

# 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 integrates Vert.x to implement different reactive features, such as asynchronous message passing, and non-blocking HTTP client. Basically, Quarkus uses Vert.x as its reactive engine. While lots of reactive features from Quarkus don't *show* Vert.x, it's used underneath. But you can also access the managed Vert.x instance and benefit from the Vert.x ecosystem.

## 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 as follows:

```
mvn io.quarkus:quarkus-maven-plugin:1.2.0.Final:create \
    -DprojectId=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:

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

## 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 RX Java 2 version uses `Single`, `Maybe`, `Completable`, `Observable` and `Flowable`.

Quarkus provides 3 Vert.x APIs:

Name	Code	Description
<i>bare</i>	<code>@Inject io.vertx.core.Vertx vertx</code>	<i>bare</i> Vert.x instance, the API uses callbacks.
RX Java 2	<code>@Inject io.vertx.reactivex.core .Vertx vertx</code>	RX Java 2 Vert.x, the API uses RX Java 2 types.
<i>Axle</i>	<code>@Inject io.vertx.axle.core.Vert x vertx</code>	<i>Axle</i> Vert.x, the API uses <code>CompletionStage</code> and <code>Reactive Streams</code> .



You may inject any of the 3 flavors of `Vertx` as well as the `EventBus` in your Quarkus application beans: `bare`, `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
- `Axle`: works well with Quarkus and MicroProfile APIs (`CompletionStage` for single results and `Publisher` for streams)
- `Rx Java 2`: when you need support for a wide range of data transformation operators on your streams

The following snippets illustrate the difference between these 3 APIs:

```
// 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());
    }
});

// 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);
        }
    });
```

## Using Vert.x in Reactive JAX-RS resources

Quarkus web resources support asynchronous processing and streaming results over [server-sent events](#).

### Asynchronous processing

Most programming guides start easy with a greeting service and this one makes no exception.

To asynchronously greet a client, the endpoint method must return a `java.util.concurrent.CompletionStage`:

```

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

    @Inject
    Vertx vertx;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("/{name}")
    public CompletionStage<String> greeting(@PathParam String name)
    {
        // When complete, return the content to the client
        CompletableFuture<String> future = new CompletableFuture<>
        ();

        long start = System.nanoTime();

        // TODO: asynchronous greeting

        return future;
    }
}

```

So far so good. Now let's use the Vert.x API to implement the artificially delayed reply with the `setTimer` provided by Vert.x:

```

// Delay reply by 10ms
vertx.setTimer(10, 1 -> {
    // Compute elapsed time in milliseconds
    long duration = TimeUnit.MILLISECONDS.convert(System.nanoTime()
    - start, TimeUnit.NANOSECONDS);

    // Format message
    String message = String.format("Hello %s! (%d ms)%n", name,
    duration);

    // Complete
    future.complete(message);
});

```

That's it. Now start Quarkus in `dev` mode with:

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

Eventually, open your browser and navigate to <http://localhost:8080/hello/Quarkus>, you should see:

Hello Quarkus! (10 ms)

## 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 an RX Java 2 [Observable](#) or [Flowable](#)

In practice, a streaming greeting service would look like:

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

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

How to create a Reactive Streams publisher? There are a few ways to do this:

1. If you use `io.vertx.axle.core.Vertx`, the API provides `toPublisher` methods (and then use RX Java 2 or Reactive Streams Operators to manipulate the stream)
2. You can also use `io.vertx.reactivex.core.Vertx` which already provides RX Java 2 (RX Java 2 [Flowable](#) implement Reactive Streams [publisher](#)).

The first approach can be implemented as follows:

```
// Use io.vertx.axle.core.Vertx;
@Inject Vertx vertx;

@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
@Path("/{name}/streaming")
public Publisher<String> greeting(@PathParam String name) {
    return vertx.periodicStream(2000).toPublisherBuilder()
        .map(1 -> String.format("Hello %s! (%s)%n", name, new
Date()))
        .buildRs();
}
```

The second approach slightly differs:

```
// Use io.vertx.reactivex.core.Vertx;
@Inject Vertx vertx;

@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
@Path("/{name}/streaming")
public Publisher<String> greeting(@PathParam String name) {
    return vertx.periodicStream(2000).toFlowable()
        .map(1 -> 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*

```
var eventSource = new EventSource("/hello/Quarkus/streaming");
eventSource.onmessage = function (event) {
    var container = document.getElementById("container");
    var paragraph = document.createElement("p");
    paragraph.innerHTML = event.data;
    container.appendChild(paragraph);
};
```



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! (Thu Mar 21 17:26:12 CET 2019)
Hello Quarkus! (Thu Mar 21 17:26:14 CET 2019)
Hello Quarkus! (Thu Mar 21 17:26:16 CET 2019)
...
```

## 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:

```
@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

`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:

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



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:

```
<dependency>
  <groupId>io.smallrye.reactive</groupId>
  <artifactId>smallrye-axle-web-client</artifactId>
</dependency>
```

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

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

```
package org.acme.vertx;
```



```

import io.vertx.axle.core.Vertx;
import io.vertx.axle.ext.web.client.WebClient;
import io.vertx.axle.ext.web.codec.BodyCodec;
import io.vertx.core.json.JsonObject;
import io.vertx.ext.web.client.WebClientOptions;

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 java.util.concurrent.CompletionStage;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

@Path("/swapi")
public class ResourceUsingWebClient {

    @Inject
    Vertx vertx;

    private WebClient client;

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

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{id}")
    public CompletionStage<JsonObject> getStarWarsCharacter
    (@PathParam int id) {
        return client.get("/api/people/" + id)
            .send()
            .thenApply(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 <https://swapi.co/> 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 `CompletionStage`.

Run the application with:

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

And then, open a browser to: <http://localhost:8080/swapi/1>. You should get *Luke Skywalker*.

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
```

## 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](#).