20

Reactive Camel

This bonus online chapters covers

- First steps with Reactive Streams
- Using Reactive Streams with Camel
- Using Eclipse Vert.x with Camel

If you read this book in chronological order, you've been on a long journey. This is the first of two bonus online-only chapters from the hands of Claus and Jonathan, and this time we'll keep it short. We have only two topics we want to bring to your attention here at the end.

Apache Camel is a well-established project that's been around for over a decade. A decade in the IT industry is like half a lifetime for humans. Only recently has reactive programming started to gain more interest, especially since Java 8 added support for `java.util.stream` in its streaming API. Further interest may be spurred with the upcoming Spring Framework 5, which now includes a reactive engine.

As for Apache Camel, the current architecture of Camel 2.*x* is based on a hybrid routing engine that executes both blocking and nonblocking processing, depending on which EIPs and components are being used. The upcoming Camel 3.*x* architecture is intended to be a dual engine comprising the current hybrid engine and a new reactive engine based on a reactive event bus.

The first half of this chapter presents a brief look at the new camel-reactive-streams component that's introduced in Camel 2.20. This component allows users to take advantage of the reactive streaming APIs and work with the many Camel components. The second half of this chapter is a personal story from Claus, who built a Vert.x application that can simulate live football scores.

20.1 Using Reactive Streams with Camel

This section covers the new camel-reactive-streams component, which you can use to integrate Camel with the Reactive Streams API (www.reactive-streams.org). This API is a small standard specification for asynchronous stream processing. Java 9 comes with the Flow API (`java.util.concurrent.flow`), which corresponds to the Reactive Streams API.

20.1.1 Reactive Streams API

The API is composed of the following four interfaces:

- `Publisher`—A `Publisher` is a provider of a potentially unbounded number of sequenced elements, publishing them according to the demand received from its `Subscriber`s.
- `Subscriber`—A `Subscriber` receives events from a `Publisher`.
- `Processor`—A `Processor` represents a processing stage, which is both a `Subscriber` and a `Publisher` and obeys the contracts of both.
- `Subscription`—A `Subscription` represents a one-to-one lifecycle of a `Subscriber` subscribing to a `Publisher`.

Because Reactive Streams is a specification, you need to use a library that implements this specification. RxJava and Reactor Core are two widely used reactive programming libraries that use Reactive Streams with back pressure included. This book presents examples using those two libraries.

The Reactive Streams specification defines a model for *back pressure*, a way to ensure that a fast publisher doesn't overwhelm a slow subscriber. Back pressure provides resilience by ensuring that all participants in a stream-based system participate in flow control to ensure a steady state of operation and graceful degradation.

20.1.2 Reactive flow control with back pressure

In an ideal situation, a publisher is able to push data to a subscriber as fast as possible, and the subscriber is able to keep up. This isn't the case in the real world we live in, and it's easy to imagine situations in which the subscriber can't keep up and becomes flooded with data. You could try to deal with this by having a buffer at the consumer side with a reasonable

capacity to store new data while the consumer is busy processing. But this often mitigates only *small bumps in the road*. If the situation continues and the producer remains faster than the consumer, the buffer will eventually run out of capacity.

Okay, you could then choose a strategy to start dropping data if the buffer is full. For example, you could choose to drop the oldest or the newest data. But that will result in data loss, which often isn't desirable.

What you need is a bidirectional flow of data. Data flows downstream from the publisher to the subscriber, and the subscriber sends a signal upstream to demand more data. Figure 20.1 illustrates this principle.

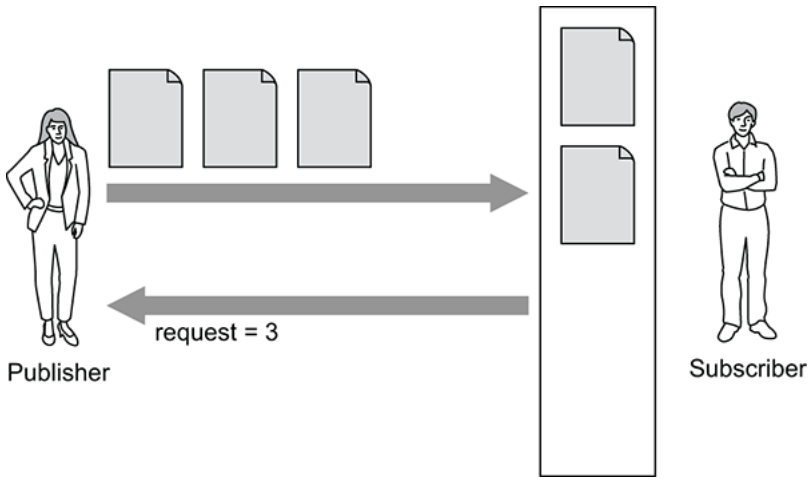

Figure 20.1 The `Publisher` sends data to the `Subscriber`, and the `Subscriber` requests more data from the `Publisher`. Both events occur asynchronously.

When the publisher receives a demand from the subscriber, it's free to publish new data up to the number of elements requested. The bidirectional flow between the publisher and the subscriber is asynchronous and guarantees the best possible flow control.

The back pressure doesn't terminate at the first publisher, as it may cascade further up to upstream publishers. This follows one of the topics from the Reactive Manifesto:

> *Back-pressure may cascade all the way up to the user, at which point responsiveness may degrade, but this mechanism will ensure that the system is resilient under load, and will provide information that may allow the system itself to apply other resources to help distribute the load.*
>
> —*www.reactivemanifesto.org/glossary#Back-Pressure*

To better understand how the flow model between the `Publisher` and `Subscriber` works, look at figure 20.2. The most interesting aspect in figure 20.2 is that the `Subscriber` can request more data using the `request(limit)` method. Then the `Publisher` sends data to the `Subscriber` up until that limit. At any time, the `Subscriber` can request more data by calling the `request(limit)` method again.

You should also know that a `Subscriber` should subscribe to only one `Publisher`, whereas a `Publisher` can have one or more `Subscriber`s.

Okay, enough talk; let's get down to action.

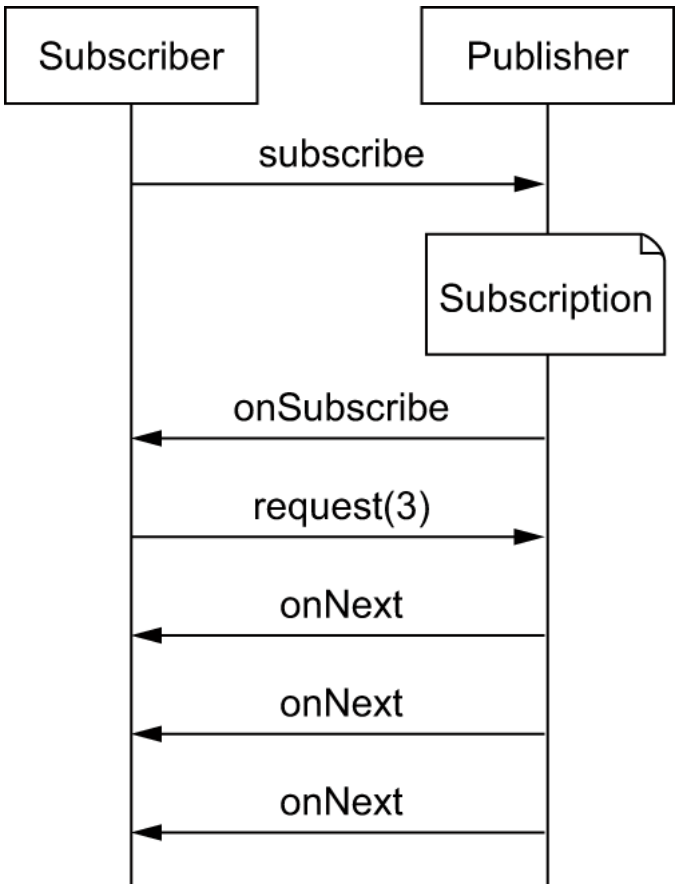

Figure 20.2 The `Subscriber` subscribes to a `Publisher`, which creates a `Subscription`. The `Subscriber` will be signaled by the `onSubscribe` callback when it has been successfully subscribed and the `Producer` is ready to send data. The `Subscriber` then requests how much data to receive. The `Subscriber` receives each data message per the `onNext` callback up until the number of elements requested.

20.1.3 First steps with Reactive Streams

To understand the gist of building reactive flows, we'll start with a simple example: we'll take a set of words, apply a function, and then log each word. To make the example as simple as possible, we'll use a fixed set of words (not a continued stream), and the reactive engine supports this by making it possible to create a `Publisher` that can do only that. The following listing shows how this can be coded using RxJava as the reactive engine.

Listing 20.1 Reactive flow takes the words that are uppercased and logged

```
Publisher<String> publisher = Flowable.just("Camel", "rocks",
                              "streams", "as", "well");  (1)
```

(1) Publisher with only these words

```
Subscriber<String> subscriber = new DefaultSubscriber<>() {    ❷
```

❷

Creates Subscriber

```
    public void onNext(String w) {    ❸
```

❸

The callback executed for each new item

```
            LOG.info(w.toUpperCase());
        }

    public void onError(Throwable throwable)    ❹
```

❹

The callback executed when an error is encountered

```
        }

    public void onComplete() {    ❺
```

❺

The callback executed when the stream ends

```
        }
    }

    publisher.subscribe(subscriber);
```

The main entry to using Reactive Streams with RxJava is the `io.reactivex.Flowable` class ❶, where you can create a `Publisher` to stream from an array. The `Subscriber` is created from the `io.reactivex.subscribers.DefaultSubscriber` class ❷ that allows you to define three callbacks (methods) to receive items and notifications from the `Publisher`. The `onNext` callback ❸ will be called once per every item of the stream. The `onError` callback ❹ will be called if an error occurs upstream. The `onComplete` callback ❺ will be called when the stream completes normally (for bounded streams, like the one used in this example), to signal that no more items will be pushed downstream. In case of error, only the `onError` callback will be called, and `onComplete` will be skipped.

In RxJava, subscribers shouldn't be created explicitly using the `DefaultSubscriber` class. Instead, callback functions and transformations are usually attached to a stream using a flow DSL, as shown in the following listing.

Listing 20.2 Reactive flow using flow DSL

```
Publisher<String> publisher = Flowable.just("Camel", "rocks",
                        "streams", "as", "well");   ❶

❶
Publisher with only these words

Flowable.fromPublisher(publisher)   ❷

❷
Defines a flow starting from the Publisher

  .map(w -> w.toUpperCase())   ❸
```

❸

Applies uppercase function

```
    .doOnNext(w -> LOG.info(w))    ❹
```

❹

onNext logs the word

```
    .subscribe();    ❺
```

❺

Subscribes

```
```

This `Publisher` is created the same way as in listing 20.1, from `Flowable` ❶ to publish just the given words from the array. The `Subscriber` is created using `Flowable` ❷, where you specify the `Publisher` to be used. You then want to uppercase each word, which is done via the `map` function, where you can use Java 8 lambda style to call the `toUpperCase` method ❸. The flow then continues, where you want to run some code that can log the word, as done in the `doOnNext` method ❹. And finally, you call the `subscribe` method ❺, and the flow gets going.

When defining Reactive Streams using this kind of flow style, it may be more common to set up the entire flow directly on one `Flowable`. The code from listing 20.2 can be compacted to four lines of code:

```
Flowable.just("Camel", "rocks", "streams", "as", "well")
    .map(String::toUpperCase)
    .doOnNext(LOG::info)
    .subscribe();
```

You can find this example with the source code in chapter20/rx-java2, and you can try the example using Maven:

```
mvn test -Dtest=FirstTest
```

You can also find an equivalent example using the Reactor Core engine instead in the chapter20/reactor-core directory, and you can try that example with the following:

```
mvn test -Dtest=FirstTest
```

The two examples are almost identical. When using Reactor Core, you use `reactor.core.publisher.Flux` instead of `io.reactivex.Flowable`.

This was a good first test, but let's move on to see how you can use Camel together with Reactive Streams.

20.1.4 Using Camel with Reactive Streams

The reactive-streams component is a Camel component that allows you to use Camel as a publisher or subscriber with your Reactive Streams. In this section, you'll add Camel to the previous example and then let Camel act as a publisher to send in the words to the Reactive Streams. The following listing shows how this can be done.

Listing 20.3 Using Camel as a Publisher to send in words to the reactive flow

```
CamelContext camel = new DefaultCamelContext();
CamelReactiveStreamsService rsCamel = CamelReactiveStreams.get(camel);   ❶
```

❶ Creates Reactive Camel

```
camel.start();   ❷
```

❷ Starts Camel

```
Publisher<String> publisher = rsCamel.from("seda:words", String.class);   ❸
```

❸

Creates Camel Publisher to seda:words endpoint

```
Flowable.fromPublisher(publisher)   ❹
```

❹

Reactive flow

```
  .map(w -> w.toUpperCase())   ❹
  .doOnNext(w -> LOG.info(w))   ❹
  .subscribe();   ❹

FluentProducerTemplate template = camel.createFluentProducerTemplate();   ❺
```

❺

Sends data using Camel

```
template.withBody("Camel").to("seda:words").send();   ❺
template.withBody("rocks").to("seda:words").send();   ❺
template.withBody("streams").to("seda:words").send();   ❺
template.withBody("as").to("seda:words").send();   ❺
template.withBody("well").to("seda:words").send();   ❺
```

To bridge Camel with Reactive Streams, you should get an instance of `CamelReativeStreamsService` ❶, whose lifecycle is controlled by the `CamelContext` ❷. Camel can be used as `Publisher` from any of its 200+ components that can act as a consumer. At first, it may sound confusing that a Reactive Streams `Publisher` is using a Camel `Consumer`. The `Publisher` is the data sink where new data comes in, which is a consumer in EIP terms and hence a Camel `Consumer`.

In this example, you want to receive data from the `seda:words` Camel endpoint ❸. Notice that you specify the type as `String`, which ensures that the `Flowable` builder ❹ is able to use this as a Java generic, so that the compiler can accept the lambda code that calls the `toUpperCase` method, because it knows it's a `String` type. The last part of the example is to use Camel to send the words to the `seda:words` endpoint ❺, which then triggers the reactive flow.

You can find this example with the source code in chapter20/rx-java2, and you can try the example using this Maven goal:

```
mvn test -Dtest=CamelFirstTest
```

Using Camel route as Reactive Streams Publisher

When using Camel, you often use Camel routes, so let's take a look at how to let a Camel route be the data sink for a reactive flow. This time, let's try numbers instead of words and use a Camel route that has a continued stream of data, as shown here:

```
from("timer:number")
    .transform(simple("${random(0,10)}"))
    .log("Generated random number ${body}")
    .to("reactive-streams:numbers");    ❶
```

❶ Sends data to reactive-streams endpoint

The Camel route is trivial, which starts from a timer that triggers once per second. You then generate a random number between 0 and 9 (inclusive), which is sent to the `reactive-streams` endpoint with the name numbers ❶. This is a Camel endpoint from the camel-reactive-streams component, which you can use as bridge between Camel and your reactive flows. The following listing shows the source code with the reactive flow.

Listing 20.4 Reactive flow with a `Publisher` as data sink from Camel route

```
CamelReactiveStreamsService rsCamel = CamelReactiveStreams.get(context);   1
```

1

Creates Reactive Camel

```
Publisher<Integer> numbers = rsCamel.fromStream("numbers",   2
```

2

Publisher from Camel reactive-streams endpoint

```
                                Integer.class);   2

Flowable.fromPublisher(numbers)   3
```

3

Reactive flow

```
  .filter(n -> n > 5)   4
```

4

Filters for only big numbers

```
  .doOnNext(n -> log.info("Streaming big number {}", n))   5
```

5

Logs the big number

```
  .subscribe();   6
```

❻
Subscribes

To use Camel with reactive flows, you need to get an instance of `CamelReativeStreamsService` ❶, which you use to create a `Publisher` from the stream with the name numbers of type `Integer` ❷. Pay attention that numbers is the same name used in the Camel route where data is being sent by the following:

```
.to("reactive-streams:numbers")
```

The reactive flow starts from the `Publisher` you created ❸. This time, you want to apply a filter that drops the low numbers so you carry only the big numbers ❹. Each of these big numbers is then logged ❺. Finally, you start this flow by calling the `subscribe` method ❻.

REACTIVE STREAMS OPERATORS

When using Reactive Streams with RxJava or Reactor Core, you have many operators at your disposal to apply all kind of functions to filter, transform, combine, and merge streams. In this chapter, you've used only a few of those operators. The ReactiveX website lists all the operators: http://reactivex.io/documentation/operators.html.

You can try this example, provided in the chapter20/rx-java2 directory, with the following:

```
mvn test -Dtest=CamelNumbersTest
```

Using Camel route as Reactive Streams Subscriber

Now let's try the opposite: letting a reactive flow publish streams with Camel acting as the subscriber by receiving the data as input to a Camel route. The Camel route is merely two lines of code:

```
from("reactive-streams:numbers")
```

```
        .log("Got number ${body}");
```

The reactive flow source code is shown in the following listing.

Listing 20.5 Reactive flow with Camel as a Subscriber

```
CamelReactiveStreamsService rsCamel = CamelReactiveStreams.get(context);   ❶

Flowable.just("3", "4", "1", "5", "2")   ❷

    .sorted(String::compareToIgnoreCase)   ❸

    .subscribe(rsCamel.streamSubscriber("numbers", String.class));   ❹
```

❶ Creates Reactive Camel

❷ Publisher that just sends those five numbers

❸ Sorts the numbers

❹ Subscriber from Camel reactive-streams endpoint

Yes, you've read it before: when you use Camel with Reactive Streams, you need to get an instance of `CamelReactiveStreamsService` ❶. To keep this example simple, you let RxJava create a `Publisher` with just five numbers ❷. Because the numbers are unordered, you can apply a sort function ❸. Then you create a Camel `Subscriber` with the name num-

bers ❹. The name of the stream is the name of the endpoint used in the Camel route, for example:

```
from("reactive-streams:numbers")
```

You can try this example from the chapter20/rx-java2 directory by executing the following Maven goal:

```
mvn test -Dtest=CamelConsumeNumbersTest
```

So far, all the integration between Camel routes and reactive flows has used the reactive-streams component. But regular Camel endpoints can also be used.

Using regular Camel components in reactive flow

The last example we want to show you uses regular Camel endpoints in the reactive flow. In this example, you'll use the Camel file component as a sink for a reactive flow and a Camel route as the subscriber. The following listing shows the source code.

Listing 20.6 Reactive flow with regular Camel endpoints and routes

```
Flowable.fromPublisher(rsCamel.from("file:target/inbox"))    ❶
```

❶ Publisher from Camel file endpoint

```
  .doOnNext(e -> rsCamel.to("direct:inbox", e))    ❷
```

❷ Calls Camel route from flow

```
  .filter(e -> e.getIn().getBody(String.class).contains("Camel"))    ❸
```

❸

Filters out files without Camel text

```
  .subscribe(rsCamel.subscriber("direct:camel"));   ❹
```

❹

Subscriber from Camel direct endpoint

```
  from("direct:inbox")   ❺
```

❺

Camel route called from flow

```
    .log("Inbox ${header.CamelFileName}")
    .wireTap("mock:inbox");

  from("direct:camel")   ❻
```

❻

Camel route used by Subscriber

```
    .log("This is a Camel file ${header.name}")
    .to("mock:camel");
```

This time, the reactive flow seems more complicated the first couple of times you read it. You start using reactive Camel to create a `Publisher` from a regular Camel file endpoint ❶. From the reactive flow, you can make calls into Camel routes from within the `doOnNext` function ❷, where you can use Reactive Camel to publish to the Camel route ❺. Then you use the `filter` function ❸ to include only files that contain the text “Camel”. Finally, you let Camel be the `Subscriber` ❹ by routing the data to the specified Camel route ❻.

You can try this example, found in the chapter20/rx-java2 directory, by running the following:

```
mvn test -Dtest=CamelFilesTest
```

TIP You can find more examples from Apache Camel at https://github.com/apache/camel/tree/master/examples/camel-example-reactive-streams.

Section 20.1.1 mentioned that one of the main goals of the Reactive Streams specification is to define a model for back pressure. The following two sections show you how to configure and control back pressure from a Camel reactive producer and consumer point of view.

20.1.5 Controlling back pressure from the producer side

When routing messages using Camel to an external subscriber, back pressure is by default handled by an internal buffer that caches the Camel messages before delivering them to the reactive subscriber. If the subscriber is slower than the rate of messages, the internal buffer may fill up and become too big. Such a situation must be avoided.

For example, suppose you have the following Camel route:

```
from("jms:queue:inbox")
  .to("reactive-streams:inbox");
```

If the JMS queue contains a lot of messages and the reactive subscriber is too slow to process the messages, the pending messages will keep piling up in an internal buffer by the reactive-streams endpoint. That can potentially degrade the performance in the JVM by taking up memory, or in the worst case, cause an out-of-memory error in the JVM and cause the application to crash.

We've provided an example, shown in the following listing, with the source code demonstrating a situation in which the subscriber is too slow, and pending messages are piling up in the internal buffer.

Listing 20.7 Reactive flow without back pressure causing messages to fill up buffer

```
public void testNoBackPressure() throws Exception {
  CamelReactiveStreamsService rsCamel = CamelReactiveStreams.get(context

  Publisher<String> inbox = rsCamel.fromStream("inbox", String.class);

  Flowable.fromPublisher(inbox)
    .doOnNext(c -> {
    log.info("Processing message {}", c);    ❶
```

❶ Reactive flow that simulates a slow subscriber

```
    Thread.sleep(1000);    ❶
    })
    .subscribe();

  for (int i = 0; i < 200; i++) {    ❷
```

❷ Sends in 200 messages to queue

```
    fluentTemplate.withBody("Hello " + i)    ❷
      .to("seda:inbox?waitForTaskToComplete=Never").send();    ❷
  }

  Thread.sleep(250 * 1000L);
}

protected RoutesBuilder createRouteBuilder() throws Exception {
  return new RouteBuilder() {
    public void configure() throws Exception {
      from("seda:inbox")
    .delay(100)    ❸
```

❸

Camel route being faster than the subscriber

```
        .log("Camel routing to Reactive Streams: ${body}")
    .to("reactive-streams:inbox");   ❹
```

❹

Messages will pile up here at the reactive buffer

```
    }
  };
}
```

This example has been constructed so that we humans can follow what happens while it runs; the reactive flow takes 1 second to process each message ❶. Instead of sending in hundreds of thousands of messages, you use a low limit of 200. That's enough to prove the point that those messages will stack up at the reactive buffer ❹. The Camel route also has a little delay ❸ to allow us to better follow from the console output what's happening.

You can run the example from the chapter20/rx-java2 directory by executing the following Maven goal:

```
mvn test -Dtest=CamelNoBackPressureTest
```

While the example runs, it outputs to the console, as shown here:

```
INFO  NoBackPressureTest - Processing message Hello 20
INFO  route1             - Camel routing to Reactive Streams: Hello 195
INFO  route1             - Camel routing to Reactive Streams: Hello 196
INFO  route1             - Camel routing to Reactive Streams: Hello 197
INFO  route1             - Camel routing to Reactive Streams: Hello 198
INFO  route1             - Camel routing to Reactive Streams: Hello 199
INFO  NoBackPressureTest - Processing message Hello 21
INFO  NoBackPressureTest - Processing message Hello 22
```

The interesting part of the output is highlighted in bold. Here you can see that Camel has routed all 200 (0–199) messages to the Reactive Streams channel. And at this time, the reactive flow is processing message number 21, which means that 200 – 21 – 1 = 178 messages are pending in the buffer. Now suppose you send 2,000 messages instead of 200; what would the situation be?

```
INFO  route1             - Camel routing to Reactive Streams: Hello 1998
INFO  route1             - Camel routing to Reactive Streams: Hello 1999
INFO  NoBackPressureTest - Processing message Hello 205
INFO  NoBackPressureTest - Processing message Hello 206
```

This time, there are 2,000 – 205 – 1 = 1,794 messages pending in the buffer. As expected, the subscriber can't keep pace with the publisher, and you could potentially cause the JVM to become unstable with out-of-memory errors or degrade in performance.

TIP Runtime statistics about the Camel Reactive Streams are available from JMX. Under the services folder, you can find the `DefaultCamelReactiveStreamsService` MBean, which has JMX operations that return the statistics in tabular format.

Adding back pressure

To avoid such problems, you can use back pressure to prevent dequeuing too many messages from the JMS queue and instead try to keep a pace that's more aligned with the subscriber.

The strategy for back pressure that you can use in this example is to use Camel's `ThrottlingInflightRoutePolicy` in the Camel route, as shown in the following listing.

Listing 20.8 Reactive flow with back pressure using Camel route policy

```
public void configure() throws Exception {
    ThrottlingInflightRoutePolicy inflight =    (1)
```

❶

Creates route policy

```
    new ThrottlingInflightRoutePolicy();    ❶
  inflight.setMaxInflightExchanges(20);    ❷
```

❷

Configures policy

```
  inflight.setResumePercentOfMax(25);    ❷

  from("seda:inbox").routePolicy(inflight)    ❸
```

❸

Uses policy in Camel route

```
            .delay(100)
            .log("Camel routing to Reactive Streams: ${body}")
            .to("reactive-streams:inbox");
    }
```

To make the reactive flow and the Camel route in this listing flow with a similar pace, you can use Camel's route policy to suspend/resume the route to keep a maximum number of inflight messages. Therefore, you create the `ThrottlingInflightRoutePolicy` ❶, which is configured to limit at most 20 inflight messages ❷ and resume again at 25 percent of the maximum (= 5 messages). In other words, the rate will be between 5 and 20 inflight messages. To use the policy, you must remember to configure it on the route ❸.

TIP You can find more information about the Camel route policy in chapter 15, section 15.2.3.

You can run this example by executing the following Maven command:

```
cd chapter20/rx-java2
mvn test -Dtest=CamelInflightBackPressureTest
```

The following output is captured at a similar moment, when we ran the example without back pressure:

```
INFO  route1           - Camel routing to Reactive Streams: Hello 198
INFO  route1           - Camel routing to Reactive Streams: Hello 199
INFO  BackPressureTest - Processing message Hello 184
INFO  BackPressureTest - Processing message Hello 185
```

This time, you can see that when the 200 messages have been routed by Camel, the reactive flow is currently processing message 184. Only 15 (200 – 184 – 1 = 15) pending messages are in the reactive buffer. This is because of the back pressure in use.

The following two outputs represent when the back pressure is in use by first suspending the route and a little while later resuming the route:

```
INFO  route1           - Camel routing to Reactive Streams: Hello 107
INFO  route1           - Throttling consumer: 22 > 20 inflight exchange
INFO  BackPressureTest - Processing message Hello 87
...
INFO  BackPressureTest - Processing message Hello 102
INFO  BackPressureTest - Processing message Hello 103
INFO  route1           - Throttling consumer: 5 <= 5 inflight exchange by
INFO  BackPressureTest - Processing message Hello 104
INFO  route1           - Camel routing to Reactive Streams: Hello 108
```

And when you run the example with 2,000 messages, you can see at the end of the test that the back pressure works as if there are only 19 pending messages in the reactive buffer:

```
INFO  route1           - Camel routing to Reactive Streams: Hello 1998
INFO  route1           - Camel routing to Reactive Streams: Hello 1999
INFO  BackPressureTest - Processing message Hello 1980
INFO  BackPressureTest - Processing message Hello 1981
```

ABOUT USING BACK PRESSURE WITH THE THROTTLING ROUTE POLICY

Using `ThrottlingInflightRoutePolicy` as back pressure works by suspending and resuming the Camel route. This works the best when the route is consuming messages from a messaging system such as JMS, AMQP, or Kafka. But if the route is an HTTP service, then the route suspending will cause the HTTP service to be unavailable and return HTTP Status 503 to clients. That may not be desirable, and you should try to scale out your applications in a cluster, as covered in chapters 17 and 18.

If a certain amount of data loss is acceptable, you can configure Camel to use a different strategy than buffering every message.

Using alternative back-pressure strategies

Table 20.1 lists the back-pressure strategies supported by Camel reactive-streams component.

Table 20.1 Back-pressure strategies supported by Camel

Strategy	Description
`BUFFER`	Buffers all messages in an unbounded buffer. This is the default strategy.
`LATEST`	Keeps only the latest message and drops all the others.
`OLDEST`	Keeps only the oldest message and drops all the others.

So far, all the examples have been using the default `BUFFER` back-pressure strategy, which is the first item in table 20.1. This strategy buffers the messages, which ensures that no data loss occurs. But you learned about the potential danger if the publisher produces messages faster than the downstream subscribers can process; this can cause the JVM to consume more and more memory, degrade in performance, or eventually run out of memory. Camel supports alternative strategies if message loss can be accepted, which is done by discarding messages from the buffer. When using the `LATEST` or `OLDEST` strategy, the discarded messages will be re-

moved from the buffer, and a `ReactiveStreamsDiscardedException` exception is thrown for each message. By throwing the exception, you can use Camel to react and use its error handling to route the message to a dead letter channel, or to log the message, or silently ignore it.

Let's see an example in which you want to process only the latest message and silently ignore the discarded messages. You can run this example from the chapter20/rx-java2 directory as follows:

```
mvn test -Dtest=CamelLatestBackPressureTest
```

The interesting output from running the example is highlighted here:

```
INFO  route1                - Camel routing to Reactive Streams: Hello
INFO  route1                - Camel routing to Reactive Streams: Hello
INFO  LatestBackPressureTest - Processing message Hello 194
INFO  route1                - Camel routing to Reactive Streams: Hello
INFO  route1                - Camel routing to Reactive Streams: Hello
INFO  route1                - Camel routing to Reactive Streams: Hello
INFO  route1                - Camel routing to Reactive Streams: Hello
INFO  route1                - Camel routing to Reactive Streams: Hello
INFO  LatestBackPressureTest - Processing message Hello 199
```

As you can see, the reactive flow processes the latest message that was sent to the buffer. All the other messages are discarded. To avoid causing every discarded message to fail and have its stacktrace logged, you can use Camel's error handler to handle the `ReactiveStreamsDiscardedException` exception by adding the following line to the Camel route:

```
onException(ReactiveStreamsDiscardedException.class).handled(true);
```

You can also use back pressure from the other side, the consumer side.

20.1.6 Controlling back pressure from the consumer side

When Camel consumes messages from a reactive publisher, it has back pressure enabled out of the box. The Camel consumer allows by default at most 128 inflight messages, which are used to determine the number of messages to request from the publisher when requesting for more data.

In other words, Camel will at most request up to 128 messages from the publisher.

You can configure the maximum number of inflight messages as an option on the endpoint:

```
from("reactive-streams:inbox?maxInflightExchanges=5")
```

The following listing shows the source code of an example using back pressure on the consumer side.

Listing 20.9 Camel reactive consumer with back pressure

```
public void testConsumerBackPressure() throws Exception {
  CamelReactiveStreamsService rsCamel = CamelReactiveStreams.get(context

  String[] inbox = new String[100];
  for (int i = 0; i < 100; i++) {
    inbox[i] = "Hello " + i;
  }

 Flowable.fromArray(inbox)    (1)
```

(1) Reactive flow

```
   .doOnRequest(n -> {    (2)
```

(2) Logs every request for more data

```
    log.info("Requesting {} messages", n);    (2)
    })
   .subscribe(rsCamel.streamSubscriber("inbox", String.class));    (3)
```

❸

Camel reactive subscriber

```
  Thread.sleep(10 * 1000L);
}

protected RoutesBuilder createRouteBuilder() throws Exception {
  return new RouteBuilder() {
    public void configure() throws Exception {
  from("reactive-streams:inbox?maxInflightExchanges=5    ❹
```

❹

Camel reactive route

```
                  &concurrentConsumers=5")    ❹
        .delay(constant(10))
        .log("Processing message ${body}");
    }
  };
}
```

The example uses a reactive flow that publishes 100 messages ❶. To be able to see when Camel requests more data, you use `doOnRequest` to log the number of messages that were requested ❷. The flow then uses Camel as the subscriber ❸ on the channel named inbox. You can see how to create a Camel route that consumes from this channel ❹. The consumer has been configured with a maximum of five inflight messages. To speed up processing, you've turned on concurrent consumers with a value of 5 also. This makes Camel create a thread pool of five threads, each of which acts as a reactive subscriber and will independently process the messages from the reactive channel. But they'll collectively act together under the limitations of the maximum inflight messages. Because those values are the same, each consumer won't exceed the maximum number of inflight messages.

The example, provided with the source code in the chapter20/rx-java2 directory, runs using the following command:

```
mvn test -Dtest=CamelConsumerBackPressureTest
```

When running the example, you'll notice the following from the output:

```
INFO  CamelConsumerBackPressureTest  - Requesting 5 messages
INFO  route1                         - Processing message Hello 0
INFO  route1                         - Processing message Hello 4
INFO  route1                         - Processing message Hello 3
INFO  route1                         - Processing message Hello 2
INFO  route1                         - Processing message Hello 1
INFO  CamelConsumerBackPressureTest  - Requesting 3 messages
INFO  CamelConsumerBackPressureTest  - Requesting 1 messages
INFO  CamelConsumerBackPressureTest  - Requesting 1 messages
INFO  route1                         - Processing message Hello 7
INFO  CamelConsumerBackPressureTest  - Requesting 1 messages
INFO  route1                         - Processing message Hello 5
INFO  CamelConsumerBackPressureTest  - Requesting 1 messages
```

As you can see, the output shows that Camel requests the maximum number of inflight messages at first, and then as it runs the request, it drops down to one or three messages. Often only one message is requested. That's because each consumer thread will refill the buffer when it has processed the message. And because the thread has just completed its own message, the buffer often has room for only one more message.

Doing frequent `request(1)` calls is generally a bad pattern, as this increases the communication costs (chatty network). Therefore, Apache Camel from version 2.20 onward provides the `exchangesRefillLowWatermark` option, which is used as a threshold to trigger when to request more data. The watermark is a percentage of the maximum inflight exchanges and has the default value of 0.25. Running the same example as before with Camel 2.20 onward will demonstrate that Camel doesn't perform any `request(1)` calls anymore.

That's all we could squeeze into this chapter about Camel, Reactive Streams, and the Camel reactive-streams component. This fairly new addition to the Apache Camel project is expected to be taken up by more

Camel users when reactive applications start to become more in use. We'll come back to some more thoughts on this in the chapter summary.

At this point, let's move on to the second half of this chapter, which is devoted to Vert.x.

20.2 Using Vert.x with Camel

On the Eclipse Vert.x website, the project describes itself as *a toolkit for building reactive applications on the JVM*. This description has three important points.

First, as a toolkit, Vert.x isn't an application server. Vert.x is just a JAR file (vertx-core), so a Vert.x application is an application that uses this JAR file.

Second, Vert.x is reactive and adheres to the Reactive Manifesto (http://reactivemanifesto.org), which is highlighted in the following four bullets:

- *Responsive*—A reactive system needs to handle requests in a reasonable amount of time.
- *Resilient*—A reactive system must stay responsive in the face of failures, so it must be designed with failure in mind.
- *Elastic*—A reactive system must stay responsive under various loads. Therefore, it must be scalable.
- *Message driven*—Reactive systems rely on asynchronous messages passing between its components. This establishes a boundary between components that ensures loose coupling, isolation, and location transparency.

Finally, Vert.x applications run on the JVM, which means Vert.x applications can be developed using any of the JVM languages such as Java, Groovy, Scala, Kotlin, Ceylon, and JavaScript. The polyglot nature of Vert.x allows you to use the most appropriate language for the task.

TIP If you're new to Vert.x, we recommend the free book *Building Reactive Microservices in Java* by Clement Escoffier, which you can download from https://developers.redhat.com/promotions/building-reactive-microservices-in-java. This book covers Apache Camel. But we want to give room for Vert.x in this book, as it's a great toolkit that becomes even greater when used together with Camel.

20.2.1 Building a football simulator using Vert.x

What follows next is a true event that happened in the life of Claus:

I hosted a Christmas party with seven of my old friends in December 2016. When we were younger, from the late 90s to the mid 00s, we'd have these football weekends to watch English football. During a football weekend, the state lottery company would issue a football pool coupon with 13 games. One of these games was the TV game, and out of the remaining 12 games, each of us would select a game. We would then watch the TV game as the other games were played at the same time. Whenever a goal was scored in any of the games, it would be announced on TV with a *bell* sound. Our rules were simple: if a goal was scored in the TV game, everyone would drink. If a goal was scored in your game, you would drink (bottoms up). To keep the spirit of the old days, a Manchester derby game from February 2004 was selected as the TV game. Figure 20.3 shows a screenshot of the game in action with the goal simulator on the right-hand side.

Figure 20.3 Football simulator playing the TV game with live goal scorer updates on the right

We had a great weekend playing this old game again. Manchester United won 4 to 2 with goals from Scholes and Ronaldo, and two from van

Nistelrooy. That was the fun part; now let's talk about how to implement this using Vert.x.

The architecture

Figure 20.4 illustrates the key components in the architecture.

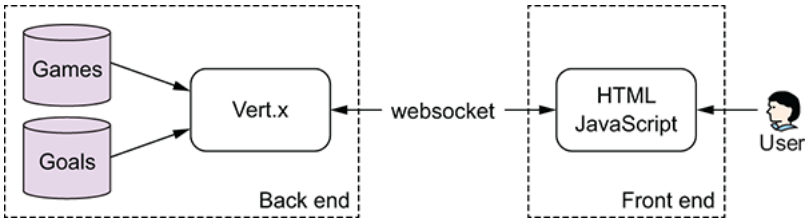

Figure 20.4 In the back end, the Vert.x application loads the games and goals from disk. The front end is an HTML web application with embedded JavaScript that uses WebSocket to communicate with the Vert.x back end.

The back end is implemented as a Vert.x application using Java. Information about the games and the goal scorers is stored in CSV files, which are loaded into the Vert.x application upon startup. The front end is a HTML file with embedded JavaScript. The JavaScript uses the Vert.x JavaScript client that handles all communication to the back end via WebSocket. The entire application is packed together as a single fat JAR, which can run as a Java application on the JVM.

The book's source code contains this example in the chapter20/vertx directory. You can try this by running the following Maven goal:

```
mvn compile vertx:run
```

Then open a web browser to http://localhost:8080.

The application is built using Java and HTML code with fascinating parts.

The Java code

You develop your Vert.x application in Java code in a *verticle*, which is deployed and running in the Vert.x instance. The verticle is an abstract class that has start and stop methods and easy access to the Vert.x instance itself.

The following listing shows the verticle for the football simulator.

Listing 20.10 Vert.x verticle for the football simulator

```
public class LiveScoreVerticle extends AbstractVerticle {   ❶
```

❶

The verticle is extending AbstractVerticle class

```
  public void start() throws Exception {
    Router router = Router.router(vertx);   ❷
```

❷

Create Vert.x router to set up WebSocket and HTTP server

```
    BridgeOptions options = new BridgeOptions()   ❸
```

❸

Allowed inbound and outbound traffic on event bus

```
        .addInboundPermitted(new PermittedOptions().setAddress("control"))
        .addOutboundPermitted(new PermittedOptions().setAddress("clock"))
        .addOutboundPermitted(new PermittedOptions().setAddress("games"))
        .addOutboundPermitted(new PermittedOptions().setAddress("goals"));

    router.route("/eventbus/*")   ❹
```

❹

Route WebSocket to Vert.x event bus

```
        .handler(SockJSHandler.create(vertx).bridge(options, event -> {
          if (event.type() == BridgeEventType.SOCKET_CREATED) {
      vertx.setTimer(500, h -> initGames());   ❺
```

❺

A new WebSocket client is connected, so initialize list of games to client

```
        }
        event.complete(true);
    }));

    router.route().handler(StaticHandler.create());  ❻
```

❻

Add static HTML resources to Vert.x router

```
    vertx.createHttpServer()
        .requestHandler(router::accept).listen(8080);  ❼
```

❼

Vert.x router listen on port 8080

```
    initControls();  ❽
```

❽

Initialize game controls and start live score streams

```
    streamLiveScore();  ❽
  }
```

This football simulator code shows how to build a Vert.x application as a verticle. The `LiveScoreVerticle` class extends `io.vertx.core.AbstractVerticle` ❶. In the `start` method, you have the necessary code to start up, such as creating a Vert.x router for HTTP and WebSocket ❷. Then you set up allowed inbound and outbound communication ❸ to the router with the following four event-bus addresses:

control, clock, games, and goals. The communication between the back end and front end uses WebSocket, which you add to the router ❹. Whenever a new client is connected, the `SOCKET_CREATED` event is emitted to the back end, which triggers initialization of the game list ❺. The HTTP router is also used to service static content such as HTML files ❻ and is started by listening to port 8080 ❼. And finally, the game controls and the goal score stream are started ❽.

The `LiveScoreVerticle` class has more code to initialize the game list and react to game control buttons pushed, game clock advances, and the actual stream of goals. For example, the list of games is initialized as shown in the following listing.

Listing 20.11 Initializing the list of games

```
private void initGames() {
  try {
    InputStream is = LiveScoreVerticle.class.getClassLoader()
                       .getResourceAsStream("games.csv");
  String text = IOHelper.loadText(is);     ❶
```

❶ Loads list of games from classpath

```
    Stream<String> games = Arrays.stream(text.split("\n"));
  games.forEach(game -> vertx.eventBus().publish("games", game));     ❷
```

❷ Publishes each game to the event bus

```
  } catch (Exception e) {
    System.out.println("Error reading games.csv file");
  }

 if (clockRunning.get()) {     ❸
```

❸

Publishes game clock time

```
    vertx.eventBus().publish("clock", "" + gameTime.get());    ❸
  } else {    ❸
    vertx.eventBus().publish("clock", "Stopped");    ❸
  }    ❸
}    ❸
```

The list of games is stored in a CSV file that's loaded ❶. Each line in the CSV file is a game that gets published to the Vert.x event bus at the game's address ❷. In addition, the game clock state is published ❸.

As you can see, it's easy to send messages to the Vert.x event bus using the one-liner code with the `publish` method ❷. This is similar to Camel's `ProducerTemplate`, which also makes it easy in one line of code to send a message to any Camel endpoint. But what if you want to consume a message instead? How can you do that from Vert.x?

The football simulator has buttons in the front end that control the game clock, so the user can start and stop the clock. Each time the user clicks those buttons, a message is sent from the front end to the back end using WebSocket on the Vert.x event bus. The following code is all it takes in the back end to set up the consumer:

```
private void initControls() {
  vertx.eventBus().<String>consumer("control", h -> {    ❶
```

❶

Sets up consumer on the Vert.x event bus control address

```
    String action = h.body();    ❷
```

❷

The message body contains action to perform

```
    if ("start".equals(action)) {
  clockRunning.set(true);    ❸
```

❸

Either starts or stops the game clock

```
  vertx.eventBus().publish("clock", "" + gameTime.get());    ❹
```

❹

Publishes the new game clock state back to the front end

```
    } else if ("stop".equals(action)) {
  clockRunning.set(false);    ❸
```

❸

Either starts or stops the game clock

```
  vertx.eventBus().publish("clock", "Stopped");    ❹
```

❹

Publishes the new game clock state back to the front end

```
    }
  });
}
```

From the Vert.x event bus, you set up a consumer to listen on the address control ❶, and each message received triggers the handler (using Java 8 lambda style). A Vert.x message also consists of a message body and headers, just like a Camel message. The message body contains what button the user clicked ❷, and you react accordingly to either start or stop the game clock ❸. You then publish the new state back so the web page can react and update its display ❹.

This was the key point from the Java code; let's switch over to the wild west of front-end programming with web frameworks and JavaScript. It's fairly simple, certainly with the Vert.x JavaScript client.

The HTML code

Vert.x allows you to embed web resources such as HTML and JavaScript files in the src/main/resources/webroot folder. You have the following files in this folder:

```
├── bell.m4r
├── index.html
└── vertx-eventbus.js
```

The bell.m4r file is the bell audio that's played when a goal is scored. The index.html file is the HTML file that we'll dive into in a moment. And the vertx-eventbus.js file is the Vert.x JavaScript client.

The index.html file is a plain HTML file with embedded CSS styles and JavaScript. In the `<head>` section, you set up Vert.x as follows:

```
<head>
  <title>Premier League 2004 Week 7 Livescores</title>
  <script src="https://code.jquery.com/jquery-1.11.2.min.js"></script>
  <script src="//cdn.jsdelivr.net/sockjs/0.3.4/sockjs.min.js"></script>
  <script src="vertx-eventbus.js"></script>
</head>
```

In this example, we're using the popular JQuery JavaScript library. For WebSocket communication, Vert.x uses SockJS, which provides fallbacks to a simulated WebSocket communication if the web browser doesn't support native WebSocket. The last script is to include Vert.x itself.

In the `<script>` section, you set up the front end to connect to the Vert.x event bus, as shown in the following listing.

Listing 20.12 Using Vert.x in the front end to handle events from the event bus

```
<script>
```

```
var eb = new EventBus("/eventbus");  (1)
```

(1)

Connects to Vert.x event bus

```
eb.onopen = function () {

  eb.registerHandler("clock", function (err, msg) {  (2)
```

(2)

Reacts when the game clock changes

```
    document.getElementById("clock").innerHTML = msg.body;
  });

  eb.registerHandler("games", function (err, msg) {  (3)
```

(3)

Reacts when the game list is updated

```
    var arr = msg.body.split(',');
    var game = arr[0];
    var home = arr[1];
    var away = arr[2];

    // more code here not shown
  });

  eb.registerHandler("goals", function (err, msg) {  (4)
```

(4)

Reacts when a goal is scored

```
      if (msg.body === 'empty') {
        clearScorer();
        return;
      }

      playsound();

      // more code here not shown
    })
  };
```

To use the Vert.x event bus from JavaScript, you need to create a new event bus object with the URL to the back end ❶. Then the `onopen` method allows you to register handlers that react when messages are sent to event-bus addresses. In this example, the front end uses three addresses: when the game clock is updated ❷, when the list of games is initialized ❸, and when a goal is scored ❹.

For example, when the game clock is updated ❷, the front end updates the HTML page by setting `innerHTML` to the message body. The game clock is an HTML `<div>` element, as shown here:

```
<div id="clock" class="clock"></div>
```

The source code in listing 20.12 has been abbreviated to not show the JavaScript code that manipulates the HTML elements to update the website.

The source code is located in the chapter20/vertx directory. You can try running the example using the following Maven goal:

```
mvn vertx:run
```

Then from a web browser, open http://localhost:8080.

The example has been accelerated, so you don't have to wait the full 90 minutes for the football game to end.

If you've been sitting back and relaxing for a while as the goals are pouring in, you may have been enlightened and noticed that the goal simula-

tor isn't using Camel at all. Vert.x is surely an awesome toolkit for building small reactive microservices. But this book is titled *Camel in Action*, so let's update the simulator to use Camel with Vert.x.

20.2.2 Using Camel together with Vert.x

Vert.x and Camel are both small, lightweight toolkits that work well together. Figure 20.5 illustrates this principle, with Vert.x and Camel working together in the same Vert.x application.

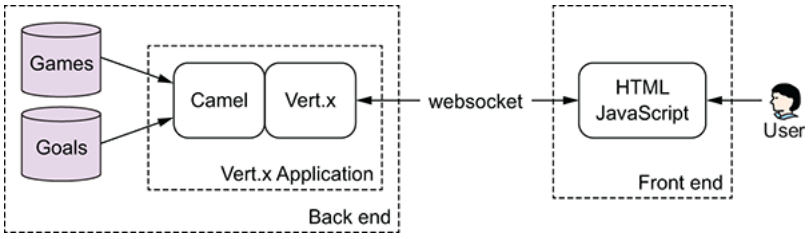

Figure 20.5 Camel and Vert.x working together in the same Vert.x application in the back end. The other parts of the architecture are unchanged.

To use Camel with Vert.x, you need to add camel-core and camel-vertx dependencies to the Maven pom.xml file. In the Vert.x application, you add Camel to the verticle class, as shown in the following listing.

Listing 20.13 Adding Camel to a Vert.x application

```
public class LiveScoreVerticle extends AbstractVerticle {

  private CamelContext camelContext;
  private FluentProducerTemplate template;

  public void start() throws Exception {
    camelContext = new DefaultCamelContext();    ❶
```

❶ Creates CamelContext

```
    camelContext.addRoutes(new LiveScoreRouteBuilder(vertx));    ❷
```

❷ Adds Camel routes that use the Vert.x instance

```
    template = camelContext.createFluentProducerTemplate();  ❸
```

❸

Creates ProducerTemplate

```
    camelContext.start();  ❹
```

❹

Starts Camel

```
        Router router = Router.router(vertx);
    ...  ❺
```

❺

Sets up Vert.x Router

```
    }

    public void stop() throws Exception {
    template.stop();  ❻
```

❻

Stops Camel

```
    camelContext.stop();  ❻
    }
}
```

The code to add Camel should be familiar to you. At first, `CamelContext` is created ❶, and then routes are added ❷. Then you create `ProducerTemplate`, ❸ which will be used by Vert.x to trigger a Camel route. Camel is then started ❹, and the subsequent code sets up Vert.x

router ❺. When the Vert.x application stops, you must remember to stop Camel as well ❻.

Calling Camel from Vert.x

When a new client connects to the back end, Vert.x will trigger the `SOCKET_CREATED` event, which you use to obtain the list of games and send to the front end so the website can be updated accordingly. In the previous example, you used Java code to load the game list from a CSV file and send the information using Vert.x. This time, you're using Camel, so the `SOCKET_CREATED` event uses `ProducerTemplate` to trigger a Camel route by sending an empty message to the `direct:init-game` endpoint ❶, as shown here:

```
router.route("/eventbus/*")
  .handler(SockJSHandler.create(vertx).bridge(options, event -> {
    if (event.type() == BridgeEventType.SOCKET_CREATED) {
      vertx.setTimer(100, h -> template.to("direct:init-games").send()); ❶
```

❶ Calling Camel route using the ProducerTemplate

```
    }
    event.complete(true);
}));
```

As you can see, calling Camel from Vert.x is simple; you use regular Camel APIs such as a `ProducerTemplate`. But what about the other way around? How do you make Camel call Vert.x?

Calling Vert.x from Camel

Camel provides the camel-vertx component that's used to route messages to/from the Vert.x event bus and Camel. The following listing shows how this is done.

Listing 20.14 Using Camel routes to stream live goal scores to Vert.x event bus

```
public class LiveScoreRouteBuilder extends RouteBuilder {

  private final Vertx vertx;

  public LiveScoreRouteBuilder(Vertx vertx) {
    this.vertx = vertx;    ❶
```

❶ Injects Vert.x instance in constructor

```
  }

  public void configure() throws Exception {
    getContext()
     .getComponent("vertx", VertxComponent.class)
    .setVertx(vertx);    ❷
```

❷ Sets up Vert.x instance on Camel vertx component

```
    from("direct:init-games").routeId("init-games")    ❸
```

❸ Routes to initialize game list

```
      .log("Init games event")
    .to("goal:games.csv")    ❹
```

❹ Gets list of games from goal component

```
      .split(body())
    .to("vertx:games");    ❺
```

❺

Splits each game and sends to the vertx event bus

```
    from("goal:goals.csv").routeId("livescore").autoStartup(false)    ❻
```

❻

Routes to stream live goal scores

```
        .log("Goal event: ${header.action} -> ${body}")
        .choice()
          .when(header("action").isEqualTo("clock"))
            .to("vertx:clock")    ❼
```

❼

Updates game clock

```
          .when(header("action").isEqualTo("goal"))
            .to("vertx:goals");    ❽
```

❽

Updates goal score

```
    from("vertx:control").routeId("control")    ❾
```

❾

Routes for control buttons

```
        .log("Control event: ${body}")
        .toD("controlbus:route?routeId=livescore&async=true&action=${body}
    }
}
```

The `LiveScoreRouteBuilder` class is injected with the Vert.x instance ❶ in the constructor because you need to configure the Camel vertx component to use this instance ❷.

Then three Camel routes follow. The first route is used to initialize the list of games ❸. The route calls the goal component ❹, which is responsible for loading the game list from the filesystem. The front end expects one message per game, hence you need to split the game list before sending to the Vert.x event bus on the game's address ❺.

NOTE To hide the complexity of loading the game list and streaming live goal scores, we've built a Camel component named goal.

The second route is responsible for streaming game-clock and goal-score updates ❻ that are routed using a Content-Based Router EIP to either the clock ❼ or goal ❽ address on the Vert.x event bus. Pay attention to the fact that the route has been configured to not automatically start. You want the user to click the Start button at the front end, which is controlled by the last route ❾. The control-bus component is capable of starting and stopping routes. Notice that you refer to the `livescore` route using the `routeId` parameter on the controlbus endpoint.

TIP Chapter 16 covers much more about the control-bus component in its discussion about managing and monitoring Camel.

The `action` parameter tells Camel what to do, such as start, stop, suspend, or resume the route. This action is triggered from the front end using the following JavaScript functions:

```
startClock = function () {
  eb.send("control", "start");
};
stopClock = function () {
  eb.send("control", "suspend");
};
```

The rest of the code for this example is the goal component, which hides the logic to read the CSV files and stream the game clock and goal updates.

We encourage you to take a moment to look at this example and pay attention to how Vert.x and Camel are loosely coupled and clearly separated.

The only touchpoint between them is the exchange of messages using the Vert.x event bus using the camel-vertx component. Figure 20.6 illustrates this principle.

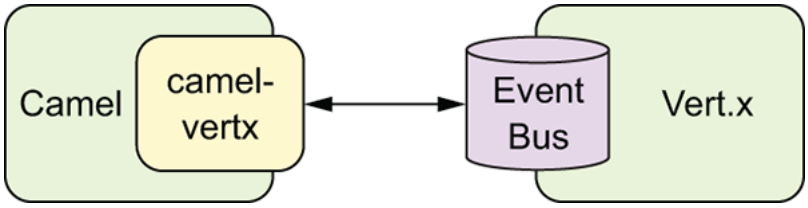

Figure 20.6 Camel and Vert.x exchange messages using the Vert.x event bus. Camel facilitates this using the camel-vertx component.

Trying the example

You can find the source code in the chapter20/vertx-camel directory, and you can try the example using the following Maven goal:

```
mvn compile vertx:run
```

Then, from a web browser, open http://localhost:8080.

Okay, let's end the goal scoring simulator and conclude our Vert.x coverage in this book with some final words.

20.2.3 Summary of using Camel with Vert.x for microservices

Building the goal scoring simulator has been a fun ride, using Vert.x and then later adding Camel to the mix. This kind of application runs well with Vert.x, which has great support for HTML and JavaScript clients. Notice how easy it is to exchange data between the Java back end and the HTML front end using the Vert.x event bus. The web front end is a modern HTML5 single-page application that reacts to a live stream of events. This is where Vert.x shines. How does this compare to Camel? Vert.x is fo-

cused on reactive applications, and Camel is focused on messaging and integration. It's the combination of the two that gives synergy (2 + 2 = 5).

There's a lot of good to say about Vert.x, and we recommend you look at the project and keep an eye on it for the future. Vert.x is reactive, asynchronous, and nonblocking; it puts the burden on the developer to understand its APIs. This takes time to master and grasp. Camel, on the other hand, hides a lot of that complexity. As a developer, you can get far with Camel routes and configuring Camel components and endpoints. Therefore, we recommend you study the Vert.x APIs and programming model if you take up using Vert.x. A good place to start is with the book *Building Reactive Microservices in Java* by Clement Escoffier, which you can free download for free from https://developers.redhat.com/promotions/building-reactive-microservices-in-java.

20.3 Summary and best practices

We're ending this bonus chapter on a high note with coverage of a complex but interesting topic of reactive applications and systems. Although reactive principles and frameworks have been around for many years, they've only recently gained momentum and attention from developers (we're not talking about the ninja developers who jump from hipster technology to hipster technology).

The addition of lambdas and streaming API in Java 8 has also helped Java developers get exposed to and become more familiar with the streaming style of programming. Another popular framework that would push in this direction is the Spring Framework, which include a reactive API from version 5.

As you've seen in this chapter, Apache Camel is also striding into the reactive world with the Reactive Streams component. The Camel team has designs for the Camel 3.*x* architecture to offer two routing engines and APIs:

- *Classic routing engine*—The classic routing engine as of today.
- *Reactive routing engine*—A reactive engine based on the Reactive Streams API and pluggable runtime reactive library such as RxJava or Reactor Core.

By having both routing engines side by side, Camel allows users to pick and choose what best suits their situations. This also allows ample time for the new reactive engine to be developed and matured over the years, with valuable feedback and influence from the community.

We do want to say that reactive streaming APIs and reactive flows can be difficult to learn and understand. If you decide to use this for serious work, make sure to take extra time to learn and experiment.

As usual, here's a bulleted list of the highlights and our thoughts:

- *Reactive applications are complex*—Learning, using, and developing reactive applications is more complex and harder than regular applications. Taming the asynchronous beast isn't easy—especially the RxJava library, which has a lot of greatness but is also harder to get working and understand once you move past the beginner stage. If you become more serious, try to find good online material or a book. We recommend *Reactive Programming with RxJava* by Tomasz Nurkiewicz and Ben Christensen (O'Reilly, 2016).
- *Vert.x has potential*—Keep an eye on the Vert.x project, as it's good and has great potential. It's a small, lightweight tool to build reactive applications. The camel-vertx component enables you to easily use the many Camel components in your Vert.x applications.
- *Reactive, reactive, reactive*—When Docker came out, it was Docker, Docker, Docker. You may hear more and more about reactive systems, streams, and programming. It's not a game changer requiring you to drop everything and rewrite your applications to these new systems, frameworks, and toolkits.

This is the end of the first bonus chapter. The second bonus chapter covers Camel and the Internet of Things (IoT). Neither Claus nor Jonathan is a domain expert on IoT, so we've invited Henryk Konsek to be the guest author for this chapter. This will be the last word you hear from Claus and Jonathan; Henryk, the floor is yours.