REACTIVE PROGRAMMING

Going Reactive with Eclipse Vert.x and RxJava

How to build responsive, scalable apps with one of the most popular reactive libraries.

by Julien Ponge and Clement Escoffier

Eclipse Vert.x is a toolkit for implementing reactive and distributed systems on top of the JVM. It was designed from the start with a reactive design and asynchrony in mind. Vert.x is also about freedom. It does not tell you how to shape your system; you are in charge. Its extensive ecosystem provides everything you need to build responsive, distributed, and interactive applications. This article describes how Vert.x combines an asynchronous execution model and a reactive implementation to let you build applications that can handle uncertain and ever-evolving development needs.

What Does It Mean to Be Reactive?

Let’s start from the beginning: what does reactive actually mean? The Oxford English Dictionary defines reactive as “showing a response to a stimulus.” So, by extension, reactive software can be defined as software that reacts to stimuli. But using that definition, software has been reactive since the early age of computers. Software is designed to react to user demands such as input, clicks, commands, and so on.

However, with the rise of distributed systems, applications started reacting to messages sent by peers and by failure events. The recent reactive renaissance is mainly due to the difficulties of building robust distributed systems. As developers painfully learned, distributed systems are difficult, and they fail for many reasons such as capacity issues, network outages, hardware problems, and bugs. In response, a few years ago, the Reactive Manifesto defined reactive systems as distributed systems with the following characteristics:

- Message-driven: They use asynchronous message passing to communicate.
- Elastic: They stay responsive under varying workloads.
- Resilient: They stay responsive in the face of failure.
- Responsive: They respond in a timely manner.

This architectural style promotes a new way to build distributed systems, infusing asynchrony into the core of these systems. While reactive systems are described as “distributed systems done right,” they can be difficult to build. Taming the asynchronous beast is particularly difficult from the developer standpoint. In addition, the traditional threading model (one thread per request) tends to create memory and CPU hogs, and, when dealing with asynchronous code, this approach is particularly inefficient.

Several development models have emerged to make the development of asynchronous applications easier, including actors, fibers, coroutines, and reactive programming. This article focuses on the latter.
Reactive programming (and its main derivative, Reactive eXtensions, or RX) is an asynchronous programming paradigm focused on the manipulation of data streams. It provides an API to compose asynchronous and event-driven applications. When using reactive programming, you are handling streams of data in which data flows. You observe these streams and react when new data is available.

But data streams have an inherent flaw. What happens if you receive too many messages and you can't process them in time? You could put a buffer between the source and the handler, but it would help only with handling small bumps. Dropping incoming data is also a solution, but that is not always acceptable. Ultimately, you need a way to control the pace. This is what the reactive streams specification proposes. It defines an asynchronous and nonblocking back-pressure protocol. In this flow of control, the consumer notifies the producer of its current capacity. So, the producer does not send too much data on the stream, and your system auto-adapts to its capacity without burning.

Why Do Reactive Systems Matter?

Why did reactive programming become so prevalent in the past few years? For a very long time, most applications have been developed using a synchronous execution model and most APIs have been designed to follow this approach.

However, computer systems and distribution systems are asynchronous. Synchronous processing is a simplification made to provide ease of comprehension. For years, the asynchronous nature of systems has been ignored, and now it's time to catch up. Many modern applications are relying on I/O operations, such as remote invocations or access to the file system. Because of the synchronous nature of application code, however, these I/O operations are designed to be blocking, so the application waits for a response before it can continue its execution. To enable concurrency, the application relies on multithreading and increases the number of threads. But, threads are expensive. First, the code has to protect itself from concurrent access to its state. Second, threads are expensive in terms of memory and—often overlooked—in CPU time, because switching between threads requires CPU cycles.

Implementing reactive systems requires two shifts: an execution shift to use an asynchronous execution model and a development shift to write asynchronous APIs and applications.

Therefore, a more efficient model is needed. The asynchronous execution model promotes a task-based concurrency in which a task releases the thread when it cannot make progress anymore (for instance, it invokes a remote service using nonblocking I/O and will be notified when the result is available). Thus, the same thread can switch to another task. As a result, a single thread can handle several interleaved tasks.

Traditional development and execution paradigms are not able to exploit this new model. However, in a world of cloud and containers, where applications are massively distributed and interconnected and they must handle continuously growing traffic, the promise made by reactive systems is a perfect match. But, implementing reactive systems requires two shifts: an execution shift to use an asynchronous execution model and a development shift to write asynchronous APIs and applications. This is what Eclipse Vert.x offers. In the rest of this article, we present how Vert.x combines both to give you superpowers.

RxJava: The Reactive Programming Toolbox for Java

Let’s focus on reactive programming—a development model for writing asynchronous code. When using reactive programming, the code manipulates streams of data. The data is generated by publishers. The data flows between a publisher and consumers, which process the data.
Consumers observing a data stream are notified when a new item is available, when the stream completes, and when an error is caught. To avoid overloading consumers, a back-pressure protocol is required to control the amount of data flowing in the stream. This is generally handled transparently by the reactive framework.

There are several implementations of the reactive programming paradigm. RxJava is a straightforward implementation of reactive extensions (RX) for the Java programming language. It is a popular library for reactive programming that can be used to develop applications in networked data processing, graphical user interfaces with JavaFX, and Android apps. RxJava is the principal toolkit for reactive libraries in Java, and it provides five data types to describe data publishers depending on the types of data streams, as shown in Table 1.

<table>
<thead>
<tr>
<th>USE CASE</th>
<th>NUMBER OF EXPECTED ITEMS IN THE STREAM</th>
<th>RXJAVA TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous data flow</td>
<td>0..N</td>
<td>Observable, Flowable</td>
</tr>
<tr>
<td>Asynchronous operation producing (maybe) a result</td>
<td>1..1</td>
<td>Single</td>
</tr>
<tr>
<td>Asynchronous operation that does not produce a result</td>
<td>0</td>
<td>Completable</td>
</tr>
</tbody>
</table>

Table 1. RxJava reactive publisher types

These types represent data publishers and convey data processed by consumers observing them. Depending on the number of items flowing in the stream, the type is different. For streams with a bounded or unbounded sequence of items, the types Observable and Flowable are used.

The difference between Observable and Flowable is that Flowable handles back-pressure (that is, it implements a reactive streams protocol) while Observable does not. Flowable is better suited for large streams of data coming from a source that supports back-pressure (for example, a TCP connection), while Observable is better suited at handling so-called “hot” observables for which back-pressure cannot be applied (such as GUI events and other user actions). It is important to note that not all streams can support back-pressure. In fact, most of the streams conveying data captured in the physical world are not capable of this. Reactive programming libraries propose strategies such as buffers and acceptable data loss for handling these cases.

Getting started with RxJava. It’s time to see some code and make reactive clearer. The complete project source code is available online. Clone or download this project and check the content of the rxjava-samples subproject. It uses RxJava 2.x and the logback-classic logging library. You will see later how it helps you understand threading with RxJava.

In the previous section, we briefly examined the different reactive types proposed by RxJava. The following class creates instances of these types and applies some basic operations:

```java
package samples;

import io.reactivex.Completable;
import io.reactivex.Flowable;
import io.reactivex.Maybe;
import io.reactivex.Single;
import io.reactivex.functions.Consumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class RxHello {

    private static final Logger logger = LoggerFactory.getLogger(RxHello.class);

    public static void main(String[] args) {
        Single.just(1)
            .map(i -> i * 10)
            .map(Object::toString)
```
It is important to note that as with Java collection streams, no processing happens until an end event takes place. In RxJava, that event is a subscription. In this example, we used subscribe() with a single parameter, which is a lambda called to receive each event. The following are other forms of Subscribe depending on the events the consumer wants to receive:

- No arguments, which just triggers the processing
- Two arguments to process events and errors
- Three arguments to process events, to process errors, and to provide notification when the processing is complete

**Creating publishers and recovering from errors.** Of course, RxJava would be quite limited if creating data streams such as Observables were limited to calling the just() factory method as we did in the previous example. All types of publishers support a create() method to define the code to deal with new subscribers:

```java
List<String> data = Arrays.asList("foo", "bar", "baz");
Random random = new Random();

Observable<String> source = Observable.create(subscriber -> {
    for (String s : data) {
        if (random.nextInt(6) == 0) {
            subscriber.onError(new RuntimeException("Bad luck");
        } else {
            subscriber.onNext(s);
        }
    }
    subscriber.onComplete();
});
```

The example above creates an Observable of String values (in other words, a stream of String values), where the values are being picked from a predefined list. We also introduced random failures. The following three methods can be used to notify subscribers:

- `onNext`, when a new value is sent to the subscriber, possibly passing through intermediate operators before it reaches the subscriber
- `onComplete` to indicate that no more values will be sent
- `onError` to indicate that an error happened and that no further value will be sent; any Throwable can be used as an error value
Note that `create()` is not the only way to define custom publishers, but presenting all options would be outside the scope of this article.

Because there is a good probability that errors will happen, we can test this observable 10 times:

```java
for (int i = 0; i < 10; i++) {
    logger.info("===================================");
    source.subscribe(
        next -> logger.info("Next: {}", next),
        error -> logger.error("Whoops"),
        () -> logger.info("Done");
}
```

We can observe successful completions as well as errors in the execution traces:

```
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
11:51:47.469 [main] ERROR samples.RxCreateObservable
11:51:47.469 [main] INFO samples.RxCreateObservable
```

RxJava supports various ways to recover from errors, such as switching to another stream or providing a default value. Another option is to use `retry()`:

```java
source.retry(5)
    .subscribe(next -> logger.info("Next: {}", next),
               error -> logger.error("Whoops"),
               () -> logger.info("Done");
```

Above, we specified that in case of error, we should retry at most five times with new subscriptions. Note that retries might use another thread for execution. Because errors are random, your exact output trace will vary across executions, but the following output shows an example of retries:

```
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
11:51:47.472 [main] ERROR samples.RxCreateObservable
11:51:47.472 [main] INFO samples.RxCreateObservable
```

RxJava and threads. So far, we haven’t cared much about multithreading. Let’s take another example and run it:

```java
Flowable.range(1, 5)
    .map(i -> i * 10)
    .map(i -> {
        logger.info("map({})", i);
        return i.toString();
    })
    .subscribe(logger::info);
Thread.sleep(1000);
```
You can see from the logs that all processing happens on the main thread:

```
12:01:01.097 [main] INFO samples.RxThreading - map(1
12:01:01.100 [main] INFO samples.RxThreading - 10
12:01:01.100 [main] INFO samples.RxThreading - map(2
12:01:01.100 [main] INFO samples.RxThreading - 20
12:01:01.100 [main] INFO samples.RxThreading - map(3
12:01:01.100 [main] INFO samples.RxThreading - 30
12:01:01.100 [main] INFO samples.RxThreading - map(4
12:01:01.100 [main] INFO samples.RxThreading - 40
12:01:01.100 [main] INFO samples.RxThreading - map(5
12:01:01.100 [main] INFO samples.RxThreading - 50
```

In fact, both the operator processing and the subscriber notifications happen from that main thread. By default, a publisher (and the chain of operators that you apply to it) will do its work, and will notify its consumers, on the same thread on which its subscribe method is called. RxJava offers Schedulers to offload work to specialized threads and executors. Schedulers are responsible for notifying the subscribers on the correct thread even if it’s not the thread used to call subscribe.

The io.reactivex.schedulers.Schedulers class offers several schedulers, with the most interesting being these:

- `computation()` for CPU-intensive work with no blocking I/O operations
- `io()` for all blocking I/O operations
- `single()`, which is a shared thread for operations to execute in order
- `from(executor)` to offload all scheduled work to a custom executor

Now, back to our previous example, we can specify how the subscription and observation will be scheduled:

```java
Flowable.range(1, 5)
.map(i -> i * 10)
.map(i -> {
    logger.info("map({}", i);
    return i.toString();
})
.observeOn(Schedulers.single())
.subscribeOn(Schedulers.computation())
.subscribe(logger::info);
Thread.sleep(1000);
logger.info("===================================");
```

The subscribeOn method specifies the scheduling for the subscription and operator processing, while the observeOn method specifies the scheduling for observing the events. In this example, the `map` operations are invoked on the computation thread pool while the `subscribe` callback (logger::info) is invoked by a different thread (which does not change). Running the example gives an execution trace where you clearly see different threads being involved:

```
12:01:03.127 [RxComputationThreadPool-1] INFO samples.RxThreading - map(10)
12:01:03.128 [RxComputationThreadPool-1] INFO samples.RxThreading - map(20)
12:01:03.128 [RxSingleScheduler-1] INFO samples.RxThreading - 10
12:01:03.128 [RxComputationThreadPool-1] INFO samples.RxThreading - map(30)
12:01:03.128 [RxSingleScheduler-1] INFO samples.RxThreading - 20
12:01:03.128 [RxComputationThreadPool-1] INFO samples.RxThreading - map(40)
12:01:03.128 [RxSingleScheduler-1] INFO samples.RxThreading - 30
```
Combining observables. RxJava offers many ways to combine streams. We'll illustrate that with the `merge` and `zip` operations. Merging streams provides a single stream that mixes elements from the various sources, as the following example shows:

```java
package samples;

import io.reactivex.Flowable;
import io.reactivex.schedulers.Schedulers;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.UUID;
import java.util.concurrent.TimeUnit;

public class RxMerge {
    private static final Logger logger = LoggerFactory.getLogger(RxMerge.class);

    public static void main(String[] args) throws InterruptedException {
        Flowable<String> intervals = Flowable
            .interval(100, TimeUnit.MILLISECONDS,
                    Schedulers.computation())
            .limit(10)
            .map(tick -> "Tick #" + tick)
            .subscribeOn(Schedulers.computation());

        Flowable<String> strings = Flowable.just(
            "abc", "def", "ghi", "jkl"
        ).subscribeOn(Schedulers.computation());

        Flowable<Object> uuids = Flowable
            .generate(emitter -> emitter.onNext(UUID
                .limit(10)
                .subscribeOn(Schedulers.computation()));

        Flowable.merge(strings, intervals, uuids)
            .subscribe(obj -> logger.info("Received: {}" , obj));

        Thread.sleep(3000);
    }
}
```

Running this example gives a trace in which elements from the various sources may be interleaved. Another useful option is `zip()`, which takes elements from various sources and assembles them:

```java
Flowable.zip(intervals, uuids, strings,
        (i, u, s) -> String.format("%s {%s} -> %s", i, u
            .subscribe(obj -> logger.info("Received: {}" , obj));
```

It produces a trace similar to this:

```
14:32:40.127 [RxComputationThreadPool-7] INFO samples.RxMerge - Received: Tick #0
{67e7cde0-3f29-49cb-b569-e01474676d98} -> abc
14:32:40.224 [RxComputationThreadPool-7] INFO samples.RxMerge - Received: Tick #1
{a0a0cc83-4bed-4793-9ee0-11baa7707610} -> def
14:32:40.324 [RxComputationThreadPool-7] INFO samples.RxMerge - Received: Tick #2
```
In real-world scenarios, zip() is useful for gathering data from other parties, such as services, and then producing a result based on what was received.

**Implementing Reactive Systems with Reactive Programming**

While reactive programming lets you compose asynchronous and event-driven applications, don’t lose sight of the overall goal. To successfully build responsive distributed systems in a world of cloud and containers, embracing the asynchronous execution model is essential. Reactive programming addresses the asynchronous development model, but you still need a task-based concurrency model and nonblocking I/O. Eclipse Vert.x provides these two missing pieces as well as RxJava-friendly APIs.

The Vert.x execution model is based on the concept of an event loop. An event loop is a thread consuming events from a queue. For each event, it looks for a handler interested in the event and calls it. Handlers are methods that receive an event as a parameter. In this model, your code can be single-threaded while handling lots of concurrent and entangled tasks. However, this approach comes with some drawbacks. The executed handlers must never block the event loop: if they do, the system loses its responsiveness and the number of unprocessed events in the queue rises.

Fortunately, Vert.x comes with a large ecosystem for implementing almost anything in an asynchronous and nonblocking way. For instance, Vert.x provides building blocks for building modern web applications, accessing databases, and interacting with legacy systems. Let’s look at a few examples. The Vert.x “hello world” application (code available online) is the following:

```java
package samples;
import io.vertx.core.Vertx;

public class HttpApplication {
    public static void main(String[] args) {
        // 1 - Create a Vert.x instance
        Vertx vertx = Vertx.vertx();

        // 2 - Create the HTTP server
        vertx.createHttpServer()
            .requestHandler(req -> req.response()
                .end("Hello, request handled from Thread.currentThread().getName()");

        // 4 - Start the server on the port 8080
        .listen(8080);
    }
}
```

For each incoming HTTP request (event), the request handler is called. Notice that the handler is always called by the same thread: the event loop thread. Now, if you want to call another service (using HTTP) in the request handler, you would do something like this:

```java
package samples;
import io.vertx.core.Vertx;
import io.vertx.ext.web.client.WebClient;

public class TwitterFeedApplication {
    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
```
This example relies on the Vert.x nonblocking I/O, so the entire code runs on the Vert.x event loop (in a single-thread manner). This does not prevent handling concurrent requests. It's actually the opposite; a single thread handles all the requests. However, you can quickly see the issue: the code becomes difficult to understand because of the nested callbacks. This is where RxJava comes into play. The previous code can be rewritten as follows:

```java
package samples;

import io.vertx.reactivex.core.Vertx;
import io.vertx.reactivex.core.http.HttpServer;
import io.vertx.reactivex.ext.web.client.HttpResponse;
import io.vertx.reactivex.ext.web.client.WebClient;

public class RXTwitterFeedApplication {
    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
        WebClient client = WebClient.create(vertx);
        HttpServer server = vertx.createHttpServer(
            server
                .requestStream().toFlowable()
                .flatMapCompletable(req ->
                    client.getAbs("https://twitter.com/ver
                        .rxSend()
                    // 3 - Extract the body as string
                        .map(HttpResponse::bodyAsString)
                        .onErrorReturn(t -> "Cannot acc
                            "feed: " + t.getMessage())
                    // 4 - In case of a failure
                        .doOnSuccess(res -> req.respo
                            // 5 - Write the response
                            .doOnSuccess(res -> res.caus
                            // 6 - Just transform the result
                            .toCompletable()
                    )
                )
        server.listen(8080);
    }
}
```

By restructuring the code around the RxJava reactive types, you benefit from the RxJava operators.

**Implementing a Reactive Edge Service**

Let's look at another simple yet effective example. Suppose that you have three services offering bids, and you want to offer an edge service
to select the best offer at a point in time. Let these services offer simple
HTTP/JSON endpoints. Obviously in real-world scenarios, these services
might fail temporarily, and their response times might greatly vary.

We will simulate such a system by developing the following:

- A bidding service, with artificial delays and random errors
- An edge service to query services through HTTP

By using RxJava, we can show how to combine request streams, deal
with failures, and provide time-bound guarantees for returning the best
offer. All verticles will be deployed within the same application as we are
prototyping, but this does not result in any loss of generality. The
complete code is available in the vertx-samples subproject.

Instead of starting the application using a main method, we are going to
use verticles. A verticle is a chunk of code, generally a Java class, that is
deployed and run by Vert.x. Verticles are simple and scalable, and they
use an actor-like deployment and concurrency model. They let you
organize your code into a set of loosely coupled components. By default,
verticles are executed by the event loop and observe different types of
events (HTTP requests, TCP frames, messages, and so on). When the
application starts, it instructs Vert.x to deploy a set of verticles.

**Bidding service verticle.** The verticle is designed with the HTTP port
being configurable, as follows:

```java
public class BiddingServiceVerticle extends AbstractVerticle {

private final Logger logger = LoggerFactory.getLogger(BiddingServiceVerticle.class);

@Override
public void start(Future<Void> verticleStartFuture) {
    Random random = new Random();
    String myId = UUID.randomUUID().toString();
    int portNumber = config().getInteger("port")
        // (...)
}
```

The `config()` method provides access to a verticle configuration, and
accessor methods such as `getInteger` support a default value as a
second argument. So here, the default HTTP port is 3000. The service
has a random UUID to identify its endpoint in responses, and it makes
use of a random number generator.

The next step is to use the Vert.x web router to accept HTTP GET
requests on path `/offer`:

```java
Router router = Router.router(vertx);
router.get("/offer").handler(context -> {
    String clientIdHeader = context.request().getHeader("Client-Request-Id");
    String clientId =
        (clientIdHeader != null) ? clientIdHeader :
        int myBid = 10 + random.nextInt(20);
    JsonObject payload = new JsonObject()
        .put("origin", myId)
        .put("bid", myBid);
    if (clientIdHeader != null) {
        payload.put("clientRequestId", clientId);
    }
    long artificialDelay = random.nextInt(1000);
    vertx.setTimer(artificialDelay, id -> {
        if (random.nextInt(20) == 1) {
            context.response()
                .setStatusCode(500)
                .end();
            logger.error("{} injects an error (clientRequestId={}",
                myId, myBid, clientId, artificialDelay=");
        } else {
            // (more logic)
        }
    });
});
```

The bidding service verticle uses artificial delays and random failures
to simulate a real-world scenario. The `artificialDelay` introduces a delay
between the request and the response, and the `random.nextInt(20)`
generates a random number to simulate failures. The `context.response()
.setStatusCode(500)` sets the HTTP status code to 500, indicating an error.

This example demonstrates how to handle failures and provide time-bound
guarantees in a real-world scenario using Vert.x and RxJava.
Note that to simulate failures, we built in a 5 percent chance of failure (in which case, the service issues an HTTP 500 response) and the final HTTP response is delayed by using a random timer between 0 and 1,000 milliseconds.

Finally, the HTTP server is started as usual:

```java
context.response()
    .putHeader("Content-Type", "application/json")
    .end(payload.encode());
logger.info("{} offers {} (client-id={}) artificialDelay={}", myId, myBid, clientId, artificialDelay);
```
There are several interesting points in this code:

- To access the RxJava API offered by Vert.x, we import and extend the `io.vertx.reactivex.core.AbstractVerticle` class.
- It is possible to specify the target services, with the defaults being on the local host and ports 3000, 3001, and 3002. Such configuration can be passed as a JSON array containing JSON objects with `host`, `port`, and `path` keys.
- Variants of the Vert.x APIs that return RxJava objects are prefixed with "rx": here `rxListen` returns a `Single<HttpServer>`.

We can now focus on the implementation of the `findBestOffer` method. It first issues HTTP requests to each service, obtaining a list of `Single<JsonObject>` responses, and then it reduces them to the single, best response and eventually ends the HTTP response:

```java
private final AtomicLong requestIds = new AtomicLong();
private static final JsonObject EMPTY_RESPONSE = new JsonObject()
    .put("empty", true)
    .put("bid", Integer.MAX_VALUE);

private void findBestOffer(HttpServerRequest request) {
    String requestId = String.valueOf(requestIds.get());
    List<Single<JsonObject>> responses = targets.stream()
        .map(t -> webClient
            .get(t.getInteger("port"),
            t.getString("host"),
            t.getString("path"))
            .putHeader("Client-Request-Id",
            String.valueOf(requestId))
            .as(BodyCodec.jsonObject())
            .rxSend()
            .retry(1)
            .timeout(500, TimeUnit.MILLISECONDS,
            RxHelper.scheduler(vertx))
            .map(HttpResponse::body)
            .map(body -> {
                logger.info("#{} received offer {}",
                body.encodePrettily());
                return body;
            }).onErrorReturnItem(EMPTY_RESPONSE))
        .collect(Collectors.toList());

    Single.merge(responses)
        .reduce((acc, next) -> {
            if (next.containsKey("bid") && isHigher
                return next;
            return acc;
        })
        .flatMapSingle(best -> {
            if (!best.containsKey("empty")) {
                return Single.just(best);
            } else {
                return Single.error(new Exception("N" +
                    "could be found for requestId=
                ");
            }
        })
        .subscribe(best -> {
            logger.info("#{} best offer: {}", request
            best.encodePrettily());
            request.response()
            .putHeader("Content-Type",
                "application/json")
            .end(best.encode());
        }, error -> {
            logger.error("#{} ends in error", request
            request.response()
            .setStatusCode(502)
            .end();
```

To access the RxJava API offered by Vert.x, we import and extend the `io.vertx.reactivex.core.AbstractVerticle` class.
It is interesting to note the following for each HTTP request:

- The response is converted to a `JsonObject` using the `as()` method.
- A retry is attempted if the service issued an error.
- The processing times out after 500 milliseconds before returning an empty response, which is how we avoid waiting for all responses and errors to arrive.

Note that all RxJava operations that expect a scheduler can use `RxHelper::scheduler` to ensure that all events remain processed on Vert.x event loops.

The whole processing is just a matter of composing functional idioms such as `map`, `flatMap`, and `reduce` and handling errors with a default value. If no service can deliver a bid within 500 milliseconds, no offer is being made, resulting in an HTTP 502 error. Otherwise, the best offer is selected among the responses received.

### Deploying verticles and interacting with the services

The main verticle code is as follows:

```java
public class MainVerticle extends AbstractVerticle {
  @Override
  public void start() {
    vertx.deployVerticle(new BiddingServiceVerticle
                        new JsonObject().put("p
    vertx.deployVerticle(new BiddingServiceVerticle
                        new DeploymentOptions().set
    vertx.deployVerticle("samples.BestOfferServiceVe
                        new DeploymentOptions().set
    }
}
```

We deploy the bidding service three times on different ports to simulate three services, passing the HTTP port those services should listen on in the JSON configuration. We also deploy the edge service verticle with two instances to process the incoming traffic on two CPU cores rather than one. The two instances will listen on the same HTTP port, but note that there will be no conflict because Vert.x distributes the traffic in a round-robin fashion.

We can now interact with the HTTP services, for instance, by using the HTTPie command-line tool. Let's talk to the service on port 3000:

```bash
$ http GET localhost:3000/offer 'Client-Request-Id:1
GET /offer HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Client-Request-Id: 1234
Connection: keep-alive
Host: localhost:3000
User-Agent: HTTPie/0.9.9

HTTP/1.1 200 OK
Content-Length: 83
Content-Type: application/json

{
  "bid": 21,
  "clientRequestId": "1234",
  "origin": "fe299565-34be-4a7b-ac09-d88fcc1e4:
```
The logs reveal both artificial delays and errors:

```
[INFO] 16:08:03.443 [vert.x-eventloop-thread-1] ERROR samples.BiddingServiceVerticle - 6358300b-3f2d-40be-93db-789f0f1cde17 injects an error (client-id=1234, artificialDelay=N/A)
```

Similarly, you can play with the edge service, observe responses, and check the logs to see how a response is being assembled. Sometimes you will get an error:

```
$ http GET localhost:8080 'Client-Request-Id:1234'
HTTP/1.1 502 Bad Gateway
Content-Length: 0
```

This is because all responses took longer than 500 milliseconds to arrive and some services injected an error:

```
```

Sometimes you will observe that only one or two responses have been taken into account.

The key point in this sample is that the combination of Vert.x and RxJava offers a declarative and functional model for describing how to perform and process a flexible number of network requests while remaining purely driven by asynchronous events.

**Conclusion**

In this article, you have seen how Eclipse Vert.x combines reactive programming and the asynchronous execution model to build reactive systems. Reactive programming lets you compose asynchronous and event-driven applications by manipulating and combining data streams. Modern reactive programming libraries such as RxJava implement reactive streams to handle back-pressure. However, a reactive approach is not limited to reactive programming. Don’t lose sight that you want to build better systems that are responsive, robust, and interactive. By using the execution model and nonblocking I/O capabilities promoted by Vert.x, you are on the path to becoming truly reactive.

This article just scratched the surface. Vert.x gives you significant power and agility to create compelling, scalable, twenty-first-century applications the way you want to. Whether it’s simple network utilities, sophisticated modern web applications, HTTP/REST microservices, high-volume event processing, or a full-blown back-end message-bus application, Vert.x is a great fit.

This article was originally published in the January/February 2018 issue of Java Magazine.
Julien Ponge
Julien Ponge (@jponge) is an associate professor at INSA Lyon and a researcher at the CITI-INRIA laboratory. He is a longtime open source developer, having created IzPack and the Golo programming language, and is now a member of the Eclipse Vert.x team. Ponge is currently on leave from INSA and working as a delegated consultant to Red Hat on the Vert.x project.

Clement Escoffier
Clement Escoffier (@clementplop) is a principal software engineer at Red Hat, where he is working as a Vert.x core developer. He has been involved in projects and products touching many domains and technologies such as OSGi, mobile app development, continuous delivery, and DevOps. Escoffier is an active contributor to many open source projects, including Apache Felix, iPOJO, Wisdom Framework, and Eclipse Vert.x.