Lightweight Application Microservices

Michal Zerola, Risk-IT

Vert.x in one sentence

Vert.x is a **toolkit** to build **distributed** and **reactive** applications on top of the JVM using an **asynchronous non-blocking** development model.

Toolkit

- not an application server
- not a framework
- not a container
- doesn’t require an application server

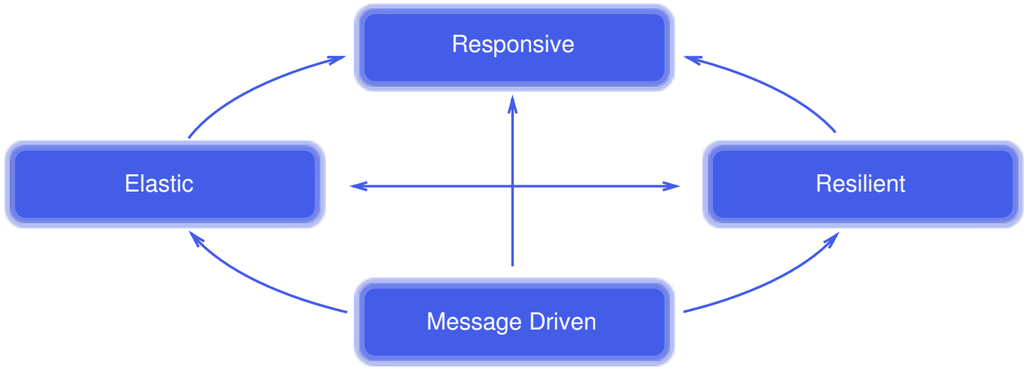
Just JAR files

Distributed systems

- defined decades ago
- instead of direct API-calls, loosely coupled components exchange messages
- you need to be prepared to fail
- you do not know when you get a response
- you do not even know if you receive a response

Distributed programming is hard

Reactive systems



- respond in an acceptable time
- scale up and down
- handle failures gracefully
- interact using async messages

Microservices

- Rebranding distributed applications
- (new) way how to develop distributed systems we were not able before
- Architectural style to develop an application as a suite of small services
- Individual services are going to evolve in their own pace
- They just need to sync on the interface level
- Lightweight interactions - loose coupling

Microservices do not give you a free lunch

- Orchestration
- Rapid provisioning and releasing of resources
- Automation of release and deployment
- Failures
- Discoverability issue
- Reliability issue
- Availability issue
- Logging, authentication, ...

Microservices offer you a good lunch

- If you know how to cook...
- They allow you to develop, test and deploy individually
- They let you improve your agility
- They improve your time to production

How?

Vert.x once again

Vert.x allows you to develop reactive microservices

- Based on multi-reactor pattern
- Everything in Vert.x is non-blocking and asynchronous
- Rich eco-system of additional extensions/libraries

Asynchronous non-blocking

- traditional imperative programming

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

- asynchronous non-blocking programming

```
compute(1, 2, res -> {
    // called with the result
});
```

Vert.x HelloWorld

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

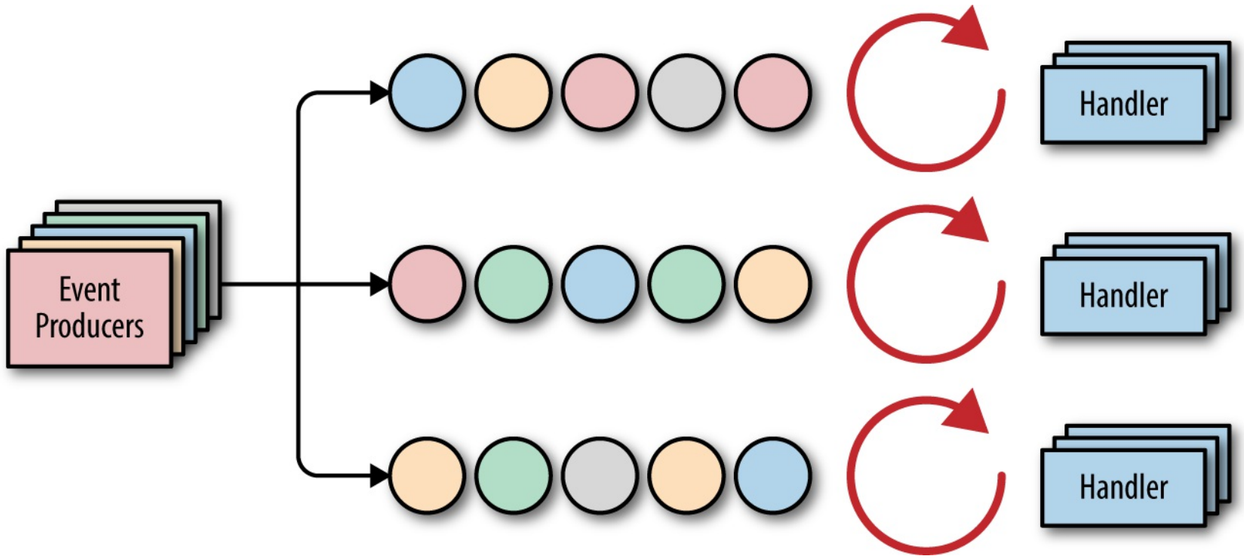
Vert.x object, control center, usually a single instance for the whole application

Vert.x is polyglot

Vert.x applications can be developed in

- Java
- Scala
- Groovy
- Kotlin
- Ruby (JRuby)
- JavaScript (Nashhorn)
- Ceylon

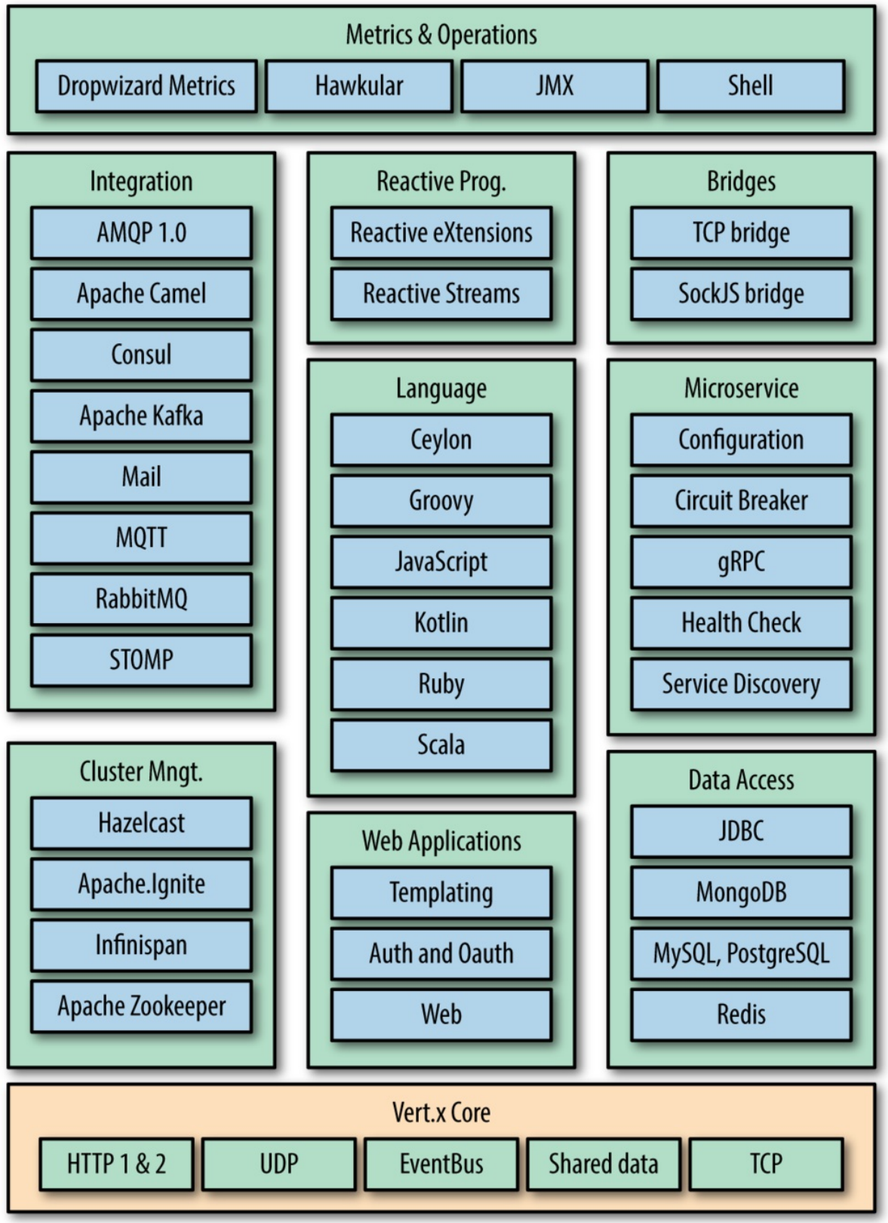
Multi-reactor pattern



Golden rule:

> *Never block an event loop*

Vert.x ecosystem



Verticle

- chunk of code that encapsulates a unit of business logic
- gets deployed and run by Vert.x application
- always executed by the same thread and never concurrently
- typically creates servers or clients, and registers a set of handlers

Verticle example

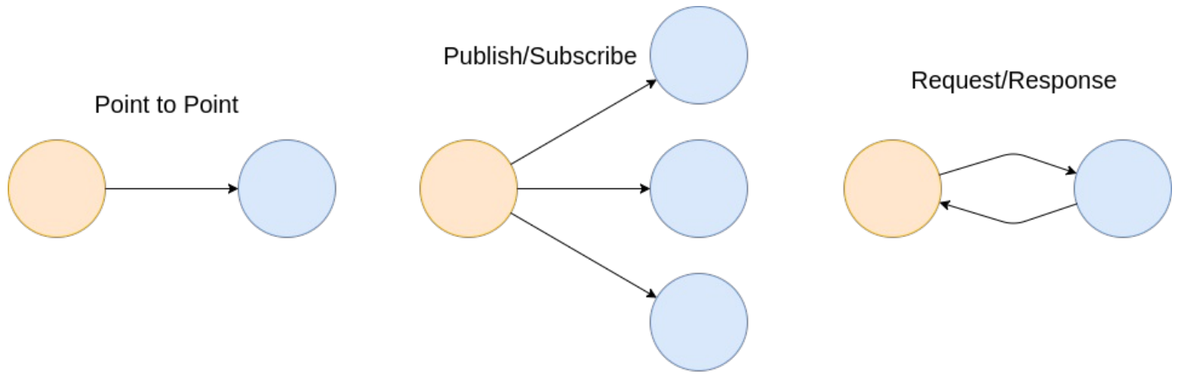
```
import io.vertx.core.AbstractVerticle;

public class Server extends AbstractVerticle {

  @Override
  public void start() {
    vertx.createHttpServer().requestHandler(req -> {
      req.response()
        .putHeader("content-type", "text/plain")
        .end("Hello from Vert.x!");
    }).listen(8080);
  }
}
```

Testing

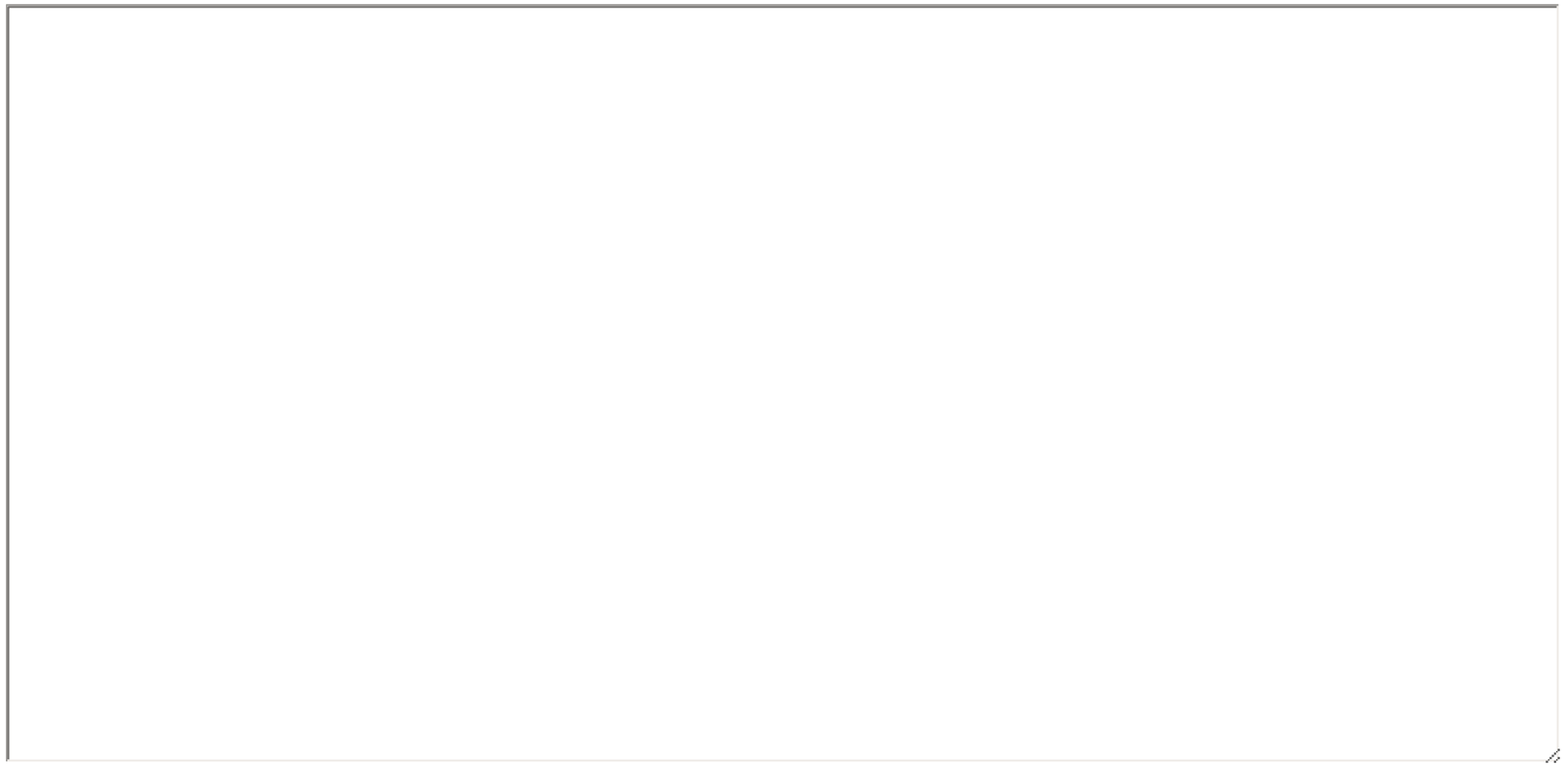
```
@RunWith(VertxUnitRunner.class)

public class ServerTest {
   private static Vertx vertx;

   @BeforeClass
   public static void setUp(TestContext context) {
      ServerTest.vertx = Vertx.vertx();
      ServerTest.vertx.deployVerticle(Server.class, context.asyncAssertSuccess());
   }

   @Test
   public void testExample(TestContext context) {
      final Async asyncClient = context.async();
      ServerTest.vertx.createHttpClient().getNow(8080, "localhost", "/", res -> {
         context.assertEquals(200, res.statusCode());
         asyncClient.complete();
      });
   }

   @After
   public static void tearDown(TestContext context) {
      ServerTest.vertx.close(context.asyncAssertSuccess());
   }
}
```

Event Bus

- the spine of the Vert.x applications
- nervous system of the Vert.x
- even different parts of Vert.x application (written in different languages) can interact
- messages are sent to an **address** and received in **handlers**

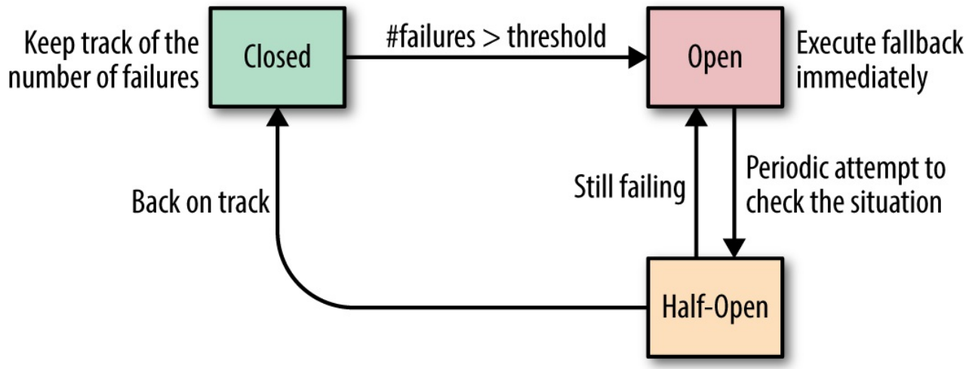


Event Bus - live example



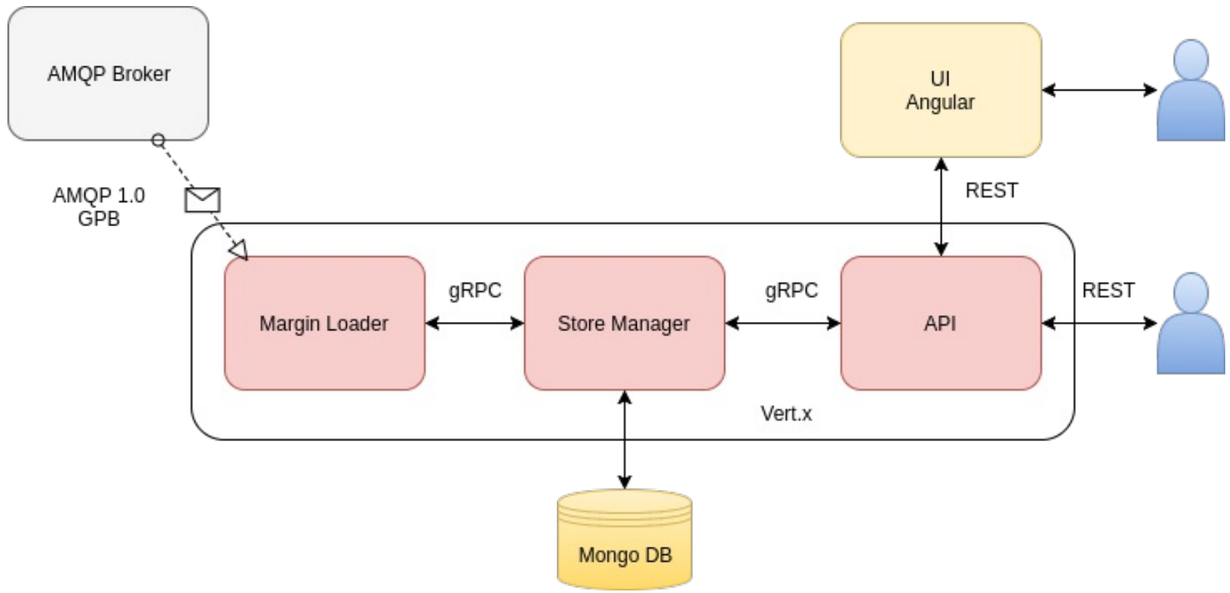
Circuit Breaker

- pattern used to deal with repetitive failures
- protects a microservice from calling a failing service again and again



DAVe - our experience with Vert.x

- one of the first cloud-native applications in DB
- fully reactive



Summary

- modern real-time applications require modern solutions
- microservices have never been easier

Vert.x toolkit gives you all the bits you need to build them

Thank you!