# Cheat Sheet: Eclipse Vert.x

## Table of Contents


1. **Introduction** 1
   - 1.A Reactive Systems 1
   - 1.B Programing with Vert.x 2
2. **Create a new Vert.x application** 2
   - 2.A Create a Vert.x application with Apache Maven 3
   - 2.B Create a Vert.x application with Gradle 3
   - 2.C Create a Vert.x application with the Vert.x CLI 3
3. **Verticles** 4
   - 3.A Creating a verticle 4
   - 3.B Deploying verticles programmatically 5
   - 3.C Configuring verticles 5
4. **Timer and Periodic Tasks** 6
5. **HTTP** 6
   - 5.A HTTP Server 6
   - 5.B HTTP Client 7
6. **EventBus** 8
   - 6.A Point to Point 8
   - 6.B Publish/Subscribe 8
   - 6.C Request-Response 9
   - 6.D Delivery Options 10
7. **Executing blocking code** 10
   - 7.A Execute blocking construct 10
   - 7.B Worker verticles 11
8. **Executing Vert.X applications** 12
9. **Further learning** 13
10. **About the author** 13


## 1. Introduction

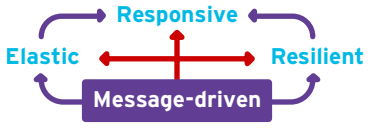
Eclipse Vert.x is a toolkit to build reactive and distributed systems. Application using Vert.x are fast, responsive, resilient and elastic. Vert.x is incredibly flexible - whether it's network utilities, modern web applications, microservices, or a full blown back-end message-bus application, Vert.x is a great fit.

Vert.x is event driven and non blocking. This means your app can handle a lot of concurrency using a small number of threads. Vert.x applications run on top of the Java Virtual Machine but can be implemented in many language such as Java, JavaScript, Groovy, Ruby and Ceylon. Vert.x provides idiomatic APIs for every supported language.
Vert.x is not a restrictive framework or container, it gives you useful bricks and let you create your app the way you want to.

### 1.A Reactive systems

Applications developed with Vert.x are reactive. The Reactive Manifesto (http://reactivemanifesto.org) defines a reactive application as having four key properties:

- Use asynchronous message-passing
- Elastic
- Resilient
- Responsive




Components forming your Vert.x application interact using asynchronous message passing regardless of whether these components are co-located or distributed. Each component reacts to the received message by using an asynchronous non-blocking development model.

This allows the application to more effectively share resources by doing work only in response to outside messages.

Vert.x applications are also elastic, meaning they react to increasing load well, because the architecture highly concurrent and distributed.

Vert.x applications are resilient, treating failure as a first-class citizen --- it can face failures, isolate them, and implement recovery strategies easily.

The final property, responsive, means the application is real-time and engaging. It continues to provide its service in a timely-fashion even when the system is facing failures or peak of demand.

## 1.B Programming with Vert.x

Vert.x applications are largely event driven, this means that when things happen in Vert.x that you are interested in, Vert.x notifies you by sending events. You handle these events by providing handlers to the Vert.x APIs. For example, to receive an HTTP request event:

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

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, you will usually provide a Handler to receive events some time later.

That means you can handle a highly concurrent work load using a small number of threads. In most cases Vert.x calls your handlers using a thread called an event loop. Vert.x APIs are non blocking and won't block the event loop, but that's not much help if you block the event loop yourself in a handler, hence the golden rule: **Don't block the event loop.** Because nothing blocks, an event loop can deliver a huge quantity of events in a short amount of time. This is called the Reactor pattern.

In a standard reactor implementation there is a single event loop thread which runs around in a loop delivering all events to all handlers as they arrive. The trouble with a single thread is it can only run on a single core at any one time. Vert.x works differently here. Instead of a single event loop, each Vertx instance maintains several event loops. This pattern is called Multi-Reactor Pattern.

# 2. Create a new Vert.x application

There are many ways to create a Vert.x application, giving you a great freedom to use your favorite tool. As Vert.x is a toolkit, it can also be embedded in your Spring or JavaEE application too. In this cheat sheet, we will demonstrate three ways to create projects with Apache Maven, Gradle, and the vertx CLI.

## 2.A Create a Vert.x application with Apache Maven

| Command | Description |
|---|---|
| `# Linux and MacOS`<br>`git clone https://github.com/vert-x3/vertx-maven-starter.git PROJECT_NAME`<br>`cd PROJECT_NAME`<br>`./redeploy.sh`<br>`# Windows`<br>`git clone https://github.com/vert-x3/vertx-maven-starter.git PROJECT_NAME`<br>`cd PROJECT_NAME`<br>`redeploy.bat` | **Generate and Run**<br>It generates the project structure and start the application in redeploy mode: your changes recompile and restart the application. The started application is accessible from http://localhost:8080 |
| `mvn package` | **Package**<br>An executable fat jar is created in the target directory. |
| Add the dependency in the `pom.xml` file | **Dependency management**<br>Add the dependency in the pom.xml file |
| Import the project as a Maven project in your favorite IDE | **IDE support** |

## 2.B Create a Vert.x application with Gradle

| | |
|---|---|
| `git clone https://github.com/vert-x3/vertx-gradle-starter.git PROJECT_NAME`<br>`cd PROJECT_NAME`<br>`./gradlew run` | **Generate and Run**<br>It generates the project structure and start the application in redeploy mode: your changes recompile and restart the application. The started application is accessible from http://localhost:8080 |
| `./gradlew shadowJar` | **Package**<br>An executable fat jar is created in the build/libs directory. |
| Add the dependency in the `gradle.project` file | **Dependency management**<br>You need to restart `gradlew` to reflect the changes |
| Import the project as a Gradle project in your favorite IDE | **IDE support** |

## 2.C Create a Vert.x application with the Vert.x CLI

| | |
|---|---|
| `git clone https://github.com/vert-x3/vertx-cli-starter.git PROJECT_NAME`<br>`cd PROJECT_NAME`<br>`./vertx.sh run src/io/vertx/starter/MainVerticle.java --redeploy="src/**/*" --launcher-class="io.vertx.core.Launcher"`<br>Use `vertx.bat` on Windows | **Generate and Run**<br>It generates the project structure and start the application in redeploy mode: your changes recompile and restart the application. The started application is accessible from http://localhost:8080 |

| Command | Description |
|---|---|
| Edit the `vertx/vertx-stack.json` to add, remove or update your dependency. Then, run: `./vertx.sh resolve` | Dependency management<br>You need to restart the application to reflect the changes. |
| Import the project as a Java project. Add the `vertx/lib` directory in your classpath. | IDE support |

# 3. Verticles

Vert.x 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 would typically be composed of many verticle instances running in the same Vert.x instance at the same time. Verticle instances communicate with each other by sending messages on the event bus.

Default verticles are executed on the Vert.x event loop and **must never block.** Vert.x ensures that each verticle is always executed by the same thread (never concurrently, hence avoiding synchronization constructs).

## 3.A Creating a verticle

All the projects created with the given instructions have created a main verticle, implemented in Java, that starts a HTTP server. Verticles can be implemented in any supported language, and to add support for another language, add the indicated dependency to your project:

| Language | Dependency to add to your project | Verticle |
|---|---|---|
| Java | N/A (default, provided) | `public class MyHttpServer extends AbstractVerticle { @Override public void start() throws Exception { vertx.createHttpServer() .requestHandler(req -> req. response() .end("Hello from Java")) .listen(8080); } }` |
| Groovy | `io.vertx::vertx-lang-groovy` | `vertx.createHttpServer() .requestHandler({ req -> req.response().end("Hello from Groovy") }) .listen(8080)` |
| JavaScript | `io.vertx::vertx-lang-js` | `vertx.createHttpServer() .requestHandler(function (req) { req.response().end("Hello from JavaScript") }) .listen(8080);` |

Verticles can also have an optional stop method that is called when the verticle is undeployed. The stop and corresponding start methods can also take a Future object as parameter to start and stop asynchronously.

## 3.B Deploying verticles programmatically

Verticles can be deployed programmatically from your code. This pattern is often used by a main verticle deploying sub-verticles. Verticles deployed in this manner are identified using the verticle file name. For a Java verticle, you can also use the fully qualified class name (FQCN).

| Language | Verticle |
|---|---|
| Java | `public class MainVerticle extends AbstractVerticle {`<br>`@Override`<br>`public void start() {`<br>`vertx.deployVerticle(MyVerticle.class.getName());`<br>`vertx.deployVerticle("verticles/my-verticle.groovy");`<br>`vertx.deployVerticle("verticles/my-verticle.js");`<br>`}`<br>`}` |
| Groovy | `vertx.deployVerticle("verticles/MyVerticle.java"`<br>`vertx.deployVerticle("verticles/my-verticle.groovy")`<br>`vertx.deployVerticle("verticles/my-verticle.js")` |
| JavaScript | `vertx.deployVerticle("verticles/MyVerticle.java");`<br>`vertx.deployVerticle("verticles/my-verticle.groovy");`<br>`vertx.deployVerticle("verticles/my-verticle.js");` |

## 3.C Configuring verticles

When deploying a verticle, you can pass deployment options to configure things such as the number of instances, or high-availability mode. You can also provide the verticle configuration. (Vert.x uses JSON as configuration format.)

| Language | Verticle | Deployment |
|---|---|---|
| Java | `System.out.`<br>`println(config().`<br>`getString("key"));` | `vertx.deployVerticle(`<br>`MyVerticle.class.getName(),`<br>`new DeploymentOptions()`<br>`.setConfig(new JsonObject()`<br>`.put("key", "value")));` |
| Groovy | `println(vertx.`<br>`getOrCreateContext().`<br>`config()['key'])` | `vertx.deployVerticle(`<br>`"verticles/my-verticle.groovy",`<br>`['config': ['key': 'value']]`<br>`)` |
| JavaScript | `console.log(vertx.`<br>`getOrCreateContext()`<br>`.config()["key"]);` | `vertx.deployVerticle(`<br>`"verticles/my-verticle.js",`<br>`{"config": {"key": "value"}});` |

# 4. Timer and periodic tasks

Vert.x lets you execute delayed tasks and periodic tasks.

| Language | Verticle | Deployment |
|---|---|---|
| Java | `vertx.setTimer(1000, l -> { System.out.println("Run later"); });` | `vertx.deployVerticle( MyVerticle.class.getName(), new DeploymentOptions() .setConfig(new JsonObject() .put("key", "value")));` |
| Groovy | `vertx.setTimer(1000, { l -> println("Run later")})` | `def taskId = vertx.setPeriodic(2000, { l -> println("Run periodically")}) //... vertx.cancelTimer(taskId)` |
| JavaScript | `vertx.setTimer(1000, function (l) { console.log("Run later"); });` | `var taskId = vertx.setPeriodic(2000, function (l) { console.log("Run periodically") }); //... vertx.cancelTimer(taskId);` |

# 5. HTTP

This section contains examples to creates Vert.x HTTP servers and client.

## 5.A HTTP Server

| Language | Verticle |
|---|---|
| Java | `public class MyHttpServer extends AbstractVerticle { @Override public void start() throws Exception { vertx.createHttpServer() .requestHandler(req -> req.response() .putHeader("content-type", "text/html") .end("<h1>Hello from Java</h1>")) .listen(8080, ar -> { if (ar.succeeded()) { System.out.println("Server started on port " + ar.result(). actualPort()); } else { System.out.println("Unable to start server " + ar.cause(). getMessage()); } }); } }` |

| Language | Verticle |
|---|---|
| Groovy | `vertx.createHttpServer()`<br>`  .requestHandler({ req ->`<br>`  req.response()`<br>`  .putHeader("content-type", "text/html")`<br>`  .end("<h1>Hello from Groovy</h1>")`<br>`  })`<br>`  .listen(8080, { ar ->`<br>`  if (ar.succeeded()) {`<br>`  println("Server started on port " + ar.result().actualPort());`<br>`  } else {`<br>`  println("Unable to start server " + ar.cause().getMessage());`<br>`  }`<br>`})` |
| JavaScript | `vertx.createHttpServer()`<br>`  .requestHandler(function (req) {`<br>`  req.response().putHeader("content-type", "text/html")`<br>`  .end("<h1>Hello from JavaScript</h1>")`<br>`  })`<br>`  .listen(8080, function(res, err) {`<br>`  if (err) {`<br>`  console.log("Unable to start the HTTP server: " + err.`<br>`getMessage());`<br>`  } else {`<br>`  console.log("Server started on port " + res.actualPort());`<br>`  }`<br>`  });` |

## 5.B HTTP Client

| | |
|---|---|
| Java | `public class MyHttpClientVerticle extends AbstractVerticle {`<br>` @Override`<br>` public void start() {`<br>`  vertx.createHttpClient().get(8080, "localhost", "/",`<br>`  response -> {`<br>`  System.out.println("Response: " + response.statusMessage());`<br>`  response.bodyHandler(buffer ->`<br>`  System.out.println("Data: " + buffer.toString())`<br>`  );`<br>`  })`<br>`  .end();`<br>` }`<br>`}` |
| Groovy | `vertx.createHttpClient().get(8080, "localhost", "/", { resp ->`<br>` println("Response ${resp.statusCode()}")`<br>` resp.bodyHandler({ body ->`<br>`  println("Data ${body}")`<br>` })`<br>`}).end()` |
| JavaScript | `vertx.createHttpClient().get(8080, "localhost", "/", function (resp) {`<br>`  console.log("Response " + resp.statusCode());`<br>`  resp.bodyHandler(function (body) {`<br>`  console.log("Data " + body);`<br>`  });`<br>`}).end();` |

# 6. EventBus

The event bus is the backbone of any Vert.x application. It allows the components composing your application to interact, regardless of the implementation language and their localization. The event bus offers three methods of delivery: point to point, publish/subscribe, and request-response. On the event bus, messages are sent to addresses. An address is a simple String. Consumers listen for messages by registering a Handler on a specific address.

## 6.A Point to Point

| Language | Sender Verticle (send a message every second) | Deployment |
|---|---|---|
| Java | `vertx.setPeriodic(1000, v -> vertx.eventBus() .send("address", "my message"));` | `vertx.eventBus().consumer("address", message -> System.out.println("Received: " + message.body()));` |
| Groovy | `vertx.setPeriodic(1000, { v -> vertx.eventBus().send("address", "my message") })` | `vertx.deployVerticle( "verticles/my-verticle.js", {"config": {"key": "value"}});` |
| JavaScript | `var eb = vertx.eventBus(); vertx.setPeriodic(1000, function (v) { eb.send("address", "my message"); });` | `var eb = vertx.eventBus(); eb.consumer("address", function (message) { console.log("Received: " + message.body()); });` |

## 6.B Publish/Subscribe

| | | |
|---|---|---|
| Java | `vertx.setPeriodic(1000, v -> vertx.eventBus() .send("address", "my message"));` | `vertx.eventBus().consumer("address", message -> System.out.println("Received: " + message.body()));` |
| Groovy | `vertx.setPeriodic(1000, { v -> vertx.eventBus().publish("address", "my broadcasted message") })` | `vertx.eventBus().consumer("address", { message -> println("Received: ${message.body()}") })` |
| JavaScript | `var eb = vertx.eventBus(); vertx.setPeriodic(1000, function (v) { eb.publish("address", "my broadcasted message"); });` | `var eb = vertx.eventBus(); eb.consumer("address", function (message) { console.log("Received: " + message.body()); });` |

## 6.C Request-Response

| Language | Sender Verticle (send a message every second) | Deployment |
|---|---|---|
| Java | `vertx.setPeriodic(1000, v -> vertx.eventBus() .send("address", "my message", reply -> { if (reply.succeeded()) { System.out.println("Response: " + reply.result().body()); } else { System.out.println("No reply"); } }));` | `vertx.eventBus().consumer("address", message -> { System.out.println("Received: " + message.body()); message.reply("my response"); });` |
| Groovy | `vertx.setPeriodic(1000, { v -> vertx.eventBus().send("address", "my message", { reply -> if (reply.succeeded()) { println("Response: ${reply.result().body()}") } else { println("No reply") } }) })` | `vertx.eventBus().consumer("address", { message -> println("Received: ${message.body()}") message.reply("my response") })` |
| JavaScript | `var eb = vertx.eventBus(); vertx.setPeriodic(1000, function (v) { eb.send("address", "my message", function (reply, reply_err) { if (reply_err == null) { console.log("Response: " + reply.body()); } else { console.log("No reply"); } }); });` | `var eb = vertx.eventBus(); eb.consumer("address", function (message) { console.log("Received: " + message.body()); message.reply("my response"); });` |

## 6.D Delivery Options

When sending or publishing a message on the event bus, you can pass delivery options to configure:

- The send timeout (if the receiver does not reply to the message, the reply handler receives a failed result).
- The message headers than can be used to pass metadata about the message content to the consumers. For example, you can pass the sending time or an identifier using a header.
- The codec name to serialize the message on the event bus (only required for non supported types).

# 7. Executing blocking code

As mentioned above, you **must not run blocking code on the event loop**. Vert.x provides two ways to execute blocking code: **`vertx.executeBlocking`** and Worker verticles.

## 7.A The "executeBlocking" construct

The **`executeBlocking`** construct lets you execute blocking code directly in your code. The **`executeBlocking`** method takes two functions as parameters: the first one is run on a worker thread; the second function is executed on the event loop once the first function has completed the provided **`Future`** object.

| Language | Verticle |
|---|---|
| Java | `vertx.<String>executeBlocking(`<br>` future -> {`<br>`  // Run some blocking code on a worker thread`<br>`  // Complete or fail the future once done`<br>` future.complete("my result");`<br>`  // Example of failure: future.fail("failure cause");`<br>` },`<br>` ar -> {`<br>`  // Run on the event loop`<br>` if (ar.succeeded()) {`<br>`  // The blocking has completed successfully`<br>` } else {`<br>`  // The blocking code has failed`<br>` }`<br>`}`<br>`);` |
| Language | |
| Groovy | `vertx.executeBlocking(`<br>` { future ->`<br>`  // Run some blocking code on a worker thread`<br>`  // Complete or fail the future once done`<br>` future.complete("my result")`<br>`  // Example of failure: future.fail("failure cause")`<br>` },`<br>` { ar ->`<br>`  // Run on the event loop`<br>` if (ar.succeeded()) {`<br>`  println(ar.result())`<br>`  // The blocking has completed successfully`<br>` } else {`<br>`  // The blocking code has failed`<br>` }`<br>`}`<br>`);` |

| Language | Verticle |
|---|---|
| JavaScript | `vertx.executeBlocking(`<br>` function (future) {`<br>`  // Run some blocking code on a worker thread`<br>`  // Complete or fail the future once done`<br>`  future.complete("my result");`<br>`  // Example of failure: future.fail("failure cause");`<br>` },`<br>` function (res, err) {`<br>`  // Run on the event loop`<br>`  if (err == null) {`<br>`  console.log(res);`<br>`  // The blocking has completed successfully`<br>`  } else {`<br>`  // The blocking code has failed`<br>`  }`<br>` }`<br>`);` |

## 7.B Worker verticles

A worker verticle is a specialized verticle that can run blocking code. Workers are not executed on the Vert.x event loop, but instead by a worker thread. To mark a verticle as worker, use deployment options as follows:

| | |
|---|---|
| Java | `public class MainVerticle extends AbstractVerticle {`<br><br>`@Override`<br>`public void start() {`<br>` vertx.deployVerticle(MyWorkerVerticle.class.getName(),`<br>`  new DeploymentOptions().setWorker(true));`<br>` }`<br>`}` |
| Groovy | `vertx.deployVerticle("verticles/MyWorkerVerticle.java", [worker: true])` |
| JavaScript | `vertx.deployVerticle("verticles/MyWorkerVerticle.java", [worker: true])` |

# 8. Executing Vert.x applications

Both the **vertx** CLI and the fat-jar created in the generated projects use the same Launcher. This Launcher is convenient but not mandatory --- alternatively, you can implement your own main class.

When using the Launcher, the execution can be configured:

| Parameter | Description | Example with the vertx CLI | Example with a fat jar |
|---|---|---|---|
| `-cluster` | Enable cluster mode | `vertx run my-verticle -cluster` | `java -jar my-fat.jar -cluster` |
| `-cp` | Add items to the classpath | `vertx run my-verticle -cp .:./conf` | `java -jar my-fat.jar -cp .:./conf` |

| Parameter | Description | Example with the vertx CLI | Example with a fat jar |
|---|---|---|---|
| `--instances=x` | Create as many instances as specified of the main verticle | `vertx run my-verticle -cluster` | `java -jar my-fat.jar --instances=2` |
| `-ha` | launch the vert.x instance in high-availability mode | `vertx run my-verticle -cp .:./conf` | `java -jar my-fat.jar -ha` |
| `--conf=x` | Configure the main verticle with the given json file | `vertx run my-verticle --conf=my-conf.json` | `java -jar my-fat.jar --conf=my-conf.json` |

When using the Launcher, the execution can be configured:

| | | | |
|---|---|---|---|
| `start` | Start an application in background | `vertx start my-verticle -id my-service` | `java -jar my-fat.jar start -id my-service` |
| `list` | List all Vert.x applications launched in background | `vertx list` | `java -jar my-fat.jar list` |
| `stop` | Stop an application launched in background | `vertx stop my-service` | `java -jar my-fat.jar stop my-service` |

## 9. Further learning

This cheat sheet covers just a small part of the Vert.x ecosystem --- Vert.x also provides components such as Vert.x Web to build modern web applications, a set of bridges and clients to interact with AMQP, MQTT, Kafka and legacy applications (using Apache Camel). It provides components to build microservices (service discovery, circuit breaker...), RPC interactions, and more.

Learn more about this ecosystem on:

- http://vertx.io
- http://vertx.io/blog
- http://vertx.io/materials/

## 10. About the author

**Clement Escoffier** is Principal Software Developer at Red Hat. Clement has had several professional lives, from academic positions to management. He has experience with many domains and technologies, such as: OSGi, mobile, continuous delivery, and devops. Clement is an active contributor on many open source projects such as Apache Felix, iPOJO, Wisdom Framework, and obviously, Eclipse Vert.x.