A Hitchiker's Introduction to Vertx

Anatole Tresch, Principal Consultant
About me...

- Principal Consultant, Trivadis AG (Switzerland)
- Star Spec Lead JSR 354
- Open Source Enthusiast

- Twitter: @atsticks
- anatole@apache.org
- anatole.tresch@trivadis.com
Agenda

- Basics
- Networking
- Clustering
- Vertx Maven Plugin
- There is more...
Vertx?
Is that something to eat?
Tool-kit for reactive applications on the JVM
• Vert.x is not a container.
• You use the components you like.
• You can combine Vert.x with all the other libraries you like.
Reactive

Responsive
Elastic
Message Driven
Resilient

Source: http://www.reactivemanifesto.org/
Hitchhikers Introduction to Vertx

Core

- TCP/HTTP client and server, Datagram Sockets
- Event bus
- Shared data
- Periodic and delayed actions
- Verticles
- DNS client
- File system access
- HA and Clustering
Starting **VERT.X**

```xml
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-core</artifactId>
  <version>3.4.1</version>
</dependency>
```

```java
Vertx vertx = Vertx.vertx();
```

```java
Vertx vertx = Vertx.vertx(new VertxOptions().setWorkerPoolSize(40));
```
The Eventloop

```java
vertex.setPeriodic(1000, id -> {
    // This handler will get called every second
    System.out.println("timer fired!");
});
```

- Events are distributed by the event loop thread aka **Reactor**
- Vertx = **Multi-Reactor** (by default Cores*2)
- **Don’t block the event loop !!!**

**NOTE**

Even though a Vertx instance maintains multiple event loops, any particular handler will never be executed concurrently, and in most cases (with the exception of worker verticles) will always be called using the **exact same event loop.**
Running Blocking Code

```java
vertx.executeBlocking(future -> {
   // Call some blocking API that takes a significant amount of time to return
   String result = someAPI.blockingMethod("hello");
   future.complete(result);
}, res -> {
   System.out.println("The result is: " + res.result());
});
```
The Event Bus

• The nervous system of Vert.x.

• One single event bus instance for every Vert.x instance.

• Allows different parts of your application to communicate.

• Can be bridged to support client side JavaScript.

• A distributed peer-to-peer messaging spanning multiple nodes and browsers.

• Publish/subscribe, point to point, and request-response messaging.
Registering Handlers to the Event Bus

EventBus eb = vertx.eventBus();

eb.consumer("news.uk.sport", message -> {
    System.out.println("I have received a message: " + message.body());
});

EventBus eb = vertx.eventBus();

MessageConsumer<String> consumer = eb.consumer("news.uk.sport");
consumer.handler(message -> {
    System.out.println("I have received a message: " + message.body());
});
Publishing Events on the Event Bus

```java
eventBus.publish("news.uk.sport", "Yay! Someone kicked a ball");
```

That message will then be delivered to all handlers registered against the address news.uk.sport.

```java
eventBus.send("news.uk.sport", "Yay! Someone kicked a ball");
```

Sending a message will result in only one handler registered at the address receiving the message. This is the point to point messaging pattern. The handler is chosen in a non-strict round-robin fashion.

```java
DeliveryOptions options = new DeliveryOptions();
options.addHeader("some-header", "some-value");
eventBus.send("news.uk.sport", "Yay! Someone kicked a ball", options);
```
Acknowledging Messages

The receiver:

```java
MessageConsumer<String> consumer = eventBus.consumer("news.uk.sport");
consumer.handler(message -> {
    System.out.println("I have received a message: " + message.body());
    message.reply("how interesting!");
});
```

The sender:

```java
eventBus.send("news.uk.sport", "Yay! Someone kicked a ball across a patch of grass")
if (ar.succeeded()) {
    System.out.println("Received reply: " + ar.result().body());
}
```
Delivery Options

Sending with timeouts

When sending a message with a reply handler you can specify a timeout in the `DeliveryOptions`.

If a reply is not received within that time, the reply handler will be called with a failure.

The default timeout is 30 seconds.

Send Failures

Message sends can fail for other reasons, including:

- There are no handlers available to send the message to
- The recipient has explicitly failed the message using `fail`

In all cases the reply handler will be called with the specific failure.

Message Codecs

You can send any object you like across the event bus if you define and register a `message codec` for it.
**JSON Support**

- Vertx comes with full JSON OOTB (based on Jackson Library)
- JSON can is by default support over the event bus
- `JsonObject.encode() / JsonArray.encode() → String`

```java
JsonObject object = new JsonObject();
object.put("foo", "bar").put("num", 123).put("mybool", true);

JsonArray array = new JsonArray();
array.add("foo").add(123).add(false);

request.bodyHandler(buf -> {
    JsonObject jsonObject = buf.toJsonObject();
    User javaObject = jsonObject.mapTo(User.class);
});
```
High Availability

Verticles can be deployed with High Availability enabled:

```
vertx run my-verticle.js -ha
```
Buffers

Most data is shuffled around inside Vert.x using buffers.

A buffer is a sequence of zero or more bytes that can read from or written to and which expands automatically as necessary to accommodate any bytes written to it. You can perhaps think of a buffer as smart byte array.

Create a new empty buffer:

```java
Buffer buff = Buffer.buffer();
```

Create a buffer from a String. The String will be encoded in the buffer using UTF-8.

```java
Buffer buff = Buffer.buffer("some string");
```

Create a buffer from a String: The String will be encoded using the specified encoding, e.g:

```java
Buffer buff = Buffer.buffer("some string", "UTF-16");
```

Create a buffer from a byte[]

```java
byte[] bytes = new byte[] {1, 3, 5};
Buffer buff = Buffer.buffer(bytes);
```
Verticles

- Simple, scalable, actor-like deployment and concurrency model
- Not a strict actor-model implementation, but similar
- Entirely optional

```java
class MyVerticle extends AbstractVerticle {

    public void start() {
        // Do something
    }

    public void stop(Future<Void> stopFuture) {
        obj.doSomethingThatTakesTime(res -> {
            if (res.succeeded()) {
                stopFuture.complete();
            } else {
                stopFuture.fail();
            }
        });
    }
}
```
Verticle Types

**Standard Verticles**
These are the most common and useful type - they are always executed using an event loop thread.

**Worker Verticles**
These run using a thread from the worker pool. An instance is never executed concurrently by more than one thread.

**Multi-threaded worker verticles**
These run using a thread from the worker pool. An instance can be executed concurrently by more than one thread.

```java
DeploymentOptions options = new DeploymentOptions().setWorker(true);
vertx.deployVerticle("com.mycompany.MyOrderProcessorVerticle", options);
```
Deploying Verticles

NOTE Deploying Verticle instances is Java only.

```
Verticle myVerticle = new MyVerticle();
vertx.deployVerticle(myVerticle);
```

Or you use references for VerticleFactories for polyglot support,

```
vertx.deployVerticle("com.mycompany.MyOrderProcessorVerticle");

// Deploy a JavaScript verticle
vertx.deployVerticle("verticles/myverticle.js");

// Deploy a Ruby verticle verticle
vertx.deployVerticle("verticles/my_verticle.rb");
```
Timers and Delays

One shot timers:

```java
long timerID = vertx.setTimer(1000, id -> {
    System.out.println("And one second later this is printed");
});
```

```java
System.out.println("First this is printed");
```

Periodic timers:

```java
long timerID = vertx.setPeriodic(1000, id -> {
    System.out.println("And every second this is printed");
});
```

```java
System.out.println("First this is printed");
```
Shared Data

- **Local shared maps**

```java
SharedData sd = vertx.sharedData();

LocalMap<String, String> map1 = sd.getLocalMap("mymap1");
```

- **Cluster-wide asynchronous maps**

```java
sd.<String, String>getClusterWideMap("mymap", res -> {
    if (res.succeeded()) {
        AsyncMap<String, String> map = res.result();
    } else {
        // Something went wrong!
    }
});
```

- **Cluster-wide locks & counters (not shown)**
Networking
TCP Server

NetServer server = vertx.createNetServer();

NetServerOptions options = new NetServerOptions().setPort(4321);
NetServer server = vertx.createNetServer(options);

NetServer server = vertx.createNetServer();
server.listen();

server.listen(1234, "localhost", res -> {
    if (res.succeeded()) {
        System.out.println("Server is now listening!");
    } else {
        System.out.println("Failed to bind!");
    }
});
TCP Server Reading/Writing

```java
NetServer server = vertx.createNetServer();
server.connectHandler(socket -> {
    socket.handler(buffer -> {
        System.out.println("I received some bytes: " + buffer.length());
    });
});

Buffer buffer = Buffer.buffer().appendFloat(12.34f).appendInt(123);
socket.write(buffer);

// Write a string in UTF-8 encoding
socket.write("some data");

// Write a string using the specified encoding
socket.write("some data", "UTF-16");
```
TCP Client

```java
NetClientOptions options = new NetClientOptions().setConnectTimeout(10000);
NetClient client = vertx.createNetClient(options);
client.connect(4321, "localhost", res -> {
    if (res.succeeded()) {
        System.out.println("Connected!");
        NetSocket socket = res.result();
    } else {
        System.out.println("Failed to connect: " + res.cause().getMessage());
    }
});
```
HTTP Server

```
HttpServer server = vertx.createHttpServer();
server.listen(8080, "myhost.com", res -> {
    if (res.succeeded()) {
        System.out.println("Server is now listening!");
    } else {
        System.out.println("Failed to bind!");
    }
});

vertx.createHttpServer().requestHandler(request -> {
    request.response().end("Hello world");
}).listen(8080);
```
HTTP Client

You create an `HttpClient` instance with default options as follows:

```java
HttpClient client = vertx.createHttpClient();
```

If you want to configure options for the client, you create it as follows:

```java
HttpClientOptions options = new HttpClientOptions().setKeepAlive(false);
HttpClient client = vertx.createHttpClient(options);
```
Vertx Web

Provides more flexible support for writing web services.

```xml
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web</artifactId>
  <version>3.4.1</version>
</dependency>
```
Vertx Web – Basic Routing

HttpServer server = vertx.createHttpServer();

Router router = Router.router(vertx);

router.route().handler(routingContext -> {

    // This handler will be called for every request
    HttpResponse response = routingContext.response();
    response.putHeader("content-type", "text/plain");

    // Write to the response and end it
    response.end("Hello World from Vert.x-Web!");

    });

server.requestHandler(router::accept).listen(8080);
public class PersonServer extends AbstractVerticle {

    private static final String APPLICATION_JSON = "application/json";

    private HttpServer server;

    @Override
    public void start() throws Exception {
        super.start();
        server = vertx.createHttpServer();
        Router router = Router.route(vertx);
        Router restAPI = Router.route(vertx);
        restAPI.get().handler(this::list);
        restAPI.get("/:id").handler(this::get);
        restAPI.post().handler(this::persist);
        restAPI.delete("/:id").handler(this::delete);
        router.mountSubRouter("/resources/persons", restAPI);
        router.route("/*").handler(StaticHandler.create("webapp"));

        server.requestHandler(router::accept).listen(8080);
    }
}
Vertx Web – Rest Endpoint

```java
private void persist(RoutingContext rc) {
    rc.request().bodyHandler(buf -> {
        vertx.eventBus().send(PersonRepository.STORE, buf.toString(),
            h -> {
                rc.response()
                    .setStatusCode(HttpResponseStatus.CREATED.code())
                    .end();
            });
    });
}

private void get(RoutingContext rc) {
    vertx.eventBus().send(PersonRepository.GET, rc.get("id"), h -> {
        rc.response()
            .putHeader(HttpHeaderNames.CONTENT_TYPE, APPLICATION_JSON)
            .end(String.valueOf(h.result().body()));
    });
}
```
Vertx CircuitBreaker

```java
CircuitBreaker breaker = CircuitBreaker.create("my-circuit-breaker", vertx,
);

breaker.executeWithFallback(
    future -> {
        vertx.createHttpClient().getNow("8080", "localhost", "/", response -> {
            if (response.statusCode() != 200) {
                future.fail("HTTP error");
            } else {
                response
                    .exceptionHandler(future::fail)
                    .bodyHandler(buffer -> {
                        future.complete(buffer.toString());
                    });
            }
        });
    }, v -> {
        // Executed when the circuit is opened
        return "Hello";
    });
    .setHandler(ar -> {
        // Do something with the result
    });
```

```xml
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-circuit-breaker</artifactId>
    <version>3.4.1</version>
</dependency>
```
Vertx ServiceDiscovery

A service provider can:

- publish a service record
- un-publish a published record
- update the status of a published service (down, out of service...)

A service consumer can:

- lookup services
- bind to a selected service (it gets a ServiceReference) and use it
- release the service once the consumer is done with it
- listen for arrival, departure and modification of services.

<dependency>
<groupId>io.vertx</groupId>
<artifactId>vertx-service-discovery</artifactId>
<version>3.4.1</version>
</dependency>
Vertx ServiceDiscovery – Provider Sample

```java
// Record creation from a type
record = HttpEndpoint.createRecord("some-rest-api", "localhost", 8080, "/api");
discovery.publish(record, ar -> {
    if (ar.succeeded()) {
        // publication succeeded
        Record publishedRecord = ar.result();
    } else {
        // publication failed
    }
});
```
Vertx ServiceDiscovery – Consumer Sample

```java
ServiceReference reference1 = discovery.getReference(record1);
ServiceReference reference2 = discovery.getReference(record2);

// Then, gets the service object, the returned type depends on the service type:
// For http endpoint:
HttpClient client = reference1.getAs(HttpClient.class);
// For message source
MessageConsumer consumer = reference2.getAs(MessageConsumer.class);

// When done with the service
reference1.release();
reference2.release();
```
Clustering

VertxOptions options = new VertxOptions();
Vertx.clusteredVertx(options, res -> {
    if (res.succeeded()) {
        Vertx vertx = res.result();
        EventBus eventBus = vertx.eventBus();
        System.out.println("We now have a clustered event bus: "+ eventBus);
    } else {
        System.out.println("Failed: "+ res.cause());
    }
});

You should also make sure you have a ClusterManager implementation on your classpath, for example the default HazelcastClusterManager.

Clustering on the command line

You can run Vert.x clustered on the command line with

vertx run my-verticle.js -cluster
High Availability

Verticles can be deployed with High Availability (HA) enabled. In that context, when a verticle is deployed on a vert.x instance that dies abruptly, the verticle is redeployed on another vert.x instance from the cluster.

To run a verticle with the high availability enabled, just append the `-ha` switch:

```
vertx run my-verticle.js -ha
```

When enabling high availability, no need to add `-cluster`.
Vertx Maven Plugin
Vertx Maven Plugin

```xml
<properties>
  <java.main.class>io.vertx.core.Launcher</java.main.class>
  <vertx.vertx>gh.atsticks.samples.k8s.person.PersonApp</vertx.vertx>
</properties>

<!-- vert.x maven plugin to generate the fat-jar -->
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>vertx-maven-plugin</artifactId>
  <version>${vertx-maven-plugin.version}</version>
  <executions>
    <execution>
      <id>vmp</id>
      <phase>package</phase>
      <goals>
        <goal>initialize</goal>
        <goal>package</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

- `vertx:debug`
- `vertx:help`
- `vertx:initialize`
- `vertx:package`
- `vertx:run`
- `vertx:setup`
- `vertx:start`
- `vertx:stop`
I want to see a demo.
Please give me more.
Not discussed...

- Vertx CLI
- MQTT Client/Server
- Configuration
- Metrics Support
- Health-Checks
- Shell
- Test Support
- Service Proxies
- Openshift / Docker Tooling
- JDBC
- NoSql Support
- Advanced:
  - Code Generation, Language Bindings, etc
Demo Time ...
Recap
Recap

- Vertx is a very flexible and modular toolkit
- It is rather lightweight (ca. 6 MB for something useful)
- It is polyglot, supporting a big range of languages
- It is amazing fast
- It supports all requirements of the reactive manifesto
- It sometimes can be a beast
- It requires Java 8+
- It is fun to work with
A Hitchhiker's Introduction to Vertx

Anatole Tresch
Principal Consultant

Tel. +41 58 459 53 93
anatole.tresch@trivadis.com

Vertx: http://vertx.io
Vertx Maven Plugin: https://vmp.fabric8.io/
Reactive Manifesto: http://www.reactivemanifesto.org/