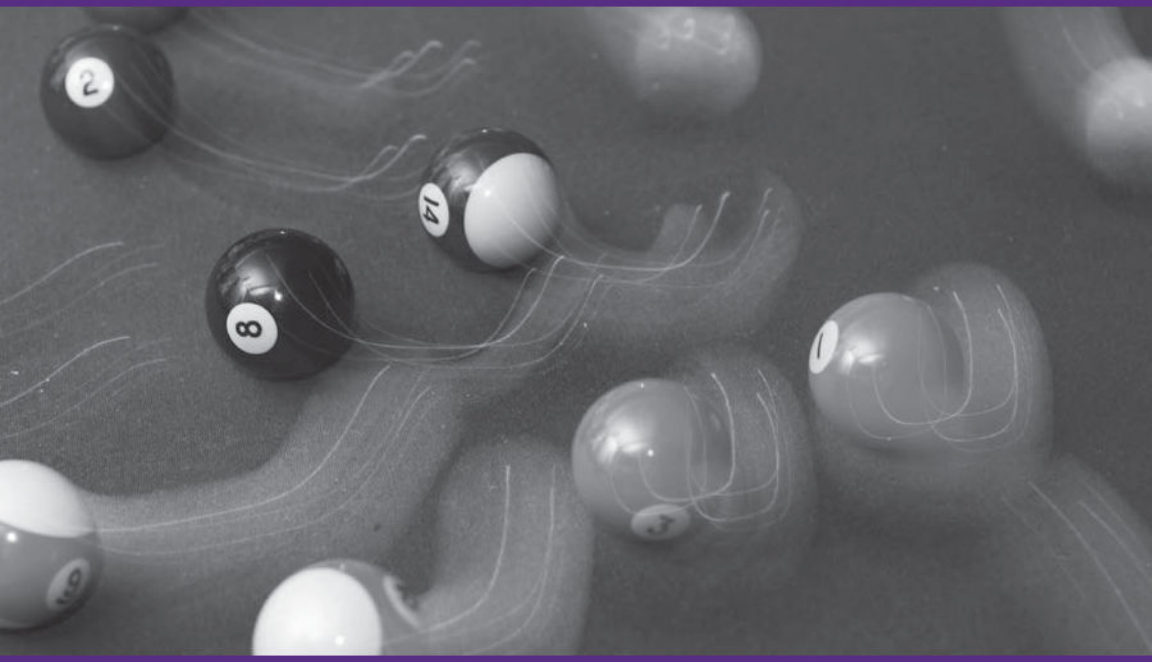
O'REILLY®

Building Reactive Microservices in Java

Asynchronous and Event-Based Application Design



Clement Escoffier

Building Reactive Microservices in Java

by Clement Escoffier

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster

Production Editor: Shiny Kalapurakkel

Copyeditor: Christina Edwards

Proofreader: Sonia Saruba

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

May 2017: First Edition

Revision History for the First Edition

- 2017-04-06: First Release

978-1-491-98628-8

[LSI]

Chapter 1. Introduction

This report is for developers and architects interested in developing microservices and distributed applications. It does not explain the basics of distributed systems, but instead focuses on the *reactive* benefits to build efficient microservice systems. Microservices can be seen as an extension of the basic idea of modularity: programs connected by message-passing instead of direct API calls so that they can be distributed among multiple services. Why are microservices so popular? It's basically due to the combination of two factors: cloud computing and the need to scale up and down quickly. Cloud computing makes it convenient to deploy thousands of small services; scaling makes it necessary to do so.

In this report, we will see how Eclipse Vert.x (*http://vertx.io*) can be used to build reactive microservice systems. Vert.x is a toolkit to build reactive and distributed systems. Vert.x is incredibly flexible. Because it's a toolkit, you can build simple network utilities, modern web applications, a system ingesting a huge amount of messages, REST services, and, obviously, microservices. This malleability gives Vert.x great popularity, a large community, and a vibrant ecosystem. Vert.x was already promoting *microservices* before it became so popular. Since the beginning, Vert.x has been tailored to build applications composed by a set of distributed and autonomous services. Systems using Vert.x are built upon the reactive system principles (*http://reactivemanifesto.org*). They are responsive, elastic, resilient, and use asynchronous message passing to interact.

This report goes beyond Vert.x and microservices. It looks at the whole environment in which a microservice system runs and introduces the many tools needed to get the desired results. On this journey, we will learn:

- What Vert.x is and how you can use it
- What *reactive* means and what reactive microservices are
- How to implement microservices using HTTP or messages
- The patterns used to build reactive microservice systems

- How to deploy microservices in a virtual or cloud environment

The code presented in this report is available from *https://github.com/redhat-developer/reactive-microservices-in-java*.

Preparing Your Environment

Eclipse Vert.x requires Java 8, which we use for the different examples provided in this report. We are going to use Apache Maven to build them. Make sure you have the following prerequisites installed:

- JDK 1.8
- Maven 3.3+
- A command-line terminal (Bash, PowerShell, etc.)

Even if not mandatory, we recommend using an IDE such as the Red Hat Development Suite (*https://developers.redhat.com/products/devsuite/overview*). In the last chapter, we use OpenShift, a container platform built on top of Kubernetes (*https://kubernetes.io*) to run containerized microservices. To install OpenShift locally, we recommend Minishift (*https://github.com/minishift/minishift*) or the Red Hat Container Development Kit (CDK) v3. You can download the CDK from *https://developers.redhat.com/products/cdk/download*.

Let’s get started.

Chapter 2. Understanding Reactive Microservices and Vert.x

Microservices are not really a *new* thing. They arose from research conducted in the 1970s and have come into the spotlight recently because microservices are a way to move faster, to deliver value more easily, and to improve *agility*. However, microservices have roots in actor-based systems, service design, dynamic and autonomic systems, domain-driven design, and distributed systems. The fine-grained modular design of microservices inevitably leads developers to create distributed systems. As I'm sure you've noticed, distributed systems are hard. They fail, they are slow, they are bound by the CAP and FLP theorems. In other words, they are very complicated to build and maintain. That's where *reactive* comes in.

30+ YEARS OF EVOLUTION

The actor model was introduced by C. Hewitt, P. Bishop, and R. Steiger in 1973. Autonomic computing, a term coined in 2001, refers to the self-managing characteristics (self-healing, self-optimization, etc.) of distributed computing resources.

But what is *reactive*? Reactive is an overloaded term these days. The Oxford dictionary defines *reactive* as "*showing a response to a stimulus*." So, reactive software reacts and adapts its behavior based on the stimuli it receives. However, the responsiveness and adaptability promoted by this definition are programming challenges because the flow of computation isn't controlled by the programmer but by the stimuli. In this chapter, we are going to see how Vert.x helps you be *reactive* by combining:

- *Reactive* programming—A development model focusing on the observation of data streams, reacting on changes, and propagating them
- *Reactive* system—An architecture style used to build responsive and robust distributed systems based on asynchronous message-passing

A *reactive microservice* is the building block of reactive microservice systems. However, due to

their asynchronous aspect, the implementation of these microservices is challenging. Reactive programming reduces this complexity. How? Let's answer this question right now.

Reactive Programming

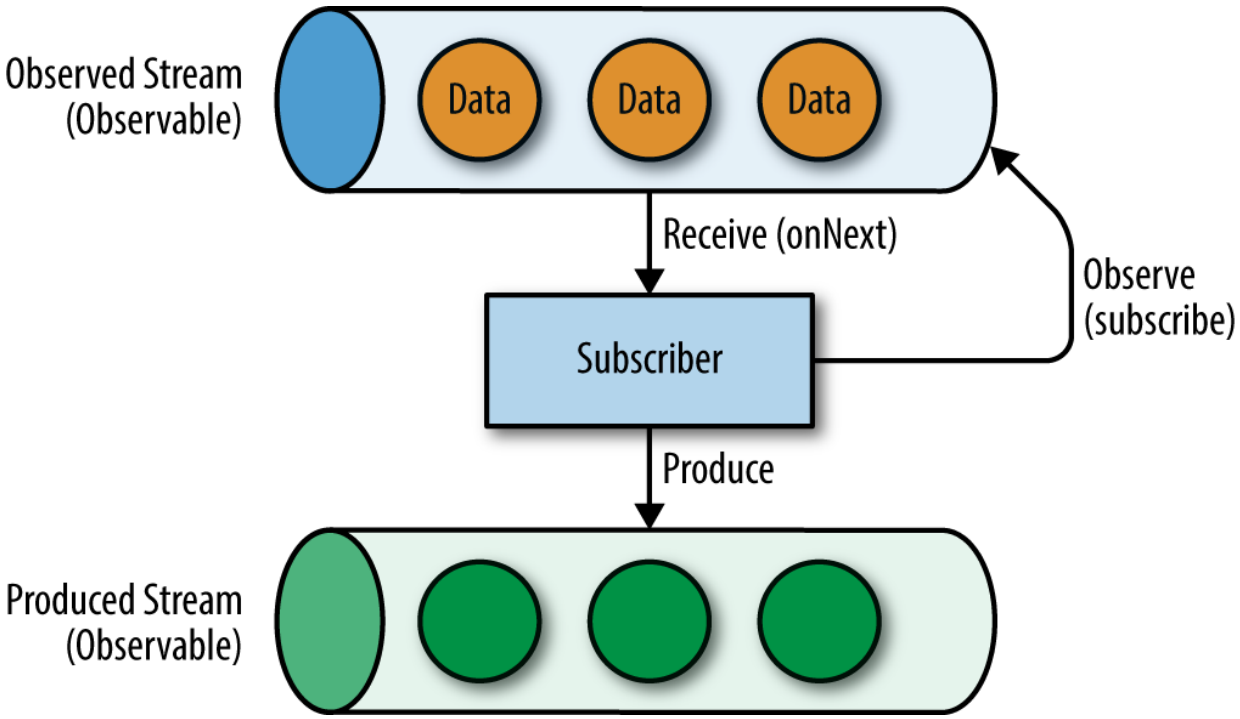


Figure 2-1. Reactive programming is about flow of data and reacting to it

Reactive programming is a development model oriented around data flows and the propagation of data. In *reactive programming*, the stimuli are the data transiting in the flow, which are called *streams*. There are many ways to implement a reactive programming model. In this report, we are going to use Reactive Extensions (*http://reactivex.io/*) where *streams* are called *observables*, and consumers *subscribe* to these *observables* and react to the values (Figure 2-1).

To make these concepts less abstract, let's look at an example using RxJava (*https://github.com/ReactiveX/RxJava*), a library implementing the Reactive Extensions in Java. These examples are located in the directory `reactive-programming` in the code repository.

```
observable.subscribe(
  data -> { // onNext
    System.out.println(data);
  },
  error -> { // onError
    error.printStackTrace();
  },
  () -> { // onComplete
    System.out.println("No more data");
  }
```

```
  );
```

In this snippet, the code is observing (`subscribe`) an `Observable` and is notified when values transit in the flow. The subscriber can receive three types of events. `onNext` is called when there is a new value, while `onError` is called when an error is emitted in the stream or a stage throws an `Exception`. The `onComplete` callback is invoked when the end of the stream is reached, which would not occur for unbounded streams. RxJava includes a set of operators to produce, transform, and coordinate `Observables`, such as `map` to transform a value into another value, or `flatMap` to produce an `Observable` or chain another asynchronous action:

```
// sensor is an unbound observable publishing values.
sensor
  // Groups values 10 by 10, and produces an observable
  // with these values.
  .window(10)
  // Compute the average on each group
  .flatMap(MathObservable::averageInteger)
  // Produce a json representation of the average
  .map(average -> "{'average': " + average + "}")
  .subscribe(
    data -> {
      System.out.println(data);
    },
    error -> {
      error.printStackTrace();
    }
  );
```

RxJava v1.x defines different types of streams as follows:

- `Observables` are bounded or unbounded streams expected to contain a sequence of values.
- `Singles` are streams with a single value, generally the deferred result of an operation, similar to futures or promises.
- `Completables` are streams without value but with an indication of whether an operation completed or failed.

RXJAVA 2

While RxJava 2.x has been recently released, this report still uses the previous version (RxJava 1.x). RxJava 2.x provides similar concepts. RxJava 2 adds two new types of streams. `Observable` is used for streams not supporting back-pressure, while `Flowable` is an `Observable` with back-pressure. RxJava 2 also introduced the `Maybe` type, which models a stream where there could be 0 or 1 item or an error.

What can we do with RxJava? For instance, we can describe sequences of asynchronous actions and orchestrate them. Let's imagine you want to download a document, process it, and upload it. The download and upload operations are asynchronous. To develop this sequence, you use something like:

```
// Asynchronous task downloading a document
Future<String> downloadTask = download();
// Create a single completed when the document is downloaded.
Single.from(downloadTask)
  // Process the content
  .map(content -> process(content))
  // Upload the document, this asynchronous operation
  // just indicates its successful completion or failure.
  .flatMapCompletable(payload -> upload(payload))
  .subscribe(
  () -> System.out.println("Document downloaded, updated
              and uploaded"),
  t -> t.printStackTrace()
);
```

You can also orchestrate asynchronous tasks. For example, to combine the results of two asynchronous operations, you use the `zip` operator combining values of different streams:

```
// Download two documents
Single<String> downloadTask1 = downloadFirstDocument();
Single<String> downloadTask2 = downloadSecondDocument();

// When both documents are downloaded, combine them
Single.zip(downloadTask1, downloadTask2,
  (doc1, doc2) -> doc1 + "\n" + doc2)
  .subscribe(
    (doc) -> System.out.println("Document combined: " + doc),
    t -> t.printStackTrace()
  );
```

The use of these operators gives you superpowers: you can coordinate asynchronous tasks and

data flow in a declarative and elegant way. How is this related to *reactive microservices*? To answer this question, let's have a look at *reactive systems*.

REACTIVE STREAMS

You may have heard of reactive streams (*http://www.reactive-streams.org/*). Reactive streams is an initiative to provide a standard for asynchronous stream processing with back-pressure. It provides a minimal set of interfaces and protocols that describe the operations and entities to achieve the asynchronous streams of data with nonblocking back-pressure. It does not define operators manipulating the *streams*, and is mainly used as an interoperability layer. This initiative is supported by Netflix, Lightbend, and Red Hat, among others.

Reactive Systems

While *reactive programming* is a development model, *reactive systems* is an architectural style used to build distributed systems (*http://www.reactivemanifesto.org/*). It's a set of principles used to achieve *responsiveness* and build systems that respond to requests in a timely fashion even with failures or under load.

To build such a system, *reactive systems* embrace a message-driven approach. All the components interact using messages sent and received asynchronously. To decouple senders and receivers, components send messages to *virtual addresses*. They also register to the virtual *addresses* to receive messages. An *address* is a destination identifier such as an opaque string or a URL. Several receivers can be registered on the same address—the delivery semantic depends on the underlying technology. Senders do not block and wait for a response. The sender may receive a response later, but in the meantime, he can receive and send other messages. This asynchronous aspect is particularly important and impacts how your application is developed.

Using asynchronous message-passing interactions provides reactive systems with two critical properties:

- Elasticity—The ability to scale horizontally (scale out/in)
- Resilience—The ability to handle failure and recover

Elasticity comes from the decoupling provided by message interactions. Messages sent to an address can be consumed by a set of consumers using a load-balancing strategy. When a reactive system faces a spike in load, it can spawn new instances of consumers and dispose of them afterward.

This resilience characteristic is provided by the ability to handle failure without blocking as well as the ability to replicate components. First, message interactions allow components to deal with failure locally. Thanks to the asynchronous aspect, components do not actively wait for responses, so a failure happening in one component would not impact other components. Replication is also a key ability to handle resilience. When one node-processing message fails, the message can be processed by another node registered on the same address.

Thanks to these two characteristics, the system becomes responsive. It can adapt to higher or lower loads and continue to serve requests in the face of high loads or failures. This set of principles is primordial when building microservice systems that are highly distributed, and when dealing with *services* beyond the control of the caller. It is necessary to run several instances of your services to balance the load and handle failures without breaking the availability. We will see in the next chapters how Vert.x addresses these topics.

Reactive Microservices

When building a microservice (and thus distributed) system, each service can change, evolve, fail, exhibit slowness, or be withdrawn at any time. Such issues must not impact the behavior of the whole system. Your system must embrace changes and be able to handle failures. You may run in a degraded mode, but your system should still be able to handle the requests.

To ensure such behavior, reactive microservice *systems* are comprised of *reactive microservices*. These microservices have four characteristics:

- Autonomy
- Asynchronisity
- Resilience
- Elasticity

Reactive microservices are autonomous. They can adapt to the availability or unavailability of the services surrounding them. However, autonomy comes paired with isolation. Reactive microservices can handle failure locally, act independently, and cooperate with others as needed. A reactive microservice uses asynchronous message-passing to interact with its peers. It also receives messages and has the ability to produce responses to these messages.

Thanks to the asynchronous message-passing, reactive microservices can face failures and adapt their behavior accordingly. Failures should not be propagated but handled close to the root cause. When a microservice blows up, the consumer microservice must handle the failure and

not propagate it. This isolation principle is a key characteristic to prevent failures from bubbling up and breaking the whole system. Resilience is not only about managing failure, it's also about self-healing. A reactive microservice should implement recovery or compensation strategies when failures occur.

Finally, a reactive microservice must be elastic, so the system can adapt to the number of instances to manage the load. This implies a set of constraints such as avoiding in-memory state, sharing state between instances if required, or being able to route messages to the same instances for stateful services.

What About Vert.x ?

Vert.x is a toolkit for building reactive and distributed systems using an asynchronous nonblocking development model. Because it's a toolkit and not a framework, you use Vert.x as any other library. It does not constrain how you build or structure your system; you use it as you want. Vert.x is very flexible; you can use it as a standalone application or embedded in a larger one.

From a developer standpoint, Vert.x a set of JAR files. Each Vert.x module is a JAR file that you add to your `$CLASSPATH`. From HTTP servers and clients, to messaging, to lower-level protocols such as TCP or UDP, Vert.x provides a large set of modules to build your application the way you want. You can pick any of these modules in addition to Vert.x Core (the main Vert.x component) to build your system. Figure 2-2 shows an excerpt view of the Vert.x ecosystem.

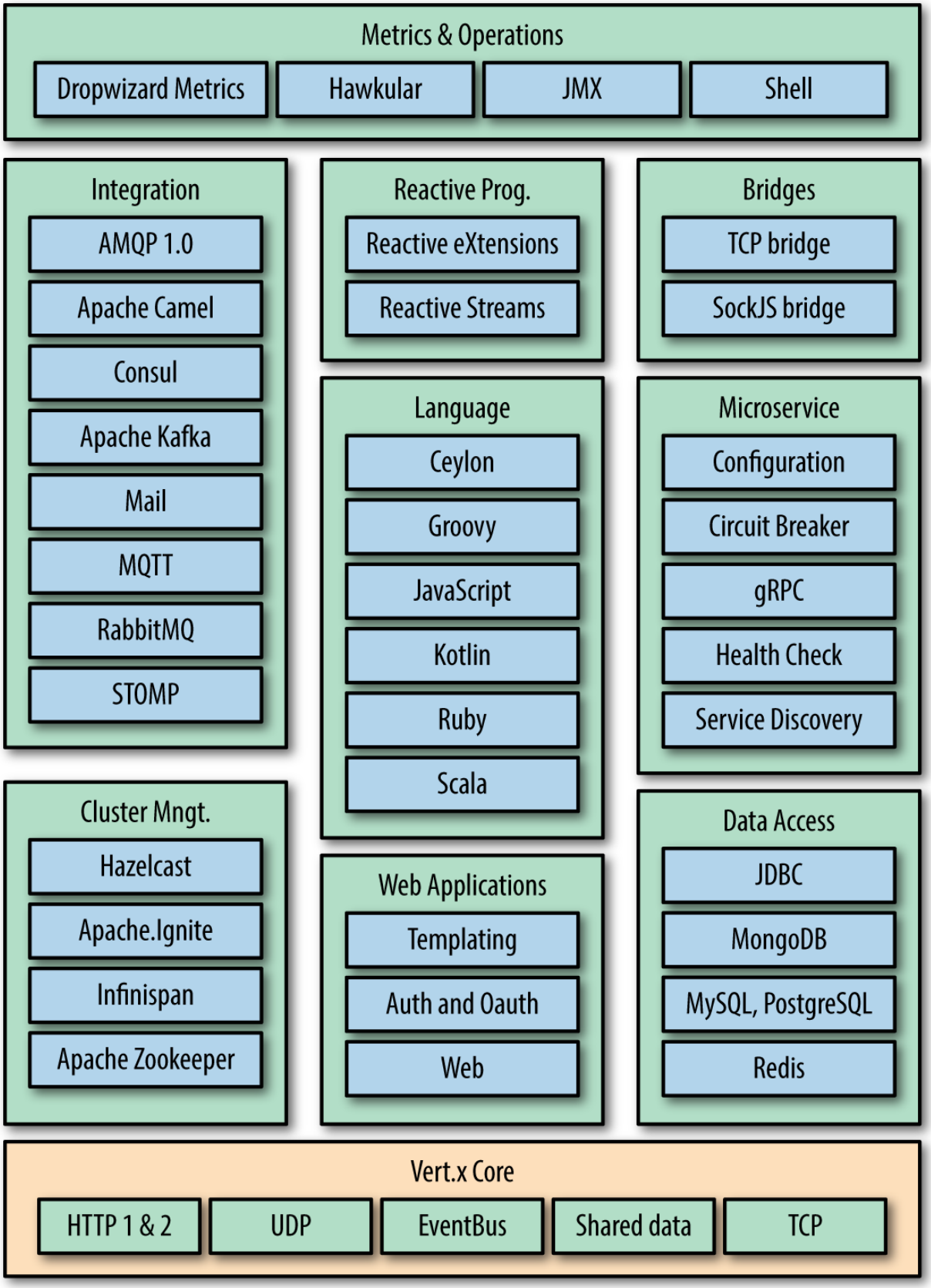


Figure 2-2. An incomplete overview of the Vert.x ecosystem

Vert.x also provides a great stack to help build microservice systems. Vert.x pushed the microservice approach before it became popular. It has been designed and built to provide an intuitive and powerful way to build microservice systems. And that's not all. With Vert.x you can build *reactive* microservices. When building a microservice with Vert.x, it infuses one of its core characteristics to the microservice: it becomes asynchronous all the way.

Asynchronous Development Model

All applications built with Vert.x are asynchronous. Vert.x applications are event-driven and nonblocking. Your application is notified when something interesting happens. Let's look at a concrete example. Vert.x provides an easy way to create an HTTP server. This HTTP server is notified every time an HTTP request is received:

```
vertx.createHttpServer()
    .requestHandler(request -> {
        // This handler will be called every time an HTTP
        // request is received at the server
        request.response().end("hello Vert.x");
    })
    .listen(8080);
```

In this example, we set a `requestHandler` to receive the HTTP requests (*event*) and send `hello Vert.x` back (*reaction*). A `Handler` is a function called when an event occurs. In our example, the code of the handler is executed with each incoming request. Notice that a `Handler` does not *return* a result. However, a `Handler` can provide a result. How this result is provided depends on the type of interaction. In the last snippet, it just writes the result into the HTTP `response`. The `Handler` is chained to a listen request on the socket. Invoking this HTTP endpoint produces a simple HTTP response:

```
HTTP/1.1 200 OK
Content-Length: 12

hello Vert.x
```

With very few exceptions, none of the APIs in Vert.x block the calling thread. If a result can be provided immediately, it will be returned; otherwise, a `Handler` is used to receive events at a later time. The `Handler` is notified when an event is ready to be processed or when the result of an asynchronous operation has been computed.

In traditional imperative programming, you would write something like:

```
int res = compute(1, 2);
```

In this code, you wait for the result of the method. When switching to an asynchronous nonblocking development model, you pass a `Handler` invoked when the result is ready: [1]

```
compute(1, 2, res -> {
```

```
        // Called with the result
    });
```

In the last snippet, `compute` does not return a result anymore, so you don't wait until this result is computed and returned. You pass a `Handler` that is called when the result is ready.

Thanks to this nonblocking development model, you can handle a highly concurrent workload using a small number of threads. In most cases, Vert.x calls your handlers using a thread called an *event loop*. This event loop is depicted in Figure 2-3. It consumes a queue of events and dispatches each event to the interested `Handlers`.

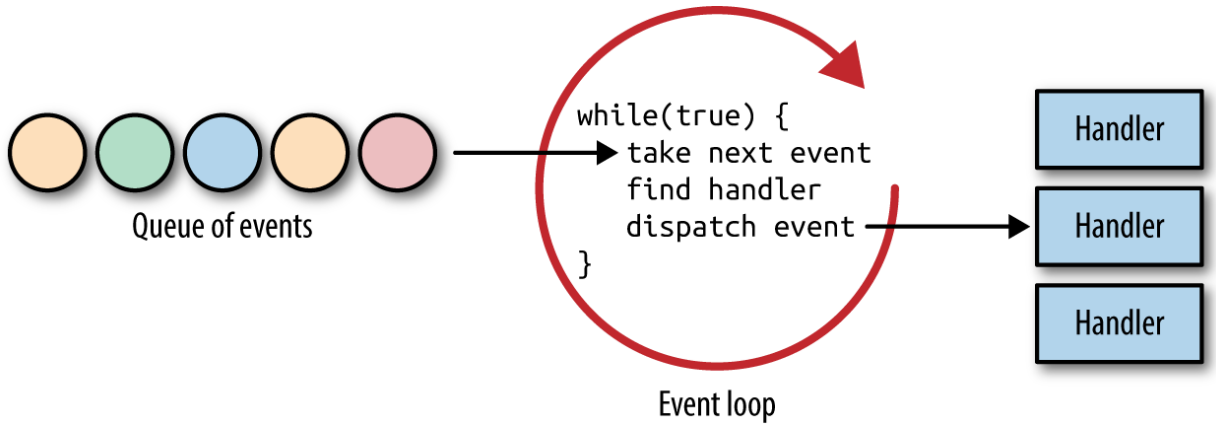


Figure 2-3. The event loop principle

The threading model proposed by the event loop has a huge benefit: it simplifies concurrency. As there is only one thread, you are always called by the same thread and never concurrently. However, it also has a very important rule that you **must** obey:

> *Don't block the event loop.*
>
> —Vert.x golden rule

Because nothing blocks, an event loop can deliver a huge number of events in a short amount of time. This is called the *reactor pattern* (*https://en.wikipedia.org/wiki/Reactor_pattern*).

Let's imagine, for a moment, that you break the rule. In the previous code snippet, the request handler is always called from the same event loop. So, if the HTTP request processing blocks instead of replying to the user immediately, the other requests would not be handled in a timely fashion and would be queued, waiting for the thread to be released. You would lose the scalability and efficiency benefit of Vert.x. So what can be *blocking*? The first obvious example is JDBC database accesses. They are blocking by nature. Long computations are also blocking. For example, a code calculating Pi to the 200,000th decimal point is definitely blocking. Don't worry—Vert.x also provides constructs to deal with blocking code.

In a standard reactor implementation, there is a single event loop thread that runs around in a loop delivering all events to all handlers as they arrive. The issue with a single thread is simple: it can only run on a single CPU core at one time. Vert.x works differently here. Instead of a single event loop, each Vert.x instance maintains several event loops, which is called a multireactor pattern, as shown in Figure 2-4.

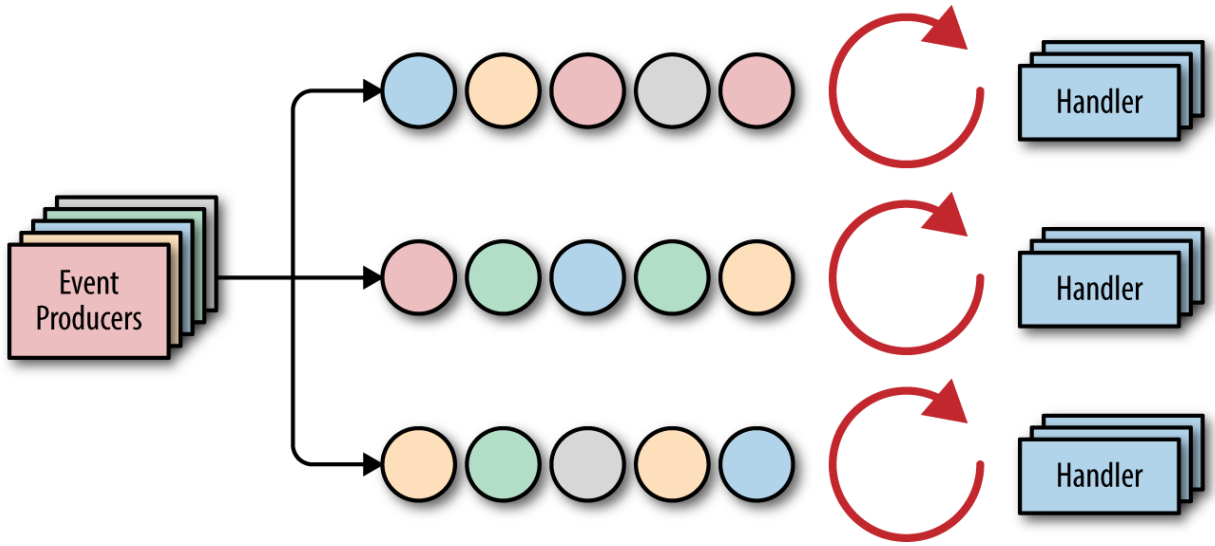


Figure 2-4. The multireactor principle

The events are dispatched by the different event loops. However, once a `Handler` is executed by an event loop, it will always be invoked by this event loop, enforcing the concurrency benefits of the reactor pattern. If, like in Figure 2-4, you have several event loops, it can balance the load on different CPU cores. How does that work with our HTTP example? Vert.x registers the socket listener once and dispatches the requests to the different event loops.

Verticles—the Building Blocks

Vert.x gives you a lot of freedom in how you can shape your application and code. But it also provides bricks to easily start writing Vert.x applications and comes with a simple, scalable, actor-like deployment and concurrency model out of the box. *Verticles* are chunks of code that get deployed and run by Vert.x. An application, such as a microservice, would typically be comprised of many verticle instances running in the same Vert.x instance at the same time. A verticle typically creates servers or clients, registers a set of `Handlers`, and encapsulates a part of the business logic of the system.

Regular verticles are executed on the Vert.x event loop and can never block. Vert.x ensures that each verticle is always executed by the same thread and never concurrently, hence avoiding synchronization constructs. In Java, a verticle is a class extending the `AbstractVerticle` class:

```
import io.vertx.core.AbstractVerticle;

public class MyVerticle extends AbstractVerticle {
    @Override
    public void start() throws Exception {
        // Executed when the verticle is deployed
    }

    @Override
    public void stop() throws Exception {
        // Executed when the verticle is un-deployed
    }
}
```

WORKER VERTICLE

Unlike *regular* verticles, *worker* verticles are not executed on the event loop, which means they can execute blocking code. However, this limits your scalability.

Verticles have access to the `vertx` member (provided by the `AbstractVerticle` class) to create servers and clients and to interact with the other verticles. Verticles can also deploy other verticles, configure them, and set the number of instances to create. The instances are associated with the different event loops (implementing the multireactor pattern), and Vert.x balances the load among these instances.

From Callbacks to Observables

As seen in the previous sections, the Vert.x development model uses callbacks. When orchestrating several asynchronous actions, this callback-based development model tends to produce complex code. For example, let's look at how we would retrieve data from a database. First, we need a connection to the database, then we send a query to the database, process the results, and release the connection. All these operations are asynchronous. Using callbacks, you would write the following code using the Vert.x JDBC client:

```
client.getConnection(conn -> {
    if (conn.failed()) {/* failure handling */}
    else {
        SQLConnection connection = conn.result();
        connection.query("SELECT * from PRODUCTS", rs -> {
            if (rs.failed()) {/* failure handling */}
            else {
                List<JsonArray> lines =
                    rs.result().getResults();
                for (JsonArray l : lines) {
```

```
                        System.out.println(new Product(l));
                    }
                    connection.close(done -> {
                        if (done.failed()) {/* failure handling */}
                    });
                }
            });
        }
    });
```

While still manageable, the example shows that callbacks can quickly lead to unreadable code. You can also use Vert.x `Futures` to handle asynchronous actions. Unlike Java `Futures`, Vert.x `Futures` are nonblocking. `Futures` provide higher-level composition operators to build sequences of actions or to execute actions in parallel. Typically, as demonstrated in the next snippet, we *compose* futures to build the sequence of asynchronous actions:

```
Future<SQLConnection> future = getConnection();
future
    .compose(conn -> {
        connection.set(conn);
        // Return a future of ResultSet
        return selectProduct(conn);
    })
    // Return a collection of products by mapping
    // each row to a Product
    .map(result -> toProducts(result.getResults()))
    .setHandler(ar -> {
        if (ar.failed()) { /* failure handling */ }
        else {
            ar.result().forEach(System.out::println);
        }
        connection.get().close(done -> {
            if (done.failed()) { /* failure handling */ }
        });
    });
```

However, while `Futures` make the code a bit more declarative, we are retrieving all the rows in one batch and processing them. This result can be huge and take a lot of time to be retrieved. At the same time, you don’t need the whole result to start processing it. We can process each row one by one as soon as you have them. Fortunately, Vert.x provides an answer to this development model challenge and offers you a way to implement reactive microservices using a reactive programming development model. Vert.x provides RxJava APIs to:

- Combine and coordinate asynchronous tasks
- React to incoming messages as a stream of input

Let's rewrite the previous code using the RxJava APIs:

```
// We retrieve a connection and cache it,
// so we can retrieve the value later.
Single<SQLConnection> connection = client
  .rxGetConnection();
connection
  .flatMapObservable(conn ->
    conn
      // Execute the query
      .rxQueryStream("SELECT * from PRODUCTS")
      // Publish the rows one by one in a new Observable
      .flatMapObservable(SQLRowStream::toObservable)
      // Don't forget to close the connection
      .doAfterTerminate(conn::close)
  )
  // Map every row to a Product
  .map(Product::new)
  // Display the result one by one
  .subscribe(System.out::println);
```

In addition to improving readability, reactive programming allows you to *subscribe* to a stream of results and process items as soon as they are available. With Vert.x you can choose the development model you prefer. In this report, we will use both callbacks and RxJava.

Let's Start Coding!

It's time for you to get your hands dirty. We are going to use Apache Maven and the Vert.x Maven plug-in to develop our first Vert.x application. However, you can use whichever tool you want (Gradle, Apache Maven with another packaging plug-in, or Apache Ant). You will find different examples in the code repository (in the `packaging-examples` directory). The code shown in this section is located in the `hello-vertx` directory.

Project Creation

Create a directory called `my-first-vertx-app` and move into this directory:

```
mkdir my-first-vertx-app
cd my-first-vertx-app
```

Then, issue the following command:

```
mvn io.fabric8:vertx-maven-plugin:1.0.5:setup \
  -DprojectGroupId=io.vertx.sample \
```

```
    -DprojectArtifactId=my-first-vertx-app \
    -Dverticle=io.vertx.sample.MyFirstVerticle
```

This command generates the Maven project structure, configures the `vertx-maven-plugin`, and creates a verticle class (`io.vertx.sample.MyFirstVerticle`), which does nothing.

Write Your First Verticle

It's now time to write the code for your first verticle. Modify the `src/main/java/io/vertx/sample/MyFirstVerticle.java` file with the following content:

```
package io.vertx.sample;

import io.vertx.core.AbstractVerticle;

/**
 * A verticle extends the AbstractVerticle class.
 */
public class MyFirstVerticle extends AbstractVerticle {

  @Override
  public void start() throws Exception {
    // We create a HTTP server object
    vertx.createHttpServer()
      // The requestHandler is called for each incoming
      // HTTP request, we print the name of the thread
      .requestHandler(req -> {
        req.response().end("Hello from "
          + Thread.currentThread().getName());
      })
      .listen(8080); // start the server on port 8080
  }
}
```

To run this application, launch:

```
mvn compile vertx:run
```

If everything went fine, you should be able to see your application by opening *http://localhost:8080* in a browser. The `vertx:run` goal launches the Vert.x application and also watches code alterations. So, if you edit the source code, the application will be automatically recompiled and restarted.

Let's now look at the application output:

```
Hello from vert.x-eventloop-thread-0
```

The request has been processed by the *event loop 0*. You can try to emit more requests. The requests will always be processed by the same event loop, enforcing the concurrency model of Vert.x. Hit Ctrl+C to stop the execution.

Using RxJava

At this point, let's take a look at the RxJava support provided by Vert.x to better understand how it works. In your `pom.xml` file, add the following dependency:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-rx-java</artifactId>
</dependency>
```

Next, change the `<vertx.verticle>` property to be `io.vertx.sample.MyFirstRXVerticle`. This property tells the Vert.x Maven plug-in which verticle is the entry point of the application. Create the new verticle class (`io.vertx.sample.MyFirstRXVerticle`) with the following content:

```
package io.vertx.sample;

// We use the .rxjava. package containing the RX-ified APIs
import io.vertx.rxjava.core.AbstractVerticle;
import io.vertx.rxjava.core.http.HttpServer;

public class MyFirstRXVerticle extends AbstractVerticle {

  @Override
  public void start() {
    HttpServer server = vertx.createHttpServer();
    // We get the stream of request as Observable
    server.requestStream().toObservable()
      .subscribe(req ->
        // for each HTTP request, this method is called
        req.response().end("Hello from "
          + Thread.currentThread().getName())
      );
    // We start the server using rxListen returning a
    // Single of HTTP server. We need to subscribe to
    // trigger the operation
    server
```

```
        .rxListen(8080)
        .subscribe();
    }
}
```

The RxJava variants of the Vert.x APIs are provided in packages with `rxjava` in their name. RxJava methods are prefixed with `rx`, such as `rxListen`. In addition, the APIs are enhanced with methods providing `Observable` objects on which you can subscribe to receive the conveyed data.

Packaging Your Application as a Fat Jar

The Vert.x Maven plug-in packages the application in a *fat jar*. Once packaged, you can easily launch the application using `java -jar <name>.jar`:

```
mvn clean package
cd target
java -jar my-first-vertx-app-1.0-SNAPSHOT.jar
```

The application is up again, listening for HTTP traffic on the port specified. Hit Ctrl+C to stop it.

As an unopinionated toolkit, Vert.x does not promote one packaging model over another—you are free to use the packaging model you prefer. For instance, you could use *fat jars*, a filesystem approach with libraries in a specific directory, or embed the application in a *war* file and start Vert.x programmatically.

In this report, we will use fat jars, i.e., self-contained JAR embedding the code of the application, its resources, as well as all of its dependencies. This includes Vert.x, the Vert.x components you are using, and their dependencies. This packaging model uses a flat class loader mechanism, which makes it easier to understand application startup, dependency ordering, and logs. More importantly, it helps reduce the number of *moving* pieces that need to be installed in production. You don't deploy an application to an existing app server. Once it is packaged in its fat jar, the application is ready to run with a simple `java -jar <name.jar>`. The Vert.x Maven plug-in builds a fat jar for you, but you can use another Maven plug-in such as the `maven-shader-plugin` too.

Logging, Monitoring, and Other Production Elements

Having fat jar is a great packaging model for microservices and other types of applications as they simplify the deployment and launch. But what about the features generally offered by app

servers that make your application *production ready*? Typically, we expect to be able to write and collect logs, monitor the application, push external configuration, add health checks, and so on.

Don't worry—Vert.x provides all these features. And because Vert.x is neutral, it provides several alternatives, letting you choose or implement your own. For example, for logging, Vert.x does not push a specific logging framework but instead allows you to use any logging framework you want, such as Apache Log4J 1 or 2, SLF4J, or even JUL (the JDK logging API). If you are interested in the messages logged by Vert.x itself, the internal Vert.x logging can be configured to use any of these logging frameworks. Monitoring Vert.x applications is generally done using JMX. The Vert.x Dropwizard Metric module provides Vert.x metrics to JMX. You can also choose to publish these metrics to a monitoring server such as Prometheus (*https://prometheus.io/*) or CloudForms (*https://www.redhat.com/en/technologies/management/cloudforms*).

Summary

In this chapter we learned about reactive microservices and Vert.x. You also created your first Vert.x application. This chapter is by no means a comprehensive guide and just provides a quick introduction to the main concepts. If you want to go further on these topics, check out the following resources:

- Reactive programming vs. Reactive systems
- The Reactive Manifesto
- RxJava website
- *Reactive Programming with RxJava*
- The Vert.x website

---

[1] This code uses the `lambda` expressions introduced in Java 8. More details about this notation can be found at *http://bit.ly/2nsyJJv*.

Chapter 3. Building Reactive Microservices

In this chapter, we will build our first microservices with Vert.x. As most microservice systems use HTTP interactions, we are going to start with HTTP microservices. But because systems consist of multiple communicating microservices, we will build another microservice that consumes the first one. Then, we will demonstrate why such a design does not completely embrace reactive microservices. Finally, we will implement message-based microservices to see how messaging improves the *reactiveness*.

First Microservices

In this chapter we are going to implement the same set of microservices twice. The first microservice exposes a *hello* service that we will call *hello microservice*. Another consumes this service twice (concurrently). The consumer will be called *hello consumer microservice*. This small system illustrates not only how a service is served, but also how it is consumed. On the left side of Figure 3-1, the microservices are using HTTP interactions. The hello consumer microservice uses an *HTTP client* to invoke the hello microservice. On the right side, the hello consumer microservice uses messages to interact with the hello microservice. This difference impacts the *reactiveness* of the system.

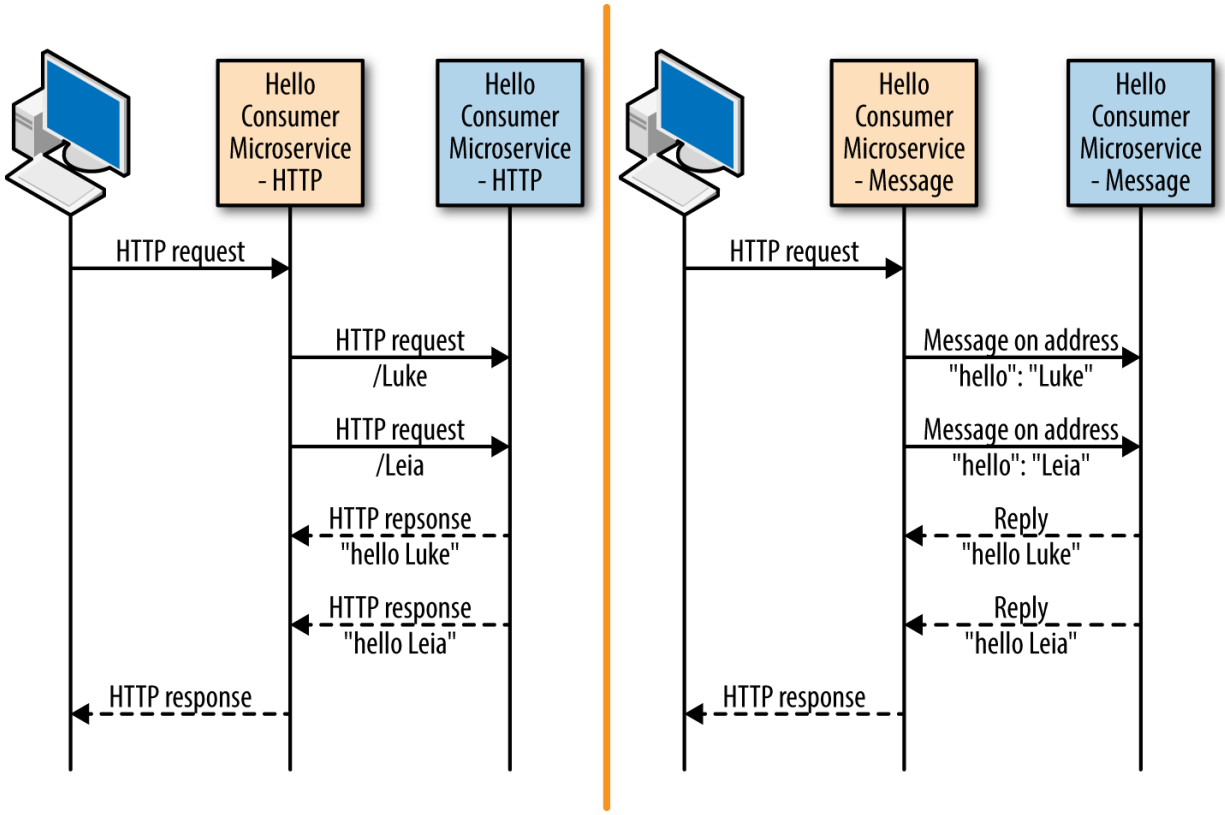


Figure 3-1. The microservices implemented in this chapter using HTTP and message-based interactions

In the previous chapter, we saw two different ways to use Vert.x APIs: callbacks and RxJava. To illustrate the differences and help you find your preferred approach, the *hello* microservices are implemented using the callback-based development model, while the consumers are implemented using RxJava.

Implementing HTTP Microservices

Microservices often expose their API via HTTP and are consumed using HTTP requests. Let's see how these HTTP interactions can be implemented with Vert.x. The code developed in this section is available in the `microservices/hello-microservice-http` directory of the code repository.

Getting Started

Create a directory called `hello-microservice-http` and then generate the project structure:

```
mkdir hello-microservice-http
cd hello-microservice-http

mvn io.fabric8:vertx-maven-plugin:1.0.5:setup \
  -DprojectGroupId=io.vertx.microservice \
  -DprojectArtifactId=hello-microservice-http \
```

```
    -Dverticle=io.vertx.book.http.HelloMicroservice \
    -Ddependencies=web
```

This command generates the Maven project and configures the Vert.x Maven plug-in. In addition, it adds the `vertx-web` dependency. Vert.x Web is a module that provides everything you need to build modern web applications on top of Vert.x.

The Verticle

Open `src/main/java/io/vertx/book/http/HelloMicroservice.java`. The generated code of the verticle does nothing very interesting, but it's a starting point:

```
package io.vertx.book.http;

import io.vertx.core.AbstractVerticle;

public class HelloMicroservice extends AbstractVerticle {

    @Override
    public void start() {

    }
}
```

Now, launch the following Maven command:

```
mvn compile vertx:run
```

You can now edit the verticle. Every time you save the file, the application will be recompiled and restarted automatically.

HTTP Microservice

It's time to make our `MyVerticle` class do something. Let's start with an HTTP server. As seen in the previous chapter, to create an HTTP server with Vert.x you just use:

```
@Override
public void start() {
    vertx.createHttpServer()
        .requestHandler(req -> req.response()
          .end("hello"))
        .listen(8080);
}
```

Once added and saved, you should be able to see `hello` at *http://localhost:8080* in a browser. This code creates an HTTP server on port 8080 and registers a `requestHandler` that is invoked on each incoming HTTP request. For now, we just write `hello` in the response.

Using Routes and Parameters

Many services are invoked through web URLs, so checking the path is crucial to knowing what the request is asking for. However, doing path checking in the `requestHandler` to implement different actions can get complicated. Fortunately, Vert.x Web provides a `Router` on which we can register *Routes*. Routes are the mechanism by which Vert.x Web checks the path and invokes the associated action. Let's rewrite the `start` method, with two routes:

```
@Override
public void start() {
    Router router = Router.router(vertx);
    router.get("/").handler(rc -> rc.response().end("hello"));
    router.get("/:name").handler(rc -> rc.response()
        .end("hello " + rc.pathParam("name")));

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

Once we have created the `Router` object, we register two routes. The first one handles requests on `/` and just writes `hello`. The second route has a path parameter (`:name`). The handler appends the passed value to the greeting message. Finally, we change the `requestHandler` of the HTTP server to use the `accept` method of the `router`.

If you didn't stop the `vertx:run` execution, you should be able to open a browser to:

- *http://localhost:8080*—You should see `hello`
- *http://localhost:8080/vert.x*—You should see `hello vert.x`

Producing JSON

JSON is often used in microservices. Let's modify the previous class to produce JSON payloads:

```
@Override
public void start() {
    Router router = Router.router(vertx);
```

```
        router.get("/").handler(this::hello);
        router.get("/:name").handler(this::hello);
        vertx.createHttpServer()
            .requestHandler(router::accept)
            .listen(8080);
    }

    private void hello(RoutingContext rc) {
        String message = "hello";
        if (rc.pathParam("name") != null) {
            message += " " + rc.pathParam("name");
        }
        JsonObject json = new JsonObject().put("message", message);
        rc.response()
            .putHeader(HttpHeaders.CONTENT_TYPE, "application/json")
            .end(json.encode());
    }
```

Vert.x provides a `JsonObject` class to create and manipulate JSON structures. With this code in place, you should be able to open a browser to:

- *http://localhost:8080*—You should see `{"message": "hello"}`
- *http://localhost:8080/vert.x*—You should see `{"message": "hello vert.x"}`

Packaging and Running

Stop the `vertx:run` execution using Ctrl+C and execute the following command from the same directory:

```
mvn package
```

This produces a *fat jar* in the `target` directory: `hello-microservice-http-1.0-SNAPSHOT.jar`. While fat jars tend to be *fat*, here the JAR has a reasonable size (~6.3 MB) and contains everything to run the application:

```
java -jar target/hello-microservice-http-1.0-SNAPSHOT.jar
```

You can check to make sure it is running by opening: *http://localhost:8080*. Keep the process running as the next microservice will invoke it.

Consuming HTTP Microservices

One microservice does not form an application; you need a system of microservices. Now that

we have our first microservice running, let's write a second microservice to consume it. This second microservice also provides an HTTP facade to invoke it, and on each invocation calls the microservice we just implemented. The code shown in this section is available in the `microservices/hello-consumer-microservice-http` directory of the code repository.

Project Creation

As usual, let's create a new project:

```
mkdir hello-consumer-microservice-http
cd hello-consumer-microservice-http

mvn io.fabric8:vertx-maven-plugin:1.0.5:setup \
  -DprojectGroupId=io.vertx.microservice \
  -DprojectArtifactId=hello-consumer-microservice-http \
  -Dverticle=io.vertx.book.http.HelloConsumerMicroservice \
  -Ddependencies=web,web-client,rx
```

The last command adds another dependency: the Vert.x web client, an asynchronous HTTP client. We will use this client to call the first microservice. The command has also added the Vert.x RxJava binding we are going to use later.

Now edit the `src/main/java/io/vertx/book/http/HelloConsumerMicroservice.java` file and update it to contain:

```
package io.vertx.book.http;

import io.vertx.core.AbstractVerticle;
import io.vertx.core.json.JsonObject;
import io.vertx.ext.web.*;
import io.vertx.ext.web.client.*;
import io.vertx.ext.web.codec.BodyCodec;

public class HelloConsumerMicroservice extends AbstractVerticle {

    private WebClient client;

    @Override
    public void start() {
        client = WebClient.create(vertx);

        Router router = Router.router(vertx);
        router.get("/").handler(this::invokeMyFirstMicroservice);
```

```
            vertx.createHttpServer()
                .requestHandler(router::accept)
                .listen(8081);
        }

        private void invokeMyFirstMicroservice(RoutingContext rc) {
            HttpRequest<JsonObject> request = client
                .get(8080, "localhost","/vert.x")
                .as(BodyCodec.jsonObject());

            request.send(ar -> {
                if (ar.failed()) {
                    rc.fail(ar.cause());
                } else {
                    rc.response().end(ar.result().body().encode());
                }
            });
        }

    }
```

In the `start` method, we create a `WebClient` and a `Router`. On the created router, we register a route on "/" and start the HTTP server, passing the router `accept` method as `requestHandler`. The handler of the route is a method reference (`hello`). This method uses the web client to invoke the first microservice with a specific path (`/vert.x`) and write the result to the HTTP response.

Once the HTTP request is created, we call `send` to emit the request. The handler we passed in is invoked when either the response arrives or an error occurs. The `if-else` block checks to see whether the invocation has succeeded. Don't forget that it's a remote interaction and has many reasons to fail. For instance, the first microservice may not be running. When it succeeds, we write the received payload to the response; otherwise, we reply with a `500` response.

Calling the Service More Than Once

Now let's change the current behavior to call the *hello* microservice twice with two different (path) parameters:

```
HttpRequest<JsonObject> request1 = client
    .get(8080, "localhost", "/Luke")
    .as(BodyCodec.jsonObject());
HttpRequest<JsonObject> request2 = client
    .get(8080, "localhost", "/Leia")
    .as(BodyCodec.jsonObject());
```

These two requests are independent and can be executed concurrently. But here we want to

write a response assembling both results. The code required to invoke the service twice and assemble the two results can become convoluted. We need to check to see whether or not the other request has been completed when we receive one of the responses. While this code would still be manageable for two requests, it becomes overly complex when we need to handle more. Fortunately, as noted in the previous chapter, we can use reactive programming and RxJava to simplify this code.

We instruct the `vertx-maven-plugin` to import the Vert.x RxJava API. In the `HelloConsumerMicroservice`, we replace the import statements with:

```
import io.vertx.core.json.JsonObject;
import io.vertx.rxjava.core.AbstractVerticle;
import io.vertx.rxjava.ext.web.*;
import io.vertx.rxjava.ext.web.client.*;
import io.vertx.rxjava.ext.web.codec.BodyCodec;
import rx.Single;
```

With RX, the complex code we would have written to call the two requests and build a response out of them becomes much simpler:

```
private void invokeMyFirstMicroservice(RoutingContext rc) {
    HttpRequest<JsonObject> request1 = client
        .get(8080, "localhost", "/Luke")
        .as(BodyCodec.jsonObject());
    HttpRequest<JsonObject> request2 = client
        .get(8080, "localhost", "/Leia")
        .as(BodyCodec.jsonObject());
    Single<JsonObject> s1 = request1.rxSend()
      .map(HttpResponse::body);
    Single<JsonObject> s2 = request2.rxSend()
      .map(HttpResponse::body);
    Single
        .zip(s1, s2, (luke, leia) -> {
            // We have the results of both requests in Luke and Leia
            return new JsonObject()
                .put("Luke", luke.getString("message"))
                .put("Leia", leia.getString("message"));
        })
        .subscribe(
            result -> rc.response().end(result.encodePrettily()),
            error -> {
              error.printStackTrace();
              rc.response()
                .setStatusCode(500).end(error.getMessage());
            }
        );
}
```

Notice the `rxSend` method calls. The RxJava methods from Vert.x are prefixed with `rx` to be easily recognizable. The result of `rxSend` is a `Single`, i.e., an observable of one element representing the deferred result of an operation. The `single.zip` method takes as input a set of `Single`, and once all of them have received their value, calls a function with the results. `Single.zip` produces another `Single` containing the result of the function. Finally, we `subscribe`. This method takes two functions as parameters:

1. The first one is called with the result of the `zip` function (a JSON object). We write the receive JSON payload into the HTTP response.

2. The second one is called if something fails (timeout, exception, etc.). In this case, we respond with an empty JSON object.

   With this code in place, if we open `http://localhost:8081` and the *hello* microservice is still running we should see:

```
{
  "Luke" : "hello Luke",
  "Leia" : "hello Leia"
}
```

Are These Microservices Reactive Microservices?

At this point we have two microservices. They are independent and can be deployed and updated at their own pace. They also interact using a lightweight protocol (HTTP). But are they reactive microservices? No, they are not. Remember, to be called reactive a microservice must be:

- Autonomous
- Asynchronous
- Resilient
- Elastic

The main issue with the current design is the tight coupling between the two microservices. The web client is configured to target the first microservice explicitly. If the first microservice fails, we won't be able to recover by calling another one. If we are under load, creating a new instance of the *hello* microservice won't help us. Thanks to the Vert.x web client, the interactions are asynchronous. However, as we don't use a *virtual* address (destination) to

invoke the microservice, but its direct URL, it does not provide the resilience and elasticity we need.

It's important to note that using reactive programming as in the second microservice does not give you the *reactive* system's benefits. It provides an elegant development model to coordinate asynchronous actions, but it does not provide the resilience and elasticity we need.

Can we use HTTP for reactive microservices? Yes. But this requires some infrastructure to route *virtual* URLs to a set of services. We also need to implement a load-balancing strategy to provide elasticity and health-check support to improve resilience.

Don't be disappointed. In the next section we will take a big step toward reactive microservices.

The Vert.x Event Bus—A Messaging Backbone

Vert.x offers an *event bus* allowing the different components of an application to interact using *messages*. Messages are sent to *addresses* and have a set of *headers* and a *body*. An address is an opaque string representing a destination. Message consumers register themselves to addresses to receive the messages. The event bus is also clustered, meaning it can dispatch messages over the network between distributed senders and consumers. By starting a Vert.x application in *cluster* mode, nodes are connected to enable shared data structure, hard-stop failure detection, and load-balancing group communication. The event bus can dispatch messages among all the nodes in the cluster. To create such a clustered configuration, you can use Apache Ignite, Apache Zookeeper, Infinispan, or Hazelcast. In this report, we are going to use Infinispan, but we won't go into advanced configuration. For that, refer to the Infinispan documentation (*http://infinispan.org/*). While Infinispan (or the technology you choose) manages the node discovery and inventory, the event bus communication uses direct peer-to-peer TCP connections.

The event bus provides three types of delivery semantics. First, the `send` method allows a component to send a message to an address. A single consumer is going to receive the message. If more than one consumer is registered on this address, Vert.x applies a round-robin strategy to select a consumer:

```
// Consumer
vertx.eventBus().consumer("address", message -> {
    System.out.println("Received: '" + message.body() + "'");
});
// Sender
vertx.eventBus().send("address", "hello");
```

In contrast to `send`, you can use the `publish` method to deliver the message to all consumers registered on the address. Finally, the `send` method can be used with a `reply` handler. This *request/response* mechanism allows implementing message-based asynchronous interactions between two components:

```
// Consumer
vertx.eventBus().consumer("address", message -> {
    message.reply("pong");
});
// Sender
vertx.eventBus().send("address", "ping", reply -> {
    if (reply.succeeded()) {
        System.out.println("Received: " + reply.result().body());
    } else {
        // No reply or failure
        reply.cause().printStackTrace();
    }
});
```

If you are using Rx-ified APIs, you can use the `rxSend` method, which returns a `Single`. This `Single` receives a value when the reply is received. We are going to see this method in action shortly.

Message-Based Microservices

Let's reimplement the *hello* microservice, this time using an event bus instead of an HTTP server to receive the request. The microservice replied to the message to provide the response.

Project Creation

Let's create a new project. This time we are going to add the *Infinispan* dependency, an in-memory data grid that will be used to manage the cluster:

```
mkdir hello-microservice-message
cd hello-microservice-message

mvn io.fabric8:vertx-maven-plugin:1.0.5:setup \
  -DprojectGroupId=io.vertx.microservice \
  -DprojectArtifactId=hello-microservice-message \
  -Dverticle=io.vertx.book.message.HelloMicroservice \
  -Ddependencies=infinispan
```

Once generated, we may need to configure Infinispan to build the cluster. The default configuration uses *multicast* to discover the nodes. If your network supports multicast, it should

be fine. Otherwise, check the `resource/cluster` directory of the code repository.

Writing the Message-Driven Verticle

Edit the `src/main/java/io/vertx/book/message/HelloMicroservice.java` file and update the `start` method to be:

```
@Override
public void start() {
    // Receive message from the address 'hello'
    vertx.eventBus().<String>consumer("hello", message -> {
        JsonObject json = new JsonObject()
            .put("served-by", this.toString());
        // Check whether we have received a payload in the
        // incoming message
        if (message.body().isEmpty()) {
            message.reply(json.put("message", "hello"));
        } else {
            message.reply(json.put("message",
              "hello " + message.body()));
        }
    });
}
```

This code retrieves the `eventBus` from the `vertx` object and registers a `consumer` on the address `hello`. When a message is received, it replies to it. Depending on whether or not the incoming message has an empty body, we compute a different response. As in the example in the previous chapter, we send a JSON object back. You may be wondering why we added the `served-by` entry in the JSON. You'll see why very soon. Now that the verticle is written, it's time to launch it with:

```
mvn compile vertx:run \
  -Dvertx.runArgs="-cluster -Djava.net.preferIPv4Stack=true"
```

The `-cluster` tells Vert.x to start in *cluster* mode.

Now let's write a microservice consuming this service.

Initiating Message-Based Interactions

In this section, we will create another microservice to invoke the *hello* microservice by sending a message to the `hello` address and get a reply. The microservice will reimplement the same logic as in the previous chapter and invoke the service twice (once with `Luke` and once with

`Leia`).

As usual, let's create a new project:

```
mkdir hello-consumer-microservice-message
cd hello-consumer-microservice-message

mvn io.fabric8:vertx-maven-plugin:1.0.5:setup \
  -DprojectGroupId=io.vertx.microservice \
  -DprojectArtifactId=hello-consumer-microservice-message \
  -Dverticle=io.vertx.book.message.HelloConsumerMicroservice \
  -Ddependencies=infinispan,rx
```

Here we also add the Vert.x RxJava support to benefit from the RX-ified APIs provided by Vert.x. If you updated the Infinispan configuration in the previous section, you need to copy it to this new project.

Now edit the `io.vertx.book.message.HelloConsumerMicroservice`. Since we are going to use RxJava, change the import statement to match `io.vertx.rxjava.core.AbstractVerticle`. Then implement the `start` method with:

```
@Override
public void start() {
  EventBus bus = vertx.eventBus();
  Single<JsonObject> obs1 = bus
    .<JsonObject>rxSend("hello", "Luke")
    .map(Message::body);
  Single<JsonObject> obs2 = bus
    .<JsonObject>rxSend("hello", "Leia")
    .map(Message::body);

  Single
    .zip(obs1, obs2, (luke, leia) ->
      new JsonObject()
        .put("Luke", luke.getString("message"))
        .put("Leia", leia.getString("message"))
    )
    .subscribe(
      x -> System.out.println(x.encode()),
      Throwable::printStackTrace);
}
```

This code is very similar to the code from the previous chapter. Instead of using a `WebClient` to invoke an HTTP endpoint, we will use the event bus to send a message to the `hello` address and extract the body of the reply. We use the `zip` operation to retrieve the two responses and

build the final result. In the `subscribe` method, we print the final result to the console or print the stack trace.

Let's combine this with an HTTP server. When an HTTP request is received, we invoke the `hello` service twice and return the built result as a response:

```
@Override
public void start() {
  vertx.createHttpServer()
    .requestHandler(
      req -> {
        EventBus bus = vertx.eventBus();
        Single<JsonObject> obs1 = bus
          .<JsonObject>rxSend("hello", "Luke")
          .map(Message::body);
        Single<JsonObject> obs2 = bus
          .<JsonObject>rxSend("hello", "Leia")
          .map(Message::body);

        Single
          .zip(obs1, obs2, (luke, leia) ->
            new JsonObject()
              .put("Luke", luke.getString("message")
                + " from "
                + luke.getString("served-by"))
              .put("Leia", leia.getString("message")
                + " from "
                + leia.getString("served-by"))
          )
          .subscribe(
            x -> req.response().end(x.encodePrettily()),
            t -> {
              t.printStackTrace();
              req.response().setStatusCode(500)
                .end(t.getMessage());
            }
          );
      })
    .listen(8082);
}
```

The last code just wraps the event bus interactions into a `requestHandler` and deals with the HTTP response. In case of failure, we return a JSON object containing an error message.

If you run this code with `mvn compile vertx:run -Dvertx.runArgs="-cluster -Djava.net.preferIPv4Stack=true"` and open your browser to `http://localhost:8082`, you should see something like:

```
{
  "Luke" : "hello Luke from ...HelloMicroservice@39721ab",
  "Leia" : "hello Leia from ...HelloMicroservice@39721ab"
}
```

Are We Reactive Now?

The code is very close to the HTTP-based microservice we wrote previously. The only difference is we used an event bus instead of HTTP. Does this change our *reactiveness*? It does! Let's see why.

Elasticity

Elasticity is one of the characteristics not enforced by the HTTP version of the microservice. Because the microservice was targeting a specific instance of the microservice (using a hard-coded URL), it didn't provide the elasticity we need. But now that we are using messages sent to an address, this changes the game. Let's see how this microservice system behaves.

Remember the output of the previous execution. The returned JSON objects display the verticle having computed the hello message. The output always displays the same verticle. The message was indicating the same instance. We expected this because we had a single instance running. Now let's see what happens with two.

Stop the `vertx:run` execution of the `Hello` microservice and run:

```
mvn clean package
```

Then, open two different terminals in the `hello-microservice-message` directory and issue the following command (in each terminal):

```
java -jar target/hello-microservice-message-1.0-SNAPSHOT.jar \
    --cluster -Djava.net.preferIPv4Stack=true
```

This launches two instances of the `Hello` microservice. Go back to your browser and refresh the page and you should see something like:

```
{
  "Luke" : "hello Luke from ...HelloMicroservice@16d0d069",
  "Leia" : "hello Leia from ...HelloMicroservice@411fc4f"
}
```

The two instances of `Hello` are used. The Vert.x cluster connects the different nodes, and the event bus is clustered. Thanks to the event bus round-robin, the Vert.x event bus dispatches messages to the available instances and thus balances the load among the different nodes listening to the same address.

So, by using the event bus, we have the elasticity characteristic we need.

Resilience

What about resilience? In the current code, if the *hello* microservice failed, we would get a failure and execute this code:

```
t -> {
  t.printStackTrace();
  req.response().setStatusCode(500).end(t.getMessage());
}
```

Even though the user gets an error message, we don't crash, we don't limit our scalability, and we can still handle requests. However, to improve the user experience, we should always reply in a timely fashion to the user, even if we don't receive the responses from the service. To implement this logic, we can enhance the code with a timeout.

To illustrate this, let's modify the `Hello` microservice to inject failures and misbehaviors. This code is located in the `microservices/hello-microservice-faulty` directory of the code repository.

This new `start` method randomly selects one of three strategies: (1) reply with an explicit failure, (2) *forget* to reply (leading to a timeout on the consumer side), or (3) send the correct result.

```
@Override
public void start() {
    vertx.eventBus().<String>consumer("hello", message -> {
        double chaos = Math.random();
        JsonObject json = new JsonObject()
            .put("served-by", this.toString());

        if (chaos < 0.6) {
            // Normal behavior
            if (message.body().isEmpty()) {
                message.reply(json.put("message", "hello"));
            } else {
                message.reply(json.put("message", "hello "
                  + message.body()));
```

```
            }
        } else if (chaos < 0.9) {
            System.out.println("Returning a failure");
            // Reply with a failure
            message.fail(500,
              "message processing failure");
        } else {
            System.out.println("Not replying");
            // Just do not reply, leading to a timeout on the
            // consumer side.
        }
    });
}
```

Repackage and restart the two instances of the `Hello` microservice.

With this fault injection in place, we need to improve the fault-tolerance of our consumer. Indeed, the consumer may get a timeout or receive an explicit failure. In the *hello consumer microservice*, change how we invoke the *hello* service to:

```
EventBus bus = vertx.eventBus();
Single<JsonObject> obs1 = bus
  .<JsonObject>rxSend("hello", "Luke")
  .subscribeOn(RxHelper.scheduler(vertx))
  .timeout(3, TimeUnit.SECONDS)
  .retry()
  .map(Message::body);
Single<JsonObject> obs2 = bus.
  <JsonObject>rxSend("hello", "Leia")
  .subscribeOn(RxHelper.scheduler(vertx))
  .timeout(3, TimeUnit.SECONDS)
  .retry()
  .map(Message::body);
```

This code is located in the `microservices/hello-consumer-microservice-timeout` directory of the code repository. The `timeout` method emits a failure if we don't receive a response in the given time. The `retry` method reattempts to retrieve the value if it gets a failure in the form of a timeout or an explicit failure. The `subscribeOn` method indicates on which thread the invocations need to be done. We use the Vert.x event loop to call our callbacks. Without this, the methods would be executed by a thread from the default RxJava thread pool, breaking the Vert.x threading model. The `RXHelper` class is provided by Vert.x. Blindly retrying service invocations is not a very clever fault tolerance strategy. It can even be harmful. The next chapter details different approaches.

Now you can reload the page. You will always get a result, even if there are failures or timeouts.

Remember that the thread is not blocked while calling the service, so you can always accept new requests and respond to them in a timely fashion. However, this timeout retry often causes more harm than good, as we will see in the next chapter.

Summary

In this section, we learned how to develop an HTTP microservice with Vert.x and also how to consume it. As we learned, hard-coding the URL of the consumed service in the code is not a brilliant idea as it breaks one of the *reactive* characteristics. In the second part, we replaced the HTTP interactions using messaging, which showed how messaging and the Vert.x event bus help build reactive microservices.

So, are we there yet? Yes and no. Yes, we know how to build reactive microservices, but there are a couple of shortcomings we need to look at. First, what if you only have HTTP services? How do you avoid hard-coded locations? What about resilience? We have seen timeouts and retries in this chapter, but what about circuit breakers, failovers, and bulkheads? Let's continue the journey.

If you want to go further on these topics:

- Vert.x Web documentation
- Vert.x Web Client documentation
- Vert.x reactive microservices

Chapter 4. Building Reactive Microservice Systems

The previous chapter focused on building microservices, but this chapter is all about building systems. Again, one microservice doesn't make a service—they come in systems. When you embrace the microservice architectural style, you will have dozens of microservices. Managing two microservices, as we did in the last chapter, is easy. The more microservices you use, the more complex the application becomes.

First, we will learn how service discovery can be used to address location transparency and mobility. Then, we will discuss resilience and stability patterns such as timeouts, circuit breakers, and fail-overs.

Service Discovery

When you have a set of microservices, the first question you have to answer is: how will these microservices locate each other? In order to communicate with another peer, a microservice needs to know its *address*. As we did in the previous chapter, we could hard-code the address (event bus address, URLs, location details, etc.) in the code or have it externalized into a configuration file. However, this solution does not enable mobility. Your application will be quite rigid and the different pieces won't be able to move, which contradicts what we try to achieve with microservices.

Client- and Server-Side Service Discovery

Microservices need to be mobile but addressable. A consumer needs to be able to communicate with a microservice without knowing its exact location in advance, especially since this location may change over time. Location transparency provides elasticity and dynamism: the consumer may call different instances of the microservice using a round-robin strategy, and between two invocations the microservice may have been moved or updated.

Location transparency can be addressed by a pattern called *service discovery*. Each microservice should announce how it can be invoked and its characteristics, including its location of course,

but also other metadata such as security policies or versions. These announcements are stored in the *service discovery infrastructure*, which is generally a *service registry* provided by the execution environment. A microservice can also decide to withdraw its service from the registry. A microservice looking for another service can also search this service registry to find matching services, select the *best* one (using any kind of criteria), and start using it. These interactions are depicted in Figure 4-1.

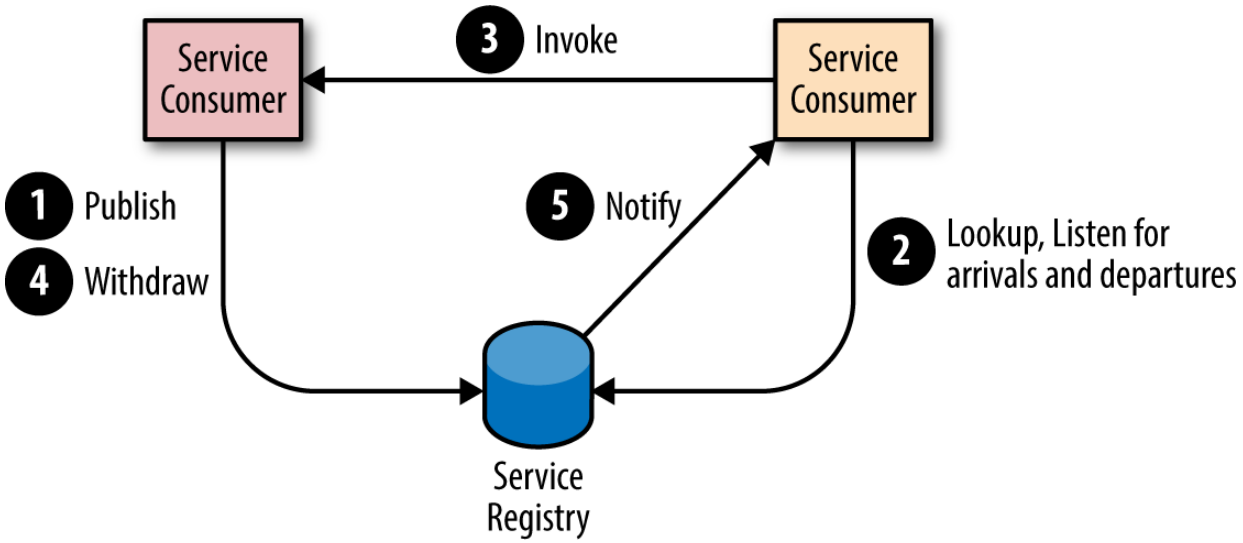


Figure 4-1. Interactions with the service registry

Two types of patterns can be used to consume services. When using *client-side service discovery*, the consumer service looks for a service based on its name and metadata in the service registry, selects a matching service, and uses it. The reference retrieved from the service registry contains a direct link to a microservice. As microservices are dynamic entities, the *service discovery infrastructure* must not only allow providers to publish their services and consumers to look for services, but also provide information about the arrivals and departures of services. When using client-side service discovery, the service registry can take various forms such as a distributed data structure, a dedicated infrastructure such as Consul, or be stored in an inventory service such as Apache Zookeeper or Redis.

Alternatively, you can use *server-side service discovery* and let a load balancer, a router, a proxy, or an API gateway manage the discovery for you (Figure 4-2). The consumer still looks for a service based on its name and metadata but retrieves a *virtual* address. When the consumer invokes the service, the request is routed to the actual implementation. You would use this mechanism on Kubernetes or when using AWS Elastic Load Balancer.

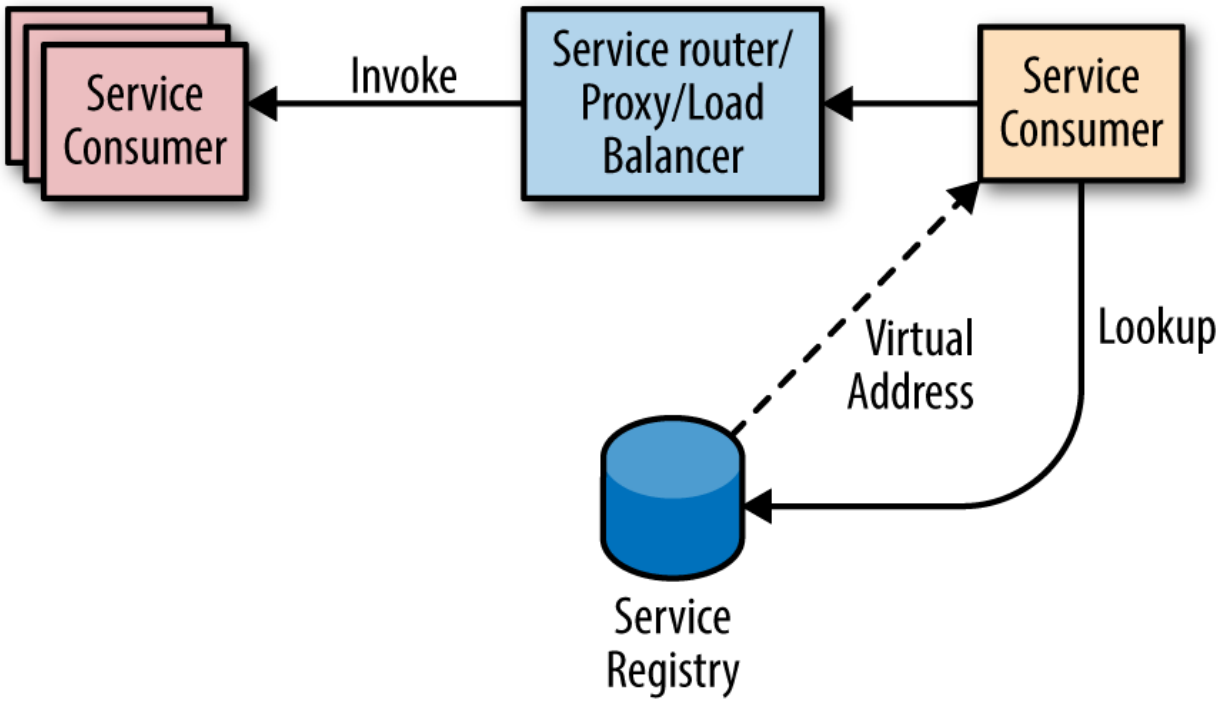


Figure 4-2. Server-side service discovery

Vert.x Service Discovery

Vert.x provides an extensible service discovery mechanism. You can use client-side or server-side service discovery using the same API. The Vert.x service discovery can import or export services from many types of service discovery infrastructures such as Consul or Kubernetes (Figure 4-3). It can also be used without any dedicated service discovery infrastructure. In this case, it uses a distributed data structure shared on the Vert.x cluster.

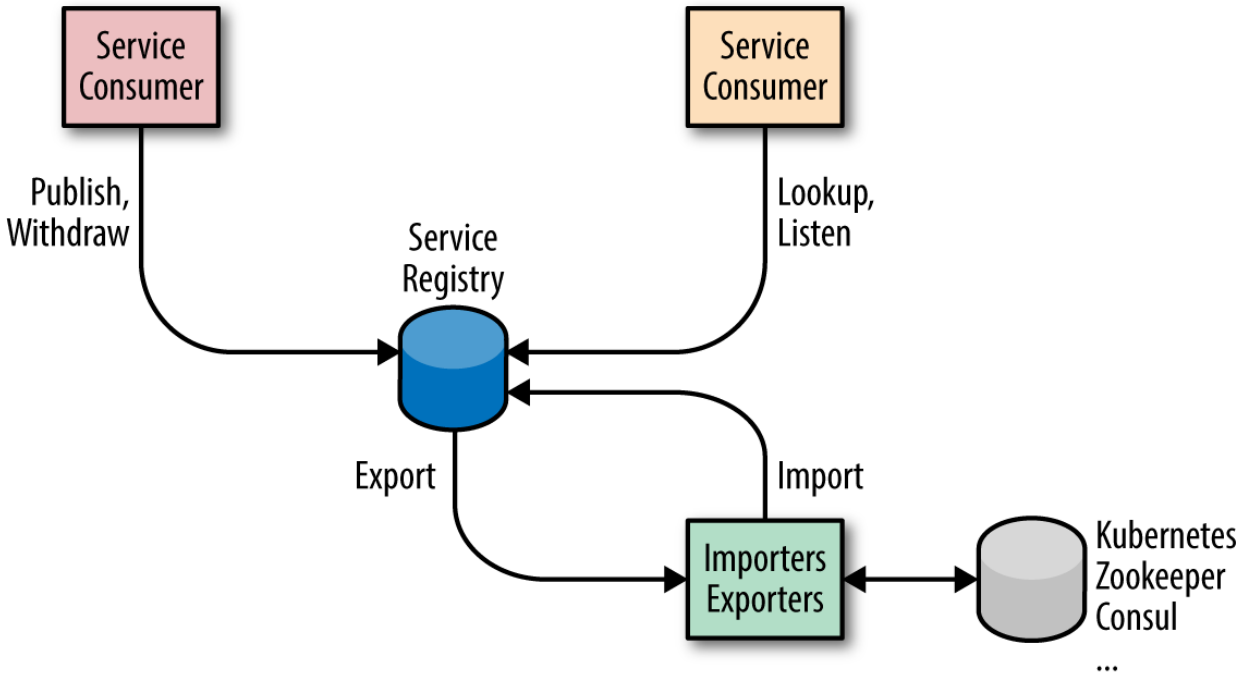


Figure 4-3. Import and export of services from and to other service discovery mechanisms

You can retrieve services by *types* to get a configured *service client* ready to be used. A service type can be an HTTP endpoint, an event bus address, a data source, and so on. For example, if you want to retrieve the HTTP endpoint named *hello* that we implemented in the previous chapter, you would write the following code:

```
// We create an instance of service discovery
ServiceDiscovery discovery = ServiceDiscovery.create(vertx);
// As we know we want to use an HTTP microservice, we can
// retrieve a WebClient already configured for the service
HttpEndpoint
  .rxGetWebClient(discovery,
    // This method is a filter to select the service
    rec -> rec.getName().endsWith("hello")
  )
  .flatMap(client ->
    // We have retrieved the WebClient, use it to call
    // the service
    client.get("/").as(BodyCodec.string()).rxSend())
  .subscribe(response -> System.out.println(response.body()));
```

The retrieved `WebClient` is configured with the service location, which means you can immediately use it to call the service. If your environment is using client-side discovery, the configured URL targets a specific instance of the service. If you are using server-side discovery, the client uses a *virtual* URL.

Depending on your runtime infrastructure, you may have to register your service. But when using server-side service discovery, you usually don't have to do this since you declare your service when it is deployed. Otherwise, you need to publish your service explicitly. To publish a service, you need to create a `Record` containing the service name, location, and metadata:

```
// We create the service discovery object
ServiceDiscovery discovery = ServiceDiscovery.create(vertx);
vertx.createHttpServer()
  .requestHandler(req -> req.response().end("hello"))
  .rxListen(8083)
  .flatMap(
    // Once the HTTP server is started (we are ready to serve)
    // we publish the service.
    server -> {
      // We create a record describing the service and its
      // location (for HTTP endpoint)
      Record record = HttpEndpoint.createRecord(
        "hello",              // the name of the service
        "localhost",          // the host
        server.actualPort(),  // the port
        "/"                   // the root of the endpoint
      );
```

```
        // We publish the service
        return discovery.rxPublish(record);
      }
    )
    .subscribe(rec -> System.out.println("Service published"));
```

Service discovery is a key component in a microservice infrastructure. It enables dynamism, location transparency, and mobility. When dealing with a small set of services, service discovery may look cumbersome, but it's a must-have when your system grows. The Vert.x service discovery provides you with a unique API regardless of the infrastructure and the type of service discovery you use. However, when your system grows, there is also another variable that grows exponentially—failures.

Stability and Resilience Patterns

When dealing with distributed systems, failures are first-class citizens and you have to live with them. Your microservices must be aware that the services they invoke can fail for many reasons. Every interaction between microservices will eventually fail in some way, and you need to be prepared for that failure. Failure can take different forms, ranging from various network errors to semantic errors.

Managing Failures in Reactive Microservices

Reactive microservices are responsible for managing failures locally. They must avoid propagating the failure to another microservice. In other words, you should not delegate the *hot potato* to another microservice. Therefore, the code of a reactive microservice considers failures as first-class citizens.

The Vert.x development model makes failures a central entity. When using the callback development model, the `Handlers` often receive an `AsyncResult` as a parameter. This structure encapsulates the result of an asynchronous operation. In the case of success, you can retrieve the result. On failure, it contains a `Throwable` describing the failure:

```
client.get("/").as(BodyCodec.jsonObject())
    .send(ar -> {
        if (ar.failed()) {
            Throwable cause = ar.cause();
            // You need to manage the failure.
        } else {
            // It's a success
            JsonObject json = ar.result().body();
        }
    });
```

When using the RxJava APIs, the failure management can be made in the `subscribe` method:

```
client.get("/").as(BodyCodec.jsonObject())
    .rxSend()
    .map(HttpResponse::body)
    .subscribe(
        json -> { /* success */ },
        err -> { /* failure */ }
    );
```

If a failure is produced in one of the observed *streams*, the error handler is called. You can also handle the failure earlier, avoiding the error handler in the `subscribe` method:

```
client.get("/").as(BodyCodec.jsonObject())
    .rxSend()
    .map(HttpResponse::body)
    .onErrorReturn(t -> {
        // Called if rxSend produces a failure
        // We can return a default value
        return new JsonObject();
    })
    .subscribe(
        json -> {
            // Always called, either with the actual result
            // or with the default value.
        }
    );
```

Managing errors is not *fun* but it has to be done. The code of a reactive microservice is responsible for making an adequate decision when facing a failure. It also needs to be prepared to see its requests to other microservices fail.

Using Timeouts

When dealing with distributed interactions, we often use timeouts. A timeout is a simple mechanism that allows you to stop waiting for a response once you *think* it will not come. Well-placed timeouts provide failure isolation, ensuring the failure is limited to the microservice it affects and allowing you to handle the timeout and continue your execution in a *degraded* mode.

```
client.get(path)
  .rxSend() // Invoke the service
  // We need to be sure to use the Vert.x event loop
  .subscribeOn(RxHelper.scheduler(vertx))
  // Configure the timeout, if no response, it publishes
  // a failure in the Observable
```

```
    .timeout(5, TimeUnit.SECONDS)
    // In case of success, extract the body
    .map(HttpResponse::bodyAsJsonObject)
    // Otherwise use a fallback result
    .onErrorReturn(t -> {
      // timeout or another exception
      return new JsonObject().put("message", "D'oh! Timeout");
    })
    .subscribe(
      json -> {
        System.out.println(json.encode());
      }
    );
```

Timeouts are often used together with retries. When a timeout occurs, we can try again. Immediately retrying an operation after a failure has a number of effects, but only some of them are beneficial. If the operation failed because of a significant problem in the called microservice, it is likely to fail again if retried immediately. However, some kinds of transient failures can be overcome with a retry, especially network failures such as dropped messages. You can decide whether or not to reattempt the operation as follows:

```
client.get(path)
  .rxSend()
  .subscribeOn(RxHelper.scheduler(vertx))
  .timeout(5, TimeUnit.SECONDS)
  // Configure the number of retries
  // here we retry only once.
  .retry(1)
  .map(HttpResponse::bodyAsJsonObject)
  .onErrorReturn(t -> {
    return new JsonObject().put("message", "D'oh! Timeout");
  })
  .subscribe(
    json -> System.out.println(json.encode())
  );
```

It's also important to remember that a timeout does not imply an operation failure. In a distributed system, there are many reasons for failure. Let's look at an example. You have two microservices, *A* and *B*. *A* is sending a request to *B*, but the response does not come in time and *A* gets a timeout. In this scenario, three types of failure could have occurred:

1. The message between *A* and *B* has been lost—the operation is not executed.
2. The operation in *B* failed—the operation has not completed its execution.
3. The response message between *B* and *A* has been lost—the operation has been executed

successfully, but *A* didn't get the response.

This last case is often ignored and can be harmful. In this case, combining the timeout with a retry can break the integrity of the system. Retries can only be used with idempotent operations, i.e., with operations you can invoke multiple times without changing the result beyond the initial call. Before using a retry, always check that your system is able to handle reattempted operations gracefully.

Retry also makes the consumer wait even longer to get a response, which is not a good thing either. It is often better to return a fallback than to retry an operation too many times. In addition, continually hammering a failing service may not help it get back on track. These two concerns are managed by another resilience pattern: the circuit breaker.

Circuit Breakers

A circuit breaker is a pattern used to deal with repetitive failures. It protects a microservice from calling a failing service again and again. A circuit breaker is a three-state automaton that manages an interaction (Figure 4-4). It starts in a *closed* state in which the circuit breaker executes operations as usual. If the interaction succeeds, nothing happens. If it fails, however, the circuit breaker makes a note of the failure. Once the number of failures (or frequency of failures, in more sophisticated cases) exceeds a threshold, the circuit breaker switches to an *open* state. In this state, calls to the circuit breaker fail immediately without any attempt to execute the underlying interaction. Instead of executing the operation, the circuit breaker may execute a fallback, providing a *default* result. After a configured amount of time, the circuit breaker decides that the operation has a chance of succeeding, so it goes into a *half-open* state. In this state, the next call to the circuit breaker executes the underlying interaction. Depending on the outcome of this call, the circuit breaker resets and returns to the *closed* state, or returns to the *open* state until another timeout elapses.

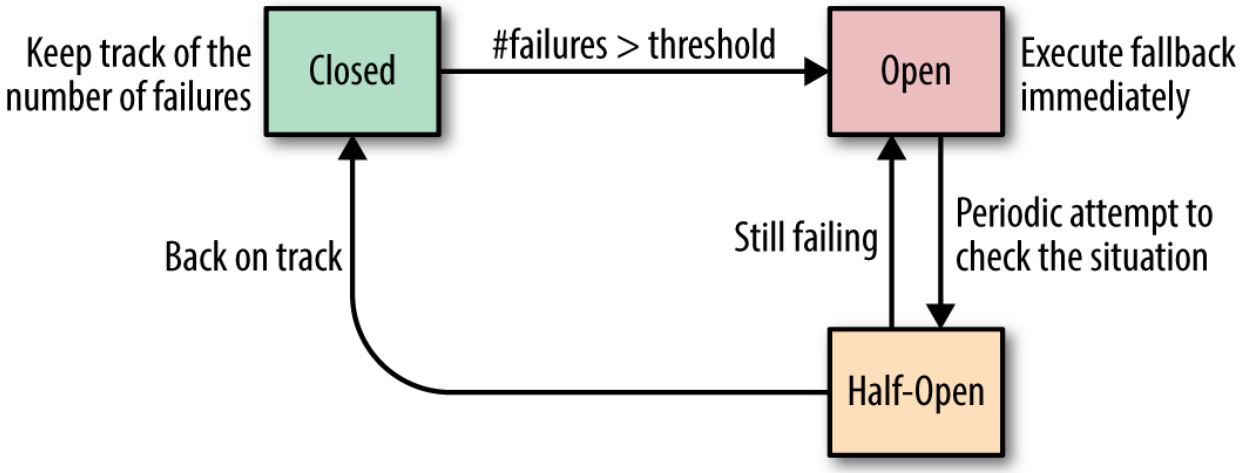


Figure 4-4. Circuit breaker states

The most well-known circuit breaker implementation in Java is Hystrix (*https://github.com/Netflix/Hystrix*). While you can use Hystrix in a Vert.x microservice (it uses a thread pool), you need to explicitly switch to the Vert.x event loop to execute the different callbacks. Alternatively, you can use the Vert.x circuit breaker built for asynchronous operations and enforce the Vert.x nonblocking asynchronous development model.

Let's imagine a failing *hello* microservice. The consumer should protect the interactions with this service and use a circuit breaker as follows:

```
CircuitBreaker circuit = CircuitBreaker.create("my-circuit",
    vertx, new CircuitBreakerOptions()
        .setFallbackOnFailure(true) // Call the fallback
                                    // on failures
        .setTimeout(2000)           // Set the operation timeout
        .setMaxFailures(5)          // Number of failures before
                                    // switching to
                                    // the 'open' state
        .setResetTimeout(5000)      // Time before attempting
                                    // to reset
                                    // the circuit breaker
);
// ...
circuit.rxExecuteCommandWithFallback(
    future ->
        client.get(path)
            .rxSend()
            .map(HttpResponse::bodyAsJsonObject)
            .subscribe(future::complete, future::fail),
    t -> new JsonObject().put("message", "D'oh! Fallback")
).subscribe(
        json -> {
            // Get the actual json or the fallback value
            System.out.println(json.encode());
```

```
            }
    );
```

In this code, the HTTP interaction is protected by the circuit breaker. When the number of failures reaches the configured threshold, the circuit breaker will stop calling the microservice and instead call a fallback. Periodically, the circuit breaker will let one invocation pass through to check whether the microservice is back on track and act accordingly. This example uses a web client, but any interaction can be managed with a circuit breaker and protect you against flaky services, exceptions, and other sorts of failures.

A circuit breaker switching to an *open* state needs to be monitored by your operations team. Both Hystrix and the Vert.x circuit breaker have monitoring capabilities.

Health Checks and Failovers

While timeouts and circuit breakers allow consumers to deal with failures on their side, what about crashes? When facing a crash, a failover strategy restarts the parts of the system that have failed. But before being able to achieve this, we must be able to detect when a microservice has died.

A health check is an API provided by a microservice indicating its state. It tells the caller whether or not the service is *healthy*. The invocation often uses HTTP interactions but is not necessary. After invocation, a set of checks is executed and the global state is computed and returned. When a microservice is detected to be unhealthy, it should not be called anymore, as the outcome is probably going to be a failure. Note that calling a healthy microservice does not guarantee a success either. A health check merely indicates that the microservice is running, not that it will accurately handle your request or that the network will deliver its answer.

Depending on your environment, you may have different levels of health checks. For instance, you may have a *readiness* check used at deployment time to determine when the microservice is ready to serve requests (when everything has been initialized correctly). Liveness checks are used to detect misbehaviors and indicate whether the microservice is able to handle requests successfully. When a liveness check cannot be executed because the targeted microservice does not respond, the microservice has probably crashed.

In a Vert.x application, there are several ways to implement health checks. You can simply implement a *route* returning the state, or even use a real request. You can also use the Vert.x health check module to implement several health checks in your application and compose the different outcomes. The following code gives an example of an application providing two levels of health checks:

```
Router router = Router.router(vertx);
HealthCheckHandler hch = HealthCheckHandler.create(vertx);
// A procedure to check if we can get a database connection
hch.register("db-connection", future -> {
  client.rxGetConnection()
    .subscribe(c -> {
        future.complete();
        c.close();
      },
      future::fail
    );
});
// A second (business) procedure
hch.register("business-check", future -> {
  // ...
});
// Map /health to the health check handler
router.get("/health").handler(hch);
// ...
```

After you have completed health checks, you can implement a fail-over strategy. Generally, the strategy just restarts the dead part of the system, hoping for the best. While failover is often provided by your runtime infrastructure, Vert.x offers a built-in failover, which is triggered when a node from the cluster dies. With the built-in Vert.x failover, you don't need a custom health check as the Vert.x cluster pings nodes periodically. When Vert.x loses track of a node, Vert.x chooses a healthy node of the cluster and redeploys the dead part.

Failover keeps your system running but won't fix the root cause—that's your job. When an application dies unexpectedly, a postmortem analysis should be done.

Summary

This chapter has addressed several concerns you will face when your microservice system grows. As we learned, service discovery is a *must-have* in any microservice system to ensure location transparency. Then, because failures are inevitable, we discussed a couple of patterns to improve the resilience and stability of your system.

Vert.x includes a pluggable service discovery infrastructure that can handle client-side service discovery and server-side service discovery using the same API. The Vert.x service discovery is also able to import and export services from and to different service discovery infrastructures. Vert.x includes a set of resilience patterns such as timeout, circuit breaker, and failover. We saw different examples of these patterns. Dealing with failure is, unfortunately, part of the job and we all have to do it.

In the next chapter, we will learn how to deploy Vert.x reactive microservices on OpenShift and

illustrate how service discovery, circuit breakers, and failover can be used to make your system almost *bulletproof*. While these topics are particularly important, don't underestimate the other concerns that need to be handled when dealing with microservices, such as security, deployment, aggregated logging, testing, etc.

If you want to learn more about these topics, check the following resources:

- *Reactive Microservices Architecture*
- The Vert.x service discovery documentation
- *Release It! Design and Deploy Production-Ready Software* (O'Reilly) A book providing a list of recipes to make your system ready for production
- Netflix Hystrix
- The Vert.x service circuit breaker documentation

Chapter 5. Deploying Reactive Microservices in OpenShift

So far, we have only deployed our microservices on a local machine. What happens when we deploy a microservice on the cloud? Most cloud platforms include services to make your deployment and operations easier. The ability to scale up and down and load balance are some of the commonly found features that are particularly relevant to developing reactive microservices. In this chapter, we will see how these features (and others) can be used to develop and deploy reactive microservices.

To illustrate these benefits, we will use OpenShift (*https://www.openshift.org/*). However, most modern cloud platforms include the features we use here. By the end of this chapter, you will see how the cloud makes *reactiveness* easy for everyone.

What Is OpenShift?

RedHat OpenShift v3 is an open source container platform. With OpenShift you deploy applications running in containers, which makes their construction and administration easy. OpenShift is built on top of Kubernetes (*https://kubernetes.io/*).

Kubernetes (in blue in Figure 5-1) is a project with lots of functionality for running clusters of microservices inside Linux containers at scale. Google has packaged over a decade of experience with containers into Kubernetes. OpenShift is built on top of this experience and extends it with build and deployment automation (in green in Figure 5-1). Use cases such as rolling updates, canary deployments, and continuous delivery pipelines are provided out of the box.

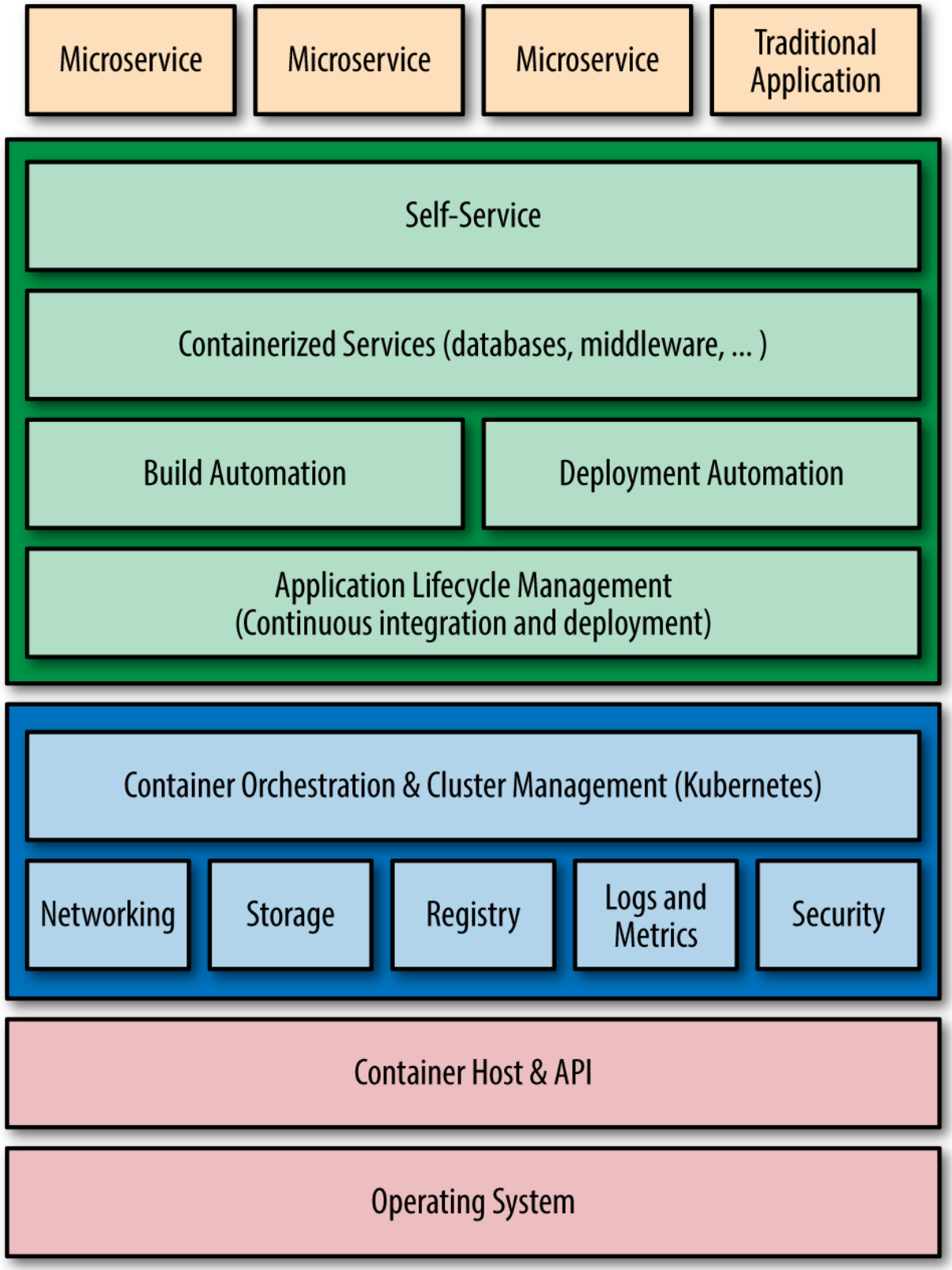


Figure 5-1. The OpenShift container platform

OpenShift has a handful of simple entities, as depicted in Figure 5-2, that we need to understand before putting it to work.

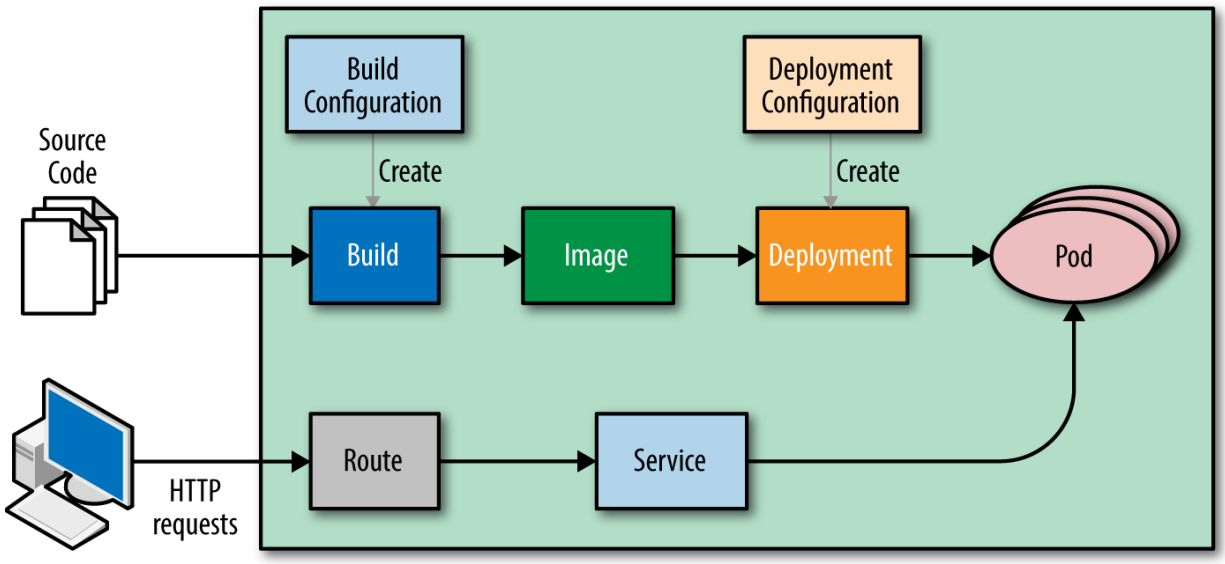


Figure 5-2. The OpenShift entities

Build Configuration

The *build* is the process of creating container images that will be used by OpenShift to instantiate the different containers that make up an application. OpenShift builds can use different strategies:

- Docker—Build an image from a `Dockerfile`
- Source to Image (S2I)—Build an image from the application source, built on OpenShift by a *builder* image
- Jenkins Pipeline—Build an image using a Jenkins pipeline (*https://jenkins.io/doc/book/pipeline*) potentially containing multiple stages such as build, tests, and deployment

A build configuration can be triggered automatically by a *git push*, a change in the configuration or an update in a dependent image, and, obviously, manually.

Deployment Configurations

A *deployment configuration* defines the instantiation of the image produced by a *build*. It defines which image is used to create the containers and the number of instances we need to keep alive. It also describes when a deployment should be triggered. A *deployment* also acts as a *replication controller* and is responsible for keeping containers alive. To achieve this, you pass the number of desired instances. The number of desired instances can be adjusted over time or based on the load fluctuation (auto-scaling). The *deployment* can also specify health checks to manage rolling updates and detect dead containers.

Pods

A *pod* is a group of one or more containers. However, it is typically comprised of a single container. The pod orchestration, scheduling, and management are delegated to Kubernetes. Pods are fungible, and can be replaced at any time by another instance. For example, if the container crashes, another instance will be spawned.

Services and Routes

Because pods are dynamic entities (the number of instances can change over time), we cannot rely on their direct IP addresses (each pod has its own IP address). *Services* allow us to communicate with the pods without relying on their addresses but by using the *service virtual* address. A service acts as a proxy in front of a group of pods. It may also implement a load-balancing strategy.

Other applications running in OpenShift can access the functionality offered by the pods using the service, but external applications need a *route*. A *route* exposes a service at a hostname like `www.myservice.com` so that external clients can reach it by name.

Installing OpenShift on Your Machine

That's enough abstract concepts. Now it's time for action. We are going to install Openshift on your machine using Minishift (*https://github.com/minishift/minishift*). Alternatively, you can use OpenShift Online (*https://www.openshift.com/devpreview/*) or the Red Hat Container Development Kit v3 (*https://developers.redhat.com/products/cdk/download/*).

Installing Minishift (*https://github.com/minishift/minishift#installation*) requires a hypervisor to run the virtual machine containing OpenShift. Depending on your host OS, you have a choice of hypervisors; check the Minishift installation guide for details.

To install Minishift, just download the latest archive for your OS from the Minishift releases page (*https://github.com/minishift/minishift/releases*), unpack it to your preferred location, and add the `minishift` binary to your `PATH` environment variable. Once installed, start Minishift using:

```
minishift start
```

Once started, you should be able to connect to your OpenShift instance on *https://192.168.64.12:8443*. You may have to validate the SSL certificate. Log in with `developer/developer`.

We also need the OpenShift *client* (`oc`), a command-line utility used to interact with your OpenShift instance. Download the latest version of the OpenShift client from *https://github.com/openshift/origin/releases/latest*. Unpack it to your preferred location and add the `oc` binary to your `PATH` environment variable.

Then, connect to your OpenShift instance using:

```
oc login https://192.168.64.12:8443 -u developer -p developer
```

OpenShift has a namespace concept called *project*. To create projects for the examples we are going to deploy, execute:

```
oc new-project reactive-microservices
oc policy add-role-to-user admin developer -n
 reactive-microservices
oc policy add-role-to-user view -n reactive-microservices
 -z default
```

In your browser, open *https://192.168.64.12:8443/console/project/reactive-microservices/*. You should be able to see the project, which is not very interesting at the moment as we haven't deployed anything (Figure 5-3).

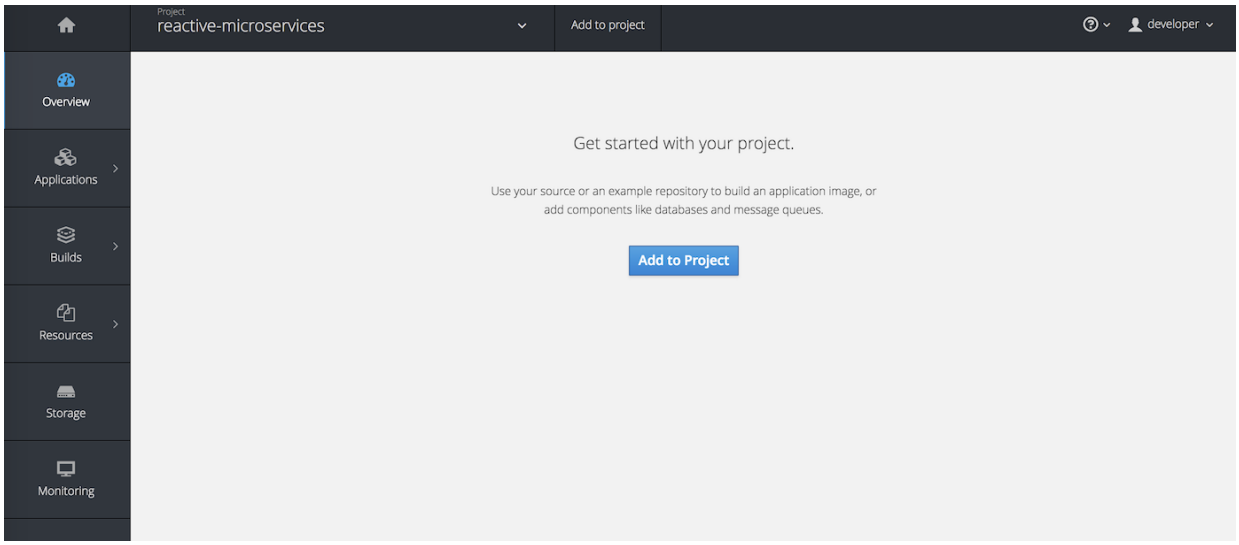


Figure 5-3. The created project

Deploying a Microservice in OpenShift

It's time to deploy a microservice to OpenShift. The code we are going to deploy is contained in the `openshift/hello-microservice-openshift` directory of the code repository. The verticle is very close to the *hello microservice (HTTP)* we developed earlier:

```
package io.vertx.book.openshift;

import io.vertx.core.AbstractVerticle;
import io.vertx.core.http.HttpHeaders;
import io.vertx.core.json.JsonObject;
import io.vertx.ext.web.*;

public class HelloHttpVerticle extends AbstractVerticle {

  static final String HOSTNAME = System.getenv("HOSTNAME");

  @Override
  public void start() {
    Router router = Router.router(vertx);
    router.get("/").handler(this::hello);
    router.get("/:name").handler(this::hello);
    vertx.createHttpServer()
      .requestHandler(router::accept)
      .listen(8080);
  }

  private void hello(RoutingContext rc) {
    String message = "hello";
    if (rc.pathParam("name") != null) {
      message += " " + rc.pathParam("name");
    }
    JsonObject json = new JsonObject()
      .put("message", message)
      .put("served-by", HOSTNAME);
    rc.response()
      .putHeader(HttpHeaders.CONTENT_TYPE, "application/json")
      .end(json.encode());
  }
}
```

This code does not rely on specific OpenShift APIs or constructs. It's your application as you would have developed it on your machine. The separation of Java code from deployment choices must be a deliberate design choice to make the code runnable on any cloud platform.

We could create all the OpenShift entities manually, but let's use the Maven plug-in (*https://maven.fabric8.io/*) provided by Fabric8, an end-to-end development platform for Kubernetes. If you open the `pom.xml` file, you will see that this plug-in is configured in the `openshift` profile and collaborates with the Vert.x Maven plug-in to create the OpenShift entities.

To package and deploy our microservice to OpenShift, launch:

```
mvn fabric8:deploy -Popenshift
```

This command interacts with your OpenShift instance (on which you have logged in with `oc`) to create a *build* (using the source to image build strategy) and trigger it. This first build can take some time as it needs to retrieve the *builder* image. Don't worry—once everything is cached, the builds will be created faster. The output of the build (and *image*) is used by the *deployment configuration*, which is also created by the Fabric8 Maven plug-in. By default, it creates one pod. A *service* is also created by the plug-in. You can find all this information in the OpenShift dashboard, as shown in Figure 5-4.

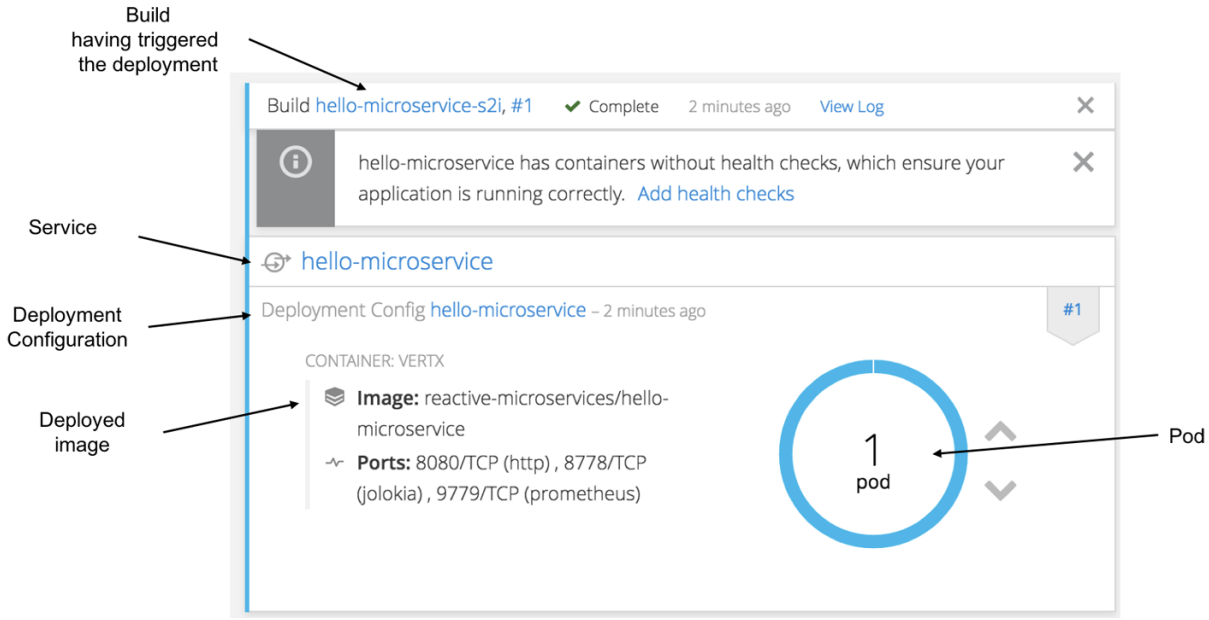


Figure 5-4. Entities created during our first deployment

Routes are not created by default by the Fabric8 Maven plug-in. However, we created one from its description (`src/main/fabric8/route.yml`). If you open your browser to *http://hello-microservice-reactive-microservices.192.168.64.12.nip.io/Luke*, you should see something like:

```
{"message":"hello Luke","served-by":
 "hello-microservice-1-9r8uv"}
```

`hello-microservice-1-9r8uv` is the name of the pod serving the request.

Service Discovery

Now that we have the *hello* microservice deployed, let's consume it from another microservice. The code we are going to deploy in this section is contained in the `openshift/hello-microservice-consumer-openshift` directory from the code repository.

To consume a microservice, we first have to find it. OpenShift provides a service discovery mechanism. Service lookup can be done using environment variables, DNS, or the Vert.x service discovery, which we use here. The project `pom.xml` is configured to import the Vert.x service discovery, the Kubernetes service importer, and a server-side service discovery. You don't have to explicitly register the service on the provider side as the Fabric8 Maven plug-in declares a *service* for us. Our consumer is going to retrieve this OpenShift *service* and not the *pods*.

```
@Override
public void start() {
  Router router = Router.router(vertx);
  router.get("/").handler(this::invokeHelloMicroservice);
  // Create the service discovery instance
  ServiceDiscovery.create(vertx, discovery -> {
    // Look for an HTTP endpoint named "hello-microservice"
    // you can also filter on 'label'
    Single<WebClient> single = HttpEndpoint.rxGetWebClient
      (discovery, rec -> rec.getName().equals
       ("hello-microservice"),
      new JsonObject().put("keepAlive", false));
    single.subscribe(
      client -> {
        // the configured client to call the microservice
        this.hello = client;
        vertx.createHttpServer()
          .requestHandler(router::accept)
          .listen(8080);
      },
      err -> System.out.println("Oh no, no service")
    );
  });
}
```

In the `start` method, we use the service discovery to find the *hello* microservice. Then, if the service is available, we start the HTTP server and keep a reference on the retrieved `WebClient`. We also pass a configuration to the `WebClient` and disable the `keep-alive` settings (we will see the reason for this in a few minutes). In the `invokeHelloMicroservice`, we don't have to pass the port and host to the `rxSend` method (as we did previously). Indeed, the `WebClient` is configured to target the *hello* service:

```
HttpRequest<JsonObject> request1 = hello.get("/Luke")
  .as(BodyCodec.jsonObject());
HttpRequest<JsonObject> request2 = hello.get("/Leia")
  .as(BodyCodec.jsonObject());
Single<JsonObject> s1 = request1.rxSend()
```

```
    .map(HttpResponse::body);
Single<JsonObject> s2 = request2.rxSend()
    .map(HttpResponse::body);
    // ...
```

In a terminal, navigate to the `openshift/hello-microservice-consumer-openshift` directory to build and deploy this consumer with:

```
mvn fabric8:deploy -Popenshift
```

In the OpenShift dashboard, you should see a second service and route (*http://bit.ly/2o4xaSk*). If you open the route associated with the *hello-consumer* service (*http://bit.ly/2p2aHTK*), you should see:

```
{
  "luke" : "hello Luke hello-microservice-1-sa5pf",
  "leia" : "hello Leia hello-microservice-1-sa5pf"
}
```

You may see a `503` error page, since the pod has not yet started. Just refresh until you get the right page. So far, nothing surprising. The displayed `served-by` values are always indicating the same pod (as we have only one).

Scale Up and Down

If we are using a cloud platform, it's mainly for scalability reasons. We want to be able to increase and decrease the number of instances of our application depending on the load. In the OpenShift dashboard we can scale the number of pods up and down, as shown in Figure 5-5.

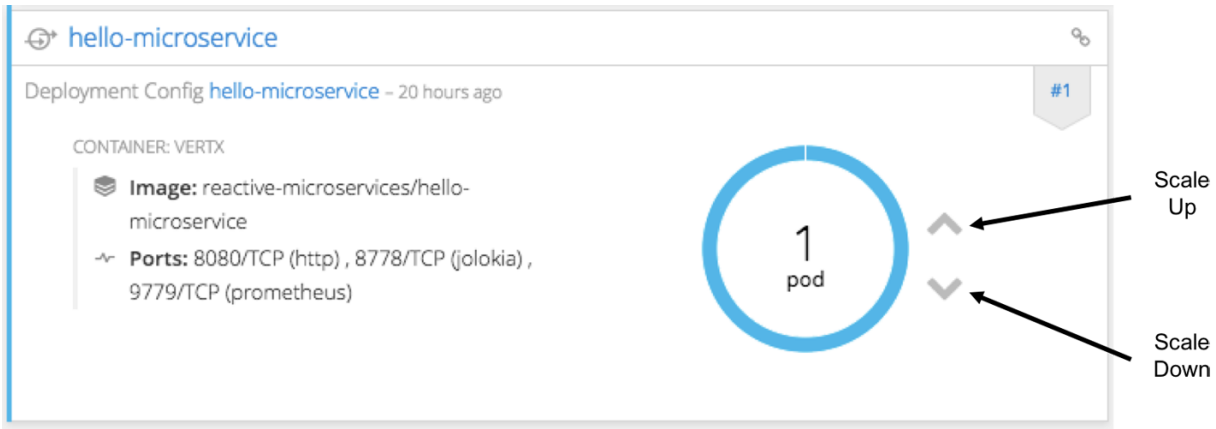


Figure 5-5. Scale up and down

You can also set the number of replicas using the `oc` command line:

```
# scale up to 2 replicas
oc scale --replicas=2 dc hello-microservice

# scale down to 0
oc scale --replicas=0 dc hello-microservice
```

Let's create a second instance of our *hello* microservice. Then, wait until the second microservice has started correctly (the wait time is annoying, but we will fix that later), and go back to the *hello-consumer* page in a browser. You should see something like:

```
{
  "luke" : "hello Luke hello-microservice-1-h6bs6",
  "leia" : "hello Leia hello-microservice-1-keq8s"
}
```

If you refresh several times, you will see that the OpenShift service balances the load between the two instances. Do you remember the `keep-alive` settings we disabled? When the HTTP connection uses a `keep-alive` connection, OpenShift forwards the request to the same pod, providing connection affinity. Note that in practice, `keep-alive` is a very desirable header as it allows reusing connections.

In the previous scenario there is a *small* issue. When we scale up, OpenShift starts dispatching requests to the new pod without checking whether the application is ready to serve these requests. So, our consumer may call a microservice that is not ready and get a failure. There are a couple of ways to address this:

1. Using health checks in the microservice
2. Be prepared to face the failure in the consumer code

Health Check and Failover

In OpenShift you can declare two types of checks. *Readiness* checks are used to avoid downtime when updating a microservice. In a rolling update, OpenShift waits until the new version is ready before shutting down the previous version. It pings the readiness check endpoint of the new microservice until it is ready, and verifies that the microservice has been successfully initialized. *Liveness* checks are used to determine whether a pod is alive. OpenShift invokes the liveness check endpoint periodically. If a pod does not reply positively to the check, it will be restarted. A liveness check focuses on the critical resources required by the microservice to behave correctly. In the following example we will use the same endpoint for both checks. However, it's best to use two different endpoints.

The code of this example is contained in the `openshift/hello-microservice-openshift-health-checks` directory. If you open the verticle, you will see the `HealthCheck` handler verifying whether or not the HTTP server has been started:

```
private boolean started;

@Override
public void start() {
  Router router = Router.router(vertx);
  router.get("/health").handler(
    HealthCheckHandler.create(vertx)
      .register("http-server-running",
        future -> future.complete(
          started ? Status.OK() : Status.KO())));
  router.get("/").handler(this::hello);
  router.get("/:name").handler(this::hello);
  vertx.createHttpServer()
    .requestHandler(router::accept)
    .listen(8080, ar -> started = ar.succeeded());
}
```

The Fabric8 Maven plug-in is configured to use `/health` for the readiness and liveness health checks. Once this version of the *hello microservice* is deployed, all subsequent deployments will use the readiness check to avoid downtime, as shown in Figure 5-6.

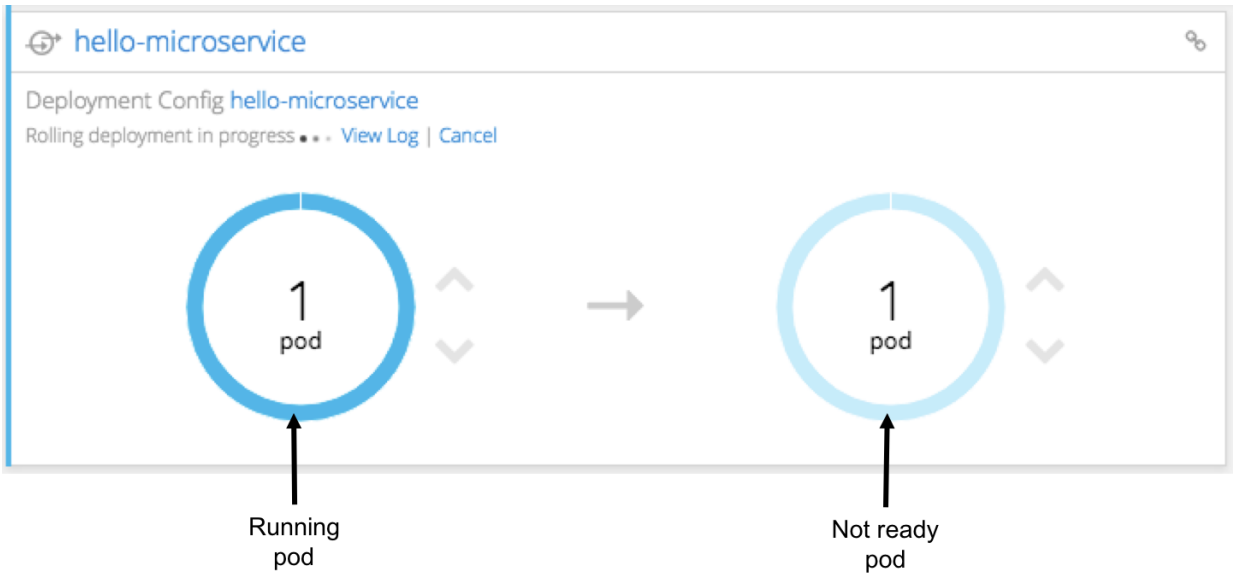


Figure 5-6. Rolling updates

When the pod is ready, OpenShift routes the requests to this pod and shuts down the old one. When we scale up, OpenShift doesn't route requests to a pod that is not ready.

Using a Circuit Breaker

While using health checks avoids calling a microservice that is not ready and restarts dead ones, we still need to protect ourselves from other failures such as timeouts, network outage, bugs in the called microservice, and so on. In this section we are going to protect the *hello consumer* using a circuit breaker. The code of this section is located in the `openshift/hello-microservice-consumer-openshift-circuit-breaker` directory.

In the verticle, we are using a single circuit breaker to protect against the two calls to the *hello microservice*. The following code uses this design; however, it's just one among a large set of possible approaches, such as using one circuit breaker per call, using a single circuit breaker but protecting the two calls independently, etc.:

```
private void invokeHelloMicroservice(RoutingContext rc) {
  circuit.rxExecuteCommandWithFallback(
    future -> {
      HttpRequest<JsonObject> request1 = hello.get("/Luke")
        .as(BodyCodec.jsonObject());
      HttpRequest<JsonObject> request2 = hello.get("/Leia")
        .as(BodyCodec.jsonObject());
      Single<JsonObject> s1 = request1
        .rxSend().map(HttpResponse::body);
      Single<JsonObject> s2 = request2
        .rxSend().map(HttpResponse::body);
      Single
        .zip(s1, s2, (luke, leia) -> {
          // We have the result of both request in Luke and Leia
          return new JsonObject()
            .put("Luke", luke.getString("message")
              + " " + luke.getString("served-by"))
            .put("Leia", leia.getString("message")
              + " " + leia.getString("served-by"));
        })
        .subscribe(future::complete, future::fail);
    },
    error -> new JsonObject().put("message", "hello (fallback, "
      + circuit.state().toString() + ")")
  ).subscribe(
    x -> rc.response().end(x.encodePrettily()),
    t -> rc.response().end(t.getMessage()));
}
```

In case of error, we provide a fallback message indicating the state of the circuit breaker. This will help us understand what's going on. Deploy this project using:

```
mvn fabric8:deploy -Popenshift
```

Now let's scale down the *hello microservice* to 0. To do this, we can click on the down arrow

near the pod, or run:

```
oc scale --replicas=0 dc hello-microservice
```

Now if you refresh the consumer page (*http://hello-consumer-reactive-microservices.192.168.64.12.nip.io/*), you should see the fallback message. The first three requests show:

```
{
  "message" : "hello (fallback, CLOSED)"
}
```

Once the number of failures is reached, it returns:

```
{
  "message" : "hello (fallback, OPEN)"
}
```

If you restore the number of replicas to 1 with:

```
oc scale --replicas=1 dc hello-microservice
```

you should get back the normal output once the microservice is ready.

But Wait, Are We Reactive?

Yes, we are. Let's see why.

All our interactions are asynchronous. They use asynchronous and nonblocking HTTP requests and responses. In addition, thanks to the OpenShift *service*, we are sending the requests to a *virtual address*, which enables elasticity. The service balances the load among a set of pods. We can easily scale up and down by adjusting the number of pods or by using auto-scaling. We are also resilient. Thanks to the health checks, we have a failover mechanism ensuring we always have the right number of pods running. On the consumer side, we can use various resilience patterns such as timeouts, retries, or circuit breakers to protect the microservice from failures. So our system is able to handle requests in a timely fashion under load and when facing failures: we are responsive!

Can any system using asynchronous nonblocking HTTP be a *reactive system* in a cloud providing load-balancing and some resilience features? Yes, but don't forget the costs. Vert.x

uses an event loop to handle a high level of concurrency with a minimum of threads, exhibiting a cloud native nature. When using approaches relying on thread pools, you will need to 1) tune the thread pool to find the right size; 2) handle the concurrency in your code, which means debugging deadlocks, race conditions, and bottlenecks; and 3) monitor performance. Cloud environments are based on virtualization, and thread scheduling can become a major issue when you have many threads.

There are many nonblocking technologies, but not all of them use the same execution model to handle the asynchronous nature We can classify these technologies in three categories:

1. Approaches using a thread pool in the background—Then you are facing a tuning, scheduling, and concurrency challenge shifting the burden to *ops*.

2. Approaches using another thread for callbacks—You still need to manage the thread-safety of your code while avoiding deadlocks and bottlenecks.

3. Approaches, such as Vert.x, using the same thread—You use a small number of threads and are freed from debugging deadlocks.

Could we use messaging systems in the cloud to implement reactive microservice systems? Of course. We could have used the Vert.x event bus to build our reactive microservice in OpenShift. But it would not have demonstrated *service virtual address* and load-balancing provided by OpenShift as it would have been handled by Vert.x itself. Here we decided to go with HTTP, one design among an infinite number of choices. Shape your system the way you want it!

Summary

In this chapter, we deployed microservices in OpenShift and saw how Vert.x and the OpenShift features are combined to build reactive microservices. Combining asynchronous HTTP servers and clients, OpenShift services, load-balancing, failover and consumer-side resilience gives us the characteristics of a *reactive system*.

This report focuses on *reactive*. However, when building a microservice system, lots of other concerns need to be managed such as security, configuration, logging, etc. Most cloud platforms, including OpenShift, provide services to handle these concerns.

If you want to learn more about these topics, check out the following resources:

- OpenShift website

- OpenShift core concepts
- Kubernetes website
- OpenShift health checks documentation

Chapter 6. Conclusion

We are at the end of our journey together, but you have many new avenues to explore. We have covered a lot of content in this small report but certainly didn't cover everything! We have just scratched the surface. There are more things to consider when moving toward reactive microservices. Vert.x is also not limited to microservices and can handle a large set of different use cases.

What Have We Learned?

So, what did you learn in this report? First, we started with microservices and what are *reactive* microservices. We learned that *reactive microservices* are the building blocks of responsive microservice systems. We also saw how *reactive programming* helps to build these microservices.

We discovered Eclipse Vert.x, a toolkit used to build reactive microservices (among many other things). Vert.x provides the perfect paradigm to embrace microservices: asynchronous, failures as first-class citizens, and nonblocking. To tame the asynchronous development model, Vert.x combines its power with RxJava. Our discovery started with HTTP microservices and how to consume them. While HTTP is often used in microservices, we also saw one of its limitations when we directly referenced an instance of a microservice. To address this, we used the Vert.x event bus and saw how message-based microservices let you build reactive microservices and thus *reactive systems*.

Of course, one microservice does not make an application. They come in systems. To build systems, we have to use service discovery. Service discovery enables location transparency and mobility, two important characteristics in microservice systems. We also covered resilience patterns, since microservice systems are distributed systems and you need to be prepared for failure.

In the last chapter, we deployed our microservices on top of OpenShift, an open source container platform based on Kubernetes. The combination of Vert.x and OpenShift simplifies the deployment and execution of reactive microservices and keeps the whole system on track.

So, is this the end? No! It's only the end of the first stage.

Microservices Aren't Easy

Microservices are expected to improve overall *agility*. This improvement is not only a technical issue, it's also an organization issue. If your organization does not embrace microservices from an organization standpoint, no technologies will help you.

While building microservices may seem simple, there's actually a lot more to it. Microservice systems are distributed systems, and thus involve distributed computing laws. Failures are also inevitable. Each microservice should own its data, and you will typically need several persistence technologies.

To build microservice systems, there are a couple of topics to study further. First, to enable the promised agility, you are going to deploy the different microservices much more often. Therefore, continuous delivery is key. You also need to automate the release and deployment of your microservices as much as possible. Don't forget that your microservices are fungible, and immutable image delivery is a must-have feature in order to scale.

However, if our deployed microservices are immutable, how do we pass in configuration information and potentially reconfigure them? For instance, how do we change the log level, configure the database location and credentials, toggle features, and so on? Configuration is a very important part of distributed systems and is difficult with microservices. All systems are different and there is no silver bullet, but there is a wide range of solutions, from environment variables to Git repositories and dedicated configuration servers. Cloud platforms also provide configuration abilities. To mitigate this diversity, Vert.x is capable of retrieving configurations from almost anywhere.

Once you deploy and configure your microservices, you need to keep your system on track. Logging, metrics, and tracing are important concerns to keep in mind when designing and developing a microservice system. You have to retrieve the logged messages, the measures, and the traces from your microservices to aggregate them in a centralized way to enable correlation and visualization. While logging and monitoring are generally well understood, distributed tracing is often ignored. However, traces are priceless in microservices because they will help you identify bottlenecks, the affinity between microservices, and give you a good idea of the responsiveness of your system.

The Evolution of the Microservice Paradigm

Microservices are dynamic and always evolving. Recently, the *serverless* trend, also called *function as a service*, is gaining a lot of attraction. In this new paradigm, your unit of

deployment is a function. These functions receive and process messages. The serverless paradigm stresses the deployment, logging, and tracing facilities, but promotes a simple development model and improves the scalability of the system as you can instantiate as many instances of functions as you need to handle the load.

HTTP/2 is also making a remarkable entrance in the microservice world. It improves the performance and scalability of HTTP 1.1 and allows multiplexing several requests over a single TCP connection. It also offers bidirectional communication. gRPC is a remote procedure call (RPC) framework relying on HTTP/2 and provides high-performance, multilanguage, secure, bidirectional interactions. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking, and authentication. It is also applicable to embedded devices, mobile applications, and browsers. While RPC was initially considered harmful in microservice systems, it's still widely popular. gRPC addresses the issues encountered with traditional RPC mechanisms, such as blocking calls and partial failures. However, be aware that sharing a contract (interface) between the provider microservice and consumers may limit your agility. Vert.x provides HTTP/2 client and servers. In addition, it has the ability to create and consume gRPC services.

Vert.x Versatility

While this report has focused on reactive microservices, this is only a single facet of Vert.x. The richness of the Vert.x ecosystem lets you develop lots of different applications. Thanks to its execution model, your applications will be asynchronous and will embrace the *reactive system* mantra.

Modern web applications provide a *real-time*, *interactive* experience for users. The information is pushed to the browser and is displayed seamlessly. The Vert.x event bus can be used as the backbone to deliver such an experience. The browser connects to the event bus and receives messages, and can also send messages on and interact with the backend or with other browsers connected to the event bus.

The Internet of things (IoT) is a thrilling domain but also very heterogeneous. There are many protocols used by smart devices. Messages often have to be translated from one protocol to another. Vert.x provides clients for a large set of protocols to implement these translations, and its execution model can handle the high concurrency required to build IoT gateways.

These two examples illustrate the richness of the Vert.x ecosystem. Vert.x offers an infinite set of possibilities where you are in charge. You can shape your system using the programming language you prefer and the development model you like. Don't let a framework lead—you are in charge.