and *WebClient* to publish and consume reactive streams.

We also looked into how to create a secured reactive endpoint with the help of Spring Security.

Other than Reactive RestController and *WebClient*, the *WebFlux* framework also supports reactive *WebSocket* and the corresponding *WebSocketClient* for socket style streaming of Reactive Streams.

Finally, the complete source code used in this chapter is available over on Github.

4. Introduction to the Functional Web Framework in Spring 5

Spring WebFlux is a new functional web framework built using reactive principles.

In this chapter, we'll learn how to work with it in practice.

We'll base this off of our existing guide to Spring 5 WebFlux. In that guide, we created a simple reactive REST application using annotation-based components. Here, we'll use the functional framework instead.

We'll need the same *spring-boot-starter-webflux* dependency as defined in the previous chapter:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <version>2.6.4</version>
</dependency>
```

The functional web framework introduces a new programming model where we use functions to route and handle requests.

As opposed to the annotation-based model where we use annotation mappings, here we'll use *HandlerFunction* and *RouterFunctions*.

Similar to the annotated controllers, the functional endpoints approach is built on the same reactive stack.

3.1. HandlerFunction

The *HandlerFunction* represents a function that generates responses for requests routed to them:

```
1. @FunctionalInterface
2. public interface HandlerFunction<T extends ServerResponse> {
3.     Mono<T> handle(ServerRequest request);
4. }
```

This interface is primarily a *Function<Request, Response<T>>*, which behaves very much like a servlet.

Although, compared to a standard *Servlet#service(ServletRequest req, ServletResponse res)*, HandlerFunction doesn't take a response as an input parameter.

3.2. *RouterFunction*

RouterFunction serves as an alternative to the *@RequestMapping* annotation. We can use it to route requests to the handler functions:

```
@FunctionalInterface
public interface RouterFunction<T extends ServerResponse> {
    Mono<HandlerFunction<T>> route(ServerRequest request);
    // ...
}
```

Typically, we can import the helper function *RouterFunctions.route()* to create routes, instead of writing a complete router function.

It allows us to route requests by applying a *RequestPredicate*. When the predicate is matched, then the second argument, the handler function, is returned:

```
public static <T extends ServerResponse> RouterFunction<T> route(
  RequestPredicate predicate,
  HandlerFunction<T> handlerFunction)
```

Because the *route()* method returns a *RouterFunction*, we can chain it to build powerful and complex routing schemes.

In our previous guide, we created a simple EmployeeManagement REST application using *@RestController* and *WebClient*.

Now we'll implement the same logic using router and handler functions.

First, **we'll need to create routes using *RouterFunction* to publish and consume our reactive streams of *Employees*.**

Routes are registered as Spring beans, and can be created inside any configuration class.

4.1. Single Resource

Let's create our first route using *RouterFunction* that publishes a single *Employee* resource:

```
1.  @Bean
2.  RouterFunction<ServerResponse> getEmployeeByIdRoute() {
3.    return route(GET("/employees/{id}"),
4.      req -> ok().body(
5.        employeeRepository().findEmployeeById(req.
6.  pathVariable("id")), Employee.class));
7.  }
```

The first argument is a request predicate. Notice how we used a statically imported RequestPredicates.GET method here. The second parameter defines a handler function that will be used if the predicate applies.

In other words, the above example routes all the GET requests for the */employees/{id}* to EmployeeRepository#findEmployeeById(String id) method.

4.2. Collection Resource

Next, for publishing a collection resource, we'll add another route:

```
@Bean
RouterFunction<ServerResponse> getAllEmployeesRoute() {
  return route(GET("/employees"),
    req -> ok().body(
      employeeRepository().findAllEmployees(), Employee.class));
}
```

4.3. Single Resource Update

Finally, we'll add a route for updating the *Employee* resource:

```
@Bean
RouterFunction<ServerResponse> updateEmployeeRoute() {
  return route(POST("/employees/update"),
    req -> req.body(toMono(Employee.class))
      .doOnNext(employeeRepository()::updateEmployee)
      .then(ok().build()));
}
```

We can also compose the routes together in a single router function.

Let's see how to combine the routes created above:

```
@Bean
RouterFunction<ServerResponse> composedRoutes() {
  return
    route(GET("/employees"),
      req -> ok().body(
        employeeRepository().findAllEmployees(), Employee.class))

    .and(route(GET("/employees/{id}"),
      req -> ok().body(
        employeeRepository().findEmployeeById(req.
pathVariable("id")), Employee.class)))

    .and(route(POST("/employees/update"),
      req -> req.body(toMono(Employee.class))
        .doOnNext(employeeRepository()::updateEmployee)
        .then(ok().build())));
}
```

Here we used *RouterFunction.and()* to combine our routes.

Finally, we implemented the complete REST API needed for our *EmployeeManagement* application using routers and handlers.

To run the application, we can either use separate routes, or the single, composed one that we created above.

We can use *WebTestClient* to test our routes.

To do so, we'll first need to bind the routes using the *bindToRouterFunction* method, and then build the test client instance.

Let's test our *getEmployeeByIdRoute*:

```
1.  @Test
2.  void givenEmployeeId_whenGetEmployeeById_thenCorrectEmployee() {
3.      WebTestClient client = WebTestClient
4.        .bindToRouterFunction(config.getEmployeeByIdRoute())
5.        .build();
6.
7.
8.      Employee employee = new Employee("1", "Employee 1");
9.
10.
11.     given(employeeRepository.findEmployeeById("1")).
12. willReturn(Mono.just(employee));
13.
14.
15.     client.get()
16.       .uri("/employees/1")
17.       .exchange()
18.       .expectStatus()
19.       .isOk()
20.       .expectBody(Employee.class)
21.       .isEqualTo(employee);
22. }
```

And similarly, getAllEmployeesRoute:

```
@Test
void whenGetAllEmployees_thenCorrectEmployees() {
    WebTestClient client = WebTestClient
      .bindToRouterFunction(config.getAllEmployeesRoute())
      .build();
    List<Employee> employees = Arrays.asList(
      new Employee("1", "Employee 1"),
      new Employee("2", "Employee 2"));
    Flux<Employee> employeeFlux = Flux.fromIterable(employees);
    given(employeeRepository.findAllEmployees()).
willReturn(employeeFlux);
    client.get()
      .uri("/employees")
      .exchange()
      .expectStatus()
      .isOk()
      .expectBodyList(Employee.class)
      .isEqualTo(employees);
}
```

We can also test our *updateEmployeeRoute* by asserting that our *Employee* instance is updated via *EmployeeRepository*:

```
@Test
void whenUpdateEmployee_thenEmployeeUpdated() {
    WebTestClient client = WebTestClient
      .bindToRouterFunction(config.updateEmployeeRoute())
      .build();

    Employee employee = new Employee("1", "Employee 1 Updated");

    client.post()
      .uri("/employees/update")
      .body(Mono.just(employee), Employee.class)
      .exchange()
      .expectStatus()
      .isOk();
    verify(employeeRepository).updateEmployee(employee);
}
```

For more details on testing with *WebTestClient*, please refer to our chapter on working with WebClient and WebTestClient.

In this chapter, we introduced the new functional web framework in Spring 5, and discussed its two core interfaces, *RouterFunction* and *HandlerFunction*. We also learned how to create various routes to handle the request and send the response.

Additionally, we recreated our *EmployeeManagement* application introduced in an earlier chapter with the functional endpoints model.

As always, the full source code can be found over on Github.

5. Spring 5 WebClient

In this chapter, we'll examine *WebClient*, which is a reactive web client introduced in Spring 5.

We'll also look at the *WebTestClient*, a *WebClient* designed to be used in tests.

Simply put, *WebClient* is an interface representing the main entry point for performing web requests.

It was created as part of the Spring Web Reactive module, and will be replacing the classic *RestTemplate* in these scenarios. In addition, the new client is a reactive, non-blocking solution that works over the HTTP/1.1 protocol.

It's important to note that even though it's a non-blocking client, and it belongs to the *spring-webflux* library, the solution offers support for both synchronous and asynchronous operations, making it also suitable for applications running on a Servlet Stack.

This can be achieved by blocking the operation to obtain the result. Of course, this practice isn't suggested if we're working on a Reactive Stack.

Finally, the interface has a single implementation, the *DefaultWebClient* class, which we'll be working with.

Since we're using a Spring Boot application, all we need is the spring-boot-starter-webflux dependency to obtain Spring Framework's Reactive Web support.

3.1. Building With Maven

Let's add the following dependencies to the *pom.xml* file:

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

3.2. Building With Gradle

With Gradle, we need to add the following entries to the *build.gradle* file:

```
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-webflux'
}
```

In order to work properly with the client, we'll need to know how to:

- create an instance
- make a request
- handle the response

4.1. Creating a WebClient Instance

There are three options to choose from. The first one is creating a *WebClient* object with default settings:

```
1. WebClient client = WebClient.create();
```

The second option is to initiate a WebClient instance with a given base URI:

```
1. WebClient client = WebClient.
2. create("http://localhost:8080");
```

The third option (and the most advanced one) is building a client by using the *DefaultWebClientBuilder* class, which allows full customization:

```
1. WebClient client = WebClient.builder()
2.   .baseUrl("http://localhost:8080")
3.   .defaultCookie("cookieKey", "cookieValue")
4.   .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_
5. JSON_VALUE)
6.   .defaultUriVariables(Collections.singletonMap("url", "http://
7. localhost:8080"))
8.   .build();
```

4.2. Creating a *WebClient* Instance With Timeouts

Often, the default HTTP timeouts of 30 seconds are too slow for our needs. To customize this behavior, we can create an *HttpClient* instance and configure our *WebClient* to use it.

We can:

- set the connection timeout via the ChannelOption.CONNECT_TIMEOUT_MILLIS option
- set the read and write timeouts using a ReadTimeoutHandler and a WriteTimeoutHandler, respectively
- configure a response timeout using the responseTimeout directive

As we previously mentioned, all of these have to be specified in the *HttpClient* instance we'll configure:

```
1.  HttpClient httpClient = HttpClient.create()
2.    .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000)
3.    .responseTimeout(Duration.ofMillis(5000))
4.    .doOnConnected(conn ->
5.      conn.addHandlerLast(new ReadTimeoutHandler(5000, TimeUnit.
6.  MILLISECONDS))
7.        .addHandlerLast(new WriteTimeoutHandler(5000, TimeUnit.
8.  MILLISECONDS)));
9.
10.
11.  WebClient client = WebClient.builder()
12.    .clientConnector(new ReactorClientHttpConnector(httpClient))
13.    .build();
```

Note that while we can call timeout on our client request as well, this is a signal timeout, not an HTTP connection, a read/write, or a response timeout; it's a timeout for the Mono/Flux publisher.

4.3. Preparing a Request – Define the Method

First, we'll need to specify an HTTP method of a request by invoking *method(HttpMethod method)*:

```
1. UriSpec<RequestBodySpec> uriSpec =
2. client.method(HttpMethod.POST);
```

Or by calling its shortcut methods, such as *get*, *post*, and *delete*:

```
1. UriSpec<RequestBodySpec> uriSpec = client.post();
```

Note: although it may seem like we reused the requestspec variables *(WebClient.UriSpec, WebClient.RequestBodySpec, WebClient.RequestHeadersSpec, WebClient.ResponseSpec)*, this is just for simplicity to present different approaches. These directives shouldn't be reused for different requests. They retrieve references, and therefore, the latter operations would modify the definitions we made in previous steps.

4.4. Preparing a Request – Define the URL

The next step is to provide a URL. Once again, we have different ways of doing this.

We can pass it to the uri API as a String:

```
1. RequestBodySpec bodySpec = uriSpec.uri("/resource");
```

Using a UriBuilder Function:

```
1. RequestBodySpec bodySpec = uriSpec.
2. uri(
3.   uriBuilder -> uriBuilder.
4. pathSegment("/resource").build());
```

Or as a java.net.URL instance:

```
1. RequestBodySpec bodySpec = uriSpec.
2. uri(URI.create("/resource"));
```

Keep in mind that if we defined a default base URL for the *WebClient*, this last method would override this value.

4.5. Preparing a Request – Define the Body

Then we can set a request body, content type, length, cookies, or headers if we need to.

For example, if we want to set a request body, there are a few available ways. Probably the most common and straightforward option is using the *bodyValue* method:

```
RequestHeadersSpec<?> headersSpec =
bodySpec.bodyValue("data");
```

Or by presenting a Publisher (and the type of elements that will be published) to the *body* method:

```
RequestHeadersSpec<?> headersSpec = bodySpec.body(
  Mono.just(new Foo("name")), Foo.class);
```

Alternatively, we can make use of the *BodyInserters* utility class. For example, let's see how we can fill in the request body using a simple object, as we did with the *bodyValue* method:

```
RequestHeadersSpec<?> headersSpec = bodySpec.body(
  BodyInserters.fromValue("data"));
```

Similarly, we can use the *BodyInserters#fromPublisher* method if we're using a Reactor instance:

```
RequestHeadersSpec headersSpec = bodySpec.body(
  BodyInserters.fromPublisher(Mono.just("data")),
  String.class);
```

This class also offers other intuitive functions to cover more advanced scenarios. For instance, if we have to send multipart requests:

```
1. LinkedMultiValueMap map = new LinkedMultiValueMap();
2. map.add(“key1”, “value1”);
3. map.add(“key2”, “value2”);
4. RequestHea4.
5. dersSpec<?> headersSpec = bodySpec.body(
6.   BodyInserters.fromMultipartData(map));
```

All of these methods create a *BodyInserter* instance that we can then present as the *body* of the request.

The *BodyInserter* is an interface responsible for populating a *ReactiveHttpOutputMessage* body with a given output message and a context used during the insertion.

A *Publisher* is a reactive component in charge of providing a potentially unbounded number of sequenced elements. It’s an interface too, and the most popular implementations are *Mono* and *Flux*.

4.6. Preparing a Request – Define the Headers

After we set the body, we can set headers, cookies, and acceptable media types. Values will be added to those that have already been set when instantiating the client.

Also, there's additional support for the most commonly used headers, like *"If-None-Match", "If-Modified-Since", "Accept" and "Accept-Charset".*

Here's an example of how these values can be used:

```
1.  HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
2.    .accept(MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML)
3.    .acceptCharset(StandardCharsets.UTF_8)
4.    .ifNoneMatch("*")
5.    .ifModifiedSince(ZonedDateTime.now())
6.    .retrieve();
```

4.7. Getting a Response

The final stage is sending the request and receiving a response. We can achieve this by using either the *exchangeToMono/exchangeToFlux* or the *retrieve* method.

The *exchangeToMono* and *exchangeToFlux* methods allow access to the **ClientResponse**, along with its status and headers:

```
1.  Mono<String> response = headersSpec.exchangeToMono(response -> {
2.    if (response.statusCode().equals(HttpStatus.OK)) {
3.        return response.bodyToMono(String.class);
4.    } else if (response.statusCode().is4xxClientError()) {
5.        return Mono.just("Error response");
6.    } else {
7.        return response.createException()
8.          .flatMap(Mono::error);
9.    }
10. });
```

While the retrieve method is the shortest path to fetching a body directly:

```
Mono<String> response = headersSpec.retrieve()
  .bodyToMono(String.class);
```

It's important to pay attention to the *ResponseSpec.bodyToMono* method, which will throw a *WebClientException* if the status code is *4xx* (client error) or *5xx* (server error).

The WebTestClient is the main entry point for testing WebFlux server endpoints. It has a very similar API to the *WebClient*, and it delegates most of the work to an internal *WebClient* instance focusing mainly on providing a test context. The *DefaultWebTestClient* class is a single interface implementation.

The client for testing can be bound to a real server, or work with specific controllers or functions.

5.1. Binding to a Server

To complete end-to-end integration tests with actual requests to a running server, we can use the *bindToServer* method:

```
WebTestClient testClient = WebTestClient
  .bindToServer()
  .baseUrl("http://localhost:8080")
  .build();
```

5.2. Binding to a Router

We can test a particular RouterFunction by passing it to the *bindToRouterFunction* method:

```
RouterFunction function = RouterFunctions.route(
  RequestPredicates.GET("/resource"),
  request -> ServerResponse.ok().build()
);


WebTestClient
  .bindToRouterFunction(function)
  .build().get().uri("/resource")
  .exchange()
  .expectStatus().isOk()
  .expectBody().isEmpty();
```

5.3. Binding to a Web Handler

The same behavior can be achieved with the *bindToWebHandler* method, which takes a *WebHandler* instance:

```
1. WebHandler handler = exchange -> Mono.empty();
2. WebTestClient.bindToWebHandler(handler).build();
```

5.4. Binding to an Application Context

A more interesting situation occurs when we're using the *bindToApplicationContext* method. It takes an *ApplicationContext* and analyzes the context for controller beans and *@EnableWebFlux* configurations.

If we inject an instance of the ApplicationContext, a simple code snippet may look like this:

```
1. @Autowired
2. private ApplicationContext context;
3.
4. WebTestClient testClient = WebTestClient.
5. bindToApplicationContext(context)
6.    .build();
```

5.5. Binding to a Controller

A shorter approach would be providing an array of controllers we want to test with the *bindToController* method. Assuming we've got a *Controller* class, and we injected it into a needed class, we can write:

```
1. @Autowired
2. private Controller controller;
3.
4. WebTestClient testClient = WebTestClient.
5. bindToController(controller).build();
```

5.6. Making a Request

After building a *WebTestClient* object, all the following operations in the chain are going to be similar to the *WebClient* until the *exchange* method (one way to get a response), which provides the *WebTestClient.ResponseSpec* interface to work with useful methods, like the *expectStatus*, *expectBody*, and *expectHeader:*

```
WebTestClient
  .bindToServer()
    .baseUrl("http://localhost:8080")
    .build()
    .post()
    .uri("/resource")
  .exchange()
    .expectStatus().isCreated()
   .expectHeader().valueEquals("Content-Type", "application/json")
    .expectBody().jsonPath("field").isEqualTo("value");
```

In this chapter, we explored *WebClient*, a new enhanced Spring mechanism for making requests on the client-side.

We also looked at the benefits it provides by going through configuring the client, preparing the request, and processing the response.

All of the code snippets mentioned in the chapter can be found in our GitHub repository.

6. Spring WebClient vs. RestTemplate

In this chapter, we'll compare two of Spring's web client implementations: *RestTemplate*, and Spring 5's new reactive alternative *WebClient*.

It's a common requirement in web applications to make HTTP calls to other services. So we need a web client tool.

2.1. *RestTemplate* Blocking Client

For a long time, Spring has been offering *RestTemplate* as a web client abstraction. Under the hood, ***RestTemplate* uses the Java Servlet API, which is based on the thread-per-request model.**

This means that the thread will block until the web client receives the response. The problem with the blocking code is due to each thread consuming some amount of memory and CPU cycles.

Let's consider having a lot of incoming requests, which are waiting for some slow service needed to produce the result.

Sooner or later, the requests waiting for the results will pile up. **Consequently, the application will create many threads, which will exhaust the thread pool or occupy all the available memory**. We can also experience performance degradation because of the frequent CPU context (thread) switching.

2.2. *WebClient* Non-Blocking Client

On the other hand, ***WebClient* uses an asynchronous, non-blocking solution provided by the Spring Reactive framework**.

While *RestTemplate* uses the caller thread for each event (HTTP call), *WebClient* will create something like a "task" for each event. Behind the scenes, the Reactive framework will queue those "tasks," and execute them only when the appropriate response is available.

The Reactive framework uses an event-driven architecture. It provides the means to compose asynchronous logic through the Reactive Streams API. As a result, the reactive approach can process more logic while using fewer threads and system resources, compared to the synchronous/blocking method.

WebClient is part of the Spring WebFlux library. As such, **we can also write client code using a functional, fluent API with reactive types (*Mono and Flux*) as a declarative composition.**

To demonstrate the differences between these two approaches, we'd need to run performance tests with many concurrent client requests.

We would see a significant performance degradation with the blocking method after a certain number of parallel client requests.

However, the reactive/non-blocking method should give constant performances, regardless of the number of requests.

For this chapter, **we'll implement two REST endpoints, one using *RestTemplate* and the other using *WebClient***. Their task is to call another slow REST web service, which returns a list of tweets.

To start, we'll need the Spring Boot WebFlux starter dependency:

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

And here's our slow service REST endpoint:

```
@GetMapping("/slow-service-tweets")
private List<Tweet> getAllTweets() {
    Thread.sleep(2000L); // delay
    return Arrays.asList(
      new Tweet("RestTemplate rules", "@user1"),
      new Tweet("WebClient is better", "@user2"),
      new Tweet("OK, both are useful", "@user1"));
}
```

3.1. Using *RestTemplate* to Call a Slow Service

Now let's implement another REST endpoint that will call our slow service via the web client.

First, we'll use *RestTemplate*:

```
@GetMapping("/tweets-blocking")
public List<Tweet> getTweetsBlocking() {
    log.info("Starting BLOCKING Controller!");
    final String uri = getSlowServiceUri();


    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<Tweet>> response = restTemplate.exchange(
      uri, HttpMethod.GET, null,
      new ParameterizedTypeReference<List<Tweet>>(){});


    List<Tweet> result = response.getBody();
    result.forEach(tweet -> log.info(tweet.toString()));
    log.info("Exiting BLOCKING Controller!");
    return result;
}
```

When we call this endpoint, due to the synchronous nature of *RestTemplate*, the code will block waiting for the response from our slow service. The rest of the code in this method will be run only when the response has been received.

Here's what we'll see in the logs:

```
Starting BLOCKING Controller!
Tweet(text=RestTemplate rules, username=@user1)
Tweet(text=WebClient is better, username=@user2)
Tweet(text=OK, both are useful, username=@user1)
Exiting BLOCKING Controller!
```

3.2. Using *WebClient* to Call a Slow Service

Then let's use *WebClient* to call the slow service:

```
@GetMapping(value = "/tweets-non-blocking",
            produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<Tweet> getTweetsNonBlocking() {
    log.info("Starting NON-BLOCKING Controller!");
    Flux<Tweet> tweetFlux = WebClient.create()
      .get()
      .uri(getSlowServiceUri())
      .retrieve()
      .bodyToFlux(Tweet.class);


    tweetFlux.subscribe(tweet -> log.info(tweet.toString()));
    log.info("Exiting NON-BLOCKING Controller!");
    return tweetFlux;
}
```

In this case, *WebClient* returns a *Flux* publisher, and the method execution gets completed. Once the result is available, the publisher will start emitting tweets to its subscribers.

Note that a client (in this case, a web browser) calling this */tweets-non-blocking* endpoint will also be subscribed to the returned *Flux* object.

Let's observe the log this time:

```
Starting NON-BLOCKING Controller!
Exiting NON-BLOCKING Controller!
Tweet(text=RestTemplate rules, username=@user1)
Tweet(text=WebClient is better, username=@user2)
Tweet(text=OK, both are useful, username=@user1)
```

Note that this endpoint method completed before the response was received.

In this chapter, we explored two different ways of using web clients in Spring. *RestTemplate* uses Java Servlet API, and is therefore synchronous and blocking.

Conversely, WebClient is asynchronous and won't block the executing thread while waiting for the response to come back. **The notification will be produced only when the response is ready**.

RestTemplate will still be used. But in some cases, the non-blocking approach uses much fewer system resources compared to the blocking one, so *WebClient* is a preferable choice.

All of the code snippets mentioned in the chapter can be found over on GitHub.

7. Spring WebClient Requests With Parameters

1. Overview

A lot of frameworks and projects are introducing **reactive programming and asynchronous request handling**. As such, Spring 5 introduced a reactive WebClient implementation as part of the WebFlux framework.

In this chapter, we'll learn how to **reactively consume REST API endpoints with WebClient**.

To start, let's define a sample REST API with the **following GET endpoints**:

/products – get all products
/products/{id} – get product by ID
/products/{id}/attributes/{attributeId} – get product attribute by id
/products/?name={name}&deliveryDate={deliveryDate}&color={color} – find products
/products/?tag[]={tag1}&tag[]={tag2} – get products by tags
/products/?category={category1}&category={category2} – get products by categories

Here we defined a few different URIs. In just a moment, we'll figure out how to build and send each type of URI with WebClient.

Please note that the URIs for getting products by tags and categories contain arrays as query parameters; however, the syntax differs because **there's no strict definition of how arrays should be represented in URIs**. This primarily depends on the server-side implementation. Accordingly, we'll cover both cases.

First, we'll need to create an instance of *WebClient*. For this chapter, we'll be using a mocked object to verify that a valid URI is requested.

Let's define the client and related mock objects:

```
1.  exchangeFunction = mock(ExchangeFunction.class);
2.  ClientResponse mockResponse = mock(ClientResponse.class);
3.  when(mockResponse.bodyToMono(String.class))
4.    .thenReturn(Mono.just("test"));
5.
6.
7.  when(exchangeFunction.exchange(argumentCaptor.capture()))
8.    .thenReturn(Mono.just(mockResponse));
9.
10.
11. webClient = WebClient
12.   .builder()
13.   .baseUrl("https://example.com/api")
14.   .exchangeFunction(exchangeFunction)
15.   .build();
```

We'll also pass a base URL that will be prepended to all requests made by the client.

Finally, to verify that a particular URI has been passed to the underlying *ExchangeFunction* instance, we'll use the following helper method:

```
1.  private void verifyCalledUrl(String relativeUrl) {
2.      ClientRequest request = argumentCaptor.getValue();
3.         assertEquals(String.format("%s%s", BASE_URL, relativeUrl),
4.  request.url().toString());
5.
6.      verify(this.exchangeFunction).exchange(request);
7.      verifyNoMoreInteractions(this.exchangeFunction);
8.  }
```

The *WebClientBuilder* class has the *uri*() method that provides the *UriBuilder* instance as an argument. Generally, we make an API call in the following manner:

```
webClient.get()
  .uri(uriBuilder -> uriBuilder
    //... building a URI
    .build())
  .retrieve()
  .bodyToMono(String.class)
  .block();
```

We'll use *UriBuilder* extensively in this guide to construct URIs. It's worth noting that we can build a URI using other methods, and then just pass the generated URI as a String.

A path component consists of a sequence of path segments separated by a slash (/). First, we'll start with a simple case where a URI doesn't have any variable segments, /products:

```
1.  webClient.get()
2.    .uri("/products")
3.    .retrieve()
4.    .bodyToMono(String.class)
5.    .block();
6.
7.  verifyCalledUrl("/products");
```

In this case, we can just pass a String as an argument.

Next, we'll take the */products/{id}* endpoint and build the corresponding URI:

```
1.  webClient.get()
2.    .uri(uriBuilder - > uriBuilder
3.      .path("/products/{id}")
4.      .build(2))
5.    .retrieve()
6.    .bodyToMono(String.class)
7.    .block();
8.
9.
10. verifyCalledUrl("/products/2");
```

From the code above, we can see that the actual segment values are passed to the *build()* method.

In a similar way, we can create a URI with multiple path segments for the */products/{id}/attributes/{attributeId}* endpoint:

```
webClient.get()
  .uri(uriBuilder - > uriBuilder
    .path(“/products/{id}/attributes/{attributeId}”)
    .build(2, 13))
  .retrieve()
  .bodyToMono(String.class)
  .block();


verifyCalledUrl(“/products/2/attributes/13”);
```

A URI can have as many path segments as required, though the final URI length must not exceed limitations. Finally, we need to remember to keep the right order of actual segment values passed to the *build()* method.

Usually, a query parameter is a simple key-value pair like title=Baeldung. Let's see how to build such URIs.

5.1. Single Value Parameters

We'll start with single value parameters, and take the */products/?name={name}&deliveryDate={deliveryDate}&color={color}* endpoint. To set a query parameter, we'll call the *queryParam()* method of the *UriBuilder* interface:

```
webClient.get()
  .uri(uriBuilder - > uriBuilder
    .path("/products/")
    .queryParam("name", "AndroidPhone")
    .queryParam("color", "black")
    .queryParam("deliveryDate", "13/04/2019")
    .build())
  .retrieve()
  .bodyToMono(String.class)
  .block();


verifyCalledUrl("/
products/?name=AndroidPhone&color=black&deliveryDate=13/04/2019");
```

Here we added three query parameters and assigned actual values immediately. Conversely, it's also possible to leave placeholders instead of exact values:

```
webClient.get()
  .uri(uriBuilder - > uriBuilder
    .path(“/products/”)
    .queryParam(“name”, “{title}”)
    .queryParam(“color”, “{authorId}”)
    .queryParam(“deliveryDate”, “{date}”)
    .build(“AndroidPhone”, “black”, “13/04/2019”))
  .retrieve()
  .bodyToMono(String.class)
  .block();


verifyCalledUrl(“/
products/?name=AndroidPhone&color=black&deliveryDate=13%2F04%2F2019”);
```

This might be especially helpful when passing a builder object further in a chain.

Note that there's one important **difference between the two code snippets above**. With attention to the expected URIs, we can see that they're **encoded differently**. Particularly, the slash character (`/`) was escaped in the last example.

Generally speaking, RFC3986 doesn't require the encoding of slashes in the query; however, some server-side applications might require such conversion. Therefore, we'll see how to change this behavior later in this guide.

5.2. Array Parameters

We might need to pass an array of values, and there aren't strict rules for passing arrays in a query string. Therefore, **an array representation in a query string differs from project to project, and usually depends on underlying frameworks**. We'll cover the most widely used formats in this chapter.

Let's start with the */products/?tag[]={tag1}&tag[]={tag2} endpoint*:

```
webClient.get()
  .uri(uriBuilder - > uriBuilder
    .path("/products/")
    .queryParam("tag[]", "Snapdragon", "NFC")
    .build())
  .retrieve()
  .bodyToMono(String.class)
  .block();

verifyCalledUrl("/products/?tag%5B%5D=Snapdragon&tag%5B%5D=NFC");
```

As we can see, the final URI contains multiple tag parameters, followed by encoded square brackets. The *queryParam()* method accepts variable arguments as values, so there's no need to call the method several times.

Alternatively, we can **omit square brackets and just pass multiple query parameters with the same key**, but different values, */products/?category={category1}&category={category2}:*

```
webClient.get()
  .uri(uriBuilder - > uriBuilder
    .path("/products/")
    .queryParam("category", "Phones", "Tablets")
    .build())
  .retrieve()
  .bodyToMono(String.class)
  .block();


verifyCalledUrl("/products/?category=Phones&category=Tablets");
```

Finally, there's one more extensively-used method to encode an array, which is to pass comma-separated values. Let's transform our previous example into comma-separated values:

```
webClient.get()
  .uri(uriBuilder - > uriBuilder
    .path("/products/")
    .queryParam("category", String.join(",", "Phones", "Tablets"))
    .build())
  .retrieve()
  .bodyToMono(String.class)
  .block();


verifyCalledUrl("/products/?category=Phones,Tablets");
```

We're simply using the *join()* method of the *String* class to create a comma-separated string. We can also use any other delimiter that's expected by the application.

Remember how we previously mentioned URL encoding?

If the default behavior doesn't fit our requirements, we can change it. We need to provide a *UriBuilderFactory* implementation while building a *WebClient* instance. In this case, we'll use the *DefaultUriBuilderFactory* class. To set encoding, we'll call the *setEncodingMode()* method. The following modes are available:

- **TEMPLATE_AND_VALUES:** Pre-encode the URI template and strictly encode URI variables when expanded
- **VALUES_ONLY:** Do not encode the URI template, but strictly encode URI variables after expanding them into the template
- **URI_COMPONENTS:** Encode URI component value after expending URI variables
- **NONE:** No encoding will be applied

The default value is **TEMPLATE_AND_VALUES.** Let's set the mode to **URI_COMPONENTS**:

```
1.  DefaultUriBuilderFactory factory = new
2.  DefaultUriBuilderFactory(BASE_URL);
3.  factory.setEncodingMode(DefaultUriBuilderFactory.EncodingMode.URI_
4.  COMPONENT);
5.  webClient = WebClient
6.     .builder()
7.     .uriBuilderFactory(factory)
8.     .baseUrl(BASE_URL)
9.     .exchangeFunction(exchangeFunction)
10.    .build();
```

As a result, the following assertion will succeed:

```
DefaultUriBuilderFactory factory = new
DefaultUriBuilderFactory(BASE_URL);
factory.setEncodingMode(DefaultUriBuilderFactory.EncodingMode.URI_
COMPONENT);
webClient = WebClient
  .builder()
  .uriBuilderFactory(factory)
  .baseUrl(BASE_URL)
  .exchangeFunction(exchangeFunction)
  .build();
```

And, of course, we can provide a completely custom *UriBuilderFactory* implementation to handle URI creation manually.

In this chapter, we learned how to build different types of URIs using WebClient and *DefaultUriBuilder*.

Along the way, we covered various types and formats of query parameters. Finally, we wrapped up by changing the default encoding mode of the URL builder.

As always, all of the code snippets from the chapter are available over on GitHub repository.

8. Handling Errors in Spring WebFlux

In this chapter, **we'll explore various strategies available for handling errors in a Spring WebFlux project** while walking through a practical example.

We'll also point out where it might be advantageous to use one strategy over another, and provide a link to the full source code at the end.

The Maven setup is the same as our previous chapter, which provides an introduction to Spring WebFlux.

For our example, **we'll use a RESTful endpoint that takes a username as a query parameter, and returns "Hello username"** as a result.

First, we'll create a router function that routes the /hello request to a method named handleRequest in the passed-in handler:

```
@Bean
public RouterFunction<ServerResponse> routeRequest(Handler
handler) {
    return RouterFunctions.route(RequestPredicates.GET("/hello")
      .and(RequestPredicates.accept(MediaType.TEXT_PLAIN)),
        handler::handleRequest);
    }
```

Next, we'll define the handleRequest() method that calls the sayHello() method, and finds a way of including/returning its result in the ServerResponse body:

```
public Mono<ServerResponse> handleRequest(ServerRequest request) {
    return
      //...
        sayHello(request)
      //...
}
```

Finally, the sayHello() method is a simple utility method that concatenates the "Hello" String and the username:

```
private Mono<String> sayHello(ServerRequest request) {
    try {
        return Mono.just("Hello, " + request.queryParam("name").get());
    } catch (Exception e) {
        return Mono.error(e);
    }
}
```

As long as a username is present as part of our request, e.g., if the endpoint is called as *"/hello?username=Tonni"*, this endpoint will always function correctly.

However, ***if we call the same endpoint without specifying a username, e.g., "/hello", it will throw an exception***.

Below, we'll look at where and how we can reorganize our code to handle this exception in WebFlux.

There are two key operators built into the Mono and Flux APIs to handle errors at a functional level.

Let's briefly explore them and their usage.

3.1. Handling Errors With onErrorReturn

We can use *onErrorReturn()* to return a static default value whenever an error occurs:

```
1. public Mono<ServerResponse> handleRequest(ServerRequest request) {
2.     return sayHello(request)
3.       .onErrorReturn("Hello Stranger")
4.       .flatMap(s -> ServerResponse.ok()
5.         .contentType(MediaType.TEXT_PLAIN)
6.         .bodyValue(s));
7. }
```

Here we're returning a static "Hello Stranger" whenever the buggy concatenation function *sayHello()* throws an exception.

3.2. Handling Errors With onErrorResume

There are three ways that we can use *onErrorResume* to handle errors:

- compute a dynamic fallback value
- execute an alternative path with a fallback method
- catch, wrap, and re-throw an error, e.g., as a custom business exception

Let's see how we can compute a value:

```
public Mono<ServerResponse> handleRequest(ServerRequest request)
{
    return sayHello(request)
      .flatMap(s -> ServerResponse.ok()
        .contentType(MediaType.TEXT_PLAIN)
        .bodyValue(s))
      .onErrorResume(e -> Mono.just("Error " + e.getMessage())
        .flatMap(s -> ServerResponse.ok()
          .contentType(MediaType.TEXT_PLAIN)
          .bodyValue(s)));
}
```

Here we're returning a String consisting of the dynamically obtained error message appended to the string "Error" whenever *sayHello()* throws an exception.

Next, let's **call a fallback method when an error occurs:**

```
public Mono<ServerResponse> handleRequest(ServerRequest request)
{
    return sayHello(request)
      .flatMap(s -> ServerResponse.ok()
        .contentType(MediaType.TEXT_PLAIN)
        .bodyValue(s))
      .onErrorResume(e -> sayHelloFallback()
        .flatMap(s -> ServerResponse.ok()
        .contentType(MediaType.TEXT_PLAIN)
        .bodyValue(s)));
}
```

Here we're calling the alternative method *sayHelloFallback()* whenever *sayHello()* throws an exception.

The final option using onErrorResume() is to **catch, wrap, and re-throw an error,** e.g., as a *NameRequiredException:*

```
1.  public Mono<ServerResponse> handleRequest(ServerRequest request)
2.  {
3.      return ServerResponse.ok()
4.        .body(sayHello(request)
5.        .onErrorResume(e -> Mono.error(new NameRequiredException(
6.          HttpStatus.BAD_REQUEST,
7.          "username is required", e))), String.class);
8.  }
```

Here we're throwing a custom exception with the message "username is required" whenever *sayHello()* throws an exception.

So far, all of the examples we've presented have tackled error handling at a functional level.

However, we can opt to handle our WebFlux errors at a global level. To do this, we only need to take two steps:

- customize the Global Error Response Attributes
- implement the Global Error Handler

The exception that our handler throws will be automatically translated to an HTTP status and a JSON error body.

To customize these, we can simply extend the DefaultErrorAttributes class, and override its getErrorAttributes() method:

```
1.  public class GlobalErrorAttributes extends
2.  DefaultErrorAttributes{
3.
4.      @Override
5.      public Map<String, Object> getErrorAttributes(ServerRequest
6.  request,
7.        ErrorAttributeOptions options) {
8.          Map<String, Object> map = super.getErrorAttributes(
9.            request, options);
10          map.put("status", HttpStatus.BAD_REQUEST);
11.         map.put("message", "username is required");
12.         return map;
13.     }
14.
15.
16. }
```

Here we want the status: *BAD_REQUEST* and the message "username is required" returned as part of the error attributes when an exception occurs.

Next, we'll **implement the Global Error Handler.**

For this, Spring provides a convenient AbstractErrorWebExceptionHandler class for us to extend and implement in handling global errors:

```
@Component
@Order(-2)
public class GlobalErrorWebExceptionHandler extends
    AbstractErrorWebExceptionHandler {

    // constructors

    @Override
    protected RouterFunction<ServerResponse> getRoutingFunction(
      ErrorAttributes errorAttributes) {
        return RouterFunctions.route(
          RequestPredicates.all(), this::renderErrorResponse);
    }

    private Mono<ServerResponse> renderErrorResponse(
       ServerRequest request) {
       Map<String, Object> errorPropertiesMap =
getErrorAttributes(request,
         ErrorAttributeOptions.defaults());
       return ServerResponse.status(HttpStatus.BAD_REQUEST)
         .contentType(MediaType.APPLICATION_JSON)
         .body(BodyInserters.fromValue(errorPropertiesMap));
    }
}
```

In this example, we set the order of our global error handler to -2. This is to **give it a higher priority than the *DefaultErrorWebExceptionHandler***, which is registered at @Order(-1).

The *errorAttributes* object will be the exact copy of the one that we pass in the Web Exception Handler's constructor. This should ideally be our customized Error Attributes class.

Then we're clearly stating that we want to route all error handling requests to the *renderErrorResponse()* method.

Finally, we'll get the error attributes and insert them inside a server response body.

This then produces a JSON response with details of the error, the HTTP status, and the exception message for machine clients. For browser clients, it has a "white-label" error handler that renders the same data in HTML format. Of course, this can be customized.

In this chapter, we focused on various strategies available for handling errors in a Spring WebFlux project, and pointed out where it might be advantageous to use one strategy over another.

As promised, the full source code that accompanies the chapter is available over on GitHub.

9. Spring Security 5 for Reactive Applications

In this chapter, we'll explore the new features of the Spring Security 5 framework for securing reactive applications. This release is aligned with Spring 5 and Spring Boot 2.

We won't go into details here about the reactive applications themselves, which are a new feature of the Spring 5 framework, so be sure to check out the chapter Intro to Reactor Core for more details.

We'll use Spring Boot starters to bootstrap our project together with all the required dependencies.

The basic setup requires a parent declaration, web starter, and security starter dependencies. We'll also need the Spring Security test framework:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.1</version>
    <relativePath/>
</parent>


<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

We can check out the current version of the Spring Boot security starter over at Maven Central.

3.1. Bootstrapping the Reactive Application

We won't use the standard *@SpringBootApplication* configuration, and instead we'll configure a **Netty-based web server. Netty is an asynchronous NIO-based framework that's a good foundation for reactive applications.**

The *@EnableWebFlux* annotation enables the standard Spring Web Reactive configuration for the application:

```
1.  @ComponentScan(basePackages = {"com.baeldung.security"})
2.  @EnableWebFlux
3.  public class SpringSecurity5Application {
4.
5.
6.      public static void main(String[] args) {
7.          try (AnnotationConfigApplicationContext context
8.           = new AnnotationConfigApplicationContext(
9.              SpringSecurity5Application.class)) {
10.
11.             context.getBean(NettyContext.class).onClose().block();
12.         }
13.     }
```

Here we'll create a new application context, and wait for Netty to shut down by calling the *.onClose().block()* chain on the Netty context.

After Netty is shut down, the context will be automatically closed using the *try-with-resources block*.

We'll also need to create a Netty-based HTTP server, a handler for the HTTP requests, and the adapter between the server and the handler:

```
@Bean
public NettyContext nettyContext(ApplicationContext context) {
    HttpHandler handler = WebHttpHandlerBuilder
      .applicationContext(context).build();
    ReactorHttpHandlerAdapter adapter
      = new ReactorHttpHandlerAdapter(handler);
    HttpServer httpServer = HttpServer.create("localhost", 8080);
    return httpServer.newHandler(adapter).block();
}
```

3.2. Spring Security Configuration Class

For our basic Spring Security configuration, we'll create a configuration class, *SecurityConfig*.

To enable WebFlux support in Spring Security 5, we only need to specify the *@EnableWebFluxSecurity* annotation:

```
@EnableWebFluxSecurity
public class SecurityConfig {
    // ...
}
```

Now we can take advantage of the *ServerHttpSecurity* class to build our security configuration.

This class is a new feature of Spring 5. It's similar to HttpSecurity builder, but it's only enabled for WebFlux applications.

The *ServerHttpSecurity* is already preconfigured with some sane defaults, so we could skip this configuration completely. But for starters, we'll provide the following minimal config:

```
@Bean
public SecurityWebFilterChain securityWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
      .anyExchange().authenticated()
      .and().build();
}
```

Also, we'll need a user details service. Spring Security provides us with a convenient mock user builder and an in-memory implementation of the user details service:

```
@Bean
public MapReactiveUserDetailsService userDetailsService() {
    UserDetails user = User
      .withUsername("user")
      .password(passwordEncoder().encode("password"))
      .roles("USER")
      .build();
    return new MapReactiveUserDetailsService(user);
}
```

Since we're in reactive land, the user details service should also be reactive. If we check out the *ReactiveUserDetailsService* interface, **we'll see that its *findByUsername* method actually returns a *Mono publisher***:

```
public interface ReactiveUserDetailsService {


    Mono<UserDetails> findByUsername(String username);
}
```

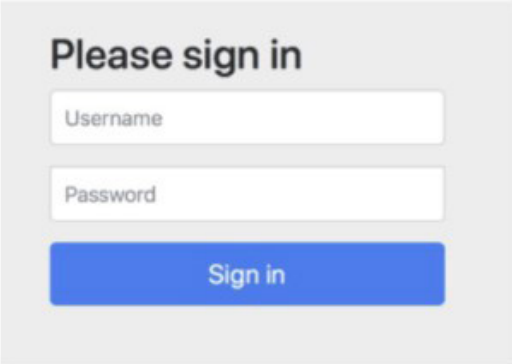
Now we can run our application and observe a regular HTTP basic authentication form.

A small, but striking improvement in Spring Security 5 is a new styled login form that uses the Bootstrap 4 CSS framework. The stylesheets in the login form link to CDN, so we'll only see the improvement when connected to the Internet.

To use the new login form, let's add the corresponding *formLogin()* builder method to the *ServerHttpSecurity* builder:

```
1.  public SecurityWebFilterChain securityWebFilterChain(
2.    ServerHttpSecurity http) {
3.      return http.authorizeExchange()
4.        .anyExchange().authenticated()
5.        .and().formLogin()
6.        .and().build();
7.  }
```

Now if we open the main page of the application, we'll see that it looks much better than the default form we're used to in previous versions of Spring Security:



Note that this isn't a production-ready form, but it's a good bootstrap of our application.

If we now log in and go to the http://localhost:8080/logout URL, we'll see the logout confirmation form, which is also styled.

To see something behind the authentication form, we'll implement a simple reactive controller that greets the user:

```
@RestController
public class GreetingController {


    @GetMapping("/")
    public Mono<String> greet(Mono<Principal> principal) {
        return principal
          .map(Principal::getName)
          .map(name -> String.format("Hello, %s", name));
    }


}
```

After logging in, we'll see the greeting. Then we'll add another reactive handler that will be accessible by admin only:

```
@GetMapping("/admin")
public Mono<String> greetAdmin(Mono<Principal> principal) {
    return principal
      .map(Principal::getName)
      .map(name -> String.format("Admin access: %s", name));
}
```

Next, we'll create a second user with the role *ADMIN*: in our user details service:

```
UserDetails admin = User.withDefaultPasswordEncoder()

  .username("admin")
  .password("password")
  .roles("ADMIN")
  .build();
```

We can now add a matcher rule for the admin URL that requires the user to have the *ROLE_ADMIN* authority.

Note that we have to put matchers before the *.anyExchange()* chain call. This call applies to all other URLs that were not yet covered by other matchers:

```
return http.authorizeExchange()

  .pathMatchers(“/admin”).hasAuthority(“ROLE_ADMIN”)
  .anyExchange().authenticated()
  .and().formLogin()
  .and().build();
```

Now if we log in with *user* or admin, we'll see that they both observe the initial greeting, as we've made it accessible for all authenticated users.

But only the *admin* user can go to the http://localhost:8080/admin URL and see their greeting.

We've seen how we can secure the URLs, but what about methods?

To enable method-based security for reactive methods, we only need to add the *@EnableReactiveMethodSecurity* annotation to our *SecurityConfig* class:

```
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity

public class SecurityConfig {
    // ...
}
```

Now we'll create a reactive greeting service with the following content:

```
@Service
public class GreetingService {


    public Mono<String> greet() {
        return Mono.just("Hello from service!");
    }
}
```

We can inject it into the controller, go to http://localhost:8080/greetingService and see that it actually works:

```
@RestController
public class GreetingController {


    private GreetingService greetingService


    // constructor...


    @GetMapping("/greetingService")
    public Mono<String> greetingService() {
        return greetingService.greet();
    }
```

But if we now add the *@PreAuthorize* annotation on the service method with the *ADMIN* role, then the greet service URL won't be accessible to a regular user:

```
@Service
public class GreetingService {


    @PreAuthorize(“hasRole(‘ADMIN’)”)
    public Mono<String> greet() {
        // ...
    }
}
```

Let's check out how easy it is to test our reactive Spring application.

First, we'll create a test with an injected application context:

```
@ContextConfiguration(classes = SpringSecurity5Application.class)
public class SecurityTest {


    @Autowired
    ApplicationContext context;


    // ...
}
```

Now we'll set up a simple reactive web test client, which is a feature of the Spring 5 test framework:

```
@Before
public void setup() {
    this.webTestClient = WebTestClient
      .bindToApplicationContext(this.context)
      .configureClient()
      .build();
}
```

If we now add the *@WithMockUser* annotation to a test method, we can provide an authenticated user for this method.

The login and password of this user will be user and *password*, respectively, and the role is *USER*. This, of course, can all be configured with the *@WithMockUser* annotation parameters.

Now we can check that the authorized user sees the greeting:

```
@Test
@WithMockUser
void whenHasCredentials_thenSeesGreeting() {
    webTestClient.get()
      .uri("/")
      .exchange()
      .expectStatus().isOk()
      .expectBody(String.class).isEqualTo("Hello, user");
}
```

The *@WithMockUser* annotation has been available since Spring Security 4; however, it was also updated in Spring Security 5 to cover reactive endpoints and methods.

In this chapter, we discovered the new features of the upcoming Spring Security 5 release, especially in the reactive programming arena.

As always, the source code for the chapter is available over on GitHub.

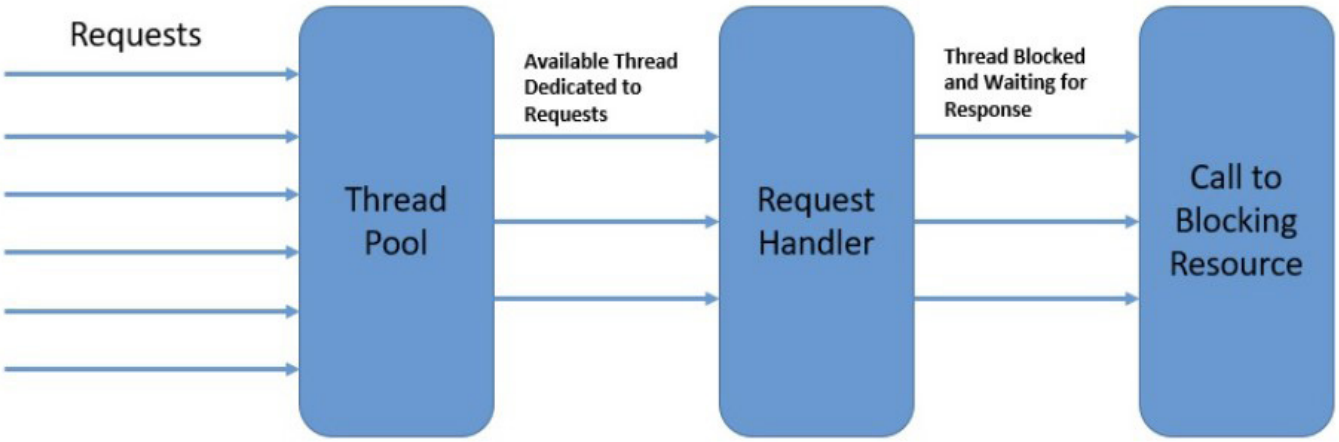
10. Concurrency in Spring WebFlux

In this chapter, we'll explore concurrency in reactive programs written with Spring WebFlux.

We'll begin by discussing concurrency in relation to reactive programming. Then we'll learn how Spring WebFlux offers concurrency abstractions over different reactive server libraries.

A typical web application comprises several complex, interacting parts. Many of these interactions are blocking in nature, such as those involving a database call to fetch or update data. Several others, however, are independent and can be performed concurrently, possibly in parallel.

For instance, two user requests to a web server can be handled by different threads. On a multi-core platform, this has an obvious benefit in terms of the overall response time. Thus, this model of concurrency is known as the thread-per-request model:



In the diagram above, each thread handles a single request at a time.

While thread-based concurrency solves a part of the problem for us, it does nothing to address the fact that most of our interactions within a single thread are still blocking. Moreover, the native threads we use to achieve concurrency in Java come at a significant cost in terms of context switches.

Meanwhile, as web applications face more and more requests, the *thread-per-request model* starts to fall short of expectations.

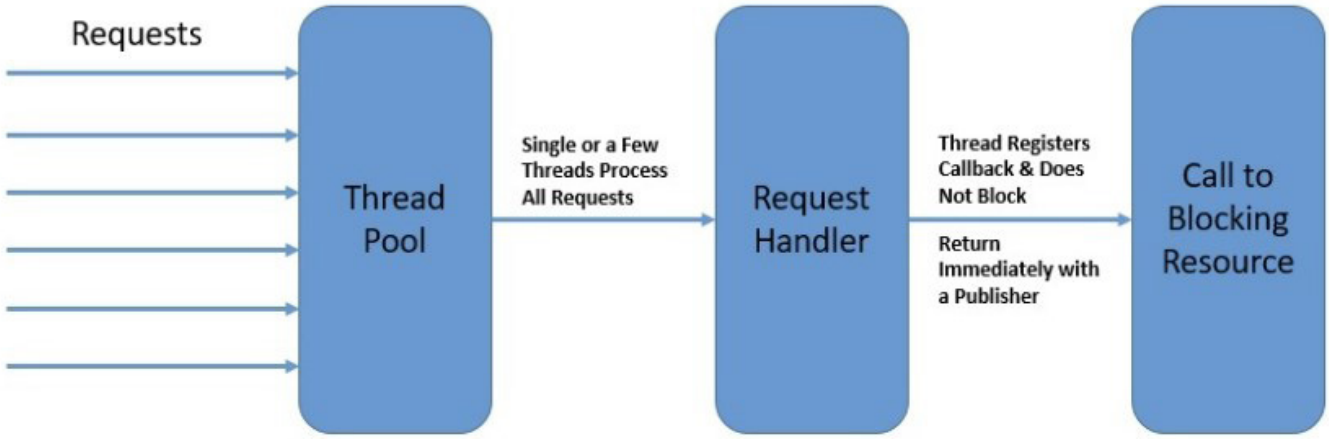
Consequently, we need a concurrency model that can help us handle increasingly more requests with a relatively fewer number of threads. This is one of the primary motivations for adopting reactive programming.

Reactive programming helps us structure the program in terms of data flows, and the propagation of change through them. In a completely non-blocking environment, this can enable us to achieve higher concurrency with better resource utilization.

However, is reactive programming a complete departure from thread-based concurrency? While this is a strong statement to make, reactive programming certainly has a very different approach to the usage of threads to achieve concurrency. So the fundamental difference that reactive programming brings on is asynchronicity.

In other words, the program flow transforms from a sequence of synchronous operations into an asynchronous stream of events.

For instance, under the reactive model, a read call to the database doesn't block the calling thread while data is fetched. The call immediately returns a publisher that others can subscribe to. The subscriber can process the event after it occurs, and may even further generate events itself:



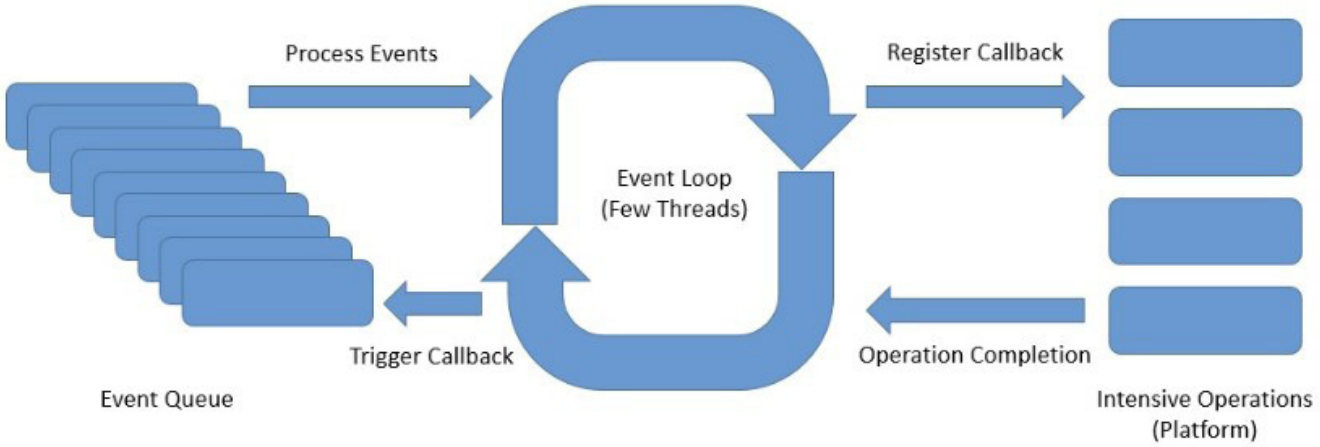
Above all, reactive programming doesn't emphasize which thread events should be generated and consumed. Rather, the emphasis is on structuring the program as an asynchronous event stream.

The publisher and subscriber here don't need to be part of the same thread. This helps us to achieve better utilization of the available threads, and therefore, higher overall concurrency.

There are several programming models that describe a reactive approach to concurrency.

In this section, we'll examine a few of them to understand how reactive programming achieves higher concurrency with fewer threads.

One such reactive asynchronous programming model for servers is the *event loop model:*



Above is an abstract design of an event loop that presents the ideas of reactive asynchronous programming:

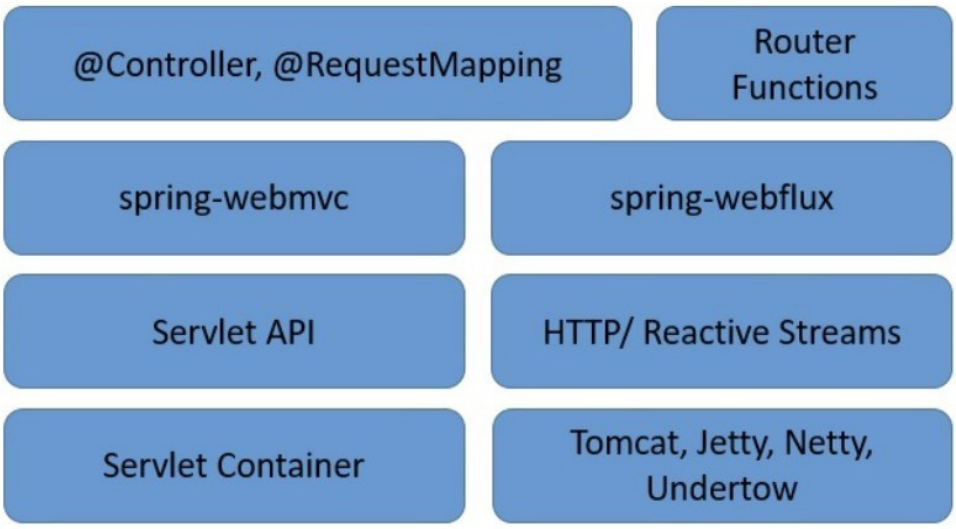
- **The event loop runs continuously in a single thread,** although we can have as many event loops as the number of available cores.
- **The event loop processes the events from an event queue sequentially, and returns immediately** after registering the callback with the platform.
- The platform can trigger the completion of an operation, like a database call or an external service invocation.
- **The event loop can trigger the callback on the operation completion notification, and send back the result to the original caller.**

The event loop model is implemented in a number of platforms, including Node.js, Netty, and Ngnix. They offer much better scalability than traditional platforms, like Apache HTTP Server, Tomcat, or JBoss.

Now we have enough insight into reactive programming and its concurrency model to explore the subject in Spring WebFlux.

WebFlux is Spring's reactive-stack web framework, which was added in version 5.0.

Let's explore the server-side stack of Spring WebFlux to understand how it complements the traditional web stack in Spring:



As we can see, Spring WebFlux sits parallel to the traditional web framework in Spring, and doesn't necessarily replace it.

There are a few important points to note here:

- Spring WebFlux extends the traditional annotation-based programming model with functional routing.
- It adapts the underlying HTTP runtimes to the Reactive Streams API, making the runtimes interoperable.
- It's able to support a wide variety of reactive runtimes, including Servlet 3.1+ containers, like Tomcat, Reactor, Netty, or Undertow.
- It includes WebClient, a reactive and non-blocking client for HTTP requests offering functional and fluent APIs.

As we discussed earlier, **reactive programs tend to work with just a few threads** and make the most of them. However, the number and nature of threads depends upon the actual Reactive Stream API runtime that we choose.

To clarify, **Spring WebFlux can adapt to different runtimes through a common API provided by *HttpHandler*.** This API is a simple contract with just one method that provides an abstraction over different server APIs, like Reactor Netty, Servlet 3.1 API, or Undertow APIs.

Let's examine the threading model implemented in a few of them.

While Netty is the default server in a WebFlux application, it's just a matter of declaring the right dependency to switch to any other supported server:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-reactor-netty</
artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
```

While it's possible to observe the threads created in a Java Virtual Machine in a number of ways, it's quite easy to just pull them from the Thread class itself:

```
Thread.getAllStackTraces()
  .keySet()
  .stream()
  .collect(Collectors.toList());
```

6.1. Reactor Netty

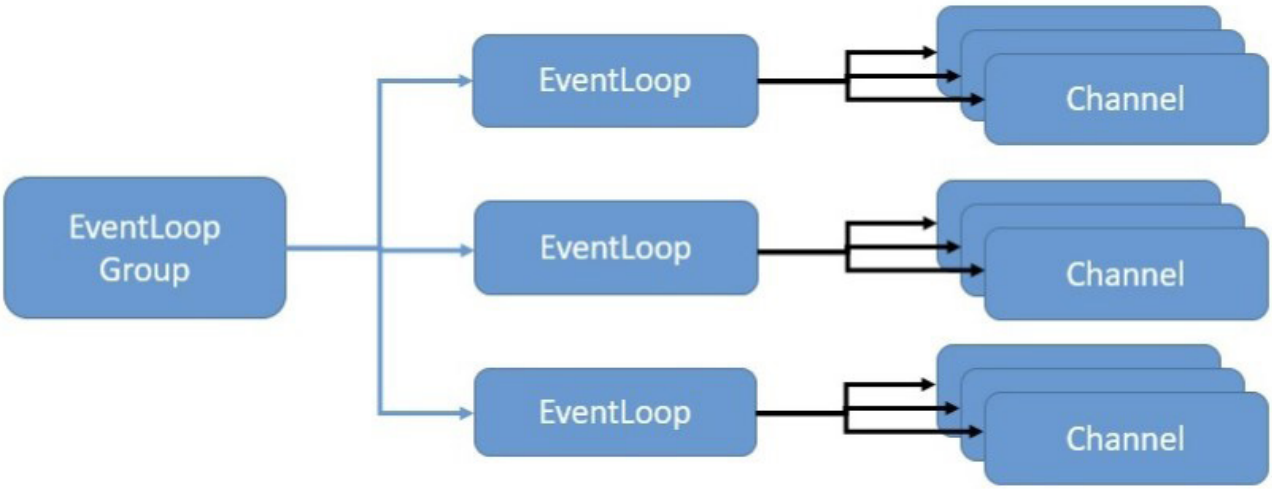
As we said, Reactor Netty is the default embedded server in the Spring Boot WebFlux starter. Let's see the threads that Netty creates by default. To begin, we won't add any other dependencies or use WebClient. So if we start a Spring WebFlux application created using its SpringBoot starter, we can expect to see some default threads it creates:

Thread Name	State	Type
server	WAITING	Normal
reactor-http-nio-1	RUNNABLE	Daemon
reactor-http-nio-2	RUNNABLE	Daemon
reactor-http-nio-3	RUNNABLE	Daemon

Threads in WebFlux Application on Netty

Note that, apart from a normal thread for the server, Netty spawns a bunch of worker threads for request processing. These are typically available CPU cores. This is the output on a quad-core machine. We'd also see a bunch of housekeeping threads typical to a JVM environment, but they aren't important here.

Netty uses the event loop model to provide highly scalable concurrency in a reactive asynchronous manner. Let's see how Netty implements an event loop levering Java NIO to provide this scalability:



Here, *EventLoopGroup* manages one or more *EventLoop*, which must be continuously running. Therefore, it isn't recommended to create more *EventLoops* than the number of available cores.

The *EventLoopGroup* further assigns an *EventLoop* to each newly created *Channel*. Thus, for the lifetime of a *Channel*, all operations are executed by the same thread.

6.2. Apache Tomcat

Spring WebFlux is also supported on a traditional Servlet Container, like Apache Tomcat.

WebFlux relies on the Servlet 3.1 API with non-blocking I/O. While it uses Servlet API behind a low-level adapter, Servlet API isn't available for direct usage.

Let's see what kind of threads we expect in a WebFlux application running on Tomcat:

Thread Name	State	Type
container-0	TIMED_WAITING	Normal
http-nio-8080-exec-1	RUNNABLE	Daemon
http-nio-8080-exec-2	WAITING	Daemon
http-nio-8080-exec-3	WAITING	Daemon
http-nio-8080-exec-4	WAITING	Daemon
http-nio-8080-exec-5	WAITING	Daemon
http-nio-8080-exec-6	WAITING	Daemon
http-nio-8080-exec-7	WAITING	Daemon
http-nio-8080-exec-8	WAITING	Daemon
http-nio-8080-exec-9	WAITING	Daemon
http-nio-8080-exec-10	WAITING	Daemon
http-nio-8080-ClientPoller	RUNNABLE	Daemon
http-nio-8080-Acceptor	RUNNABLE	Daemon
http-nio-8080-BlockPoller	RUNNABLE	Daemon

Threads in WebFlux Application on Tomcat

The number and type of threads that we can see here are quite different from what we observed earlier

.

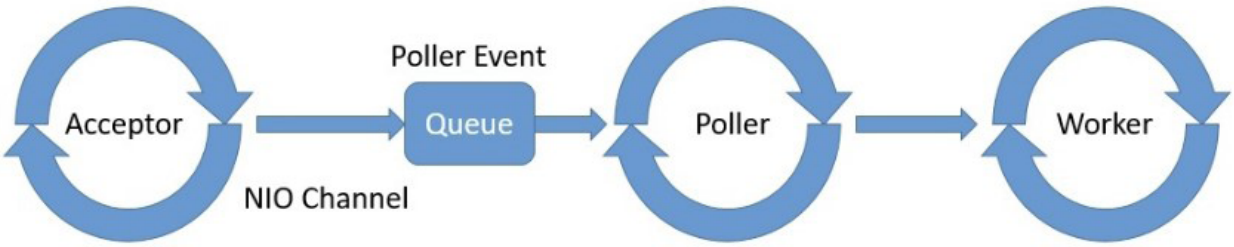
To begin with, Tomcat starts with more worker threads, which defaults to ten. Of course, we'll also see some housekeeping threads typical to the JVM, and the Catalina container, which we can ignore for this discussion.

We need to understand the architecture of Tomcat with Java NIO to correlate it with the threads we see above.

Tomcat 5 and onward supports NIO in its Connector component, which is primarily responsible for receiving the requests.

The other Tomcat component is the Container component, which is responsible for the container management functions.

The point of interest for us here is the threading model that the Connector component implements to support NIO. It's comprised of Acceptor, Poller, and Worker as part of the NioEndpoint module:



Tomcat spawns one or more threads for *Acceptor*, *Poller*, and *Worker,* typically with a thread pool dedicated to *Worker*.

While a detailed discussion on Tomcat architecture is beyond the scope of this chapter, we should now have enough insight to understand the threads we saw earlier.

Tomcat spawns one or more threads for *Acceptor*, *Poller*, and *Worker*, typically with a thread pool dedicated to Worker.

While a detailed discussion on Tomcat architecture is beyond the scope of this chapter, we should now have enough insight to understand the threads we saw earlier.

WebClient is the reactive HTTP client that's part of Spring WebFlux. We can use it anytime we require REST-based communication, which enables us to create applications that are end-to-end reactive.

As we've seen before, reactive applications work with just a few threads, so there's no margin for any part of the application to block a thread. Therefore, WebClient plays a vital role in helping us realize the potential of WebFlux.

7.1. Using WebClient

Using WebClient is quite simple as well. **We don't need to include any specific dependencies, as it's part of Spring WebFlux**.

Let's create a simple REST endpoint that returns a *Mono*:

```
1. @GetMapping("/index")
2. public Mono<String> getIndex() {
3.     return Mono.just("Hello World!");
4. }
```

Then we'll use WebClient to call this REST endpoint and consume the data reactively:

```
1. WebClient.create("http://localhost:8080/index").get()
2.   .retrieve()
3.   .bodyToMono(String.class)
4.   .doOnNext(s -> printThreads());
```

Here we're also printing the threads that are created using the method we discussed earlier.

7.2. Understanding the Threading Model

So how does the threading model work in the case of WebClient?

Well, not surprisingly, WebClient also implements concurrency using the event loop model. Of course, it relies on the underlying runtime to provide the necessary infrastructure.

If we're running WebClient on the Reactor Netty, it shares the event loop that Netty uses for the server. Therefore, in this case, we may not notice much difference in the threads that are created.

WebClient is also supported on a Servlet 3.1+ container, like Jetty, but the way it works there is different.

If we compare the threads that are created on a WebFlux application running Jetty with and without WebClient, we'll notice a few additional threads.

Here, WebClient has to create its event loop. So we can see the fixed number of processing threads that this event loop creates:

Thread Name	State	Type
HttpClient@7a860e9a-146	RUNNABLE	Normal
HttpClient@7a860e9a-144	RUNNABLE	Normal
HttpClient@7a860e9a-142	RUNNABLE	Normal
HttpClient@7a860e9a-145	TIMED_WAITING	Normal
HttpClient@7a860e9a-148	TIMED_WAITING	Normal
HttpClient@7a860e9a-149	TIMED_WAITING	Normal
HttpClient@7a860e9a-147	TIMED_WAITING	Normal
HttpClient@7a860e9a-143	RUNNABLE	Normal

Threads in WebFlux Application on Jetty with WebClient

In some cases, having a separate thread pool for client and server can provide better performance. While it's not the default behavior with Netty, it's always possible to declare a dedicated thread pool for *WebClient* if needed.

We'll see how this is possible in a later section.

As we saw earlier, even a simple application usually consists of several parts that need to be connected.

Typical examples of these parts include databases and message brokers. The existing libraries to connect with many of them are still blocking, but that's quickly changing.

There are several databases now that offer reactive libraries for connectivity. Many of these libraries are available within Spring Data, while we can use others directly as well.

The threading model these libraries use is of particular interest to us.

8.1. Spring Data MongoDB

Spring Data MongoDB provides reactive repository support for MongoDB built on top of the MongoDB Reactive Streams driver. Most notably, this driver **fully implements the Reactive Streams API to provide asynchronous stream processing** with *non-blocking back-pressure.*

Setting up support for the reactive repository for MongoDB in a Spring Boot application is as simple as adding a dependency:

```
1. <dependency>
2.     <groupId>org.springframework.boot</groupId>
3.     <artifactId>spring-boot-starter-data-mongodb-reactive</
4. artifactId>
5. </dependency>
```

This will allow us to create a repository, and use it to perform some basic operations on MongoDB in a non-blocking manner:

```
public interface PersonRepository extends
ReactiveMongoRepository<Person, ObjectId> {
}
.....
personRepository.findAll().doOnComplete(this::printThreads);
```

So what kind of threads can we expect to see when we run this application on the Netty server?

Well, not surprisingly, we won't see much difference, as a Spring Data reactive repository makes use of the same event loop that's available for the server.

8.2. Reactor Kafka

Spring is still in the process of building full-fledged support for reactive Kafka. However, we do have options available outside Spring.

Reactor Kafka is a reactive API for Kafka based on Reactor. Reactor Kafka enables messages to be published and consumed using functional APIs, also with non-blocking back-pressure.

First, we need to add the required dependency in our application to start using Reactor Kafka:

```
<dependency>
    <groupId>io.projectreactor.kafka</groupId>
    <artifactId>reactor-kafka</artifactId>
    <version>1.3.10</version>
</dependency>
```

This should enable us to produce messages to Kafka in a non-blocking manner:

```
// producerProps: Map of Standard Kafka Producer Configurations
SenderOptions<Integer, String> senderOptions = SenderOptions.
create(producerProps);
KafkaSender<Integer, String> sender =  KafkaSender.
create(senderOptions);
Flux<SenderRecord<Integer, String, Integer>> outboundFlux = Flux
  .range(1, 10)
  .map(i -> SenderRecord.create(new ProducerRecord<>("reactive-
test", i, "Message_" + i), i));
sender.send(outboundFlux).subscribe();
```

Similarly, we should be able to consume messages from Kafka, also in a non-blocking manner:

```
// consumerProps: Map of Standard Kafka Consumer Configurations
ReceiverOptions<Integer, String> receiverOptions =
ReceiverOptions.create(consumerProps);
receiverOptions.subscription(Collections.singleton("reactive-
test"));
KafkaReceiver<Integer, String> receiver = KafkaReceiver.
create(receiverOptions);
Flux<ReceiverRecord<Integer, String>> inboundFlux = receiver.
receive();
inboundFlux.doOnComplete(this::printThreads)
```

This is pretty simple and self-explanatory.

We're subscribing to a topic, *reactive-test*, in Kafka and getting a *Flux* of messages.

The **interesting thing for us is the threads that get created:**

Thread	State	Type
kafka-producer-network-thread | producer-1	RUNNABLE	Daemon
single-1	WAITING	Daemon
reactive-kafka-my-group-1	BLOCKED	Daemon
parallel-1	TIMED_WAITING	Daemon
parallel-2	WAITING	Daemon

Threads in WebFlux Application on Netty with Reactor Kafka

We can see a few threads that aren't typical to the Netty server.

This indicates that Reactor Kafka manages its own thread pool, with a few worker threads that participate in Kafka message processing exclusively. Of course, we'll see a bunch of other threads related to Netty and the JVM that we can ignore

Kafka producers use a separate network thread for sending requests to the broker. Furthermore, they deliver responses to the application on a *single-threaded pooled scheduler.*

Kafka consumer, on the other hand, has one thread per consumer group that blocks to listen for incoming messages. The incoming messages are then scheduled for processing on a different thread pool.

.

So far, we've seen that **reactive programming really shines in a completely non-blocking environment with just a few threads.** But this also means that, if there is indeed a part that's blocking, it will result in far worse performance. This is because a blocking operation can freeze the event loop entirely.

So **how do we handle long-running processes or blocking operations in reactive programming?**

Honestly, the best option would be to just avoid them. However, this may not always be possible, and **we may need a dedicated scheduling strategy for those parts of our application.**

Spring *WebFlux* **offers a mechanism to switch processing to a different thread pool in between a data flow chain.** This can provide us with precise control over the scheduling strategy that we want for certain tasks. Of course, *WebFlux* is able to offer this based on the thread pool abstractions, known as schedulers, available in the underlying reactive libraries.

9.1. Reactor

In Reactor, the **Scheduler class defines the execution model, as well as where the execution takes place.**

The Schedulers class provides a number of execution contexts, like *immediate, single, elastic,* and *parallel*. These provide different types of thread pools, which can be useful for different jobs. Moreover, we can always create our own Scheduler with a preexisting ExecutorService.

While *Schedulers* give us several execution contexts, Reactor **also provides us with different ways to switch the execution context**. These are the methods *publishOn* and *subscribeOn*.

We can use *publishOn* with a Scheduler anywhere in the chain, with that Scheduler affecting all the subsequent operators.

We can use *publishOn* with a *Scheduler* anywhere in the chain, with that *Scheduler* affecting all the subsequent operators.

While we can also use *subscribeOn* with a *Scheduler* anywhere in the chain, it will only affect the context of the source of emission.

If we recall, *WebClient* on Netty shares the same *event loop* created for the server as a default behavior. However, we may have valid reasons to create a dedicated thread pool for WebClient.

Let's see how we can achieve this in Reactor, which is the default reactive library in WebFlux:

```
Scheduler scheduler = Schedulers.newBoundedElastic(5, 10,
“MyThreadGroup”);


WebClient.create(“http://localhost:8080/index”).get()
  .retrieve()
  .bodyToMono(String.class)
  .publishOn(scheduler)
  .doOnNext(s -> printThreads());
```

Earlier, we didn't observe any difference in the threads created on Netty with or without *WebClient*. However, if we now run the code above, **we'll observe a few new threads being created**:

Thread	State	Type
boundedElastic-evictor-1	TIMED_WAITING	Daemon
MyThreadGroup-1	RUNNABLE	Daemon

Threads in WebFlux Application on Netty with WebClient with Scheduler

Here we can see the threads created as part of our bounded elastic thread pool. This is where responses from the WebClient are published once subscribed.

This leaves the main thread pool for handling the server requests.

9.2. RxJava

The **default behavior in RxJava isn't very different from that of the Reactor.**

The *Observable*, and the chain of operators we apply on it, do their work and notify the observers on the same thread where the subscription was invoked. Also, RxJava, like Reactor, offers ways to introduce prefixed or custom scheduling strategies into the chain.

RxJava also features a *Schedulers* class, which offers a number of execution models for the *Observable* chain. These include *new thread, immediate, trampoline, io, computation,* and *test*. Of course, it also allows us to define a *Scheduler* from a *Java Executor*.

Moreover, RxJava also **offers two extension methods to achieve this**, *subscribeOn* and *observeOn*.

The subscribeOn method changes the default behavior by specifying a different Scheduler on which Observable should operate. The observeOn method, on the other hand, specifies a different Scheduler that the Observable can use to send notifications to the observers.

As we discussed before, Spring WebFlux uses Reactor as its reactive library by default. But since it's fully compatible with Reactive Streams API, it's **possible to switch to another Reactive Streams implementation, like RxJava** (for RxJava 1.x with its Reactive Streams adapter).

To do so, we need to explicitly add the dependency:

```
<dependency>
    <groupId>io.reactivex.rxjava2</groupId>
    <artifactId>rxjava</artifactId>
    <version>2.2.21</version>
</dependency>
```

Then we can start to use RxJava types, like Observable, in our application, along with RxJava specific Schedulers:

```
1. io.reactivex.Observable
2.   .fromIterable(Arrays.asList("Tom", "Sawyer"))
3.   .map(s -> s.toUpperCase())
4.   .observeOn(io.reactivex.schedulers.Schedulers.trampoline())
5.   .doOnComplete(this::printThreads);
```

Now if we run this application, apart from the regular Netty and JVM related threads, **we should see a few threads related to our RxJava *Scheduler***:

Thread	State	Type
RxCachedWorkerPoolEvictor-1	TIMED_WAITING	Daemon
RxSchedulerPurge-1	TIMED_WAITING	Daemon

Threads in WebFlux Application on Netty with RxJava Scheduler

In this chapter, we explored the premise of reactive programming from the context of concurrency. We observed the difference in the concurrency model between traditional and reactive programming. This allowed us to examine the concurrency model in Spring WebFlux, and its take on the threading model to achieve it.

Then we explored the threading model in WebFlux in combination with different HTTP runtime and reactive libraries. We also learned how the threading model differs when we use WebClient versus a data access library.

Finally, we touched upon the options for controlling the scheduling strategy in our reactive program within WebFlux.

As always, the source code for this chapter can be found over on GitHub.