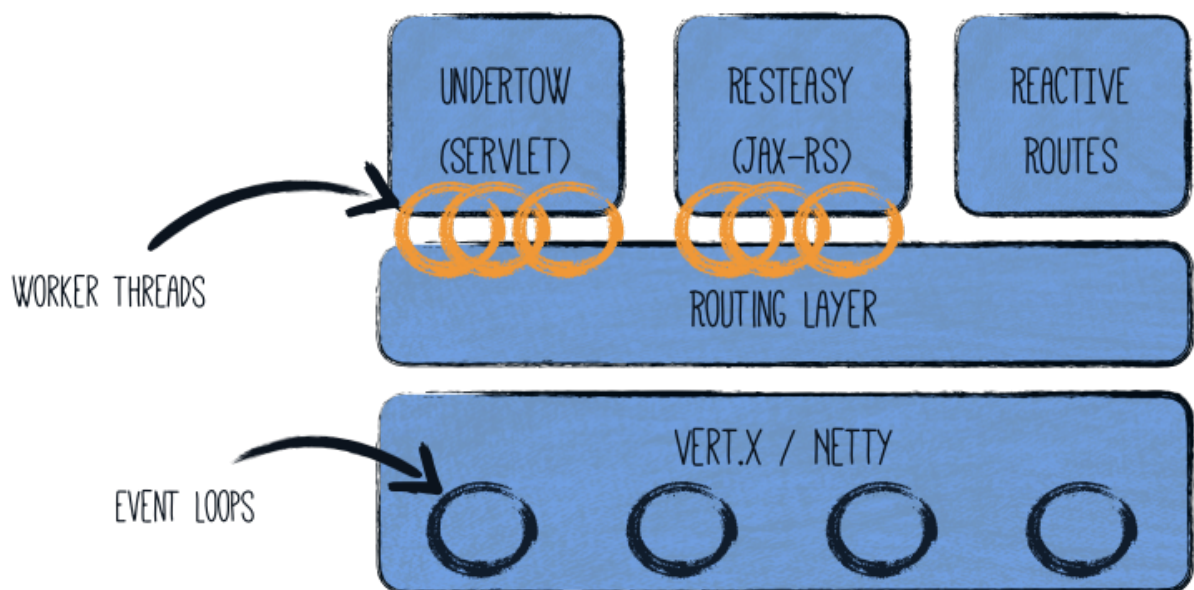# Using Reactive Routes

Reactive routes propose an alternative approach to implement HTTP endpoints where you declare and chain *routes*. This approach became very popular in the JavaScript world, with frameworks like Express.Js or Hapi. Quarkus also offers the possibility to use reactive routes. You can implement REST API with routes only or combine them with JAX-RS resources and servlets.

The code presented in this guide is available in this Github repository under the `reactive-routes-quickstart` directory

Quarkus HTTP

Before going further, let's have a look at the HTTP layer of Quarkus. Quarkus HTTP support is based on a non-blocking and reactive engine (Eclipse Vert.x and Netty). All the HTTP requests your application receive are handled by *event loops* (IO Thread) and then are routed towards the code that manages the request. Depending on the destination, it can invoke the code managing the request on a worker thread (Servlet, Jax-RS) or use the IO Thread (reactive route). Note that because of this, a reactive route must be non-blocking or explicitly declare its blocking nature (which would result by being called on a worker thread).

Declaring reactive routes

The first way to use reactive routes is to use the `@Route` annotation. To have access to this annotation, you need to add the `quarkus-vertx-web` extension:

In your `pom.xml` file, add:

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

Then in a *bean*, you can use the @Route annotation as follows:

```
package org.acme.reactive.routes;

import io.quarkus.vertx.web.Route;
import io.quarkus.vertx.web.RoutingExchange;
import io.vertx.core.http.HttpMethod;
import io.vertx.ext.web.RoutingContext;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped ①
public class MyDeclarativeRoutes {

    // neither path nor regex is set - match a path derived from
the method name
    @Route(methods = HttpMethod.GET) ②
    void hello(RoutingContext rc) { ③
        rc.response().end("hello");
    }

    @Route(path = "/world")
    String helloWorld() { ④
        return "Hello world!";
    }

    @Route(path = "/greetings", methods = HttpMethod.GET)
    void greetings(RoutingExchange ex) { ⑤
        ex.ok("hello " + ex.getParam("name").orElse("world"));
    }
}
```

① If there is a reactive route found on a class with no scope annotation then @javax.inject.Singleton is added automatically.

② The @Route annotation indicates that the method is a reactive route. Again, by default, the code contained in the method must not block.

③ The method gets a RoutingContext as a parameter. From the RoutingContext you can retrieve the HTTP request (using request()) and write the response using response().end(…).

④ If the annotated method does not return void the arguments are optional.

⑤ `RoutingExchange` is a convenient wrapper of `RoutingContext` which provides some useful methods.

More details about using the `RoutingContext` is available in the Vert.x Web documentation.

The `@Route` annotation allows you to configure:

- The `path` - for routing by path, using the Vert.x Web format
- The `regex` - for routing with regular expressions, see for more details
- The `methods` - the HTTP verb triggering the route such as `GET`, `POST`…
- The `type` - it can be *normal* (non-blocking), *blocking* (method dispatched on a worker thread), or *failure* to indicate that this route is called on failures
- The `order` - the order of the route when several routes are involved in handling the incoming request. Must be positive for regular user routes.
- The produced and consumed mime types using `produces`, and `consumes`

For instance, you can declare a blocking route as follows:

```
@Route(methods = HttpMethod.POST, path = "/post", type =
Route.HandlerType.BLOCKING)
public void blocking(RoutingContext rc) {
    // ...
}
```

The `@Route` annotation is repeatable and so you can declare several routes for a single method:

```
@Route(path = "/first") ①
@Route(path = "/second")
public void route(RoutingContext rc) {
    // ...
}
```

① Each route can use different paths, methods…

If no content-type header is set then we will try to use the most acceptable content type as defined by `io.vertx.ext.web.RoutingContext.getAcceptableContentType()`.

```
@Route(path = "/person", produces = "text/html") ①
String person() {
    // ...
}
```

① If the `accept` header matches `text/html` we set the content type automatically.

Handling conflicting routes

You may end up with multiple routes matching a given path. In the following example, both route matches `/accounts/me`:

```
@Route(path = "/accounts/:id", methods = HttpMethod.GET)
void getAccount(RoutingContext ctx) {
   ...
}

@Route(path = "/accounts/me", methods = HttpMethod.GET)
void getCurrentUserAccount(RoutingContext ctx) {
   ...
}
```

As a consequence, the result is not the expected one as the first route is called with the path parameter `id` set to `me`. To avoid the conflict, use the `order` attribute:

```
@Route(path = "/accounts/:id", methods = HttpMethod.GET, order = 2)
void getAccount(RoutingContext ctx) {
   ...
}

@Route(path = "/accounts/me", methods = HttpMethod.GET, order = 1)
void getCurrentUserAccount(RoutingContext ctx) {
   ...
}
```

By giving a lower order to the second route, it gets evaluated first. If the request path matches, it is invoked, otherwise the other routes are evaluated.

@RouteBase

This annotation can be used to configure some defaults for reactive routes declared on a class.

```
@RouteBase(path = "simple", produces = "text/plain") ① ②
public class SimpleRoutes {

    @Route(path = "ping") // the final path is /simple/ping
    void ping(RoutingContext rc) {
        rc.response().end("pong");
    }
}
```

① The `path` value is used as a prefix for any route method declared on the class where

`Route#path()` is used.

② The value of `produces()` is used for content-based routing for all routes where `Route#produces()` is empty.

Reactive Route Methods

A route method must be a non-private non-static method of a CDI bean. If the annotated method returns `void` then it has to accept at least one argument - see the supported types below. If the annotated method does not return `void` then the arguments are optional.

A route method can accept arguments of the following types:

- `io.vertx.ext.web.RoutingContext`
- `io.vertx.reactivex.ext.web.RoutingContext`
- `io.quarkus.vertx.web.RoutingExchange`
- `io.vertx.core.http.HttpServerRequest`
- `io.vertx.core.http.HttpServerResponse`
- `io.vertx.reactivex.core.http.HttpServerRequest`
- `io.vertx.reactivex.core.http.HttpServerResponse`

Furthermore, it is possible to inject the `HttpServerRequest` parameters into a method parameter annotated with `@io.quarkus.vertx.web.Param`:

Parameter Type	Obtained via
`java.lang.String`	`routingContext.request().getParam()`
`java.util.Optional<String>`	`routingContext.request().getParam()`
`java.util.List<String>`	`routingContext.request().params().getAll()`

Request Parameter Example

```
@Route
String hello(@Param Optional<String> name) {
    return "Hello " + name.orElse("world");
}
```

The `HttpServerRequest` headers can be injected into a method parameter annotated with `@io.quarkus.vertx.web.Header`:

Parameter Type	Obtained via
`java.lang.String`	`routingContext.request().getHeader()`
`java.util.Optional<String>`	`routingContext.request().getHeader()`

Parameter Type	Obtained via
`java.util.List<String>`	`routingContext.request().headers().getAll()`

Request Header Example

```
@Route
String helloFromHeader(@Header("My-Header") String header) {
    return header;
}
```

The request body can be injected into a method parameter annotated with `@io.quarkus.vertx.web.Body`.

Parameter Type	Obtained via
`java.lang.String`	`routingContext.getBodyAsString()`
`io.vertx.core.buffer.Buffer`	`routingContext.getBody()`
`io.vertx.core.json.JsonObject`	`routingContext.getBodyAsJson()`
`io.vertx.core.json.JsonArray`	`routingContext.getBodyAsJsonArray()`
any other type	`routingContext.getBodyAsJson().mapTo(MyPojo.class)`

Request Body Example

```
@Route(produces = "application/json")
Person createPerson(@Body Person person, @Param("id")
Optional<String> primaryKey) {
    person.setId(primaryKey.map(Integer::valueOf).orElse(42));
    return person;
}
```

Returning Unis

In a reactive route, you can return a `Uni` directly:

```
@Route(path = "/hello")
Uni<String> hello(RoutingContext context) {
    return Uni.createFrom().item("Hello world!");
}

@Route(path = "/person")
Uni<Person> getPerson(RoutingContext context) {
    return Uni.createFrom().item(() -> new Person("neo", 12345));
}
```

Returning `Unis` is convenient when using a reactive client:

```
@Route(path = "/mail")
Uni<Void> sendEmail(RoutingContext context) {
    return mailer.send(...);
}
```

The item produced by the returned `Uni` can be:

- a string - written into the HTTP response directly

- a buffer - written into the HTTP response directly

- an object - written into the HTTP response after having been encoded into JSON. The `content-type` header is set to `application/json` if not already set.

If the returned `Uni` produces a failure (or is `null`), an HTTP 500 response is written.

Returning a `Uni<Void>` produces a 204 response (no content).

Returning results

You can also return a result directly:

```
@Route(path = "/hello")
String helloSync(RoutingContext context) {
    return "Hello world";
}
```

Be aware, the processing must be **non-blocking** as reactive routes are invoked on the IO Thread. Otherwise, use the `blocking` attribute of the `@Route` annotation.

The method can return:

- a string - written into the HTTP response directly

- a buffer - written into the HTTP response directly

- an object - written into the HTTP response after having been encoded into JSON. The `content-type` header is set to `application/json` if not already set.

Returning Multis

A reactive route can return a `Multi`. The items are written one by one, in the response. The response `Transfer-Encoding` header is set to `chunked`.

```
@Route(path = "/hello")
Multi<String> hellos(RoutingContext context) {
    return Multi.createFrom().items("hello", "world", "!");   ①
}
```

1. Produces `helloworld!`

The method can return:

- a `Multi<String>` - the items are written one by one (one per *chunk*) in the response.
- a `Multi<Buffer>` - the buffers are written one by one (one per *chunk*) without any processing.
- a `Multi<Object>` - the items are encoded to JSON written one by one in the response.

```
@Route(path = "/people")
Multi<Person> people(RoutingContext context) {
    return Multi.createFrom().items(
            new Person("superman", 1),
            new Person("batman", 2),
            new Person("spiderman", 3));
}
```

The previous snippet produces:

```
{"name":"superman", "id": 1} // chunk 1
{"name":"batman", "id": 2} // chunk 2
{"name":"spiderman", "id": 3} // chunk 3
```

Streaming JSON Array items

You can return a `Multi` to produce a JSON Array, where every item is an item from this array. The response is written item by item to the client. The `content-type` is set to `application/json` if not set already.

To use this feature, you need to wrap the returned `Multi` using `io.quarkus.vertx.web.ReactiveRoutes.asJsonArray`:

```
@Route(path = "/people")
Multi<Person> people(RoutingContext context) {
    return ReactiveRoutes.asJsonArray(Multi.createFrom().items(
            new Person("superman", 1),
            new Person("batman", 2),
            new Person("spiderman", 3)));
}
```

The previous snippet produces:

```
[
  {"name":"superman", "id": 1} // chunk 1
  ,{"name":"batman", "id": 2} // chunk 2
  ,{"name":"spiderman", "id": 3} // chunk 3
]
```

Only `Multi<String>`, `Multi<Object>` and `Multi<Void>` can be written into the JSON Array.
Using a `Multi<Void>` produces an empty array. You cannot use `Multi<Buffer>`. If you need to use
`Buffer`, transform the content into a JSON or String representation first.

Event Stream and Server-Sent Event support

You can return a `Multi` to produce an event source (stream of server sent events). To enable this
feature, you need to wrap the returned `Multi` using
`io.quarkus.vertx.web.ReactiveRoutes.asEventStream`:

```
@Route(path = "/people")
Multi<Person> people(RoutingContext context) {
    return ReactiveRoutes.asEventStream(Multi.createFrom().items(
            new Person("superman", 1),
            new Person("batman", 2),
            new Person("spiderman", 3)));
}
```

This method would produce:

```
data: {"name":"superman", "id": 1}
id: 0

data: {"name":"batman", "id": 2}
id: 1

data: {"name":"spiderman", "id": 3}
id: 2
```

You can also implement the `io.quarkus.vertx.web.ReactiveRoutes.ServerSentEvent` interface to customize the `event` and `id` section of the server sent event:

```
class PersonEvent implements ReactiveRoutes.ServerSentEvent<Person>
{
    public String name;
    public int id;

    public PersonEvent(String name, int id) {
        this.name = name;
        this.id = id;
    }

    @Override
    public Person data() {
        return new Person(name, id); // Will be JSON encoded
    }

    @Override
    public long id() {
        return id;
    }

    @Override
    public String event() {
        return "person";
    }
}
```

Using a `Multi<PersonEvent>` (wrapped using `io.quarkus.vertx.web.ReactiveRoutes.asEventStream`) would produce:

```
event: person
data: {"name":"superman", "id": 1}
id: 1

event: person
data: {"name":"batman", "id": 2}
id: 2

event: person
data: {"name":"spiderman", "id": 3}
id: 3
```

Using the Vert.x Web Router

You can also register your route directly on the *HTTP routing layer* by registering routes directly on the `Router` object. To retrieve the `Router` instance at startup:

```
public void init(@Observes Router router) {
    router.get("/my-route").handler(rc -> rc.response().end("Hello
from my route"));
}
```

Check the Vert.x Web documentation to know more about the route registration, options, and available handlers.

> `Router` access is provided by the `quarkus-vertx-http` extension. If you use `quarkus-resteasy` or `quarkus-vertx-web`, the extension will be added automatically.

Intercepting HTTP requests

You can also register filters that would intercept incoming HTTP requests. Note that these filters are also applied for servlets, JAX-RS resources, and reactive routes.

For example, the following code snippet registers a filter adding an HTTP header:

```
package org.acme.reactive.routes;

import io.vertx.ext.web.RoutingContext;

public class MyFilters {

    @RouteFilter(100) ①
    void myFilter(RoutingContext rc) {
        rc.response().putHeader("X-Header", "intercepting the
request");
        rc.next(); ②
    }
}
```

① The `RouteFilter#value()` defines the priority used to sort the filters - filters with higher priority are called first.

② The filter is likely required to call the `next()` method to continue the chain.

Adding OpenAPI and Swagger UI

You can add support for OpenAPI and Swagger UI by using the `quarkus-smallrye-openapi` extension.

Add the extension by running this command:

```
./mvnw quarkus:add-extension -Dextensions="io.quarkus:quarkus
-smallrye-openapi"
```

This will add the following to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-openapi</artifactId>
</dependency>
```

This is enough to generate a basic OpenAPI schema document from your Vert.x Routes:

```
curl http://localhost:8080/openapi
```

You will see the generated OpenAPI schema document:

```
---
openapi: 3.0.3
info:
  title: Generated API
  version: "1.0"
paths:
  /greetings:
    get:
      responses:
        "204":
          description: No Content
  /hello:
    get:
      responses:
        "204":
          description: No Content
  /world:
    get:
      responses:
        "200":
          description: OK
          content:
            '*/*':
              schema:
                type: string
```

Also see the OpenAPI Guide.

Adding MicroProfile OpenAPI Annotations

You can use MicroProfile OpenAPI to better document your schema, example, adding header info, or specifying the return type on `void` methods might be usefull :

```
@OpenAPIDefinition(①
    info = @Info(
        title="Greeting API",
        version = "1.0.1",
        contact = @Contact(
            name = "Greeting API Support",
            url = "http://exampleurl.com/contact",
            email = "techsupport@example.com"),
        license = @License(
            name = "Apache 2.0",
            url = "http://www.apache.org/licenses/LICENSE-
2.0.html"))
)
@ApplicationScoped
public class MyDeclarativeRoutes {

    // neither path nor regex is set – match a path derived from
the method name
    @Route(methods = HttpMethod.GET)
    @APIResponse(responseCode="200",
            description="Say hello",
            content=@Content(mediaType="application/json",
schema=@Schema(type=SchemaType.STRING)))②
    void hello(RoutingContext rc) {
        rc.response().end("hello");
    }

    @Route(path = "/world")
    String helloWorld() {
        return "Hello world!";
    }

    @Route(path = "/greetings", methods = HttpMethod.GET)
    @APIResponse(responseCode="200",
            description="Greeting",
            content=@Content(mediaType="application/json",
schema=@Schema(type=SchemaType.STRING)))
    void greetings(RoutingExchange ex) {
        ex.ok("hello " + ex.getParam("name").orElse("world"));
    }
}
```

① Header information about your API.

② Defining the response

This will generate this OpenAPI schema:

```
---
openapi: 3.0.3
info:
  title: Greeting API
  contact:
    name: Greeting API Support
    url: http://exampleurl.com/contact
    email: techsupport@example.com
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.1
paths:
  /greetings:
    get:
      responses:
        "200":
          description: Greeting
          content:
            application/json:
              schema:
                type: string
  /hello:
    get:
      responses:
        "200":
          description: Say hello
          content:
            application/json:
              schema:
                type: string
  /world:
    get:
      responses:
        "200":
          description: OK
          content:
            '*/*':
              schema:
                type: string
```

Using Swagger UI

Swagger UI is included by default when running in `dev` or `test` mode, and can optionally added to `prod` mode. See the Swagger UI Guide for more details.

Navigate to localhost:8080/swagger-ui/ and you will see the Swagger UI screen:

Conclusion

This guide has introduced how you can use reactive routes to define an HTTP endpoint. It also describes the structure of the Quarkus HTTP layer and how to write filters.