# My first Vert.x 3 Application

Let's say, you heard someone saying that Vert.x is *awesome*. Ok great, but you may want to try it by yourself. Well, the next natural question is "where do I start ?". This post is a good starting point. It shows how is built a very simple vert.x application (nothing fancy), how it is tested and how it is packaged and executed. So, everything you need to know before building your own groundbreaking application.

# Let's start !

First, let's create a project. In this post, we use Apache Maven, but you can use Gradle or the build process tool you prefer. You could use the Maven jar archetype to create the structure, but basically, you just need a directory with:

1. a `src/main/java` directory
2. a `src/test/java` directory
3. a `pom.xml` file

So, you would get something like:

```
.
├── pom.xml
├── src
│   ├── main
│   │   └── java
│   └── test
│       └── java
```

Let's create the `pom.xml` file with the following content:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
              http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>io.vertx.blog</groupId>
 <artifactId>my-first-app</artifactId>
 <version>1.0-SNAPSHOT</version>
 <dependencies>
  <dependency>
   <groupId>io.vertx</groupId>
   <artifactId>vertx-core</artifactId>
   <version>3.0.0</version>
  </dependency>
 </dependencies>
```

```
 <build>
  <plugins>
   <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.3</version>
    <configuration>
     <source>1.8</source>
     <target>1.8</target>
    </configuration>
   </plugin>
  </plugins>
 </build>
</project>
```

This `pom.xml` file is pretty straightforward:

- it declares a dependency on `vertx-core`
- it configures the *maven-compiler-plugin* to use Java 8.

This second point is important, Vert.x applications require Java 8.

# Let's code !

Ok, now we have made the `pom.xml` file. Let's do some real coding… Create the `src/main/java/io/vertx/blog/first/MyFirstVerticle.java` file with the following content:

```
package io.vertx.blog.first;

import io.vertx.core.AbstractVerticle;
import io.vertx.core.Future;

public class MyFirstVerticle extends AbstractVerticle {

 @Override
 public void start(Future<Void> fut) {
  vertx
    .createHttpServer()
    .requestHandler(r -> {
     r.response().end("<h1>Hello from my first " +
        "Vert.x 3 application</h1>");
    })
    .listen(8080, result -> {
     if (result.succeeded()) {
      fut.complete();
     } else {
```

```
        fut.fail(result.cause());
      }
    });
  }
}
```

This is actually our not fancy application. The class extends `AbstractVerticle`. In the Vert.x world, a *verticle* is a component. By extending `AbstractVerticle`, our class gets access to the `vertx` field.

The `start` method is called when the verticle is deployed. We could also implement a `stop` method, but in this case Vert.x takes care of the garbage for us. The `start` method receives a `Future` object that will let us inform Vert.x when our start sequence is completed or report an error. One of the particularity of Vert.x is its asynchronous / non-blocking aspect. When our verticle is going to be deployed it won't wait until the start method has been completed. So, the `Future` parameter is important to notify of the completion.

The `start` method creates a HTTP server and attaches a request handler to it. The request handler is a lambda, passed in the `requestHandler` method, called every time the server receives a request. Here, we just reply `Hello ...` (nothing fancy I told you). Finally, the server is bound to the 8080 port. As this may fails (because the port may already be used), we pass another lambda expression checking whether or not the connection has succeeded. As mentioned above it calls either `fut.complete` in case of success or `fut.fail` to report an error.

Let's try to compile the application using:

```
mvn clean compile
```

Fortunately, it should succeed.

That's all for the application.

# Let's test

Well, that's good to have developed an application, but we can never be too careful, so let's test it. The test uses JUnit and vertx-unit - a framework delivered with vert.x to make the testing of vert.x application more natural.

Open the `pom.xml` file to add the two following dependencies:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-unit</artifactId>
```

```
  <version>3.0.0</version>
  <scope>test</scope>
</dependency>
```

Now create the src/test/java/io/vertx/blog/first/MyFirstVerticleTest.java with the following content:

```
package io.vertx.blog.first;

import io.vertx.core.Vertx;
import io.vertx.ext.unit.Async;
import io.vertx.ext.unit.TestContext;
import io.vertx.ext.unit.junit.VertxUnitRunner;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(VertxUnitRunner.class)
public class MyFirstVerticleTest {

  private Vertx vertx;

  @Before
  public void setUp(TestContext context) {
    vertx = Vertx.vertx();
    vertx.deployVerticle(MyFirstVerticle.class.getName(),
        context.asyncAssertSuccess());
  }

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

  @Test
  public void testMyApplication(TestContext context) {
    final Async async = context.async();

    vertx.createHttpClient().getNow(8080, "localhost", "/",
      response -> {
       response.handler(body -> {
        context.assertTrue(body.toString().contains("Hello"));
        async.complete();
       });
    });
  }
```

}
This is a JUnit test for our verticle. The test uses vertx-unit, so we use a custom runner. vert.x-unit makes easy to test asynchronous interactions, which are the basis of vert.x applications.

In the `setUp` method, we creates an instance of `Vertx` and deploy our verticle. You may have noticed that unlike the traditional JUnit `@Before` method, it receives a `TestContext`. This object lets us control the asynchronous aspect of our test. For instance, when we deploy our verticle, it starts asynchronously, as most Vert.x interactions. We cannot check anything until it gets started correctly. So, as second argument of the `deployVerticle` method, we pass a result handler: `context.asyncAssertSuccess()`. It fails the test if the verticle does not start correctly. In addition it waits until the verticle has completed its start sequence. Remember, in our verticle, we call `fut.complete()`. So it waits until this method is called, and in the case of a failures, fails the test.

Well, the `tearDown` method is straightforward, and just terminates the `vertx` instance we created.

Let's now have a look to the test of our application: the `testMyApplication` method. The test emits a request to our application and checks the result. Emitting the request and receiving the response is asynchronous. So we need a way to control this. As the `setUp` and `tearDown` methods, the test method receives a `TestContext`. From this object we creates an *async handle* (`async`) that lets us notify the test framework when the test has completed (using `async.complete()`).

So, once the *async handle* is created, we create a HTTP client and emits a HTTP request handled by our application with the `getNow()` method (`getNow` is just a shortcut for `get(...).end()`). The response is handled by a lambda. In this lambda we retrieves the response body by passing another lambda to the `handler` method. The `body` argument is the response body (as a `buffer` object). We check that the body contains the `"Hello"` String and declare the test complete.

Let's take a minute to mention the *assertions*. Unlike in traditional JUnit tests, it uses `context.assert...`. Indeed, if the assertion fails, it will interrupt the test immediately. So it's pretty important to always uses these assertion methods because of the asynchronous aspect of the Vert.x application and so tests.

Our test can be run from an IDE, or using Maven:

```
mvn clean test
```

# Packaging

So, let's sum up. We have an application and a test. Well, let's now package the application. In this post we package the application in a *fat jar*. A *fat jar* is a standalone executable Jar file containing all the dependencies required to run the application. This is a very convenient way to package Vert.x applications as it's only one file. It also make them easy to execute.

To create a *fat jar*, edit the `pom.xml` file and add the following snippet just before `</plugins>`:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-shade-plugin</artifactId>
 <version>2.3</version>
 <executions>
  <execution>
   <phase>package</phase>
   <goals>
    <goal>shade</goal>
   </goals>
   <configuration>
    <transformers>
     <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer
">
       <manifestEntries>
        <Main-Class>io.vertx.core.Starter</Main-Class>
        <Main-Verticle>io.vertx.blog.first.MyFirstVerticle</Main-Verticle>
       </manifestEntries>
      </transformer>
    </transformers>
    <artifactSet/>
    <outputFile>${project.build.directory}/${project.artifactId}-${project.version}-
fat.jar</outputFile>
   </configuration>
  </execution>
 </executions>
</plugin>
```

It uses the maven-shade-plugin to create the `fat jar`. In the `manifestEntries` it indicates the name of our verticle. You may wonder from where comes the `Starter` class. It's actually a class from vert.x, that is going to create the `vertx` instance and deploy our verticle.

So, with this plugin configured, let's launch:

```
mvn clean package
```

This is going to create `target/my-first-app-1.0-SNAPSHOT-fat.jar` embedding our application along with all the dependencies (including vert.x itself).

# Executing our application

Well, it's nice to have a *fat jar*, but we want to see our application running! As said above, thanks to the *fat jar* packaging, running Vert.x application is easy as:

```
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar
```

Then, open a browser to http://localhost:8080.

To stop the application, hit `CTRL+C`.

# Conclusion

This Vert.x 3 crash class has presented how you can develop a simple application using Vert.x 3, how to test it, package it and run it. So, you now know everything you need to build amazing system on top of Vert.x 3. Next time we will see how to configure our application.

# Vert.x Application Configuration

## Previously in 'Introduction to Vert.x'

In this lab, we developed a very simple Vert.x 3 application, and saw how this application can be tested, packaged and executed. That was nice, isn't it ? Well, ok, that was only the beginning. In this post, we are going to enhance our application to support *external* configuration.

So just to remind you, we have an application starting a HTTP server on the port 8080 and replying a polite "Hello" message to all HTTP requests.

## So, why do we need configuration?

That's a good question. The application works right now, but well, let's say you want to deploy it on a machine where the port 8080 is already taken. We would need to change the port in the application code and in the test, just for this machine. That would be sad. Fortunately, Vert.x applications are configurable.

Vert.x configurations are using the JSON format, so don't expect anything complicated. They can be passed to verticle either from the command line, or using an API. Let's have a look.

## No '8080' anymore

The first step is to modify the `io.vertx.blog.first.MyFirstVerticle` class to not bind to the port 8080, but to read it from the configuration:

```
public void start(Future<Void> fut) {
  vertx
      .createHttpServer()
      .requestHandler(r -> {
        r.response().end("<h1>Hello from my first " +
            "Vert.x 3 application</h1>");
      })
      .listen(
          // Retrieve the port from the configuration,
          // default to 8080.
          config().getInteger("http.port", 8080),
          result -> {
```

```
        if (result.succeeded()) {
          fut.complete();
        } else {
          fut.fail(result.cause());
        }
      }
    );
}
```

So, the only difference with the previous version is `config().getInteger("http.port", 8080)`. Here, our code is now requesting the configuration and check whether the *http.port* property is set. If not, the port 8080 is used as fall-back. The retrieved configuration is a `JsonObject`.

As we are using the port 8080 by default, you can still package our application and run it as before:

```
mvn clean package
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar
```

Simple right ?

# API-based configuration - Random port for the tests

Now that the application is configurable, let's try to provide a configuration. In our test, we are going to configure our application to use the port 8081. So, previously we were deploying our verticle with:

```
vertx.deployVerticle(MyFirstVerticle.class.getName(), context.asyncAssertSuccess());
```

Let's now pass some *deployment options*:

```
port = 8081;
DeploymentOptions options = new DeploymentOptions()
   .setConfig(new JsonObject().put("http.port", port)
);
vertx.deployVerticle(MyFirstVerticle.class.getName(), options,
context.asyncAssertSuccess());
```

The `DeploymentOptions` object lets us customize various parameters. In particular, it lets us inject the `JsonObject` retrieved by the verticle when using the `config()` method.

Obviously, the test connecting to the server needs to be slightly modified to use the right port (`port` is a field):

```
vertx.createHttpClient().getNow(port, "localhost", "/", response -> {
  response.handler(body -> {
```

```
    context.assertTrue(body.toString().contains("Hello"));
    async.complete();
  });
});
```

Ok, well, this does not really fix our issue. What happens when the port 8081 is used too. Let's now pick a random port:

```
ServerSocket socket = new ServerSocket(0);
port = socket.getLocalPort();
socket.close();

DeploymentOptions options = new DeploymentOptions()
    .setConfig(new JsonObject().put("http.port", port)
    );

vertx.deployVerticle(MyFirstVerticle.class.getName(), options,
context.asyncAssertSuccess());
```

So, the idea is very simple. We open a *server socket* that would pick a random port (that's why we put 0 as parameter). We retrieve the used port and close the socket. Be aware that this method is **not** perfect and may fail if the picked port becomes used between the `close`method and the start of our HTTP server. However, it would work fine in the very high majority of the case.

With this in place, our test is now using a random port. Execute them with:

```
mvn clean test
```

# External configuration - Let's run on another port

Ok, well random port is not what we want in *production*. Could you imagine the face of your production team if you tell them that your application is picking a random port. It can actually be funny, but we should never mess with the production team.

So for the actual execution of your application, let's pass the configuration in an external file. The configuration is stored in a *json* file.

Create the `src/main/conf/my-application-conf.json` with the following content:

```
{
  "http.port" : 8082
}
```

And now, to use this configuration just launch your application with:

```
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar -conf src/main/conf/my-application-conf.json
```

Open a browser on http://localhost:8082, here it is !

How does that work ? Remember, our *fat jar* is using the `Starter` class (provided by Vert.x) to launch our application. This class is reading the `-conf` parameter and create the corresponding deployment options when deploying our verticle.

# Conclusion

After having developed your first Vert.x application, we have seen how this application is configurable, and this without adding any complexity to our application.

# Some Rest with Vert.x

Well, nothing fancy… Let's go a bit further this time and develop a *CRUD-ish* application. So an application exposing an HTML page interacting with the backend using a REST API. The level of *RESTfullness* of the API is not the topic of this post, I let you decide as it's a very slippery topic.

So, in other words we are going to see:

- Vert.x Web - a framework that let you create Web applications easily using Vert.x
- How to expose static resources
- How to develop a REST API

# Vert.x Web

As you may have notices in the previous posts, dealing with complex HTTP application using only Vert.x Core would be kind of cumbersome. That's the main reason behind Vert.x Web. It makes the development of Vert.x base web applications really easy, without changing the philosophy.

To use Vert.x Web, you need to update the `pom.xml` file to add the following dependency:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web</artifactId>
  <version>3.0.0</version>
</dependency>
```

That's the only thing you need to use Vert.x Web. Sweet, no ?

Let's now use it. Remember, in the previous post, when we requested http://localhost:8080, we reply a nice *Hello World* message. Let's do the same with Vert.x Web. Open the `io.vertx.blog.first.MyFirstVerticle` class and change the `start` method to be:

```
@Override
public void start(Future<Void> fut) {
 // Create a router object.
 Router router = Router.router(vertx);

 // Bind "/" to our hello message - so we are still compatible.
```

```
router.route("/").handler(routingContext -> {
  HttpServerResponse response = routingContext.response();
  response
      .putHeader("content-type", "text/html")
      .end("<h1>Hello from my first Vert.x 3 application</h1>");
});

// Create the HTTP server and pass the "accept" method to the request handler.
vertx
    .createHttpServer()
    .requestHandler(router::accept)
    .listen(
        // Retrieve the port from the configuration,
        // default to 8080.
        config().getInteger("http.port", 8080),
        result -> {
          if (result.succeeded()) {
            fut.complete();
          } else {
            fut.fail(result.cause());
          }
        }
    );
}
```

You may be surprise by the length of this snippet (in comparison to the previous code). But as we are going to see, it will make our app on steroids, just be patient.

As you can see, we start by creating a `Router` object. The router is the cornerstone of Vert.x Web. This object is responsible for dispatching the HTTP requests to the right *handler*. Two other concepts are very important in Vert.x Web:

- Routes - which let you define how request are dispatched
- Handlers - which are the actual action processing the requests and writing the result. Handlers can be chained.

If you understand these 3 concepts, you have understood everything in Vert.x Web.

Let's focus on this code first:

```
router.route("/").handler(routingContext -> {
  HttpServerResponse response = routingContext.response();
  response
      .putHeader("content-type", "text/html")
      .end("<h1>Hello from my first Vert.x 3 application</h1>");
```

```
});
```

It *routes* requests arriving on "/" to the given *handler*. Handlers receive a `RoutingContext` object. This handler is quite similar to the code we had before, and it's quite normal as it manipulates the same type of object: `HttpServerResponse`.

Let's now have a look to the rest of the code:

```
vertx
    .createHttpServer()
    .requestHandler(router::accept)
    .listen(
        // Retrieve the port from the configuration,
        // default to 8080.
        config().getInteger("http.port", 8080),
        result -> {
          if (result.succeeded()) {
            fut.complete();
          } else {
            fut.fail(result.cause());
          }
        }
    );
}
```

It's basically the same code as before, except that we change the request handler. We pass `router::accept` to the handler. You may not be familiar with this notation. It's a reference to a method (here the method `accept` from the `router` object). In other worlds, it instructs vert.x to call the `accept` method of the `router` when it receives a request.

Let's try to see if this work:

```
mvn clean package
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar
```

By opening `http://localhost:8080` in your browser you should see the *Hello* message. As we didn't change the behavior of the application, our tests are still valid.

# Exposing static resources

Ok, so we have a first application using vert.x web. Let's see some of the benefits. Let's start with serving static resources, such as an `index.html` page. Before we go further, I should start with a disclaimer: "the HTML page we are going to see here is ugly like hell : I'm not a UI guy". I should also add that there are probably plenty of better ways to implement this and a myriad of frameworks I should try, but that's not the point. I tried to keep things simple and just relying on JQuery and Twitter Bootstrap, so if you know a bit of JavaScript you can understand and edit the page.

Let's create the HTML page that will be the entry point of our application. Create an `index.html` page in `src/main/resources/assets`with the content from here. As it's just a HTML page with a bit of JavaScript, we won't detail the file here. If you have questions, just post comments.

Basically, the page is a simple *CRUD* UI to manage my collection of *not-yet-finished* bottles of Whisky. It was made in a generic way, so you can transpose it to your own collection. The list of product is displayed in the main table. You can create a new product, edit one or delete one. These actions are relying on a REST API (that we are going to implement) through AJAX calls. That's all.

Once this page is created, edit the `io.vertx.blog.first.MyFirstVerticle` class and change the `start` method to be:

```
@Override
public void start(Future<Void> fut) {
 Router router = Router.router(vertx);
 router.route("/").handler(routingContext -> {
   HttpServerResponse response = routingContext.response();
   response
      .putHeader("content-type", "text/html")
      .end("<h1>Hello from my first Vert.x 3 application</h1>");
 });

 // Serve static resources from the /assets directory
 router.route("/assets/*").handler(StaticHandler.create("assets"));

 vertx
    .createHttpServer()
    .requestHandler(router::accept)
    .listen(
       // Retrieve the port from the configuration,
       // default to 8080.
       config().getInteger("http.port", 8080),
```

```
      result -> {
        if (result.succeeded()) {
          fut.complete();
        } else {
          fut.fail(result.cause());
        }
      }
    );
}
```

The only difference with the previous code is the `router.route("/assets/*").handler(StaticHandler.create("assets"));` line. So, what does this line mean? It's actually quite simple. It *routes* requests on "/assets/*" to resources stored in the "assets" directory. So our `index.html` page is going to be served using http://localhost:8080/assets/index.html.

Before testing this, let's take a few seconds on the handler creation. All processing actions in Vert.x web are implemented as *handler*. To create a handler you always call the `create` method.

So, I'm sure you are impatient to see our beautiful HTML page. Let's build and run the application:

```
mvn clean package
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar
```

Now, open your browser to http://localhost:8080/assets/index.html. Here it is… Ugly right? I told you.

As you may notice too… the table is empty, this is because we didn't implement the REST API yet. Let's do that now.

# REST API with Vert.x Web

Vert.x Web makes the implementation of REST API really easy, as it basically *routes* your URL to the right handler. The API is very simple, and will be structured as follows:

- `GET /api/whiskies` => get all bottles (`getAll`)
- `GET /api/whiskies/:id` => get the bottle with the corresponding id (`getOne`)
- `POST /api/whiskies` => add a new bottle (`addOne`)

- `PUT /api/whiskies/:id` => update a bottle (`updateOne`)
- `DELETE /api/whiskies/id` => delete a bottle (`deleteOne`)

# We need some data…

But before going further, let's create our *data* object. Create the `src/main/java/io/vertx/blog/first/Whisky.java` with the following content:

```
package io.vertx.blog.first;

import java.util.concurrent.atomic.AtomicInteger;

public class Whisky {

  private static final AtomicInteger COUNTER = new AtomicInteger();

  private final int id;

  private String name;

  private String origin;

  public Whisky(String name, String origin) {
    this.id = COUNTER.getAndIncrement();
    this.name = name;
    this.origin = origin;
  }

  public Whisky() {
    this.id = COUNTER.getAndIncrement();
  }

  public String getName() {
    return name;
  }

  public String getOrigin() {
    return origin;
  }

  public int getId() {
    return id;
  }

  public void setName(String name) {
```

```
    this.name = name;
  }

  public void setOrigin(String origin) {
    this.origin = origin;
  }
}
```

It's a very simple *bean* class (so with getters and setters). We choose this format because Vert.x is relying on Jackson to handle the JSON format. Jackson automates the serialization and deserialization of *bean* classes, making our code much simpler.

Now, let's create a couple of bottles. In the `MyFirstVerticle` class, add the following code:

```
// Store our product
private Map<Integer, Whisky> products = new LinkedHashMap<>();
// Create some product
private void createSomeData() {
  Whisky bowmore = new Whisky("Bowmore 15 Years Laimrig", "Scotland, Islay");
  products.put(bowmore.getId(), bowmore);
  Whisky talisker = new Whisky("Talisker 57° North", "Scotland, Island");
  products.put(talisker.getId(), talisker);
}
```

Then, in the `start` method, call the `createSomeData` method:

```
@Override
public void start(Future<Void> fut) {

  createSomeData();

  // Create a router object.
  Router router = Router.router(vertx);

  // Rest of the method
}
```

As you have noticed, we don't really have a *backend* here, it's just a (in-memory) map. Adding a backend will be covered by another post.

# Get our products

Enough decoration, let's implement the REST API. We are going to start with `GET /api/whiskies`. It returns the list of bottles in a JSON Array.

In the `start` method, add this line just below the static handler line:

```
router.get("/api/whiskies").handler(this::getAll);
```

This line instructs the `router` to handle the `GET` requests on "/api/whiskies" by calling the `getAll` method. We could have inlined the handler code, but for clarity reasons let's create another method:

```
private void getAll(RoutingContext routingContext) {
  routingContext.response()
    .putHeader("content-type", "application/json; charset=utf-8")
    .end(Json.encodePrettily(products.values()));
}
```

As every *handler* our method receives a `RoutingContext`. It populates the `response` by setting the `content-type` and the actual content. Because our content may contain *weird* characters, we force the charset to UTF-8. To create the actual content, no need to compute the JSON string ourself. Vert.x lets us use the `Json` API.
So `Json.encodePrettily(products.values())` computes the JSON string representing the set of bottles.

We could have used `Json.encodePrettily(products)`, but to make the JavaScript code simpler, we just return the set of bottles and not an object containing `ID => Bottle` entries.

With this in place, we should be able to retrieve the set of bottle from our HTML page. Let's try it:

```
mvn clean package
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar
```

Then open the HTML page in your browser (`http://localhost:8080/assets/index.html`), and should should see:


I'm sure you are curious, and want to actually see what is returned by our REST API. Let's open a browser to `http://localhost:8080/api/whiskies`. You should get:

```
[ {
  "id" : 0,
```

```
  "name" : "Bowmore 15 Years Laimrig",
  "origin" : "Scotland, Islay"
}, {
  "id" : 1,
  "name" : "Talisker 57° North",
  "origin" : "Scotland, Island"
} ]
```

# Create a product

Now we can retrieve the set of bottles, let's create a new one. Unlike the previous REST API endpoint, this one need to read the request's body. For performance reason, it should be explicitly enabled. Don't be scared… it's just a handler.

In the `start` method, add these lines just below the line ending by `getAll`:

```
router.route("/api/whiskies*").handler(BodyHandler.create());
router.post("/api/whiskies").handler(this::addOne);
```

The first line enables the reading of the request body for all routes under "/api/whiskies". We could have enabled it globally with `router.route().handler(BodyHandler.create())`.

The second line maps `POST` requests on `/api/whiskies` to the `addOne` method. Let's create this method:

```
private void addOne(RoutingContext routingContext) {
  final Whisky whisky = Json.decodeValue(routingContext.getBodyAsString(),
    Whisky.class);
  products.put(whisky.getId(), whisky);
  routingContext.response()
    .setStatusCode(201)
    .putHeader("content-type", "application/json; charset=utf-8")
    .end(Json.encodePrettily(whisky));
}
```

The method starts by retrieving the `Whisky` object from the request body. It just reads the body into a String and passes it to the `Json.decodeValue` method. Once created it adds it to the *backend* map and returns the created bottle as JSON.

Let's try this. Rebuild and restart the application with:

```
mvn clean package
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar
```

Then, refresh the HTML page and click on the `Add a new bottle` button. Enter the data such as: "Jameson" as name and "Ireland" as origin (purists

would have noticed that this is actually a Whiskey and not a Whisky). The bottle should be added to the table.

> **Status 201 ?**
> As you can see, we have set the response status to `201`. It means `CREATED`, and is the generally used in REST API that create an entity. By default vert.x web is setting the status to `200` meaning `OK`.

# Finishing a bottle

Well, bottles do not last forever, so we should be able to delete a bottle. In the `start` method, add this line:

```
router.delete("/api/whiskies/:id").handler(this::deleteOne);
```

In the URL, we define a *path parameter* `:id`. So, when handling a matching request, Vert.x extracts the path segment corresponding to the parameter and let us access it in the handler method. For instance, `/api/whiskies/0` maps `id` to `0`.

Let's see how the parameter can be used in the handler method. Create the `deleteOne` method as follows:

```
private void deleteOne(RoutingContext routingContext) {
  String id = routingContext.request().getParam("id");
  if (id == null) {
    routingContext.response().setStatusCode(400).end();
  } else {
    Integer idAsInteger = Integer.valueOf(id);
    products.remove(idAsInteger);
  }
  routingContext.response().setStatusCode(204).end();
}
```

The *path parameter* is retrieved using `routingContext.request().getParam("id")`. It checks whether it's `null` (not set), and in this case returns a `Bad Request` response (status code 400). Otherwise, it removes it from the *backend* map.

> **Status 204 ?**
> As you can see, we have set the response status to `204 – NO CONTENT`. Response to the HTTP Verb `delete` have generally no content.

# The other methods

We won't detail `getOne` and `updateOne` as the implementations are straightforward and very similar. Their implementations are available on GitHub.

# Cheers !

It's time to conclude this lab. We have seen how Vert.x Web lets you implement a REST API easily and how it can serve static resources. A bit more fancy than before, but still pretty easy.

# Unit and Integration Tests

## Tests, Tests, Tests…

This lab is mainly about tests. We distinguish two types of tests: unit tests and integration tests. Both are equally important, but have different focus. Unit tests ensure that one *component* of your application, generally a class in the Java world, behaves as expected. The application is not tested as a whole, but pieces by pieces. Integration tests are more *black box* in the sense that the application is started and tested generally externally.

In this post we are going to start with some more unit tests as a warm up session and then focus on integration tests. If you already implemented integration tests, you may be a bit scared, and it makes sense. But don't worry, with Vert.x there are no hidden surprises.

## Warmup: Some more unit tests

Let's start slowly. Remember in the first post we have implemented a unit test with vertx-unit. The test we did is dead simple:

1. we started the application before the test
2. we checks that it replies "Hello"

Just to refresh your mind, let's have a look at the code

```
@Before
public void setUp(TestContext context) throws IOException {
  vertx = Vertx.vertx();
  ServerSocket socket = new ServerSocket(0);
  port = socket.getLocalPort();
  socket.close();
  DeploymentOptions options = new DeploymentOptions()
     .setConfig(new JsonObject().put("http.port", port)
     );
  vertx.deployVerticle(MyFirstVerticle.class.getName(), options,
context.asyncAssertSuccess());
}
```

The `setUp` method is invoked before each test (as instructed by the `@Before` annotation). It, first, creates a new instance of Vert.x. Then, it gets a free port and then deploys our verticle with the right configuration. Thanks to the `context.asyncAssertSuccess()` it waits until the successful deployment of the verticle.

The `tearDown` is straightforward and just closes the Vert.x instance. It automatically un-deploys the verticles:

```
@After
```

```
public void tearDown(TestContext context) {
  vertx.close(context.asyncAssertSuccess());
}
```

Finally, our single test is:

```
@Test
public void testMyApplication(TestContext context) {
  final Async async = context.async();
  vertx.createHttpClient().getNow(port, "localhost", "/", response -> {
    response.handler(body -> {
      context.assertTrue(body.toString().contains("Hello"));
      async.complete();
    });
  });
 }
```

It is only checking that the application replies "Hello" when we emit a HTTP request on `/`. Let's now try to implement some unit tests checkin that our web application and the REST API behave as expected. Let's start by checking that the `index.html` page is correctly served. This test is very similar to the previous one:

```
@Test
public void checkThatTheIndexPageIsServed(TestContext context) {
  Async async = context.async();
  vertx.createHttpClient().getNow(port, "localhost", "/assets/index.html", response -> {
    context.assertEquals(response.statusCode(), 200);
    context.assertEquals(response.headers().get("content-type"), "text/html");
    response.bodyHandler(body -> {
      context.assertTrue(body.toString().contains("<title>My Whisky Collection</title>"));
      async.complete();
    });
  });
}
```

We retrieve the `index.html` page and check:

1. it's there (status code 200)
2. it's a HTML page (content type set to "text/html")
3. it has the right title ("My Whisky Collection")

> **Retrieving content**
> As you can see, we can test the status code and the headers directly on the HTTP response, but ensure that the body is right, we need to retrieve it. This is done with a body handler that receives the complete body as parameter. Once the last check is made, we release the `async` by calling `complete`.

Ok, great, but this actually does not test our REST API. Let's ensure that we can add a bottle to the collection. Unlike the previous tests, this one is using `post` to *post* data to the server:

```
@Test
```

```
public void checkThatWeCanAdd(TestContext context) {
  Async async = context.async();
  final String json = Json.encodePrettily(new Whisky("Jameson", "Ireland"));
  final String length = Integer.toString(json.length());
  vertx.createHttpClient().post(port, "localhost", "/api/whiskies")
      .putHeader("content-type", "application/json")
      .putHeader("content-length", length)
      .handler(response -> {
        context.assertEquals(response.statusCode(), 201);
        context.assertTrue(response.headers().get("content-
type").contains("application/json"));
        response.bodyHandler(body -> {
          final Whisky whisky = Json.decodeValue(body.toString(), Whisky.class);
          context.assertEquals(whisky.getName(), "Jameson");
          context.assertEquals(whisky.getOrigin(), "Ireland");
          context.assertNotNull(whisky.getId());
          async.complete();
        });
      })
      .write(json)
      .end();
}
```

First we create the content we want to add. The server consumes JSON data, so we need a JSON string. You can either write your JSON document manually, or use the Vert.x method (`Json.encodePrettily`) as done here. Once we have the content, we create a `post` request. We need to configure some headers to be correctly read by the server. First, we say that we are sending JSON data and we also set the content length. We also attach a response handler very close to the checks made in the previous test. Notice that we can rebuild our object from the JSON document send by the server using the `JSON.decodeValue` method. It's very convenient as it avoids lots of boilerplate code. At this point the request is not emitted, we need to write the data and call the `end()` method. This is made using `.write(json).end();`.

The order of the methods is important. You cannot *write* data if you don't have a response handler configured. Finally don't forget to call `end`.

So, let's try this. You can run the test using:

```
mvn clean test
```

We could continue writing more unit test like that, but it could become quite complex. Let's see how we could continue our tests using integration tests.

# IT hurts

Well, I think we need to make that clear, integration testing hurts. If you have experience in this area, can you remember how long did it take to setup everything correctly? I get new

white hairs by just thinking about it. Why are integration tests more complicated? It's basically because of the setup:

1. We must start the application in a *close to production* way
2. We must then run the tests (and configure them to hit the right application instance)
3. We must stop the application

That does not sound unconquerable like that, but if you need Linux, MacOS X and Windows support, it quickly get messy. There are plenty of great frameworks easing this such as Arquillian, but let's do it without any framework to understand how it works.

# We need a battle plan

Before rushing into the complex configuration, let's think a minute about the tasks:

**Step 1 - Reserve a free port** We need to get a free port on which the application can *listen*, and we need to inject this port in our integration tests.

**Step 2 - Generate the application configuration** Once we have the free port, we need to write a JSON file configuring the application HTTP Port to this port.

**Step 3 - Start the application** Sounds easy right? Well it's not that simple as we need to launch our application in a background process.

**Step 4 - Execute the integration tests** Finally, the central part, run the tests. But before that we should implement some integration tests. Let's come to that later.

**Step 5 - Stop the application** Once the tests have been executed, regardless if there are failures or errors in the tests, we need to stop the application.

There are multiple way to implement this plan. We are going to use a *generic* way. It's not necessarily the better, but can be applied almost everywhere. The approach is tight to Apache Maven. If you want to propose an alternative using Gradle or a different tool, I will be happy to add your way to the post.

# Implement the plan

As said above, this section is Maven-centric, and most of the code goes in the pom.xml file. If you never used the different Maven lifecycle phases, I recommend you to look at the introduction to the Maven lifecycle.

We need to add and configure a couple of plugins. Open the `pom.xml` file and in the `<plugins>` section add:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>1.9.1</version>
  <executions>
```

```
   <execution>
    <id>reserve-network-port</id>
    <goals>
     <goal>reserve-network-port</goal>
    </goals>
    <phase>process-sources</phase>
    <configuration>
     <portNames>
      <portName>http.port</portName>
     </portNames>
    </configuration>
   </execution>
  </executions>
</plugin>
```

We use the `build-helper-maven-plugin` (a plugin to know if you are often using Maven) to pick up a free port. Once found, the plugin assigns the `http.port` variable to the picked port. We execute this plugin early in the build (during the `process-sources` phase), so we can use the `http.port` variable in the other plugin. This was for the first step.

Two actions are required for the second step. First, in the `pom.xml` file, just below the `<build>` opening tag, add:

```
<testResources>
 <testResource>
  <directory>src/test/resources</directory>
  <filtering>true</filtering>
 </testResource>
</testResources>
```

This instructs Maven to *filter* resources from the `src/test/resources` directory. *Filter* means replacing placeholders by actual values. That's exactly what we need as we now have the `http.port` variable. So create the `src/test/resources/my-it-config.json` file with the following content:

```
{
 "http.port": ${http.port}
}
```

This configuration file is similar to the one we did in previous posts. The only difference is the `${http.port}` which is the (default) Maven syntax for filtering. So, when Maven is going to process or file it will replace `${http.port}` by the selected port. That's all for the second step.

The step 3 and 5 are a bit more tricky. We should start and stop the application. We are going to use the `maven-antrun-plugin` to achieve this. In the `pom.xml` file, below the `build-helper-maven-plugin`, add:

```
<!-- We use the maven-antrun-plugin to start the application before the integration tests and stop them afterward -->
```

```
<plugin>
 <artifactId>maven-antrun-plugin</artifactId>
 <version>1.8</version>
 <executions>
  <execution>
   <id>start-vertx-app</id>
   <phase>pre-integration-test</phase>
   <goals>
    <goal>run</goal>
   </goals>
   <configuration>
    <target>
     <!--
     Launch the application as in 'production' using the fatjar.
     We pass the generated configuration, configuring the http port to the picked one
     -->
     <exec executable="${java.home}/bin/java"
         dir="${project.build.directory}"
         spawn="true">
      <arg value="-jar"/>
      <arg value="${project.artifactId}-${project.version}-fat.jar"/>
      <arg value="-conf"/>
      <arg value="${project.build.directory}/test-classes/my-it-config.json"/>
     </exec>
    </target>
   </configuration>
  </execution>
  <execution>
   <id>stop-vertx-app</id>
   <phase>post-integration-test</phase>
   <goals>
    <goal>run</goal>
   </goals>
   <configuration>
    <!--
     Kill the started process.
     Finding the right process is a bit tricky. Windows command in in the windows profile
(below)
     -->
    <target>
     <exec executable="bash"
         dir="${project.build.directory}"
         spawn="false">
      <arg value="-c"/>
      <arg value="ps ax | grep -Ei '[\-]DtestPort=${http.port}\s+\-jar\s+${project.artifactId}'
| awk 'NR==1{print $1}' | xargs kill -SIGTERM"/>
     </exec>
```

```
      </target>
     </configuration>
    </execution>
   </executions>
</plugin>
```

That's a huge piece of XML, isn't it ? We configure two executions of the plugin. The first one, happening in the `pre-integration-test`phase, executes a set of bash command to start the application. It basically executes:

```
java -jar my-first-app-1.0-SNAPSHOT-fat.jar -conf .../my-it-config.json
```

> **Is the fatjar created ?**
> The fat jar embedding our application is created in the `package` phase, preceding the `pre-integration-test`, so yes, the fat jar is created.

As mentioned above, we launch the application as we would in a production environment.

Once, the integration tests are executed (step 4 we didn't look at it yet), we need to stop the application (so in the the `post-integration-test` phase). To close the application, we are going to invoke some shell magic command to find our process in with the `ps` command and send the `SIGTERM` signal. It is equivalent to:

```
ps
.... -> find your process id
kill your_process_id -SIGTERM
```

> **And Windows ?**
> I mentioned it above, we want Windows to be supported and these commands are not going to work on Windows. Don't worry, Windows configuration is below….

We should now do the fourth step we (silently) skipped. To execute our integration tests, we use the `maven-failsafe-plugin`. Add the following plugin configuration to your `pom.xml` file:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-failsafe-plugin</artifactId>
 <version>2.18.1</version>
 <executions>
  <execution>
   <goals>
    <goal>integration-test</goal>
    <goal>verify</goal>
   </goals>
   <configuration>
    <systemProperties>
     <http.port>${http.port}</http.port>
    </systemProperties>
   </configuration>
```

```
    </execution>
  </executions>
</plugin>
```

As you can see, we pass the `http.port` property as a system variable, so our tests are able to connect on the right port.

That's all! Wow… Let's try this (for windows users, you will need to be patient or to jump to the last section).

```
mvn clean verify
```

We should not use `mvn integration-test` because the application would still be running. The `verify` phase is after the `post-integration-test` phase and will analyse the integration-tests results. Build failures because of integration tests failed assertions are reported in this phase.

# Hey, we don't have integration tests !

And that's right, we set up everything, but we don't have a single integration test. To ease the implementation, let's use two libraries: AssertJ and Rest-Assured.

AssertJ proposes a set of assertions that you can chain and use fluently. Rest Assured is a framework to test REST API.

In the `pom.xml` file, add the two following dependencies just before `</dependencies>`:

```
<dependency>
  <groupId>com.jayway.restassured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>2.4.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>2.0.0</version>
  <scope>test</scope>
</dependency>
```

Then, create the `src/test/java/io/vertx/blog/first/MyRestIT.java` file. Unlike unit test, integration test ends with `IT`. It's a convention from the Failsafe plugin to distinguish unit (starting or ending with *Test*) from integration tests (starting or ending with *IT*). In the created file add:

```
package io.vertx.blog.first;

import com.jayway.restassured.RestAssured;
import org.junit.AfterClass;
```

```
import org.junit.BeforeClass;

public class MyRestIT {

  @BeforeClass
  public static void configureRestAssured() {
    RestAssured.baseURI = "http://localhost";
    RestAssured.port = Integer.getInteger("http.port", 8080);
  }

  @AfterClass
  public static void unconfigureRestAssured() {
    RestAssured.reset();
  }
}
```

The methods annotated with `@BeforeClass` and `@AfterClass` are invoked once before / after all tests of the class. Here, we just retrieve the http port (passed as a system property) and we configure REST Assured.

> **Am I ready to serve ?**
> You may need to wait in the `configureRestAssured` method that the HTTP server has been started. We recommend the awaitilitytest framework to check that the request can be served. It would fail the test if the server does not start.

It's now time to implement a real test. Let's check we can retrieve an individual product:

```
@Test
public void checkThatWeCanRetrieveIndividualProduct() {
  // Get the list of bottles, ensure it's a success and extract the first id.
  final int id = get("/api/whiskies").then()
      .assertThat()
      .statusCode(200)
      .extract()
      .jsonPath().getInt("find { it.name=='Bowmore 15 Years Laimrig' }.id");
  // Now get the individual resource and check the content
  get("/api/whiskies/" + id).then()
      .assertThat()
      .statusCode(200)
      .body("name", equalTo("Bowmore 15 Years Laimrig"))
      .body("origin", equalTo("Scotland, Islay"))
      .body("id", equalTo(id));
}
```

Here you can appreciate the power and expressiveness of Rest Assured. We retrieve the list of product, ensure the response is correct, and extract the *id* of a specific bottle using a JSON (Groovy) Path expression.

Then, we try to retrieve the metadata of this individual product, and check the result.

Let’s now implement a more sophisticated scenario. Let’s add and delete a product:

```
@Test
public void checkWeCanAddAndDeleteAProduct() {
  // Create a new bottle and retrieve the result (as a Whisky instance).
  Whisky whisky = given()
      .body("{\"name\":\"Jameson\",
\"origin\":\"Ireland\"}").request().post("/api/whiskies").thenReturn().as(Whisky.class);
  assertThat(whisky.getName()).isEqualToIgnoringCase("Jameson");
  assertThat(whisky.getOrigin()).isEqualToIgnoringCase("Ireland");
  assertThat(whisky.getId()).isNotZero();
  // Check that it has created an individual resource, and check the content.
  get("/api/whiskies/" + whisky.getId()).then()
      .assertThat()
      .statusCode(200)
      .body("name", equalTo("Jameson"))
      .body("origin", equalTo("Ireland"))
      .body("id", equalTo(whisky.getId()));
  // Delete the bottle
  delete("/api/whiskies/" + whisky.getId()).then().assertThat().statusCode(204);
  // Check that the resource is not available anymore
  get("/api/whiskies/" + whisky.getId()).then()
      .assertThat()
      .statusCode(404);
}
```

So, now we have integration tests let's try:

```
mvn clean verify
```

Simple no? Well, simple once the setup is done right… You can continue implementing other integration tests to be sure that everything behave as you expect.

# Dear Windows users…

This section is the bonus part for Windows user, or people wanting to run their integration tests on Windows machine too. The command we execute to stop the application is not going to work on Windows. Luckily, it’s possible to extend the `pom.xml` with a profile executed on Windows.

In your `pom.xml`, just after `</build>`, add:

```
<profiles>
  <!-- A profile for windows as the stop command is different -->
  <profile>
    <id>windows</id>
    <activation>
```

```
    <os>
      <family>windows</family>
    </os>
  </activation>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.8</version>
        <executions>
          <execution>
            <id>stop-vertx-app</id>
            <phase>post-integration-test</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <target>
                <exec executable="wmic"
                   dir="${project.build.directory}"
                   spawn="false">
                  <arg value="process"/>
                  <arg value="where"/>
                  <arg value="CommandLine like '%${project.artifactId}%' and not
name='wmic.exe'"/>
                  <arg value="delete"/>
                </exec>
              </target>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
 </profile>
</profiles>
```

This profile replaces the actions described above to stop the application with a version working on windows. The profile is automatically enabled on Windows. As on others operating systems, execute with:

```
mvn clean verify
```

# Conclusion

Wow, what a trip ! We are done… In this post we have seen how we can gain confidence in Vert.x applications by implementing both unit and integration tests. Unit tests, thanks to vert.x unit, are able to check the asynchronous aspect of Vert.x application, but could be

complex for large scenarios. Thanks to Rest Assured and AssertJ, integration tests are dead simple to write… but the setup is not straightforward. This post have shown how it can be configured easily. Obviously, you could also use AssertJ and Rest Assured in your unit tests.

# Using the asynchronous SQL client

Finally, back… This post is the fifth post of the introduction to vert.x blog series, after a not-that-small break. In this post we are going to see how we can use JDBC in a vert.x application, and this, using the asynchronous API provided by the vertx-jdbc-client.

# Previously in the introduction to vert.x series

As it was quite some time since the last post, let's start by refreshing our mind about the four previous posts:

1. The first post has described how to build a vert.x application with Maven and execute unit tests.
2. The second post has described how this application can become configurable.
3. The third post has introduced vertx-web, and a small collection management application has been developed. This application offers a REST API used by a HTML/JavaScript frontend.
4. The previous post has presented how you can run integration tests to ensure the behavior of your application.

In this post, back to code. The current application uses an in-memory map to store the products. It's time to use a database. In this post we are going to use HSQL, but you can use any database providing a JDBC driver. Interactions with the database will be asynchronous and made using the vertx-jdbc-client.

The code of this post are available on this Github project, in the post-5 branch branch.

# Asynchronous?

One of the vert.x characteristics is being asynchronous. With an asynchronous API, you don't wait for a result, but you are notified when this result is ready. Just to illustrate this, let's take a very simple example.

Let's imagine an `add` method. Traditionally, you would use it like this: `int r = add(1, 1)`. This is a synchronous API as you are waiting for the result. An asynchronous version of this API would be: `add(1, 1, r -> { /* do something with the result */ })`. In this version, you pass a `Handler` called when the result has been computed. The method does not return anything, and could be implemented as:

```
public void add(int a, int b, Handler<Integer> resultHandler) {
  int r = a + b;
  resultHandler.handle(r);
}
```

Just to avoid misconceptions, asynchronous API are not about threads. As we can see in the `add` example, there are no threads involved.

# JDBC yes, but asynchronous

So, now that we have seen some basics about asynchronous API, let's have a look to the vertx-jdbc-client. This component lets us interact with a database through a JDBC driver. These interactions are asynchronous, so when you were doing:

```
String sql = "SELECT * FROM Products";
ResultSet rs = stmt.executeQuery(sql);
```

it will be:

```
connection.query("SELECT * FROM Products", result -> {
    // do something with the result
});
```

This model is more efficient as it avoids waiting for the result. You are notified when the result is available.

Let's now modify our application to use a database to store our products.

# Some maven dependencies

The first things we need to do it to declare two new Maven dependencies in our `pom.xml` file:

```
<dependency>
 <groupId>io.vertx</groupId>
 <artifactId>vertx-jdbc-client</artifactId>
```

```
  <version>3.1.0</version>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.3.3</version>
</dependency>
```

The first dependency provides the vertx-jdbc-client, while the second one provide the HSQL JDBC driver. If you want to use another database, change this dependency. You will also need to change the JDBC url and JDBC driver class name later.

# Initializing the JDBC client

Now that we have added these dependencies, it's time to create our JDBC client:

In the `MyFirstVerticle` class, declare a new field `JDBCClient jdbc;`, and add the following line at the beginning of the `start` method:

```
jdbc = JDBCClient.createShared(vertx, config(), "My-Whisky-Collection");
```

This creates an instance of JDBC client, configured with the configuration provided to the verticle. To work correctly this configuration needs to provide:

- *url* - the JDBC url such as `jdbc:hsqldb:mem:db?shutdown=true`
- _driver*class* - the JDBC driver class such as `org.hsqldb.jdbcDriver`

Ok, we have the client, we need a connection to the database. This is achieved using the `jdbc.getConnection` that take a `Handler<AsyncResult<SQLConnection>>` as parameter. Let's have a deeper look to this type. It's a `Handler`, so it is called when the result is ready. This result is an instance of `AsyncResult<SQLConnection>`. `AsyncResult` is a structure provided by vert.x that lets us know if the operation was completed successfully or failed. In case of success, it provides the result, here an instance of `SQLConnection`.

When you receive an instance of `AsyncResult`, your code generally looks like:

```
if (ar.failed()) {
  System.err.println("The operation has failed...: "
     + ar.cause().getMessage());
```

```
} else {
  // Use the result:
  result = ar.result();
 }
```

So, let's go back to our `SQLConnection`. We need to retrieve it, and then start the rest of the application. This changes how we start the application, as it will become asynchronous. So, if we divide our startup sequence into small chunks it would be something like:

```
startBackend(
 (connection) -> createSomeData(connection,
    (nothing) -> startWebApp(
       (http) -> completeStartup(http, fut)
    ), fut
 ), fut);
```

with:

1. `startBackend` - retrieves a `SQLConnection` and then calls the next step
2. `createSomeData` - initializes the database and inserts some data. When done, it calls the next step
3. `startWebApp` - starts our web application
4. `completeStartup` - finalizes our start sequence

`fut` is the completion future passed by vert.x that let us report when we are started, or if an issue has been encountered while starting.

Let's have a look to `startBackend`:

```
private void startBackend(Handler<AsyncResult<SQLConnection>> next, Future<Void> fut) {
   jdbc.getConnection(ar -> {
     if (ar.failed()) {
       fut.fail(ar.cause());
     } else {
       next.handle(Future.succeededFuture(ar.result()));
     }
   });
 }
```

This method retrieves a `SQLConnection`, check whether this operation succeeded. If so, it calls the next step. In case of failure, it reports it.

The other methods follow the same pattern: 1) check if the last operation has succeeded, 2) do the task, 3) call the next step.

# A bit of SQL...

Our client is ready, let's now write some SQL statements. Let's start by the `createSomeData` method that is part of the startup sequence:

```
private void createSomeData(AsyncResult<SQLConnection> result,
   Handler<AsyncResult<Void>> next, Future<Void> fut) {
   if (result.failed()) {
     fut.fail(result.cause());
   } else {
     SQLConnection connection = result.result();
     connection.execute(
        "CREATE TABLE IF NOT EXISTS Whisky (id INTEGER IDENTITY, name varchar(100), " +
        "origin varchar(100))",
        ar -> {
          if (ar.failed()) {
            fut.fail(ar.cause());
            connection.close();
            return;
          }
          connection.query("SELECT * FROM Whisky", select -> {
            if (select.failed()) {
              fut.fail(ar.cause());
              connection.close();
              return;
            }
            if (select.result().getNumRows() == 0) {
              insert(
                 new Whisky("Bowmore 15 Years Laimrig", "Scotland, Islay"),
                 connection,
                 (v) -> insert(new Whisky("Talisker 57° North", "Scotland, Island"),
                    connection,
                    (r) -> {
                      next.handle(Future.<Void>succeededFuture());
                      connection.close();
                    }));
            } else {
              next.handle(Future.<Void>succeededFuture());
              connection.close();
            }
          });
        });
   }
 }
```

This method checks that the `SQLConnection` is available and then start executing some SQL statements. First, it creates the tables if there are not there yet. As you can see, the method called is structured as follows:

```
connection.execute(
  SQL statement,
  handler called when the statement has been executed
)
```

The handler receives an `AsyncResult<Void>`, *i.e.* a notification of the completion without an actual result.

> **Closing connection**
> Don't forget to close the SQL connection when you are done. The connection will be given back to the connection pool and be reused.

In the code of this handler, we check whether or not the statement has been executed correctly, and if so we check to see if the table already contains some data, if not, it inserts data using the `insert` method:

```
private void insert(Whisky whisky, SQLConnection connection,
Handler<AsyncResult<Whisky>> next) {
  String sql = "INSERT INTO Whisky (name, origin) VALUES ?, ?";
  connection.updateWithParams(sql,
      new JsonArray().add(whisky.getName()).add(whisky.getOrigin()),
      (ar) -> {
        if (ar.failed()) {
          next.handle(Future.failedFuture(ar.cause()));
          return;
        }
        UpdateResult result = ar.result();
        // Build a new whisky instance with the generated id.
        Whisky w = new Whisky(result.getKeys().getInteger(0), whisky.getName(),
whisky.getOrigin());
        next.handle(Future.succeededFuture(w));
      });
}
```

This method uses the `updateWithParams` method with an *INSERT* statement, and pass values. This approach avoids SQL injection. Once the the statement has been executed, we creates a new `Whisky` object with the created (auto-generated) id.

# Some REST with a pinch of SQL

The method described above is part of our start sequence. But what about the method invoked by our REST API. Let's have a look to

the `getAll` method. This method is called by the web front-end to retrieve all stored products:

```
private void getAll(RoutingContext routingContext) {
  jdbc.getConnection(ar -> {
    SQLConnection connection = ar.result();
    connection.query("SELECT * FROM Whisky", result -> {
      List<Whisky> whiskies =
result.result().getRows().stream().map(Whisky::new).collect(Collectors.toList());
      routingContext.response()
        .putHeader("content-type", "application/json; charset=utf-8")
        .end(Json.encodePrettily(whiskies));
      connection.close(); // Close the connection
    });
  });
 }
```

This method gets a `SQLConnection`, and then issue a query. Once the result has been retrieved it writes the HTTP response as before.
The `getOne`, `deleteOne`, `updateOne` and `addOne` methods follow the same pattern. Notice that the connection can be closed after the response has been written.

Let's have a look to the result provided to the handler passed to the `query` method. It gets a `ResultSet`, which contains the query result. Each row is a `JsonObject`, so if your data object has a constructor taking a `JsonObject` as unique argument, creating there objects is straightforward.

# Test, test, and test again

We need to slightly update our tests to configure the `JDBCClient`. In the `MyFirstVertilceTest` class, change the `DeploymentOption`object created in the `setUp` method to be:

```
DeploymentOptions options = new DeploymentOptions()
    .setConfig(new JsonObject()
      .put("http.port", port)
      .put("url", "jdbc:hsqldb:mem:test?shutdown=true")
      .put("driver_class", "org.hsqldb.jdbcDriver")
    );
```

In addition to the `http.port`, we also put the JDBC url and the class of the JDBC driver. We use an in-memory database for tests.

The same modification needs to be done in the `src/test/resources/my-it-config.json` file:

```
{
```

```
  "http.port": ${http.port},
  "url": "jdbc:hsqldb:mem:it-test?shutdown=true",
  "driver_class": "org.hsqldb.jdbcDriver"
}
```

The `src/main/conf/my-application-conf.json` file also needs to be updated, not for the tests, but to run the application:

```
{
  "http.port" : 8082,
  "url": "jdbc:hsqldb:file:db/whiskies",
  "driver_class": "org.hsqldb.jdbcDriver"
}
```

The JDBC url is a bit different in this last file, as we store the database on the file system.

# Show time!

Let's now build our application:

```
mvn clean package
```

As we didn't change the API (neither the public java one nor the REST), test should run smoothly.

Then launch the application with:

```
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar -conf src/main/conf/my-application-conf.json
```

Open your browser to http://localhost:8082/assets/index.html, and you should see the application using the database. This time the products are stored in a database persisted on the file system. So, if we stop and restart the application, the data is restored.

# Conclusion

In this lab, we saw how you can use JDBC database with vert.x, and thus without too much burden. You may have been surprised by the asynchronous development model, but once you start using it, it's hard to come back.

# Combine vert.x and mongo to build a giant

Last time, we have seen how we can use the `vertx-jdbc-client` to connect to a database using a JDBC driver. In this post, we are going to replace this JDBC client by the `vertx-mongo-client`, and thus connect to a Mongo database.

You don't understand the title, check the mongoDB website.

But before going further, let's recap.

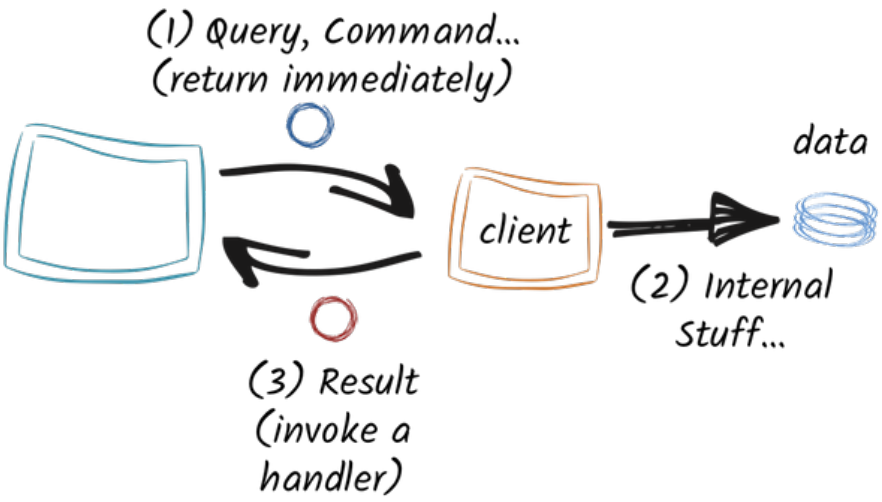
# Previously in 'introduction to vert.x'

1. The first lab has described how to build a vert.x application with Maven and execute unit tests.
2. The second lab has described how this application can become configurable.
3. The third lab has introduced vertx-web, and a small collection management application has been developed. This application offers a REST API used by a HTML/JavaScript frontend.
4. The fourth lab has presented how you can run integration tests to ensure the behavior of your application.
5. The last lab has presented how you can interact with a JDBC database using the vertx-jdbc-client.

This post shows another client that lets you use MongoDB in a vert.x application. This client provides an vert.x API to access asynchronously to the Mongo database. We won't compare whether or not JDBC is superior to Mongo, they have both pros and cons, and you should use the one that meet your requirements. Vert.x lets you choose, that's the point.

# Asynchronous data access

One of the vert.x characteristics is being asynchronous. With an asynchronous API, you don't wait for a result, but you are notified when this result is ready. Thanks to vert.x, this notification happens in the same thread (understand event loop) as the initial request:




Your code (on the left) is going to invoke the mongo client and pass a callback that will be invoked when the result is available. The invocation to the mongo client is non blocking and returns immediately. The client is dealing with the mongo database and when the result has been computed / retrieved, it invokes the callback in the same event loop as the request.

This model is particularly powerful as it avoids the synchronization pitfalls. Indeed, your code is only called by a single thread, no need to synchronize anything.

# As with every Maven project....

… we need to update the `pom.xml` file first.

In the `pom.xml` file, replace the `vertx-jdbc-client` by the `vertx-mongo-client`:

```
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-mongo-client</artifactId>
    <version>3.1.0</version>
  </dependency>
```

Unlike JDBC where we were instantiating a database on the fly, here we need to explicitly starts a MongoDB server. In order to launch a Mongo server in our test, we are going to add another dependency:

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo</artifactId>
    <version>1.50.0</version>
    <scope>test</scope>
  </dependency>
```

This dependency will be used in our unit tests, as it lets us start a mongo server programmatically. For our integration tests, we are going to use a Maven plugin starting and stopping the mongo server before and after our integration tests. Add this plugin to the `<plugins/>`section of your `pom.xml` file.

```
<plugin>
    <groupId>com.github.joelittlejohn.embedmongo</groupId>
    <artifactId>embedmongo-maven-plugin</artifactId>
    <version>0.2.0</version>
    <executions>
     <execution>
      <id>start</id>
      <goals>
       <goal>start</goal>
      </goals>
      <configuration>
       <port>37017</port>
      </configuration>
     </execution>
     <execution>
      <id>stop</id>
      <goals>
       <goal>stop</goal>
      </goals>
     </execution>
    </executions>
  </plugin>
```

Notice the port we use here (37017), we will use this port later.

# Enough XML for today

Now that we have updated our `pom.xml` file, it's time to change our verticle. The first thing to do is to replace the jdbc client by the mongo client:

```
mongo = MongoClient.createShared(vertx, config());
```

This client is configured with the configuration given to the verticle (more on this below).

Once done, we need to change how we start the application. With the mongo client, no need to acquire a connection, it handles this internally. So our startup sequence is a bit more simple:

```
createSomeData(
    (nothing) -> startWebApp(
      (http) -> completeStartup(http, fut)
    ), fut);
```

As in the previous post, we need to insert some predefined data if the database is empty:

```
private void createSomeData(Handler<AsyncResult<Void>> next, Future<Void> fut) {
  Whisky bowmore = new Whisky("Bowmore 15 Years Laimrig", "Scotland, Islay");
  Whisky talisker = new Whisky("Talisker 57° North", "Scotland, Island");
  System.out.println(bowmore.toJson());
  // Do we have data in the collection ?
```

```
  mongo.count(COLLECTION, new JsonObject(), count -> {
    if (count.succeeded()) {
      if (count.result() == 0) {
        // no whiskies, insert data
        mongo.insert(COLLECTION, bowmore.toJson(), ar -> {
          if (ar.failed()) {
            fut.fail(ar.cause());
          } else {
            mongo.insert(COLLECTION, talisker.toJson(), ar2 -> {
              if (ar2.failed()) {
                fut.failed();
              } else {
                next.handle(Future.<Void>succeededFuture());
              }
            });
          }
        });
      } else {
        next.handle(Future.<Void>succeededFuture());
      }
    } else {
      // report the error
      fut.fail(count.cause());
    }
  });
}
```

To detect whether or not the database already contains some data, we retrieve the number of *documents* from the `whiskies` collection. This is done with : `mongo.count(COLLECTION, new JsonObject(), count -> {})`. The second parameter is the query. In our case, we want to count all documents. This is done using `new JsonObject()` that would create a query accepting all documents from the collection (it's equivalent to a `SELECT * FROM ...`).

Also notice the `insert` calls. Documents are passed as JSON object, so to insert an object, just serialize it to JSON and use `mongo.insert(COLLECTION, json, completion handler)`.

# Mongo-ize the REST handlers

Now that the application boot sequence has been migrated to mongo, it's time to update the code handling the REST requests.

Let's start by the `getAll` method that returns all stored products. To implement this, we use the `find` method. As we saw for the `count`method, we pass an empty json object to describe a query accepting all documents:

```
private void getAll(RoutingContext routingContext) {
  mongo.find(COLLECTION, new JsonObject(), results -> {
```

```
    List<JsonObject> objects = results.result();
    List<Whisky> whiskies = objects.stream().map(Whisky::new).collect(Collectors.toList());
    routingContext.response()
        .putHeader("content-type", "application/json; charset=utf-8")
        .end(Json.encodePrettily(whiskies));
  });
}
```

The query results are passed as a list of JSON objects. From this list we can create our product instances, and fill the HTTP response with this set.

To delete a specific document we need to select the document using its `id`:

```
private void deleteOne(RoutingContext routingContext) {
  String id = routingContext.request().getParam("id");
  if (id == null) {
    routingContext.response().setStatusCode(400).end();
  } else {
    mongo.removeOne(COLLECTION, new JsonObject().put("_id", id),
        ar -> routingContext.response().setStatusCode(204).end());
  }
}
```

The `new JsonObject().put("_id", id)` describes a query selecting a single document (selected by its unique `id`, so it's the equivalent to `SELECT * WHERE id=...`). Notice the `_id` which is a mongo trick to select a document by id.

Updating a document is a less trivial:

```
private void updateOne(RoutingContext routingContext) {
  final String id = routingContext.request().getParam("id");
  JsonObject json = routingContext.getBodyAsJson();
  if (id == null || json == null) {
    routingContext.response().setStatusCode(400).end();
  } else {
    mongo.update(COLLECTION,
        new JsonObject().put("_id", id), // Select a unique document
        // The update syntax: {$set, the json object containing the fields to update}
        new JsonObject()
            .put("$set", json),
        v -> {
          if (v.failed()) {
            routingContext.response().setStatusCode(404).end();
          } else {
            routingContext.response()
                .putHeader("content-type", "application/json; charset=utf-8")
                .end(Json.encodePrettily(
                new Whisky(id, json.getString("name"),
                  json.getString("origin"))));
          }
        });
```

```
  }
}
```

As we can see, the `update` method takes two JSON objects as parameter:

1. The first one denotes the query (here we select a single document using its id).
2. The second object expresses the change to apply to the selected document. It uses a mongo syntax. In our case, we update the document using the `$set` operator.

> **Replace document**
> In this code we update the document and replace only a set of fields. You can also replace the whole document using `mongo.replace(...)`.

I definitely recommend to have a look to the MongoDB documentation, especially:

- Query syntax documentation
- Update syntax documentation

# Time for configuration

Well, the code is migrated, but we still need to update the configuration. With JDBC we passed the JDBC url and the driver class in the configuration. With mongo, we need to configure the `connection_string` - the `mongo://` url on which the application is connected, and `db_name` - a name for the data source.

Let's start by the unit test. Edit the `MyFirstVerticleTest` file and add the following code:

```
private static MongodProcess MONGO;
  private static int MONGO_PORT = 12345;
  @BeforeClass
  public static void initialize() throws IOException {
    MongodStarter starter = MongodStarter.getDefaultInstance();
    IMongodConfig mongodConfig = new MongodConfigBuilder()
       .version(Version.Main.PRODUCTION)
       .net(new Net(MONGO_PORT, Network.localhostIsIPv6()))
       .build();
    MongodExecutable mongodExecutable =
        starter.prepare(mongodConfig);
   MONGO = mongodExecutable.start();
  }

  @AfterClass
  public static void shutdown() {  MONGO.stop(); }
```

Before our tests, we start (programmatically) a mongo database on the port 12345. When all our tests have been executed, we shutdown the database.

So now that the mongo server is managed, we need to to give the right configuration to our verticle. Update the `DeploymentOption`instance with:

```
DeploymentOptions options = new DeploymentOptions()
      .setConfig(new JsonObject()
         .put("http.port", port)
         .put("db_name", "whiskies-test")
         .put("connection_string",
            "mongodb://localhost:" + MONGO_PORT)
   );
```

That's all for the unit tests.

For the integration-test, we are using an externalized json file. Edit the `src/test/resources/my-it-config.json` with the following content:

```
{
    "http.port": ${http.port},
    "db_name": "whiskies-it",
    "connection_string": "mongodb://localhost:37017"
  }
```

Notice the port we are using for the mongo server. This port was configured in the `pom.xml` file.

Last but not least, we still have a configuration file to edit: the configuration you use to launch the application in `production`:

```
{
    "http.port": 8082,
    "db_name": "whiskies",
    "connection_string": "mongodb://localhost:27017"
  }
```

Here you would need to edit the `localhost:27017` with the right url for your mongo server.

> **Some changes in the integration tests**
> Because mongo document id are String and not integer, we have to slightly change document selection in the integration test.

# Time for a run

It's time to package and run the application and check that everything works as expected. Let's package the application using:

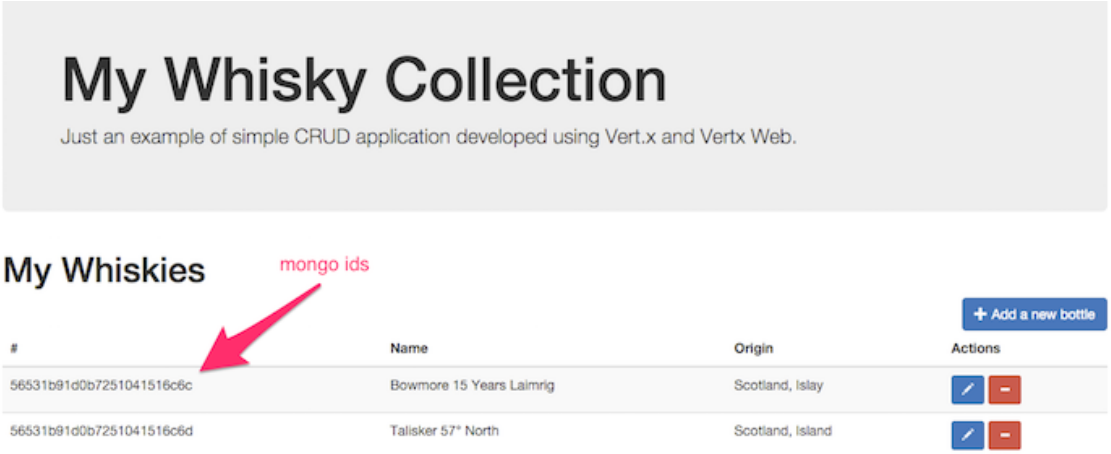
```
mvn clean verify
```

And then to launch it, start your mongo server and launch:

```
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar \
  -conf src/main/conf/my-application-conf.json
```

If you are, like me, using docker / docker-machine for almost everything, edit the configuration file to refer to the right host (localhost for docker, the docker-machine ip if you use docker-machine) and then launch:

```
docker run -d -p 27017:27017 mongo
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar \
  -conf src/main/conf/my-application-conf.json
# or
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar \
  -conf src/main/conf/my-application-conf-docker-machine.json
```




# That's all folks !

We are reaching the end of this post. We saw how you can use the vert-mongo-client to access asynchronously data stored inside a mongo database as well as inserting/updating this data. Now you have the choice between JDBC or Mongo. In addition, vert.x provides a client for Redis.

Stay tuned & Happy coding !