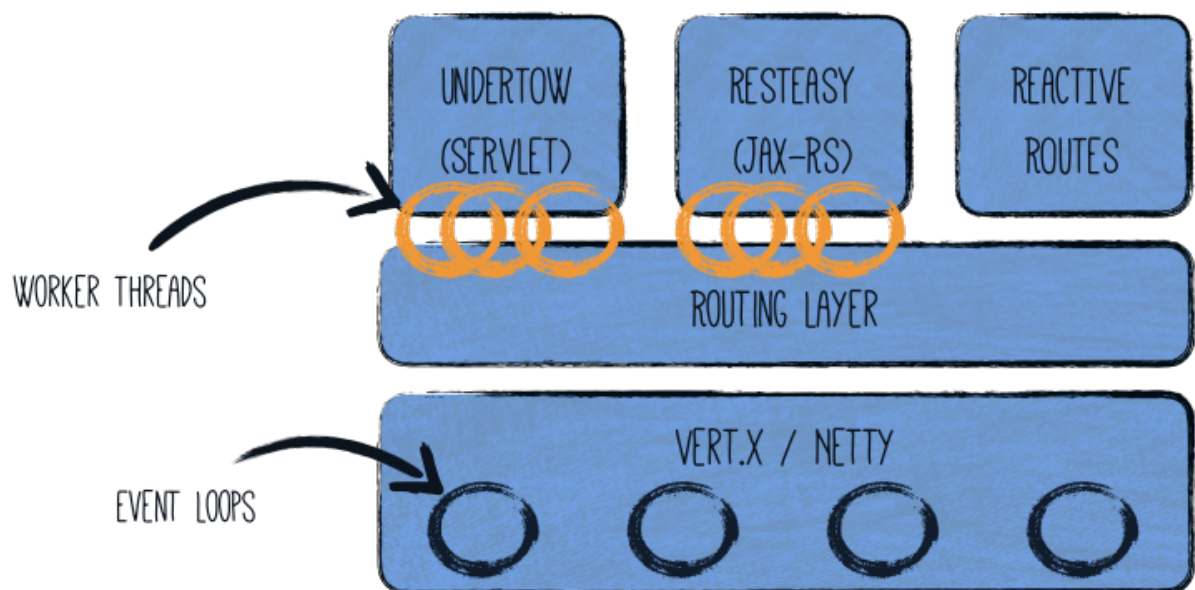# Using Reactive Routes

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:

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

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

```java
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 = "/greetings", methods = HttpMethod.GET)
    void greetings(RoutingExchange ex) { ④
        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 `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 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) {
    // ...
}
```

You can also declare several routes for a single method using `@Routes`:

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

Each route can use different paths, methods…

@RouteBase

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

```
@RouteBase(path = "simple", produces = "text/plain") ①
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. The `produces` value is used for content-based routing for all routes where `Route#produces()` is empty.

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:

```java
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.

> ℹ️ `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:

```java
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.

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.