Vert.x

The reactive, event-driven, microservices framework for the JVM

Erwin de Gier, Software Architect Open Source, Sogeti Nederland B.V.

July 2016

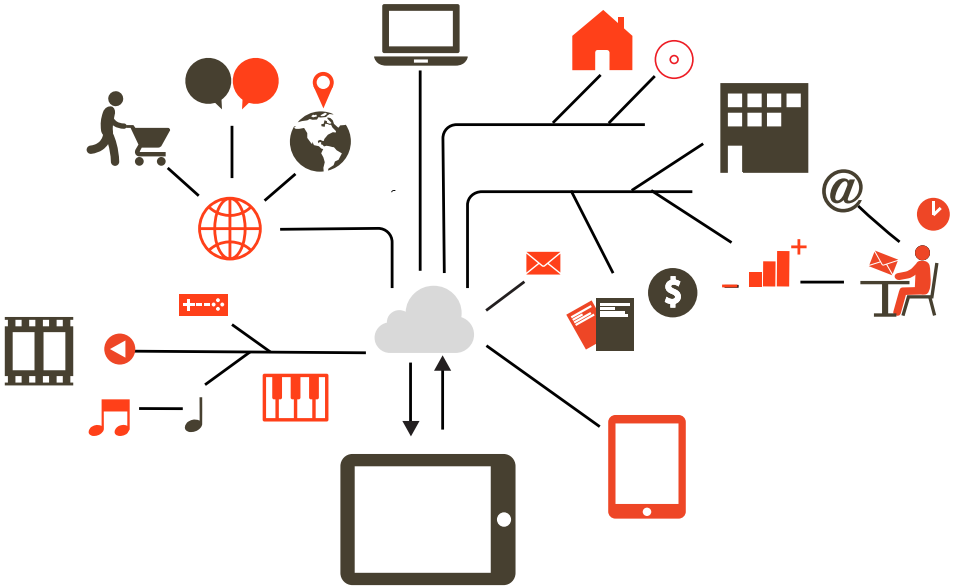
Changes in software development

Event-driven, microservices, scalable

According to Gartner (http://www.gartner.com/newsroom/id/3165317), the amount of devices connected to the internet in 2016 will be about 6.4 billion. Users of these devices expect real-time information everywhere and anytime. This amounts to an annual IP traffic of over 1 zettabyte (1 billion terabyte).

Handling huge amounts of (near) real-time data puts high demands on applications and their underlying architecture. To fulfill these demands, we need systems that are Responsive, Resilient, Elastic and Message driven. The reactive Manifesto (http://www.reactivemanifesto.org/) calls these kind of applications Reactive Systems. These systems are message driven, work by reacting to events and are usually realized as a microservice architecture. A combination of these concepts makes them scalable and responsive.

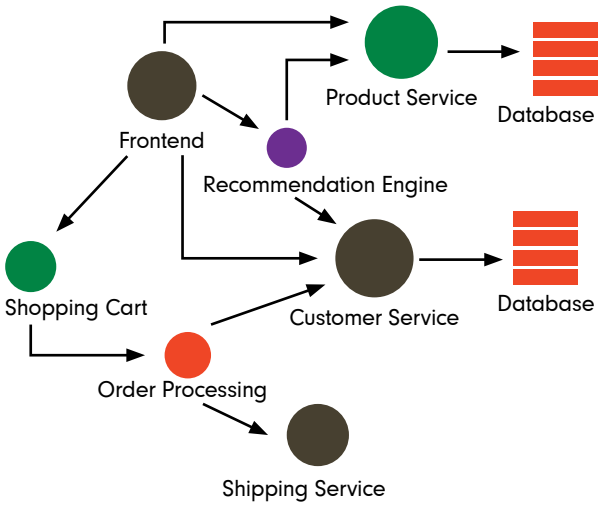
In this paper we will introduce the Vert.x toolkit as a solution for building applications that meet these changing demands by implementing the Reactive Systems approach.



Trends

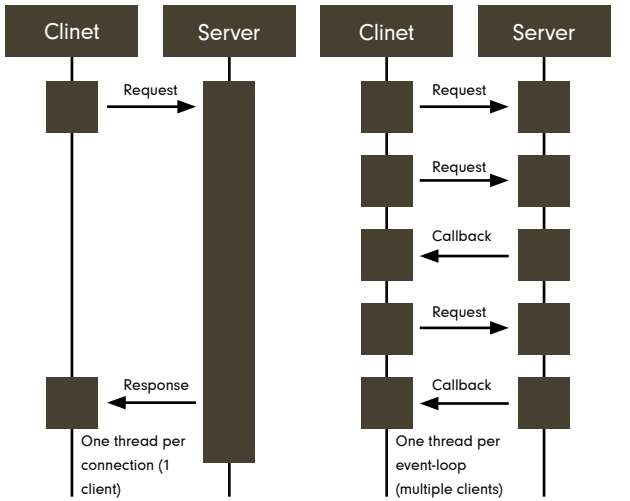
Microservices

In a microservice architecture, the application functionality is divided into independent lightweight components. Usually there is a standard interface between these compo-nents, for example REST, and the components are small, which makes the technology or programming language less important. The component independence makes it possible to individually deploy and scale the compo-nents, according to usage levels. Compo-nents can be reused by different applica-tions. Components can also fail individually, which makes the application more resilient.



Scalable

Traditionally, Operating System threads are employed to scale applications.
Thread-based scaling however is limited by the underlying hardware. Furthermore, thread based programming is difficult. An alternative solution is to use a non-blocking approach in which an event loop is used to listen for events and to call a function when such an event is detected. Results are propa-gated back through callback functions. This makes it possible to handle multiple requests simultaneously in a single thread. Between a request and a callback, the event loop is free to handle other events.



Event-driven

In an event driven architecture, components subscribe and react to messages which describe events. This as opposed to the constant updating of the current state. Events can be real life occurrences, such as the adding of a product to a shopping cart. Using events makes sense from a business perspective, as these domains are usually event based instead of state based. Using events promotes loose coupling between components.

A reactive, asynchronous approach

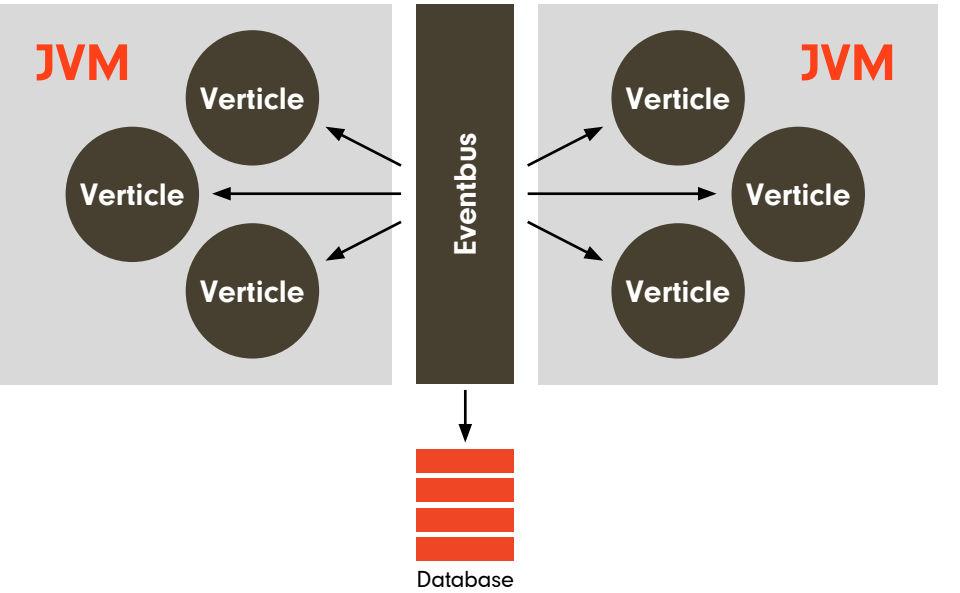
Realtime, scalable apps with Vert.x

Building asynchronous non-blocking applications requires a different programming model than synchronous applications. Popular existing Java frameworks are built around the thread per connection model. Webservice clients and database drivers tend to be synchronous in nature. This leaves an opportunity for new solutions which provide a full stack approach to asynchronous applications on the JVM.

One of the more mature solutions is Vert.x. Vert.x is an open source toolkit for creating reactive applications on the JVM. It also endorses a microservice approach to software architecture and is event-driven in nature. In the next chapters we will explore why Vert.x is a sound solu-tion for building modern scalable applications for the JVM and explain the various components which make it a powerful and mature frame-work.

Vert.x employs the one thread per event-loop model. It is important not to block this event-loop, because that would stop the consumption of new events. The complete stack is built using the reactive approach, using callbacks to return information, when necessary. This approach makes Vert.x applications highly performant and scalable.

A Vert.x application is split up into separate services, called Verticles. Each Verticle has its own purpose and implements a specific business functionality. Verticles operate individually and are individually deploy-able. This makes it easy to build microservice architecture with Vert.x. While Vert.x applications can be embedded in other application pack-ages, like a deployable war, the intended deployment model is as runnable jar files or verticles running in the Vert.x runtime.



Vert.x is event-driven, using an event bus to distribute the events to the different services. Services can subscribe to one or more channels on the event bus to receive events. The event bus can be distributed over multiple JVM's and even over multiple servers in the network.
Vert.x is polyglot, meaning that the team can choose the language they are most proficient with. It currently supports Java, Javascript, Groovy, Ruby and Ceylon. Support for more languages like Scala, Python and Typescript is in the works or added by the community.

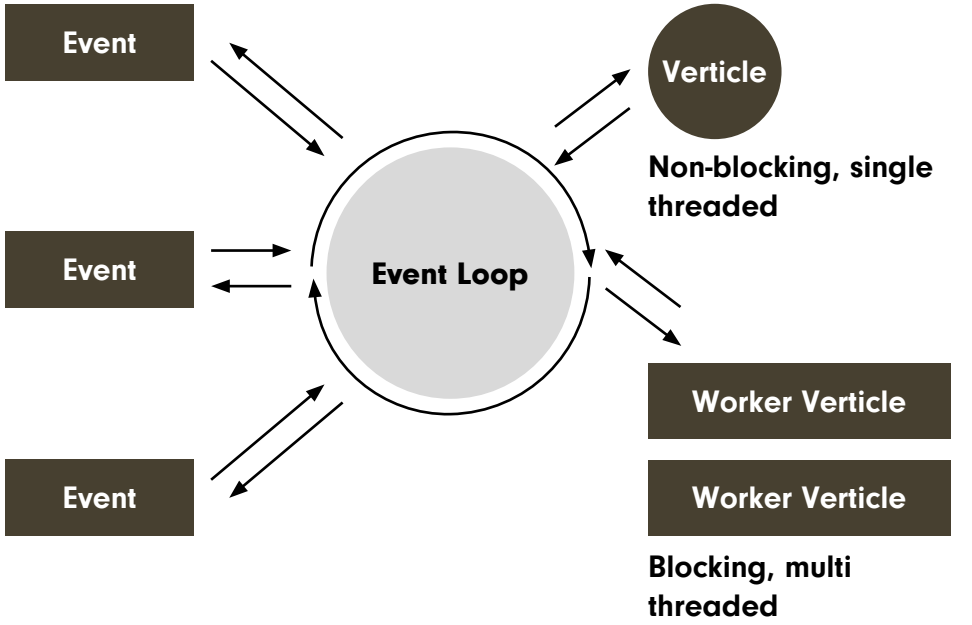
Event Loop

Making efficient use of threads

Similar to NodeJS, Vert.x employs a non-blocking model with a single threaded event-loop to handle work. The event loop is used to listen for events and to call a function when such an event is detected. The functions are called on Vert.x's units of work, called Verticles. Callbacks are registered to propa-gate back the results. Different from NodeJS, Vert.x can run multiple event-loops in parallel to make use of multi-core servers.

Non-blocking Verticles are used for all the work that can be processed asynchronously. For synchronous scenarios, like for instance legacy JDBC drivers, a Worker Verticle can be used. Worker Verticles run in their own thread and can thus perform blocking work.

The combination of single- and multi-threaded execution makes Vert.x highly flexible towards multiple programming scenarios. It allows the developer to extract maximum performance from the available server resources by employing vertical scaling in an effortless manner.



Vert.x programming model

The power of the event-bus

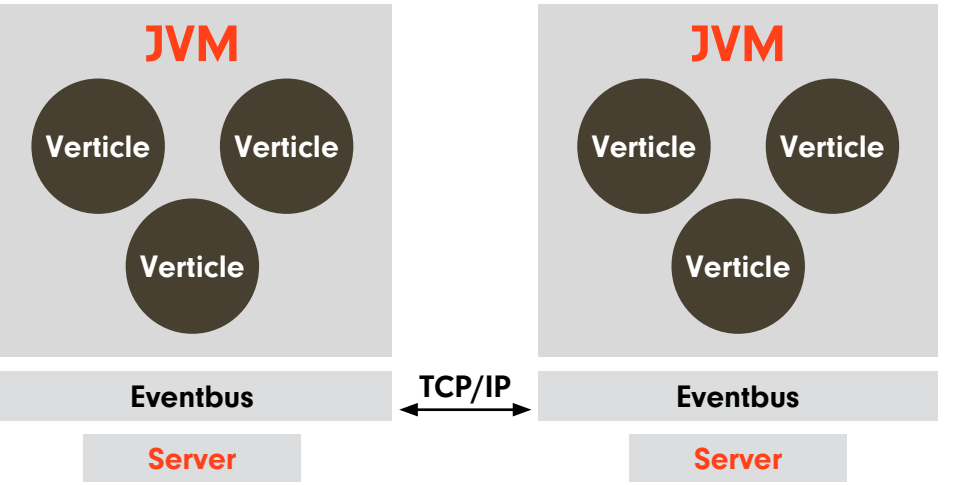
In its most simplest form, a Vert.x application can be realized as a single Verticle. Each Verticle has a single point of entry, the start() method.

A Verticle has access to the vertx object instance, which provides access to generic components like the event bus. Listing 1 shows a basic "hello world" example. In this example a JSON message is sent to the event bus every 1000 ms. From version 3, Vert.x is Java 8 only. The method providing this functionality is passed to the setPeriodic method as a lambda function, which enhances readability over the use of anonymous classes.

```
public class PublisherVerticle extends AbstractVerticle {

    @Override
    public void start() throws Exception {
        vertx.setPeriodic(1000, arg -> {
            vertx.eventBus().publish("event",
            new JsonObject().put("eventmessage", "hello"));
        });
    }
```

Listing 1



One of the most powerful features is the event bus. The event bus is the central communication hub between all Verticles. In contrast to a tradi-tional message queue, the event bus is not a separately deployable piece of middleware, but a core part of any Vert.x application. By starting a Vert.x application in cluster mode, it will automatically find other cluster enabled Vert.x applications on the network and connect to them. Together they will have one distributed event bus. This makes synchronizing state and messages over a potentially large distributed environment easy. The default underlying technology is Hazelcast, however this mechanism is pluggable, so other distributed map provid-ers can be used.

The distributed event-bus allows for easy horizontal scaling over multiple nodes without extra development effort.

Vert.x - web

HTTP server and URL mapping

The core Vert.x distribution provides a low level set of HTTP functionality. The following snippet shows how to start a HTTP server and return a text message as a response. Following the callback style programming model, a function is defined which gets called when a HTTP request comes in. Listing 2 shows how to start a HTTP server and return a text as a response. Following the callback style programming model, a function is defined and gets called when a HTTP request comes in. The server is configured to run on port 8080 by using the listen() function.

For building more sophisticated web applications, Vert.x–Web can be used. Vert.x–Web is an extension to Vert.x for building web applications. It allows us to build REST applications, template based web applications and even real-time web applications with server push functionality.

Vert.x-Web introduces the concept of a Router, which allows us to map a URL to a specific function. This function is implemented in a Handler. In the example in listing 3 all get requests to the path /hello-world are mapped to a function which returns the text "Hello World". The handler model allows multiple handlers for the same paths. This makes it possible to add more generic functionality, like authentication and session management, to a wide range of URLs. The example shows a Router with a StaticHandler, a CookieHandler and a SessionHandler configured.

Using the same technology as the distributed event bus, Vert.x provides a distributed Session Handler. This allows applications to share session data over multiple servers, avoiding the need of sticky sessions.

```
HttpServer server = vertx.createHttpServer();
server.requestHandler(request -> {
    HttpServerResponse response = request.response();
    response.end("Hello World");
});
server.listen(8080);
```

Listing 2

```
Router router = Router.router(vertx);
router.route().handler(CookieHandler.create());

SessionStore store = ClusteredSessionStore.create(vertx);
router.route().handler(SessionHandler.create(store));

router.route("/static/*").handler(StaticHandler.create());

router.get("/hello-world").handler(routingContext ->
routingContext.request().response().end("Hello World"));

HttpServer server = vertx.createHttpServer();
server.requestHandler(router::accept);
server.listen(8080);
```

Listing 3

Vert.x - web

REST API and realtime webapplications

The Router concept allows us to build REST APIs by mapping combina-tions of HTTP Methods and URLs to functions. In the code in Listing 4, a router is created for creating a REST API. The Router has methods which correspond to the HTTP Verbs. By making use of function references, the methods and URLs are mapped to the methods of the ProductService. This example also shows how to make use of path parameters. For deleting a product, the path parameter "id" is used. This parameter is available in the deleteProduct method.

```
Router apiRouter = Router.router(vertx);
apiRouter.get("/product").handler(productService::getProducts);
apiRouter.post("/product").handler(productService::createProduct);
apiRouter.delete("/product/:id").handler(productService::de-
leteProduct);
apiRouter.put("/product/:id").handler(productService::update-
Product);
```

Listing 4

The Vert.x framework allows us to build realtime web applications in several ways. First of all there is support for websockets. Websockets is a full duplex communication channel which allows realtime and asyn-chronous communication with browser clients. It also allows the server to push messages to the browser.

```
SockJSHandler sockJSHandler = SockJSHandler.create(vertx);
sockJSHandler.bridge(new BridgeOptions().addOutboundPermitted(new
PermittedOptions().setMatch(new JsonObject())));
router.route("/eventbus/*").handler(sockJSHandler);
```

Listing 5

It is possible to bridge the event bus to the browser clients. This makes it possible to receive some or all events in the browser clients. The clients can also send events back to the server. This allows the constuction of complete event based applications, from backend to frontend.

Data access

Asynchronous clients for various datastores

To develop a true asynchronous non-blocking application, it is important that all the components in the stack are non-blocking, including the calls to external systems like databases. Vert.x provides multiple asynchro-nous clients for accessing various datastores. These clients make data access easy by providing a typed interface and proxy, which hides the messaging over the event bus to the actual verticles handling the connection with the database.

The following example shows the asynchronous calls to the MongoCli-ent, an API for using MongoDB.

The MongoClient actually deploys a Verticle which listens to an eventbus address. Methods called on the MongoClient will be converted to Json messages which are sent to this same address. This is all hidden from the user of the MongoClient, by making use of a concept called service proxies. A service proxy makes it possible to define an interface for sending messages to a deployed Verticle. This makes it possible to define typed methods with Objects instead of using raw JsonObjects. Next to MongoDB, there are also clients for JDBC, Redis, SQL Common, MySQL and PostgreSQL. Other vendor's Java drivers can also be used.

```
private void insertMessage(Message<String> message){
      this.mongo.insert("message", new JsonObject().put("message",
      message.body()), result -> {
                  if (result.succeeded()) {
                        message.reply("save success");
                  } else {
                        message.fail(1, "save failed");
                  }
            });
      }
```

Listing 6

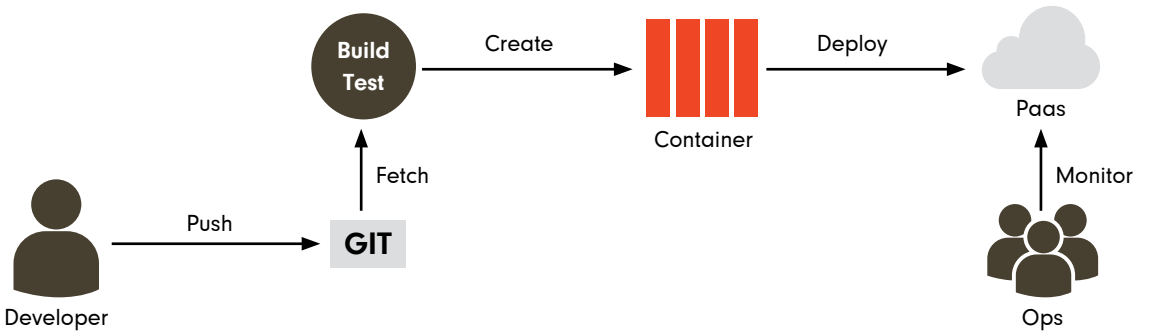
DevOps

Deployment

Vert.x applications usually consist of multiple services running on multiple machines. Vert.x shell allows you to log in to a running (distrib-uted) environment with an ssh client. This allows you to get a runtime overview of the state of the application and services. It also provides an option to deploy new or updated Verticles inside the running application, preventing downtime.

Deployment can be done from disk, but also from a Maven repository, which makes handling large amounts of artefacts and versions easy.

Docker images are available for easy deployment and management of the environ-ment. There is support for multiple cloud platforms, like RedHat Openshift, via docker images or cartridges.



Metrics

In any application, even more so for Micros-ervice applications, knowing the state of your application components is really important. Vert.x has support for Dropwizard style metrics, using the concept of a Measured interface. Various components like the event-bus, http server and vertx itself implement this interface. Using the MetricsService it is possi-ble to get Metrics snapshots of Measured components.

To represent data, Dropwizard representa-tions are used. For instance: gauge, counter, histogram, timer, etc.

Testing

Vert.x-Unit is the Unit testing extension of Vert.x. It specifically solves the problem of tests finishing before asynchronous functional-ity is completed. Vertx unit tests use the @RunWith(VertxUnitRunner. class) annotation. It introduces the concept of a TestContext, which helps testing asynchronous code. Listing 7 shows a typical example usage of the context to start an async test. On success, the async context is completed. On failure, the context.fail() method is used to fail the test.

```
@Test
public void testInsert(TestContext context) {
    Async async = context.async();
    vertx.eventBus().send("message.insert", "hello",
    result -> {
        if(result.succeeded()){
                async.complete();
        } else {
                context.fail();
        }
    });
}
```

Listing 7

Vert.x the frame-work of choice

For building microservices architectures on the JVM

The Java platform is hugely popular as a technology for hosting complex enterprise grade applications. Knowledge of the language and the platform is widespread, not only in terms of development but also regarding operations. The platform is also supported well through Oracle, third party suppliers and the open source community.

To advance the Java platform into the new age of software applications and the ever changing demands, a new approach is needed. Reactive programming and an asynchronous and event-driven architecture make it possible to meet the demands of rapid change, high performance and applications that can handle tens of thousands of connections.

Vert.x is a unique solution for the Java platform which has all the needed traits to fulfill these demands. The fact that ts runs on the Java platform makes it a well known technology for developers and opera-tions, which lowers the learning curve for adoption. Its event loop model combined with multi threading make it highly scalable.

Microservices	Scalable	Event-driven
The framework endorses the developers to divide the functionality over multiple services. Services are runtime deployable. Vert.x is polyglot, supporting a wide range of popular languages.	Vert.x is asynchronous and non-blocking. It pro-vides non-blocking APIs for HTTP, data access and file handling. The distrib-uted event-bus makes it easy to create distributed applications. Horizontal and vertical scaling are easily achieved.	Verticles communicate over the event-bus by use of messages. This makes Vert.x applications event-driven and allows for loose-coupling between services. The event-bus can be connected to non-Vert.x applications over tcp/ip.

Sogeti

Erwin de Gier
erwin.de.gier@sogeti.nl

Sogeti Nederland B.V.
Lange Dreef 17
Postbus 76 4130 EB Vianen
Tel +31 (0)88 660 66 00 Fax +31 (0)88 660 67 00
www.sogeti.nl

Sogeti is a leading provider of technology and software testing, specializing in Application, Infrastructure and Engineering Services. Sogeti offers cutting-edge solutions around Testing, Business Intelligence & Analytics, Mobile, Cloud and Cyber Security, combining world class methodologies and its global delivery model, Rightshore®. Sogeti brings together more than 20,000 professionals in 15 countries and has a strong local presence in over 100 locations in Europe, USA and India. Sogeti is a wholly-owned subsidiary of Cap Gemini S.A., listed on the Paris Stock Exchange.

For more information please visit **www.sogeti.com**

Thank you to Robert van Alphen and Steven Klein for reviewing this paper and to Livia Rickli for the layout.