Chapter 10: Concurrent Microservices with Ktor

In the previous chapter, we explored how we should write idiomatic Kotlin code that will be readable and maintainable, as well as performant.

In this chapter, we'll put the skills we've learned so far to use by building a microservice using the **Ktor framework**. We also want this microservice to be reactive and to be as close to real life as possible. For that, we'll use the Ktor framework, the benefits of which we'll list in the first section of this chapter.

In this chapter, we will cover the following topics:

- Getting started with Ktor
- Routing requests
- Testing the service
- Modularizing the application
- Connecting to a database
- Creating new entities
- Making the test consistent
- Fetching entities
- Organizing routes in Ktor
- Achieving concurrency in Ktor

By the end of this chapter, you'll have a microservice written in Kotlin that is well tested and can read data from a PostgreSQL database and store data in it.

Technical requirements

This is what you'll need to get started:

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

This chapter will assume that you have `PostgreSQL` already installed and that you have the basic knowledge for working with it. If you don't, please refer to the official documentation: https://www.postgresql.org/docs/14/tutorial-install.html.

You can find the source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter10.

Getting started with Ktor

You're probably tired of creating to-do or shopping lists.

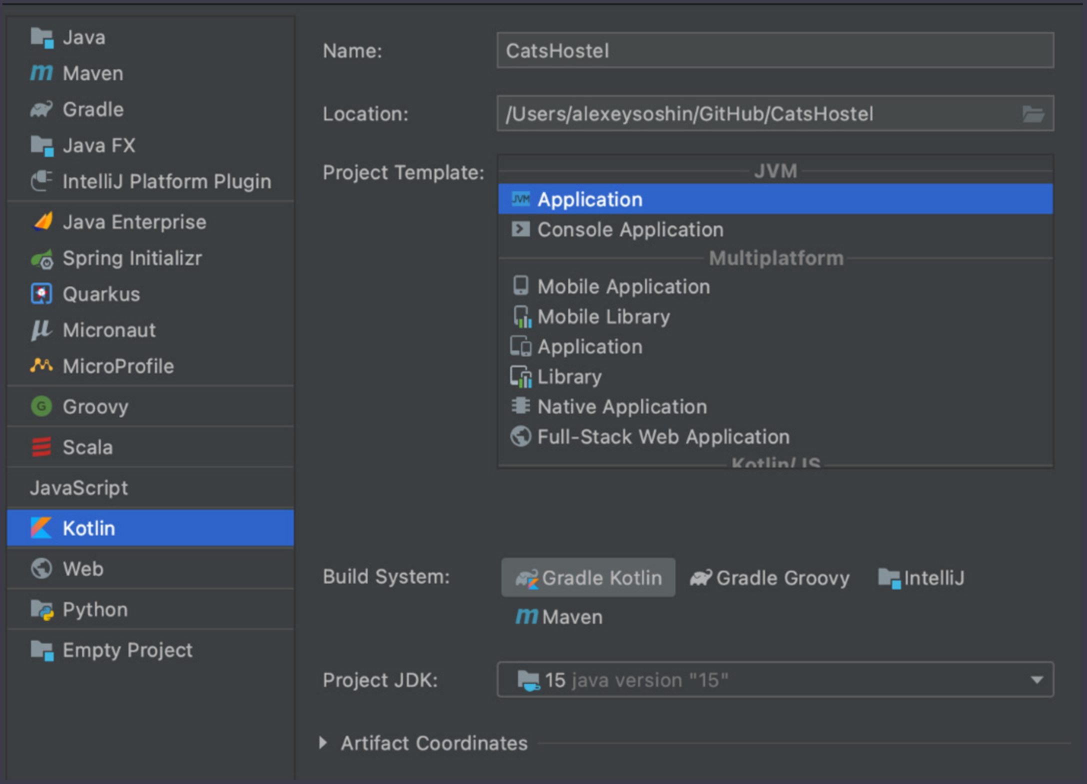
So, instead, in this chapter, the microservice will be for a `cat shelter`. The microservice should be able to do the following:

- Supply an endpoint we can ping to check whether the service is up and running
- List the cats currently in the shelter
- Provide us with a means to add new cats

The framework we'll be using for our microservice in this chapter is called **Ktor**. It's a concurrent framework that's developed and maintained by the creators of the Kotlin programming language.

Let's start by creating a new Kotlin Gradle project:

1. From your IntelliJ IDEA, select **File** | **New** | **Project** and choose **Kotlin** from **New Project** and **Gradle Kotlin** as your **Build System**.
2. Give your project a descriptive name – `CatsHostel`, in my case – and choose **Project JDK** (in this case, we are using **JDK 15**):



Name: CatsHostel

Location: /Users/alexeysoshin/GitHub/CatsHostel

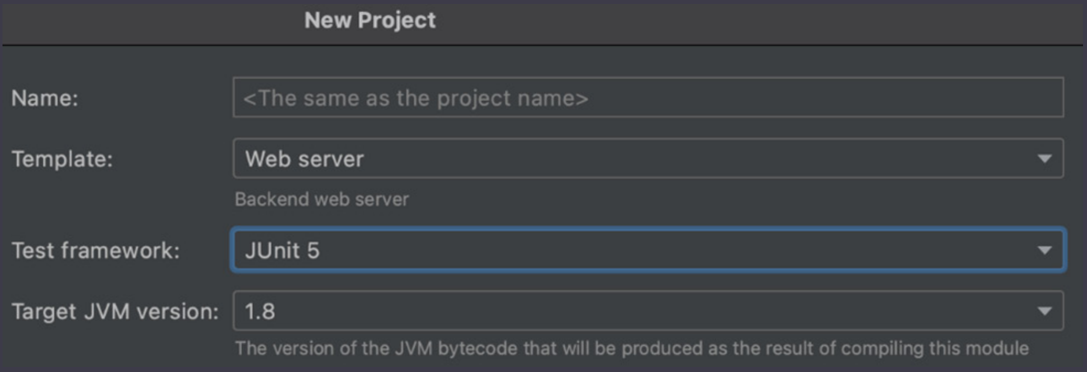
Project Template: Application

Build System: Gradle Kotlin

Project JDK: 15 java version "15"

Figure 10.1 – Selecting the Project JDK type

3. On the next screen, select **JUnit 5** as your **Test framework** and set **Target JVM version** to **1.8**. Then, click **Finish**:



New Project

Name: <The same as the project name>

Template: Web server

Backend web server

Test framework: JUnit 5

Target JVM version: 1.8

The version of the JVM bytecode that will be produced as the result of compiling this module

Figure 10.2 – Selecting Test framework and Target JVM version

4. Now, you should see the following structure:

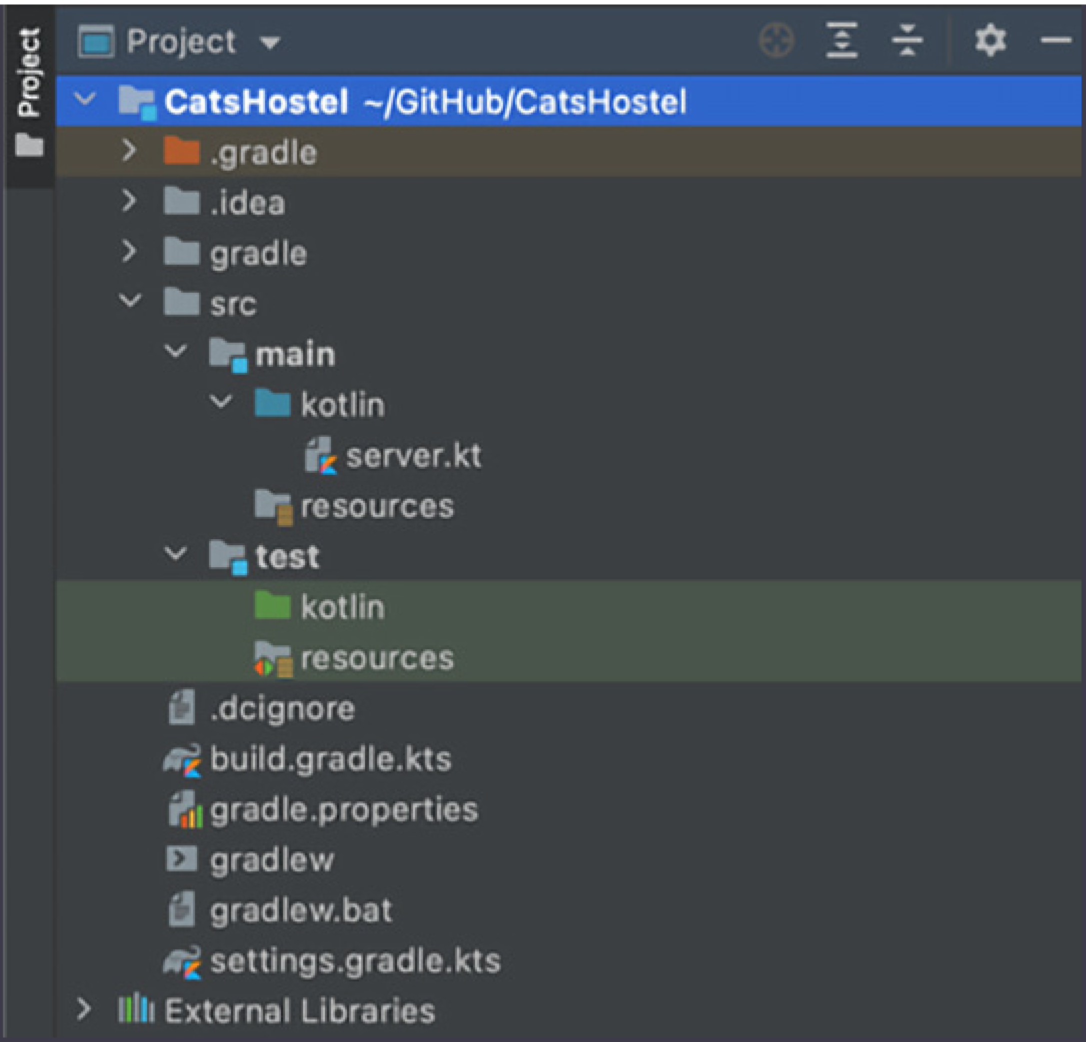


Figure 10.3 – Project structure

Next, let's open **`build.gradle.kts`**. This file controls how your project is built, its dependencies, and the libraries that the project is going to use. Depending on the version of your IntelliJ IDEA, the file's contents may differ a bit, but the general structure stays the same.

The **`.kts`** extension means that the configuration file for our Kotlin project is written in Kotlin, or to be precise, in **Kotlin Script**. Now, we would like to start using the Ktor framework to write our server. To do that, let's find our **`dependencies`** block, which should look like this:

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

```
    testRuntimeOnly("org.junit.jupiter:junit-
jupiter-        engine:5.6.0")

}
```

The preceding code mentions all the libraries that your project will be using. The **implementation()** configuration means that the library will be used at all times. The **testImplementation()** configuration means that the library will only be used during tests.

Now, let's take a look at how a library is defined in the following example:

```
"org.junit.jupiter:junit-jupiter-api:5.6.0"
```

This is a regular string that has been separated into three parts, as follows:

```
"group:name:version"
```

The **group** and **name** strings identify the library; the **version** configuration should be self-explanatory.

Now, let's modify the **dependencies** block, as follows:

```
val ktorVersion = "1.6.0"
dependencies {
    implementation("io.ktor:ktor-server-
        netty:$ktorVersion")
    ...
}
```

Since the files with **.kts** extensions are Kotlin files, we can use regular Kotlin syntax in them. In this case, we are using values and string interpolation to extract the version of our library.

The latest version of Ktor to date is **1.6.4**, but when you read this book, it will be greater than this. You can find the latest version here: https://ktor.io/.

As a general rule, all Ktor libraries should be the same version and that's when the variable becomes useful.

TIP:

If you have followed the steps from the beginning of this section, you should have a file called **`server.kt`** *in the* **`src/main/kotlin`** *folder in your project. If you don't, create one now.*

Now, let's add the following content to the **`server.kt`** file:

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") {
                call.respondText("OK")
            }
        }
    }.start(wait = true)
    println("open http://localhost:8080")
}
```

That's all the code we need to write to start a web server that will respond with **`OK`** when you open **`http://localhost:8080`** in your browser.

Now, let's understand what happens here:

- To interact with the request and return a response, we can use the **`call`** object, also known as the **context**. This object provides all the convenient methods for parsing requests and returning responses in different formats, and we'll see the different methods available for it throughout this chapter.
- The **`embeddedServer()`** function is an implementation of the Builder pattern, which we discussed in *Chapter 2*, *Working with Creational Patterns*. It allows us to configure our server. Most of the arguments have the same defaults. We override **`port`** to **`8080`** just for convenience.

- We specify the `wait` argument to be `true` so that our server will wait for incoming requests.
- The only mandatory argument to the `embeddedServer` function is the server engine. In our example, we use `Netty`, which is a very well-known JVM library, but there are others as well. The most interesting of them is `CIO`, which was developed by JetBrains themselves.

Now, let's understand what `CIO` and `Netty` are. They are both **Factory** patterns that create the actual instance of our server when invoked. This is a really interesting combination of different design patterns in one place to create a very flexible and extensible architecture.

To switch to using `CIO`, all we need to do is add a new dependency:

```
dependencies {
    ...
    implementation("io.ktor:ktor-server-
cio:$ktorVersion")
    ...
}
```

Then, we need to pass another server engine, `CIO`, to our `embeddedServer` function:

```
embeddedServer(CIO, port = 8080) {
    ...
}.start(wait = true)
```

Notice that we didn't have to change anything else in our code when we switched the server engine. That is because `embeddedServer()` uses the Bridge design pattern to make components interchangeable.

Now that our server has been started, let's investigate how we define different responses for each request to the server.

Routing requests

Now, let's take a look at the **routing** block:

```
routing {
    get("/") {
        call.respondText("OK")
    }
}
```

This block describes all the URLs that will be handled by our server. In this case, we only handle the root URL. When that URL is requested, a text response, **OK**, will be returned to the user.

The following code returns a text response. Now, let's see how we can return a JSON response instead:

```
get("/status") {
    call.respond(mapOf("status" to "OK"))
}
```

Instead of using the **respondText()** method, we'll use **respond()**, which receives an object instead of a string. In our example, we're passing a map of strings to the **respond()** function. If we run this code, though, we'll get an exception.

This is because, by default, objects are not serialized into JSON. Multiple libraries can do this for us. In this example, we'll use the **kotlinx-serialization** library. Let's start by adding it to our dependencies:

```
dependencies {
    ...
    implementation("org.jetbrains.kotlinx:kotlinx-
     serialization-json-jvm:1.3.0")
    ...
}
```

Next, we need to add the following lines before our **routing** block:

```
install(ContentNegotiation) {
    json()
}
```

Now, if we run our code again, it will output this on our browser:

```
> {"status":"OK"}
```

We've just created our first route, which returns an object serialized as JSON. Now, we can check whether our application works by opening **`http://localhost:8080/status`** in our browser. But that is a bit cumbersome. In the next section, we'll learn how to write a test for the **`/status`** endpoint.

Testing the service

To write our first test, let's create a new file called **`ServerTest.kt`** under the **`src/test/kotlin`** directory.

Now, let's add a new dependency:

```
dependencies {
    ...
    testImplementation("io.ktor:ktor-server-
        tests:$ktorVersion")
}
```

Next, let's add the following contents to our **`ServerTest.kt`** file:

```
internal class ServerTest {
    @Test
    fun testStatus() {
        withTestApplication {
            val response =
handleRequest(HttpMethod.Get,
                "/status").response
            assertEquals(HttpStatusCode.OK,
                response.status())
            assertEquals("""{"status": "OK"}""",
                response.content)
        }
    }
}
```

Tests in Kotlin are grouped into classes, and each test is a method in the class, which is marked with the **`@Test`** annotation.

Inside the **`test`** method, we start a test server, issue a **`GET`** request to the **`/status`** endpoint, and check that the endpoint responds with a correct status code and JSON body.

If you run this test now, though, it will fail, because we haven't started our server yet. To do so, we'll need to refactor it a bit, which we'll do in the next section.

Modularizing the application

So far, our server has been started from the **`main()`** function. This was simple to set up, but this doesn't allow us to test our application.

In Ktor, the code is usually organized into modules. Let's rewrite our **`main`** function, as follows:

```
fun main() {
    embeddedServer(
        CIO,
        port = 8080,
        module = Application::mainModule
    ).start(wait = true)
}
```

Here, instead of providing the logic of our server within a block, we specified a module that will contain all the configurations for our server.

This module is defined as an extension function on the **`Application`** object:

```
fun Application.mainModule() {
    install(ContentNegotiation) {
        json()
    }
    routing {
        get("/status") {
            call.respond(mapOf("status" to "OK"))
        }
    }
    println("open http://localhost:8080")
}
```

As you can see, the content of this function is the same as that of the block that we passed to our **`embeddedService`** function earlier.

Now, all we need to do is go back to our test and specify which module we would like to test:

```
@Test
fun testStatus() {
    withTestApplication(Application::mainModule) {
        ...
    }
}
```

If you run this test now, it should pass, because our server has started properly in test mode.

So far, we've only dealt with the infrastructure of our service; we haven't touched on its business logic: *managing cats*. To do so, we'll need a database. In the next section, we'll discuss how Ktor solves this problem using the Exposed library.

Connecting to a database

To store and retrieve cats, we'll need to connect to a database. We'll use PostgreSQL for that purpose, although using another SQL database won't be any different.

First, we'll need a new library to connect to the database. We'll use the Exposed library, which is also developed by JetBrains.

Let's add the following dependency to our **`build.gradle.kts`** file:

```
dependencies {
    implementation("org.jetbrains.exposed:exposed:0.1
7.14")
    implementation("org.postgresql:postgresql:42.2.24
")
    ...
}
```

Once the libraries are in place, we need to connect to them. To do that, let's create a new file called **`DB.kt`** under **`/src/main/kotlin`** with the following contents:

```
object DB {
    private val
host=System.getenv("DB_HOST")?:"localhost"
    private val port =
        System.getenv("DB_PORT")?.toIntOrNull() ?:
5432
    private val dbName = System.getenv("DB_NAME") ?:
        "cats_db"
    private val dbUser = System.getenv("DB_USER") ?:
        "cats_admin"
    private val dbPassword =
System.getenv("DB_PASSWORD")         ?: "abcd1234"
  fun connect() =
Database.connect(      "jdbc:postgresql://$host:$port
/$dbName",      driver =
"org.postgresql.Driver",      user =
dbUser,      password = dbPassword
  )
}
```

Since our application needs exactly one instance of a database, the **`DB`** object can use the Singleton pattern, which we discussed in *Chapter 2*, *Working with Creational Patterns*. For that, we will use the **`object`** keyword.

Then, for each of the variables that we need to connect to the database, we will attempt to read them from our environment. If the **`environment`** variable is not set, we will use a default value using the **Elvis** operator.

TIP:

Creating a database and a user is beyond the scope of this book, but you can refer to the official documentation for this, at https://www.postgresql.org/docs/14/app-createuser.html and https://www.postgresql.org/docs/14/app-createdb.html.

Alternatively, you can simply run the following two commands in your command line:

```
$ createuser cats_admin -W –d
$ createdb cats_db -U cats_admin
```

The first command creates a database user called **`cats_admin`** and asks you to specify a password for this user. Our application will use this **`cats_admin`** user to interact with the database. The second command creates a database called **`cats_db`** that belongs to the **`cats_admin`** user. Now that our database has been created, all we need to do is create a table that will store our cats in it.

For that, let's define another Singleton object in our **`DB.kt`** file that will represent a table:

```
object CatsTable : IntIdTable() {
    val name = varchar("name", 20).uniqueIndex()
    val age = integer("age").default(0)
}
```

Let's understand what the preceding definition means:

- **`IntIdTable`** means that we want to create a table with a primary key of the **`Int`** type.
- In the body of our object, we define the columns. In addition to the **`ID`** column, we'll have a **`name`** column that is of the **`varchar`** type, or in other words, a string, and is **`20`** characters at the most.
- The cat's **`name`** column is also unique, meaning that no two cats can have the same name.
- We also have a third column that is of the **`integer`** type, or **`Int`** in Kotlin terms, and is defaulted to **`0`**.

We'll also have a **`data`** class to represent a single cat:

```
data class Cat(val id: Int,
               val name: String,
               val age: Int)
```

The only thing that is left for us to do is add the following lines of code to our **`mainModule()`** function:

```
DB.connect()
transaction {
    SchemaUtils.create(CatsTable)
}
```

Each time our application starts, the preceding code will connect to the database. Then, it will attempt to create a table that stores our entities. If a table already exists, nothing will happen.

Now that we have established a connection to our database, let's examine how we can use this connection to store a few cats in it.

Creating new entities

Our next task is adding the first cat to our virtual shelter.

Following the REST principles, it should be a **`POST`** request, where the body of the request may look something like this:

```
{"name": "Meatloaf", "age": 4}
```

We'll start by writing a new test:

```
@Test
fun `POST creates a new cat`() {
    ...
}
```

Backticks are a useful Kotlin feature that allows us to have spaces in the names of our functions. This helps us create descriptive test names.

Next, let's look at the body of our test:

```
withTestApplication(Application::mainModule) {
    val response = handleRequest(HttpMethod.Post,
"/cats") {
        addHeader(
          HttpHeaders.ContentType,
          ContentType.Application.FormUrlEncoded.toSt
ring()
        )
        setBody(
            listOf(
                "name" to "Meatloaf",
                "age" to 4.toString()
            ).formUrlEncode()
        )
    }.response
    assertEquals(HttpStatusCode.Created,
response.status())
}
```

We discussed the **withTestApplication** and **handleRequest** functions in the previous section. This time, we are using a **POST** request. These types of requests should have the correct header, so we must set those headers using the **addHeader()** function. We must also set the body to the contents discussed previously.

Finally, we must check whether the response header is set to the **Created** HTTP code.

If we run this test now, it will fail with an HTTP code of **404** since we haven't implemented the **post /cats** endpoint yet.

Let's go back to our **routing** block and add a new endpoint:

```
post("/cats") {
    ...
    call.respond(HttpStatusCode.Created)
}
```

To create a new cat, we'll need to read the body of the **POST** request. We'll use the **receiveParameters()** function for this:

```
val parameters = call.receiveParameters()
val name = requireNotNull(parameters["name"])
val age = parameters["age"]?.toInt() ?: 0
```

The **receiveParameters** function returns a case-insensitive map. First, we will attempt to fetch the cat's **name** from this map, and if there's no name in the request, we will fail the call. This will be handled by Ktor.

Then, if we didn't receive **age**, we will default it to **0** using the Elvis operator.

Now, we must insert those values into the database:

```
transaction {
    CatsTable.insert { cat ->
        cat[CatsTable.name] = name
        cat[CatsTable.age] = age
    }
}
```

Here, we open a **transaction** block to make changes to the database. Then, we use the **insert()** method, which is available on every table. Inside the **insert** lambda, the **cat** variable refers to the new row we are going to populate. We set the name of that row to the value of the **name** parameter and do the same for **age**.

If you run your test now, it should pass. But if you run it again, it will fail. That's because the name of a cat in the database is unique. Also, we don't clean the database between test runs. So, the first run creates a cat named **Meatloaf**, while the second run fails. This is because such a cat already exists.

To make our tests consistent, we need a way to clean our database between runs.

Making the tests consistent

Let's go back to our test and add the following piece of code:

```
@BeforeEach
fun setup() {
    DB.connect()
    transaction {
        SchemaUtils.drop(CatsTable)
    }
}
```

Here, we are using the **@BeforeEach** annotation on a function. As its name suggests, this code will run before each test. The function will establish a connection to the database and drop the table completely. Then, our application will recreate the table.

Now, our tests should pass consistently. In the next section, we'll learn how to fetch a cat from the database using the Exposed library.

Fetching entities

Following the REST practices, the URL for fetching all cats should be **`/cats`**, while for fetching a single cat, it should be **`/cats/123`**, where **`123`** is the ID of the cat we are trying to fetch.

Let's add two new routes for that:

```
get("/cats") {
    ...
}
get("/cats/{id}") {
    ...
}
```

The first route is very similar to the **`/status`** route we introduced earlier in this chapter. But the second round is slightly different: it uses a query parameter in the URL. You can recognize query parameters by the curly brackets around their name.

To read a query parameter, we can access the **`parameters`** map:

```
val id =
requireNotNull(call.parameters["id"]).toInt()
```

If there is an ID on the URL, we need to try and fetch a cat from the database:

```
val cat = transaction {
    CatsTable.select {
        CatsTable.id.eq(id)
    }.firstOrNull()
}
```

Here, we open a transaction and use the **select** statement to get a cat with an ID equal to what we were provided previously.

If an object was returned, we would convert it into JSON. Otherwise, we would return an HTTP code of **404**, **Not Found**:

```
if (row == null) {
    call.respond(HttpStatusCode.NotFound)
} else {
    call.respond(
        Cat(
            row[CatsTable.id].value,
            row[CatsTable.name],
            row[CatsTable.age]
        )
    )
}
```

Now, let's add a test for fetching a single cat as well:

```
@Test
fun `GET with ID fetches a single cat`() {
    withTestApplication(Application::mainModule) {
        val id = transaction {
            CatsTable.insertAndGetId { cat ->
                cat[name] = "Fluffy"
            }
        }
        val response = handleRequest(HttpMethod.Get,
            "/cats/$id").response
        assertEquals("""{"id":1,"name":
            "Fluffy","age":0}""", response.content)
    }
}
```

In this test, we create a cat using Exposed. Here, we're using a new method called **insertAndGetId**. As its name suggests, it will return the ID of a newly created row. Then, we try to fetch that cat using our newly created endpoint.

If we try to run this test, though, it will fail with the following exception:

```
> Serializer for class 'Cat' is not found.
```

By default, Ktor doesn't know how to turn our custom data class into JSON. To fix that, we'll need to add a new plugin to our **`build.gradle.kts`** file:

```
plugins {
    kotlin("jvm") version "1.5.10"
    application
    kotlin("plugin.serialization") version "1.5.10"
}
```

This plugin will create serializers at compile time for any class marked with the **`@Serializable`** annotation. All we need to do now for the test to pass is add that annotation to our **`Cat`** class:

```
@Serializable
data class Cat(
    val id: Int,
    val name: String,
    val age: Int
)
```

That's it; now, our test for fetching a cat by its ID should pass.

Finally, we would like to be able to fetch all the cats from the database. To do that, we must change our test setup a little:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class ServerTest {
    @BeforeAll
    fun setup() {
        DB.connect()
        transaction {
            SchemaUtils.create(CatsTable)
        }
    }
    @AfterAll
    fun cleanup() {
        DB.connect()
        transaction {
            SchemaUtils.drop(CatsTable)
        }
    }
    ...
}
```

Here, we changed the setup of our test to drop the table once all the tests have been executed. So, instead of the **`@BeforeEach`** annotation, which executes the function before each test, we use the **`@AfterAll`** annotation, which executes the function after all tests have been executed.

For this annotation to work, we also need to add the **`@TestInstance`** annotation to our class. The default for that is **`PER_METHOD`**, but since we want to execute multiple tests at once, and then clean up after, we need to set the life cycle of our test class to **`PER_CLASS`**.

Next, let's wrap our test into a nested class, like this:

```
@Nested
inner class `With cat in DB` {

    @Test
    fun `GET with ID fetches a single cat`() {
        ...
    }
}
```

Nested test classes are a great way to encapsulate specific test situations. In our case, we would like to run two tests when there is a cat in our database already.

Now, let's add the following setup code to our nested test:

```
lateinit var id: EntityID<Int>
@BeforeEach
fun setup() {
    DB.connect()
    id = transaction {
        CatsTable.insertAndGetId { cat ->
            cat[name] = "Fluffy"
        }
    }
}
@AfterEach
fun teardown() {
    DB.connect()
    transaction {
        CatsTable.deleteAll()
    }
}
```

Before we execute each test in this nested class, we will create a cat in the database and after each test, we will delete all the cats from our database. Since we would like to keep track of the ID of the cat that we create, we will store it in a variable.

Now, our test class for fetching a single entity looks like this:

```
@Test
fun `GET with ID fetches a single cat`() {
    withTestApplication(Application::mainModule) {
        val response = handleRequest(HttpMethod.Get,
            "/cats/$id").response
        assertEquals("""
{"id":$id,"name":"Fluffy",            "age":0}""",
response.content)
    }
}
```

Notice that we interpolate the cat's ID into our expected response since it will change with each test execution.

The test for fetching all the cats from the database will look almost the same:

```
@Test
fun `GET without ID fetches all cats`() {
    withTestApplication(Application::mainModule) {
        val response = handleRequest(HttpMethod.Get,
            "/cats").response
        assertEquals("""
[{"id":$id,"name":"Fluffy",            "age":0}]""",
response.content)
    }
}
```

We just don't specify the ID, and the response is wrapped into a JSON array, as you can see by the square brackets around it.

Now, all we need to do is implement this new route:

```
get("/cats") {
    val cats = transaction {
        CatsTable.selectAll().map { row ->
            Cat(
                row[CatsTable.id].value,
                row[CatsTable.name],
                row[CatsTable.age]
            )
        }
    }
    call.respond(cats)
}
```

If you followed the example for fetching a single entity from the database (from the beginning of this section), then this example won't be very different for you. We use the **`selectAll()`** function to fetch all the rows from the table. Then, we map each row to our **`data`** class. The only problem that is left for us to solve is that our code is quite messy and resides in a single file. It would be better if we split all the cat routes into a separate file. We'll do that in the next section.

Organizing routes in Ktor

In this section, we'll see what the idiomatic approach in Ktor is for structuring multiple routes that belong to the same domain.

Our current **`routing`** block looks like this:

```
routing {
    get("/status") {
        ...
    }
    post("/cats") {
        ...
    }
    get("/cats") {
        …
    }
    get("/cats/{id}") {
        ...
    }
}
```

It would be good if we could extract all the routes that are related to cats into a separate file. Let's start by replacing all the cat routes with a function:

```
routing {
    get("/status") {
        ...
    }
    cats()
}
```

If you are using IntelliJ IDEA, it will even suggest that you generate an extension function on the **`Routing`** class:

```
fun Routing.cats() {
    ...
}
```

Now, we can move all our cat routes to this function:

```
fun Routing.cats() {
    post("/cats") {
        ...
    }
    get("/cats") {
        ...
    }
    get("/cats/{id}") {
        ...
    }
}
```

Now, you can see that the **`/cats`** URL is repeated many times. We can lift it using the **`route()`** block:

Before	After
`fun Routing.cats() {`	`fun Routing.cats() {`
`    post("/cats") {`	`    route("/cats") {`
`        ...`	`        post {`
`    }`	`            ...`
`    get("/cats") {`	`        }`
`        ...`	`        get {`
`    }`	`            ...`
`    get("/cats/{id}") {`	`        }`
`        ...`	`        get("/{id}") {`
`    }`	`            ...`
`}`	`        }`
	`    }`
	`}`

Table 10.1 - Cleaner code after using the route() block

Notice how much cleaner our code has become now.

Now, there's one last important topic for us to cover. At the beginning of this chapter, we mentioned that Ktor is a highly concurrent framework. And in *Chapter 6*, *Threads and Coroutines*, we said that concurrency in Kotlin is mainly achieved by using coroutines. But we have started a single coroutine in this chapter. We'll look at this in the next section.

Achieving concurrency in Ktor

Looking back at the code we've written in this chapter, you may be under the impression that the Ktor code is not concurrent at all. However, this couldn't be further from the truth.

All the Ktor functions we've used in this chapter are based on coroutines and the concept of **suspending functions**.

For every incoming request, Ktor will start a new coroutine that will handle it, thanks to the CIO server engine, which is based on coroutines at its core. Having a concurrency model that is performant but not obtrusive is a very important principle in Ktor.

In addition, the **`routing`** blocks we used to specify all our endpoints have access to **`CoroutineScope`**, meaning that we can invoke suspending functions within those blocks.

One of the examples for such a suspending function is **`call.respond()`**, which we were using throughout this chapter. Suspending functions provide our application with opportunities to context switch, and to execute other code concurrently. This means that the same number of resources can serve far more requests than they would be able to otherwise. We'll stop here and summarize what we've learned about developing applications using Ktor.

Summary

In this chapter, we have built a well-tested service using Kotlin that uses the Ktor framework to store entities in the database. We've also discussed how the multiple design patterns that we encountered at the beginning of this book, such as Factory, Singleton, and Bridge, are used in the Ktor framework to provide a flexible structure for our code.

Now, you should be able to interact with the database using the Exposed framework. We've learned how we can declare, create, and drop tables, how to insert new entities, and how to fetch and delete them.

In the next chapter, we'll look at an alternative approach to developing web applications, but this time using a Reactive framework called **Vert.x**. This will allow us to compare the concurrent and Reactive approaches for developing web applications and discuss the tradeoffs of each of the approaches.

Questions

1. How are the Ktor applications structured and what are their benefits?
2. What are plugins in Ktor and what are they used for?

3. What is the main problem that the Exposed library solves?