Chapter 11: Reactive Microservices with Vert.x

In the previous chapter, we familiarized ourselves with the Ktor framework. We created a web service that could store cats in its database.

In this chapter, we'll continue working on the example from the previous chapter, but this time using the Vert.x framework and Kotlin. **Vert.x** is a Reactive framework that is built on top of Reactive principles, which we discussed in *Chapter 7*, *Controlling the Data Flow*. We'll list some of the other benefits of the Vert.x framework in this chapter. You can always read more about Vert.x by going to the official website: https://vertx.io.

The microservice we'll develop in this chapter will provide an endpoint for health checks – the same as the one we created in Ktor – and will be able to delete and update the cats in our database.

In this chapter, we will cover the following topics:

- Getting started with Vert.x
- Routing in Vert.x
- Verticles
- Handling requests
- Testing Vert.x applications
- Working with databases
- Understanding Event Loop
- Communicating with Event Bus

Technical requirements

For this chapter, you'll need the following:

- **JDK 11** or later
- IntelliJ IDEA
- **Gradle 6.8** or later
- **PostgreSQL 14** or later

Like the previous chapter, this chapter will also assume that you have PostgreSQL already installed and that you have basic knowledge of working with it. We'll also use the same table structure we created with Ktor.

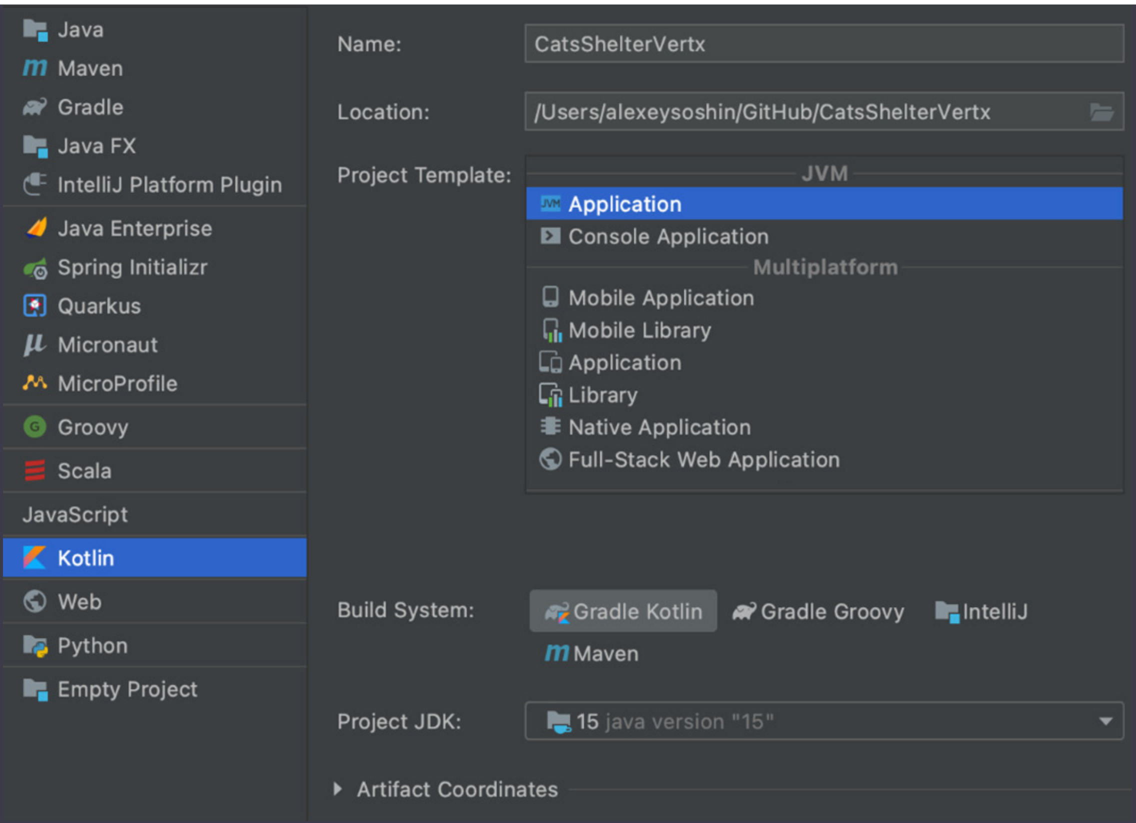
You can find the full source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter11.

Getting started with Vert.x

Vert.x is a Reactive framework that is asynchronous and non-blocking. Let's understand what this means by looking at a concrete example.

We'll start by creating a new Kotlin Gradle project or by using start.vertx.io:

1. From your IntelliJ IDEA application, select **File** | **New** | **Project** and choose **Kotlin** from the **New Project** wizard.
2. Then, specify a name for your project – `CatsShelterVertx`, in my case – and choose **Gradle Kotlin** as your **Build System**.
3. Then, select the **Project JDK** version that you have installed from the dropdown. The output should look as follows:



Name: CatsShelterVertx

Location: /Users/alexeysoshin/GitHub/CatsShelterVertx

Project Template: Application

Build System: Gradle Kotlin

Project JDK: 15 java version "15"

Figure 11.1 – Creating a Kotlin application

Next, add the following dependencies to your **`build.gradle.kts`** file:

```
val vertxVersion = "4.1.5"
dependencies {
    implementation("io.vertx:vertx-
core:$vertxVersion")
    implementation("io.vertx:vertx-
web:$vertxVersion")
    implementation("io.vertx:vertx-lang-
        kotlin:$vertxVersion")
    implementation("io.vertx:vertx-lang-kotlin-
        coroutines:$vertxVersion")
    ...
}
```

Similar to what we discussed in the previous chapter, all the dependencies must be of the same version to avoid any conflicts. That's the reason we are using a variable for the library version – to be able to change all of them together.

The following is an explanation of each dependency:

- **`vertx-core`** is the core library.
- **`vertx-web`** is needed since we want our service to be REST-based.
- **`vertx-lang-kotlin`** provides idiomatic ways to write Kotlin code with Vert.x.
- Finally, **`vertx-lang-kotlin-coroutines`** integrates with the coroutines, which we discussed in detail in Chapter 6, *Threads and Coroutines*.

Then, we must create a file called **`server.kt`** in the **`src/main/kotlin`** folder with the following content:

```
fun main() {
    val vertx = Vertx.vertx()
    vertx.createHttpServer().requestHandler{ ctx ->
        ctx.response().end("OK")
    }.listen(8081)
    println("open http://localhost:8081")
}
```

That's all you need to start a web server that will respond with **`OK`** when you open **`http://localhost:8081`** in your browser.

Now, let's understand what happens here. First, we create a Vert.x instance using the Factory method from Chapter 3, *Understanding Structural Patterns*.

The **`requestHandler`** method is just a simple listener or a subscriber. If you don't remember how it works, check out Chapter 4, *Getting Familiar with Behavioral Patterns*, for the Observable design pattern. In our case, it will be called for each new request. That's the asynchronous nature of Vert.x in action.

Next, let's learn how to add routes in Vert.x.

Routing in Vert.x

Notice that no matter which URL we specify, we always get the same result. Of course, that's not what we want to achieve. Let's start by adding the most basic endpoint, which will only tell us that the service is up and running.

For that, we'll use **`Router`**:

```
val vertx = Vertx.vertx()
val router = Router.router(vertx)
...
```

`Router` lets you specify handlers for different HTTP methods and URLs.

Now, let's add a **`/status`** endpoint that will return an HTTP status code of **`200`** and a message stating **`OK`** to our user:

```
router.get("/status").handler { ctx ->
    ctx.response()
        .setStatusCode(200)
        .end("OK")
}
vertx.createHttpServer()
    .requestHandler(router)
    .listen(8081)
```

Now, instead of specifying the request handler as a block, we will pass this function to our **`router`** object. This makes our code easier to manage.

We learned how we return a flat text response in the very first example. So, now, let's return JSON instead. Most real-life applications use JSON for communication. Let's replace the body of our status handler with the following code:

```
val json = json {
    obj(
        "status" to "OK"
    )
}
ctx.response()
    .setStatusCode(200)
    .end(json.toString())
```

Here, we are using a DSL, which we discussed in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, to create a JSON object.

You can open **`http://localhost:8081/status`** in your browser and make sure that you get **`{"status": "OK"}`** as a response.

Now, let's discuss how we can structure our code better with the Vert.x framework.

Verticles

Our current code resides in the **`server.kt`** file, which is getting bigger and bigger. We need to find a way to split it. In Vert.x, code is split into classes called **verticles**.

You can think of a verticle as a lightweight actor. We discussed Actors in *Chapter 5*, *Introducing Functional Programming*.

Let's see how we can create a new verticle that will encapsulate our server:

```
class ServerVerticle : CoroutineVerticle() {
    override suspend fun start() {
        val router = router()
        vertx.createHttpServer()
            .requestHandler(router)
            .listen(8081)
        println("open http://localhost:8081")
    }
    private fun router(): Router {
        // Our router code comes here now
        val router = Router.router(vertx)
        ...
        return router
    }
}
```

Every verticle has a **start()** method that handles its initialization. As you can see, we moved all the code from our **main()** function to the **start()** method. If we run the code now, though, nothing will happen. That's because the verticle hasn't been started yet.

There are different ways to start a verticle, but the simplest way is to pass the instance of the class to the **deployVerticle()** method. In our case, this is the **ServerVerticle** class:

```
fun main() {
    val vertx = Vertx.vertx()
    vertx.deployVerticle(ServerVerticle())
}
```

Here is another, more flexible way to specify the class name as a string:

```
fun main() {
    val vertx = Vertx.vertx()
    vertx.deployVerticle("ServerVerticle")
}
```

If our verticle class is not in the default package, we'll need to specify the fully qualified path for Vert.x to be able to initialize it.

Now, our code has been split into two files, **ServerVerticle.kt** and **server.kt**, and is organized better. Next, we'll learn how we can do the same refactoring to organize our routes in a better way.

Handling requests

As we discussed earlier in this chapter, all requests in Vert.x are handled by the **`Router`** class. We covered the concept of routing in the previous chapter, so now, let's just discuss the differences between the Ktor and Vert.x approaches to routing requests.

We'll declare two endpoints to delete cats from the database and update information about a particular cat. We'll use the **`delete`** and **`put`** verbs, respectively, for this:

```
router.delete("/cats/:id").handler { ctx ->
    // Code for deleting a cat
}
router.put("/cats/:id").handler { ctx ->
    // Code for updating a cat
}
```

Both endpoints receive a URL parameter. In Vert.x, we use a colon notation for this.

To be able to parse JSON requests and responses, Vert.x has a **`BodyHandler`** class. Now, let's declare it as well. This should come just after the instantiation of our router:

```
val router = Router.router(vertx)
router.route().handler(BodyHandler.create())
```

This will tell Vert.x to parse the request body into JSON for any request.

Notice that the **`/cat`** prefix is repeated multiple times in our code now. To avoid that and make our code more modular, we can use a subrouter, which we'll discuss in the next section.

Subrouting the requests

Subrouting allows us to split routes into multiple classes to keep our code more organized. Let's move the new routes to a new function by following these steps:

1. We'll leave the **`/alive`** endpoint as is, but we'll extract all the other endpoints into a separate function:

   ```
   private fun catsRouter(): Router {
       val router = Router.router(vertx)
       router.delete("/:id").handler { ctx ->
           // Code for deleting a cat
       }
       router.put("/:id").handler { ctx ->
           // Code for updating a cat
       }
       return router
   }
   ```

   Inside this function, we create a separate **`Router`** object that will only handle the routes for cats, not the status routes.

2. Now, we need to connect **`SubRouter`** to our main router:

   ```
   router.mountSubRouter("/cats", catsRouter())
   ```

   Keeping our code clean and well separated is very important. Extracting routes into subrouters helps us with that.

Now, let's discuss how this code can be tested.

Testing Vert.x applications

To test our Vert.x application, we'll use the **JUnit 5** framework, which we discussed in the previous chapter.

You'll need the following two dependencies in your **`build.gradle.kts`** file:

```
dependencies {
    ...
    testImplementation("org.junit.jupiter:junit-
jupiter-
        api:5.6.0")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-
```

```
        engine:5.6.0")
}
```

Our first test will be located in the **/src/test/kotlin/ServerTest.kt** file.

The basic structure of all the integration tests looks something like this:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class ServerTest {
    private val vertx: Vertx = Vertx.vertx()
    @BeforeAll
    fun setup() {
        runBlocking {
            vertx.deployVerticle(ServerVerticle()).aw
ait()
        }
    }
    @AfterAll
    fun tearDown() {
        // And you want to stop your server once
        vertx.close()
    }

    @Test
    fun `status should return 200`() {
    }
}
```

This structure is different from what we've seen in Ktor. Here, we start the server ourselves, in the **setup()** method.

Since Vert.x is Reactive, the **deployVerticle()** method will return a **Future** object immediately, releasing the thread, but that doesn't mean that the server verticle has started yet.

To avoid this race, we can use the **await()** method, which will block the execution of our tests until the server is ready to receive requests.

Now, we want to issue an actual HTTP call to our **/status** endpoint, for example, and check the response code. For that, we'll use the Vert.x web client.

Let's add it to our **build.gradle.kts** dependencies section:

```
testImplementation("io.vertx:vertx-web-
client:$vertxVersion")
```

Since we only plan to use **WebClient** in tests, we specify **testImplementation** instead of **implementation**. But **WebClient** is so useful that you'll probably end up using it in your production code anyway.

After adding this new dependency, we need to instantiate our web client in the **setup** method:

```
lateinit var client: WebClient
@BeforeAll
fun setup() {
    vertx.deployVerticle(ServerVerticle())
    client = WebClient.create(
        vertx,
        WebClientOptions()
            .setDefaultPort(8081)
            .setDefaultHost("localhost")
    )
}
```

The **setup()** method will be called once before all the tests start. In this method, we are deploying our server verticle and creating a web client with some defaults for all our tests to share.

Now, let's write a test to check that our server is up and running:

```
@Test
fun `status should return 200`() {
    runBlocking {
        val response =
client.get("/status").send().await()
        assertEquals(201, response.statusCode())
    }
}
```

Now, let's understand what happens in this test:

- **`client`** is an instance of **`WebClient`** that is shared by all our tests. We invoke the **`/status`** endpoint using the **`get`** verb. This is a Builder design pattern, so to issue our request, we need to use the **`send()`** method. Otherwise, nothing will happen.
- Vert.x is a Reactive framework, so instead of blocking our thread until a response is received, the **`send()`** method returns a Future. Then, we use **`await()`**, which adapts a Future to a Kotlin coroutine to be able to wait for the results concurrently.
- Once the response is received, we check it in the same way that we did in other tests – by using the **`assertEquals`** function, which comes from JUnit.

Now that we know how to write tests in Vert.x, let's discuss how we can work with databases in a Reactive manner.

Working with databases

To be able to progress further with our tests, we need the ability to create entities in the database. For that, we'll need to connect to the database.

First, let's add the following two lines to our **`build.gradle.kts`** dependencies section:

```
implementation("org.postgresql:postgresql:42.3.0")
implementation("io.vertx:vertx-pg-
client:$vertxVersion")
```

The first line of code fetches the PostgreSQL driver. The second one adds the Vert.x JDBC client, which allows Vert.x, which has the driver, to connect to any database that supports JDBC.

Managing configuration

Now, we want to hold the database configuration somewhere. For local development, it may be fine to have those configurations hardcoded. We'll execute the following steps to do this:

1. When we connect to the database, we need to specify the following parameters at the very least:
   1. Username
   2. Password
   3. Host
   4. Database name

   We'll store the preceding parameters in a **Singleton** object:

   ```
   object Db {
       val username =
   System.getenv("DATABASE_USERNAME")         ?:
   "cats_admin"
       val password =
   System.getenv("DATABASE_PASSWORD")         ?:
   "abcd1234"
       val database = System.getenv("DATABASE_NAME")
           ?: "cats_db"
       val host = System.getenv("DATABASE_HOST")
           ?: "localhost"
   }
   ```

   Our **Singleton** object has four members. For each, we check whether an environment variable was set, and if there's no such environment variable, we provide a default value using the Elvis operator.
2. Now, let's add a function that will return a connection pool:

   ```
   fun connect(vertx: Vertx): SqlClient {
       val connectOptions = PgConnectOptions()
   ```

```
            .setPort(5432)
            .setHost(host)
            .setDatabase(database)
            .setUser(username)
            .setPassword(password)
        val poolOptions = PoolOptions()
            .setMaxSize(20)
        return PgPool.client(
            vertx,
            connectOptions,
            poolOptions
        )
    }
```

Our **`connect()`** method creates two configuration objects: **`PgConnectOptions`** sets the configuration for the database we want to connect to, while **`PoolOptions`** specifies the configuration of the connection pool.

3. Now, all we need to do is instantiate the database client in our test:

```
...
lateinit var db: SqlClient
@BeforeAll
fun setup() {
    runBlocking {
        ...
        db = Db.connect(vertx)
    }
}
```

4. Having done that, let's create a new **`Nested`** class in our test file for cases where we expect to have a cat in our database:

```
@Nested
inner class `With Cat` {
    @BeforeEach
    fun createCats() {
        ...
    }
    @AfterEach
    fun deleteAll() {
        ...
    }
}
```

Unlike the Exposed framework, which we discussed in the previous chapter, the database client in Vert.x doesn't have specific methods for insertion, deletion, and so on. Instead, it provides a lower-level API that allows us to execute any type of query on the database.

5. First, let's write a query that will clean our database:

```
@AfterEach
fun deleteAll() {
    runBlocking {
        db.preparedQuery("DELETE FROM
cats")            .execute().await()
    }
}
```

The basic structure for working with the database client in Vert.x is to pass a query to the **`prepareQuery()`** method, then execute it using **`execute()`**.

We want to wait for the query to complete before we move on to the next test, so we use the **`await()`** function to wait for the current coroutine, and we use the **`runBlocking()`** adapter method to have a coroutine context to do so.

6. Now, let's write another query that will add a cat to the database before each test runs:

```
lateinit var catRow: Row
@BeforeEach
fun createCats() {
    runBlocking {
        val result = db.preparedQuery(
            """INSERT INTO cats (name, age)
            VALUES ($1, $2)
            RETURNING ID""".trimIndent()
        ).execute(Tuple.of("Binky", 7)).await()
        catRow = result.first()
    }
}
```

Here, we are using the **`preparedQuery()`** method once more, but this time, our SQL query string contains placeholders. Each placeholder starts with a dollar sign and their indexes start with **`1`**.

Then, we pass the values for those placeholders to the **`execute()`** method. **`Tuple.of`** is a Factory method design pattern that you should be able to recognize well by now.

We also want to remember the ID of the cat that we create since we'll use that ID to delete or update the cat. For this reason, we store the created row in a **`lateinit`** variable.

7. We now have everything prepared to write our test:

```
@Test
fun `delete deletes a cat by ID`() {
    runBlocking {
        val catId = catRow.getInteger(0)
        client.delete("/cats/${catId}").send().awai
t()
        val result = db.preparedQuery("SELECT *
FROM             cats WHERE id =
$1")            .execute(Tuple.of(catId)).await()
        assertEquals(0, result.size())
    }
}
```

First, we get the ID of the cat we want to delete from the database row using the **`getInteger()`** method. Unlike parameters that start with **`1`**, the columns of a database row start with **`0`**. So, by getting an integer at index **`0`**, we get the ID of our cat.

Then, we invoke the web client's **`delete()`** method and wait for it to complete.

Afterward, we execute a **`SELECT`** statement on our database, checking that the row was indeed deleted.

If you run this test now, it will fail, because we haven't implemented the **delete** endpoint yet. We'll do that in the next section.

Understanding Event Loop

The goal of the **Event Loop** design pattern is to continuously check for new events in a queue, and each time a new event comes in, to quickly dispatch it to someone who knows how to handle it. This way, a single thread or a very limited number of threads can handle a huge number of events.

In the case of web frameworks such as Vert.x, events may be requests to our server.

To understand the concept of the Event Loop better, let's go back to our server code and attempt to implement an endpoint for deleting a cat:

```
val db = Db.connect(vertx)
router.delete("/:id").handler { ctx ->
    val id = ctx.request().getParam("id").toInt()
    db.preparedQuery("DELETE FROM cats WHERE ID =
$1")        .execute(Tuple.of(id)).await()
    ctx.end()
}
```

This code is very similar to what we've written in our tests in the previous section. We read the URL parameter from the request using the **getParam()** function, then we pass this ID to the prepared query. This time, though, we can't use the **runBlocking** adapter function, since it will block the Event Loop.

Vert.x uses a limited number of threads, as many as twice the number of your CPU cores, to run all its code efficiently. However, this means that we cannot execute any blocking operations on those threads since it will negatively impact the performance of our application.

To solve this issue, we can use a coroutine builder we're already familiar with: **`launch()`**. Let's see how this works:

```
router.delete("/:id").handler { ctx ->
    launch {
        val id = ctx.request().getParam("id").toInt()
        db.preparedQuery("DELETE FROM cats WHERE ID =
$1")            .execute(Tuple.of(id)).await()
        ctx.end()
    }
}
```

Since our verticle extends **`CoroutineVerticle`**, we have access to all the regular coroutine builders that will run on the Event Loop.

Now, all we need to do is mark our routing functions with the **`suspend`** keyword:

```
private suspend fun router(): Router {
    ...
}
private suspend fun catsRouter(): Router {
    ...
}
```

Now, let's add another test for updating a cat:

```
@Test
fun `put updates a cat by ID`() {
    runBlocking {
        val catId = catRow.getInteger(0)
        val requestBody = json {
            obj("name" to "Meatloaf", "age" to 4)
        }
        client.put("/cats/${catId}")
            .sendBuffer(Buffer.buffer(requestBody.toS
tring()))
```

```
            .await()
        val result = db.preparedQuery("SELECT * FROM
cats
            WHERE id = $1")
            .execute(Tuple.of(catId)).await()
        assertEquals("Meatloaf",
            result.first().getString("name"))
        assertEquals(4,
result.first().getInteger("age"))
    }
}
```

This test is very similar to the deletion test, with the only major difference being that we use **`sendBuffer`** and not the **`send()`** method, so we can send a JSON body to our **`put`** endpoint.

We create the JSON similarly to what we saw when we implemented the **`/status`** endpoint earlier in this chapter.

Now, let's implement the **`put`** endpoint for the test to pass:

```
router.put("/:id").handler { ctx ->
    launch {
        val id = ctx.request().getParam("id").toInt()
        val body = ctx.bodyAsJson
        db.preparedQuery("UPDATE cats SET name = $1,
age =             $2 WHERE ID = $3")
            .execute(
                Tuple.of(
                    body.getString("name"),
                    body.getInteger("age"),
                    id
                )
            ).await()
        ctx.end()
    }
```

```
  }
```

Here, the main difference from the previous endpoint we've implemented is that this time, we need to parse our request **body**. We can do that by using the **bodyAsJson** property. Then, we can use the **getString** and **getInteger** methods, which are available in JSON, to get the new values for **name** and **age**.

With this, you should have all the required knowledge to implement other endpoints as needed. Now, let's learn how to structure our code in a better way using the concept of Event Bus since it all resides in a single large class.

Communicating with Event Bus

Event Bus is an implementation of the Observable design pattern, which we discussed in *Chapter 4*, *Getting Familiar with Behavioral Patterns*.

We've already mentioned that Vert.x is based on the concept of verticles, which are isolated actors. We've already seen the other types of actors in *Chapter 6*, *Threads and Coroutines*. Kotlin's **coroutines** library provides the **actor()** and **producer()** coroutine generators, which create a coroutine bound to a channel.

Similarly, all the verticles in the Vert.x framework are bound by Event Bus and can pass messages to one another using it. Now, let's extract the code from our **ServerVerticle** class into a new class, which we'll call **CatVerticle**.

Any verticle can send a message over Event Bus by choosing between the following methods:

- **request()** will send a message to only one subscriber and wait for a response.
- **send()** will send a message to only one subscriber, without waiting for a response.

- **publish()** will send a message to all subscribers, without waiting for a response.

No matter which method is used to send the message, you subscribe to it using the **consumer()** method on Event Bus.

Now, let's subscribe to an event in our **CatsVerticle** class:

```
class CatsVerticle : CoroutineVerticle() {
    override suspend fun start() {
        val db = Db.connect(vertx)
        vertx.eventBus().consumer<Int>("cats:delete")
{req->
            launch {
                val id = req.body()
                db.preparedQuery("DELETE FROM
                  cats WHERE ID = $1")
                    .execute(Tuple.of(id)).await()
                req.reply(null)
            }
        }
    }
}
```

The generic type of the **consumer()** method specifies the type of message we'll receive. In this case, it's **Int**.

The string that we provide to the method – in our case, **cats:delete** – is the address we subscribe to. It can be any string, but it is good to have some convention, such as what type of object we operate on and what we want to do with it.

Once the delete action has been executed, we respond to our publisher with the **reply()** method. Since we don't have any information to send back, we simply send **null**.

Now, let's replace our previous **`delete`** route with the following code:

```
router.delete("/:id").handler { ctx ->
    val id = ctx.request().getParam("id").toInt()
    vertx.eventBus().request<Nothing>("cats:delete",
id) {
        ctx.end()
    }
}
```

Here, we send the ID of the cat we received from the request to one of our listeners using the **`request()`** method, and we specify that the type of our message is **`Int`**. We also use the same address we specified in the consumer code.

Since we have split our code into a new verticle, we need to remember to start it as well. Add the following line to both the **`main()`** function and the **`setup()`** method in your test:

```
vertx.deployVerticle(CatsVerticle())
```

Next, let's learn how to send complex objects over Event Bus.

Sending JSON over Event Bus

As our final exercise, let's learn how to update a cat. For that, we'll need to send more than just an ID over Event Bus.

Let's rewrite our **`put`** handler, as follows:

```
router.put("/:id").handler { ctx ->
    launch {
        val id = ctx.request().getParam("id").toInt()
        val body: JsonObject =
ctx.bodyAsJson.mergeIn(json{            obj("id" to
id)
        })
        vertx.eventBus().request<Int>("cats:update",
body)
          { res ->
            ctx.end(res.result().body().toString())
        }
    }
}
```

Here, you can see that we can send JSON objects over Event Bus easily. We merge the ID we receive as a URL parameter with the rest of the request **body** and send this JSON over an Event Bus. When a response is received, we output it back to the user.

Now, let's see how we consume the event we just sent:

```
vertx.eventBus().consumer<JsonObject>("cats:update")
{req ->    launch {
        val body = req.body()
        db.preparedQuery("UPDATE cats SET name = $1,
age =            $2 WHERE ID = $3")
            .execute(
                Tuple.of(
                    body.getString("name"),
                    body.getInteger("age"),
                    body.getInteger("id")
                )
            ).await()
        req.reply(body.getInteger("id"))
    }
}
```

We moved our logic from **Router** to our **CatsVerticle** class, but since we use JSON to communicate, the code stayed almost the same. In our verticle, we listen to the **cats:update** event, and once we receive the response, we extract **name**, **age**, and **id** from the JSON object to confirm that the operation was successful.

This concludes our chapter. There is still much for you to learn about the Vert.x framework in case you're curious, but with the knowledge you've gained from this chapter at hand, you should be able to do so with some confidence.

Summary

This chapter concludes our journey into the design patterns in Kotlin. Vert.x uses actors, called verticles, to organize the logic of the application. Actors communicate between themselves using Event Bus, which is an implementation of the Observable design pattern.

We also discussed the Event Loop pattern, how it allows Vert.x to process lots of events concurrently, and why it's important not to block its execution.

Now, you should be able to write microservices in Kotlin using two different frameworks, and you can choose what approach works best for you.

Vert.x provides a lower-level API than Ktor, which means that we may think more about how we structure our code, but the resulting application may be more performant as well. Since this is the end of this book, all that's left is for me to wish you the best of luck in learning about Kotlin and its ecosystem. You can always get some help from me and other Kotlin enthusiasts by going to https://stackoverflow.com/questions/tagged/kotlin and https://discuss.kotlinlang.org/.

Happy learning!

Questions

1. What's a verticle in Vert.x?
2. What's the goal of the Event Bus?

3. Why shouldn't we block the Event Loop?