**INTRODUCTION**

Vert.x is an open source, reactive, polyglot toolkit or platform running on the Java Virtual Machine. You can think of Vert.x as an alternative to the Java Enterprise Edition but with a different approach to solving the same problem - developing networked, highly concurrent applications.

**Vert.x as a Toolkit or Platform**

Vert.x can be used like toolkit. You can embed Vert.x in your standalone Java application, instantiate a Vertx object and call methods on it to get Vert.x to do what you need it to. In toolkit mode your code controls Vert.x.

Vert.x can also function like a platform. You can startup Vert.x from the command line and tell it what components of yours to run. This is similar to how you normally start up a Java EE application server.

## Vert.x is Reactive

Vert.x calls itself a *reactive toolkit*. Reactive applications consists of components that send messages or events to each other. This is quite a different internal design than Java EE. This internal design makes Vert.x suitable for different types of applications than Java EE (for instance chat and game servers). In fact, I would risk the bold claim that Vert.x is suitable for more types of applications than Java EE.

In my tutorial about [**concurrency models**](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html) I explain the different concurrency models that Java EE and Vert.x use. Java EE uses the parallel worker model. Vert.x on the other hand, uses the assembly line model with channels.

## Vert.x is Polyglot

Vert.x is polyglot meaning you can implement the components you want Vert.x to execute (called "Verticles") in many different languages. Vertx. can execute Java, JavaScript, Ruby, Python and Groovy. Support for Scala and Clojure should be arriving soon (it was originally planned for v. 3.0 but I am not sure if they got it in). I have also seen rumors about support for PHP and Ceylon.

Vert.x can even deploy verticles written in different languages into the same application. This gives you the freedom to choose the most suitable language for each job. Some tasks might be easier to implement using a functional language like Scala. Others easier in a more traditionally imperative language like Java.

The polyglot nature of Vert.x also gives you the freedom to experiment with new languages to see if they suit your type of applications better. You can write the same tasks in different languages and compare.

If you are new to Vert.x but know one of the programming languages Vert.x can execute, you just have to focus on learning Vert.x and not a new programming language too. This makes Vert.x easier to learn regardless of what programming language you already know (as long as it is a language that runs on Vert.x).

In the same way Vert.x makes it easier to learn a new programming language. You don't have to learn both a language and a new platform (e.g. Ruby and Rails) from scratch. The internal concepts of Vert.x are the same, even if you write verticles in different languages. That means you just have to focus on the new language, and not the whole platform.

Finally, if you choose to change language, Vert.x gives you the freedom to make a slow transition from one language to another. You can keep all your old code and still use it in the same application. You can then write new verticles in the new language, as well as slowly rewrite old verticles in the new language.

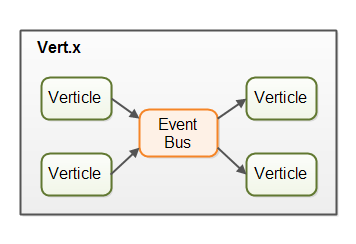
**Verticles**

Vert.x can deploy and execute components called Verticles. You can think of verticles as being similar to servlets or message driven EJBs in the Java Enterprise Edition model. Here is a simple diagram illustrating the Vert.x platform with 4 verticles running:

## Vert.x overview with verticles illustrated.

## The Event Bus

Verticles are event driven, meaning they do not run unless they receive a message. Until then they remain dormant. Verticles can communicate with each other via the Vert.x event bus. This diagram illustrates how the verticles communicate via the Vert.x event bus:



Messages can be simple objects (e.g. Java objects), strings, CSV, JSON, binary data or whatever else you need.

Verticles can send and listen to *addresses*. An address is like a named channel. When a message is sent to a given address, all verticles that listen on that address receive the message. Verticles can subscribe and unsubscribe to addresses without the senders knowing. This results in a very loose coupling between message senders and message receivers.

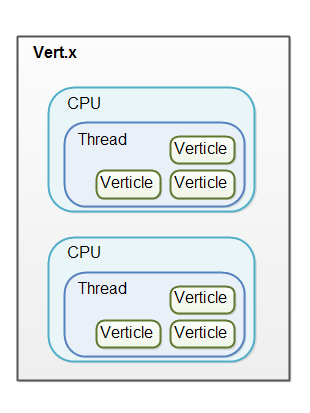
All message handling is asynchronous. If a verticle sends a message to another verticle, that message is first put on the event bus, and control returned to the sending verticle. Later, the message is dequeued and given to the verticles listening on the address the message was sent to.

**The Vert.x Thread Model**

Verticles run in single thread mode. That means, that a verticle is only ever executed by a single thread, and always by the same thread. That means that you will never have to think about multithreading inside your verticle (unless you yourself start other threads which your verticle communicates with etc).

Vert.x is capable of using all the CPUs in your machine, or cores in your CPU. Vert.x does this by creating one thread per CPU. Each thread can send messages to multiple verticles. Remember, verticles are event driven and are only running when receiving messages, so a verticle does not need to have its own exclusive thread. A single thread can distribute messages to multiple verticles.

When a thread delivers a message to a verticle, the message handling code of that verticle is executed by the thread. The message delivery and message handling logic is executed by calling a method in a handler (listener object) registered by the verticle. Once the verticle's message handling logic finishes, the thread can deliver a message to another verticle.



**Vert.x Services**

Vert.x comes with a set of built-in services (functionality). Some of these services are:

* HTTP server
* JDBC connector
* MongoDB connector
* SMTP Mail
* Message queue connectors

And these are only a few of the many, many services Vert.x provides, and which the community have provided for Vert.x.

# Vert.x Installation

# Vert.x is distributed in a zip file containing a bunch of JAR files. You can just unzip the zip file and add these JAR files to the classpath of your Java application, and you are good to go. Pretty simple!

Vert.x is not like a clunky Java application server which needs to be installed, have environment variables set up etc. You can just use Vert.x like any other API, from inside your Java code. As long as the JAR files are available on the classpath of your application.

You can download the binary distributions (ZIP files) here:

[**https://bintray.com/vertx/downloads/distribution/view**](https://bintray.com/vertx/downloads/distribution/view)

**First Vert.x Application**

* [Create Vertx Instance](http://tutorials.jenkov.com/vert.x/your-first-vertx-application.html#create-vertx-instance)
* [Creating a Verticle](http://tutorials.jenkov.com/vert.x/your-first-vertx-application.html#creating-a-verticle)
* [Deploying a Verticle](http://tutorials.jenkov.com/vert.x/your-first-vertx-application.html#deploying-a-verticle)
* [What is Next?](http://tutorials.jenkov.com/vert.x/your-first-vertx-application.html#what-is-next)

Once you have [**installed Vert.x**](http://tutorials.jenkov.com/vert.x/installation.html) you are ready to create your first Vert.x application. This tutorial will take you through creating your first Vert.x application step by step.

## Create Vertx Instance

The first step to using Vert.x embedded in your own Java application is to create a Vertx instance. Here is how you create a Vertx instance:

import io.vertx.core.Vertx;

public class VertxApp {

public static void main(String[] args) {

Vertx vertx = Vertx.vertx();

}

}

You create a Vertx instance by calling Vertx.vertx().

The Vertx instance creates a number of threads internally to handle the exchange of messages between verticles. These threads are not daemon threads, so they prevent the JVM from shutting down, even if the main thread creating the Vertx instance terminates.

## Creating a Verticle

The Vertx instance by itself doesn't do much except all the thread management, creating an event bus etc. which are all communication and infrastructure tasks. In order to get the application to do something useful, you need to deploy one or more verticles (component) inside the Vertx instance.

Before you can deploy a verticle you need to create it. You do so by creating a class that extendsAbstractVerticle. Here is a verticle example:

package examples.vertx;

import io.vertx.core.AbstractVerticle;

import io.vertx.core.Future;

public class MyVerticle extends AbstractVerticle {

@Override

public void start(Future<Void> startFuture) {

System.out.println("MyVerticle started!");

}

@Override

public void stop(Future stopFuture) throws Exception {

System.out.println("MyVerticle stopped!");

}

}

A verticle has a start() and a stop() method which are called when the verticle is deployed and when it is undeployed. You should perform any necessary initialization work inside the start() method, and any necessary cleanup work inside the stop() method.

## Deploying a Verticle

Once you have created a verticle you need to deploy it to the Vertx instance. You deploy a verticle using one of the deployVerticle() methods on the Vertx instance. Here is a Vert.x verticle deployment example:

import io.vertx.core.Vertx;

public class VertxVerticleMain {

public static void main(String[] args) {

Vertx vertx = Vertx.vertx();

vertx.deployVerticle(new MyVerticle());

}

}

The verticle is deployed using this method call:

vertx.deployVerticle(new MyVerticle());

This method call deploys the MyVerticle instance passed as parameter to the deployVerticle() method.

The Vertx instance has another deployVerticle() method which takes the fully qualified class name of the verticle to deploy as parameter. Here is how deploying the MyVerticle using that method would look:

vertx.deployVerticle("examples.vertx.MyVerticle");

There are more options available for deploying verticles. For instance, you can specify how many verticle instances of a given verticle class to deploy. All that will be covered in later texts.

## What is Next?

Once you know how to create a Vertx instance and know how to deploy verticles, you are ready to start playing around with the various parts of Vert.x . Each part will be covered in later texts in this tutorial trail.

# Vert.x Verticles

* [Implementing a Verticle](http://tutorials.jenkov.com/vert.x/verticles.html#implementing-a-verticle)
  + [start()](http://tutorials.jenkov.com/vert.x/verticles.html#start)
  + [stop()](http://tutorials.jenkov.com/vert.x/verticles.html#stop)
* [Deploying a Verticle](http://tutorials.jenkov.com/vert.x/verticles.html#deploying-a-verticle)
* [Deploying a Verticle From Another Verticle](http://tutorials.jenkov.com/vert.x/verticles.html#deploying-a-verticle-from-another-verticle)
* [Using The Event Bus](http://tutorials.jenkov.com/vert.x/verticles.html#using-the-event-bus)
  + [Listening for Messages](http://tutorials.jenkov.com/vert.x/verticles.html#listening-for-messages)
  + [Sending Messages](http://tutorials.jenkov.com/vert.x/verticles.html#sending-messages)

The term *verticle* is the name of the components you can deploy to Vert.x. A verticle is in some ways similar to a Servlet or a message driven EJB in Java EE. However, verticles work differently, and the concurrency model under which verticles and Servlets and EJBs execute is different. This text will take a closer look at how to create and deploy verticles in Vert.x, and how verticles can communicate with each other over the internal event bus.

## Implementing a Verticle

You implement a verticle by creating a class that extends io.vertx.core.AbstractVerticle. Here is an example verticle class:

import io.vertx.core.AbstractVerticle;

public class BasicVerticle extends AbstractVerticle {

}

The class BasicVerticle extends AbstractVerticle but it doesn't have any functionality. We will add that in the following sections.

### start()

The AbstractVerticle class contains a start() method which you can override in your verticle class. The start() method is called by Vert.x when the verticle is deployed and ready to start. Here is how implementing the start() method looks:

public class BasicVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

System.out.println("BasicVerticle started");

}

}

The start() method is where you initialize your verticle. Inside the start() method you will normally create e.g. HTTP or TCP server, register event handlers on the event bus, deploy other verticles, or whatever else your verticle needs to do its work.

The AbstracVerticle class also contains another version of start() which takes a Future as parameter. This Future can be used to asynchronously tell Vert.x if the Verticle was deployed successfully.

You will see later in this tutorial how to use both versions of the start() method.

### stop()

The AbstractVerticle class also contains a stop() method you can override. The stop() method is called when Vert.x shuts down and your verticle needs to stop. Here is an example of overriding the stop()method in your own verticle:

public class BasicVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

System.out.println("BasicVerticle started");

}

@Override

public void stop() throws Exception {

System.out.println("BasicVerticle stopped");

}

}

## Deploying a Verticle

Once you have created a Verticle you need to deploy it inside Vert.x in order to execute it. Here is how you deploy a verticle:

public class VertxVerticleMain {

public static void main(String[] args) throws InterruptedException {

Vertx vertx = Vertx.vertx();

vertx.deployVerticle(new BasicVerticle());

}

}

First a Vertx instance is created. Second, the deployVerticle() method is called on the Vertx instance, with an instance of your verticle (BasicVerticle in this example) as parameter.

Vert.x will now deploy the verticle internally. Once Vert.x deploys the verticle, the verticle's start() method is called.

The verticle will be deployed asynchronously, so the verticle may not be deployed by the time thedeployVerticle() method returns. If you need to know exactly when a verticle is fully deployed, you can provide a Handler implementation to the the deployVerticle(). Here is how that looks:

vertx.deployVerticle(new BasicVerticle(), new Handler<AsyncResult<String>>() {

@Override

public void handle(AsyncResult<String> stringAsyncResult) {

System.out.println("BasicVerticle deployment complete");

}

});

Or using a [**Java lambda expression**](http://tutorials.jenkov.com/java/lambda-expressions.html) :

vertx.deployVerticle(new BasicVerticle(), stringAsyncResult -> {

System.out.println("BasicVerticle deployment complete");

});

## Deploying a Verticle From Another Verticle

It is possible to deploy one verticle from inside another verticle. Here is an example:

public class BasicVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

System.out.println("BasicVerticle started");

vertx.deployVerticle(new SecondVerticle());

}

@Override

public void stop() throws Exception {

System.out.println("BasicVerticle stopped");

}

}

## Using The Event Bus

It is very common for verticles to either listen for incoming messages from the event bus, or to write messages to other verticles via the event bus. Therefore I will show you how to do both in the following sections.

### Listening for Messages

When a verticle wants to listen for messages from the event bus, it listens on a certain address. An address is just a name (a String) which you can choose freely.

Multiple verticles can listen for messages on the same address. This means that an address is not unique to a single verticle. An address is thus more like the name of a channel you can communicate via. Multiple verticles can listen for messages on an address, and multiple verticles can send messages to an address.

A verticle can obtain a reference to the event bus via the vertx instance inherited from AbstractVerticle

Here is how listening for messages on a given address looks:

public class EventBusReceiverVerticle extends AbstractVerticle {

public void start(Future<Void> startFuture) {

vertx.eventBus().consumer("anAddress", message -> {

System.out.println("1 received message.body() = "

+ message.body());

});

}

}

This example shows a verticle that registers a *consumer* (listener) of messages on the Vert.x event bus. The consumer is registered with the address anAddress, meaning it consumes messages sent to this address via the event bus.

The consumer is a handler object which contains a single method. That is why it is implemented above using a lambda expression.

### Sending Messages

Sending messages via the event bus can be done via either the send() or publish() method on the event bus.

The publish method sends the message to all verticles listening on a given address.

The send() method sends the message to just one of the listening verticles. Which verticle receives the message is decided by Vert.x. At the time of writing the Vert.x docs says that a verticle is chosen using a "non-strict round robin" algorithm. This basically means that Vert.x will try to distribute the messages evenly among the listening verticles. This is useful for distributing work load over multiple verticles (e.g. threads or CPUs).

Here is an example that deploys two event bus consumers (listeners), and one event bus sender. The sender sends two messages to a given address. The first message is sent via the publish() method, so both consumers receive the message. The second message is sent via the send() method, so only one of the consumers will receive the message.

Vertx vertx = Vertx.vertx();

vertx.deployVerticle(new EventBusReceiverVerticle("R1"));

vertx.deployVerticle(new EventBusReceiverVerticle("R2"));

Thread.sleep(3000);

vertx.deployVerticle(new EventBusSenderVerticle());

public class EventBusSenderVerticle extends AbstractVerticle {

public void start(Future<Void> startFuture) {

vertx.eventBus().publish("anAddress", "message 2");

vertx.eventBus().send ("anAddress", "message 1");

}

}

public class EventBusReceiverVerticle extends AbstractVerticle {

private String name = null;

public EventBusReceiverVerticle(String name) {

this.name = name;

}

public void start(Future<Void> startFuture) {

vertx.eventBus().consumer("anAddress", message -> {

System.out.println(this.name +

" received message: " +

message.body());

});

}

}

If you run this code you will see that the first message is received by both consumers (R1 + R2) whereas the second message is only received by one of the consumers.

# Vert.x Buffers

* [Creating a Buffer](http://tutorials.jenkov.com/vert.x/buffers.html#creating-a-buffer)
* [Buffer Length](http://tutorials.jenkov.com/vert.x/buffers.html#buffer-length)
* [Writing to a Buffer](http://tutorials.jenkov.com/vert.x/buffers.html#writing-to-a-buffer)
* [Reading From a Buffer](http://tutorials.jenkov.com/vert.x/buffers.html#reading-from-a-buffer)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-04-21 |

Verticles running in Vert.x often need to process blocks of data. For instance, data from an incoming HTTP request, data loaded from disk, or data generated as response to an HTTP request etc. Vert.x provides a Buffer interface (io.vertx.core.buffer.Buffer) to help you handle such data blocks.

A Buffer in Vert.x can hold binary data. As such a Buffer is similar to a byte array, except the Buffer can expand its capacity dynamically as you write data to it.

## Creating a Buffer

Creating a Buffer is done using the static buffer() method in the Buffer interface. Here is how creating a Vert.x Buffer looks:

Buffer buffer = Buffer.buffer();

You can also create a Buffer with some data inside from the start. Here is how you create a Bufferinitialized with bytes from a byte array:

byte[] initialData = new byte[]{1, 2, 3};

Buffer buffer2 = Buffer.buffer(initialData);

You can also initialize a Buffer with the contents of a String. Here is an example of creating a Bufferinitialized to the value of a String:

Buffer buffer3 = Buffer.buffer("initial data");

If you want the bytes stored in the Buffer to be encoded using a special encoding (e.g. UTF-8, UTF-16 etc.) you can specify the encoding as the second parameter to the buffer() method, like this:

Buffer buffer4 = Buffer.buffer("initial data", "UTF-8");

Buffer buffer5 = Buffer.buffer("initial data", "UTF-16");

This is equivalent to this:

Buffer buffer6 = Buffer.buffer("initial data".getBytes("UTF-8"));

Buffer buffer7 = Buffer.buffer("initial data".getBytes("UTF-16"));

## Buffer Length

You can read the length of a Buffer using its length() method. Here is an example:

Buffer buffer = Buffer.buffer();

System.out.println("buffer.length() = " + buffer.length());

## Writing to a Buffer

You can write to a specific position inside a buffer using one of the set...() methods. Here are some examples:

Buffer buffer = Buffer.buffer();

System.out.println("buffer.length() = " + buffer.length());

buffer.setByte ( 0, (byte) 127);

buffer.setShort ( 2, (short) 127);

buffer.setInt ( 4, 127);

buffer.setLong ( 8, 127);

buffer.setFloat (16, 127.0F);

buffer.setDouble(20, 127.0D);

System.out.println("buffer.length() = " + buffer.length());

Notice that the length of the buffer starts as 0. Then, as data is written into it, the buffer expands to fit the required size. After executing the code above, the length of the buffer is 28.

You can also write data to a Buffer using one of the append...() methods. Here are some examples:

Buffer buffer = Buffer.buffer();

System.out.println("buffer.length() = " + buffer.length());

buffer.appendByte ((byte) 127);

buffer.appendShort ((short) 127);

buffer.appendInt ( 127);

buffer.appendLong ( 127);

buffer.appendFloat ( 127.0F);

buffer.appendDouble( 127.0D);

System.out.println("buffer.length() = " + buffer.length());

The append...() methods do not need an index as parameter. They always append the data to the end of the Buffer. The length of the Buffer will be 27 after executing this code (because all data following the first byte is inserted from index 1, and not index 2 as in the previous example).

## Reading From a Buffer

You can read the data stored in a Buffer using the many get...() methods. Here are some examples:

byte aByte = buffer.getByte ( 0);

short aShort = buffer.getShort ( 2);

int anInt = buffer.getInt ( 4);

long aLong = buffer.getLong ( 8);

float aFloat = buffer.getFloat (16);

double aDouble = buffer.getDouble(20);

# Vert.x HTTP Server

* [Creating an HTTP Server](http://tutorials.jenkov.com/vert.x/http-server.html)
* [Starting the HTTP Server](http://tutorials.jenkov.com/vert.x/http-server.html#starting-the-http-server)
* [Setting a Request Handler on the HTTP Server](http://tutorials.jenkov.com/vert.x/http-server.html#setting-a-request-handler-on-the-http-server)
  + [Request Headers and Parameters](http://tutorials.jenkov.com/vert.x/http-server.html#request-headers-and-parameters)
  + [Handling POST Requests](http://tutorials.jenkov.com/vert.x/http-server.html#handling-post-requests)
* [Sending Back an HTTP Response](http://tutorials.jenkov.com/vert.x/http-server.html#sending-back-an-http-response)
* [Closing the HTTP Server](http://tutorials.jenkov.com/vert.x/http-server.html#closing-the-http-server)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-11-13 |

Vert.x makes it easy to create an HTTP server so your application can receive HTTP requests. You can create one or more HTTP servers, depending on your need. In this tutorial I will look at how to create an HTTP server from inside a verticle and how to handle requests.

## Creating an HTTP Server

Creating an HTTP server is done using the Vertx instance method createHttpServer(). Here is an example of creating an HTTP server in Vert.x:

HttpServer httpServer = vertx.createHttpServer();

It is common to start an HTTP server from within an verticle. That way all handlers registered on the HTTP server will be executed by the same thread that started the verticle.

import io.vertx.core.AbstractVerticle;

import io.vertx.core.http.HttpServer;

public class VertxHttpServerVerticle extends AbstractVerticle {

private HttpServer httpServer = null;

@Override

public void start() throws Exception {

httpServer = vertx.createHttpServer();

}

}

## Starting the HTTP Server

Once you have created the HTTP server, you can start it using its listen() method. Here is how starting the HTTP server looks:

public class VertxHttpServerVerticle extends AbstractVerticle {

private HttpServer httpServer = null;

@Override

public void start() throws Exception {

httpServer = vertx.createHttpServer();

**httpServer.listen(9999);**

}

}

The HttpServer class has more versions of the listen() method too, which gives you different options for starting the HTTP server.

## Setting a Request Handler on the HTTP Server

In order to handle incoming HTTP requests you must set a request handler on the HTTP server. This is normally done before starting the server. Here is a Vert.x HTTP server request handler example:

public class VertxHttpServerVerticle extends AbstractVerticle {

private HttpServer httpServer = null;

@Override

public void start() throws Exception {

httpServer = vertx.createHttpServer();

**httpServer.requestHandler(new Handler<HttpServerRequest>() {**

**@Override**

**public void handle(HttpServerRequest request) {**

**System.out.println("incoming request!");**

**}**

**});**

httpServer.listen(9999);

}

}

Every time an HTTP request arrives at the HTTP server, the handle() method of the Handler object is called. Inside the handle() method you can execute the code needed to handle the HTTP request.

### Request Headers and Parameters

You can access HTTP headers and parameters from the HttpServerRequest object passed as parameter to the handle() method. Here is an example showing how to access a few of the properties of a HTTP request:

httpServer.requestHandler(new Handler<HttpServerRequest>() {

@Override

public void handle(HttpServerRequest request) {

System.out.println("incoming request!");

request.uri();

request.path();

request.getParam("p1");

}

});

### Handling POST Requests

If the HTTP request is a HTTP POST request you need to handle it a bit differently. You need to attach a body handler to the HTTP request. The body handler is called whenever data from the request body arrives. Here is how that looks:

httpServer.requestHandler(new Handler<HttpServerRequest>() {

@Override

public void handle(HttpServerRequest request) {

System.out.println("incoming request!");

if(request.method() == HttpMethod.POST){

request.handler(new Handler<Buffer>() {

@Override

public void handle(Buffer buffer) {

}

});

}

}

});

If you want to wait until the full HTTP POST body has arrived you can attach an end handler instead. The end handler is not called until the full HTTP POST body has been received. However, the end handler does not have direct access to the full HTTP POST body. You need to collect that in the request handler. Here is a Vert.x HTTP request end handler example which does all that:

httpServer.requestHandler(new Handler<HttpServerRequest>() {

@Override

public void handle(HttpServerRequest request) {

System.out.println("incoming request!");

Buffer fullRequestBody = Buffer.buffer();

if(request.method() == HttpMethod.POST){

request.handler(new Handler<Buffer>() {

@Override

public void handle(Buffer buffer) {

fullRequestBody.appendBuffer(buffer);

}

}

request.endHandler(new Handler<Buffer>() {

@Override

public void handle(Buffer buffer) {

// here you can access the

// fullRequestBody Buffer instance.

}

});

}

}

});

## Sending Back an HTTP Response

You can of course send back an HTTP response for an incoming HTTP request. To do so you need to obtain the HttpServerResponse instance from the request object. This is how you obtain the HttpServerResponse object:

HttpServerResponse response = request.response();

Once you have obtained a HttpServerResponse instance you can set the HTTP response status code and headers like this:

response.setStatusCode(200);

response.headers()

.add("Content-Length", String.valueOf(57))

.add("Content-Type", "text/html")

;

After writing the headers back you can write the response body back via the write() method, like this:

response.write("Vert.x is alive!");

response.end();

You can call write() multiple times to add more data to the response body. The write() method also exists in a version that takes a Vert.x Buffer instance as parameter. This method will write the contents of the Buffer to the HTTP response.

The write() method is asynchronous and returns immediately after queuing up the string or buffer.

Once you have finished writing the HTTP response body you should end the HTTP response. This is done by calling the end() method as shown in the previous example. You can also write the HTTP response body and end the response in a single method call, like this:

response.end("Vert.x is alive!");

The end() method can take either a String or Buffer as parameter. The parameter will be written to the response body, and the response ended after that.

## Closing the HTTP Server

To close an HTTP server you simply call its close() method like this:

httpServer.close();

The close() method executes asynchronously, so the HTTP server may not be fully closed by the time the close() method returns. You can pass a close handler as parameter to the close() method, to be notified when the HTTP server is fully closed.

# Vert.x HTTP Client

* [Creating an HTTP Client](http://tutorials.jenkov.com/vert.x/http-client.html#creating-an-http-client)
* [Sending a GET Request](http://tutorials.jenkov.com/vert.x/http-client.html#sending-a-get-request)
* [Handling the HTTP Response](http://tutorials.jenkov.com/vert.x/http-client.html#handling-the-http-response)
* [More Methods](http://tutorials.jenkov.com/vert.x/http-client.html#more-methods)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-04-28 |

Vert.x contains a HTTP client that makes it easy to make HTTP requests asynchronously. Remember, every IO action in Vert.x must be performed asynchronously to avoid blocking the event loop. The event loop is the thread(s) that manages your verticles and their handlers. Java's built-in URL and URLConnectionare not asynchronous, so you should not use these classes in your Vert.x apps.

## Creating an HTTP Client

You create a Vert.x HTTP client like this:

HttpClient httpClient = vertx.createHttpClient();

When you create a Vert.x HTTP client from inside a verticle, it looks like this:

public class VertxHttpClientVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

HttpClient httpClient = vertx.createHttpClient();

}

}

## Sending a GET Request

Once you have created the Vert.x HTTP client you can send a GET request using its getNow() method. Here is a Vert.x HttpClient getNow() example:

httpClient.getNow(80, "tutorials.jenkov.com", "/", new Handler<HttpClientResponse>() {

@Override

public void handle(HttpClientResponse httpClientResponse) {

System.out.println("Response received");

}

});

The first parameter to the getNow() method is the TCP port to connect to the remote HTTP server on. The getNow() method exists in a version where you can leave the port out. The port will then default to port 80 which is the HTTP protocol's default TCP port.

The second parameter to the getNow() method is the domain name of the remote HTTP server to connect to. Notice that there is not "http://" in front of the domain name. The Vert.x HTTP client knows that this is an HTTP request, so you don't need to include the protocol in the domain name.

The third parameter to the getNow() method is the URI to the resource to retrieve. The example above retrieves the frontpage of the website ("/"). Another URI could have been "/vert.x/index.html".

The fourth parameter is a Handler implementation which is called when the response for the HTTP request is received. How to handle the HTTP response is explained below.

## Handling the HTTP Response

The Handler implementation passed to the getNow() method is called when the headers of the HTTP response are received. If you do not need to access the response body, you can process the response already in this handler.

However, if you do need to access the body of the HTTP response, you need to register another handler on the HttpClientResponse that is passed as parameter to the first Handler's handle() method. Here is how that looks:

httpClient.getNow(80, "tutorials.jenkov.com", "/", new Handler<HttpClientResponse>() {

@Override

public void handle(HttpClientResponse httpClientResponse) {

httpClientResponse.bodyHandler(new Handler<Buffer>() {

@Override

public void handle(Buffer buffer) {

System.out.println("Response (" + buffer.length() + "): ");

System.out.println(buffer.getString(0, buffer.length()));

}

});

}

});

The Handler implementation passed to the bodyHandler() method of the HttpClientResponse is called when the full HTTP response body is received.

The Buffer passed as parameter to the body Handler contains the full HTTP response.

## More Methods

The HttpClient contains a lot more methods for sending GET, POST, PUT, DELETE and HEAD requests. It also contains more options for configuring the HttpClient with defaults, and for handling the HTTP response asynchronously, incrementally, as the response body is received (instead of when the full body is received). See the Vert.x JavaDoc for more info about these options.

# Vert.x TCP Server

* [Creating a TCP Server](http://tutorials.jenkov.com/vert.x/tcp-server.html#creating-a-tcp-server)
* [Starting the TCP Server](http://tutorials.jenkov.com/vert.x/tcp-server.html#starting-the-tcp-server)
* [Setting a Connect Handler on the TCP Server](http://tutorials.jenkov.com/vert.x/tcp-server.html#setting-a-connect-handler-on-the-tcp-server)
* [Reading Data From The Socket](http://tutorials.jenkov.com/vert.x/tcp-server.html#reading-data-from-the-socket)
* [Writing Data to The Socket](http://tutorials.jenkov.com/vert.x/tcp-server.html#writing-data-to-the-socket)
* [Closing The TCP Server](http://tutorials.jenkov.com/vert.x/tcp-server.html#closing-the-tcp-server)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-04-29 |

Vert.x comes with a TCP server which makes it easy to create TCP servers in your application. You can create one or more TCP servers, depending on your need. In this tutorial I will look at how to create a TCP server from inside a verticle and how to handle incoming connections and data.

## Creating a TCP Server

You create a Vert.x TCP server by calling the createNetServer() method on the Vertx object. Here is how that looks:

NetServer server = vertx.createNetServer();

You will often create a TCP server from inside a verticle. Here is how you create a Vert.x TCP server from inside a verticle:

import io.vertx.core.AbstractVerticle;

import io.vertx.core.net.NetServer;

public class VertxTcpServerVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

NetServer server = vertx.createNetServer();

}

}

## Starting the TCP Server

Once you have created the TCP server, you can start it using its listen() method. Here is how starting the TCP server looks:

public class VertxTcpServerVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

NetServer server = vertx.createNetServer();

**server.listen(10000);**

}

}

This example starts the server and instructs it to listen on TCP port 10.000 .

The NetServer class contains more versions of the listen() method which gives you different options for starting the TCP server.

## Setting a Connect Handler on the TCP Server

In order to handle incoming TCP connections you need to set a connect handler on the TCP server. This is normally done before starting the TCP server. Here is a Vert.x TCP server connect handler example:

public class VertxTcpServerVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

NetServer server = vertx.createNetServer();

server.connectHandler(new Handler<NetSocket>() {

@Override

public void handle(NetSocket netSocket) {

System.out.println("Incoming connection!");

}

});

server.listen(10000);

}

}

The handle() method of the connect handler is called whenever a new TCP connection is created by a client of the TCP server. The NetSocket object passed as parameter to the handle() method gives access to the incoming connection (socket etc.).

## Reading Data From The Socket

In order to read data from the incoming connections you need to set a handler on the NetSocket object for the connection. Here is how that is done:

public class VertxTcpServerVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

NetServer server = vertx.createNetServer();

server.connectHandler(new Handler<NetSocket>() {

@Override

public void handle(NetSocket netSocket) {

System.out.println("Incoming connection!");

**netSocket.handler(new Handler<Buffer>() {**

**@Override**

**public void handle(Buffer buffer) {**

**System.out.println("incoming data: "+buffer.length());**

**buffer.getString(0,buffer.length());**

**}**

**});**

}

});

server.listen(10000);

}

}

## Writing Data to The Socket

Once you have processed the incoming data you may want to write a response back to the client. You can do so via the NetSocket instance passed to the connect handler's handle() method. Here is an example of writing data back to the client via the NetSocket:

public class VertxTcpServerVerticle extends AbstractVerticle {

@Override

public void start() throws Exception {

NetServer server = vertx.createNetServer();

server.connectHandler(new Handler<NetSocket>() {

@Override

public void handle(NetSocket netSocket) {

System.out.println("Incoming connection!");

netSocket.handler(new Handler<Buffer>() {

@Override

public void handle(Buffer inBuffer) {

System.out.println("incoming data: " + inBuffer.length());

String data =

inBuffer.getString(0, inBuffer.length());

**Buffer outBuffer = Buffer.buffer();**

**outBuffer.appendString("response...");**

**netSocket.write(outBuffer);**

}

});

}

});

server.listen(10000);

}

}

The write() method of the NetSocket method is asynchronous. Thus, it returns immediately. The data will be written to the underlying socket at a later time.

The NetSocket class contains more versions of the write() method which enables you to write e.g. a String directly to the socket without using a Buffer. See the Vert.x JavaDoc for more details.

## Closing The TCP Server

When you are finished with the TCP server you can close it using its close() method. Here is how closing a Vert.x TCP server looks:

server.close();

Closing the TCP server is an asynchronous actions, so the close() method may exit before the TCP server is actually closed. If you need to be notified of when the TCP server has shut down, you can pass aHandler to the close() method. This handler will then be called when the TCP server is fully closed. Here is how that looks:

server.close(new Handler<AsyncResult<Void>>() {

@Override

public void handle(AsyncResult result) {

if(result.succeeded()){

//TCP server fully closed

}

}

});

# Vert.x TCP Client

* [Creating a TCP Client](http://tutorials.jenkov.com/vert.x/tcp-client.html#creating-a-tcp-client)
* [Connecting to a Remote Server](http://tutorials.jenkov.com/vert.x/tcp-client.html#connecting-to-a-remote-server)
* [Writing Data](http://tutorials.jenkov.com/vert.x/tcp-client.html#writing-data)
* [Reading Data](http://tutorials.jenkov.com/vert.x/tcp-client.html#reading-data)
* [Closing the TCP Connection](http://tutorials.jenkov.com/vert.x/tcp-client.html#closing-the-tcp-connection)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-05-04 |

Vert.x comes with a built-in TCP client that makes it easy to create TCP clients and work with them in asynchronous mode. The Vert.x TCP client class is called io.vertx.core.net.NetClient.

## Creating a TCP Client

You create a TCP client by creating an instance of NetClient. You create an instance of NetClient via the Vertx object's method createNetClient(). Here is how you create a NetClient instance:

NetClient tcpClient = vertx.createNetClient();

Often you will create a NetClient from inside a verticle. Here is how creating a Vert.x NetClient from inside a verticle looks:

import io.vertx.core.AbstractVerticle;

import io.vertx.core.net.NetClient;

public class VertxTcpClientVerticle extends AbstractVerticle {

public void start() {

NetClient tcpClient = vertx.createNetClient();

}

}

## Connecting to a Remote Server

You connect to a remote server by calling the connect() method. Here is how calling the connect()method looks:

public class VertxTcpClientVerticle extends AbstractVerticle {

public void start() {

NetClient tcpClient = vertx.createNetClient();

**tcpClient.connect(80, "jenkov.com",**

**new Handler<AsyncResult<NetSocket>>(){**

**@Override**

**public void handle(AsyncResult<NetSocket> result) {**

**NetSocket socket = result.result();**

**}**

**});**

}

}

You pass the TCP port of the remote server as well as the domain name, and a Handler object which is called when the connection is established. You obtain a reference to the NetSocket connected to the remote server via the AsyncResult instance passed to the handler's handle() method.

## Writing Data

You can write data to the TCP connection via the NetSocket write() method. Here is how that looks:

public class VertxTcpClientVerticle extends AbstractVerticle {

public void start() {

NetClient tcpClient = vertx.createNetClient();

tcpClient.connect(80, "jenkov.com",

new Handler<AsyncResult<NetSocket>>(){

@Override

public void handle(AsyncResult<NetSocket> result) {

NetSocket socket = result.result();

**socket.write("GET / HTTP/1.1\r\nHost: jenkov.com\r\n\r\n");**

}

});

}

}

The write() method is asynchronous and returns immediately. The data may not be sent by the time thewrite() method returns.

The Vert.x NetSocket contains more versions of the write() method which enables you to write e.g. Buffers of data to the NetSocket.

## Reading Data

In order to read data from the NetSocket you need to register a Handler method on the NetSocket. Here is how you do register a Handler on the NetSocket:

public class VertxTcpClientVerticle extends AbstractVerticle {

public void start() {

NetClient tcpClient = vertx.createNetClient();

tcpClient.connect(80, "jenkov.com",

new Handler<AsyncResult<NetSocket>>(){

@Override

public void handle(AsyncResult<NetSocket> result) {

NetSocket socket = result.result();

socket.write("GET / HTTP/1.1\r\nHost: jenkov.com\r\n\r\n");

**socket.handler(new Handler<Buffer>(){**

**@Override**

**public void handle(Buffer buffer) {**

**System.out.println("Received data: " + buffer.length());**

**System.out.println(buffer.getString(0, buffer.length()));**

**}**

**});**

}

});

}

}

The Handler's handle() method will get called when data is received from the remote server.

## Closing the TCP Connection

Once you are finished using the TCP client you need to close it again. You close the TCP client by calling the close() method of the NetClient instance. Here is how that looks:

tcpClient.close();

Again, the NetClient's close() method is asynchronous, so the underlying TCP connection may not yet be closed by the time the close() method returns.

# Vert.x Timers

* [One-time Timers](http://tutorials.jenkov.com/vert.x/timers.html#one-time-timers)
* [Periodic Timers](http://tutorials.jenkov.com/vert.x/timers.html#periodic-timers)
* [Canceling a Timer](http://tutorials.jenkov.com/vert.x/timers.html#canceling-a-timer)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-05-05 |

Vert.x has two types of timers which can be used to time the execution of code. The first timer type is a one-time timer which fires an event after a certain amount of time has passed. The second timer is a period timer which keeps firing events whenever an interval of the specified period has elapsed. This tutorial will cover both types of Vert.x timers in the following sections.

## One-time Timers

A Vert.x one-time timer is a timer that fires a single time after a specified delay. Here is how you create a one-time timer:

long timerID = vertx.setTimer(3000, new Handler<Long>() {

@Override

public void handle(Long aLong) {

System.out.println("Timer fired");

}

});

The setTimer() method takes a time interval in milliseconds as first parameter, and a Handlerimplementation as second parameter. The Handler's handle() method is called when the time interval has elapsed.

The setTimer() method returns a timer ID. This timer ID is also passed as parameter to the Handler's handle() method. This timer ID is also used if you want to cancel the timer later.

## Periodic Timers

A Vert.x periodic timer is a timer that fires every time a specified period of time has passed. Here is how you create a periodic timer:

long timerID = vertx.setPeriodic(3000, new Handler<Long>() {

@Override

public void handle(Long aLong) {

System.out.println("Timer 1 fired: " + aLong);

}

});

The setPeriodic() method takes a time interval in milliseconds as first parameter, and a Handlerimplementation as second parameter. The Handler's handle() method is called every time the time interval has elapsed.

The setPeriodic() returns a timer ID. This timer ID is also passed to the Handler's handle() method whenever it is called. The timer ID is also used if you want to cancel the timer later.

## Canceling a Timer

You can cancel a one-time or periodic timer using the cancelTimer() method of the Vertx object. The cancel() method can cancel both one-time and periodic timers. Here is an example of canceling a Vertx timer:

vertx.cancelTimer(timerID);

# Vert.x Command Line

* [Adding the Vert.x bin Directory to the Path Environment Variable](http://tutorials.jenkov.com/vert.x/command-line.html#adding-vertx-bin-to-path-environment-variable)
* [Testing the Vert.x Command Line](http://tutorials.jenkov.com/vert.x/command-line.html#testing-the-vertx-command-line)
* [Deploying a Java Verticle](http://tutorials.jenkov.com/vert.x/command-line.html#deploying-a-java-verticle)
  + [Longer Casspath](http://tutorials.jenkov.com/vert.x/command-line.html#longer-classpath)
  + [Building the Classpath in a Script](http://tutorials.jenkov.com/vert.x/command-line.html#building-the-classpath-in-a-script)
* [Deploying Verticles Written in Other Languages](http://tutorials.jenkov.com/vert.x/command-line.html#deploying-verticles-written-in-other-languages)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-08-29 |

It is possible to run Vert.x from the command line using the vertx command. The vertx command makes it possible to start up Vert.x and deploy a verticle to it, directly from the command line. That means that you don't have to create a Vertx instance and deploy verticles to it programmatically.

The Vert.x command line is pretty handy in some situations. For instance, the Vert.x command line makes it easy to deploy verticles implemented in different programming languages in the same Vertx instance. This makes Vert.x a true polyglot platform.

## Adding the Vert.x bin Directory to the Path Environment Variable

Before the Vert.x command line works you must have installed Vert.x (download and unzip to the desired directory).

When Vert.x is installed, add the vertx-install-dir/bin to your Path environment variable. That is all that is needed. The vertx-install-dir part should be replaced with the actual directory into which you installed Vert.x.

## Testing the Vert.x Command Line

Once you have added the vertx-install-dir/bin to the path environment variable, open a command prompt and run this command:

vertx -version

If Vert.x is correctly installed you should see an output similar to this:

3.0.0

This is the version of Vert.x that you have installed.

## Deploying a Java Verticle

To deploy (run) a verticle on Vert.x via the Vert.x command line, you must use the vertx run command. After the vertx run part of the command you must write the fully qualified class name of the verticle to deploy. Here is a Vert.x command line Java verticle deployment example:

vertx run com.jenkov.vertx.MyVerticle

This command runs the Java verticle class named com.jenkov.vertx.MyVerticle .

To deploy a verticle class you must run the vertx run classname command from the root directory of the compiled classes for the verticle to deploy. Thus, the root directory of your classes becomes the classpath for deploying the verticle. Whatever directory you run the vertx run command from is automatically added to the classpath when looking for the verticle class.

You can also run a Java verticle class which is located inside a JAR file. Here is how the Vert.x command looks for that:

vertx run com.jenkov.vertx.MyVerticle -cp MyVerticle.jar

### Longer Casspath

Sometimes you need a classpath which contains more than one directory or JAR file. In those situations you can provide a normal classpath string as the -cp argument value. Each entry (directory or JAR file) in the classpath string must be separated by either a semicolon (on Windows) or a colon (on Unix / Linux). Here is a longer classpath example:

vertx run com.jenkov.vertx.MyVerticle -cp MyVerticle.jar;SomeUtilLib.jar

This example contains two JAR files in the classpath separated by a semicolon.

### Building the Classpath in a Script

If you have a very long classpath your command line may get very long, and thus be hard to read in a text editor. Instead you can build the classpath up as an environment variable, one entry per line, and use that environment variable in the command line. I will show you how to do that in both Windows and Linux.

First a Windows .cmd script:

set MY\_CP=build\classes

set MY\_CP=%MY\_CP%;lib\util-1.jar

set MY\_CP=%MY\_CP%;lib\util-2.jar

set MY\_CP=%MY\_CP%;lib\util-3.jar

vertx run com.jenkov.vertx.MyVerticle -cp %MY\_CP%

Save this script as e.g. run.cmd and you can execute it from a Windows command line. Notice how the script builds up an environment variable named MY\_CP and then use that environment variable as argument value for the -cp switch.

Here is the Linux version of the same script:

JAVA\_HOME=/mystuff/java/jdk1.8.0\_60

export JAVA\_HOME

MY\_CP=build/classes

MY\_CP=$MY\_CP:lib/util-1.jar

MY\_CP=$MY\_CP:lib/util-2.jar

MY\_CP=$MY\_CP:lib/util-3.jar

/mystuff/vertx/vert.x-3.0.0/bin/vertx run com.jenkov.vertx.MyVerticle -cp $MY\_CP

First of all, notice that the script creates a JAVA\_HOME environment variable. Vert.x seems to need this environment variable set to run on Linux. Probably because Vert.x uses the Java SDK to run the Vert.x platform and possibly some of the other tools in the Java SDK. Make sure the JAVA\_HOME point to the install directory of your Java SDK.

Second, the script builds a variable named MY\_CP just like the Windows version did (though the code looks different). Notice the use of / as directory separator and the use of : as classpath entry separator.

Third, the script runs Vert.x, passing the MY\_CP variable as argument value for the -cp switch. Notice how the command line contains the full path to where the vertx script is located (inside the Vert.x install directory). By referencing the vertx command using a full path to its location, you do not need to add the vertx command to your path environment variable (I have only tested this on Linux - not on Windows).

## Deploying Verticles Written in Other Languages

As I mentioned in the [**Vert.x introduction**](http://tutorials.jenkov.com/vert.x/index.html) - Vert.x is a polyglot platform. That means that you can deploy verticles written in other languages than Java. You can even deploy all these verticles into the same running instance of Vert.x .

Here is an example of deploying a verticle written in JavaScript:

vertx run my-verticle.js

As you can see, you just list the name of the verticle's JavaScript file and Vert.x will deploy it. It works the same for other languages like Ruby, Python etc. Vert.x will guess what language the file is written in based on the file extension.

If you want to you can tell Vert.x explicitly what language the file is implemented in. That might be useful if you deploy a class file that is written in Groovy or Scala (but which are compiled into Java classes). You tell Vert.x the language by prefixing the file name with the language, like this:

vertx run javascript:my-verticle.js

Notice the javascript: in front of the file name.