Quarkus - Using Eclipse Vert.x

Eclipse Vert.x is a toolkit for building reactive applications. It is designed to be lightweight and embeddable. Vert.x defines a reactive execution model and provides a large ecosystem.

Quarkus is based on Vert.x, and almost all network-related features rely on Vert.x. While lots of reactive features from Quarkus don't *show* Vert.x, it's used underneath. Quarkus also integrates smoothly with the Vert.x event bus (to enable asynchronous messaging passing between application components) and some reactive clients. You can also use various Vert.x APIs in your Quarkus application, such as deploying *verticles*, instantiating clients…

Installing

To access Vert.x, well, you need to enable the `vertx` extension to use this feature. If you are creating a new project, set the `extensions` parameter are follows:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=vertx-quickstart \
    -Dextensions="vertx"
cd vertx-quickstart
```

If you have an already created project, the `vertx` extension can be added to an existing Quarkus project with the `add-extension` command:

```
./mvnw quarkus:add-extension -Dextensions="vertx"
```

Otherwise, you can manually add this to the dependencies section of your `pom.xml` file:

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-vertx</artifactId>
</dependency>
```

Native Transport

Vert.x is capable of using Netty's native transports which offers performance improvements on certain platforms. To enable them you must include the appropriate dependency for your platform. It's usually a good idea to include both to keep your application platform agnostic. Netty is smart enough to use the correct one, that includes none at all on unsupported platforms:

```xml
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-transport-native-epoll</artifactId>
  <classifier>linux-x86_64</classifier>
</dependency>

<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-transport-native-kqueue</artifactId>
  <classifier>osx-x86_64</classifier>
</dependency>
```

You will also have to explicitly configure Vert.x to use the native transport. In `application.properties` add:

```
quarkus.vertx.prefer-native-transport=true
```

Or in `application.yml`:

```yaml
quarkus:
  vertx:
    prefer-native-transport: true
```

If all is well quarkus will log:

```
[io.qua.ver.cor.run.VertxCoreRecorder] (main) Vertx has Native
Transport Enabled: true
```

Native Linux Transport

On Linux you can enable the following socket options:

- SO_REUSEPORT

```
quarkus.http.so-reuse-port=true
```

- TCP_QUICKACK

```
quarkus.http.tcp-quick-ack=true
```

- TCP_CORK

```
quarkus.http.tcp-cork=true
```

- TCP_FASTOPEN

```
quarkus.http.tcp-fast-open=true
```

Native MacOS Transport

On MacOS Sierra and above you can enable the following socket options:

- SO_REUSEPORT

```
quarkus.http.so-reuse-port=true
```

Accessing Vert.x

Once the extension has been added, you can access the *managed* Vert.x instance using `@Inject`:

```
@Inject Vertx vertx;
```

If you are familiar with Vert.x, you know that Vert.x provides different API models. For instance *bare* Vert.x uses callbacks, the Mutiny variants uses `Uni` and `Multi`, the RX Java 2 version uses `Single`, `Maybe`, `Completable`, `Observable` and `Flowable`...

Quarkus provides 4 Vert.x APIs:

Name	Code	Description
bare	`@Inject io.vertx.core.Vertx vertx`	*bare* Vert.x instance, the API uses callbacks.
Mutiny	`@Inject io.vertx.mutiny.core.Vertx vertx`	The Mutiny API for Vert.x.
RX Java 2	`@Inject io.vertx.reactivex.core.Vertx vertx`	RX Java 2 Vert.x, the API uses RX Java 2 types (deprecated).
Axle	`@Inject io.vertx.axle.core.Vertx vertx`	Axle Vert.x, the API uses `CompletionStage` and `Reactive Streams` (deprecated).

You may inject any of the 4 flavors of `Vertx` as well as the `EventBus` in your Quarkus application beans: `bare`, `Mutiny`, `Axle`, `RxJava2`. They are just shims and rely on a single *managed* Vert.x instance.

You will pick one or the other depending on your use cases.

- `bare`: for advanced usage or if you have existing Vert.x code you want to reuse in your Quarkus application

- `mutiny`: Mutiny is an event-driven reactive programming API. It uses 2 types: `Uni` and `Multi`. This is the recommended API.

- `Axle`: works well with Quarkus and MicroProfile APIs (`CompletionStage` for single results and `Publisher` for streams) - deprecated, it is recommended to switch to Mutiny

- `Rx Java 2`: when you need support for a wide range of data transformation operators on your streams - deprecated, it is recommended to switch to Mutiny

The following snippets illustrate the difference between these 4 APIs:

```java
// Bare Vert.x:
vertx.fileSystem().readFile("lorem-ipsum.txt", ar -> {
    if (ar.succeeded()) {
        System.out.println("Content:" + ar.result().toString("UTF-8"));
    } else {
        System.out.println("Cannot read the file: " + ar.cause().getMessage());
    }
});

// Mutiny Vert.x:
vertx.fileSystem().readFile("lorem-ipsum.txt")
    .onItem().apply(buffer -> buffer.toString("UTF-8"))
    .subscribe(
            content -> System.out.println("Content: " + content),
            err -> System.out.println("Cannot read the file: " + err.getMessage())
    );

// Rx Java 2 Vert.x
vertx.fileSystem().rxReadFile("lorem-ipsum.txt")
    .map(buffer -> buffer.toString("UTF-8"))
    .subscribe(
            content -> System.out.println("Content: " + content),
            err -> System.out.println("Cannot read the file: " + err.getMessage())
    );

// Axle API:
vertx.fileSystem().readFile("lorem-ipsum.txt")
    .thenApply(buffer -> buffer.toString("UTF-8"))
    .whenComplete((content, err) -> {
        if (err != null) {
            System.out.println("Cannot read the file: " + err.getMessage());
        } else {
            System.out.println("Content: " + content);
        }
    });
```

💡 *Mutiny*

If you're not familiar with Mutiny, we recommend to read the Getting Started with Reactive guide first.

Using Vert.x in Reactive JAX-RS resources

Quarkus web resources support asynchronous processing and streaming results over server-sent events.

Asynchronous processing

To asynchronously handle the HTTP request, the endpoint method must return a `java.util.concurrent.CompletionStage` or an `io.smallrye.mutiny.Uni` (requires the `quarkus-resteasy-mutiny` extension):

```java
@Path("/lorem")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public Uni<String> doSomethingAsync() {
        // Mimic an asynchronous computation.
        return Uni.createFrom()
                .item(() -> "Hello!")
                .onItem().delayIt().by(Duration.ofMillis(10));
    }
}
```

```
./mvnw compile quarkus:dev
```

Then, open your browser to 'http://localhost:8080/lorem' and you should get the message.

So far so good. Now let's use the Vert.x API instead of this artificial delay:

```java
package org.acme.vertx;

import io.smallrye.mutiny.Uni;
import io.vertx.mutiny.core.Vertx;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/lorem")
public class GreetingResource {

    @Inject
    Vertx vertx;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public Uni<String> doSomethingAsync() {
        return vertx.fileSystem().readFile("/META-
INF/resources/lorem.txt")
                .onItem().apply(b -> b.toString("UTF-8"));
    }
}
```

In this code, we inject the `vertx` instance (`io.vertx.mutiny.core.Vertx`) and read a file from the file system.

Create the `src/main/resources/META_INF/resources/lorem.txt` file with the following content:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.
```

Then, refresh the page, you should see the *lorem ipsum* text.

Streaming using Server-Sent Events

Quarkus web resources that need to send content as server-sent events must have a method:

- declaring the `text/event-stream` response content type

- returning a Reactive Streams `Publisher` or Mutiny `Multi` (requires the `quarkus-resteasy-mutiny` extension)

In practice, a streaming greeting service would look like:

```
@Path("/hello")
public class StreamingResource {

    @GET
    @Produces(MediaType.SERVER_SENT_EVENTS)
    @Path("/{name}")
    public Multi<String> greeting(@PathParam String name) {
        // TODO: create a Reactive Streams publisher or a Mutiny
Multi
        return publisher;
    }
}
```

Now we just need to return our `Publisher` or `Multi`:

```
package org.acme.vertx;

import io.smallrye.mutiny.Multi;
import io.vertx.mutiny.core.Vertx;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.Date;

@Path("/stream")
public class StreamingResource {

    @Inject
    Vertx vertx;

    @GET
    @Produces(MediaType.SERVER_SENT_EVENTS)
    @Path("/{name}")
    public Multi<String> greeting(@PathParam String name) {
        return vertx.periodicStream(2000).toMulti()
                .map(l -> String.format("Hello %s! (%s)%n", name,
new Date())));
    }
}
```

The server side is ready. In order to see the result in the browser, we need a web page.

META-INF/resources/streaming.html

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>SSE with Vert.x – Quarkus</title>
    <script type="application/javascript" src="streaming.js"
></script>
</head>
<body>
<div id="container"></div>
</body>
</html>
```

Our web page just has an empty `<div>` container. The magic, as always, lies in the Javascript code:

META-INF/resources/streaming.js

```
if (!!window.EventSource) {
    var eventSource = new EventSource("/stream/Quarkus");
    eventSource.onmessage = function (event) {
        var container = document.getElementById("container");
        var paragraph = document.createElement("p");
        paragraph.innerHTML = event.data;
        container.appendChild(paragraph);
    };
} else {
    window.alert("EventSource not available on this browser.")
}
```

Most browsers support SSE but some don't. More about this in Mozilla's SSE browser-compatibility list.

Navigate to http://localhost:8080/streaming.html. A new greeting should show-up every 2 seconds.

```
Hello Quarkus! (Wed Feb 12 17:13:55 CET 2020)

Hello Quarkus! (Wed Feb 12 17:13:57 CET 2020)

Hello Quarkus! (Wed Feb 12 17:13:59 CET 2020)

Hello Quarkus! (Wed Feb 12 17:14:01 CET 2020)

Hello Quarkus! (Wed Feb 12 17:14:03 CET 2020)

...
```

Using Vert.x JSON

Vert.x API heavily relies on JSON, namely the `io.vertx.core.json.JsonObject` and `io.vertx.core.json.JsonArray` types. They are both supported as Quarkus web resource request and response bodies.

Consider these endpoints:

```java
@Path("/hello")
@Produces(MediaType.APPLICATION_JSON)
public class VertxJsonResource {

    @GET
    @Path("{name}/object")
    public JsonObject jsonObject(@PathParam String name) {
        return new JsonObject().put("Hello", name);
    }

    @GET
    @Path("{name}/array")
    public JsonArray jsonArray(@PathParam String name) {
        return new JsonArray().add("Hello").add(name);
    }
}
```

In your browser, navigate to http://localhost:8080/hello/Quarkus/object. You should see:

```
{"Hello":"Quarkus"}
```

Then, navigate to http://localhost:8080/hello/Quarkus/array:

```
["Hello","Quarkus"]
```

Needless to say, this works equally well when the JSON content is a request body or is wrapped in a `Uni`, `Multi`, `CompletionStage` or `Publisher`.

Using Vert.x Clients

As you can inject a Vert.x instance, you can use Vert.x clients in a Quarkus application. This section gives an example with the `WebClient`.

Picking the right dependency

Depending on the API model you want to use you need to add the right dependency to your `pom.xml` file:

```xml
<!-- bare API -->
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-web-client</artifactId>
</dependency>

<!-- Mutiny API -->
<dependency>
    <groupId>io.smallrye.reactive</groupId>
    <artifactId>smallrye-mutiny-vertx-web-client</artifactId>
</dependency>

<!-- Axle API -->
<dependency>
    <groupId>io.smallrye.reactive</groupId>
    <artifactId>smallrye-axle-web-client</artifactId>
</dependency>

<!-- RX Java 2 API -->
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-rx-java2</artifactId>
</dependency>
```

> ℹ️ The `vertx-rx-java2` provides the RX Java 2 API for the whole Vert.x stack, not only the web client.

In this guide, we are going to use the Axle API, so:

```xml
<dependency>
  <groupId>io.smallrye.reactive</groupId>
  <artifactId>smallrye-mutiny-vertx-web-client</artifactId>
</dependency>
```

Now, create a new resource in your project with the following content:

src/main/java/org/acme/vertx/ResourceUsingWebClient.java

```java
package org.acme.vertx;


import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import io.smallrye.mutiny.Uni;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import io.vertx.mutiny.core.Vertx;
import io.vertx.mutiny.ext.web.client.WebClient;
import io.vertx.core.json.JsonObject;
import io.vertx.ext.web.client.WebClientOptions;

@Path("/fruit-data")
public class ResourceUsingWebClient {

    @Inject
    Vertx vertx;

    private WebClient client;

    @PostConstruct
    void initialize() {
        this.client = WebClient.create(vertx,
                new WebClientOptions().setDefaultHost(
"fruityvice.com")
                        .setDefaultPort(443).setSsl(true).setTrustAll
(true));
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{name}")
    public Uni<JsonObject> getFruitData(@PathParam("name") String
```

```
name) {
        return client.get("/api/fruit/" + name)
                .send()
                .onItem().apply(resp -> {
                    if (resp.statusCode() == 200) {
                        return resp.bodyAsJsonObject();
                    } else {
                        return new JsonObject()
                                .put("code", resp.statusCode())
                                .put("message", resp.bodyAsString(
));
                    }
                });
    }

}
```

This resource creates a `WebClient` and upon request use this client to invoke the *fruityvice* API. Depending on the result the response is forwarded as it's received, or a new JSON object is created with the status and body. The `WebClient` is obviously asynchronous (and non-blocking), to the endpoint returns a `Uni`.

Run the application with:

```
./mvnw compile quarkus:dev
```

And then, open a browser to: http://localhost:8080/fruit-data/pear. You should get some details about pears.

The application can also run as a native executable. But, first, we need to instruct Quarkus to enable *ssl*. Open the `src/main/resources/application.properties` and add:

```
quarkus.ssl.native=true
```

Then, create the native executable with:

```
./mvnw package -Pnative
```

Deploying verticles

https://vertx.io/docs/vertx-core/java/#verticles[Verticles] is "a simple, scalable, actor-like deployment and concurrency model" provided by _Vert.x. This model does not claim to be a strict actor-model implementation, but it does share similarities especially with respect to concurrency, scaling and deployment. To use this model, you write and *deploy* verticles, communicating with each other by sending messages on the event bus.

You can deploy *verticles* in Quarkus. It supports:

- *bare* verticle - Java classes extending `io.vertx.core.AbstractVerticle`

- *Mutiny* verticle - Java classes extending `io.smallrye.mutiny.vertx.core.AbstractVerticle`

To deploy verticles, use the regular Vert.x API:

```java
@Inject Vertx vertx;

// ...
vertx.deployVerticle(MyVerticle.class.getName(), ar -> { });
vertx.deployVerticle(new MyVerticle(), ar -> { });
```

You can also pass deployment options to configure the verticle as well as set the number of instances.

Verticles are not *beans* by default. However, you can implement them as *ApplicationScoped* beans and get injection support:

```java
package io.quarkus.vertx.verticles;

import io.smallrye.mutiny.Uni;
import io.smallrye.mutiny.vertx.core.AbstractVerticle;
import org.eclipse.microprofile.config.inject.ConfigProperty;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class MyBeanVerticle extends AbstractVerticle {

    @ConfigProperty(name = "address") String address;

    @Override
    public Uni<Void> asyncStart() {
        return vertx.eventBus().consumer(address)
                .handler(m -> m.replyAndForget("hello"))
                .completionHandler();
    }
}
```

You don't have to inject the `vertx` instance but instead leverage the instance stored in the protected field of `AbstractVerticle`.

Then, deploy the verticle instance with:

```
package io.quarkus.vertx.verticles;

import io.quarkus.runtime.StartupEvent;
import io.vertx.mutiny.core.Vertx;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;

@ApplicationScoped
public class VerticleDeployer {

    public void init(@Observes StartupEvent e, Vertx vertx,
MyBeanVerticle verticle) {
        vertx.deployVerticle(verticle).await().indefinitely();
    }
}
```

If you want to deploy every exposed `AbstractVerticle`, you can use:

```
public void init(@Observes StartupEvent e, Vertx vertx, Instance
<AbstractVerticle> verticles) {
    for (AbstractVerticle verticle : verticles) {
        vertx.deployVerticle(verticle).await().indefinitely();
    }
}
```

Listening to a Unix Domain Socket

Listening on a unix domain socket allows us to dispense with the overhead of TCP if the connection to the quarkus service is established from the same host. This can happen if access to the service goes through a proxy which is often the case if you're setting up a service mesh with a proxy like Envoy.

> ❗ This will only work on platforms that support Native Transport.

To setup please enable the appropriate Native Transport and set the following environment property:

```
quarkus.http.domain-socket=/var/run/io.quarkus.app.socket
quarkus.http.domain-socket-enabled=true
```

By itself this will not disable the tcp socket which by default will open on `0.0.0.0:8080`. It can be explicitly disabled:

```
quarkus.http.host-enabled=false
```

These properties can be set through Java's `-D` command line parameter or on `application.properties`.

Read only deployment environments

In environments with read only file systems you may receive errors of the form:

```
java.lang.IllegalStateException: Failed to create cache dir
```

Assuming `/tmp/` is writeable this can be fixed by setting the `vertx.cacheDirBase` property to point to a directory in `/tmp/` for instance in OpenShift by creating an environment variable `JAVA_OPTS` with the value `-Dvertx.cacheDirBase=/tmp/vertx`.

Going further

There are many other facets of Quarkus using Vert.x underneath:

- The event bus is the connecting tissue of Vert.x applications. Quarkus integrates it so different beans can interact with asynchronous messages. This part is covered in the Async Message Passing documentation.

- Data streaming and Apache Kafka are a important parts of modern systems. Quarkus integrates data streaming using Reactive Messaging. More details on Interacting with Kafka.

- Learn how to implement highly performant, low-overhead database applications on Quarkus with the Reactive SQL Clients.