PRACTICAL FP IN SCALA

A HANDS-ON APPROACH

GABRIEL VOLPE

Practical FP in Scala

A hands-on approach

Gabriel Volpe

May 11, 2020

First Edition

Contents

Preface

Scala is a hybrid language that mixes both the Object-Oriented Programming (OOP) and Functional Programming (FP) paradigms. This allows you to get up-and-running pretty quickly without knowing the language in detail. Over time, as you learn more, you are hopefully going to appreciate what makes Scala great: its *functional building blocks*.

Pattern matching, folds, recursion, higher-order functions, etc. If you decide to continue down this road, you will discover the functional subset of the community and its great ecosystem of libraries.

Sooner rather than later, you will come across the Cats[1] library and its remarkable documentation. You might even start using it in your projects! Once you get familiar with the power of typeclasses such as `Functor`, `Monad`, and `Traverse`, I am sure you will love it!

As you evolve into a functional programmer, you will learn about purely functional effects and referential transparency. You might as well start using the popular `IO` Monad present in Cats Effect[2] and other similar libraries.

One day you will need to process a lot of data that doesn't fit into memory; a suitable solution to this engineering problem is streaming. While searching for a valuable candidate, you might stumble upon a purely functional streaming library: Fs2[3]. You will quickly learn that it is also a magnificent library for control flow.

You might decide to adopt a message broker as a communication protocol between microservices and to distribute data. You name it: `Kafka`, `RabbitMQ`, to mention a few. Each of these wonderful technologies has a library that can fulfill every client's needs.

Unless you have taken the *stateless* train, you will need a database or a cache as well. Whether it is `PostgreSQL`, `ElasticSearch`, or `Redis`, the Scala FP ecosystem of libraries has got your back.

So far so good! There seems to be a wide set of tools available to write a complete purely functional application and finally ditch the enterprise framework.

[1] https://typelevel.org/cats
[2] https://typelevel.org/cats-effect
[3] https://fs2.io

At this point, you find yourself in a situation where many programmers that are enthusiastic about functional programming find themselves: *needing to deliver business value in a time-constrained manner*.

Answering this and many other fundamental questions are the aims of this book. Even if at times it wouldn't give you a straightforward answer, it will show you the way. It will give you choices and hopefully enlighten you.

Throughout the following chapters, we will develop a shopping cart application that tackles system design from different angles. We will architect our system, making both sound business and technical decisions at every step, using the best possible techniques I am knowledgeable of at this moment.

Acknowledgments

First of all, I'd like to thank my beloved partner Alicja for her endless support.

A huge thanks to my friend John Regan for his invaluable feedback, which has raised the bar on my writing skills onto the next level.

Special thanks to the talented @impurepics[1], author of the book's cover.

Thanks to Jakub Kozłowski for reviewing almost every pull request of the book and the Shopping Cart application.

Thanks to my OSS friends, Fabio Labella, Frank Thomas, Luka Jacobowitz, Piotr Gawryś, Rob Norris and Ross A. Baker, for proofreading the first drafts of the book.

Thanks to the +500 early readers who supported my work for the extra motivation and the early feedback.

Last but not least, thanks to the free tools that have made this book possible: NeoVim[2], Pandoc[3], LaTeX[4] and CSS[5].

[1] https://twitter.com/impurepics
[2] https://neovim.io/
[3] https://pandoc.org/
[4] https://www.latex-project.org/
[5] https://en.wikipedia.org/wiki/Cascading_Style_Sheets

Dependency versions

At the moment of writing, all the standalone examples use Scala 2.13.0 and Sbt 1.3.8, as well as the following dependencies defined in this minimal `build.sbt` file:

```
ThisBuild / scalaVersion := "2.13.0"

libraryDependencies ++= Seq(
  compilerPlugin(
    "org.typelevel" %% "kind-projector" % "0.11.0" cross CrossVersion.full
  ),
  compilerPlugin(
    "org.augustjune" %% "context-applied" % "0.1.2"
  ),
  "org.typelevel"  %% "cats-core"           % "2.1.0",
  "org.typelevel"  %% "cats-effect"         % "2.1.0",
  "dev.profunktor" %% "console4cats"        % "0.8.1",
  "org.manatki"    %% "derevo-cats"         % "0.10.5",
  "org.manatki"    %% "derevo-cats-tagless" % "0.10.5",
  "co.fs2"         %% "fs2-core"            % "2.2.2",
  "com.olegpy"     %% "meow-mtl-core"       % "0.4.0",
  "com.olegpy"     %% "meow-mtl-effects"    % "0.4.0,
  "io.estatico"    %% "newtype"             % "0.4.3",
  "eu.timepit"     %% "refined"             % "0.9.12",
  "com.github.julien-truffaut" %% "monocle-core" % "2.0.1",
  "com.github.julien-truffaut" %% "monocle-macro" % "2.0.1"
)

scalacOptions += "-Ymacro-annotations"
```

The `sbt-tpolecat` plugin is also necessary. Here is a minimal `plugins.sbt` file:

```
addSbtPlugin("io.github.davidgregory084" % "sbt-tpolecat" % "0.1.6")
```

Please note that Scala Steward[1] keeps on updating the project's dependencies on a daily basis, which may not reflect the versions described in this book.

[1] https://github.com/fthomas/scala-steward

Prerequisites

This book is considered intermediate to advanced. Familiarity with functional programming concepts and basic FP libraries such as Cats and Cats Effect will be of tremendous help even though I will do my best to be as clear and concise as I can.

Recommended sources for learning these concepts are the Scala with Cats[1] book and the Cats Effect[2] official documentation.

The following list details the topics required to understand this book.

- Higher-Kinded Types (HKTs)[3].
- Typeclasses[4].
- IO Monad[5].
- Referential Transparency[6].

These topics are not going to be explained in this book but some examples may be shown. The reader is expected to be acquainted with them.

If the requirements feel overwhelming, it is not because the entire book is difficult, but rather because some specific parts are. You can try and read it, and if at some point you get stuck, you can skip that section. You could also make a pause, go to read about these resources, and then continue where you left off.

Remember that we are going to develop an application together, which will help you learn a lot, even if you haven't employed these techniques and libraries before.

[1] https://underscore.io/books/scala-with-cats/
[2] https://typelevel.org/cats-effect/
[3] https://typelevel.org/blog/2016/08/21/hkts-moving-forward.html
[4] https://typelevel.org/cats/typeclasses.html
[5] https://typelevel.org/cats-effect/datatypes/io.html
[6] https://en.wikipedia.org/wiki/Referential_transparency

How to read this book

For conciseness, most of the imports and some datatype definitions are elided from the book, so it is recommended to read it by following along the two Scala projects that supplement it:

- pfps-examples[1]: Standalone examples.
- pfps-shopping-cart[2]: Shopping cart application.

The first project includes self-contained examples that demonstrate some features or techniques explained independently.

The latter contains the source code of the full-fledged application that we will develop in the next nine chapters, including a test suite and instructions on how to deploy it.

Bear in mind that the presented Shopping Cart application only acts as a guideline. To get a better learning experience, readers are encouraged to write their own application from scratch; getting your hands dirty is the best way to learn.

There is also a Gitter channel[3] where you are welcome to ask any kind of questions related to the book or functional programming in general.

[1] https://github.com/gvolpe/pfps-examples
[2] https://github.com/gvolpe/pfps-shopping-cart
[3] https://gitter.im/pfp-scala/community

Conventions used in this book

Colored boxes might indicate either notes, tips, or warnings.

Notes

`A note on what's being explained`

Tips

`A tip about a particular topic`

Warning

`The author might be biased towards some claim or decision`

If you are reading this on Kindle, you won't see colors, unfortunately.

Chapter 1: Shopping Cart project

Here is the beginning of our endeavor. We will develop a shopping cart application utilizing the best libraries, architecture, and design patterns I am aware of. We are going to start with understanding the business requirements and see how we can materialize them into our system design.

By the end of this chapter, we should have a clearer view of the business expectations.

Business requirements

A Guitar store located in the US has hired our services to develop the backend system of their online store. The requirements are clear to the business. However, they don't know much about what the necessities of the backend might be. So this is our task. We are free to architect and design the backend system in the best way possible.

For now, they only need to sell guitars. Though, in the future, they want to add other products. Here are the requirements we have got from them:

- A guest user should be able to:
  - register into the system with a unique username and password.
  - login into the system given some valid credentials.
  - see all the guitar catalog as well as to search per brand.
- A registered user should be able to:
  - add products to the shopping cart.
  - remove products from the shopping cart.
  - modify the quantity of a particular product in the shopping cart.
  - check out the shopping cart, which involves:
    - sending the user Id and cart to an external payment system (see below).
    - persisting order details including the Payment Id.
  - list existing orders as well as retrieving a specific one by Id.
  - log out of the system.
- An admin user should be able to:
  - add brands.
  - add categories.
  - add products to the catalog.
  - modify the prices of products.
- The frontend should be able to:
  - consume data using an HTTP API that we need to define.

Notes

```
For now, there will be a single admin user created manually
```

Third-party Payments API

The external payment system exposes an HTTP API. We are told it is idempotent, meaning that it is capable of handling duplicate payments. If we happen to make a

request for the same payment twice, we are going to get a specific HTTP response code containing the Payment Id.

POST *Request body*

```
{
  "user_id": "hf8hf...",
  "total": 324.35,
  "card": {
    "name": "Albert Einstein",
    "number": 5555222288881111,
    "expiration": "0821",
    "ccv": 123
  }
}
```

Response body on success

```
{
  "payment_id":"eyJ0eXA..."
}
```

- **Response codes**:
  - **200**: the payment was successful.
  - **400**: e.g. invalid request body.
  - **409**: duplicate payment (returns Payment Id).
  - **500**: unknown server error.

With this information, we should be able to design the system and get back to the business with our proposal.

Identifying the domain

We could try to represent guitars as a generic `Item` since, in the future, they want to add other products. A possible sketch of our domain model is presented below.

Tips

`Understanding the product is fundamental to design a good system`

Item

- `uuid`: a unique item identifier.
- `model`: the item's model (guitar model to start with).
- `brand`: a relationship with a `Brand` entity.
- `category`: a relationship with a `Category` entity.
- `description`: more information about the item.
- `price`: we will use USD as the currency with two decimal digits.

Brand

- `name`: the unique name of the brand (cannot be duplicated).

Category

- `name`: the name of the category (cannot be duplicated).

Cart

- `uuid`: a unique cart identifier.
- `items`: a key-value store (Map) of item ids and quantities.

Order

- `uuid`: a unique order identifier.
- `paymentId`: a unique payment identifier given by a 3rd party client.
- `items`: a key-value store (Map) of item ids and quantities.
- `total`: the total amount of the order, in USD.

Card

- `name`: card holder's name.
- `number`: a 16-digit number.
- `expiration`: a 4-digit number as a string, to not lose zeros: the first two digits indicate the month, and the last two, the year, e.g. "0821".
- `cvv`: a 3-digit number. CVV stands for Card Verification Value.

Basic details of any credit or debit card.

Let's now continue with the representation of the users. Based on the requirements, we know there are guest users, registered users, and admin users. Let's try to write the domain model for them.

Guest User

Since we don't know anything about such users, we are not going to represent them in our domain model, but we know they should be able to register and login to the system given some valid credentials. We are going to see in Chapter 5, that this belongs to the HTTP request body of our authentication endpoints.

User

It represents a registered user that has been logged into the system.

- `uuid`: a unique user identifier.
- `username`: a unique username registered in the system.
- `password`: the username's password.

Admin User

It has special permissions, such as adding items into the system's catalog.

- `uuid`: a unique admin user identifier.
- `username`: a unique admin username.

Identifying HTTP endpoints

Our API should be versioned to allow a smooth evolution as the requirements change in the future. Therefore, all the endpoints will start with the `/v1` prefix. First, a quick few notes:

Notes about format

```
The first item describes the HTTP endpoint and the sub-items below
outline the possible response statuses the endpoint can return
```

Notes about error codes

```
Every endpoint is implicitly assumed to possibly fail with the 500
status code - Internal Server Error (e.g. the database is down)
```

Open Routes

These are the HTTP routes that don't require authentication.

Authentication routes

- `POST` **/users**
  - **201**: the user was successfully created.
  - **400**: invalid input data, e.g. empty username.
  - **409**: the username is already taken.

Request body

```
{
  "username": "csagan",
  "password": "<c05m05>"
}
```

Response body on success

```
{
  "access_token":"eyJ0eXA..."
}
```

- `POST` **/auth/login**
  - **200**: the user was successfully logged in.
  - **403**: e.g. invalid username or credentials.

Request body

```
{
  "username": "csagan",
  "password": "<c05m05>"
}
```

Response body on success

```
{
  "access_token":"eyJ0eXA..."
}
```

- `POST` **/auth/logout**
  - **204**: the user was successfully logged out.

No *request body* required and no *response body* given.

Brand routes

- `GET` **/brands**
  - **200**: returns a list of brands.

Response body on success

```
[
  {
    "uuid": "7a465b27-0db...",
    "name": "Fender"
  },
  {
    "uuid": "f40e8104-9be...",
    "name": "Gibson"
  }
]
```

Category routes

- `GET` **/categories**
  - **200**: returns a list of categories.

Response body on success

```
[
  {
    "uuid": "10739c61-c93...",
    "name": "Guitars"
  }
]
```

Item routes

- `GET` **/items**
  - **200**: returns a list of items.

Response body on success

```
[
  {
    "uuid": "509b77fd-3a...",
    "name": "Telecaster",
    "description": "Classic guitar",
    "price": 578,
    "brand": {
      "uuid": "7a465b27-0d...",
      "name": "Fender"
    },
    "category": {
      "uuid": "10739c61-c93...",
      "name": "Guitars"
    }
  }
]
```

To search by brand, we will have an optional query parameter: `brand`.

- `GET` **/items?brand=gibson**
  - **200**: returns a list of items.

Secured Routes

These are the HTTP routes that require registered users to be logged in. All of them can return the following response statuses in addition to the specific ones.

- **401**: unauthorized user, needs to log in.
- **403**: the user does not have permission to perform this action.

Cart routes

- `GET` **/cart**
  - **200**: returns the cart for the current user.

Response body on success

```
{
  "items": [
    {
      "item": {
        "uuid": "509b77fd-3a...",
        "name": "Telecaster",
        "description": "Classic guitar",
        "price": 578,
        "brand": {
          "uuid": "7a465b27-0d...",
          "name": "Fender"
        },
        "category": {
          "uuid": "10739c61-c93...",
          "name": "Guitars"
        }
      },
      "quantity": 4
    }
  ],
  "total": 2312
}
```

- `POST` **/cart**
  - **201**: the item was added to the cart.
  - **409**: the item is already in the cart.

Request body

```
{
    "items": {
      "509b77fd-3a...": 4
  }
}
```

No *Response body*.

- `PUT` **/cart**
  - **200**: the quantity of some items were updated in the cart.
  - **400**: quantities must be greater than zero.

Request body

```
{
    "items": {
      "509b77fd-3a...": 1
  }
}
```

No *Response body*.

- `DELETE` **/cart/{itemId}**
  - **204**: the specified item was removed from the cart, if it existed.

No *Request body* and no *Response body*.

Checkout routes

- `POST` **/checkout**
  - **201**: the order was processed successfully.
  - **400**: e.g. invalid card number.

Request body

```
{
  "name": "Isaac Newton",
  "number": 1111444422223333,
  "expiration": "0422",
  "ccv": 131
}
```

Response body

```
{
  "order_id": "gf34y54g..."
}
```

Order routes

- `GET` **/orders**
  - **200**: returns the list of orders for the current user.

Response body on success

```
[
  {
    "uuid": "54312359...",
    "payment_id": "Ex4dfd4...",
    "items": [
      {
        "uuid": "14427832...",
        "quantity": 1
      },
      {...}
    ],
    "total": 3769.45
  }
]
```

- `GET` **/orders/{orderId}**
  - **200**: returns specific order for the current user.
  - **404**: order not found.

Response body on success

```
{
  "uuid": "54312359...",
  "payment_id": "Ex4dfd4...",
  "items": [
    {
      "uuid": "14427832...",
      "quantity": 1
    },
    {...}
  ],
  "total": 3769.45
}
```

Admin Routes

These are the HTTP routes that can be accessed only by administrators with a specific *API Access Token*. All of the following response statuses can be returned in addition to the specific ones.

- **401**: unauthorized user, needs to login.
- **403**: the user does not have permissions to perform this action.

Brand routes

- `POST` **/brands**
  - **201**: brand successfully created.
  - **409**: the brand name is already taken.

Request body

```
{
  "name": "Ibanez"
}
```

No *Response body*.

Category routes

- `POST` **/categories**
  - **201**: category successfully created.
  - **409**: the category name is already taken.

Request body

```
{
  "name": "Guitars"
}
```

No *Response body*.

Item routes

- `POST` **/items**
  - **201**: items successfully created.
  - **409**: some of the items already exist.

Request body

```
{
    "name": "Telecaster",
    "description": "Classic guitar",
    "price": 578,
    "brandId": "7a465b27-0d...",
    "categoryId": "10739c61-c9..."
}
```

No *Response body*.

- `PUT` **/items**
  - **200**: item's price successfully updated.
  - **400**: the price must be greater than zero.

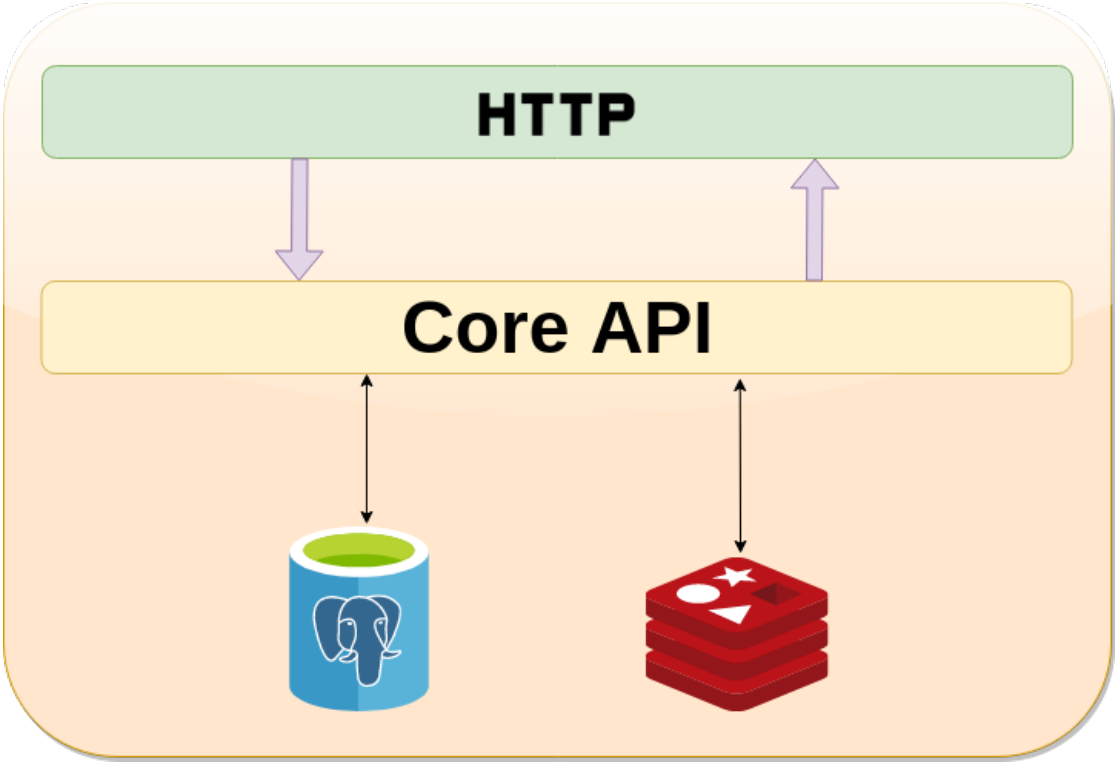
Request body

```
{
    "uuid": "509b77fd-3a...",
    "description": "Classic guitar",
    "price": 5046.14,
    "brandId": "7a465b27-0d...",
    "categoryId": "10739c61-c9..."
}
```

No *Response body*.

Architecture

This is what our system's architecture will look like in the end:



In addition to the HTTP API, we will have a PostgreSQL database as our main storage system as well as a caching layer backed by Redis to store session tokens and other data, such as the current shopping cart of a user.

Technical stack

Below is the complete list of all the libraries we will be using in our application:

- `cats`: basic functional blocks. From typeclasses such as `Functor` to syntax and instances for some datatypes and monad transformers.
- `cats-effect`: concurrency and functional effects. It ships the default `IO` monad.
- `cats-mtl`: typeclasses for monad transformer capabilities.
- `cats-retry`: retrying actions that can fail in a purely functional fashion.
- `circe`: standard JSON library to create encoders and decoders.
- `ciris`: flexible configuration library with support for different environments.
- `fs2`: powerful streaming in constant memory and control flow.
- `http4s`: purely functional HTTP server and client, built on top of `fs2`.
- `http4s-jwt-auth`: opinionated JWT authentication built on top of `jwt-scala`.
- `log4cats`: standard logging framework for Cats.
- `meow-mtl`: classy optics for `cats-mtl` typeclasses.
- `newtype`: zero-cost wrappers for strongly typed functions.
- `redis4cats`: client for Redis compatible with `cats-effect`.
- `refined`: refinement types for type-level validation.
- `skunk`: purely functional, non-blocking PostgreSQL client.
- `squants`: strongly-typed units of measure such as "money".

A note on Cats Effect

At the moment of writing, the design of the next major version of Cats Effect (CE for short) is being actively discussed[1]; as a result, some typeclasses or datatypes we are going to see in this book might not be part of CE3.

Caveat disclosed, all the design patterns and best practices using CE2 will still be relevant even if we need to adapt some code.

[1] https://github.com/typelevel/cats-effect/issues/634

Chapter 2: Design patterns

Here we are going to explore some well-known and some non-standard design patterns. The latter being biased towards my preferences, though.

These patterns will appear at least once in the application we will develop, so you can think of this chapter as a preparation for what's to come.

Strongly-typed functions

One of the most significant benefits of functional programming is that it lets us reason about functions by looking at their type signature. Yet, the truth is that these are commonly created by us, imperfect humans, who often end up with weakly-typed functions.

For instance, let's look at a function that takes two parameters `username` and `email`.

```
def lookup(username: String, email: String): F[Option[User]]
```

Do you see any problems with this function? Let's see how we can use it.

```
$ lookup("aeinstein@research.com", "aeinstein")
$ lookup("aeinstein", "123")
$ lookup("", "")
```

See the issue? It is not only easy to confuse the order of the parameters but is also straightforward to feed our function with invalid data! So what can we do about it? We could make this better by introducing *value classes*.

Value classes

In vanilla Scala, we can wrap a single field and extend the `AnyVal` abstract class to avoid some runtime costs. Here is how we can define value classes for `username` and `email`:

```
case class Username(val value: String) extends AnyVal
case class Email(val value: String) extends AnyVal
```

Now we can re-define our function using these types.

```
def lookup(username: Username, email: Email): F[Option[User]]
```

Notice that we can no longer confuse the order of the parameters.

```
$ lookup(Username("aeinstein"), Email("aeinstein@research.com"))
```

Or can we?

```
$ lookup(Username("aeinstein@research.com"), Email("aeinstein"))
$ lookup(Username("aeinstein"), Email("123"))
$ lookup(Username(""), Email(""))
```

Fine, we are doing this on purpose. But the important take away is that the compiler doesn't help us and that is all we need. A way around this is to make the case class constructors private and give the user *smart constructors*.

```
case class Username private(val value: String) extends AnyVal
case class Email private(val value: String) extends AnyVal

def mkUsername(value: String): Option[Username] =
  if (value.nonEmpty) Username(value).some
  else none[Username]

def mkEmail(value: String): Option[Email] =
  if (value.contains("@")) Email(value).some
  else none[Email]
```

Smart constructors are functions such as `mkUsername` and `mkEmail`, which take a raw value and return an optional validated one. The optionality can be denoted using types such as `Option`, `Either`, `Validated`, or any other higher-kinded type.

So let's pretend that these functions validate the raw values properly and give us back some valid data. We can now use them in the following way:

```
(
  mkUsername("aeinstein"),
  mkEmail("aeinstein@research.com")
).mapN {
  case (username, email) => lookup(username, email)
}
```

But guess what? We can still do wrong...

```
(
  mkUsername("aeinstein"),
  mkEmail("aeinstein@research.com")
).mapN {
  case (username, email) =>
    lookup(username.copy(value = ""), email)
}
```

Unfortunately, we are still using case classes, which means the `copy` method is still there. A proper way to finally get around this issue is to use `sealed abstract class`es.

```
sealed abstract class Username(value: String)
sealed abstract class Email(value: String)
```

Having this encoding in combination with smart constructors will mitigate the issue at the cost of boilerplate and more memory allocation.

Newtypes

Value classes are fine if used with caution, but we haven't talked about their limitations and performance issues. In many cases, Scala needs to allocate extra memory when using value classes, as described in the article Value classes and universal traits[1]. Quoting the relevant part:

> A value class is actually instantiated when:
>
> - a value class is treated as another type.
> - a value class is assigned to an array.
> - doing runtime type tests, such as pattern matching.

The language cannot guarantee that these primitive type wrappers won't actually allocate more memory, in addition to the pitfalls described in the previous section.

Thus, my recommendation is to avoid value classes and sealed abstract classes completely and instead use the Newtype[2] library, which gives us *zero-cost wrappers* with no runtime overhead.

This is how we can define our data using newtypes:

```
import io.estatico.newtype.macros._

@newtype case class Username(value: String)
@newtype case class Email(value: String)
```

It uses macros, for which we need the macro paradise compiler plugin in Scala versions below 2.13.0, and only an extra compiler flag `-Ymacro-annotations` in versions 2.13.0 and above.

Despite eliminating the extra allocation issue and removing the `copy` method, notice how we can still trigger the functionality incorrectly:

```
Email("foo")
```

This means that smart constructors are still needed to avoid invalid data.

Notes

`Newtypes will eventually be replaced by opaque types`

[1] https://docs.scala-lang.org/overviews/core/value-classes.html
[2] https://github.com/estatico/scala-newtype

Opaque types[3] are a feature of *Scala 3*, formerly known as Dotty[4]. This new version of the language is expected to be released by the end of 2020.

Newtypes can also be constructed using the `coerce` method in the following way:

```
import io.estatico.newtype.ops._

"foo".coerce[Email]
```

Though, this is considered an *anti-pattern* since its implementation uses a safe cast (via `asInstanceOf`), having proof that the coercion is lawful via its `Coercible` typeclass instance. Hence, if we know the concrete type, we should always prefer to use the `apply` method for constructing newtypes.

In Chapter 5, we are going to see the correct usage of the `coerce` method, when writing generic JSON codecs.

Refinement types

We have seen how newtypes help us tremendously in our strongly-typed functions quest. Nevertheless, it requires smart constructors to validate input data, which adds boilerplate and leaves us with a bittersweet feeling. Therefore, taking us to our last hope: refinement types and the Refined[5] library.

Refinement types allow us to validate data in compile time as well as in runtime. Let's look at the example below.

```
import eu.timepit.refined._
import eu.timepit.refined.auto._
import eu.timepit.refined.types.string.NonEmptyString

def lookup(username: NonEmptyString): F[Option[User]]
```

We are saying that a valid username is any non-empty string; though, we could also say that a valid username is any string containing the letter "g", in which case, we would need to define a custom refinement type instead of using a built-in one like `NonEmptyString`. The following example demonstrates how we can do this.

```
import eu.timepit.refined.api.Refined
import eu.timepit.refined.collection.Contains

type Username = String Refined Contains['g']
```

[3] https://docs.scala-lang.org/sips/opaque-types.html
[4] http://dotty.epfl.ch/
[5] https://github.com/fthomas/refined

```
def lookup(username: Username): F[Option[User]]
```

By saying that it should contain a letter "g" (using string literals), we are also implying that it should be non-empty. If we try to pass some invalid arguments, we are going to get a compiler error.

```
$ lookup("")          // error
$ lookup("aeinstein") // error
$ lookup("csagan")    // compiles
```

Refinement types are great and let us define custom validation rules. Though, in many cases, a simple rule applies to many possible types. For example, a `NonEmptyString` applies to almost all our inputs. In such cases, we can combine forces and use Refined and Newtype together!

```
@newtype case class Brand(value: NonEmptyString)
@newtype case class Category(value: NonEmptyString)

val brand: Brand = Brand("foo")
```

These two types share the same validation rule, so we use refinement types, but since they represent different concepts, we create a newtype for each of them. This combination is ever so powerful that I couldn't recommend it enough.

Another feature that makes the Refined library very appealing is its integration with multiple libraries such as Circe[6], Doobie[7], and Monocle[8], to name a few. Having support for these third-party libraries means that we don't need to write custom refinement types to integrate with them as the most common ones are provided out of the box.

This is all I have on this topic. I hope it was enough to convince you of the benefits. Though, to get acquainted with this technique, we need some practice which we are going to get throughout the development of the shopping cart application.

[6] https://github.com/circe/circe
[7] https://github.com/tpolecat/doobie
[8] https://github.com/julien-truffaut/Monocle

Encapsulating state

Mostly every application needs to thread some kind of state, and in functional Scala, we have great tools to manage state properly. Whether we use `MonadState`, `StateT`, `MVar`, or `Ref`, we can write good software by following some design guidelines.

One of the best approaches to managing state is to encapsulate state in the interpreter and only expose an abstract interface with the functionality the user needs.

> Tips
>
> `Our interface should know nothing about state`

By doing so, we control exactly how the users interact with state. Conversely, if we use something like `MonadState[F, AppState]` or `Ref[F, AppState]` directly, functions can potentially access and modify the entire state of the application at any given time (unless used together with *classy lenses*, which are a bit more advanced and less obvious than using plain old interfaces).

In-memory counter

Let's say we need to keep a counter in memory that needs to be accessed and modified by other components. Here is what our interface could look like:

```
trait Counter[F[_]] {
  def incr: F[Unit]
  def get: F[Int]
}
```

It has a higher-kinded type `F[_]`, representing an abstract effect, which most of the time ends up being `IO`, and maybe some other effect type for testing.

Next, we need to define an interpreter, in this case using a `Ref` - we will talk more about it in the next section.

```
import cats.effect.concurrent.Ref

class LiveCounter[F[_]] private (
    ref: Ref[F, Int]
) extends Counter[F] {
  def incr: F[Unit] = ref.update(_ + 1)
  def get: F[Int] = ref.get
}
```

Notice how we made the interpreter's constructor *private*. This has a fundamental reason: *state shall not leak*. If we were to make this constructor public, our `Ref` could be used in places where we have no control.

A best practice is to make it private and provide a smart constructor.

```
import cats.effect.Sync
import cats.implicits._

object LiveCounter {
  def make[F[_]: Sync]: F[Counter[F]] =
    Ref.of[F, Int](0).map(new LiveCounter(_))
}
```

It is also a good practice to return our new interface wrapped in `F` since *its creation is effectful* (allocates a mutable reference) and because the counter itself is also *mutable state*. We will later expand on this topic.

We can avoid creating the `LiveCounter` class altogether and define it as an anonymous class in the smart constructor. In fairness, this is most of the time preferred since it reduces the amount of boilerplate.

```
object LiveCounter {
  def make[F[_]: Sync]: F[Counter[F]] =
    Ref.of[F, Int](0).map { ref =>
      new Counter[F] {
        def incr: F[Unit] = ref.update(_ + 1)
        def get: F[Int] = ref.get
      }
    }
}
```

The only issue with this approach is that we can neither define local variables (unless `private`) inside the implementation nor override `def`s as `val`s - the Scala compiler won't allow it. Because of this limitation, we will be defining all of our interpreters using classes. You can always come back to it and simplify it, if possible.

Other programs will interact with this counter via its interface. For example:

```
def program(counter: Counter[IO]): IO[Unit] =
  counter.incr *> otherStuff
```

In the next chapter, we will discuss whether it is best to pass the dependency implicitly or explicitly.

Sequential vs concurrent state

In the previous section, we have seen how our `Counter` keeps state using a `Ref`, but we haven't discussed whether that is a good idea or not.

In a few words, it all boils down to whether we need sequential or concurrent state.

State Monad

If we have a program where state could be sequential, it would be safe to use the `State` monad whose `run` function has roughly the following signature:

```
S => (S, A)
```

The first `S` represents an initial state. The tuple in the result contains both the new state and the result of the state transition (if any). Because of the arrow, the `State` monad is inherently sequential (there is no way to run two `State` actions in parallel and have both changes applied to the initial state).

The following snippet showcases the State monad.

```
val nextInt: State[Int, Int] =
  State(s => (s + 1, s * 2))

def seq: State[Int, Int] =
  for {
    n1 <- nextInt
    n2 <- nextInt
    n3 <- nextInt
  } yield n1 + n2 + n3
```

State is threaded sequentially after each `flatMap` call, which returns the new state that is used to run the following instruction and so on. Certainly, this makes it impossible to work in a concurrent setup.

Atomic Ref

In the case of our `Counter`, we have an interface that might be invoked from many places at the same time, so it is particularly safe to assume we want a concurrency-safe implementation. Here is where `Ref` shines and where the `State` monad wouldn't work.

`Ref` is a purely functional model of a concurrent mutable reference, provided by Cats Effect. Its atomic `update` and `modify` functions allow compositionality and concurrency safety that would otherwise be hard to get right. Internally, it uses a *compare-and-set* loop (or simply CAS loop), but that is something we don't need to worry about.

Shared state

To understand shared state, we need to talk about *regions of sharing*. These regions are denoted by a simple `flatMap` call. Let's look at a simple example presented below to understand what this means.

Regions of sharing

Say that we need two different programs to concurrently acquire a permit and perform some expensive task. We will use a `Semaphore` (another concurrent data structure provided by Cats Effect) of one permit.

```
import cats.effect._
import cats.effect.concurrent.Semaphore
import cats.effect.implicits._
import cats.implicits._
import scala.concurrent.duration._

object SharedState extends IOApp {

  def someExpensiveTask: IO[Unit] =
    IO.sleep(1.second) >>
      putStrLn("expensive task") >>
      someExpensiveTask

  def p1(sem: Semaphore[IO]): IO[Unit] =
    sem.withPermit(someExpensiveTask) >> p1(sem)

  def p2(sem: Semaphore[IO]): IO[Unit] =
    sem.withPermit(someExpensiveTask) >> p2(sem)

  def run(args: List[String]): IO[ExitCode] =
    Semaphore[IO](1).flatMap { sem =>
      p1(sem).start.void *>
        p2(sem).start.void
    } *> IO.never.as(ExitCode.Success)

}
```

Notice how both programs use the same `Semaphore` to control the execution of the expensive tasks. The `Semaphore` is created in the `run` function, by calling its `apply` function with an argument `1`, indicating the number of permits, and then calling `flatMap`

to share it with both `p1` and `p2`. The enclosing `flatMap` block is what denotes our region of sharing. We are in control of how we share such data structure within this block.

This is one of the main reasons why all the concurrent data structures are wrapped in `F` when we create a new one. `Ref`, `Deferred`, `Semaphore`, and even an HTTP `Client` (when using `http4s`).

Leaky state

To illustrate this better, let's look at what this program would look like if our shared state, the `Semaphore`, wasn't wrapped in `IO` (or any other abstract effect).

```
object LeakySharedState extends IOApp {

  // global access
  val sem: Semaphore[IO] =
    Semaphore[IO](1).unsafeRunSync()

  def someExpensiveTask: IO[Unit] =
    IO.sleep(1.second) »
      putStrLn("expensive task") »
      someExpensiveTask

  new LaunchMissiles(sem).run // Unit

  val p1: IO[Unit] =
    sem.withPermit(someExpensiveTask) » p1

  val p2: IO[Unit] =
    sem.withPermit(someExpensiveTask) » p2

  def run(args: List[String]): IO[ExitCode] =
    p1.start.void *> p2.start.void *>
      IO.never.as(ExitCode.Success)

}
```

We have now lost our `flatMap`-denoted region of sharing, and we no longer control where our data structure is being shared. We don't know what `LaunchMissiles` does internally. Perhaps, it acquires the single permit and never releases it, which would block our `p1` and `p2` programs. This is just a tiny example, imagine how difficult it would be to track similar issues in a large application.

Anti-patterns

Seq: a base trait for sequences

Strong claim

Thou shalt not use Seq in your interface

`Seq` is a generic representation of collection-like data structures, defined in the standard library. It is so generic that `List`, `Vector`, and `Stream` share it as a parent interface. This is problematic, since these types are completely different.

To illustrate the problem, let's see the following example:

```
trait Items[F[_]] {
  def getAll: F[Seq[Item]]
}
```

Users of this interface might use it to calculate the total price of all the items.

```
class Program[F[_]](items: Items[F[_]]) {

  def calcTotalPrice: F[BigDecimal] =
    items.getAll.map { seq =>
      seq.toList
        .map(_.price)
        .foldLeft(0)((acc, p) => acc + p)
    }

}
```

How do we know it is safe to call `toList`? What if the `Items` interpreter uses a `Stream` (or `LazyList` since Scala 2.13.0) representing possibly infinite items? It would still be compatible with our interface, yet, it will have different semantics.

To be safe, prefer to use a more specific datatype such as `List`, `Vector`, `Chain`, or `fs2.Stream`, depending on your specific goal and desired performance characteristics.

About monad transformers

Strong claim

Thou shalt not use Monad Transformers in your interface

It is completely fine to use monad transformers in local functions, more likely in the interpreters, but please leave them out of the interface.

For example, using `OptionT` in our API signature is *generally undesirable*.

```
trait Users[F[_]] {
  def findUser(id: UUID): OptionT[F, User]
}
```

We should never expose it in our interface when we already have an abstract effect `F`. So here is how we should do it instead:

```
trait Users[F[_]] {
  def findUser(id: UUID): F[Option[User]]
}
```

This is a common API design, sometimes taken for granted. Committing to a specific Monad Transformer kills compositionality for the API users.

Let's say we are operating in terms of an abstract `F` and suddenly we need to use a function that returns `OptionT[F, User]` and another that returns `EitherT[F, Error, Customer]`. We would need to call `value` in both cases to get back to our abstract effect `F`, an unnecessary wrapping.

Alternatively, let typeclass constraints dictate what `F` is capable of in your programs.

Error handling

There are some known and widely accepted conventions for error handling, but there is no standard. So let me be biased here, and recommend what it has worked well for me.

MonadError and ApplicativeError

We normally work in the context of some parametric effect `F[_]`. Particularly, when using Cats Effect, we can rely on `MonadError` / `ApplicativeError` and its functions `attempt`, `handleErrorWith`, `rethrow`, among others, to deal with errors, since the `IO` monad implements `MonadError[F, Throwable]`.

Say we have a `Categories` algebra that lets us find all the categories available.

```
trait Categories[F[_]] {
  def findAll: F[List[Category]]
}
```

We also have an ADT (Algebraic Data Type) of our error hierarchy.

```
sealed trait BusinessError extends NoStackTrace
case object RandomError extends BusinessError
```

And the following interpreter (details about `Random` are not relevant):

```
class LiveCategories[
    F[_]: MonadError[*[_], Throwable]: Random
] extends Categories[F] {

  def findAll: F[List[Category]] =
    Random[F].bool.ifM(
      List.empty[Category].pure[F],
      RandomError.raiseError[F, List[Category]]
    )

}
```

Its interface doesn't say anything about `RandomError` so you might wonder whether it would be better to be specific about the error type and change its signature to something like this:

```
trait Categories[F[_]] {
  def maybeFindAll: F[Either[RandomError, List[Category]]]
}
```

The answer is *it depends*. My recommendation is to only do it when it is really necessary. At all times, the question you need to ask yourself is **What am I going to do with the error information?**

Most of the time you only need to deal with the successful case and let it fail in case of error. However, there are valid cases for explicit error handling and one of my favorites is to handle business errors at the HTTP layer to return different HTTP response codes, in which case we don't really need to use `Either`.

We can handle the errors that are relevant to the business and forget about the rest. The problem is that we are dealing with an interface that doesn't say anything about what kind of errors might arise, so it is a compromise we need to be aware of.

Tips

`Code the happy path and watch the frameworks do the right thing`

This means we only need to worry about the successful cases and the business errors. In other cases, the higher-level frameworks will do the correct thing. In other words, if you're using Http4s and forget to handle a `RandomError`, you will get a `500 Internal Server Error` as a response. Your application will not blow up because of it.

Either Monad

In some other cases, it is perfectly valid to use `F[Either[E, A]]`. Say we have a `Program` using `Categories[F]` and depending on whether it gets `BusinessError` or a `List[Category]` the business logic changes. This is a fair case and you can see how we can, at the same time, eliminate `F[Either[E, A]]` and go back to `F[A]` in the example below.

```
class Program[F[_]: Functor](
    categories: Categories[F]
) {

  def findAll: F[List[Category]] =
    category.maybeFindAll.map {
      case Right(c)          => c
      case Left(RandomError) => List.empty[Category]
    }

}
```

Notice that we could have done the same without having the error type in the interface by using `ApplicativeError`.

```
type ApThrow[F[_]] = ApplicativeError[F, Throwable]

class SameProgram[F[_]: ApThrow](
    categories: Categories[F]
) {

  def findAll: F[List[Category]] =
    category.findAll.handleError {
      case RandomError => List.empty[Category]
    }

}
```

But what happens if we were to add another error case to the `BusinessError` ADT? The compiler will not warn us about it; on the other hand, we would get a compiler error if our interface had the error type information.

```
[error] It would fail on the following input: Left(AnotherError)
[error]     category.maybeFindAll.map {
[error]                               ^
[error] one error found
```

This is one of the benefits of using `F[Either[E, A]]`, but I would argue the cons outweigh the benefits. Composing functions of this type signature is cumbersome since we need to lift nearly every operation into the `EitherT` monad transformer, and most of the time, the compiler can not infer the types correctly, so we end up needing to annotate each part.

Also, when `E <: Throwable`, we are better off using `F[A]` and relying on `MonadError`, which has better ergonomics at the cost of losing the error type.

This is seen as a trade-off. The important take away is to be aware of the different ways of doing error handling and make a conscious decision.

Classy prisms

Another more advanced error handling technique we have as an option is using *classy prisms*, or more generically called *classy optics*, using the Meow MTL[9] library. It gives us back typed errors and exhausting pattern matching without polluting our interfaces with `F[Either[E, A]]` nor using monad transformers.

It all starts with defining the hierarchy of errors as an ADT.

[9] https://github.com/oleg-py/meow-mtl

```
sealed trait UserError extends NoStackTrace
final case class UserAlreadyExists(username: String) extends UserError
final case class UserNotFound(username: String) extends UserError
final case class InvalidUserAge(age: Int) extends UserError
```

We then need to define a generic error handler for any error that is a subtype of `Throwable` to be compatible with Cats Effect IO. In this case, we are going to define an `HttpErrorHandler` for `http4s`, but it could be other kinds of error handler.

Here is our generic interface for handling business errors in our HTTP routes:

```
trait HttpErrorHandler[F[_], E <: Throwable] {
  def handle(routes: HttpRoutes[F]): HttpRoutes[F]
}
```

For convenience, we also define a generic `RoutesHttpErrorHandler` that wraps the same logic we need in all our Http error handlers, to avoid repeating ourselves.

```
abstract class RoutesHttpErrorHandler[F[_], E <: Throwable]
    extends HttpErrorHandler[F, E]
    with Http4sDsl[F] {

  def A: ApplicativeError[F, E]
  def handler: E => F[Response[F]]
  def handle(routes: HttpRoutes[F]): HttpRoutes[F] =
    Kleisli { req =>
      OptionT {
        A.handleErrorWith(
          routes.run(req).value
        )(e =>
          A.map(handler(e))(Option(_))
        )
      }
    }
}
```

Notice the requirement of an `ApplicativeError[F, E]` for our specific error type, and a generic `handle` function that takes a `HttpRoutes[F]` and returns another `HttpRoutes[F]`, after handling the possible errors of type `E`.

Finally, we define a specific error handler for `UserError`, making use of our generic handler.

```
object UserHttpErrorHandler {
  def apply[F[_]: MonadError[*[_], UserError]]
  : HttpErrorHandler[F, UserError] =
```

```
    new RoutesHttpErrorHandler[F, UserError] {
      val A: ApplicativeError[F, UserError] = implicitly

      val handler: UserError => F[Response[F]] = {
        case InvalidUserAge(age) =>
          BadRequest(s"Invalid age $age".asJson)
        case UserAlreadyExists(username) =>
          Conflict(username.asJson)
        case UserNotFound(username) =>
          NotFound(username.asJson)
      }
    }
}
```

All we need to do is to require a `MonadError[F, UserError]` and define the `handler` function. With all this machinery in place, we can proceed to use our error handler in our HTTP routes. For example:

```
val users: Users[F] = ???

val httpRoutes: HttpRoutes[F] =
  HttpRoutes.of {
    case GET -> Root =>
      Ok(users.findAll)
  }

def routes(
    ev: HttpErrorHandler[F, UserError]
): HttpRoutes[F] =
  ev.handle(httpRoutes)
```

The avid reader might have noticed that there is no relationship between `Users[F]` and `HttpErrorHandler[F, UserError]`. If the `Users[F]` interpreter raises other kinds of errors, our handler will not see them, but this is intentional. We are only interested in `UserError`s. Any other errors should be seen as unexpected failures that should be handled correctly by our HTTP framework.

In any case, we could continue making changes in our quest for typed errors and make our interface and error handler have a relationship as I have demonstrated in these[10] blog post series[11]; but for simplicity and ergonomics, this is where we are going to settle.

Last but not least, we need to see why we need classy prisms.

[10] https://typelevel.org/blog/2018/08/25/http4s-error-handling-mtl.html
[11] https://typelevel.org/blog/2018/11/28/http4s-error-handling-mtl-2.html

```
class MyRoutes[F[_]: MonadError[*[_], MyError]] { ... }

def foo[F[_]: Sync] = new MyRoutes[F] {}
```

It would be great if we could do this but unfortunately, the Scala compiler yells at us "ambiguous implicit values". If we have a look at the hierarchy of typeclasses of Cats Effect, we will see that `Sync` implies `MonadError[F, Throwable]`, the reason why we can't have another `MonadError` instance of a different error type in scope.

Luckily, if we have `MonadError[F, Throwable]` in scope, the Meow MTL library can derive any other `MonadError` instances for us, as long as our error type is a subtype of `Throwable`. All it takes is a single import.

```
import oleg.meow.mtl.hierarchy._
```

Thanks to this technique, we can have our error types back on-demand, and with it, we put an end to this section, where we have only seen a subset of the features this library offers.

Don't worry if you feel you didn't grasp it; this is arguably the most advanced topic of the book. In the last chapter, we are going to apply this technique to our application as well as expand on classy optics to see what else is possible.

Chapter 3: Tagless final encoding

The tagless final encoding (also called *finally tagless*) is a method of embedding domain-specific languages (DSLs) in a typed functional host language such as Haskell, OCaml, Scala, or Coq. It is an alternative to the *initial encoding* promoted by *Free Monads*.

This technique is well described in Oleg Kiselyov's papers[1]. However, in Scala, it has diverged into a more ergonomic encoding that doesn't necessarily follow the original semantics.

In this chapter, we will dive deep into the practical meaning of this technique and also explore best practices.

[1] http://okmij.org/ftp/tagless-final/index.html

Algebras

An algebra describes a new language (DSL) within a host language, in this case, Scala.

This may surprise some of you, but tagless final encoded algebras are not a new concept in this book. We have already seen them in Chapter 2 without having to mention their other name; we called them instead, interfaces with a higher-kinded type.

Remember our `Counter`? Let's recap.

```
trait Counter[F[_]] {
  def incr: F[Unit]
  def get: F[Int]
}
```

This is a tagless final encoded algebra or *tagless algebra* for short: a simple interface that abstracts over the effect type using a type constructor `F[_]`. Do not confuse algebras with typeclasses, which in Scala, happen to share the same encoding.

The difference is that typeclasses should have coherent instances, whereas tagless algebras could have many implementations, or more commonly called *interpreters*. This makes them a perfect fit for encoding business concepts.

For example, an algebra responsible for managing items could be encoded as follows:

```
trait Items[F[_]] {
  def getAll: F[List[Item]]
  def add(item: Item): F[Unit]
}
```

Nothing new, right? A tagless final encoded algebra is merely an interface that abstracts over the effect type. Notice that neither the algebra nor its functions have any typeclass constraint.

> Tips
>
> `Tagless algebras should not have typeclass constraints`

If you find yourself needing to add a typeclass constraint, such as `Monad`, to your algebra, what you truly need is a *program*.

Naming conventions

Due to my preferences, we named our previous algebra `Items`, albeit not being a standard. Out in the wild, you will find people using other names such as `ItemService`, `ItemAlgebra`, or `ItemAlg`, to name a few. You are free to choose the name you like the most; however, it is important to be consistent with your choices across your entire application.

For interpreters, I like to prefix the word `Live` to the algebra's name. For example, our production interpreter for `Items` would be `LiveItems`, and so on. You will see this naming convention used consistently throughout the chapters.

Interpreters

We would normally have two interpreters per algebra: one for testing and one for doing real things. For instance, we could have two different implementations of our `Counter`.

A default interpreter using `Redis`:

```
@newtype case class RedisKey(value: String)

class LiveCounter[F[_]: Functor](
    key: RedisKey,
    cmd: RedisCommands[F, String, Int]
) extends Counter[F] {

  def incr: F[Unit] =
    cmd.incr(key).void

  def get: F[Int] =
    cmd.get(key).map(_.getOrElse(0))

}
```

And a test interpreter using an in-memory data structure:

```
class TestCounter[F[_]] (
    ref: Ref[F, Int]
) extends Counter[F] {
  def incr: F[Unit] = ref.update(_ + 1)
  def get: F[Int]   = ref.get
}
```

Interpreters help us encapsulate state and allow separation of concerns: the interface knows nothing about the implementation details. Moreover, interpreters can be written either using a concrete datatype such as `IO` or going polymorphic all the way, as we did in this case.

Building interpreters

Our `LiveCounter` needs a `RedisCommands`, which lets us operate with a Redis instance. This is the interpreter we would use to run our application in production. But remember that other programs will only interact with its algebra, `Counter`, and they will know nothing about what kind of data storage we are using.

If Redis is only used by our `Counter` interpreter, then no other component in our application should know about it. Our Redis connection should be seen as state that must be encapsulated. Does that sound familiar?

As we have seen in Chapter 2, we can make our constructor private and provide a smart constructor that encapsulates the state. In this case, creating a Redis connection.

```
class LiveCounter[F[_]: Functor] private (
    key: RedisKey,
    cmd: RedisCommands[F, String, Int]
) extends Counter[F] { ... }

object LiveCounter {
  def make[F[_]: Sync]: Resource[F, Counter[F]] =
    cmdApi.map { cmd =>
      new LiveCounter(RedisKey("myKey"), cmd)
    }

  private val cmdApi
    : Resource[IO, RedisCommands[IO, String, Int]] = ???
}
```

Notice how instead of `F[Counter[F]]`, we are returning `Resource[F, Counter[F]]`. Since our `LiveCounter` requires a Redis connection, which is treated as a resource, then we also need to make our counter's smart constructor a resource itself; this is a common practice.

At usage site, this will trivially become something along these lines:

```
LiveCounter.make[IO].use { counter =>
  p1(counter) *> p2(counter) *> sthElse
}
```

Our Redis connection will only live within the `use` block. The `Resource` datatype guarantees the clean up of the resource (closing Redis connection) when the program has terminated, as well as in the presence of failures or interruption.

In this example, we used the Redis4Cats[2] library, but the same principle applies when utilizing other libraries.

[2] https://redis4cats.profunktor.dev/

Programs

Tagless final is all about algebras and interpreters. Yet, something is missing when it comes to writing applications: we need to use these algebras to describe business logic, and this logic belongs in what I like to call programs.

> Notes
>
> `Programs can make use of algebras and other programs`

Although it is not an official name - and it is not mentioned in the original tagless final paper - it is how we will be referring to such interfaces in this book.

Say we need to increase a counter every time there is a new item added. We could encode it as follows:

```
class ItemsProgram[F[_]: Apply](
    counter: Counter[F],
    items: Items[F]
) {

  def addItem(item: Item): F[Unit] =
    items.add(item) *>
      counter.incr

}
```

Observe the characteristics of this program, it is pure business logic, and it holds no state at all, which in any case, must be encapsulated in the interpreters. Notice the typeclass constraints as well; it is a good practice to have them in programs instead of tagless algebras.

Here, the program doesn't need to consider concurrent or parallel effects, but that should be fine too. Parallelism can be conveyed using the `Parallel` typeclass and concurrency using the `Concurrent` typeclass.

Unfortunately, the latter implies `Async` and `Sync`, which allow encapsulating arbitrary side-effects. This situation will improve in Cats Effect 3, though.

Moreover, we can discuss typeclass constraints. In this case, we only need `Apply` to use `*>` (alias for `productR`). However, it would also work with `Applicative` or `Monad`. The rule of thumb is to limit ourselves to adopt the least powerful typeclass that gets the job done.

It is worth mentioning that `Apply` itself doesn't specify the semantics of composition solely with this constraint, `*>` might combine its arguments sequentially or parallelly,

depending on the underlying typeclass instance. To ensure our composition is sequential, we could use `FlatMap` instead of `Apply`.

> **Tips**
>
> When adding a typeclass constraint, remember about the principle of least power

Other kinds of programs might be directly encoded as functions.

```
def program[F[_]: Console: Monad]: F[Unit] =
  for {
    _ <- Console[F].putStrLn("Enter your name: ")
    n <- Console[F].readLn
    _ <- Console[F].putStrLn(s"Hello $n!")
  } yield ()
```

Furthermore, we could have programs composed of other programs.

```
class BiggerProgram[F[_]: Console: Monad](
    items: ItemsProgram[F],
    counter: Counter[F]
) {

  def logic(item: Item): F[Unit] =
    for {
      _ <- items.addItem(item)
      c <- counter.get
      _ <- Console[F].putStrLn(s"Number of items: $c")
    } yield ()

}
```

Whether we encode programs in one way or another, they should describe pure business logic and nothing else.

The question is: *what is pure business logic*? We could try and define a set of rules to abide by. It is allowed to:

- Combine pure computations in terms of tagless algebras and programs.
  - Only doing what our effect constraints allows us to do.
- Perform logging (or console stuff) only via a tagless algebra.
  - In Chapter 7, we will see how to ignore logging or console stuff in tests, which are most of the time irrelevant in such context.

You can use this as a reference. However, the answer should come up as a collective agreement within your team.

Implicit vs explicit parameters

So far, we have talked about algebras, interpreters, and programs. Yet, little did we talk about implicit parameters and when we should be using them.

In Scala, tagless final has been misused quite considerably. Have you ever seen anything like this?

```
def program[
  F[_]: Cache: Console: Users: Monad
      : Parallel: Items: EventsManager
      : HttpClient: KafkaClient: EventsPublisher
]: F[Unit] = ???
```

This is something I would consider an **anti-pattern**.

Implicits are a way to encode coherent typeclass instances. However, there are other practical usages for this mechanism, and here is where I am going to be biased and recommend in what other cases I consider fine to use them.

Let's start by saying that any business logic related algebra should not, by any means, be encoded as an implicit parameter.

> Tips
>
> `Business logic algebras should always be passed explicitly`

So let's go ahead and modify our first example.

```
def program[
  F[_]: Cache: Console: Monad: Parallel
      : EventsManager: EventsPublisher
      : KafkaClient: HttpClient
](
   users: Users[F],
   items: Items[F]
): F[Unit] = ???
```

We have improved slightly, but it certainly isn't ideal. Next, we can assume that all the `Events` algebras are also *business-related* and pass them explicitly instead.

```
def program[
  F[_]: Cache: Console: Monad: Parallel
      : KafkaClient: HttpClient
](
   users: Users[F],
   items: Items[F],
```

```
    eventsManager: EventsManager[F],
    eventsPublisher: EventsPublisher[F]
): F[Unit] = ???
```

We are left with the following two algebras: `KafkaClient` and `HttpClient`. Frequently, such clients have a lifecycle, best managed as resources; hence, they need to be passed explicitly since creating a resource is an effectful action. Last but not least, we could arguably do the same for `Cache`, which might be backed by `Redis`, for example.

```
def program[
  F[_]: Console: Monad: Parallel
](
    users: Users[F],
    items: Items[F],
    eventsManager: EventsManager[F],
    eventsPublisher: EventsPublisher[F],
    cache: Cache[F],
    kafkaClient: KafkaClient[F],
    httpClient: HttpClient[F]
): F[Unit] = ???
```

Much better. But now we seem to face another problem: we have too many dependencies, which makes dealing with them a cumbersome task.

Though, I would argue that all we need is a better organization. We usually encounter these kinds of programs at the top level of our application, thus explaining the number of dependencies.

In the next section, we will learn how modules help us dealing with this issue.

Achieving modularity

Grouping tagless algebras that share some commonality in a higher-level interface is one simple way to achieve modularity and avoid ending up with twenty arguments per function, as previously seen.

These higher-level interfaces are what I like to call *modules*, and the Scala language is pretty good at this.

We can now try and identify the common things among our algebras and put them together in different modules, which we will be representing using traits.

```
package modules

trait Algebras[F[_]] {
  def users: Users[F]
```

```
  def items: Items[F]
}

trait Events[F[_]] {
  def manager: EventsManager[F]
  def publisher: EventsPublisher[F]
}

trait Clients[F[_]] {
  def kafka: KafkaClient[F]
  def http: HttpClient[F]
}

trait Database[F[_]] {
  def cache: Cache[F]
}
```

Does it make sense? Having our dependencies organized in this way makes our codebase much easier to understand and maintain.

To build our modules, we can use a smart constructor in the interface's companion object. For example, this is what our `Clients` implementation could look like:

```
object Clients {

  def make[F[_]: Concurrent]: Resource[F, Clients[F]] =
    (KafkaClient.make[F], HttpClient.make[F]).mapN {
      case (k, h) =>
        new Clients[F] {
          def kafka = k
          def http  = h
        }
    }

}
```

In this hypothetical case, we are requiring an instance of `Concurrent[F]` since it might be commonly required by either `KafkaClient` or `HttpClient`, but it could be different. Always remember the principle of least power.

Moving forward, let's see the final version of our program.

```
def program[
  F[_]: Console: Monad: Parallel
](
   algebras: Algebras[F],
```

```
    events: Events[F],
    cache: Cache[F],
    clients: Clients[F]
): F[Unit] = ???
```

Neat! With a little bit of organization, we have arrived at a simple solution.

Implicit convenience

There are some examples of implementations that are passed as implicits and that are not typeclasses. In Cats Effect, for example, the types `ContextShift`, `Clock`, and `Timer` fit this usage pattern.

Why are they used implicitly if they are not typeclasses? It is merely for convenience since instances for these datatypes are normally given by `IOApp` as the "environment".

They are seen as common effects, and this vision allows us to have different instances for testing purposes, which would not be possible if using typeclasses as we would be creating *orphan instances*.

> **Notes**
>
> `Common effects do not hold business logic`

In such cases, we can say it is fine to thread instances implicitly. It is convenient, and it doesn't break anything, so I would personally endorse this usage.

In the example above, we are left with this implicit encoding:

```
def program[F[_]: Console: Monad: Parallel]
```

From what we can gather, the only valid and lawful typeclasses are `Monad` and `Parallel`. What about `Console`? Although it is not a typeclass, it is more convenient to pass it implicitly since it fits the description of a common effect that would rarely need more than a single instance. Nevertheless, if we need to, we can create another instance for testing purposes as well.

In the end, it is all about common sense, consistency, and good practices. The language is flexible enough to allow typeclasses and convenient interfaces to be encoded in the same way. Let's just remember to adhere to our practical rules and use the language as it was intended to be used.

Chapter 4: Business logic

In the previous chapters, we have distilled the business requirements into technical specifications and have identified the possible HTTP endpoints our application should expose. We have also explored some functional design patterns and unleashed the power of abstraction with tagless final encoding.

Now it is time to get to work, as we apply these techniques to our business domain. A common way to get a featured design started is to decompose business requirements into small, self-contained algebras.

Identifying algebras

Summarizing, this list contains the secured, admin, and open HTTP endpoints:

- `GET` **/brands**
- `POST` **/brands**
- `GET` **/categories**
- `POST` **/categories**
- `GET` **/items**
- `GET` **/items?brand=gibson**
- `POST` **/items**
- `PUT` **/items**
- `GET` **/cart**
- `POST` **/cart**
- `PUT` **/cart**
- `DELETE` **/cart/{itemId}**
- `GET` **/orders**
- `GET` **/orders/{orderId}**
- `POST` **/checkout**
- `POST` **/auth/users**
- `POST` **/auth/login**
- `POST` **/auth/logout**

Our mission is to identify common functionality between these endpoints and create a tagless algebra. Let's get started with the `brands` group of endpoints.

Brands

Our `Brand` domain consists of two endpoints: a `GET` to retrieve the list of brands and a `POST` to create new brands. The POST endpoint should only be used by administrators. However, we don't consider permission details at the algebra level. So let's condense this functionality into a single algebra.

```
trait Brands[F[_]] {
  def findAll: F[List[Brand]]
  def create(name: BrandName): F[Unit]
}
```

As we have identified in Chapter 1, `Brand` is our datatype representing the business domain. In order to translate this to code, we will be representing the model using case classes and the Newtype library.

```
@newtype case class BrandId(value: UUID)
@newtype case class BrandName(value: String)

case class Brand(uuid: BrandId, name: BrandName)
```

That is all we need, a clear algebra that programs can use to implement some functionality. At this point, we don't particularly care about implementation details.

Categories

Next is the `Categories` domain, which is very similar to `Brands`.

```
trait Categories[F[_]] {
  def findAll: F[List[Category]]
  def create(name: CategoryName): F[Unit]
}
```

Once again, we model our input and output; our `Category` datatype is defined as follows:

```
@newtype case class CategoryId(value: UUID)
@newtype case class CategoryName(value: String)

case class Category(uuid: CategoryId, name: CategoryName)
```

Items

The next domain on the list is `Items`, which has two `GET` endpoints: one to retrieve a list of all the items, and another to retrieve items filtering by brand. It also has a `POST` endpoint to create an item and a `PUT` endpoint to update an item. Both are administrative tasks, but as we mentioned before, it is not a concern at this level.

```
trait Items[F[_]] {
  def findAll: F[List[Item]]
  def findBy(brand: BrandName): F[List[Item]]
  def findById(itemId: ItemId): F[Option[Item]]
  def create(item: CreateItem): F[Unit]
  def update(item: UpdateItem): F[Unit]
}
```

The `Item` datatype is a bit more interesting than our previous domain datatypes on closer inspection.

```
import squants.market.Money

@newtype case class ItemId(value: UUID)
@newtype case class ItemName(value: String)
@newtype case class ItemDescription(value: String)

case class Item(
    uuid: ItemId,
    name: ItemName,
    description: ItemDescription,
    price: Money,
    brand: Brand,
    category: Category
)

case class CreateItem(
    name: ItemName,
    description: ItemDescription,
    price: Money,
    brandId: BrandId,
    categoryId: CategoryId
)

case class UpdateItem(
    id: ItemId,
    price: Money
)
```

Our price field is going to be represented using the `Money` type provided by Squants, which supports many different currencies. In the future, we may need to support other markets; this can be easily achieved by converting between currencies, e.g. using the exchange rate of the day.

Shopping Cart

Next is our `Cart` domain. It has a `GET` endpoint to retrieve the shopping cart of the current user, a `POST` endpoint to add items to the cart, a `PUT` endpoint to edit the quantity of any item, and a `DELETE` endpoint to remove an item from the cart. The following algebra encodes this functionality in the respective order.

```
trait ShoppingCart[F[_]] {
  def add(
    userId: UserId,
```

```
    itemId: ItemId,
    quantity: Quantity
  ): F[Unit]
  def delete(userId: UserId): F[Unit]
  def get(userId: UserId): F[CartTotal]
  def removeItem(userId: UserId, itemId: ItemId): F[Unit]
  def update(userId: UserId, cart: Cart): F[Unit]
}
```

Here we have some new datatypes, including a few we haven't classified in Chapter 1:

```
@newtype case class Quantity(value: Int)
@newtype case class Cart(items: Map[ItemId, Quantity])
@newtype case class CartId(value: UUID)

case class CartItem(item: Item, quantity: Quantity)
case class CartTotal(items: List[CartItem], total: Money)
```

Our `Cart` is a simple key-value store of `ItemId`s and `Quantity`s, respectively, so we can easily avoid duplicates and tell how many specific items there are in the cart. Furthermore, `CartItem` is a simple wrapper of `Item` and `Quantity`, so we can provide more details about the item.

Orders

Once we process a payment, we need to persist the order; we also want to be able to query past orders. Here is our algebra:

```
trait Orders[F[_]] {
  def get(
    userId: UserId,
    orderId: OrderId
  ): F[Option[Order]]

  def findBy(userId: UserId): F[List[Order]]

  def create(
    userId: UserId,
    paymentId: PaymentId,
    items: List[CartItem],
    total: Money
  ): F[OrderId]
}
```

We have some new entities here.

```
@newtype case class OrderId(uuid: UUID)
@newtype case class PaymentId(uuid: UUID)

case class Order(
  id: OrderId,
  pid: PaymentId,
  items: Map[ItemId, Quantity],
  total: Money
)
```

This is the information we will be persisting in PostgreSQL. The persisted object contains the `PaymentId` returned by the external payment system and the total amount specified in US Dollars.

Users

Our system should be able to store basic information about users, such as usernames and encrypted passwords.

```
trait Users[F[_]] {
  def find(
    username: UserName,
    password: Password
  ): F[Option[User]]

  def create(
    username: UserName,
    password: Password
  ): F[UserId]
}
```

This algebra will be used by our next algebra `Auth[F]`.

Authentication

There are also the authentication endpoints. We are going to use JSON Web Tokens (JWT) as the authentication method, as we will further expand in Chapter 5. Until we get there, we can sketch something out with what we currently have and make some modifications later on, if necessary.

> **Warning**
>
> Interface subject to change in future iterations

```
trait Auth[F[_]] {
  def findUser(token: JwtToken): F[Option[User]]
  def newUser(username: UserName, password: Password): F[JwtToken]
  def login(username: UserName, password: Password): F[JwtToken]
  def logout(token: JwtToken, username: UserName): F[Unit]
}
```

Remember that we have guest users, common users, and admin users. The former is the only one that doesn't require authentication, so we don't need to represent it in our domain model. Next, we have a few common types.

```
@newtype case class UserId(value: UUID)
@newtype case class UserName(value: String)
@newtype case class Password(value: String)
@newtype case class JwtToken(value: String)

case class User(id: UserId, name: UserName)
```

Payments

Finally, let's not forget about our external payments API. A good practice is to also define a tagless algebra for remote clients.

```
trait PaymentClient[F[_]] {
  def process(payment: Payment): F[PaymentId]
}
```

The `Payment` datatype is defined as follows:

```
case class Payment(
    id: UserId,
    total: Money,
    card: Card
)
```

This is all we know about the payment system's input. In Chapter 1, we have defined the `Card` datatype, and in the next chapter, we are going to see its full implementation.

Our work defining the algebras for our application is now complete. We skipped `checkout`, as you might have noticed, and you will soon find out why.

Data access and storage

Before we can create the interpreters for our algebras, we should identify what kind of state we need in each of them, which takes us to the next question: What kind of storage are we going to use?

We are going to persist Brands, Categories, Items, Orders, and Users in PostgreSQL. For fast access, we are going to store the Shopping Cart in Redis, as well as the authentication tokens, which we are going to see in Chapter 6.

This is great information, but we are not yet ready to write these interpreters. For now, we can focus on writing the business logic of our application and leave the implementation details for later, once we reach Chapter 6 and see how to deal with both PostgreSQL and Redis in a purely functional fashion.

Defining programs

So far, we have defined a lot of new functionality in our tagless algebras. Some parts of our application are going to make direct use of some of these algebras; other parts will require more than just calling simple functions in our algebras. The principal role of our programs is to describe business logic operations as a kind of a DSL, without needing to know about implementation details.

This is arguably one of the most exciting challenges in this book. Let's dive into it!

Checkout

The following checkout function conveys the idea of the simplest implementation: a sequence of actions denoted as a *for-comprehension* - syntactic sugar for a sequence of `flatMap` calls and a final `map` call. In essence, this function is retrieving the cart for the current user, calling the remote API that processes the payment, and finally persisting a new order.

```
final class CheckoutProgram[F[_]: Monad](
    paymentClient: PaymentClient[F],
    shoppingCart: ShoppingCart[F],
    orders: Orders[F]
) {

  def checkout(userId: UserId, card: Card): F[OrderId] =
    for {
      cart <- shoppingCart.get(userId)
      paymentId <- paymentClient.process(
                     Payment(userId, cart.total, card)
                   )
      orderId <- orders.create(
                   userId, paymentId, cart.items, cart.total
                 )
      _ <- shoppingCart.delete(userId)
    } yield orderId

}
```

It seems we are done here, but if we think about it, this is the *most critical* piece of code in our application! How does this function behave if a failure occurs at any stage? How should we react to the various failure types? The answer strictly depends on the business requirements. However, we should notify them about the suggested alternatives so they can make an informed decision.

As good software engineers, let's dissect the former application and explore how we could mitigate some of the issues.

Deep technical analysis

Let's look at the first line.

```
cart <- shoppingCart.get(userId)
```

What happens if we cannot find the shopping cart of the current user? In this case, the user's cart is either empty, or there is an issue communicating to our database. In any case, there is not much we can do, and we can argue it is not a critical step. So we can let it fail, returning some kind of error message.

Next, we have the most critical part, the payment itself.

```
paymentId <- paymentClient.process(Payment(userId, total, card))
```

It is handled by a third-party HTTP API that we are told is idempotent, to avoid duplicate payments, so this is one less scenario to worry about.

Though, we need to cautiously think about the worst possible scenarios: server crashing, network going down, etc. Let's explore this in detail as we analyze each case.

Payment failure #1 Either there is an error response code, distinct from 409, from the external HTTP API; or something went wrong and our HTTP request didn't complete as we expected (e.g. network issues, request timeouts, etc).

This is the simplest error, in which case we can retry. The most common procedure is to log the error and make the request once again, using a specific retry policy, as we will see soon.

Payment failure #2 The next case scenario involves a duplicate request. Let's say we make an HTTP request and the payment is processed successfully on their end but we fail to get a response (again, due to some network issues). In such a case, we are going to retry a few moments later.

When we retry, we get a specific response code (409) from the remote API, indicating the payment has already been processed. Additionally, we get the Payment ID as the body of the response. This is easy. We are only interested in the Payment Id, so all we need to do is to handle this specific error, extract the Payment Id, and continue with the checkout process.

To follow, we have the creation of the order.

```
orderId <- orders.create(userId, paymentId, items)
```

Order failure #1 We didn't get to see the implementation yet, but we know that the orders are to be persisted in PostgreSQL. Thus, we need to handle possible database or connection failures.

If our database call fails to be processed (e.g. network failure), we can again retry a limited amount of times.

Order failure #2 Let's say that our retry mechanism has completed, and we finally give the user a response. This has taken some time, but let's be honest, nobody likes to wait more than a couple of milliseconds when purchasing goods online; it's not the 1990s anymore.

Since the payment has been processed and the customer has been charged, we can try to revert the payment and return an error. Unfortunately, we are told the remote payment system doesn't support this feature yet, so we would need to solve this issue differently.

The payment is immutable and cannot be reverted from our end. All we can do is deal with this error later. One approach would be to reschedule the order creation to run in the background at some point in the future and, in the meantime, get back to the user saying the payment was successful and that the order should be available soon.

One arbitrary decision would be to run this background action forever until it succeeds. The order needs to be created, **no matter what**. But we need to be realistic and contemplate the possible drawbacks of such a drastic decision, even if they are minimal. The issue that has been affecting our order creation might be unrecoverable, let's say, the database server went on fire.

We can either live with this decision, knowing that our application restarts regularly; or give this background task a limited amount of retries as well, possibly never persisting such order in our local database. In this case, we are going to go with the first option, informing the business of the choices made.

Last but not least, we delete the shopping cart for the current user.

```
_ <- shoppingCart.delete(userId)
```

There is nothing critical in this part, but just in case we should `.attempt` the action (which means changing our `Monad` constraint to `MonadError[F, Throwable]`), to make our program resilient to possible failures. If it fails for some reason, it is a small problem we can deal with later. It should result as follows:

```
_ <- shoppingCart.delete(userId).attempt.void
```

We also add an explicit `void` to discard its result. Although unnecessary, I believe discarding a result in a for-comprehension should be rejected by the compiler. Unfortunately, in this case, it is not.

Retrying effects

Retrying arbitrary effects using Cats Effect is fairly easy. For instance, we could delay the execution of a specific action, and then do it all over again, recursively.

```
def retry[A](fa: F[A]): F[A] =
  Timer[F].sleep(50.milliseconds) » fa
```

We can either build more complex retrying functions in this way, or we can choose a library that does it all for us.

Cats Retry[1] is a great choice, offering different retry policies, powerful combinators, and a nice DSL. Let's see how we can exploit its power.

First, we need to define a common function to log errors for both cases: processing the payment and persisting the order.

```
def logError(
    action: String
)(
    e: Throwable,
    details: RetryDetails
): F[Unit] =
  details match {
    case r: WillDelayAndRetry =>
      Logger[F].error(
        s"Failed on $action. We retried ${r.retriesSoFar} times."
      )
    case g: GivingUp =>
      Logger[F].error(
        s"Giving up on $action after ${g.totalRetries} retries."
      )
  }
```

We also need to have a retry policy. In both cases, we are going to have a maximum of three retries with an exponential back-off of 10 milliseconds between retries.

```
import retry.RetryPolicies._

val retryPolicy =
  limitRetries[F](3) |+| exponentialBackoff[F](10.milliseconds)
```

Easy right? Retry policies have a `Semigroup` instance that makes combining them straightforwardly.

We can now define a helper function that retries payments.

[1] https://github.com/cb372/cats-retry

```
def processPayment(payment: Payment): F[PaymentId] = {
  val action = retryingOnAllErrors[PaymentId](
    policy = retryPolicy,
    onError = logError("Payments")
  )(paymentClient.process(payment))

  action.adaptError {
    case e =>
      PaymentError(
        Option(e.getMessage).getOrElse("Unknown")
      )
  }
}
```

The last part of our function is quite interesting. Using `adaptError`, we are adapting (pun intended) the error given by the payment client (re-thrown after our retry function gives up) into our custom `PaymentError`. We also need to wrap `e.getMessage` in an `Option` because it may be `null`; remember that we are dealing with `java.lang.Throwable` here.

Here is another helper function for creating and persisting orders:

```
def createOrder(
    userId: UserId,
    paymentId: PaymentId,
    items: List[CartItem],
    total: Money
): F[OrderId] = {
  val action = retryingOnAllErrors[OrderId](
    policy = retryPolicy,
    onError = logError("Order")
  )(orders.create(userId, paymentId, items, total))

  def bgAction(fa: F[OrderId]): F[OrderId] =
    fa.adaptError {
        case e => OrderError(e.getMessage)
      }
      .onError {
        case _ =>
          Logger[F].error(
            s"Failed to create order for: ${paymentId}"
          ) *>
            Background[F].schedule(bgAction(fa), 1.hour)
      }
```

```
  bgAction(action)
}
```

Besides our retry mechanism, we have now introduced a new effect `Background`, which lets us schedule tasks to run in the background sometime in the future. Let's have a look at its interface.

```
trait Background[F[_]] {
  def schedule[A](
      fa: F[A],
      duration: FiniteDuration
  ): F[Unit]
}
```

We could have done this directly using `Concurrent` and `Timer`; in fact, this is what our default implementation does, though, there are a few reasons why having a custom interface is a better approach.

- We gain more control by restricting what the final user can do.
- We avoid having `Concurrent` as a constraint, which allows arbitrary side-effects.
- We achieve better testability, as we will see in Chapter 7.

For completeness, here is our default `Background` instance:

```
implicit def concurrentBackground[
    F[_]: Concurrent: Timer
]: Background[F] =
  new Background[F] {

    def schedule[A](
        fa: F[A],
        duration: FiniteDuration
    ): F[Unit] =
      (Timer[F].sleep(duration) *> fa).start.void

  }
```

This is the simplest implementation with the desired semantics. We could have chosen to do it differently, e.g. using a `Queue`. We could have also used the native `background` method provided by Cats Effect, which is a safer alternative to `start`. However, if we understand its trade-offs, this is more than acceptable.

Finally, let's stare in awe at our final checkout implementation.

```
def checkout(userId: UserId, card: Card): F[OrderId] =
  shoppingCart.get(userId)
    .ensure(EmptyCartError)(_.items.nonEmpty)
    .flatMap {
      case CartTotal(items, total) =>
        for {
          pid <- processPayment(Payment(userId, total, card))
          order <- createOrder(userId, pid, items, total)
          _ <- shoppingCart.delete(userId).attempt.void
        } yield order
    }
```

It has never been easier to manage effects in a purely functional way in Scala. Composing retry policies using standard typeclasses and sequencing actions using monadic combinators led us to our ultimate solution.

Our final class is defined as follows (`MonadThrow` is a type alias for `MonadError[F, Throwable]`):

```
final class CheckoutProgram[
    F[_]: Background: Logger: MonadThrow: Timer
](
    paymentClient: PaymentClient[F],
    shoppingCart: ShoppingCart[F],
    orders: Orders[F],
    retryPolicy: RetryPolicy[F]
) { ... }
```

In Chapter 7, when we talk about testing, we are going to see how to test `checkout` and other complex functions.

Chapter 5: HTTP layer

Our library of choice for serving requests via HTTP is going to be Http4s, a purely functional HTTP library built on top of Fs2 and Cats Effect. It is an extensive library, so it is recommended to read its documentation[1] if you're not familiar with it.

It is fundamental to understand functional effects to work with Http4s, specifically Cats Effect. In some cases, `fs2.Stream` is used as well, for which acquaintanceship with both libraries would help.

Notwithstanding, let's explore its API and unravel its potential.

[1] https://http4s.org/v0.21/

A server is a function

A simple HTTP server can be represented with the following function:

```
Request => Response
```

However, we commonly need to perform an effectful operation such as retrieving data from PostgreSQL before returning a response, so we need something more.

```
Request => F[Response]
```

In order to compose routes, we need to model the possibility that not every single request will have a matching route, so we can iterate over the list of routes and try to match the next one. When we reach the end, we give up and return a default response, more likely a 404 (Not Found). For such cases, we need a type that lets us express this optionality.

```
Request => F[Option[Response]]
```

This can also be expressed using the `OptionT` monad transformer, as shown below.

```
Request => OptionT[F, Response]
```

Finally, `Kleisli` - or also known as `ReaderT` - is a monad transformer for functions, so we can replace the `=>` (arrow) with it.

```
Kleisli[OptionT[F, *], Request, Response]
```

With a bit of modification to our `Request` and `Response` types, we get the following:

```
Kleisli[OptionT[F, *], Request[F], Response[F]]
```

> This is one of the *core types* of the library, aliased `HttpRoutes[F]`.

There are some cases where we need to guarantee that given a request, we can return a response (even if it is a default one). In such cases, we need to remove the optionality.

```
Kleisli[F, Request[F], Response[F]]
```

> Hereby we declare another core type of the library, aliased `HttpApp[F]`.

Both `HttpRoutes[F]` and `HttpApp[F]` share the same abstract definition.

```
type Http[F[_], G[_]] = Kleisli[F, Request[G], Response[G]]

type HttpApp[F[_]]    = Http[F, F]
type HttpRoutes[F[_]] = Http[OptionT[F, *], F]
```

This is a fine detail we don't really need to know about, though. Just remember the core types, we are going to be using them a lot.

Lastly, don't worry if you still don't understand everything. Let's try and make some sense of these definitions with some usage examples.

HTTP Routes #1

Now that we have introduced Http4s, let's see how we can model our HTTP endpoints, or more commonly called HTTP routes.

We are going to represent routes using `final class`es with an abstract effect type that can at least provide instances of `Applicative` and `Defer`, required by the `HttpRoutes`' constructor.

Imports are going to be omitted for conciseness; please refer to the source code of the project for the complete working version.

Brands

This is one of the easiest routes. It only exposes a `GET` endpoint to retrieve all the existing brands. We are going to model it as follows:

```
final class BrandRoutes[F[_]: Defer: Monad](
    brands: Brands[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/brands"

  private val httpRoutes: HttpRoutes[F] = HttpRoutes.of[F] {
    case GET -> Root =>
      Ok(brands.findAll)
  }

  val routes: HttpRoutes[F] = Router(
    prefixPath -> httpRoutes
  )

}
```

There are a few things going on here:

- We have some constraints on `F[_]`:
  - `Defer` is required by `HttpRoutes.of[F]`.
  - `Monad` is needed to create a response, and it also gives us `Applicative`, also needed for `HttpRoutes.of[F]`.
- We have the `Brands[F]` algebra as an argument to our class.
- We extend `Http4sDsl[F]`, which should be self-explanatory.
- There is a `prefixPath` made `private`, which indicates the root of our endpoint.

- We have a private `httpRoutes` defining all our endpoints, only one in this case.
- Finally, we have a public `routes` which uses a `Router` that lets us add a `prefixPath` to a group of endpoints denoted as `HttpRoutes`.

Having a `prefixPath` and `httpRoutes` as `private` functions is just my preference, but I do consider it a good practice. This is roughly the same structure we will be using for the rest of our HTTP routes.

One last thing, when we say `Ok(brands.findAll)`, a few things are happening under the hood:

- `Ok.apply` builds a response with code 200 (Ok) for us.
- To build the response body, Http4s requires an `EntityEncoder[F, A]`, where `A` is the return type of `brands.findAll`, in this case, `List[Brand]`. Well, technically it is `F[List[Brand]]`, but the library will `flatMap` that for us and return a `Response[F]`.
- The most common encoding is `JSON`, for which we can use the Circe library. We will see how to deal with it in the last section of this chapter.

Categories

Our Category routes is fairly similar to the Brand routes.

```
final class CategoryRoutes[F[_]: Defer: Monad](
    categories: Categories[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/categories"

  private val httpRoutes: HttpRoutes[F] = HttpRoutes.of[F] {
    case GET -> Root =>
      Ok(categories.findAll)
  }

  val routes: HttpRoutes[F] = Router(
    prefixPath -> httpRoutes
  )

}
```

We are using a different algebra, `Categories[F]`, and a distinct `prefixPath`. The rest remains the same.

Items

Our Item routes introduces something new.

```
final class ItemRoutes[F[_]: Defer: Monad](
    items: Items[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/items"

  object BrandQueryParam extends
    OptionalQueryParamDecoderMatcher[BrandParam]("brand")

  private val httpRoutes: HttpRoutes[F] = HttpRoutes.of[F] {

    case GET -> Root :? BrandQueryParam(brand) =>
      Ok(brand.fold(items.findAll)(b => items.findBy(b.toDomain)))

  }

  val routes: HttpRoutes[F] = Router(
    prefixPath -> httpRoutes
  )

}
```

Since we should be able to filter by Brand, we have introduced an optional query parameter named "brand". What is great is that we can perform validation using Refined too! See how `BrandParam` is defined below.

```
@newtype case class BrandParam(value: NonEmptyString) {
  def toDomain: BrandName =
    BrandName(value.value.toLowerCase.capitalize)
}
```

Exactly what we have learned in Chapter 2: combining Newtype and Refined to obtain strongly-typed functions. We require our `BrandParam` to be a `NonEmptyString`.

To get this compiling, we need a `QueryParamDecoder` instance for refinement types.

```
implicit def refinedParamDecoder[T: QueryParamDecoder, P](
    implicit ev: Validate[T, P]
): QueryParamDecoder[T Refined P] =
  QueryParamDecoder[T].emap(
    refineV[P](_).leftMap(m => ParseFailure(m, m))
  )
```

If we make a `GET` request to `/items?brands=`, we will get a response code 400 (Bad Request) along with a message indicating that our input is empty. Otherwise, we will retrieve the list of items filtering by the given brand.

If we omit the `brand` parameter, making a `GET` request to `/items`, we will just return all the items, folding over our optional query parameter as we can see below.

```
case GET -> Root :? BrandQueryParam(brand) =>
  Ok(brand.fold(items.findAll)(b => items.findBy(b.toDomain)))
```

This is how Http4s lets us indicate that a parameter is optional, using the symbol `:?`, provided by its DSL.

Authentication

In order to get access to the authenticated user, we need to use `AuthedRequest[F, User]`, which is a wrapper for `(User, Request[F])`, where `User` is some arbitrary datatype we declare to represent a user in our system.

In the same way, we should use `AuthedRoutes[User, F]` instead of `HttpRoutes[F]`, if we want to access the authenticated user in every request. For example:

```
val authedRoutes: AuthedRoutes[User, IO] =
  AuthedRoutes.of {
    case GET -> Root as user =>
      Ok(s"Welcome, ${user.name}")
  }
```

`AuthedRoutes[T, F]` is a type alias for `Kleisli[OptionT[F, *], AuthedRequest[F, T], Response[F]]`, same as `HttpRoutes`, except the request type now contains information about the user.

Http4s allows us to determine how to authenticate a user. All we need is a function that decides whether a user could be authenticated or not given a `Request[F]`, and a function that dictates what to do in case of failure. Once we have both functions, we can apply them to `AuthMiddleware`, which can be used as another ordinary middleware. We are going to explain middlewares in detail later in this chapter.

As a demonstration, see the example below.

```
val authUser: Kleisli[F, Request[F], Either[String,User]] =
  Kleisli(_ => myUser.asRight.pure[F]) // Authenticate user

val onFailure: AuthedRoutes[String, F] =
  Kleisli(req => OptionT.liftF(Forbidden(req.authInfo)))

val middleware = AuthMiddleware(authUser, onFailure)

val routes: HttpRoutes[F] = middleware(authedRoutes)
```

The most common methods of authentication are *cookies* and *bearer token*. You can find examples of both in the official docs. However, we are going to specialize in the latter.

JWT Auth

Our library of choice will be the opinionated Http4s JWT Auth[2], which offers some functionality on top of Http4s and JWT Scala[3].

Http4s provides an `AuthMiddleware`, previously mentioned. It is defined as follows:

```
type AuthMiddleware[F[_], T] = Middleware[
  OptionT[F, *], AuthedRequest[F, T],
  Response[F], Request[F], Response[F]
]
```

We are going to see middlewares shortly. For now, it is fine to think of them as functions `AuthedRoutes[T, F] => HttpRoutes[F]`.

Instead of a normal `AuthMiddleware`, we are going to use a custom `JwtAuthMiddleware`, defined by Http4s JWT Auth.

```
val usersAuth: JwtToken => JwtClaim => F[Option[User]] =
  t => c => User("Joe").some.pure[F]

val usersMiddleware: AuthMiddleware[F, User] =
  JwtAuthMiddleware[F, User](jwtAuth, usersAuth)
```

It requires a `JwtAuth` and a function `JwtToken => JwtClaim => F[Option[A]]`, as shown above. The former can be created as follows:

```
val jwtAuth = JwtAuth.hmac("53cr3t", JwtAlgorithm.HS256)
```

Once we have defined everything we need, we can use our `AuthMiddleware` as any other middleware. For example:

```
val routes: HttpRoutes[F] = usersMiddleware(authedRoutes)
```

Following the same principle, we could implement authentication for other kinds of users, such as admin users.

```
val adminAuth: JwtToken => JwtClaim => F[Option[AdminUser]] =
  t => c => AdminUser("admin").some.pure[F]

val adminMiddleware: AuthMiddleware[F, AdminUser] =
  JwtAuthMiddleware[F, AdminUser](jwtAuth, adminAuth)
```

This is how we could use it:

[2] https://github.com/profunktor/http4s-jwt-auth
[3] https://github.com/pauldijou/jwt-scala

```
val adminRoutes: AuthedRoutes[AdminUser, F] =
  AuthedRoutes.of {
    case ar @ POST -> Root as adminUser =>
      Ok(s"You have admin rights, $adminUser!")
  }

val routes: HttpRoutes[F] = adminMiddleware(adminRoutes)
```

It is interesting to notice that we can combine the HTTP routes of `Users` and `AdminUsers`, achieving the functionality of having different roles.

```
val allRoutes: HttpRoutes[F] =
  usersMiddleware(authedRoutes) <+> adminMiddleware(adminRoutes)
```

Http4s is highly compositional.

HTTP Routes #2

Now that we have learned about authentication, let's continue defining the secured and admin HTTP routes of our application.

Shopping Cart

So far, we have only dealt with open routes that don't require authentication. This is not the case for our shopping cart routes, though, which needs a user to be logged in.

```
final class CartRoutes[F[_]: Defer: JsonDecoder: Monad](
    shoppingCart: ShoppingCart[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/cart"

  private val httpRoutes: AuthedRoutes[CommonUser, F] =
    AuthedRoutes.of {
      // Get shopping cart
      case GET -> Root as user =>
        Ok(shoppingCart.get(user.value.id))

      // Add items to the cart
      case ar @ POST -> Root as user =>
        ar.req.asJsonDecode[Cart].flatMap { cart =>
          cart.items
            .map {
              case (id, quantity) =>
                shoppingCart
                  .add(user.value.id, id, quantity)
            }
            .toList
            .sequence *> Created()
        }

      // Modify items in the cart
      case ar @ PUT -> Root as user =>
        ar.req.asJsonDecode[Cart].flatMap { cart =>
          shoppingCart
            .update(user.value.id, cart) *> Ok()
        }

      // Remove item from the cart
```

```
      case DELETE -> Root / UUIDVar(uuid) as user =>
        shoppingCart.removeItem(
          user.value.id,
          ItemId(uuid)
        ) *> NoContent()
    }

  def routes(
      authMiddleware: AuthMiddleware[F, CommonUser]
  ): HttpRoutes[F] = Router(
    prefixPath -> authMiddleware(httpRoutes)
  )

}
```

Let's break it apart since there is a lot going on here.

- We have a new constraint `JsonDecoder`, which is just an interface that lets us decode our request as the required entity `A`, given a `Decoder[A]`.
- We are using `AuthedRoutes[CommonUser, F]` instead of `HttpRoutes[F]`.
- Our `routes` takes an `AuthMiddleware[F, CommonUser]` as an argument.
- We are decoding data in our `POST` and `PUT` endpoints, which is done via `JsonDecoder` (it requires a `Decoder[Cart]`).
- We are capturing a path parameter in our `DELETE` method, using `UUIDVar`.

We will see how to deal with JSON decoding in the next section. Other than that, the rest should be self-explanatory.

Orders

Without much ado, here is the definition of `OrderRoutes`:

```
final class OrderRoutes[F[_]: Defer: Monad](
    orders: Orders[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/orders"

  private val httpRoutes: AuthedRoutes[CommonUser, F] =
    AuthedRoutes.of {

      case GET -> Root as user =>
        Ok(orders.findBy(user.value.id))
```

```
      case GET -> Root / UUIDVar(orderId) as user =>
        Ok(orders.get(user.value.id, OrderId(orderId)))

    }

  def routes(
      authMiddleware: AuthMiddleware[F, CommonUser]
  ): HttpRoutes[F] = Router(
    prefixPath -> authMiddleware(httpRoutes)
  )

}
```

Another authenticated endpoint, nothing new here.

Checkout

This endpoint is very interesting, as it performs quite a lot of error handling.

```
final class CheckoutRoutes[F[_]: Defer: JsonDecoder: MonadThrow](
    program: CheckoutProgram[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/checkout"

  private val httpRoutes: AuthedRoutes[CommonUser, F] =
    AuthedRoutes.of {

      case ar @ POST -> Root as user =>
        ar.req.decodeR[Card] { card =>
          program
            .checkout(user.value.id, card)
            .flatMap(Created(_))
            .recoverWith {
              case CartNotFound(userId) =>
                NotFound(
                  s"Cart not found for user: ${userId.value}"
                )
              case EmptyCartError =>
                BadRequest("Shopping cart is empty!")
              case PaymentError(cause) =>
                BadRequest(cause)
              case OrderError(cause) =>
```

```
                BadRequest(cause)
            }
        }

    }

  def routes(
      authMiddleware: AuthMiddleware[F, CommonUser]
  ): HttpRoutes[F] = Router(
    prefixPath -> authMiddleware(httpRoutes)
  )

}
```

Using `recoverWith`, provided by the `ApplicativeError` instance we have in scope, we can recover from business errors and return the appropriate response. In the last chapter, we are going to modify it to use classy prisms instead, as explained in Chapter 2.

Another new function to discern is `decodeR`, which is a custom decoding function that deals with validation errors from the Refined library and returns a response code 400 (Bad Request) along with an error message, instead of the default response code 422 (Unprocessable Entity), when there is an invalid input such as an empty name. Find its definition below.

```
implicit class RefinedRequestDecoder[F[_]: JsonDecoder: MonadThrow](
    req: Request[F]
) extends Http4sDsl[F] {

  def decodeR[A: Decoder](
      f: A => F[Response[F]]
  ): F[Response[F]] =
    req.asJsonDecode[A].attempt.flatMap {
      case Left(e) =>
        Option(e.getCause) match {
          case Some(c) if c.getMessage.startsWith("Predicate") =>
            BadRequest(c.getMessage)
          case _ =>
            UnprocessableEntity()
        }
      case Right(a) => f(a)
    }

}
```

We need to validate the credit card details entered by the user, to avoid hitting the remote payment system with invalid data, for which we have defined the `Card` datatype using refinement types, as shown below.

```
type Rgx = W.`"^[a-zA-Z]+(([',. -][a-zA-Z ])?[a-zA-Z]*)*$"`.T

type CardNamePred = String Refined MatchesRegex[Rgx]

type CardNumberPred     = Long Refined Size[16]
type CardExpirationPred = Int Refined Size[4]
type CardCCVPred        = Int Refined Size[3]

@newtype case class CardName(value: CardNamePred)
@newtype case class CardNumber(value: CardNumberPred)
@newtype case class CardExpiration(value: CardExpirationPred)
@newtype case class CardCCV(value: CardCCVPred)

case class Card(
    name: CardName,
    number: CardNumber,
    expiration: CardExpiration,
    ccv: CardCCV
)
```

This is what we have for now, but software evolves quickly and might require further refinements to avoid invalid data in our application.

Login

```
final class LoginRoutes[F[_]: Defer: JsonDecoder: MonadThrow](
    auth: Auth[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/auth"

  private val httpRoutes: HttpRoutes[F] =
    HttpRoutes.of[F] {

      case req @ POST -> Root / "login" =>
        req.decodeR[LoginUser] { user =>
          auth
            .login(user.username.toDomain, user.password.toDomain)
            .flatMap(Ok(_))
```

```
            .recoverWith {
              case InvalidUserOrPassword(_) => Forbidden()
            }
        }

    }

  val routes: HttpRoutes[F] = Router(
    prefixPath -> httpRoutes
  )

}
```

The only thing to notice here is the use of the extension method `toDomain`, which converts refined values into common domain values.

Logout

```
final class LogoutRoutes[F[_]: Defer: Monad](
    auth: Auth[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/auth"

  private val httpRoutes: AuthedRoutes[CommonUser, F] =
    AuthedRoutes.of {
      case ar @ POST -> Root / "logout" as user =>
        AuthHeaders
          .getBearerToken(ar.req)
          .traverse_(t =>
            auth.logout(t, user.value.name)
          ) *> NoContent()
    }

  def routes(
      authMiddleware: AuthMiddleware[F, CommonUser]
  ): HttpRoutes[F] = Router(
    prefixPath -> authMiddleware(httpRoutes)
  )

}
```

We are accessing the headers of the request to find the current access token and invalidate it, which means removing it from our cache, as we will see in the `Auth` interpreter.

Users

```
final class UserRoutes[F[_]: Defer: JsonDecoder: MonadThrow](
    auth: Auth[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/auth"

  private val httpRoutes: HttpRoutes[F] =
    HttpRoutes.of[F] {

      case req @ POST -> Root / "users" =>
        req
          .decodeR[CreateUser] { user =>
            auth
              .newUser(
                user.username.toDomain,
                user.password.toDomain
              )
              .flatMap(Created(_))
              .recoverWith {
                case UserNameInUse(u) =>
                  Conflict(u.value)
              }
          }

    }

  val routes: HttpRoutes[F] = Router(
    prefixPath -> httpRoutes
  )

}
```

We are only able to register new common users; it is not possible to create new admin users. Once again, we are using `decodeR` for validation, `toDomain` for data conversion, and `recoverWith` for business logic error handling.

Brands Admin

Finally, we reached the administrative endpoints. In this case, admin users should be able to create new brands.

```
final class AdminBrandRoutes[F[_]: Defer: JsonDecoder: MonadThrow](
    brands: Brands[F]
) extends Http4sDsl[F] {

  private[admin] val prefixPath = "/brands"

  private val httpRoutes: AuthedRoutes[AdminUser, F] =
    AuthedRoutes.of {
      case ar @ POST -> Root as _ =>
        ar.req.decodeR[BrandParam] { bp =>
          Created(brands.create(bp.toDomain))
        }
    }

  def routes(
      authMiddleware: AuthMiddleware[F, AdminUser]
  ): HttpRoutes[F] = Router(
    prefixPath -> authMiddleware(httpRoutes)
  )

}
```

Using `decodeR`, we validate the brand received in our request body is not empty.

Categories Admin

Pretty similar to our admin brands endpoint.

```
final class AdminCategoryRoutes[
    F[_]: Defer: JsonDecoder: MonadThrow
](
    categories: Categories[F]
) extends Http4sDsl[F] {

  private[admin] val prefixPath = "/categories"

  private val httpRoutes: AuthedRoutes[AdminUser, F] =
    AuthedRoutes.of {
      case ar @ POST -> Root as _ =>
        ar.req.decodeR[CategoryParam] { c =>
          Created(categories.create(c.toDomain))
        }
    }
```

```
  def routes(
      authMiddleware: AuthMiddleware[F, AdminUser]
  ): HttpRoutes[F] = Router(
    prefixPath -> authMiddleware(httpRoutes)
  )

}
```

A Category must not be empty, as well.

```
@newtype case class CategoryParam(value: NonEmptyString) {
  def toDomain: CategoryName =
    CategoryName(value.value.toLowerCase.capitalize)
}
```

Items Admin

Admin users should be able to create items as well as updating their prices.

```
final class AdminItemRoutes[F[_]: Defer: JsonDecoder: MonadThrow](
    items: Items[F]
) extends Http4sDsl[F] {

  private[admin] val prefixPath = "/items"

  private val httpRoutes: AuthedRoutes[AdminUser, F] =
    AuthedRoutes.of {
      // Create new item
      case ar @ POST -> Root as _ =>
        ar.req.decodeR[CreateItemParam] { item =>
          Created(items.create(item.toDomain))
        }

      // Update price of item
      case ar @ PUT -> Root as _ =>
        ar.req.decodeR[UpdateItemParam] { item =>
          Ok(items.update(item.toDomain))
        }
    }

  def routes(
      authMiddleware: AuthMiddleware[F, AdminUser]
  ): HttpRoutes[F] = Router(
```

```
    prefixPath -> authMiddleware(httpRoutes)
  )

}
```

At this point, nothing here should come as a surprise. Let's have a look at the domain model for item creation.

```
@newtype
case class ItemNameParam(value: NonEmptyString)
@newtype
case class ItemDescriptionParam(value: NonEmptyString)
@newtype
case class PriceParam(value: String Refined ValidBigDecimal)

case class CreateItemParam(
    name: ItemNameParam,
    description: ItemDescriptionParam,
    price: PriceParam
) {
  def toDomain: CreateItem =
    CreateItem(
      ItemName(name.value.value),
      ItemDescription(description.value.value),
      USD(price)
    )
}
```

`USD` is one of the many concrete implementations of the `Money` type.

Below we define the domain model for the item's price update.

```
@newtype case class ItemIdParam(value: String Refined Uuid)

case class UpdateItemParam(
    id: ItemIdParam,
    price: PriceParam
) {
  def toDomain: UpdateItem =
    UpdateItem(
      ItemId(UUID.fromString(id.value.value)),
      USD(price)
    )
}
```

```
case class UpdateItem(
    id: ItemId,
    price: Money
)
```

Once again, leveraging our new favorite team Newtype-Refined, aiming for a strongly-typed application.

Composing routes

HTTP routes are functions, and functions compose; we can compose HTTP routes too!

Say we have the following routes:

```
val userRoutes: HttpRoutes[F] = ???
val itemRoutes: HttpRoutes[F] = ???
```

We can use the SemigroupK[4] instance for `Kleisli` to combine them.

```
val allRoutes: HttpRoutes[F] =
  userRoutes <+> itemRoutes
```

`SemigroupK` comes from Cats Core, so be sure to have `import cats.implicits._` in scope. It is very similar to `Semigroup`; the difference is that `SemigroupK` operates on type constructors of one argument, i.e. `F[_]`.

[4] https://typelevel.org/cats/typeclasses/semigroupk.html

Middlewares

Middlewares allow us to manipulate `Requests` and `Responses`. It is expressed as a function. The two most common middlewares have either of the following shapes:

```
HttpRoutes[F] => HttpRoutes[F]
```

Or:

```
HttpApp[F] => HttpApp[F]
```

Even though, its definition is more generic.

```
type Middleware[F[_], A, B, C, D] =
  Kleisli[F, A, B] => Kleisli[F, C, D]
```

There are a few predefined middlewares we can make use of, e.g. `CORS` middleware. If we wanted to support `CORS` (Cross-Origin Resource Sharing) for all our routes, we could do the following:

```
val modRoutes: HttpRoutes[F] = CORS(allRoutes)
```

The official documentation is pretty good, you can find all this information right there, so we are not going to be repeating the same thing in this book. Instead, we are going to focus on compositionality and best practices.

Compositionality

Since middlewares are functions, we can define a single function that combines all the middlewares we want to apply to all our HTTP routes. Here is one simple way to do it:

```
val middleware: HttpRoutes[F] => HttpRoutes[F] = {
  { http: HttpRoutes[F] =>
    AutoSlash(http)
  } andThen { http: HttpRoutes[F] =>
    CORS(http, CORS.DefaultCORSConfig)
  } andThen { http: HttpRoutes[F] =>
    Timeout(60.seconds)(http)
  }
}
```

Some middlewares require an `HttpApp[F]` instead of `HttpRoutes[F]`. In such a case, it is better to declare them separately.

```
val closedMiddleware: HttpApp[F] => HttpApp[F] = {
  { http: HttpApp[F] =>
    RequestLogger.httpApp(true, true)(http)
  } andThen { http: HttpApp[F] =>
    ResponseLogger.httpApp(true, true)(http)
  }
}
```

Then, we can compose them together as follows:

```
val finalRoutes: HttpApp[F] =
  closedMiddleware(middleware(allRoutes).orNotFound)
```

The extension method `orNotFound` comes from `import org.http4s.implicits._`. It turns `HttpRoutes` into `HttpApp`s by returning a default response code 404 (Not Found) in case it cannot match on any of our HTTP routes.

This is truly capitalizing the power of functions and compositionality of the Http4s library. I couldn't recommend it enough.

Running server

We are going to quickly see how to run our server. Up to this point, we have only seen functions, but we need something else to get a running HTTP server. Let me introduce you to the default server implementation, *Blaze*.

Here is one way to start a server:

```
val httpApp: HttpApp[F] = ???

BlazeServerBuilder[IO]
  .bindHttp(8080, "0.0.0.0")
  .withHttpApp(httpApp)
  .serve
```

The `serve` function returns a `Stream[F, ExitCode]`. If we are using `IOApp`, we can just call `compile.drain.as(ExitCode.Success)` to get up and running.

Another useful function is `resource`, which unmistakably, returns a `Resource[F, Server[F]]`. This function comes in handy when you want to compose different resources together, such as a remote database or a message broker.

In Chapter 8, when we put all the pieces together, we are going to see how to initialize our dependencies and start our server up.

Entity codecs

Previously, I have briefly mentioned `EntityEncoder[F, A]` and shortly explained `JsonDecoder`. Since we are going to expose a JSON API, we will only need the following implicit in scope when encoding data in our HTTP routes:

```
implicit def deriveEntityEncoder[
    F[_]: Applicative,
    A: Encoder
]: EntityEncoder[F, A] = jsonEncoderOf[F, A]
```

The `jsonEncoderOf` method comes from the `http4s-circe` module.

> **Notes**
>
> `We refer to codecs as having both an encoder and a decoder`

Decoding, on the other hand, it is already abstracted away by `JsonDecoder`, which provides a default instance for any `F[_]: Sync` that can be summoned at the edge of the application. If we were working on an API that needed to decode other formats such as XML, we would need an `EntityDecoder[F, A]`.

JSON codecs

`Decoder`s and `Encoder`s come from the Circe library. It is recommended to use `semiauto` derivation in most cases. For example, let's see how we could derive codecs for our `Brand` domain datatype.

Here is our datatype once again:

```
@newtype case class BrandId(value: UUID)
@newtype case class BrandName(value: String)

case class Brand(uuid: BrandId, name: BrandName)
```

If we were not using the Newtype library, we could derive a `Decoder[Brand]` and `Encoder[Brand]` with the following code:

```
implicit val brandDecoder: Decoder[Brand] =
  deriveDecoder[Brand]

implicit val brandEncoder: Encoder[Brand] =
  deriveEncoder[Brand]
```

This is because Circe already comes with codecs for common types such as `UUID` and `String`. However, we need to do something else in order to get the derivation working using Newtype.

```
implicit def coercibleDecoder[
    A: Coercible[B, *],
    B: Decoder
]: Decoder[A] = Decoder[B].map(_.coerce[A])

implicit def coercibleEncoder[
    A: Coercible[B, *],
    B: Encoder
]: Encoder[A] = Encoder[B].contramap(_.repr.asInstanceOf[B])
```

`Coercible` is a typeclass defined in the Newtype library. Though, it is not completely necessary to fully understand what is going on here. It suffices to know that if we are wrapping a type supported by Circe with a newtype, then we can also get codecs for our newtype.

Unfortunately, the Scala compiler is not able to infer the type of `repr`, which is effectively `B`, so we need to perform a (safe) type-cast.

Another useful addition would be to have a `KeyEncoder` and `KeyDecoder` for coercible types (newtypes), which are required if we need to encode or decode Maps, respectively.

```
implicit def coercibleKeyDecoder[
    A: Coercible[B, *],
    B: KeyDecoder
]: KeyDecoder[A] =
    KeyDecoder[B].map(_.coerce[A])

implicit def coercibleKeyEncoder[
    A: Coercible[B, *],
    B: KeyEncoder
]: KeyEncoder[A] =
    KeyEncoder[B].contramap[A](_.repr.asInstanceOf[B])
```

Having these useful codecs in scope makes our semi-automatic derivation work:

```
implicit val brandDecoder: Decoder[Brand] =
   deriveDecoder[Brand]

implicit val brandEncoder: Encoder[Brand] =
  deriveEncoder[Brand]
```

This is how we are going to derive most of our codecs. Yet, we might need to do it manually sometimes:

```
implicit val tokenEncoder: Encoder[JwtToken] =
  Encoder.forProduct1("access_token")(_.value)

implicit val cartDecoder: Decoder[Cart] =
  Decoder.forProduct1("items")(Cart.apply)
```

As we can see, it is quite straightforward.

Where to place our JSON codecs?

A good question we haven't asked ourselves yet is, where do we place all our JSON codecs? We have a few options.

A common practice is to place such instances in the companion object of our datatypes, as Scala can easily find them there.

```
@newtype case class PersonAge(value: Int)
@newtype case class PersonName(value: String)

case class Person(name: PersonName, age: PersonAge)

object Person {
  implicit val jsonEncoder: Encoder[Person] = deriveEncoder[Person]
  implicit val jsonDecoder: Decoder[Person] = deriveDecoder[Person]
}
```

This guarantees *global coherence*, as orphan instances would be immediately rejected by the Scala compiler.

> **Notes**
>
> Global coherence allows only one typeclass instance per type

There is a drawback with this approach, though. Generic codecs for newtypes (`Coercible` types) can be defined only once, as we have previously seen, so there is no need to write them one by one in their companion objects.

Sometimes we may also want to have a JSON encoder to be used in our HTTP API and another JSON encoder to be used to store serialized data in our database.

For this reason, I believe it is acceptable to place JSON codecs to be used in our HTTP API in a specific file, e.g. `shop.http.json.scala`. This way, we can bring them on demand when we need them by writing `import shop.http.json._`.

Occasionally we may also need codecs for datatypes we do not own, e.g. those coming from a third-party library. This is another incentive to group codecs in a single file instead of defining them in companion objects spread around our codebase.

The downside of this strategy is that we need to manually ensure there are no orphan instances.

There is a trade-off for every decision we make. So whatever yours is, hold on to it.

Validation

In many cases, we are using refinement types that need to be either encoded or decoded as JSON. For this purpose, we are going to use the `circe-refined` module, which can derive a few instances for us.

Our `Card` domain model is one of the most refined types we have so far. If we tried to derive a `Decoder` for it, it would fail.

```
implicit val cardDecoder: Decoder[Card] = deriveDecoder[Card]
```

However, this can be easily fixed with a single import.

```
import io.circe.refined._
```

Well, not that easy. Our derivation still wouldn't compile. We have the following refinement types in our `Card` model:

```
type Rgx = W.`"^[a-zA-Z]+(([',. -][a-zA-Z ])?[a-zA-Z]*)*$"`.T

type CardNamePred       = String Refined MatchesRegex[Rgx]
type CardNumberPred     = Long Refined Size[16]
type CardExpirationPred = String Refined (Size[4] And ValidInt)
type CardCCVPred        = Int Refined Size[3]
```

Followed by its definition:

```
@newtype case class CardName(value: CardNamePred)
@newtype case class CardNumber(value: CardNumberPred)
@newtype case class CardExpiration(value: CardExpirationPred)
@newtype case class CardCCV(value: CardCCVPred)

case class Card(
    name: CardName,
    number: CardNumber,
    expiration: CardExpiration,
    ccv: CardCCV
)
```

Unfortunately, the Circe Refined module doesn't come with instances for `Size[N]`, where `N` is an arbitrary literal number. So we should come up with it.

```
implicit def validateSizeN[N <: Int, R](
    implicit w: ValueOf[N]
): Validate.Plain[R, Size[N]] =
    Validate.fromPredicate[R, Size[N]](
        _.toString.size == w.value,
        _ => s"Must have ${w.value} digits",
        Size[N](w.value)
    )
```

Refined needs a `Validate` instance for every possible size. Fortunately, we can abstract over its arity with a little bit of magic and finally get our `Card` derivation working.

HTTP client

Up until now, we have only talked about the server-side of what Http4s offers. Yet, little did we talk about the client-side.

Expectedly, Http4s also comes with support for clients, the most popular implementation being Blaze as on the server-side.

Payment client

Let's recap on our payment's algebra.

```
trait PaymentClient[F[_]] {
  def process(payment: Payment): F[PaymentId]
}
```

As usual, we do not have any implementation details in our interface. This is going to be delegated to our interpreter, where we are going to use a real HTTP client.

```
class LivePaymentClient[F[_]: JsonDecoder: MonadThrow](
    client: Client[F]
) extends PaymentClient[F]
    with Http4sClientDsl[F] {

  private val baseUri = "http://localhost:8080/api/v1"

  def process(payment: Payment): F[PaymentId] =
    Uri
      .fromString(baseUri + "/payments")
      .liftTo[F]
      .flatMap { uri =>
        client.fetchAs[PaymentId](POST(payment, uri))
      }

}
```

Our interpreter takes a `Client[F]` as an argument, which is the abstract interface for all the different HTTP clients the library supports. It comes from the package `org.http4s.client`.

Notice how we also mix-in the `Http4sClientDsl` interface, which will grant us access to a friendly DSL to build HTTP requests.

Our `process` function only makes a call to the remote API, expecting a `PaymentId` as the response body. The `fetchAs` function is defined as follows:

```
def fetchAs[A](
  req: Request[F]
)(implicit d: EntityDecoder[F, A]): F[A]
```

This is the most optimistic scenario as we are not handling the possibility of a duplicate payment error. To do so, we need a function different from `fetchAs` that lets us manipulate the response we get from the client. What we need is `fetch`.

```
def fetch[A](req: Request[F])(f: Response[F] => F[A]): F[A]
```

This is another function given by `Client[F]` which, in addition to a `Request[F]`, takes a function `Response[F] => F[A]`. This is our opportunity to do things right.

```
def process(payment: Payment): F[PaymentId] =
  Uri
    .fromString(baseUri + "/payments")
    .liftTo[F]
    .flatMap { uri =>
      client.fetch[PaymentId](POST(payment, uri)) { r =>
        if (r.status == Status.Ok || r.status == Status.Conflict)
          r.asJsonDecode[PaymentId]
        else
          PaymentError(
            Option(r.status.reason).getOrElse("Unknown")
          ).raiseError[F, PaymentId]
      }
    }
```

When we get a `Response`, we check its status. If it is either 200 (Ok) or 409 (Conflict), we know we can expect a `PaymentId` as the body of the response. In such a case, we try to automatically decode it using our JSON decoders. This is what the `asJsonDecode[A]` function does, defined as follows:

```
def asJsonDecode[A: Decoder](m: Message[F]): F[A]
```

That is all we have to do. If other kinds of errors occur, such as a network failure, we are going to let it fail. Whatever component makes use of it, should handle that.

Creating a client

A `Client` is created in a similar way a `Server` is created. Instead of `serve`, not present in `BlazeClientBuilder`, we are going to use the `resource` function which gives us a `Resource[F, Client[F]]`.

```
BlazeClientBuilder[F](ExecutionContext.global)
  .resource
  .use { client =>
    new LivePaymentClient[F](client)
  }
```

See what just happened here. Once we have a resource, we call `use` and pass the `Client[F]` we get as an argument to our payment client interpreter. This is exactly the shared state design pattern we have explored in Chapter 2.

A `Client` is treated as a resource because it contains a connection pool and a scheduler, which have a lifecycle.

In Chapter 8, we are going to see how all the resources in our application are composed together, including our HTTP Client.

Chapter 6: Persistent layer

We are halfway through the book, time to talk about interpreters! Some of the algebras need implementations based on PostgreSQL; others based on Redis.

In this chapter, we are going to learn how to deal with blocking and non-blocking operations, and how to manage a connection pool, among other things.

Skunk & Doobie

In the Scala ecosystem, there are a couple of libraries that let us interact with Postgres. Arguably, the most popular one in the FP ecosystem is Doobie[1], having more than 1.5k stars at the moment of writing.

Quoting the Wikipedia[2]:

> PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and technical standards compliance. It is designed to handle a range of workloads, from single machines to data warehouses or Web services with many concurrent users.

Doobie is a JDBC wrapper that integrates very well with Cats Effect and Fs2. Those looking for a mature and battle-tested library should go for it.

For those brave enough, there is Skunk[3], a purely functional and asynchronous Postgres library for Scala. It talks the Postgres protocol directly (no JDBC), and it features excellent error reporting. Although it is quite immature and not production-ready (still in development), it will eventually replace Doobie (in my humble opinion) because of its promising features.

Warning

`Skunk is in development and is considered not production-ready`

Given our disclaimer, let's start exploring Skunk's API[4] as we will be using it in our application. If you decide to use Doobie instead, you can learn from its great documentation[5]. This chapter will be helpful, regardless.

If you are already acquainted with Doobie, you will observe that many properties are shared with Skunk.

Session Pool

First, we need to connect to the Postgres server. Skunk supports acquiring a connection using `Session.single[F](...)`, which returns a `Resource[F, Session[F]]`. This is

[1] https://github.com/tpolecat/doobie
[2] https://en.wikipedia.org/wiki/PostgreSQL
[3] https://github.com/tpolecat/skunk
[4] https://tpolecat.github.io/skunk/index.html
[5] https://tpolecat.github.io/doobie/book.html

fine for simple examples, but for an application, we need a pool of sessions to be able to handle concurrent operations. What we need is the following construct:

```
Session
  .pooled[F](
    host = "localhost",
    port = 5432,
    user = "postgres",
    database = "store",
    max = 10
  )
```

What we get back is a `SessionPool[F]`, which is defined as follows:

```
type SessionPool[F[_]] = Resource[F, Resource[F, Session[F]]]
```

It is a pool of sessions limited to a maximum of 10 open sessions at a time. So all we need to pass to the interpreters is `Resource[F, Session[F]]` (shared state) and call `use` for every transaction we need to run. Skunk will handle concurrent access for us.

This is the safest usage of sessions since our Postgres interpreters will only perform standalone operations, which might then be combined concurrently at a higher level. Skunk also supports transactions[6], though, we are not going to make use of this feature.

Queries

We need to be able to retrieve rows of information from one or more database tables. For this purpose, there exists the `Query` type.

Notes

```
A Query is a SQL statement that can return rows
```

Let's look at the following example:

```
val countryQuery: Query[Void, String] =
  sql"SELECT name FROM country".query(varchar)
```

We can observe a `sql` interpolator that parses a SQL statement into a `Fragment` to then be turned into a `Query` by calling the `query` method. Lastly, we have `varchar`, which is a `Decoder` defining a relationship between the Postgres type `VARCHAR` and the Scala type `String`.

[6] https://tpolecat.github.io/skunk/tutorial/Transactions.html

To learn more about it, have a look at the Schema Types[7] reference. You can also explore its source code; they can all be found under the `skunk.codec` package.

In order to execute the query, we need a `Session[F]`. For example:

```
val result: F[List[String]] =
  session.execute(countryQuery)
```

In addition to the `execute` method, there are the `option` and `unique` methods, returning `F[Option[A]]` and `F[A]`, respectively.

Commands

We have seen how we can get a result from a Query. In order to insert, update, or delete some records, we need a Command, which typically performs state mutation in the database.

Notes

A Command is a SQL statement that does not return rows

Let's see how we can create a new country in our database.

```
val insertCmd: Command[Long ~ String] =
  sql"""
      INSERT INTO country
      VALUES ($int8, $varchar)
     """.command
```

Allegedly, the country table has only two columns: an `id` of type `INT8`, and a `name` of type `VARCHAR`.

> Skunk speaks in terms of Postgres schema types rather than ANSI types or common aliases, thus we use `INT8` here rather than `BIGINT`.

The return type indicates the number of arguments we need to supply in order to execute the statement, defined as a product type (aliased `~`). For example:

```
session.prepare(insertCmd).use { cmd =>
  cmd.execute(1L ~ "Argentina").void
}
```

[7] https://tpolecat.github.io/skunk/reference/SchemaTypes.html

We have created a *prepared statement* by calling the `prepare` method, and we have got back a `Resource[F, PreparedCommand[F, A]]`. Once we are ready to execute the statement, we call `use` to access the inner prepared command that lets us `execute` it by supplying the required arguments. Finally, we call `void` to ignore its result, which might indicate the number of rows inserted.

We could also do something with its result (exercise left to the reader). However, Postgres rarely returns "0 rows inserted". If anything goes wrong, we will more likely get an error raised in our effect type.

Instead of creating a `Command[Long ~ String]` we could model it using a `case class`.

```
case class Country(id: Long, name: String)
```

It lets us maintain our model as our database evolves. Skunk lets us generically derive a `Codec`, which is both a `Decoder` and an `Encoder`, as demonstrated below.

```
val codec: Codec[Country] =
  (int8 ~ varchar).gimap[Country]
```

The method `gimap` is a version of `imap` that maps out to a product type based on a shapeless generic, hence the `g`. Or we could also do it manually.

```
val codec: Codec[Country] =
  (int8 ~ varchar).imap {
    case i ~ n => Country(i, n)
  }(c => c.id ~ c.name)
```

Having this `Codec`, we can redefine our command as follows:

```
val insertCmd: Command[Country] =
  sql"""
      INSERT INTO country
      VALUE ($codec)
     """.command
```

All we need to do is to maintain our codecs!

Interpreters

Now that Skunk has been introduced, let's get to work. It has been mentioned that Brands, Categories, Items, Orders, and Users will be persisted in PostgreSQL.

Next, let's explore in detail how we can define such interpreters.

Brands

Let's recap on what its algebra looks like.

```
trait Brands[F[_]] {
  def findAll: F[List[Brand]]
  def create(name: BrandName): F[Unit]
}
```

First of all, we need to define the Postgres table, or also called *schema definition*. We are going to call it `brands`.

```
CREATE TABLE brands (
  uuid UUID PRIMARY KEY,
  name VARCHAR UNIQUE NOT NULL
);
```

Once we have the schema, we need to define the codecs, queries, and commands. A good practice is to define them in a private object in the same file. I like to add the suffix *Queries*, even though it also has codecs and commands.

Let's start explaining codecs, which are going to be defined within a `private object BrandQueries`.

```
val codec: Codec[Brand] =
  (uuid.cimap[BrandId] ~ varchar.cimap[BrandName]).imap {
    case i ~ n => Brand(i, n)
  }(b => b.uuid ~ b.name)
```

Here we use a custom extension method `cimap`, defined as follows:

```
implicit class CodecOps[B](codec: Codec[B]) {
  def cimap[A: Coercible[B, *]]: Codec[A] =
    codec.imap(_.coerce[A])(_.repr.asInstanceOf[B])
}
```

We need to perform a (safe) type-cast as we did with our JSON codecs, due to Scala's type system's limitation.

We can also choose to use predefined codecs and construct each value manually.

```
val codec: Codec[Brand] =
  (uuid ~ varchar).imap {
    case i ~ n =>
      Brand(
        BrandId(i),
        BrandName(n)
```

```
      )
  }(b => b.uuid.value ~ b.name.value)
```

Whichever method to use is up to the user. Next are a query and a command, also defined within `BrandQueries`.

```
val selectAll: Query[Void, Brand] =
  sql"""
      SELECT * FROM brands
     """.query(codec)

val insertBrand: Command[Brand] =
  sql"""
      INSERT INTO brands
      VALUES ($codec)
      """.command
```

In order to run these queries and commands, we need a session pool of type `Resource[F, Session[F]]`, so this will be the argument of the interpreter.

```
final class LiveBrands[F[_]: BracketThrow: GenUUID] private (
    sessionPool: Resource[F, Session[F]]
) extends Brands[F] {
  import BrandQueries._

  def findAll: F[List[Brand]] =
    sessionPool.use(_.execute(selectAll))

  def create(name: BrandName): F[Unit] =
    sessionPool.use { session =>
      session.prepare(insertBrand).use { cmd =>
        GenUUID[F].make[BrandId].flatMap { id =>
          cmd.execute(Brand(id, name)).void
        }
      }
    }
}
```

`BracketThrow[F[_]]` is just a type alias for `Bracket[F, Throwable]`. We use it to improve readability when using context bound constraints. Note that you can also use kind-projector's syntax to make it work without a type alias, but it is a bit more verbose.

```
def foo[F[_]: Bracket[*[_], Throwable]]
```

In the `findAll` query, we access the `Session[F]` of the pool by calling the `use` method, and then call the `execute` method passing our previously defined query as a parameter.

```
def findAll: F[List[Brand]] =
  sessionPool.use(_.execute(selectAll))
```

We can use `execute` because there are no inputs to our query, indicated by its first type `Void`. It intentionally returns a `List[Brand]` because we are assuming that the number of brands in our database is considerably small, so it can all fit into memory. Later in this chapter, we are going to see how we can deal with large records that do not fit into memory.

In the `create` method, we use a prepared statement. Once we access the `Session[F]`, we call the `prepare` method passing the insert command as a parameter, which returns a `Resource[F, PreparedCommand[F, Brand]]`.

```
session.prepare(insertBrand).use { cmd =>
  GenUUID[F].make[BrandId].flatMap { id =>
    cmd.execute(Brand(id, name)).void
  }
}
```

Next, we call `use` on this resource, call `execute` on our prepared command (passing a `Brand` as an argument), and finally call `void` to ignore its result.

Besides, there is a new algebra making an appearance.

- `GenUUID` lets us create and read coercible `UUID`s

Ultimately, as we have learned in Chapter 2, we define a smart constructor for the interpreter.

```
object LiveBrands {
  def make[F[_]: Sync](
      sessionPool: Resource[F, Session[F]]
  ): F[Brands[F]] =
    Sync[F].delay(
      new LiveBrands[F](sessionPool)
    )
}
```

It helps us encapsulate state, as we have previously discussed.

Tips

`Creation of mutable state must be suspended in F`

If you are wondering why the creation of a simple `LiveBrands` class is being suspended in `F`, this is the reason. `LiveBrands` is an interpreter that can mutate state, which is going to be shared in the application, and *all creation of mutable state must be suspended.*

In this case, the mutable state is in Postgres; it could be in memory (e.g. in a `Ref` in our interpreter), though, and that would be easier to understand. However, both interpreters are conceptually the same in terms of state management.

Categories

Next are Categories, which are almost identical to Brands. Here is the schema definition:

```
CREATE TABLE categories (
  uuid UUID PRIMARY KEY,
  name VARCHAR UNIQUE NOT NULL
);
```

Followed by codecs, queries, and commands:

```
private object CategoryQueries {

  val codec: Codec[Category] =
    (uuid.cimap[CategoryId] ~ varchar.cimap[CategoryName]).imap {
      case i ~ n => Category(i, n)
    }(c => c.uuid ~ c.name)

  val selectAll: Query[Void, Category] =
    sql"""
        SELECT * FROM categories
       """.query(codec)

  val insertCategory: Command[Category] =
    sql"""
        INSERT INTO categories
        VALUES ($codec)
        """.command

}
```

Below is the interpreter:

```
final class LiveCategories[F[_]: BracketThrow: GenUUID] private (
    sessionPool: Resource[F, Session[F]]
) extends Categories[F] {
  import CategoryQueries._
```

```
  def findAll: F[List[Category]] =
    sessionPool.use(_.execute(selectAll))

  def create(name: CategoryName): F[Unit] =
    sessionPool.use { session =>
      session.prepare(insertCategory).use { cmd =>
        GenUUID[F].make[CategoryId].flatMap { id =>
          cmd.execute(Category(id, name)).void
        }
      }
    }

}
```

Of course, it also defines a smart constructor.

```
object LiveCategories {
  def make[F[_]: Sync](
      sessionPool: Resource[F, Session[F]]
  ): F[Categories[F]] =
    Sync[F].delay(
      new LiveCategories[F](sessionPool)
    )
}
```

Almost identical to Brands, nothing new to discuss here.

Items

This one is very interesting because it defines five different methods. Let's recap on its algebra:

```
trait Items[F[_]] {
  def findAll: F[List[Item]]
  def findBy(brand: BrandName): F[List[Item]]
  def findById(itemId: ItemId): F[Option[Item]]
  def create(item: CreateItem): F[Unit]
  def update(item: UpdateItem): F[Unit]
}
```

Let's start with the schema definition:

```
CREATE TABLE items (
  uuid UUID PRIMARY KEY,
  name VARCHAR UNIQUE NOT NULL,
  description VARCHAR NOT NULL,
  price NUMERIC NOT NULL,
  brand_id UUID NOT NULL,
  category_id UUID NOT NULL,
  CONSTRAINT brand_id_fkey FOREIGN KEY (brand_id)
    REFERENCES brands (uuid) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT cat_id_fkey FOREIGN KEY (category_id)
    REFERENCES categories (uuid) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
);
```

This one is arguably our most complex table definition, as it has foreign key constraints to other tables. Still, it should be straightforward to follow.

We are now going to define the following values within a `private object ItemQueries`; in this case, we are going to do so step by step because of its length.

The first function is `selectAll`, which joins values from three different tables.

```
val selectAll: Query[Void, Item] =
  sql"""
      SELECT i.uuid, i.name, i.description, i.price,
             b.uuid, b.name, c.uuid, c.name
      FROM items AS i
      INNER JOIN brands AS b ON i.brand_id = b.uuid
      INNER JOIN categories AS c ON i.category_id = c.uuid
     """.query(decoder)
```

We are selecting eight different columns, for which we need a `Decoder`.

```
val decoder: Decoder[Item] =
  (
   uuid ~ varchar ~ varchar ~ numeric ~
   uuid ~ varchar ~ uuid ~ varchar
  ).map {
    case i ~ n ~ d ~ p ~ bi ~ bn ~ ci ~ cn =>
      Item(
        ItemId(i),
        ItemName(n),
```

```
        ItemDescription(d),
        USD(p),
        Brand(BrandId(bi), BrandName(bn)),
        Category(CategoryId(ci), CategoryName(cn))
      )
  }
```

We could have defined a `Codec` as well, but we will see soon why we haven't.

The second function is `selectByBrand`, which takes an extra argument.

```
val selectByBrand: Query[BrandName, Item] =
  sql"""
      SELECT i.uuid, i.name, i.description, i.price,
             b.uuid, b.name, c.uuid, c.name
      FROM items AS i
      INNER JOIN brands AS b ON i.brand_id = b.uuid
      INNER JOIN categories AS c ON i.category_id = c.uuid
      WHERE b.name LIKE ${varchar.cimap[BrandName]}
    """.query(decoder)
```

See how the first type of `Query` has become `BrandName` instead of `Void`? It will be the argument of this query. We are also making use of our custom extension method `cimap`, this time in the body of the query.

The third function is similar to the previous one, but it takes an `ItemId` instead of a `BrandName`.

```
val selectById: Query[ItemId, Item] =
  sql"""
      SELECT i.uuid, i.name, i.description, i.price,
             b.uuid, b.name, c.uuid, c.name
      FROM items AS i
      INNER JOIN brands AS b ON i.brand_id = b.uuid
      INNER JOIN categories AS c ON i.category_id = c.uuid
      WHERE i.uuid = ${uuid.cimap[ItemId]}
    """.query(decoder)
```

The fourth function is `insertItem`, which is defined as a `Command`.

```
val insertItem: Command[ItemId ~ CreateItem] =
  sql"""
      INSERT INTO items
      VALUES ($uuid, $varchar, $varchar, $numeric, $uuid, $uuid)
    """.command.contramap {
    case id ~ i =>
      id.value ~ i.name.value ~ i.description.value ~
      i.price.amount ~ i.brandId.value ~ i.categoryId.value
  }
```

We could have defined the encoding function as a separate function of type `Encoder[ItemId ~ CreateItem]`, though, it is done this way to demonstrate the use of the `contramap` function. In any case, this `Encoder` would only be used here so it makes sense to *inline* it.

The last function is `updateItem`, also defined as a `Command`.

```
val updateItem: Command[UpdateItem] =
  sql"""
      UPDATE items
      SET price = $numeric
      WHERE uuid = ${uuid.cimap[ItemId]}
    """.command.contramap(i => i.price.amount ~ i.id)
```

This one is fairly simple as we only need to update the price of a specific item.

Finally, here is the Items interpreter, presented without much introduction:

```
final class LiveItems[F[_]: Sync] private (
    sessionPool: Resource[F, Session[F]]
) extends Items[F] {
  import ItemQueries._

  def findAll: F[List[Item]] =
    sessionPool.use(_.execute(selectAll))

  def findBy(brand: BrandName): F[List[Item]] =
    sessionPool.use { session =>
```

```
      session.prepare(selectByBrand).use { ps =>
        ps.stream(brand, 1024).compile.toList
      }
    }

  def findById(itemId: ItemId): F[Option[Item]] =
    sessionPool.use { session =>
      session.prepare(selectById).use { ps =>
        ps.option(itemId)
      }
    }

  def create(item: CreateItem): F[Unit] =
    sessionPool.use { session =>
      session.prepare(insertItem).use { cmd =>
        GenUUID[F].make[ItemId].flatMap { id =>
          cmd.execute(id ~ item).void
        }
      }
    }

  def update(item: UpdateItem): F[Unit] =
    sessionPool.use { session =>
      session.prepare(updateItem).use { cmd =>
        cmd.execute(item).void
      }
    }

}
```

Let's focus on the `findBy` and `findById` methods, which are distinct from our previous examples.

```
session.prepare(selectByBrand).use { ps =>
  ps.stream(brand, 1024).compile.toList
}
```

This is how we execute queries that have arguments. We use a *prepared query*, which in this case returns a `Resource[F, PreparedQuery[F, BrandName]]` (similar to a prepared command). Once we access the resource, we call the `stream` method, which returns an Fs2's `Stream[F, Item]`, supplying a brand name and a *chunk size*. Yet, we want to return a `List[Item]`, so we call `compile.toList`, which is an effectful operation.

Streaming & Pagination

The avid reader might have noticed that items could possibly not fit into memory, so forcing this stream into a list might not be a wise decision. We have a few options here:

1. Change the algebra's return type from `F[List[Item]]` to `Stream[F, Item]`. In this case, the implementation would become:

```
def findBy(brand: BrandName): Stream[F, Item] =
  for {
    sn <- Stream.resource(sessionPool)
    ps <- Stream.resource(sn.prepare(selectByBrand))
    rs <- ps.stream(brand, 1024)
  } yield rs
```

We could paginate the results before returning the HTTP response, or we could return the stream directly. Http4s supports streams out of the box (it returns a chunked transfer encoding[8] response). Try it out yourself.

2. Keep the original return type `F[List[Item]]` but limit the amount of results. We can easily achieve this by receiving a `limit` argument and writing the according SQL (e.g. `LIMIT 10`).
3. Change the algebra's return type from `F[List[Item]]` to a custom type `F[PaginatedItems]`, which contains the current list of items, and a flag indicating whether there are more items or not. It requires some extra amount of work, but it is doable using *cursor*s, provided by Skunk. Instead of calling `ps.stream`, we can call `ps.cursor`, which gives us a `Resource[F, Cursor[F, Item]]`.

A `Cursor` gives us with the following method:

```
def fetch(maxRows: Int): F[(List[A], Boolean)]
```

You can already imagine how to implement it, right? Readers are encouraged to try and solve it as an exercise.

Next is Orders. Here is the schema definition:

[8] https://en.wikipedia.org/wiki/Chunked_transfer_encoding

```
CREATE TABLE orders (
  uuid UUID PRIMARY KEY,
  user_id UUID NOT NULL,
  payment_id UUID UNIQUE NOT NULL,
  items JSONB NOT NULL,
  total NUMERIC,
  CONSTRAINT user_id_fkey FOREIGN KEY (user_id)
    REFERENCES users (uuid) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
);
```

In addition to the `user_id` foreign key, we can see how `items` are going to be represented using the native `JSONB` type.

Here is the `Decoder` (again, not a `Codec`) for `Order`, defined within the `private object OrderQueries`:

```
val decoder: Decoder[Order] =
  (
    uuid.cimap[OrderId] ~ uuid ~ uuid.cimap[PaymentId] ~
    jsonb[Map[ItemId, Quantity]] ~ numeric.map(USD.apply)
  ).map {
      case o ~ _ ~ p ~ i ~ t =>
        Order(o, p, i, t)
    }
```

We are using a new codec `jsonb`, which is backed by the Circe library. It takes a type parameter `A`, and it requires instances of both `io.circe.Encoder` and `io.circe.Decoder` to be in scope for `A`. To use this codec, you need to add the extra dependency `skunk-circe` and have `import skunk.circe.codec.all._` in scope.

Next are the queries:

```
val selectByUserId: Query[UserId, Order] =
  sql"""
      SELECT * FROM orders
      WHERE user_id = ${uuid.cimap[UserId]}
     """.query(decoder)

val selectByUserIdAndOrderId: Query[UserId ~ OrderId, Order] =
  sql"""
      SELECT * FROM orders
      WHERE user_id = ${uuid.cimap[UserId]}
      AND uuid = ${uuid.cimap[OrderId]}
     """.query(decoder)
```

Followed by a single command and its encoder:

```
val encoder: Encoder[UserId ~ Order] =
  (
    uuid.cimap[OrderId] ~ uuid.cimap[UserId] ~
    uuid.cimap[PaymentId] ~ jsonb[Map[ItemId, Quantity]] ~
    numeric.contramap[Money](_.amount)
  ).contramap {
      case id ~ o =>
        o.id ~ id ~ o.paymentId ~ o.items ~ o.total
    }

val insertOrder: Command[UserId ~ Order] =
  sql"""
      INSERT INTO orders
      VALUES ($encoder)
     """.command
```

You can see why we haven't defined a `Codec[Order]`; because creating a new order also takes a `UserId`, hence our `Encoder[UserId ~ Order]`.

Next is the Orders interpreter:

```
private class LiveOrders[F[_]: Sync](
    sessionPool: Resource[F, Session[F]]
) extends Orders[F] {
  import OrderQueries._

  def get(userId: UserId, orderId: OrderId): F[Option[Order]] =
    sessionPool.use { session =>
      session.prepare(selectByUserIdAndOrderId).use { q =>
        q.option(userId ~ orderId)
      }
    }

  def findBy(userId: UserId): F[List[Order]] =
    sessionPool.use { session =>
      session.prepare(selectByUserId).use { q =>
        q.stream(userId, 1024).compile.toList
      }
    }

  def create(
      userId: UserId,
      paymentId: PaymentId,
```

```
      items: List[CartItem],
      total: Money
  ): F[OrderId] =
    sessionPool.use { session =>
      session.prepare(insertOrder).use { cmd =>
        GenUUID[F].make[OrderId].flatMap { id =>
          val itMap = items.map(x => x.item.uuid -> x.quantity).toMap
          val order = Order(id, paymentId, itMap, total)
          cmd.execute(userId ~ order).as(id)
        }
      }
    }

}
```

There is nothing we haven't seen before. Finally, a smart constructor, as usual:

```
object LiveOrders {
  def make[F[_]: Sync](
      sessionPool: Resource[F, Session[F]]
  ): F[Orders[F]] =
    Sync[F].delay(
      new LiveOrders[F](sessionPool)
    )
}
```

Next is Users. Below is the schema definition:

```
CREATE TABLE users (
  uuid UUID PRIMARY KEY,
  name VARCHAR UNIQUE NOT NULL,
  password VARCHAR NOT NULL
);
```

Let's now recap on its algebra:

```
trait Users[F[_]] {
  def find(
    username: UserName,
    password: Password
  ): F[Option[User]]
  def create(
```

```
    username: UserName,
    password: Password
  ): F[UserId]
}
```

Both functions take a `Password` value that represents plain text. However, we need to encrypt passwords before they are persisted. For this purpose, we introduce a new datatype `EncryptedPassword`. Let's look at `UserQueries`.

```
private object UserQueries {

  val codec: Codec[User ~ EncryptedPassword] =
    (
      uuid.cimap[UserId] ~ varchar.cimap[UserName] ~
      varchar.cimap[EncryptedPassword]
    ).imap {
        case i ~ n ~ p =>
          User(i, n) ~ p
      } {
        case u ~ p =>
          u.id ~ u.name ~ p
      }

  val selectUser: Query[UserName, User ~ EncryptedPassword] =
    sql"""
        SELECT * FROM users
        WHERE name = ${varchar.cimap[UserName]}
       """.query(codec)

  val insertUser: Command[User ~ EncryptedPassword] =
    sql"""
        INSERT INTO users
        VALUES ($codec)
        """.command

}
```

Both `selectUser` and `insertUser` take an `EncryptedPassword` instead of the normal `Password`. This means that the interpreter should be responsible for encrypting passwords.

```
final class LiveUsers[F[_]: BracketThrow: GenUUID] private (
    sessionPool: Resource[F, Session[F]],
    crypto: Crypto
) extends Users[F] {
```

```
  import UserQueries._

  def find(
      username: UserName,
      password: Password
  ): F[Option[User]] =
    sessionPool.use { session =>
      session.prepare(selectUser).use { q =>
        q.option(username).map {
          case Some(u ~ p)
            if p.value == crypto.encrypt(password).value =>
              u.some
          case _ => none[User]
        }
      }
    }

  def create(
      username: UserName,
      password: Password
  ): F[UserId] =
    sessionPool.use { session =>
      session.prepare(insertUser).use { cmd =>
        GenUUID[F].make[UserId].flatMap { id =>
          cmd
            .execute(User(id, username) ~ crypto.encrypt(password))
            .as(id)
            .handleErrorWith {
              case SqlState.UniqueViolation(_) =>
                UserNameInUse(username).raiseError[F, UserId]
            }
        }
      }
    }

}
```

There is a new argument in the interpreter: `Crypto`. It is an interface defined as follows:

```
trait Crypto {
  def encrypt(value: Password): EncryptedPassword
  def decrypt(value: EncryptedPassword): Password
}
```

Its implementation is not relevant here so we are not going to see it (please refer to

the source code for more). All we need to know is that we can encrypt and decrypt passwords.

Now let's look at the first function, `find`. We use `q.option`, another function on `PreparedQuery`, which expects exactly zero or one result; otherwise, it raises an error. Next, we pattern match and compare the encrypted password. If they match, we return the user; otherwise, we return `None`.

The second function, `create`, does a few things. It:

- creates a new `UserId`.
- encrypts the password.
- executes the command.
- handles the possible `UniqueViolation` SQL error (this could happen if the username already exists in our database).

Redis for Cats

Redis4Cats[9] is a purely functional and asynchronous Redis client built on top of Cats Effect, Fs2, and Java's Lettuce.

Quoting the official Redis website[10]:

> Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

We will be using Redis to store authentication tokens and shopping carts. It seems a perfect fit since we need to set an expiration time for both, and Redis supports this feature natively.

Connection

By now, it shouldn't come as a surprise that a Redis connection is treated as a resource. See below how this is done with Redis for Cats:

```
val mkRedisResource: Resource[F, RedisCommands[F, String, String]] =
  for {
    uri <- Resource.liftF(RedisURI.make[F]("redis://localhost"))
    cli <- RedisClient[F](uri)
    cmd <- Redis[F, String, String](cli, RedisCodec.Utf8)
  } yield cmd
```

`RedisCommands[F, K, V]` is an interface from which we can access all the commands available. Both `K` and `V` are the types of keys and values, respectively, making it type-safe. We cannot increment values of type `String`, for example.

We can acquire as many `RedisCommands` as we need. In our application, we will need a single one of types `String`, as in the example above.

[9] https://github.com/profunktor/redis4cats
[10] https://redis.io/

Interpreters

There is not much ceremony in getting started with Redis for Cats. Once we acquire a `RedisCommands` instance, we are ready to make use of it.

Shopping Cart

Let's first have a look at the structure of the Shopping Cart interpreter and later analyze its functions in detail:

```
final class LiveShoppingCart[F[_]: GenUUID: MonadThrow] private (
    items: Items[F],
    redis: RedisCommands[F, String, String],
    exp: ShoppingCartExpiration
) extends ShoppingCart[F] { ... }
```

In addition to `RedisCommands`, it takes an `Items[F]` and a `ShoppingCartExpiration` (a newtype wrapping a `FiniteDuration`). Before we start analyzing each function, let's see what kind of data structure we are going to use for the cart.

We will use hashes[11], which has the format KEY FIELD VALUE. For example:

```
redis> HSET myhash field1 "Hello"
(integer) 1
redis> HGET myhash field1
"Hello"
redis>
```

Let's now get started with the first function `add`:

```
def add(
    userId: UserId,
    itemId: ItemId,
    quantity: Quantity
): F[Unit] =
  redis.hSet(
    userId.value.toString,
    itemId.value.toString,
    quantity.value.toString
  ) *>
    redis.expire(
      userId.value.toString,
```

[11] https://redis.io/commands#hash

```
      exp.value
    )
```

It adds an item (field) and a quantity (value) to the `userId` key, and it sets the expiration time of the shopping cart for the user.

Next is `get`, which does a little bit more:

```
def get(userId: UserId): F[CartTotal] =
  redis.hGetAll(userId.value.toString).flatMap { it =>
    it.toList
      .traverseFilter {
        case (k, v) =>
          for {
            id <- GenUUID[F].read[ItemId](k)
            qt <- ApThrow[F].catchNonFatal(Quantity(v.toInt))
            rs <- items.findById(id).map(
                    _.map(i => CartItem(i, qt))
                  )
          } yield rs
      }
      .map(items => CartTotal(items, calcTotal(items)))
  }

private def calcTotal(items: List[CartItem]): Money =
  USD(
    items
      .foldMap { i =>
        i.item.price.value * i.quantity.value
      }
  )
```

It tries to find the shopping cart for the user using the `hGetAll` function, which returns a `Map[String, String]`, or a `Map[K, V]`, generically speaking.

If it exists, it parses both fields and values into a `List[CartItem]` and finally, it calculates the total amount. `GenUUID[F].read` takes a `String` and returns an `F[A]`, where `A: Coercible[UUID, *]]`. In this case, it is `ItemId`.

Next is `delete`, which simply deletes the shopping cart for the user:

```
def delete(userId: UserId): F[Unit] =
  redis.del(userId.value.toString)
```

Followed by `removeItem`, which removes a specific item from the shopping cart:

```
def removeItem(userId: UserId, itemId: ItemId): F[Unit] =
  redis.hDel(userId.value.toString, itemId.value.toString)
```

Finally, the `update` function:

```
def update(userId: UserId, cart: Cart): F[Unit] =
  redis.hGetAll(userId.value.toString).flatMap { it =>
    it.toList.traverse_ {
      case (k, _) =>
        GenUUID[F].read[ItemId](k).flatMap { id =>
          cart.items.get(id).traverse_ { q =>
            redis.hSet(userId.value.toString, k, q.value.toString)
          }
        }
    } *>
      redis.expire(userId.value.toString, exp.value)
  }
```

It retrieves the shopping cart for the user (if it exists) and it updates the quantity of each matching item, followed by updating the shopping cart expiration.

Authentication

It has been previously noted that the `Auth` algebra might need to be changed, and inevitably, it is going to happen. We are going to define two different algebras; the first one, `Auth`, is the most generic one:

```
trait Auth[F[_]] {
  def newUser(username: UserName, password: Password): F[JwtToken]
  def login(username: UserName, password: Password): F[JwtToken]
  def logout(token: JwtToken, username: UserName): F[Unit]
}
```

Our second algebra is specialized in retrieving a specific kind of user, indicated by its second type parameter `A`:

```
trait UsersAuth[F[_], A] {
  def findUser(token: JwtToken)(claim: JwtClaim): F[Option[A]]
}
```

The `findUser` function is curried to make the integration with Http4s JWT Auth much easier. Remember that the `authenticate` function from `JWTAuthMiddleware` has the shape `JwtToken => JwtClaim => F[Option[A]]`.

Next, let's have a look at the interpreter for `CommonUser`:

```
class LiveUsersAuth[F[_]: Functor](
    redis: RedisCommands[F, String, String]
) extends UsersAuth[F, CommonUser] {

  def findUser(token: JwtToken)
              (claim: JwtClaim): F[Option[CommonUser]] =
    redis
      .get(token.value)
      .map(_.flatMap { u =>
        decode[User](u).toOption.map(CommonUser.apply)
      })

}
```

Our function tries to find the user by token in Redis, and if there is a result, it tries to decode the JSON as the desired `User` type. A token is persisted as a simple key, with its value being the serialized user in JSON format.

Next, we have an interpreter for `AdminUser`:

```
class LiveAdminAuth[F[_]: Applicative](
    adminToken: JwtToken,
    adminUser: AdminUser
) extends UsersAuth[F, AdminUser] {

  def findUser(token: JwtToken)
              (claim: JwtClaim): F[Option[AdminUser]] =
    (token == adminToken)
      .guard[Option]
      .as(adminUser)
      .pure[F]

}
```

It compares the token with the unique Admin Token that has been passed to the interpreter on initialization (more on this in Chapter 8), and in case of match, it returns the `adminUser` stored in memory (remember that there is a unique Admin User).

Ultimately, we define smart constructors for both interpreters:

```
object LiveUsersAuth {
  def make[F[_]: Sync](
      redis: RedisCommands[F, String, String]
  ): F[UsersAuth[F, CommonUser]] =
    Sync[F].delay(
      new LiveUsersAuth(redis)
```

```
    )
}

object LiveAdminAuth {
  def make[F[_]: Sync](
      adminToken: JwtToken,
      adminUser: AdminUser
  ): F[UsersAuth[F, AdminUser]] =
    Sync[F].delay(
      new LiveAdminAuth(adminToken, adminUser)
    )
}
```

Let's now see what the structure of the `Auth` interpreter looks like and then analyze each function step by step:

```
final class LiveAuth[F[_]: GenUUID: MonadThrow] private (
    tokenExpiration: TokenExpiration,
    tokens: Tokens[F],
    users: Users[F],
    redis: RedisCommands[F, String, String]
) extends Auth[F] {

  private val TokenExpiration = tokenExpiration.value

  // .... functions go here ....

}
```

Its constructor takes four different arguments:

- `TokenExpiration` is a newtype that wraps a `FiniteDuration`.
- `Tokens[F]` allows us to create new JWT tokens.

```
trait Tokens[F[_]] {
  def create: F[JwtToken]
}
```

- `Users[F]` allows us to find and create new users (defined in Chapter 4).
- `RedisCommands` is self-explanatory.

Our first function is `newUser`:

```
def newUser(username: UserName, password: Password): F[JwtToken] =
  users.find(username, password).flatMap {
    case Some(_) => UserNameInUse(username).raiseError[F, JwtToken]
```

```
    case None =>
      for {
        i <- users.create(username, password)
        t <- tokens.create
        u = User(i, username).asJson.noSpaces
        _ <- redis.setEx(t.value, u, TokenExpiration)
        _ <- redis.setEx(username.value, t.value, TokenExpiration)
      } yield t
  }
```

Here we try to find the user in Postgres. If it doesn't exist, we proceed with its creation; otherwise, we raise a `UserNameInUse` error.

Creating a user means persisting it in Postgres, creating a JWT token, serializing the user as JSON, and persisting both the token and the serialized user in Redis for fast access, indicating an expiration time.

Our `login` function comes next:

```
def login(username: UserName, password: Password): F[JwtToken] =
  users.find(username, password).flatMap {
    case None =>
      InvalidUserOrPassword(username).raiseError[F, JwtToken]
    case Some(user) =>
      redis.get(username.value).flatMap {
        case Some(t) => JwtToken(t).pure[F]
        case None =>
          tokens.create.flatTap { t =>
            redis.setEx(
              t.value, user.asJson.noSpaces, TokenExpiration
            ) *>
              redis.setEx(username.value, t.value, TokenExpiration)
          }
      }
  }
```

We try to find the user in Postgres. If it doesn't exist, we simply raise an error; if it exists, we search for the token by user in Redis (in case the user has already been logged in). If we get a token, we return it; otherwise, we create a new token and persist both the user and the token with an expiration time.

When we get an existing token, we could also extend its lifetime, i.e. updating its expiration. However, this will only make sense when the expiration time of our JWTs is greater than the one we configure for our tokens stored in Redis.

Lastly, we have the `logout` function, which is the simplest:

```
def logout(
    token: JwtToken,
    username: UserName
): F[Unit] =
  redis.del(token.value) *>
    redis.del(username.value)
```

All it does is deleting the token and the user from Redis, if any.

As usual, there is a smart constructor for the interpreter:

```
object LiveAuth {
  def make[F[_]: Sync](
      tokenExpiration: TokenExpiration,
      tokens: Tokens[F],
      users: Users[F],
      redis: RedisCommands[F, String, String]
  ): F[Auth[F]] =
    Sync[F].delay(
      new LiveAuth(tokenExpiration, tokens, users, redis)
    )
}
```

Blocking operations

We have learned about Skunk and Redis4Cats, which are both asynchronous, so we didn't have to deal with blocking operations. However, it is very common in the database world to deal with such cases.

For this purpose, Cats Effect provides a `Blocker` datatype that merely wraps an `ExecutionContext`. Most functional libraries that need to deal with blocking operations would take a `Blocker` instead of an implicit `ExecutionContext` such as `global`, which would affect the performance of our application.

For example, if we were using Doobie instead of Skunk, this is how we would acquire a connection:

```
val transactor: Resource[IO, HikariTransactor[IO]] =
  for {
    ce <- ExecutionContexts.fixedThreadPool[IO](10) // connect EC
    be <- Blocker[IO]                               // blocking EC
    xa <- HikariTransactor.newHikariTransactor[IO](
            "org.h2.Driver",                        // driver
            "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1", // connect URL
            "sa",                                   // username
            "",                                   // password
            ce,                                 // await connection here
            be                                // exec JDBC ops here
          )
  } yield xa
```

Health check

Last but not least, our application needs to report its health via HTTP. We will start defining the algebra:

```
trait HealthCheck[F[_]] {
  def status: F[AppStatus]
}
```

Where `AppStatus` is defined as follows:

```
@newtype case class RedisStatus(value: Boolean)
@newtype case class PostgresStatus(value: Boolean)

case class AppStatus(
    redis: RedisStatus,
    postgres: PostgresStatus
)
```

This will indicate the connection status of both Redis and Postgres. If the status is OK, it would be `true`; otherwise, `false`. Let's have a look at its interpreter:

```
final class LiveHealthCheck[
    F[_]: Concurrent: Parallel: Timer
] private (
    sessionPool: Resource[F, Session[F]],
    redis: RedisCommands[F, String, String]
) extends HealthCheck[F] {

  val q: Query[Void, Int] =
    sql"SELECT pid FROM pg_stat_activity".query(int4)

  val redisHealth: F[RedisStatus] =
    redis.ping
      .map(_.nonEmpty)
      .timeout(1.second)
      .orElse(false.pure[F])
      .map(RedisStatus.apply)

  val postgresHealth: F[PostgresStatus] =
    sessionPool
      .use(_.execute(q))
      .map(_.nonEmpty)
      .timeout(1.second)
      .orElse(false.pure[F])
```

```
      .map(PostgresStatus.apply)

  val status: F[AppStatus] =
    (redisHealth, postgresHealth).parMapN(AppStatus)

}
```

For Redis, we simply `ping` the server; for Postgres, we make a simple query. Both actions have a timeout of one second and are performed in parallel, using the `parMapN` function.

In both cases, if anything goes wrong (e.g. cannot connect to server), we return `false` using the `orElse` function from `ApplicativeError`.

Lastly, we define a smart constructor:

```
object LiveHealthCheck {
  def make[F[_]: Concurrent: Parallel: Timer](
      sessionPool: Resource[F, Session[F]],
      redis: RedisCommands[F, String, String]
  ): F[HealthCheck[F]] =
    Sync[F].delay(
      new LiveHealthCheck[F](sessionPool, redis)
    )
}
```

Having the health check functionality in place, all is left is to define the HTTP routes:

```
final class HealthRoutes[F[_]: Sync](
    healthCheck: HealthCheck[F]
) extends Http4sDsl[F] {

  private[routes] val prefixPath = "/healthcheck"

  private val httpRoutes: HttpRoutes[F] =
    HttpRoutes.of[F] {
      case GET -> Root =>
        Ok(healthCheck.status)
    }

  val routes: HttpRoutes[F] = Router(
    prefixPath -> httpRoutes
  )

}
```

A simple HTTP service reporting the status of our application.

Below is the expected *Response Body*:

```
{
  "redis": true,
  "postgres": true
}
```

Or `false` if there is something wrong with either connection.

Our health check endpoint always returns a successful response (200 OK). However, we can argue the design can be changed to return other response status codes if something is not healthy. There are many valid designs in this space, but we are going to stick with this simple one.

Chapter 7: Testing

Tests are as significant as types. While the latter prevent us from writing programs that wouldn't compile, they are not sufficient to leave all the incorrect programs out. This is the small scope where tests are useful, even though not 100% bulletproof.

There are different kinds of tests. We will be focusing on *unit tests* and *integration tests*, and see how both can be more than adequate for a business application such as the Shopping Cart we are building.

We will also make use of *property-based testing*[1] by generating random data.

[1] http://www.scalatest.org/user_guide/property_based_testing

Functional test suite

In the Scala ecosystem, most of the popular test frameworks don't operate well with purely functional libraries such as Cats Effect. They are mostly side-effectful, forcing us into an undesirable imperative road.

However, in the same way `IOApp` exists to embrace pureness all the way, we can come up with our purely functional test suite in a few lines of code, building atop of an existing testing framework. We will see it doesn't take too much, and that we can reuse it in other projects as well.

I am all for simplicity, and I don't like DSL-based test frameworks, so this is all I think a purely functional test suite should have. The ability to:

- express pure programs based on `IO`.
- run some assertions.
- run asynchronously.

In our case, we are going to be testing our programs in `IO` (more on this soon), so we need the equivalent of `IOApp`. Let's see how this would look like on top of the Scalatest framework.

```
trait PureTestSuite
  extends AsyncFunSuite
  with ScalaCheckDrivenPropertyChecks
  with CatsEquality {

  implicit val cs: ContextShift[IO] =
    IO.contextShift(ExecutionContext.global)

  implicit val timer: Timer[IO] =
    IO.timer(ExecutionContext.global)

  private def mkUnique(name: String): String =
    s"$name - ${UUID.randomUUID}"

  def spec(testName: String)
          (f: => IO[Assertion])
          (implicit pos: Position): Unit =
    test(mkUnique(testName))(IO.suspend(f).unsafeToFuture())

}
```

A few design decisions:

- `AsyncFunSuite` is used for a simple asynchronous `test("my test")(assertion)` function.
- `ScalaCheckDrivenPropertyChecks` lets us use property-based testing.
- `CatsEquality` is a workaround that lets us use Cats equality in our tests.
- A default `ContextShift[IO]` and `Timer[IO]` are provided (needed for Cats Retry and Skunk).
- A `spec` function is given as the more pure equivalent of `test`, which takes an `IO[Assertion]` instead of a `Future[Assertion]`.
- Our `f` must be a *by-name* argument that can be suspended. This is essential in side-effects land.
- Test names are made unique by appending a randomly generated `UUID` to them.

Considering we want to run our tests asynchronously, we need to give them a unique name when combined with property-based testing because of a limitation with the integration between Scalatest and Scalacheck. Otherwise, names wouldn't be necessary when only using Scalacheck's `forAll`, the function we will explore soon.

Warning

`Scalatest doesn't support asynchronous property-based testing`

Why testing using `IO`?

Why not? We can easily compose programs in `IO` as we can with other monads. If you think about it, there is no difference in using `Id`, `Either`, or `IO` for testing, even if what you are testing has a single constraint `Monad[F]`. You can think of `IO` as another interpreter of the typeclass constraints our functions may have, a concrete implementation.

Tests should be seen as `main`, another "end of the world", where we choose an effect type. Testing in `IO` is *perfectly fine*, and sometimes necessary (e.g. in the presence of concurrency).

Generators

Scalatest is well integrated with Scalacheck, providing the interface named `ScalaCheckDrivenPropertyChecks`.

The property-based tests we are going to use have the following shape:

```
forAll { (a: A, b: B) => ... }
```

This kind of tests require instances of `org.scalacheck.Arbitrary` for every value we intend to generate. In this case, instances for both `A` and `B`.

`Arbitrary` is a typeclass that wraps a generator, or a user-defined function to generate a particular value. Generators are represented using the `org.scalacheck.Gen` type. The most common way of creating an `Arbitrary` instance is by using its apply method.

```
val fooGen: Gen[Foo] = ???

implicit val arbInstance: Arbitrary[Foo] =
  Arbitrary(fooGen)
```

Generators have multiple useful methods for generating constrained random data.

```
$ Gen.posNum[Int]                    // positive number
$ Gen.alphaStr                       // string
$ Gen.nonEmptyListOf(Gen.alphaNumStr) // non-empty list of strings
$ Gen.choose[Int](0, 10)             // pick a number within a range
$ Gen.oneOf(List(1,3,5))             // pick one of the list
```

Using these functions, we can define generators for our custom data.

Let's start with defining generators for `Coercible[UUID, *]` types such as `ItemId`, `PaymentId`, and `OrderId`:

```
implicit def arbCoercibleUUID[A: Coercible[UUID, *]]: Arbitrary[A] =
  Arbitrary(cbUuid[A])

def cbUuid[A: Coercible[UUID, *]]: Gen[A] =
  Gen.uuid.map(_.coerce[A])
```

This one is fairly easy since Scalacheck comes with a predefined generator for UUIDs.

Generators and arbitraries will be defined in separate files, both under a new module `tests`. This will allow us to reuse them for property-based testing in both our unit and integration tests.

Next is `CartTotal`, which requires an `Item` generator, which requires both `Brand` and `Category` generators. So let's split it into two parts, defining its dependencies first.

```
val brandGen: Gen[Brand] =
  for {
    i <- cbUuid[BrandId]
    n <- cbStr[BrandName]
  } yield Brand(i, n)

val categoryGen: Gen[Category] =
  for {
    i <- cbUuid[CategoryId]
    n <- cbStr[CategoryName]
  } yield Category(i, n)

val genMoney: Gen[Money] =
  Gen.posNum[Long].map(n =>
    USD(BigDecimal(n))
  )

val itemGen: Gen[Item] =
  for {
    i <- cbUuid[ItemId]
    n <- cbStr[ItemName]
    d <- cbStr[ItemDescription]
    p <- genMoney
    b <- brandGen
    c <- categoryGen
  } yield Item(i, n, d, p, b, c)
```

Now we can proceed with `CartItem`, followed by `CartTotal`:

```
val cartItemGen: Gen[CartItem] =
  for {
    i <- itemGen
    q <- cbInt[Quantity]
  } yield CartItem(i, q)

val cartTotalGen: Gen[CartTotal] =
  for {
    i <- Gen.nonEmptyListOf(cartItemGen)
    t <- genMoney
  } yield CartTotal(i, t)
```

All the functions prefixed with `cb` mean that they are `Coercible` to the specified type parameter.

```
def cbStr[A: Coercible[String, *]]: Gen[A] =
  genNonEmptyString.map(_.coerce[A])

def cbInt[A: Coercible[Int, *]]: Gen[A] =
  Gen.posNum[Int].map(_.coerce[A])
```

When generating strings, we want to make sure they are non-empty. We could do this in the following way:

```
val genNonEmptyString: Gen[String] =
  Gen.alphaStr.suchThat(_.nonEmpty)
```

However, this is considerably slow, and we need our tests to run fast. To solve this issue, we can use the following trick (numbers 10-30 are picked arbitrarily):

```
val genNonEmptyString: Gen[String] =
  Gen
    .chooseNum(10, 30)
    .flatMap { n =>
      Gen.buildableOfN[String, Char](n, Gen.alphaChar)
    }
```

Lastly, here is the `Arbitrary` instance:

```
implicit val arbCartTotal: Arbitrary[CartTotal] =
  Arbitrary(cartTotalGen)
```

Next is `Card`, which is somewhat distinctive since it is composed of refinement types:

```
implicit val arbCard: Arbitrary[Card] =
  Arbitrary(cardGen)

val cardGen: Gen[Card] =
  for {
    n <- genNonEmptyString.map[CardNamePred](Refined.unsafeApply)
    u <- Gen.posNum[Long].map[CardNumberPred](Refined.unsafeApply)
    x <- Gen.posNum[Int].map[CardExpirationPred](x =>
           Refined.unsafeApply(x.toString)
         )
    c <- Gen.posNum[Int].map[CardCCVPred](Refined.unsafeApply)
```

```
} yield Card(
    CardName(n),
    CardNumber(u),
    CardExpiration(x),
    CardCCV(c)
  )
```

You might have to refresh your mind with the definition of the `Card` type to get this. It is noteworthy observing how we are creating random data and refining our types without any validation whatsoever. This is what the `Refined.unsafeApply` method does, and it is only right to use in tests; **in any other case, it should not be used**.

Business logic

Our main business logic resides in the `checkout` program, so this is the critical piece of software we need to test. Writing tests for all the possible scenarios is what we need to figure out.

Let's recap on its definition:

```
final class CheckoutProgram[
    F[_]: Background: Logger: MonadThrow: Timer
](
    paymentClient: PaymentClient[F],
    shoppingCart: ShoppingCart[F],
    orders: Orders[F],
    retryPolicy: RetryPolicy[F]
) {

  def checkout(
    userId: UserId,
    card: Card
  ): F[OrderId] = ???

}
```

In addition to a `RetryPolicy`, it also takes three different algebras for which we need to provide fake implementations to be able to test our program. We don't want to be hitting a real payments service, or persisting test orders in a database, for example. Although setting up a test environment with a payment service and a database is not wrong, I would say it is not exactly necessary, and we can instead get away with test interpreters. After all, what we want to test are not these components but the interaction with the main piece of logic: the `checkout` program.

Happy path

We will first define the interpreters to test the happy path. Let's start with `PaymentClient[F]`:

```
def successfulClient(pid: PaymentId): PaymentClient[IO] =
  new PaymentClient[IO] {
    def process(payment: Payment): IO[PaymentId] =
      IO.pure(pid)
  }
```

A test client that just returns the same `PaymentId` it receives as an argument.

Next is `ShoppingCart`:

```
def successfulCart(cartTotal: CartTotal): ShoppingCart[IO] =
  new TestCart {
    def get(userId: UserId): IO[CartTotal] =
      IO.pure(cartTotal)
    def delete(userId: UserId): IO[Unit] =
      IO.unit
}
```

It does nothing on `delete`, and it returns the same `CartTotal` it is given on `get`. `TestCart` is a dummy empty implementation that returns `???` on each method (we don't need the other methods for this test).

Next is `Orders[F]`:

```
def successfulOrders(oid: OrderId): Orders[IO] =
  new TestOrders {
    def create(
      userId: UserId,
      paymentId: PaymentId,
      items: List[CartItem],
      total: Money
    ): IO[OrderId] =
      IO.pure(oid)
  }
```

It returns the same `OrderId` it is given. `TestOrders` is a dummy empty implementation that returns `???` on each method (again, we don't need the other methods for this test).

Lastly, our testing retry policy:

```
val MaxRetries = 3

val retryPolicy: RetryPolicy[IO] =
  limitRetries[IO](MaxRetries)
```

The maximum number of retries is defined as a constant `MaxRetries`, so we can use it to write assertions, as we will see soon.

Now that we have defined all the test interpreters, we are ready to instantiate our checkout program and write a test for the happy path.

```
forAll {
  (
    uid: UserId, pid: PaymentId, oid: OrderId,
    ct: CartTotal, card: Card
  ) =>
    spec("successful checkout") {
      implicit val bg = shop.background.NoOp
      import shop.logger.NoOp
      new CheckoutProgram[IO](
        successfulClient(pid),
        successfulCart(ct),
        successfulOrders(oid),
        retryPolicy
      ).checkout(uid, card)
        .map { id =>
          assert(id === oid)
        }
    }
}
```

Let's break this apart and see what is happening.

First of all, we see the `forAll` function taking values of all the data we need to test our program. These values are called *generator-driven properties* because they are created using user-defined generators, as we have previously seen.

We are also using the `===` (triple equals) method from Scalactic's equality. However, using Cats' equality - via its `cats.Eq` typeclass - would be more than ideal since it provides lawful equality. The problem is that `assert` is defined as a macro, which doesn't understand Cats' equality, leaving us with cryptic error messages like the one below.

```
@ val x = 1
x: Int = 1

@ assert(Eq[Int].eqv(x, 2))
org.scalatest.exceptions.TestFailedException:
  cats.`package`.Eq.apply[Int](
```

```
    cats.implicits.catsKernelStdOrderForInt
  ).eqv(ammonite.$sess.cmd5.x, 2) was false
```

If we use universal equality (==) or Scalactic's equality (===) instead, we get a friendly error message.

```
@ assert(x === 2)
org.scalatest.exceptions.TestFailedException: 1 did not equal 2
```

> Tips
>
> Prefer not to use universal equality (==)

Fortunately, there is a way to get helpful error messages while using Cats' equality. We need to teach Scalactic's equality how to use `cats.Eq` instead of its pre-defined methods. This can be done by defining a custom instance of `org.scalactic.Equivalence[A]`, though, explaining this technique is a bit out of the scope of this book.

All we need to know is that, from now on, we get to use Cats' equality with practical assertion error messages. For the complete custom implementation, please refer to the source code of the application.

Continuing with our program, we can observe it requires implicit instances of `Background[F]` and `Logger[F]`, so all we do is to provide a few no-op implementations that serve our purpose. We are not interested in seeing what is logged or what is scheduled to run in the background for the happy path.

Here is our no-op `Background` implementation:

```
val NoOp: Background[IO] =
  new Background[IO] {
    def schedule[A](
      fa: IO[A],
      duration: FiniteDuration
    ): IO[Unit] = IO.unit
  }
```

We ignore whatever is being scheduled. In the case of `Logger`, we do something similar.

```
implicit object NoOp extends NoLogger

private[logger] class NoLogger extends Logger[IO] {
  def warn(message: => String): IO[Unit]                = IO.unit
  def warn(t: Throwable)(message: => String): IO[Unit]  = IO.unit
  def debug(t: Throwable)(message: => String): IO[Unit] = IO.unit
  def debug(message: => String): IO[Unit]               = IO.unit
  def error(t: Throwable)(message: => String): IO[Unit] = IO.unit
```

```
  def error(message: => String): IO[Unit]                = IO.unit
  def info(t: Throwable)(message: => String): IO[Unit]   = IO.unit
  def info(message: => String): IO[Unit]                 = IO.unit
  def trace(t: Throwable)(message: => String): IO[Unit] = IO.unit
  def trace(message: => String): IO[Unit]                = IO.unit
}
```

The only difference is that we expose our `NoOp` instance as an `implicit object`. However, we cannot do the same for `Background` because we are providing a default implicit instance based on `Concurrent` and `Timer` in the companion object.

So now we can focus on the real test.

```
.checkout(uid, card)
.map { id =>
  assert(id === oid)
}
```

We pass a generated `UserId` (it doesn't really matter) and a valid generated `Card`. Subsequently, we `assert` the `OrderId` returned is the same our test `Orders` interpreter returns. This is what most of our tests will look like.

We now have all the pieces we need to run our test! Open a terminal, type `sbt test`, and witness the magic.

Empty cart

This is one of the first lines of our `checkout` function:

```
shoppingCart
  .get(userId)
  .ensure(EmptyCartError)(_.items.nonEmpty)
```

If the cart is empty, we get an `EmptyCartError`, so let's write a test for this.

All we need is a test interpreter for the `ShoppingCart` that returns an empty list of items.

```
val emptyCart: ShoppingCart[IO] =
  new TestCart {
    def get(userId: UserId): IO[CartTotal] =
      IO.pure(CartTotal(List.empty, USD(0)))
  }
```

Next, we `attempt` to invoke the `checkout` function and assert on its result.

```
forAll {
  (
    uid: UserId, pid: PaymentId, oid: OrderId, card: Card
  ) =>
    spec("empty cart") {
      implicit val bg = shop.background.NoOp
      import shop.logger.NoOp
      new CheckoutProgram[IO](
        successfulClient(pid),
        emptyCart,
        successfulOrders(oid),
        retryPolicy
      ).checkout(uid, card)
        .attempt
        .map {
          case Left(EmptyCartError) =>
            assert(true)
          case _ =>
            fail("Cart was not empty as expected")
        }
    }
}
```

We expect an `EmptyCartError`; otherwise, the test fails.

Unreachable payment client

If the remote payment client is unresponsive, our system needs to be resilient. Here is where our retrying logic should be tested. First, we need to simulate an unreachable payment client. Here is a possible interpreter:

```
val unreachableClient: PaymentClient[IO] =
  new PaymentClient[IO] {
    def process(payment: Payment): IO[PaymentId] =
      IO.raiseError(PaymentError(""))
  }
```

Every time the `process` method is invoked, it raises an error. Let's see our unit test.

```
forAll {
  (
    uid: UserId, oid: OrderId,
    ct: CartTotal, card: Card
  ) =>
```

```
    spec("unreachable payment client") {
      Ref.of[IO, List[String]](List.empty).flatMap { logs =>
        implicit val bg     = shop.background.NoOp
        implicit val logger = shop.logger.acc(logs)
        new CheckoutProgram[IO](
          unreachableClient,
          successfulCart(ct),
          successfulOrders(oid),
          retryPolicy
        ).checkout(uid, card)
          .attempt
          .flatMap {
            case Left(PaymentError(_)) =>
              logs.get.map {
                case (x :: xs) =>
                  assert(
                    x.contains("Giving up") &&
                    xs.size === MaxRetries
                  )
                case _ =>
                  fail(s"Expected $MaxRetries retries")
              }
            case _ => fail("Expected payment error")
          }
      }
    }
}
```

Here we have something new:

- a `Ref[IO, List[String]]`.
- a different instance for `Logger` that takes this `Ref`.

Let's examine the implementation of the `acc` logger:

```
def acc(ref: Ref[IO, List[String]]): Logger[IO] =
  new NoLogger {
    override def error(message: => String): IO[Unit] =
      ref.update(xs => message :: xs)
  }
```

It extends the `NoLogger` implementation, overriding the `error` method to accumulate error messages in a `List[String]`.

If you recall, our retry function logs an error message every time it fails, and it logs a final error message saying "Giving up" when the maximum amount of retries is reached.

```
assert(
  x.contains("Giving up") &&
  xs.size === MaxRetries
)
```

This is precisely what we are asserting here by accessing our in-memory log.

Recovering payment client

The previous client fails every time the `process` method is invoked. So, how can we simulate a client that recovers after a certain amount of retries? We need some internal state. Let's analyze the following recovering client implementation:

```
def recoveringClient(
  attemptsSoFar: Ref[IO, Int],
  paymentId: PaymentId
): PaymentClient[IO] =
  new PaymentClient[IO] {
    def process(payment: Payment): IO[PaymentId] =
      attemptsSoFar.get.flatMap {
        case n if n === 1 =>
          IO.pure(paymentId)
        case _ =>
          attemptsSoFar.update(_ + 1) *>
            IO.raiseError(PaymentError(""))
      }
  }
```

The `Ref[IO, Int]` keeps the count of the number of retries. If it equals to one, we emit the given `PaymentId`; otherwise, we increment the counter and raise a `PaymentError` that will hit our retrying mechanism.

We will also assert on the logs. Let's look at this test implementation:

```
forAll {
  (
    uid: UserId, pid: PaymentId, oid: OrderId,
    ct: CartTotal, card: Card
  ) =>
    spec("failing payment client succeeds after one retry") {
      Ref.of[IO, List[String]](List.empty).flatMap { logs =>
        Ref.of[IO, Int](0).flatMap { attemptsSoFar =>
          implicit val bg     = shop.background.NoOp
          implicit val logger = shop.logger.acc(logs)
          new CheckoutProgram[IO](
```

```
            recoveringClient(attemptsSoFar, pid),
            successfulCart(ct),
            successfulOrders(oid),
            retryPolicy
          ).checkout(uid, card)
            .attempt
            .flatMap {
              case Right(id) =>
                logs.get.map { xs =>
                  assert(id === oid && xs.size === 1)
                }
              case Left(_) => fail("Expected Payment Id")
            }
        }
      }
    }
}
```

Once again, we use our error-messages-accumulating logger. We also create an extra `Ref` as a counter for our recovering client. In the end, we assert we get the expected `OrderId`, and that the count is equal to one, meaning the service recovered after one retry.

Failing orders

If the order fails to be created, we retry a configured number of times, specified in our retry policy. When the maximum number of retries is reached, we return the `OrderId` and schedule this action to run again in the background. In order to test this complex case, we need a new interpreter for our `Background` interface.

```
def counter(ref: Ref[IO, Int]): Background[IO] =
  new Background[IO] {
    def schedule[A](
      fa: IO[A],
      duration: FiniteDuration
    ): IO[Unit] =
      ref.update(_ + 1)
  }
```

We are just going to have a counter that checks how many actions have been submitted to be scheduled to run in the background. This is as much as we can do since we cannot possibly know what an `IO[A]` does.

We also need a failing interpreter for `Orders`.

```
val failingOrders: Orders[IO] =
  new TestOrders {
    override def create(
      userId: UserId,
      paymentId: PaymentId,
      items: List[CartItem],
      total: Money
    ): IO[OrderId] =
      IO.raiseError(OrderError(""))
  }
```

With all these components in place, let's study our test implementation, where we also need to check the logs for the retrying logic.

```
forAll {
  (
    uid: UserId, pid: PaymentId,
    ct: CartTotal, card: Card
  ) =>
    spec("cannot create order, run in the background") {
      Ref.of[IO, Int](0).flatMap { counterRef =>
        Ref.of[IO, List[String]](List.empty).flatMap { logs =>
          implicit val bg     = shop.background.counter(counterRef)
          implicit val logger = shop.logger.acc(logs)
          new CheckoutProgram[IO](
            successfulClient(pid),
            successfulCart(ct),
            failingOrders,
            retryPolicy
          ).checkout(uid, card)
            .attempt
            .flatMap {
              case Left(OrderError(_)) =>
                (ref.get, logs.get).mapN {
                  case (c, (x :: y :: xs)) =>
                    assert(
                      x.contains("Rescheduling") &&
                      y.contains("Giving up") &&
                      xs.size === MaxRetries &&
                      c === 1
                    )
                  case _ =>
                    fail(s"Expected $MaxRetries retries and reschedule")
                }
```

```
              case _ =>
                fail("Expected order error")
            }
        }
      }
    }
}
```

We await an `OrderError`, which we get after the maximum number of retries is reached. If this condition is met, we expect the background counter to have the value one and the error logs to have the size of the configured `MaxRetries`, as well as containing both the "Rescheduling" and "Giving up" words; otherwise, the test fails.

Failing cart deletion

The last and less critical action of our `checkout` function is deleting the shopping cart from the cache. We have mentioned that if this fails, we don't care too much and should continue operating normally.

So let's create a failing shopping cart to test this case.

```
def failingCart(cartTotal: CartTotal): ShoppingCart[IO] =
  new TestCart {
    override def get(userId: UserId): IO[CartTotal] =
      IO.pure(cartTotal)
    override def delete(userId: UserId): IO[Unit] =
      IO.raiseError(new Exception(""))
  }
```

All we need to check is that our `checkout` function returns an expected `OrderId` without failing.

```
forAll {
  (
    uid: UserId, pid: PaymentId, oid: OrderId,
    ct: CartTotal, card: Card
  ) =>
    spec("failing to delete cart does not affect checkout") {
      implicit val bg = shop.background.NoOp
      import shop.logger.NoOp
      new CheckoutProgram[IO](
        successfulClient(pid),
        failingCart(ct),
        successfulOrders(oid),
        retryPolicy
```

```
      ).checkout(uid, card)
        .map { id =>
          assert(id === oid)
        }
    }
  }
```

We can now say we are covered from the scenarios we could think of. In addition to our custom cases, we are also testing different inputs to our program thanks to property-based testing, which is sometimes underrated. Thus, we can conclude with the most interesting testing piece in our application to continue testing our `HttpRoutes`.

Http routes

In Chapter 5, I have said *a server is a function*, and I literally meant it! We don't need to spin up a server to test our `HttpRoutes` since they are plain functions. Moreover, we can seize the power of property-based testing to write accurate tests.

Here is a test for `BrandRoutes`, which exposes a single `GET` endpoint to retrieve all the brands:

```
forAll { (b: List[Brand]) =>
  spec("GET brands [OK]") {
    GET(Uri.uri("/brands")).flatMap { req =>
      val routes = new BrandRoutes[IO](dataBrands(b)).routes
      routes.run(req).value.flatMap {
        case Some(resp) =>
          resp.as[Json].map { json =>
            assert(
              resp.status === Status.Ok &&
              json.dropNullValues === b.asJson.dropNullValues
            )
          }
        case None => fail("route nout found")
      }
    }
  }
}
```

Step by step, this is what is going on:

- A `List[Brand]` is obtained using generators (`Arbitrary` instance).
- A `GET` request is built using the client DSL provided by Http4s.
- Our `HttpRoutes` are executed by feeding our `Request` as the argument.
  - We `flatMap` to access the inner value of type `Option[Response[IO]]`.
  - If `Some(resp)`, we assert the `body` and the `status` are the expected ones.
  - Otherwise, we fail the test.

Since this pattern will become repetitive (running routes and asserting on the response), we can extract it out into another function we can reuse.

```
def assertHttp[A: Encoder](
    routes: HttpRoutes[IO],
    req: Request[IO]
)(
    expectedStatus: Status,
    expectedBody: A
```

```
) =
  routes.run(req).value.flatMap {
    case Some(resp) =>
      resp.as[Json].map { json =>
        assert(
          resp.status === expectedStatus &&
          json.dropNullValues === expectedBody.asJson.dropNullValues
        )
      }
    case None => fail("route nout found")
  }
```

Furthermore, we can define an `HttpTestSuite` where this and other functions can be placed.

```
trait HttpTestSuite extends PureTestSuite
```

Now our test looks much nicer and concise.

```
forAll { (b: List[Brand]) =>
  spec("GET brands [OK]") {
    GET(Uri.uri("/brands")).flatMap { req =>
      val routes = new BrandRoutes[IO](dataBrands(b)).routes
      assertHttp(routes, req)(Status.Ok, b)
    }
  }
}
```

Next is `ItemRoutes`, which can additionally receive a query parameter. So let's examine this particular test and skip the rest to avoid repetition.

```
forAll { (it: List[Item], b: Brand) =>
  spec("GET items by brand [OK]") {
    GET(Uri.uri("/items").withQueryParam(b.name.value))
      .flatMap { req =>
        val routes = new ItemRoutes[IO](dataItems(it)).routes
        assertHttp(routes, req)(Status.Ok, it)
      }
  }
}
```

We construct our `Uri` using both the `uri` and the `withQueryParam` methods.

Finally, let's see how to test authenticated routes. We will only focus on `CartRoutes` and skip the rest, as they are almost identical.

We first need a fake `AuthMiddleware` that bypasses security (always returns a `User`).

```
val authUser =
  CommonUser(
    User(
      UserId(UUID.randomUUID),
      UserName("user")
    )
  )

val authMiddleware: AuthMiddleware[IO, CommonUser] =
  AuthMiddleware(Kleisli.pure(authUser))
```

Afterward, we can create our `HttpRoutes` and define our unit test.

```
forAll { (ct: CartTotal) =>
  spec("GET shopping cart [OK]") {
    GET(Uri.uri("/cart")).flatMap { req =>
      val routes =
        new CartRoutes[IO](dataCart(ct)).routes(authMiddleware)
      assertHttp(routes, req)(Status.Ok, ct)
    }
  }
}
```

The only difference is that we need to supply an `AuthMiddleware` to obtain our `HttpRoutes`; the rest should be reasonably straightforward at this point.

Next, we can see how to test a `POST` endpoint:

```
forAll { (c: Cart) =>
  spec("POST add item to shopping cart [OK]") {
    POST(c, Uri.uri("/cart")).flatMap { req =>
      val routes =
        new CartRoutes[IO](
          new TestShoppingCart
        ).routes(authMiddleware)
      assertHttpStatus(routes, req)(Status.Created)
    }
  }
}
```

The `POST.apply` method takes in a body (a `Cart` in this case) and a `Uri`. There should be an `EntityEncoder[IO, Cart]` in scope, otherwise, it would not compile. There is also a new generic method `assertHttpStatus`, which is similar to `assertHttp` but it only checks the status of the response. The rest should be self-explanatory.

Other HTTP methods such as `GET`, `PUT`, and `DELETE` also support taking a request body `A` as an argument, given an `EntityEncoder[F, A]`.

Integration tests

In this last section, we will see why integration tests are also essential. But first, let's be clear: What do integration tests mean, exactly?

We can interpret them in many ways. The usual meaning refers to starting up our entire application to be tested against all the external components, which in our case are PostgreSQL, Redis, and the remote Payment client.

However, this is tedious, and the benefits don't justify the cost of having such an exclusive testing environment.

A good approach is to test external interpreters in isolation. For example, we could test the Postgres interpreters in a single test suite, and the Redis interpreters in another test suite. If we have a real test payment client, we could also test that. In this case, we don't, so we are going to move forward with the first two.

Resource allocation

When we have to deal with resources that must be shared across tests such as a Postgres connection, we realize we have a problem since Scalatest is not very friendly with purely functional resource management.

Our only hope is to wait for the release of a purely functional testing library such as Flawless[2], or give the work-in-progress Kallikrein[3] a try. Other than that, we are left alone in side-effects land. Though, we can try to hide this in the same way we are hiding the call to `unsafeToFuture()` in our tests.

We need to define a new test suite that extends `PureTestSuite` and mixes-in the `BeforeAndAfterAll` trait from Scalatest.

```
trait ResourceSuite[A]
  extends PureTestSuite
  with BeforeAndAfterAll {

  def resources: Resource[IO, A]

  // ... more here ...

}
```

It is parameterized on the resource type `A`, and it defines an abstract method `resources`. Next, we need some private *mutable* variables and a *latch* (backed by a `Deferred`).

[2] https://github.com/kubukoz/flawless
[3] https://github.com/tek/kallikrein

```
private[this] var res: A            = _
private[this] var cleanUp: IO[Unit] = _

private[this] val latch = Deferred[IO, Unit].unsafeRunSync()
```

Followed by overriding the methods from the `BeforeAndAfterAll` trait:

```
override def beforeAll(): Unit = {
  super.beforeAll()
  val (r, h) = resources.allocated.unsafeRunSync()
  res = r
  cleanUp = h
  latch.complete(()).unsafeRunSync()
}

override def afterAll(): Unit = {
  cleanUp.unsafeRunSync()
  super.afterAll()
}
```

The `allocated` method on `Resource` returns a tuple of the acquired resource and the release handle. **It should only be used if you know what you are doing.**

After calling `unsafeRunSync`, we assign its values to our mutable variables. In our `afterAll` method, we simply execute our clean up handle by running its side-effects via `unsafeRunSync`.

Next, we define a new method `withResources`, taking a function `(=> A) => Unit`, where `A` is the type of the resource.

```
def withResources(f: (=> A) => Unit): Unit = f {
  latch.get.unsafeRunSync
  res
}
```

Our `latch` is only necessary in some specific cases, due to the side-effectful nature of our testing framework. Though, by having it here, we are covered from all the possible scenarios.

We can now make use of our new suite by specifying the type of our resource.

```
class MyTest extends ResourceSuite[MyConnection[IO]] {

  override def resources: Resource[IO, MyConnection[IO]] = ???

}
```

Our tests should now look as follows:

```
withResources { res =>

  forAll { (a: A) =>
    spec("test using shared resource") {
      IO(assert(...))
    }
  }

  // ... more tests here ...
}
```

Hiding the ugly stuff in our custom suite makes our tests look neat.

Postgres

Let's start by creating a test class that extends our new resource-aware test suite.

```
class PostgreSQLTest
  extends ResourceSuite[Resource[IO, Session[IO]]] {

  // For it:tests, one test is enough
  val MaxTests: PropertyCheckConfigParam = MinSuccessful(1)

  override def resources =
    Session.pooled[IO](
      host = "localhost",
      port = 5432,
      user = "postgres",
      database = "store",
      max = 10
    )

}
```

Besides our Postgres session pool, which will be used by all our tests, we have a `PropertyCheckConfigParam` value indicating the number of tests we want to generate per scenario. We can argue that one single test is enough for such an expensive test.

This test class by itself doesn't spin up any Postgres server. It is our responsibility to have a running server in order to execute our integration tests. In our application, it is recommended to use `docker-compose`, which can be configured to run in our CI build, as we will see in Chapter 9.

All our Postgres tests will be making use of our shared resource (the session pool), so we will define the acquisition only once.

```
withResources { pool =>
  // tests go here
}
```

We can now write our first test for the `LiveBrands` interpreter, reusing the `Arbitrary[Brand]` instance.

```
forAll(MinTests) { (brand: Brand) =>
  spec("Brands") {
    LiveBrands.make[IO](pool).flatMap { b =>
      for {
        x <- b.findAll
        _ <- b.create(brand.name)
        y <- b.findAll
        z <- b.create(brand.name).attempt
      } yield
        assert(
          x.isEmpty &&
          y.count(_.name === brand.name) === 1 &&
          z.isLeft
        )
    }
  }
}
```

We are invoking `findAll` and `create` a couple of times to later assert on the specified conditions. We can highlight that the first result should be empty (since there is no data), whereas the last `create` action fails because the same `Brand` already exists.

The test for `Categories` is almost identical, so we will skip it. Next is `Items`.

```
forAll(MaxTests) { (item: Item) =>
  spec("Items") {
    def newItem(
        bid: Option[BrandId],
        cid: Option[CategoryId]
    ) = CreateItem(
      name = item.name,
      description = item.description,
```

```
      price = item.price,
      brandId = bid.getOrElse(item.brand.uuid),
      categoryId = cid.getOrElse(item.category.uuid)
    )

    for {
      b <- LiveBrands.make[IO](pool)
      c <- LiveCategories.make[IO](pool)
      i <- LiveItems.make[IO](pool)
      x <- i.findAll
      _ <- b.create(item.brand.name)
      d <- b.findAll.map(_.headOption.map(_.uuid))
      _ <- c.create(item.category.name)
      e <- c.findAll.map(_.headOption.map(_.uuid))
      _ <- i.create(newItem(d, e))
      y <- i.findAll
    } yield
      assert(
        x.isEmpty &&
        y.count(_.name === item.name) === 1
      )
  }
}
```

Since we are hitting a Postgres database with key constraints, we can no longer create an `Item` with an inexistent `BrandId` or `CategoryId`. Therefore, we need to make sure these are created before creating an `Item`. Ultimately, we assert the first result is empty and that the second result contains the `Item` that was persisted.

Next is `Users`, which also needs a `Crypto` interpreter.

```
lazy val salt =
  PasswordSalt(Secret("53kr3t": NonEmptyString))

forAll(MaxTests) { (username: UserName, password: Password) =>
  spec("Users") {
    for {
      c <- LiveCrypto.make[IO](salt)
      u <- LiveUsers.make[IO](pool, c)
      d <- u.create(username, password)
      x <- u.find(username, password)
      y <- u.find(username, Password("foo"))
```

```
      z <- u.create(username, password).attempt
    } yield
      assert(
        x.count(_.id === d) === 1 &&
        y.isEmpty && z.isLeft
      )
  }
}
```

We create a new user given the properties `UserName` and `Password`, retrieve the user, and then assert that the `UserId` matches the one we got on creation. Next, we try to find the same user using a different password, which should be empty. Finally, we try to create the same user once again, which should raise an error.

Lastly, we have `Orders`.

```
forAll(MaxTests) {
  (
    oid: OrderId, pid: PaymentId, un: UserName,
    pw: Password, items: List[CartItem], price: Money
  ) =>
    spec("Orders") {
      for {
        o <- LiveOrders.make[IO](pool)
        c <- LiveCrypto.make[IO](salt)
        u <- LiveUsers.make[IO](pool, c)
        d <- u.create(un, pw)
        x <- o.findBy(d)
        y <- o.get(d, oid)
        i <- o.create(d, pid, items, price)
      } yield
        assert(
          x.isEmpty && y.isEmpty &&
          i.value.version === 4 // UUID version
        )
    }
}
```

Again, to create an `Order`, we need an existent `User` given the `user_id` constraint in our `orders` table.

Redis

We have only two Redis interpreters: `Auth` and `ShoppingCart`, both taking some algebras whose interpreters use Postgres.

Let's start defining our test class by extending our `ResourceSuite`.

```
class RedisTest
  extends ResourceSuite[RedisCommands[IO, String, String]] {

  // For it:tests, one test is enough
  val MaxTests: PropertyCheckConfigParam = MinSuccessful(1)

  override def resources =
    for {
      uri <- Resource.liftF(RedisURI.make[IO]("redis://localhost"))
      client <- RedisClient[IO](uri)
      cmd <- Redis[IO, String, String](client, RedisCodec.Utf8)
    } yield cmd
```

As our Postgres tests, it will run a maximum of one test per scenario. We have also defined a resource to acquire a Redis connection, which will be used by our tests.

Shopping Cart interpreter

Let's review the arguments necessary to create a `LiveShoppingCart`.

```
object LiveShoppingCart {
  def make[F[_]: GenUUID: MonadThrow](
      items: Items[F],
      redis: RedisCommands[F, String, String],
      exp: ShoppingCartExpiration
  ): F[ShoppingCart[F]] = ???
}
```

The `LiveItems` interpreter uses Postgres, so here we have two options. Either we use the real interpreter, or we use an in-memory interpreter.

We will choose the latter to avoid mixing Postgres tests with Redis tests. Let's have a look at the following in-memory implementation:

```
class TestItems(
    ref: Ref[IO, Map[ItemId, Item]]
) extends Items[IO] {
```

```
  def findAll: IO[List[Item]] =
    ref.get.map(_.values.toList)

  def findBy(brand: BrandName): IO[List[Item]] =
    IO.pure(List.empty)

  def findById(itemId: ItemId): IO[Option[Item]] =
    ref.get.map(_.get(itemId))

  def create(item: CreateItem): IO[Unit] =
    GenUUID[IO].make[ItemId].flatMap { id =>
      val brand    = Brand(
        item.brandId, BrandName("foo")
      )
      val category = Category(
        item.categoryId, CategoryName("foo")
      )
      val newItem  = Item(
        id, item.name, item.description,
        item.price, brand, category
      )
      ref.update(_.updated(id, newItem))
    }

  def update(item: UpdateItem): IO[Unit] =
    ref.update { x =>
      x.get(item.id)
       .fold(x)(i =>
         x.updated(item.id, i.copy(price = item.price))
       )
    }
}
```

We use a `Ref`, as seen in Chapter 2, as our concurrent in-memory storage. The relevant methods are `create` and `update`. The first one creates an `Item` of random `Brand` and `Category` (this is irrelevant); the second method updates the price of a given `Item`, if it exists.

The missing argument is `ShoppingCartExpiration`, which is just a newtype for a `FiniteDuration`.

```
val Exp = ShoppingCartExpiration(30.seconds)
```

Shopping Cart test

Again, all our tests will start with the acquisition of the shared resource, which is defined only once.

```
withResources { cmd =>
  // tests go here
}
```

With all the pieces in place, let's proceed to write our `ShoppingCart` test.

```
forAll(MaxTests) {
  (
    uid: UserId, it1: Item, it2: Item,
    q1: Quantity, q2: Quantity
  ) =>
    spec("Shopping Cart") {
      Ref.of[IO, Map[ItemId, Item]](
        Map(it1.uuid -> it1, it2.uuid -> it2)
      ).flatMap { itemsRef =>
        val items = new TestItems(itemsRef)
        LiveShoppingCart.make[IO](
          items, cmd, Exp
        ).flatMap { c =>
          for {
            x <- c.get(uid)
            _ <- c.add(uid, it1.uuid, q1)
            _ <- c.add(uid, it2.uuid, q1)
            y <- c.get(uid)
            _ <- c.removeItem(uid, it1.uuid)
            z <- c.get(uid)
            _ <- c.update(uid, Cart(Map(it2.uuid -> q2)))
            w <- c.get(uid)
            _ <- c.delete(uid)
            v <- c.get(uid)
          } yield
            assert(
              x.items.isEmpty && y.items.size === 2 &&
              z.items.size === 1 && v.items.isEmpty &&
              w.items.headOption.fold(false)(_.quantity === q2)
            )
        }
      }
    }
  }
```

Let's break it apart. We start by creating a `Ref` with the two `Items` we got from our generators, which is used by our `TestItems` interpreter. Finally, we proceed to instantiate our `ShoppingCart` interpreter and write the appropriate assertions.

Auth interpreter

Let's now review the arguments taken by the `LiveAuth` interpreter.

```
object LiveAuth {
  def make[F[_]: Sync](
      tokenExpiration: TokenExpiration,
      tokens: Tokens[F],
      users: Users[F],
      redis: RedisCommands[F, String, String]
  ): F[Auth[F]] = ???
}
```

We need a simple `TokenExpiration`, which wraps a `FiniteDuration`. Next is `Tokens`, which takes some data from `AuthData`. Finally, we need a `Users` interpreter, in addition to our `RedisCommands`.

Here we are in the same situation we were before. We can either use the real Postgres interpreter, or we can roll our in-memory implementation. Once more, we will choose the latter.

Let's have a look at this interpreter.

```
class TestUsers(un: UserName) extends Users[IO] {

  def find(
    username: UserName,
    password: Password
  ): IO[Option[User]] =
    (Eq[UserName].eqv(username, un))
      .guard[Option]
      .as(User(UserId(UUID.randomUUID), un))
      .pure[IO]

  def create(
    username: UserName,
    password: Password
  ): IO[UserId] =
    GenUUID[IO].make[UserId]

}
```

That was fairly easy. The only internal state it has is a single `UserName` we pass as an argument. When the `find` method is invoked with this username, we get a `User` back; otherwise, we get none. The `create` method always creates a new `UserId`.

Before we write our test, we need all the necessary test data.

```
lazy val tokenConfig =
  JwtSecretKeyConfig(Secret("bar": NonEmptyString))
lazy val tokenExp = TokenExpiration(30.seconds)
lazy val jwtClaim = JwtClaim("test")
lazy val userJwtAuth =
  UserJwtAuth(JwtAuth.hmac("bar", JwtAlgorithm.HS256))
```

Auth test

We are now ready to implement our `Auth` test.

```
forAll(MaxTests) {
  (un1: UserName, un2: UserName, pw: Password) =>
    spec("Authentication") {
      for {
        t <- LiveTokens.make[IO](tokenConfig, tokenExp)
        a <- LiveAuth.make(tokenExp, t, new TestUsers(un2), cmd)
        u <- LiveUsersAuth.make[IO](cmd)
        x <- u.findUser(JwtToken("invalid"))(jwtClaim)
        j <- a.newUser(un1, pw)
        e <- jwtDecode[IO](j, userJwtAuth.value).attempt
        k <- a.login(un2, pw)
        f <- jwtDecode[IO](k, userJwtAuth.value).attempt
        _ <- a.logout(k, un2)
        y <- u.findUser(k)(jwtClaim)
        w <- u.findUser(j)(jwtClaim)
      } yield
        assert(
          x.isEmpty && e.isRight && f.isRight && y.isEmpty &&
          w.fold(false)(_.value.name === un1)
        )
    }
}
```

We get two different `UserName`: `un1` and `un2`. Our in-memory `Users` interpreter takes in `un2`. Next, we try to retrieve the user using an invalid JWT token, which should return nothing.

Next, we create a new user using `un1` and the given password. We check that the JWT token we get back is valid in the following step. Next, we try to login using `un2` and the given password, and expect a successful response. Then, we log out `un2` and expect not to find the user anymore using its token. Finally, we expect to find the `un1` user, which has been assigned the `j` token since this user didn't log out.

Conclusion

Integration tests are necessary to catch bugs that would otherwise be hard to catch. For example, Postgres or Redis specific issues, or bugs related to database codecs.

We have seen that testing our interpreters is enough, and we don't need to write integration tests for the entire application, even though this is good to have if you can allow yourself the time and environment to make it happen.

To prove this right, I can attest to have solved two bugs in our Shopping Cart application related to codecs that I wouldn't have spotted otherwise. It saved my day.

In the next chapter, we will continue assembling the pieces of our application.

Chapter 8: Assembly

We have come a long way. Having turned business requirements into technical specifications, we then designed our system using purely functional programming and finally wrote property-based tests.

Now is the time to put all the pieces together, and here is where I would like to quote one of my favorite song-writers that says:

> I know the pieces fit
>
> *Maynard James Keenan*

In this chapter, we are going to assemble our application, and we will ultimately spin up our HTTP server, connect to our database, and start serving requests.

Logging

Logging actions are seen as those invasive blocks of code that pollute our codebase. Though, these are a necessary evil if we intend to troubleshoot issues in our application.

When and how much to log?

Everyone is free to make a choice. A good practice is to only log critical information, such as our retrying functions being invoked, but leaving everything else out.

On the one hand, because it pollutes our codebase. On the other hand, because it creates unnecessary logs.

Distributed tracing

Alternatively, there is distributed tracing, which is a hot topic these days and might eventually replace logging as we know it. Quoting the Open Tracing[1] definition:

> Distributed tracing, also called distributed request tracing, is a method used to profile and monitor applications, especially those built using a microservices architecture. Distributed tracing helps pinpoint where failures occur and what causes poor performance.

In terms of Scala libraries, we have the following options:

- Natchez[2]: comes with support for Jaeger[3], Honeycomb[4], OpenCensus[5] and LightStep[6], and is used by Skunk.
- Pure Tracing[7]: aims to be the standard library for tracing in Scala but it is not so active at the moment.

There is another Scala library, focused on simple tracing for Http4s, which I maintain:

- Http4s Tracer[8]

Natchez seems to be the most stable so far if you want to give distributed tracing a try.

[1] https://opentracing.io/docs/overview/what-is-tracing/
[2] https://github.com/tpolecat/natchez
[3] https://www.jaegertracing.io/
[4] https://www.honeycomb.io/
[5] https://opencensus.io/
[6] https://lightstep.com/
[7] https://github.com/tabdulradi/puretracing
[8] https://github.com/profunktor/http4s-tracer

Log for Cats

In our application, we are going to use Log4cats[9], which is by now the standard logging library of the Cats ecosystem.

Whenever we need logging capability, all we need is to add a `Logger` constraint to our effect type. For instance:

```
def program[F[_]: Apply: Logger]: F[Unit] =
  Logger[F].info("starting program") *>
    doSomething
```

It is not the first time we see `Logger`. In Chapter 4, our retrying functions were making use of it, even though we didn't dive into details since it is easy to deduct its usage.

Normally, people use the `Slf4j` implementation, which is created as shown below.

```
implicit val logger: Logger[IO] = Slf4jLogger.getLogger[IO]
```

In fact, this is how it is created in our `Main` class, as we will see soon.

In Chapter 7, we have seen how to turn off logging by creating a custom interpreter. What we didn't see is that Log4cats comes with its own `TestingLogger` that might be suitable for your needs. Make sure to check it out before you roll your own.

The reason why we didn't use it is because it appends a message for every logging method, and we were only interested in appending only error messages.

Alternative logging libraries

There are other promising Scala logging libraries out there you might want to consider: LogStage[10] and Odin[11].

Both libraries provide *structural* and *contextual* logging, among other features.

[9] https://github.com/ChristopherDavenport/log4cats
[10] https://izumi.7mind.io/latest/release/doc/logstage/index.html
[11] https://github.com/valskalla/odin

Configuration

Every application demands different configurations. The classic methodology is to use a configuration file, load it into memory at start-up, and make use of its values. This allows us to discern between configuration for production and test environments, for example.

The traditional file-based configuration is fine, but it comes with its drawbacks. We need to write a file, whether it is HOCON, YAML, TOML, or any of the thousands new formats that come out every day, and we also need a library able to read these files.

If we were writing a Haskell application, I would recommend using Dhall[12] anytime, which supports converting to YAML and JSON, among others. However, Scala support for Dhall is not great at the moment.

The main reason for using configuration files is so we can change settings without recompiling our application. In reality, this rarely happens, as developers tend to make changes by committing them to a version control repository, and then a continuous integration (CI) system would deploy the application to the specified environments.

Another feature we are interested in is the early detection of configuration failures[13]. Those values that are known at compile time can be validated with our type checker; other values need to be validated as soon as possible at the start-up of our application.

For these cases, and in general, it is easier to have our configurations directly in our source code. Except for secrets such as passwords or private keys; never store secrets in your code.

Configuration values can be stored not only in a configuration file but also in an *environment variable* or a *system property*, which are perfect for storing secrets. We would then need to aggregate all the different sources of configuration.

In our application, we will be using Ciris[14], a purely functional configuration library that embraces *configuration as code* and supports all of these features!

To better understand why we are going to use Ciris, you can read this blog post[15] written by its author. From now on, we are only going to focus on its usage.

Application

Let's get right into the configuration of our application to later analyze each part.

First of all, we define a `default` method taking some parameters that are different based on the environment (`test` or `production`).

[12] https://dhall-lang.org/
[13] https://www.usenix.org/system/files/conference/osdi16/osdi16-xu.pdf
[14] http://cir.is
[15] https://typelevel.org/blog/2017/06/21/ciris.html

```
private def default(
    redisUri: RedisURI,
    paymentUri: PaymentURI
): ConfigValue[AppConfig] =
  (
    env("SC_JWT_SECRET_KEY").as[NonEmptyString].secret,
    env("SC_JWT_CLAIM").as[NonEmptyString].secret,
    env("SC_ACCESS_TOKEN_SECRET_KEY").as[NonEmptyString].secret,
    env("SC_ADMIN_USER_TOKEN").as[NonEmptyString].secret,
    env("SC_PASSWORD_SALT").as[NonEmptyString].secret
  ).parMapN { (secretKey, claimStr, tokenKey, adminToken, salt) =>
    AppConfig(
      AdminJwtConfig(
        JwtSecretKeyConfig(secretKey),
        JwtClaimConfig(claimStr),
        AdminUserTokenConfig(adminToken)
      ),
      JwtSecretKeyConfig(tokenKey),
      PasswordSalt(salt),
      TokenExpiration(30.minutes),
      ShoppingCartExpiration(30.minutes),
      CheckoutConfig(
        retriesLimit = 3,
        retriesBackoff = 10.milliseconds
      ),
      PaymentConfig(paymentUri),
      HttpClientConfig(
        connectTimeout = 2.seconds,
        requestTimeout = 2.seconds
      ),
      PostgreSQLConfig(
        host = "localhost",
        port = 5432,
        user = "postgres",
        database = "store",
        max = 10
      ),
      RedisConfig(redisUri)
    )
  }
```

Both our URIs are defined as `NonEmptyString`. We could be more strict and define a better refinement type, but this would do for now.

Next, we see the use of the `env` function, which unsurprisingly, reads an environment variable. The `as` function will attempt to decode the `String` value into the specified type. Lastly, the `secret` function will encode this value as "secret" using the `Secret` datatype, explained at the end of this section.

Another cool feature of this library is that we can compose expressions, as it uses the `ConfigValue[A]` monad. Here is an example taken from the official documentation:

```
env("API_PORT").or(prop("api.port")).as[UserPortNumber].option,
```

It lets us read an environment variable. If it is not present, it tries to read a system property, and it returns an optional type.

Continuing with our function, we see `parMapN`, the standard function from Cats that will execute all the actions in parallel, and it will give us all values at once, if there is no error.

The rest is just building values using case classes and newtypes.

The interesting part comes next, where we discern between our two different environments and load our configuration.

```
def apply[F[_]: Async: ContextShift]: F[AppConfig] =
  env("SC_APP_ENV")
    .as[AppEnvironment]
    .flatMap {
      case Test =>
        default(
          redisUri = RedisURI("redis://localhost"),
          paymentUri = PaymentURI("http://10.123.154.10/api")
        )
      case Prod =>
        default(
          redisUri = RedisURI("redis://10.123.154.176"),
          paymentUri = PaymentURI("https://payments.net/api")
        )
    }
    .load[F]
```

Again, it reads an environment variable that tells the application which environment it is on, it attempts to decode it as our `AppEnvironment` datatype (ADT), it `flatMap`s on the result, and it invokes our `default` method with the corresponding arguments. Ultimately, it invokes the `load[F]` method to load the configuration into memory, which requires both `Async[F]` and `ContextShift[F]`.

Here is our ADT definition:

```
sealed abstract class AppEnvironment
    extends EnumEntry
    with Lowercase

object AppEnvironment
    extends Enum[AppEnvironment]
    with CirisEnum[AppEnvironment] {

  case object Test extends AppEnvironment
  case object Prod extends AppEnvironment

  val values = findValues

}
```

It uses the Enumeratum[16] library together with the Ciris Enumeratum module.

Our configuration domain model is mostly defined as either case classes or newtypes, and sometimes combined with refinement types. Here is the definition of some of them:

```
@newtype case class PasswordSalt(value: Secret[NonEmptyString])

@newtype case class ShoppingCartExpiration(value: FiniteDuration)

case class CheckoutConfig(
        retriesLimit: PosInt,
        retriesBackoff: FiniteDuration
)
```

The `Secret` datatype, provided by Ciris, allows us to protect sensitive data from being undesirably logged or stored.

For conciseness, we will skip the rest since they are very similar. Check out the source code for more!

[16] https://github.com/lloydmeta/enumeratum

Modules

In Chapter 3, we have explored the tagless final encoding and seen how modules help us organizing our codebase. Here we are going to make use of this design pattern for our application. Let's start by enumerating the modules we will have.

- `Algebras`: it groups all our algebras, including our shared Redis and PostgreSQL connections.
- `HttpApi`: it defines all our HTTP routes and middlewares.
- `HttpClients`: it defines our only HTTP client: the payments client.
- `Programs`: it defines our `checkout` program and retry policy.
- `Security`: it defines our instance of `Users` and all the authentication related functionality.

Algebras

Below is the definition of our `Algebras` module, including its smart constructor.

```
object Algebras {
  def make[F[_]: Concurrent: Parallel: Timer](
      redis: RedisCommands[F, String, String],
      sessionPool: Resource[F, Session[F]],
      cartExpiration: ShoppingCartExpiration
  ): F[Algebras[F]] =
    for {
      brands <- LiveBrands.make[F](sessionPool)
      categories <- LiveCategories.make[F](sessionPool)
      items <- LiveItems.make[F](sessionPool)
      cart <- LiveShoppingCart.make[F](items, redis, cartExpiration)
      orders <- LiveOrders.make[F](sessionPool)
      health <- LiveHealthCheck.make[F](sessionPool, redis)
    } yield new Algebras[F](
        cart, brands, categories, items, orders, health
      )
}

final class Algebras[F[_]] private (
    val cart: ShoppingCart[F],
    val brands: Brands[F],
    val categories: Categories[F],
    val items: Items[F],
    val orders: Orders[F],
```

```
    val healthCheck: HealthCheck[F]
)
```

Our `Algebras` class is the interface we will be using in other modules. We make it `final` because no other component should extend it. Also, its smart constructor initializes all the `Live` interpreters of the algebras we need.

HTTP Clients

Next is our implementation of the `HttpClients` module, which only contains our `PaymentClient`.

```
object HttpClients {
  def make[F[_]: Sync](
      cfg: PaymentConfig,
      client: Client[F]
  ): F[HttpClients[F]] =
    Sync[F].delay(
      new HttpClients[F] {
        def payment: PaymentClient[F] = new LivePaymentClient[F](cfg, client)
      }
    )
}

trait HttpClients[F[_]] {
  def payment: PaymentClient[F]
}
```

In this case, we have defined a simple interface to be used by other modules.

Programs

Next is `Programs`, which makes use of both `Algebras` and `HttpClients`.

```
object Programs {
  def make[F[_]: Background: Logger: Sync: Timer](
      checkoutConfig: CheckoutConfig,
      algebras: Algebras[F],
      clients: HttpClients[F]
  ): F[Programs[F]] =
    Sync[F].delay(
      new Programs[F](checkoutConfig, algebras, clients)
    )
```

```
}

final class Programs[
    F[_]: Background: Logger: MonadThrow: Timer
] private (
    cfg: CheckoutConfig,
    algebras: Algebras[F],
    clients: HttpClients[F]
) {

  val retryPolicy: RetryPolicy[F] =
    limitRetries[F](cfg.retriesLimit.value) |+|
    exponentialBackoff[F](cfg.retriesBackoff)

  def checkout: CheckoutProgram[F] =
    new CheckoutProgram[F](
      clients.payment,
      algebras.cart,
      algebras.orders,
      retryPolicy
    )

}
```

Besides the modules, it takes a `CheckoutConfig` used to create our program's `RetryPolicy`.

Security

Our next module contains all the authentication stuff.

```
object Security {
  def make[F[_]: Sync](
      cfg: AppConfig,
      sessionPool: Resource[F, Session[F]],
      redis: RedisCommands[F, String, String]
  ): F[Security[F]] = {

    val adminJwtAuth: AdminJwtAuth =
      AdminJwtAuth(
        JwtAuth
          .hmac(
            cfg.adminJwtConfig.secretKey.value.value.value,
```

```
            JwtAlgorithm.HS256
          )
      )

    val userJwtAuth: UserJwtAuth =
      UserJwtAuth(
        JwtAuth
          .hmac(
            cfg.tokenConfig.value.value.value,
            JwtAlgorithm.HS256
          )
      )

    val adminToken = JwtToken(
      cfg.adminJwtConfig.adminToken.value.value.value
    )

    for {
      adminClaim <- jwtDecode[F](adminToken, adminJwtAuth.value)
      content <- ApThrow[F].fromEither(
                   jsonDecode[ClaimContent](adminClaim.content)
                 )
      adminUser = AdminUser(
                    User(UserId(content.uuid), UserName("admin"))
                  )
      tokens <- LiveTokens.make[F](
                  cfg.tokenConfig, cfg.tokenExpiration
                )
      crypto <- LiveCrypto.make[F](cfg.passwordSalt)
      users <- LiveUsers.make[F](sessionPool, crypto)
      auth <- LiveAuth.make[F](
                cfg.tokenExpiration, tokens, users, redis
              )
      adminAuth <- LiveAdminAuth.make[F](adminToken, adminUser)
      usersAuth <- LiveUsersAuth.make[F](redis)
    } yield new Security[F](
        auth, adminAuth, usersAuth, adminJwtAuth, userJwtAuth
      )

  }
}

final class Security[F[_]] private (
    val auth: Auth[F],
```

```
    val adminAuth: UsersAuth[F, AdminUser],
    val usersAuth: UsersAuth[F, CommonUser],
    val adminJwtAuth: AdminJwtAuth,
    val userJwtAuth: UserJwtAuth
)
```

It takes a Postgres and a Redis connection as arguments. Then, it creates the JWT authentication schema for both `AdminUser` and `CommonUser`. Lastly, it creates all the necessary algebras related to authentication.

HTTP API

Finally, our `HttpApi` module groups all our HTTP routes and middlewares. Let's start with the smart constructor.

```
object HttpApi {
  def make[F[_]: Concurrent: Timer](
      algebras: Algebras[F],
      programs: Programs[F],
      security: Security[F]
  ): F[HttpApi[F]] =
    Sync[F].delay(
      new HttpApi[F](
        algebras,
        programs,
        security
      )
    )
}
```

Simple and without much ceremony. Next, let's look at its implementation.

```
final class HttpApi[F[_]: Concurrent: Timer] private (
    algebras: Algebras[F],
    programs: Programs[F],
    security: Security[F]
) {

  private val adminAuth:
    JwtToken => JwtClaim => F[Option[AdminUser]] =
      t => c => security.adminAuth.findUser(t)(c)
  private val usersAuth:
    JwtToken => JwtClaim => F[Option[CommonUser]] =
      t => c => security.usersAuth.findUser(t)(c)
```

```
private val adminMiddleware =
  JwtAuthMiddleware[F, AdminUser](
    security.adminJwtAuth.value, adminAuth
  )
private val usersMiddleware =
  JwtAuthMiddleware[F, CommonUser](
    security.userJwtAuth.value, usersAuth
  )

// Auth routes
private val loginRoutes  =
  new LoginRoutes[F](security.auth).routes
private val logoutRoutes =
  new LogoutRoutes[F](security.auth).routes(usersMiddleware)
private val userRoutes =
  new UserRoutes[F](security.auth).routes

// Open routes
private val healthRoutes =
  new HealthRoutes[F](algebras.healthCheck).routes
private val brandRoutes =
  new BrandRoutes[F](algebras.brands).routes
private val categoryRoutes =
  new CategoryRoutes[F](algebras.categories).routes
private val itemRoutes =
  new ItemRoutes[F](algebras.items).routes

// Secured routes
private val cartRoutes =
  new CartRoutes[F](algebras.cart).routes(usersMiddleware)
private val checkoutRoutes =
  new CheckoutRoutes[F](programs.checkout).routes(usersMiddleware)
private val orderRoutes =
  new OrderRoutes[F](algebras.orders).routes(usersMiddleware)

// Admin routes
private val adminBrandRoutes =
  new AdminBrandRoutes[F](algebras.brands).routes(adminMiddleware)
private val adminCategoryRoutes =
  new AdminCategoryRoutes[F](algebras.categories).routes(adminMiddleware)
private val adminItemRoutes  =
  new AdminItemRoutes[F](algebras.items).routes(adminMiddleware)
```

```
  // Combining all the http routes
  private val openRoutes: HttpRoutes[F] =
    healthRoutes <+> itemRoutes <+> brandRoutes <+>
      categoryRoutes <+> loginRoutes <+> userRoutes <+>
      logoutRoutes <+> cartRoutes <+> orderRoutes <+>
      checkoutRoutes

  private val adminRoutes: HttpRoutes[F] =
    adminItemRoutes <+> adminBrandRoutes <+> adminCategoryRoutes

  private val routes: HttpRoutes[F] = Router(
    version.v1 -> openRoutes,
    version.v1 + "/admin" -> adminRoutes
  )

  private val middleware: HttpRoutes[F] => HttpRoutes[F] = {
    { http: HttpRoutes[F] =>
      AutoSlash(http)
    } andThen { http: HttpRoutes[F] =>
      CORS(http, CORS.DefaultCORSConfig)
    } andThen { http: HttpRoutes[F] =>
      Timeout(60.seconds)(http)
    }
  }

  private val loggers: HttpApp[F] => HttpApp[F] = {
    { http: HttpApp[F] =>
      RequestLogger.httpApp(true, true)(http)
    } andThen { http: HttpApp[F] =>
      ResponseLogger.httpApp(true, true)(http)
    }
  }

  val httpApp: HttpApp[F] = loggers(middleware(routes).orNotFound)

}
```

Step by step, this is what is happening:

- We define the `authenticate` function for both users, required by our `JWTAuthMiddleware`.
- We define two `JWTAuthMiddleware`, one for each kind of user.
- Next, we define all our HTTP routes, both the open and the secured.
- A `Router` lets us add the `admin` prefix to all our Admin routes.

- Next, we define all our middlewares, including our loggers.
- Finally, we create our `HttpApp[F]` by composing middlewares and routes.

This is all. We managed to group all the relevant functionality in distinct modules. Now it is time to talk about resources.

Resources

Some of our interpreters and modules take either a `RedisCommands` or a `Resource[F, Session[F]]`, or both. Some other components might need an HTTP Client as well. These resources must be created once and shared with the respective components that need to make use of them (shared state).

We are going to create all the resources of our application in a single place, for what we will define the following type:

```
final case class AppResources[F[_]](
    client: Client[F],
    psql: Resource[F, Session[F]],
    redis: RedisCommands[F, String, String]
)
```

These are the resources we have identified in our application. Now, we need to provide a smart constructor to create and compose all of them.

```
object AppResources {

  def make[F[_]: ConcurrentEffect: ContextShift: Logger](
      cfg: AppConfig
  ): Resource[F, AppResources[F]] = {

    def mkPostgreSqlResource(
        c: PostgreSQLConfig
    ): SessionPool[F] =
      Session
        .pooled[F](
          host = c.host.value,
          port = c.port.value,
          user = c.user.value,
          database = c.database.value,
          max = c.max.value
        )

    def mkRedisResource(
        c: RedisConfig
    ): Resource[F, RedisCommands[F, String, String]] =
      for {
        uri <- Resource.liftF(RedisURI.make[F](c.uri.value.value))
        client <- RedisClient[F](uri)
        cmd <- Redis[F, String, String](client, RedisCodec.Utf8)
      } yield cmd
```

```
    def mkHttpClient(
        c: HttpClientConfig
    ): Resource[F, Client[F]] =
      BlazeClientBuilder[F](ExecutionContext.global)
        .withConnectTimeout(c.connectTimeout)
        .withRequestTimeout(c.requestTimeout)
        .resource

    (
      mkHttpClient(cfg.httpClientConfig),
      mkPostgreSqlResource(cfg.postgreSQL),
      mkRedisResource(cfg.redis)
    ).mapN(AppResources.apply[F])

  }

}
```

`Resource` forms a `Monad`, and thus an `Applicative`, so we can take advantage of this property and compose all our different resources into a single one using the `mapN` function.

Since Cats Effect 2.1.0, it is also possible to compose resources in parallel, using functions such as `parMapN`.

Main

Our main entry point extends `IOApp`, provided by Cats Effect.

```
object Main extends IOApp {

  implicit val logger = Slf4jLogger.getLogger[IO]

  override def run(args: List[String]): IO[ExitCode] =
    config.load[IO].flatMap { cfg =>
      Logger[IO].info(s"Loaded config $cfg") *>
        AppResources.make[IO](cfg).use { res =>
          for {
            sec <- Security.make[IO](cfg, res.psql, res.redis)
            alg <- Algebras.make[IO](
                     res.redis, res.psql, cfg.cartExp
                   )
            cli <- HttpClients.make[IO](
                     cfg.paymentConfig, res.client
                   )
            pro <- Programs.make[IO](cfg.checkoutConfig, alg, cli)
            api <- HttpApi.make[IO](alg, pro, sec)
            _ <- BlazeServerBuilder[IO]
                   .bindHttp(
                     cfg.httpServerConfig.port.value,
                     cfg.httpServerConfig.host.value
                   )
                   .withHttpApp(api.httpApp)
                   .serve
                   .compile
                   .drain
          } yield ExitCode.Success
        }
    }

}
```

We first create a default logger. Then, we load the configuration and create the application resources.

Next, we create our `HttpApi` by instantiating all our modules via their smart constructors, and finally, we start our HTTP server by choosing the default `Blaze` implementation.

There are a few ways of composing our main application. Another popular choice is to use `Resource` for everything, in which case, we can use the `resource` method on our

server builder instead of `serve`, which returns a stream.

In fairness, that is all! We are now ready to start up our server. In an SBT session, we can run the following command, which uses the `sbt-revolver` plugin:

```
sbt:shopping-cart> reStart
```

Assuming we have all our environment variables set up, and both our PostgreSQL and Redis instances are running, we should see something like this when running it within `project core` (output has been trimmed for readability):

```
sbt:core> reStart
[info] Application core not yet started
[info] Starting application core in the background ...
core Starting shop.Main.main()
[success] Total time: 0 s, completed Nov 22, 2019 10:25:45 AM
core [ioapp-compute-0] INFO  shop.Main - Loaded config (...)
core [ioapp-compute-0] INFO  io.lettuce.core.EpollProvider (...)
core [ioapp-compute-0] INFO  io.lettuce.core.KqueueProvider (...)
core [ioapp-compute-1] INFO  o.h.b.c.nio1.NIO1SocketServerGroup -
       Service bound to address /0:0:0:0:0:0:0:0:8080
core [ioapp-compute-1] INFO  o.h.server.blaze.BlazeServerBuilder -
core   _   _   _        _ _
core  | |_| |_| |_ _ __| | | ___
core  | ' \  _|  _| '_ \_  _(_-<
core  |_||_\__|\__| .__/ |_|/__/
core              |_|
core [ioapp-compute-1] INFO  o.h.server.blaze.BlazeServerBuilder -
       http4s v0.21.0-M5 on blaze v0.14.8 started at http://[::]:8080/
```

To stop the server, run the following command:

```
sbt:shopping-cart> reStop
```

Source code

The source code of the Shopping Cart application can be found here[17].

[17] https://github.com/gvolpe/pfps-shopping-cart

Chapter 9: Deploying

We are almost done with our application. It is already tested and serving requests. Now we need to deploy it in a real environment where it can run with high uptime.

There are many ways to deploy an application. We could create a *fat jar*, a *war*, or even a simple *binary* file. Once we have this, we can run it using either `java` or `bash` in our production server, for example. All of these methodologies come with pros and cons.

Nowadays, most environments are virtual machines running in our physical computer, or more likely, in the *cloud*, using services such as AWS, GCP, and Azure.

While talking about virtualized environments, I must mention *Docker* and *Kubernetes*. Docker allows us to pack our application and its dependencies in a single container that can be shared and deployed into any environment. Kubernetes lets us orchestrate different containers.

Given the simplicity of Docker and the exceptional support for it in Scala, we are going to choose it for our application. If you use Kubernetes, you are expected to understand what you can do with a Docker container. This book does not aim to explain this topic.

We are now going to focus on creating and shipping our application as a Docker image.

Docker image

Quoting the official Docker documentation[1]:

> A Docker image is a read-only template with instructions for creating a Docker container. A container is a runnable instance of an image.

The easiest way to create a Docker image of a Scala application is by using the SBT Native Packager[2] plugin. It not only supports Docker but also `zip` and `tar.gz` files, `deb` and `rpm` packages for Debian/RHEL based systems, and GraalVM native images, among others.

First, we need to add it into our `plugins.sbt` file.

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % VERSION)
```

Notes

`Replace VERSION with the current version of the plugin`

Next, we need to enable the Docker plugin in our project.

```
lazy val core = (project in file("modules/core"))
  .enablePlugins(DockerPlugin)
  .settings(
    name := "shopping-cart-core",
    packageName in Docker := "shopping-cart",
    dockerExposedPorts ++= Seq(8080),
    dockerUpdateLatest := true,
    // more settings here
  )
```

Here we can configure the exposed port (used to access our HTTP server), the name of our Docker image, and many other settings that are detailed in the documentation.

To create our Docker image, run the following command:

```
sbt docker:publishLocal
```

It will create a Docker image named `shopping-cart` and publish it into the local Docker server. We can verify its existence as follows:

[1] https://docs.docker.com/engine/docker-overview/
[2] https://github.com/sbt/sbt-native-packager

```
> docker images | grep shopping-cart
REPOSITORY      TAG      IMAGE ID       CREATED      SIZE
shopping-cart   latest   e836c97e5673   1 hour ago   541MB
```

It seems we are done here. However, the size of the image should have caught your attention. Let's see what we can do to reduce its size.

Optimizing image

At the time of writing, the latest version of the SBT Native Packager plugin (1.5.2) uses `openjdk:8` as the base Docker image by default. Though we can specify one, and this will be our first and most important optimization: we do not need the JDK (Java Development Kit) to run our application. We only need the JRE (Java Runtime Environment).

If you do some research, you will find that there are many images we could use. We will choose an Alpine image, which is rather small and well tested.

In our settings, we need to add the following custom image:

```
.settings(
  dockerBaseImage := "openjdk:8u201-jre-alpine3.9",
)
```

This greatly reduces the size of our image. However, there is a problem. Our generated script is Bash-specific, which is not compatible with the Ash[3] shell used by Docker, and thus making it impossible to run our application.

Luckily, this issue goes away by enabling the Ash plugin, which tells our package manager to generate our binary using Ash instead of Bash.

```
.enablePlugins(AshScriptPlugin)
```

There is one last optimization we can make. By default, both Linux and Windows scripts will be created, and considering we are going to run our application in a Linux environment, we can tell the plugin to skip the creation of the `bat` script file.

```
.settings(
  makeBatScripts := Seq(),
)
```

We can now verify the new size of our image.

[3] https://en.wikipedia.org/wiki/Almquist_shell

```
> docker images | grep shopping-cart
REPOSITORY      TAG      IMAGE ID       CREATED      SIZE
shopping-cart   latest   646501a87362   1 hour ago   138MB
```

We have reduced the size of our image almost four times! We will stop here and move on, but you are invited to investigate deeper and see if you can further optimize it.

Run it locally

Having a Docker image, we can now try to run it in our local machine to verify it works as expected. Assuming a PostgreSQL and Redis services running in our same network, we can create the following `docker-compose.yml` file:

```
version: '3.4'
services:
  shopping_cart:
    restart: always
    image: shopping-cart:latest
    network_mode: host
    ports:
      - "8080:8080"
    environment:
      - DEBUG=false
      - SC_ACCESS_TOKEN_SECRET_KEY=YourToken
      - SC_JWT_SECRET_KEY=YourSecret
      - SC_JWT_CLAIM=YourClaim
      - SC_ADMIN_USER_TOKEN=YourAdminToken
      - SC_PASSWORD_SALT=YourEncryptionKey
      - SC_APP_ENV=test
```

Followed by this simple command to start it up:

```
> docker-compose up
Creating app_shopping_cart_1 ... done
Attaching to app_shopping_cart_1
... more logs here ...
```

We should see our HTTP server starting up as usual.

Continuous Integration

A Continuous Integration (CI) build is almost mandatory these days. It automates the build to keep us from deploying broken code. Among the most popular services are Travis CI, Circle CI, and recently, Github Actions has joined the club. All of these services are good and free for open-source software (OSS).

Github Actions is coming in strong, and it has proven to be fast and work well, so this will be the service we are going to use for our Shopping Cart application.

Dependencies

Our application needs both PostgreSQL and Redis in order to run. We are going to provide a `docker-compose` file to make this easy.

```
PostgreSQL:
  restart: always
  image: postgres:12.0-alpine
  ports:
    - "5432:5432"
  environment:
    - DEBUG=false
    - POSTGRES_DB=store
  volumes:
    - ./tables.sql:/docker-entrypoint-initdb.d/init.sql

Redis:
  restart: always
  image: redis:5.0.0
  ports:
    - "6379:6379"
  environment:
    - DEBUG=false
```

Our first service is `PostgreSQL`. It is almost self-explanatory, except for `volumes`. The `tables.sql` is a SQL script that describes the structure of our database, and it allows Docker to create the necessary tables on start-up. For more details, please look at the source code.

Our second service is `Redis`, which is much simpler than the previous one. Next, we can start and stop our services.

To start both services:

```
> docker-compose up
Creating pfps-shopping-cart_PostgreSQL_1 ... done
Creating pfps-shopping-cart_Redis_1      ... done
... more logs here ...
```

To stop both services:

```
> docker-compose down
Stopping pfps-shopping-cart_Redis_1      ... done
Stopping pfps-shopping-cart_PostgreSQL_1 ... done
Removing pfps-shopping-cart_Redis_1      ... done
Removing pfps-shopping-cart_PostgreSQL_1 ... done
```

CI build

Now that we have our dependencies defined in a `docker-compose.yml` file, we can continue with the configuration of our CI build using Github Actions.

In a `.github/workflows/ci.yml` file, we are going to have the following content:

```yaml
name: Build

on:
  push:
    branches:
      - master
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - uses: olafurpg/setup-scala@v5
    - name: Starting up Postgres and Redis
      run: docker-compose up -d
    - name: Tests
      run: sbt -J-Xms4G -J-Xmx4G test it:test
    - name: Shutting down Postgres and Redis
      run: docker-compose down
```

The build will be triggered either when some code is pushed to `master`, or when there is a new pull request.

Our build job involves checking out the code from the Git repository, setting up the Scala tools, starting PostgreSQL and Redis, running both unit and integration tests, and finally, stopping our services.

As an optimization, we could also add caching support, as it has been done in our application.

Regardless, this is all we need to have a continuous integration build set up.

Furthermore

We now have a Docker image and a CI build set up. Next step is to deploy our application into our production environment. Depending on your resources, this may vary.

It is worth mentioning that, by using Github Actions, we should be able to configure a Continuous Deployment (CD) build. This way, we would have a fully automated deployment pipeline.

For now, though, we will stop here to avoid getting too much into DevOps land, which is not my area of expertise.

Summary

This is the end of our Shopping Cart application development. You should now be ready to dive into new endeavors and apply the techniques learned in this book.

The Gitter channel[4] of the book will remain active, so feel free to join and ask questions, either about the book's topics or functional programming in general.

Once again, thanks to all of you who have supported my work. It has been a real pleasure, and I hope this book helps you even with the smallest task at hand.

Writing a book is not easy. Writing your first book in your second language (English) is tough. Yet, you all made it real, and I am delighted with the final result.

Eternally thankful, Gabriel.

[4] https://gitter.im/pfp-scala/community

Chapter 10: Advanced techniques

Take in this chapter as a bonus track. Although our Shopping Cart application is complete, there are a few changes we could still make, using advanced features.

In this chapter, we are going to explore some cutting-edge techniques and libraries in the functional Scala ecosystem. We will also modify our application to incorporate these new concepts, leaving them in a new Git branch.

Tagless Final plugin

Before we start, I would like to briefly introduce a *Tagless-Final-friendly compiler plugin* named Context Applied[1]. I would probably have used this plugin from the beginning but it came out recently, by the time I am almost done with the book. So instead of rewriting everything, we will make use of it in the following examples and you can decide whether you would like to use it or not. I definitely recommend it because I believe it makes grasping effect constraints and capabilities much easier.

See the following code snippet:

```
def program[F[_]: Background: Console: FlatMap: Timer]: F[Unit] =
  Background[F].schedule(
    Console[F].putStrLn("foo"), 3.seconds
  ) » Timer[F].sleep(1.second) » program
```

Notice how we needed to summon every typeclass instance manually, using their *apply* summoner methods, to access their functionality. Even though the constraints are added to our same type `F[_]`, we need to either summon the instances in this way, or give up the use of context-bound constraints and instead use implicit values as below.

```
def program[F[_]](
    implicit B: Background,
    C: Console,
    F: FlatMap,
    T: Timer
): F[Unit] =
  B.schedule(C.putStrLn("foo"), 3.seconds) »
    T.sleep(1.second) » program
```

This is fine, yet quite verbose, and my preference for context-bound constraints has been made clear in a blog post[2] I wrote some time ago. Describing their differences in this book is out of the scope, though.

If we use the Context Applied plugin, we get access to a variable that has all the capabilities indicated by its constraints. Thus, we can rewrite this program as follows:

```
def program[F[_]: Background: Console: FlatMap: Timer]: F[Unit] =
  F.schedule(F.putStrLn("foo"), 3.seconds) »
    F.sleep(1.second) » program
```

[1] https://github.com/augustjune/context-applied
[2] https://gvolpe.github.io/blog/context-bound-vs-implicit-evidence/

The `F` variable, matching the letter of our type constructor `F[_]`, has been introduced by this plugin, which gives us the *union of the capabilities* offered by each typeclass. It also improves type inference; it is a win-win.

On the other hand, it introduces a new idiom in an already complex language, reason why some people discourage its use.

MTL (Monad Transformers Library)

MTL stands for Monad Transformers Library. Its name comes from Haskell's MTL[3].

It encodes effects as typeclasses instead of using data structures (as Monad Transformers do), from which we can deduce the MTL name is a historical accident at this point.

In the Scala ecosystem, there exists Cats MTL[4], which holds quite some differences with its Haskell counterpart. The main one being having more granular typeclasses such as `ApplicativeAsk` instead of `MonadReader`.

Cats MTL's documentation is remarkable, so rather than repeating what is already there, we are going to focus on two specific effects: `MonadState` and `ApplicativeAsk`.

Managing state

`MonadState` lets us manage state. Here is its definition:

```
trait MonadState[F[_], S] {
  val monad: Monad[F]

  def get: F[S]
  def set(s: S): F[Unit]
  def inspect[A](f: S => A): F[A]
  def modify(f: S => S): F[Unit]
}
```

It favors composition over inheritance to avoid the ambiguous implicits issue.

Effects are encoded as typeclasses. Therefore, to make use of `MonadState`, we need to add it as a constraint to our `F[_]`.

```
def program[F[_]: Console: MonadState[*[_], FooState]]: F[Unit] =
  for {
    current <- F.get
    _ <- F.putStrLn(current)
    _ <- F.set(FooState("foo"))
    updated <- F.get
    _ <- F.putStrLn(updated)
  } yield ()
```

Usually, polymorphic MTL style programs are materialized using Monad Transformers, and `StateT` would be our most natural choice here.

[3] https://hackage.haskell.org/package/mtl
[4] https://typelevel.org/cats-mtl/

```
val p1: IO[Unit] =
  program[StateT[IO, FooState, *]]
    .run(FooState("mt"))
    .void
```

Though, for performance and ergonomic reasons, we could use a Ref-backed instance provided by Meow MTL[5].

```
import com.olegpy.meow.effects._

val p2: IO[Unit] =
  Ref.of[IO, FooState](FooState("bar"))
    .flatMap { ref =>
      ref.runState { implicit ms =>
        program[IO]
      }
    }
```

Such instances are commonly acquired effectfully; use with caution.

Another reason to prefer the latter is that `Ref` supports concurrency, unlike `StateT`, which is inherently sequential (see Chapter 2).

Tips

`Prefer interfaces over dealing with raw state directly`

`MonadState` allows us to manage stateful programs across different functions. Though, as we have seen in Chapter 2, state is better managed when encapsulated behind interfaces.

Accessing context

`ApplicativeAsk` lets us access some context, sometimes called an *environment*. It is defined as follows:

```
trait ApplicativeAsk[F[_], E] {
  val applicative: Applicative[F]

  def ask: F[E]
  def reader[A](f: E => A): F[A]
}
```

It also favors composition over inheritance for the same reasons.

[5] https://github.com/oleg-py/meow-mtl

- `ask` allows us to access the context.
- `reader` is a shortcut for `ask.map(f)`.

Let's see an example. First, we need some datatypes to represent a context.

```
final case class Foo(value: String)
final case class Bar(value: Int)
final case class Ctx(foo: Foo, bar: Bar)
```

Next, we define a few handy type aliases.

```
type HasFoo[F[_]] = ApplicativeAsk[F, Foo]
type HasBar[F[_]] = ApplicativeAsk[F, Bar]
type HasCtx[F[_]] = ApplicativeAsk[F, Ctx]
```

With all this in place, we can write functions as follows:

```
def p1[F[_]: Console: FlatMap: HasCtx]: F[Unit] =
  F.ask.flatMap(ctx => F.putStrLn(ctx))
```

It accesses the current context, and it prints it out to the console.

We can now materialize our program using `Kleisli` (also known as `ReaderT`).

```
val ctx = Ctx(Foo("foo"), Bar(123))

p1[Kleisli[IO, Ctx, *]].run(ctx) // IO[Unit]
```

We could also materialize it using `IO` directly, but it would require us to either write a rather *hacky* `ApplicativeAsk[IO, Ctx]` instance, or to use a Ref-backed instance provided by Meow MTL.

Both ways of acquiring such instance are effectful, reason why this technique is discouraged by purists, as something alike wouldn't be possible in language with global coherence of typeclasses like Haskell.

Usually, we do it anyway, having an understanding of its trade-offs. If we were to use Monad Transformers instead, we would be introducing more boilerplate (type inference tends to be limited), as well as a performance penalty (nested bind calls).

```
Ref.of[IO, Ctx](ctx).flatMap { ref =>
  ref.runAsk { implicit ioCtxAsk =>
    p1[IO]
  }
}
```

Let's now look at a program that wouldn't compile.

```
def p2[F[_]: Console: FlatMap: HasFoo]: F[Unit] =
  F.ask.flatMap(foo => F.putStrLn(foo))

def p3[F[_]: Console: FlatMap: HasBar]: F[Unit] =
  F.ask.flatMap(bar => F.putStrLn(bar))

// could not find implicit value for evidence
// parameter of type HasFoo[IO]
p2[IO]
```

We have an instance of `HasCtx[F]` but not of `HasFoo[F]` and `HasBar[F]`. However, this issue can be easily solved using *classy optics*.

Classy optics

First of all, what are optics? In a nutshell, optics are a first-class composable functional abstraction that lets us manipulate data structures.

In Scala, there is a great library named Monocle[6] that defines the entire hierarchy of optics, as well as defining algebraic laws for such types.

If I am not mistaken, the word *classy* comes from the `makeClassy` function defined in Haskell's Lens package[7]. There is another function called `makeLenses`, which generates lenses for a given type. The classy variant does the same but, it additionally creates a typeclass and an instance of that typeclass, along with some lenses.

There is also another function `makeClassPrisms`, which does the same but for prisms.

So it can be said we have classy optics when we can associate with each type a typeclass full of optics for that type.

Optics are a gigantic topic, though, and one can probably write a book about it (in fact, someone has recently done it[8]). For this reason, we are only going to discuss what is relevant for the examples ahead: lenses and prisms.

To learn more, I recommend reading the Optics' documentation[9].

Lenses

Lenses provide first-class access for *product types*. This means we can zoom-in into case classes, for example, which are product types. We can also think of lenses as a pair of *getter* and *setter* functions.

Given an instance of `Person`, we could access and modify the `StreetName`.

```
case class Address(
  streetName: StreetName,
  streetNumber: StreetNumber
)

case class Person(
  name: PersonName,
  age: PersonAge,
  address: Address
)
```

[6] https://github.com/julien-truffaut/Monocle
[7] https://hackage.haskell.org/package/lens
[8] https://leanpub.com/optics-by-example
[9] https://hackage.haskell.org/package/optics-0.1/docs/Optics.html

```
person.copy(
  address = person.address.copy(
    streetName = StreetName("new st")
  )
)
```

By using the native `copy` method, we can get away with the task at hand, but we can see where this is going. Not only is it cumbersome; it doesn't compose either.

We could use lenses instead, which can be composed with other optics. This is how we define lenses using Monocle:

```
val addressLens: Lens[Person, Address] =
  Lens[Person, Address](_.address)(a => p => p.copy(address = a))

val streetNameLens: Lens[Address, StreetName] =
  Lens[Address, StreetName](_.streetName)(
    s => a => a.copy(streetName = s)
  )
```

If this seems too verbose, Monocle provides a macros module that enables a simpler syntax.

```
val streetNameLens: Lens[Address, StreetName] =
  GenLens[Address](_.streetName)

// or using annotations
@lenses
case class Address(
  streetName: StreetName,
  streetNumber: StreetNumber
)
```

Now that we have defined these lenses, we can make use of them to modify the street name of a given address.

```
// returns new Address
streetNameLens.set(StreetName("foo"))(address)
```

If we wanted to modify the street name of the address of a person, we would need a `Lens[Person, StreetName]`. Instead of defining this lens manually, we can use composition to avoid code repetition.

Composing lenses

Lenses, and optics in general, are highly composable. We can reuse the lenses we already have to create the lens we need.

```
val mixLens: Lens[Person, StreetName] =
  addressLens.composeLens(streetNameLens)
```

With this new lens, we can access the current street name as well as modifying it, as demonstrated below.

```
// returns current street name
mixLens.get(person)

// returns a Person with a new address
mixLens.set(StreetName("foo"))(person)
```

If we wanted to zoom-in into multiple values, we would need a `Traversal` instead.

Prisms

Prisms provide first-class access for *sum types*, or also called *co-product types*. A prism allows us to select a single branch of a sum type, e.g. `Option`, `Either`, or any other ADT.

Below we can see an example using Monocle's prisms to manipulate an ADT.

```
sealed trait Vehicle
case object Car extends Vehicle
case object Boat extends Vehicle

type Car = Car.type

val vPrism: Prism[Vehicle, Car] =
  Prism.partial[Vehicle, Car] { case c: Car => c }(identity)

vPrism(Car) // Car
vPrism.getOption(Boat) // None
vPrism.getOption(Car) // Some(Car)
```

Given an instance of `Vehicle`, a prism would let us to access either branch. Thus, we can say a prism can traverse a tree-like data structure and select a branch.

We can think of prisms as an abstraction that lets us zoom-in to a part of a value that may not be there, therefore, returning an optional value.

Composing prisms

Prisms also compose. Let's look at the following example:

```
val ps: Prism[Option[String], String] =
  Prism.partial[Option[String], String] {
    case Some(v) => v
  }(Option.apply)

val pi: Prism[String, Int] =
  Prism.partial[String, Int] {
    case v if v.toIntOption.isDefined => v.toInt
  }(_.toString)

ps.composePrism(pi) // Prism[Option[String], Int]
```

As previously mentioned, optics in general are highly composable. Not only we can compose prisms with prisms, but also prisms with lenses, traversals, folds, and more.

Another compelling example where prisms shine is the same as lenses: accessors and modifiers for case classes. Wait, case classes are product types, so you might be wondering how can prisms help here? See the example below.

```
@newtype case class AlbumName(value: String)
case class Album(name: AlbumName, year: Int)
case class Song(name: String, album: Option[Album])

val albumNameLens: Lens[Album, AlbumName] =
  GenLens[Album](_.name)
val songAlbumLens: Lens[Song, Option[Album]] =
  GenLens[Song](_.album)
```

We have a product type `Song` that also contains an optional field `Album`, which forms a sum type. In such cases, we can compose lenses and prisms to fulfill our needs.

```
val songAlbumNameOpt: Optional[Song, AlbumName] =
  songAlbumLens.composePrism(some).composeLens(albumNameLens)

val album = Album(AlbumName("Peluso of Milk"), 1991)
val song1 = Song("Ganges", Some(album))
val song2 = Song("State of unconsciousness", None)

songAlbumNameOpt.getOption(song1) // Some(Peluso of Milk)
songAlbumNameOpt.getOption(song2) // None
```

Notice the usage of `some`, which is a predefined prism that allows us to compose lenses of optional types.

The composition of a `Lens` and a `Prism` yields an `Optional`, which sits right between them in the optics hierarchy. You can learn more about it in Monocle's documentation.

Classy lenses

Sometimes we can be more precise when referring to classy optics. For instance, if we only have lenses, we say *classy lenses*; if we only have prisms, we say *classy prisms*, and so on.

Back to our `ApplicativeAsk` example that doesn't compile, let's see how classy lenses can make it work.

```
def p2[F[_]: Console: FlatMap: HasFoo]: F[Unit] =
  F.ask.flatMap(foo => F.putStrLn(foo))

def p3[F[_]: Console: FlatMap: HasBar]: F[Unit] =
  F.ask.flatMap(bar => F.putStrLn(bar))

def program[F[_]: Console: FlatMap: HasCtx]: F[Unit] =
  p2[F] » p3[F] » F.putStrLn("Done")

program[IO] // does not compile
```

Using Meow MTL, we are a single import away to get this compiling.

```
import com.olegpy.meow.hierarchy._

program[IO] // yay!
```

So, how does this work? In a nutshell, this is what Meow MTL does under the hood:

1. It identifies the existence of an `ApplicativeAsk[F, Ctx]` instance.
2. It proceeds to create lenses for the product type `Ctx` and its content `Foo` and `Bar`.
3. It derives `ApplicativeAsk[F, Foo]` and `ApplicativeAsk[F, Bar]` from the root instance.

This is an automated process. Alternatively, we could use its low-level API and define these classy lenses manually.

```
case class User(name: String)
type HasUser[A] = MkLensToType[A, User]

def userLens[A: HasUser]: Lens[A, User] = A.apply()
```

```
def userName[A: HasUser](a: A) =
  userLens[A].get(a).name

case class Ctx(user: User, id: String)

userName(Ctx(User("Oleg"), "0x42")) // Oleg
```

It is worth mentioning that the lenses defined by Meow MTL are Shapeless' lenses, and the prisms are custom ones defined in the library. If you are looking for a library that derives classy optics using Monocle, I recommend checking out the Sbt classy plugin[10] or the Tofu library[11] that has an interop module.

Configuration

We can now try to apply this technique in our application. Let's start with resources, which require `HttpClientConfig`, `PostgreSQLConfig`, and `RedisConfig`.

Our current implementation takes in an `AppConfig`.

```
def make[F[_]: ConcurrentEffect: ContextShift: Logger](
  cfg: AppConfig
): Resource[F, AppResources[F]] = { ... }
```

This is for convenience, to avoid having three different parameters, which becomes boilerplatey. Ideally, we should share as little information as it is needed.

So let's create two handy type aliases to use `ApplicativeAsk` to access context instead of manually passing arguments.

```
type HasAppConfig[F[_]]       = ApplicativeAsk[F, AppConfig]
type HasResourcesConfig[F[_]] = ApplicativeAsk[F, ResourcesConfig]
```

Our new `ResourcesConfig` datatype is part of `AppConfig` and is defined as follows:

```
case class ResourcesConfig(
    httpClientConfig: HttpClientConfig,
    postgreSQL: PostgreSQLConfig,
    redis: RedisConfig
)
```

We can now modify our smart constructor's signature.

[10] https://github.com/cb372/sbt-classy
[11] https://github.com/TinkoffCreditSystems/tofu

```
def make[
    F[_]: ConcurrentEffect: ContextShift: HasResourcesConfig: Logger
]: Resource[F, AppResources[F]] = { ... }
```

Again, we could have used specific `Has` typeclasses instead of a single one `HasResourcesConfig` but you can imagine how ugly it gets when there are too many typeclass constraints.

Our smart constructor can now be written as follows:

```
for {
  h <- Resource.liftF(F.reader(_.httpClientConfig))
  p <- Resource.liftF(F.reader(_.postgreSQL))
  r <- Resource.liftF(F.reader(_.redis))
  client <- mkHttpClient(h)
  psql   <- mkPostgreSqlResource(p)
  redis  <- mkRedisResource(r)
} yield AppResources(client, psql, redis)
```

We need to lift our effect into the `Resource` monad to put it all together.

We also need to make the resources loading call polymorphic, due to Meow MTL not supporting derivation for concrete types.

```
import com.olegpy.meow.hierarchy._

def loadResources[
    F[_]: ConcurrentEffect: ContextShift: FlatMap: HasAppConfig: Logger
](
    fa: AppConfig => AppResources[F] => F[ExitCode]
): F[ExitCode] =
  F.ask.flatMap { cfg =>
    F.info(s"Loaded config $cfg") »
      AppResources.make[F].use(res => fa(cfg)(res))
  }
```

Last but not least, we need an `ApplicativeAsk[IO, AppConfig]` instance to get this working using `IO`; this is where it gets polemic.

When using MTL typeclasses, the right approach is to instantiate our functions using Monad Transformers. In this case, it would be `Kleisli[IO, AppConfig, *]`.

Though, we can either create a custom effectful instance of `ApplicativeAsk[IO, A]` or resort to Meow MTL to create a Ref-backed instance for us, as we have explored with `MonadState`. Let's pick the latter.

Warning

For performance reasons, it is fine to choose an effectful instance

Meow MTL can give us an `ApplicativeAsk[IO, A]` instance if we have a `Ref` holding the context `A`. So let's define a function that loads the configuration and creates a mutable reference with our `AppConfig`.

```
val configLoader: IO[Ref[IO, AppConfig]] =
  config.load[IO].flatMap(Ref.of[IO, AppConfig])
```

We can now modify our entry point as follows:

```
import com.olegpy.meow.effects._

override def run(args: List[String]): IO[ExitCode] =
  configLoader.flatMap(_.runAsk { implicit ioAsk =>
    loadResources[IO] { cfg => res =>
       restOfTheProgram
    }
  })
```

Once again, Meow MTL helped us gain in performance and ergonomics.

What are the benefits?

In this tiny example, we have seen `ApplicativeAsk`'s usage, but what are the benefits of using it compared to plain function arguments? This is a great question. In fairness, one can choose either approach and it would be fine.

Using plain arguments, a program may look as follows:

```
def program[F[_]: Concurrent]: F[Unit] =
  makeContext[F].flatMap { ctx =>
    p1(ctx)
  }

def p1[F[_]: FlatMap](ctx: AppCtx): F[Unit] =
  p2(ctx.foo) » p3(ctx.bar)

def p2[F[_]: FlatMap](foo: Foo): F[Unit] =
  p4(foo.t1) » p5(foo.t2)

def p3[F[_]: FlatMap](bar: Bar): F[Unit] =
  p6(bar.t1) » p7(bar.t2)
```

Every small program takes in the piece of context it needs and nothing more, potentially hiding delicate information from other functions. It does get a bit cumbersome, though, especially when we are talking about a medium to big size application. However, since records are great in Scala (case classes with dot notation), we can argue it is an acceptable approach.

On the other hand, using a typeclass constraint to access some context, free us from manually threading arguments in every layer. All we need is to provide this context at the top layer, which usually is our main entry point to the application.

```
def program[F[_]: Concurrent]: F[Unit] =
  makeContext[F].flatMap { ctx =>
    p1[Kleisli[F, Ctx, *]].run(ctx)
  }

def p1[F[_]: FlatMap: HasAppCtx]: F[Unit] = p2 » p3

def p2[F[_]: FlatMap: HasFoo]: F[Unit] = p4 » p5

def p3[F[_]: FlatMap: HasBar]: F[Unit] = p6 » p7
```

Notice how adopting `ApplicativeAsk` and classy lenses grant us access to the piece of context we need (as we do it passing arguments) while drastically reducing the amount of code we need to write. Allegedly, the number of typeclass constraints may get much bigger, except we learned how to organize our code into modules to ease the complexity.

That is all on `ApplicativeAsk`. Still, we could extend its usage in our application, but this task is left as a challenge to the reader.

The modifications presented here can be found in the `feature/classy-optics` branch.

Classy prisms

Meow MTL can also derive instances of `MonadError[F, A]`, given `A` forms a co-product type. The most common scenario is when employing a custom error, which is a subtype of `Throwable`, to make it compatible with Cats Effect.

Say we have the following ADT of errors:

```
sealed trait UserError extends NoStackTrace
case object UserNotFound extends UserError
case object UserExists extends UserError
```

In Scala, we represent sum types using subtyping, which might be confusing to those coming from other languages with first-class support for them.

For instance, this is how we can write it in Haskell:

```haskell
data UserError = UserNotFound | UserExists
```

In OCaml, where sum types are also called *variants*, we can can write it as follows:

```ocaml
type userError =
  | UserNotFound
  | UserExists
```

However, don't let this distract you from the main subject: sum types and prisms.

Previously, we saw that prisms allow us to select a single branch of a sum type. Taking the current example, it would be either `UserNotFound` or `UserExists`.

However, in the Scala implementation, a value of type `Throwable` can be a lot of things. Subtyping allows a generic type like `Throwable` to be replaced by a more granular type such as `Exception` or `UserError`, for example.

This is why Meow MTL can derive a `MonadError[F, MyError]`, given that `MyError <: Throwable` and there is an instance of `MonadError[F, Throwable]` in scope.

For example, the program below wouldn't compile without the help of this library.

```scala
def program[F[_]: MonadError[*[_], UserError], A](
    fa: F[A],
    fallback: A
): F[A] =
  fa.handleError(_ => fallback)

val ioa: IO[Int] = IO.raiseError(UserExists)
program(ioa, 123)
```

This is what Meow MTL is doing behind the scenes:

1. It identifies the existence of a `MonadError[IO, Throwable]` instance.
2. It proves `MyError` is a subtype of `Exception`, which is a subtype of `Throwable`.
3. It proceeds to create prisms for the sum type `MyError`.
4. It derives `MonadError[IO, MyError]` from the root instance (`Throwable`).

We could create these prisms and derive such instances manually, as we have done with classy lenses, in a very similar fashion.

Error Handling

In Chapter 2, we have explored an error handling technique using `MonadError` and classy prisms. Let's now try to modify our `CheckoutRoutes` to make use of it.

Right now, our main `POST` method looks as follows:

```
case ar @ POST -> Root as user =>
  ar.req.decodeR[Card] { card =>
    program
      .checkout(user.value.id, card)
      .flatMap(Created(_))
      .recoverWith {
        case CartNotFound(userId) =>
          NotFound(s"Cart not found for user: ${userId.value}")
        case EmptyCartError =>
          BadRequest("Shopping cart is empty!")
        case PaymentError(cause) =>
          BadRequest(cause)
        case OrderError(cause) =>
          BadRequest(cause)
      }
  }
```

Whereas the main class looks as below.

```
final class CheckoutRoutes[F[_]: Defer: MonadThrow](
    program: CheckoutProgram[F]
) extends Http4sDsl[F] { ... }
```

We can clearly see there is no relationship between the possible errors that can occur in our `CheckoutProgram`. We do not have such information. So this is our first task: we need to group the possible errors into an ADT.

```
sealed trait CheckoutError extends NoStackTrace
case class CartNotFound(userId: UserId) extends CheckoutError
case object EmptyCartError extends CheckoutError
case class OrderError(cause: String) extends CheckoutError
case class PaymentError(cause: String) extends CheckoutError
```

Using `NoStackTrace` is a good alternative to `Exception`, since stack traces are heavy-weight on the JVM and provide little benefits.

> Tips
>
> `Use NoStackTrace instead of Exception for custom error ADTs`

Next, we remove the error handling from the routes, leaving just the happy path of the request processing.

```
case ar @ POST -> Root as user =>
  ar.req.decodeR[Card] { card =>
    program
      .checkout(user.value.id, card)
      .flatMap(Created(_))
  }
```

Instead, we are going to require an instance of `HttpErrorHandler[F, CheckoutError]` to be available when creating the `HttpRoutes`.

```
def routes(
    authMiddleware: AuthMiddleware[F, CommonUser]
)(implicit H: HttpErrorHandler[F, CheckoutError]): HttpRoutes[F] =
  Router(
    prefixPath -> H.handle(authMiddleware(httpRoutes))
  )
```

Remember, this is how our `HttpErrorHandler` interface is defined:

```
trait HttpErrorHandler[F[_], E <: Throwable] {
  def handle(routes: HttpRoutes[F]): HttpRoutes[F]
}
```

Next, we need to create an error handler instance for our specific error type.

```
object CheckoutHttpErrorHandler {
  def apply[
      F[_]: MonadError[*[_], CheckoutError]
  ]: HttpErrorHandler[F, CheckoutError] =
    new RoutesHttpErrorHandler[F, CheckoutError] {
      val A = implicitly

      val handler: CheckoutError => F[Response[F]] = {
        case CartNotFound(userId) =>
          NotFound(s"Cart not found for user: ${userId.value}")
        case EmptyCartError =>
          BadRequest("Shopping cart is empty!")
        case PaymentError(cause) =>
          BadRequest(cause)
        case OrderError(cause) =>
          BadRequest(cause)
      }
```

```
    }
}
```

If you recall, we have defined a common `RoutesHttpErrorHandler` in Chapter 2. You can always go back to it or check out the source code if you don't.

We have now moved all the error handling to this specific implementation. All that is left is to create this instance at the moment of instantiating our `HttpRoutes`.

```
implicit val handler: HttpErrorHandler[F, CheckoutError] =
  CheckoutHttpErrorHandler[F]

val checkoutRoutes =
  new CheckoutRoutes[F](programs.checkout)
    .routes(usersMiddleware)
```

Two more routes can be modified to make use of this technique: `LoginRoutes` and `UserRoutes`. This challenge is left to the reader, though.

The modifications presented here can be found in the `feature/classy-optics` branch.

Typeclass derivation

In Scala, typeclasses are normally represented by `trait`s, or sometimes `abstract class`es. Below is the common definition of the `Functor` typeclass:

```
trait Functor[F[_]] {
  def map[A, B](f: A => B)(fa: F[A]): F[B]
}
```

We don't normally need to create typeclasses, as the common ones such as `Functor` and `Monad` already come with the Cats library, in addition to some instances.

However, it is very common to create new datatypes that need instances of typeclasses such as `Eq`, `Order`, and `Show`, to abide global coherence of typeclasses. In this space, we have two options: either we write the instances manually, or we derive them.

There are two great libraries capable of such a thing in Scala: Shapeless[12] and Magnolia[13]. However, these libraries are bare metal; they only provide the machinery to derive typeclasses. To get something fruitful out of it, we need some extra work.

Fortunately, other folks have gone through this path before us, so we now have a good selection of libraries:

- Derevo[14]: it uses Magnolia.
- Magnolify Cats[15]: it uses Magnolia as well.
- Kittens[16]: it uses Shapeless.
- Catnip[17]: it provides a macro-annotation for Kittens.

Libraries built upon Shapeless are usually slower to compile (at the cost of versatility) than those built on top of Magnolia, which is known to have great compile-time performance. Therefore, we are going to choose between Magnolify and Derevo.

Magnolify supports only five typeclasses from Cats, whereas Derevo provides support for more typeclasses, including higher-kinded types; it also supports derivation for Circe and Ciris, among other libraries.

So it makes sense to select Derevo, but you should do your research and determine whether it is your best choice.

Derevo provides a macro annotation to derive typeclasses for datatypes.

[12] https://github.com/milessabin/shapeless/
[13] https://github.com/propensive/magnolia
[14] https://github.com/manatki/derevo
[15] https://github.com/spotify/magnolify
[16] https://github.com/typelevel/kittens
[17] https://github.com/scalalandio/catnip

Kinds

Before we get into automated typeclass derivation, let's understand what kinds are.

Quoting the Wikipedia[18]:

> In the area of mathematical logic and computer science known as *type theory*, a kind is the type of a type constructor or, less commonly, the type of a higher-order type operator. A kind system is essentially a simply typed lambda calculus "one level up", endowed with a primitive type, denoted `*` and called "type", which is the kind of any datatype which does not need any type parameters.

Concrete types, also called *monotypes* or *nullary* type constructors, have kind `*` (also pronounced *star*). Higher-order type constructors have kinds of the form `P -> Q`, where `P` and `Q` are kinds.

However, instead of diving too much into the theory behind kinds, let's look at some examples of typeclass derivation for types of different kinds.

Concrete types

Concrete types, also expressed as `A` (and kind `*`), are types that have no holes; they are complete. Therefore, we can create instances of such types. Examples of such types include `Int`, `Option[String]`, `List[Int]`, `Boolean`, `Either[String, Int]`, etc.

We can start a REPL and ask the compiler to tell us what the kind of a given type is.

```
scala> :kind Int
Int's kind is A

scala> :kind Option[String]
Option[String]'s kind is A

scala> :kind List[Int]
List[Int]'s kind is A

scala> :kind Boolean
Boolean's kind is A

scala> :kind Either[String, Int]
Either[String,Int]'s kind is A
```

[18] https://en.wikipedia.org/wiki/Kind_(type_theory)

Derevo can easily derive instances of typeclasses that require concrete types. We can name `Eq[A]`, `Order[A]`, and `Show[A]`, among others.

Notice how all these typeclasses take a single type parameter `A`, which is of kind `*`.

The following example makes use of the `derevo-cats` dependency.

```
import derevo.cats.{ eq => eqv, _ }
import derevo.derive

@derive(eqv, order, semigroup, show)
final case class Concrete(value: String)
```

We can then make use of the power such instances bring to our type.

```
import cats.implicits._

val a = Concrete("a")
val b = Concrete("b")

a === b // from Eq
a.min(b) // from Order
a |+| b // from Semigroup
a.show // from Show
```

Of course, we can also summon their typeclass instances on request (using the `apply` summoner method).

```
Eq[Concrete]
Order[Concrete]
Semigroup[Concrete]
Show[Concrete]
```

> Warning
>
> `Derevo doesn't support derivation for @newtypes yet`

It will be supported soon, though. Follow up on their issue tracker to be up to date.

Higher-kinded types

Higher-kinded types are types that have type arguments. We will focus on the most common example of such types, which has shape `F[A]` (and kind `* -> *`, meaning that they take a concrete type and return a concrete type).

An example is `Option`, which takes a concrete type (e.g. `Int`) and produces another concrete type (`Option[Int]`). Kind signatures can precisely describe complex types, but a full description is out of scope.

We can start a REPL and let the compiler confirm this information for us.

```
scala> :kind Option
Option's kind is F[+A]

scala> :kind List
List's kind is F[+A]

scala> :kind Either[String, *]
scala.util.Either[String,?]'s kind is F[+A]
```

Scala tells you it is `F[+A]` instead of `F[A]` due to variance, but it is not very relevant when it comes to defining kinds.

The following examples makes use of the `derevo-cats-tagless` dependency.

```
import derevo.tagless.flatMap

@derive(flatMap)
sealed trait HigherKind[A]
case object KindOne extends HigherKind[Int]
case object KindTwo extends HigherKind[String]
```

Once again, we can put the power of such typeclasses to use.

```
KindOne.map(_ * 2) // from Functor
KindTwo.void // from Functor
KindTwo » KindOne // from FlatMap
(KindOne, KindTwo).tupled // from Semigroupal
(KindOne, KindTwo).mapN { // from Apply
  case (x, y) => s"$x - $y"
}
```

It is worth noticing that by deriving `FlatMap`, we get access to another set of typeclasses such as `Functor`, `Apply`, and `Semigroupal`.

So, the kind of `HigherKind` is `* -> *`, but can you tell what is the kind of `KindOne` and `KindTwo`? Let's ask the REPL.

```
scala> :kind HigherKind
HigherKind's kind is F[A]

scala> :kind KindOne.type
```

```
KindOne.type's kind is A

scala> :kind KindTwo.type
KindTwo.type's kind is A
```

Yes, these are concrete types.

Higher-order functors

Derevo also supports derivation for *higher-order functors*, loosely speaking. Once again, we will only explore the most common of such types, which have shape `X[F[A]]` and kind `(* -> *) -> *`. Most of our Tagless Final encoded algebras fit this shape.

For instance, a Tagless algebra `Alg[F[_]]` takes a type constructor `F[_]` as a type parameter, which has kind `* -> *`. Examples may include `IO`, `Option`, or `Either[String, *]`, among others.

Let's see how an instance of `ApplyK` can be derived for our custom Tagless algebra.

```
import derevo.tagless.applyK

@derive(applyK)
trait Alg[F[_]] {
  def foo: F[String]
}

case object ListAlg extends Alg[List] {
  def foo: List[String] = List("1", "2")
}

case object OptionAlg extends Alg[Option] {
  def foo: Option[String] = "foo".some
}
```

By deriving `ApplyK`, we get access to a few other typeclasses such as `FunctorK` and `SemigroupalK`. Let's look at the following example, which shows how these typeclasses' methods are used.

```
FunctorK[Alg].mapK(ListAlg)(λ[List ~> Option](_.headOption))
SemigroupalK[Alg].productK(ListAlg, OptionAlg)
```

In a REPL session, let's ask the compiler what are the kinds of these types.

```
scala> :kind Alg
Alg's kind is X[F[A]]

scala> :kind ListAlg.type
ListAlg.type's kind is A

scala> :kind OptionAlg.type
OptionAlg.type's kind is A
```

It is worth mentioning Cats Tagless[19], which can only derive instances for Tagless Final encoded algebras, without any external dependency; and also Scalaz Deriving[20], which provides a core for deriving typeclasses as well as a module to derive Scalaz typeclasses. Unfortunately, it doesn't support Cats yet, reason why it wasn't included in our options.

[19] https://typelevel.org/cats-tagless
[20] https://github.com/scalaz/scalaz-deriving

Effectful streams

Fs2 is a library that provides purely functional, effectful, resource-safe, and concurrent streams for Scala.

At first glance, it may seem intimidating, but don't let that first impression put you off it. Many other great libraries are built on top of this giant: Http4s, Doobie, and Skunk, to name a few.

The killer application for streams is dealing with I/O while processing data in constant memory; it is a great choice when your data doesn't fit into memory. However, Fs2 offers much more, as we are going to explore in the next section.

Concurrency

In big applications, it is very common to run both an HTTP server and a message broker such as Kafka or RabbitMQ, concurrently serving HTTP requests while consuming and producing a stream of values.

We could try to do this at the effect level, but we would need to deal with a lot of corner cases related to concurrency and resource safety. It is always recommended to choose a high-level library over bare bones, and this is where, among other areas, Fs2 shines.

If we have both an HTTP server and a Kafka consumer represented as a stream (checkout Fs2 Kafka[21] for the latter), we can do the following:

```
val server: Stream[IO, ExitCode] = ???
val consumer: Stream[IO, Unit] = ???

val program: Stream[IO, ExitCode] =
  Stream(server, consumer).parJoin(2)
```

The `parJoin` method will nondeterministically merge a stream of streams into a single stream; it races all the inner streams simultaneously, opening at most `maxOpen` streams at any point in time.

The value of `maxOpen` is `2` in our example, as we want to keep the server and consumer running concurrently.

If the processes are unrelated to one another, as it is in this case with our `server` and `consumer`, we more likely need `parJoin`. In some other cases, e.g. a consumer/producer program, we might want to make them dependent on each other. In cases like this, `concurrently` may be a better fit.

[21] https://github.com/fd4s/fs2-kafka

```
val producer: Stream[IO, Unit] = ???

val program: Stream[IO, Unit] =
  consumer.concurrently(producer)
```

As its name suggests, it runs `consumer` while running `producer` in the background. Upon finalization of the `consumer`, the `producer` will be interrupted and awaited for its finalizers to run.

By *compiling* our stream, we can go back to `IO` (or any `F[_]: Sync`).

```
program.compile.drain // IO[Unit]
```

We can also combine different semantics. Say we want to run a server, a consumer, and a producer. The server is independent of the others, whereas the producer depends on the consumer. We can achieve this behavior by combining `parJoin` and `concurrently`.

```
val program: Stream[IO, ExitCode] =
  Stream(
    server,
    consumer.concurrently(producer)
  ).parJoin(2)
```

There is another variant of `parJoin` named `parJoinUnbounded`, which opens as many streams as it can at a given point in time.

As a rule of thumb, remember about the relationship between processes. If they are related, go for `concurrently`; if they are not, go for `parJoin`.

Failing to understand the difference may end in obtaining undesired semantics.

There are a few other functions, such as `merge` and `mergeHaltR`, that may be of interest.

Resource safety

Cats Effect provides a `Resource` datatype, which allows us to perform a clean-up either in case of completion or failure. We can revisit our `Background` effect implementation and see if we can do better.

Below is our simple `Background` implementation.

```
(Timer[F].sleep(duration) *> fa).start.void
```

It spawns a new fiber (`start`) for every scheduled computation to then ignore it (`void`). At this moment, we had lost control over the process since we discarded its corresponding fiber. In most cases, it is acceptable to fire-off computations this way, but it could easily become a difficulty when the application starts to grow.

We could treat fibers as a resource that needs to be cleaned up (`cancel`) instead. Let's see a safer implementation based on `Resource`, and powered by a `Queue`.

```
def make[F[_]: Concurrent: Timer]: Resource[F, Background[F]] =
  Resource.suspend(
    Queue.unbounded[F, (FiniteDuration, F[Any])].map { q =>
      val bg = new Background[F] {
        def schedule[A](
          fa: F[A],
          duration: FiniteDuration
        ): F[Unit] =
          q.enqueue1(duration -> fa.widen)
      }

      val bgStream = q.dequeue.map {
        case (duration, fa) =>
          Stream.eval_(fa.attempt).delayBy(duration)
      }.parJoinUnbounded

      Resource
        .make(bgStream.compile.drain.start)(_.cancel)
        .as(bg)
    }
  )
```

A concurrent `Queue` is a fine datatype provided by the Fs2 library. In this implementation, we are creating an *unbounded* queue consisting of a duration and a computation to be scheduled.

The `schedule` method will enqueue elements in our queue, whereas a concurrent stream will dequeue them to be immediately scheduled. All this functionality is packed as a resource, responsible for the cancellation of the spawned fibers when the program terminates.

Notice that since Cats Effect `v2.1.0`, there is a new `background` method we could use.

Resource from a stream

The previous program is mainly implemented using `Resource`, and dealing with fibers, which are low-level. Preferably, we should define our program in terms of streams, using

Fs2's high-level API, which already considers many corner cases we might be missing.

Since Fs2 is built on top of Cats Effect, it also understands `Resource`, and it can create one from a stream.

```
def resource[F[_]: Concurrent: Timer]: Resource[F, Background[F]] =
  Stream
    .eval(Queue.unbounded[F, (FiniteDuration, F[Any])])
    .flatMap { q =>
      val bg = new Background[F] {
        def schedule[A](
          fa: F[A],
          duration: FiniteDuration
        ): F[Unit] =
          q.enqueue1(duration -> fa.widen)
      }

      val process = q.dequeue.map {
        case (duration, fa) =>
          Stream.eval_(fa.attempt).delayBy(duration)
      }.parJoinUnbounded

      Stream.emit(bg).concurrently(process)
    }
    .compile
    .resource
    .lastOrError
```

Notice how the `concurrently` method replaces the manual `start` and `cancel`, which already manages interruption for us.

In the end, we perform a `compile.resource.lastOrError`, which is ideal when a stream produces a single element. In this case, it is a single `Background` instance.

Finally, we need to change how we are using `Background`. It can no longer be an implicit effect, and it should now be considered a resource.

```
Background.resource[IO].use { bg =>
  restOfTheProgram(bg)
}
```

This would be the most correct usage, though, it means we need to modify our entire application to take an explicit `Background`. We could instead make an exception and make it implicit, as the semantics will remain the same.

```
Background.resource[IO].use { implicit bg =>
  restOfTheProgram
}
```

This is arguably an acceptable trade-off.

Interruption

Another particularly good use of Fs2 streams for control flow is managing interruption. It allows us to do so in a few lines, utilizing its high-level API.

The following program interrupts the action of printing out "ping" after 3 seconds.

```
Stream
  .emit[IO, String]("ping")
  .repeat
  .metered(1.second)
  .evalTap(putStrLn(_))
  .interruptAfter(3.seconds)
  .onComplete(Stream.eval(putStrLn("pong")))
```

Here is one possible output:

```
[info] running examples.streams.interruption
[2020-01-14T16:18:20.870Z] - ping
[2020-01-14T16:18:21.871Z] - ping
[2020-01-14T16:18:22.872Z] - ping
[2020-01-14T16:18:22.910Z] - pong
```

Let's analyze every function. Most of them are self-explanatory, though:

- `emit`: it emits a single value "ping".
- `repeat`: it repeats a stream forever.
- `metered`: it throttles the stream to the specified rate.
- `evalTap`: it peeks into the current value as it performs an effectful computation.
- `interruptAfter`: it interrupts the stream after a given time.
- `onComplete`: it prints "pong" when the stream successfully completes.

Using `interruptAfter`, we can only interrupt the stream at a specified time. If we wanted to interrupt the stream on a given condition, we should use `interruptWhen` instead.

There are a few variants of the same function. Let's see an example based on `Deferred`:

```
Stream
  .eval(Deferred[IO, Either[Throwable, Unit]])
  .flatMap { promise =>
    Stream
      .eval(IO(Random.nextInt(5)))
      .repeat
      .metered(1.second)
      .evalTap(putStrLn(_))
      .evalTap {
        case 0 => promise.complete(().asRight)
        case _ => IO.unit
      }
      .interruptWhen(promise)
      .onComplete(Stream.eval(putStrLn("interrupted!")))
  }
```

We first create an instance of `Deferred` called "promise", and then proceed to generate and print out random numbers from 0 to 4 infinitely. If we get the number zero, we complete our promise, which will trigger the interruption of the whole stream.

We will see an output similar to the one below when we run it.

```
[info] running examples.streams.interruption
[2020-01-14T16:16:35.122Z] - 4
[2020-01-14T16:16:36.125Z] - 3
[2020-01-14T16:16:37.125Z] - 0
[2020-01-14T16:16:37.167Z] - interrupted!
```

It is a powerful function that can be further composed to achieve better control flow.

Pausing a stream

Interruption - and the ability to control it - is great, but it is not always what we want. What if we wanted to pause our stream for a while (e.g. waiting for an external result) and then continue from where we left off?

In such a case, what we need is `pauseWhen`, which takes either a `Signal[F, Boolean]` or a `Stream[F, Boolean]`.

The following example makes use of a signal, which is responsible for pausing and resuming the stream after a supplied time.

```
Stream
  .eval(SignallingRef[IO, Boolean](false))
  .flatMap { signal =>
    val src =
      Stream
        .emit[IO, String]("ping")
        .repeat
        .metered(1.second)
        .evalTap(putStrLn(_))
        .pauseWhen(signal)

    val pause =
      Stream
        .sleep[IO](3.seconds)
        .evalTap(_ => putStrLn("» Pausing stream «"))
        .evalTap(_ => signal.set(true))

    val resume =
      Stream
        .sleep[IO](7.seconds)
        .evalTap(_ => putStrLn("» Resuming stream «"))
        .evalTap(_ => signal.set(false))

    Stream(src, pause, resume).parJoinUnbounded
  }
  .interruptAfter(10.seconds)
  .onComplete(Stream.eval(putStrLn("pong")))
```

We first create a signal to then build the rest of our program, which is composed of three smaller programs: `src`, `pause`, and `resume`. The first one is the source stream that prints out "ping" on every second, which can be paused or resumed by using `pauseWhen(signal)`. The `pause` program will set our signal value to `true` after 3 seconds, and the `resume` program does the opposite after 7 seconds.

All these small programs are put together as a stream of streams, using `parJoinUnbounded` to run them concurrently.

This composed program will be interrupted after 10 seconds, no matter what. In case of successful completion, it will also print out "pong". You should see an output like the one below when you run it:

```
[info] running examples.streams.interruption
[2020-01-14T16:15:41.442Z] - ping
[2020-01-14T16:15:42.440Z] - ping
[2020-01-14T16:15:43.428Z] - » Pausing stream «
```

```
[2020-01-14T16:15:43.440Z] - ping
[2020-01-14T16:15:47.429Z] - » Resuming stream «
[2020-01-14T16:15:47.439Z] - ping
[2020-01-14T16:15:48.441Z] - ping
[2020-01-14T16:15:49.443Z] - ping
[2020-01-14T16:15:50.374Z] - pong
```

We have only touched the tip of the iceberg; there is much more we can accomplish by using Fs2, which provides a fascinating high-level API. It is a fantastic tool to manage effectful streams as well as control flow in large applications.

PRACTICAL FP IN SCALA

A HANDS-ON APPROACH

A BOOK FOR INTERMEDIATE TO ADVANCED SCALA DEVELOPERS. AIMED AT THOSE WHO UNDERSTAND FUNCTIONAL EFFECTS, REFERENTIAL TRANSPARENCY AND THE BENEFITS OF FUNCTIONAL PROGRAMMING TO SOME EXTENT BUT WHO ARE MISSING SOME PIECES TO PUT ALL THESE CONCEPTS TOGETHER TO BUILD A LARGE APPLICATION IN A TIME-CONSTRAINED MANNER.

THROUGHOUT THE CHAPTERS WE WILL DESIGN, ARCHITECT AND DEVELOP A COMPLETE STATEFUL APPLICATION SERVING AN API VIA HTTP, ACCESSING A DATABASE AND DEALING WITH CACHED DATA, USING THE BEST PRACTICES AND BEST FUNCTIONAL LIBRARIES AVAILABLE IN THE CATS ECOSYSTEM.

YOU WILL ALSO LEARN ABOUT COMMON DESIGN PATTERNS SUCH AS MANAGING STATE, ERROR HANDLING AND ANTI-PATTERNS, ALL ACCOMPANIED BY CLEAR EXAMPLES. FURTHERMORE, AT THE END OF THE BOOK, WE WILL DIVE INTO SOME ADVANCED CONCEPTS SUCH AS MTL, CLASSY OPTICS AND TYPECLASS DERIVATION.

GABRIEL VOLPE IS A SOFTWARE ENGINEER, SPECIALIZED IN FUNCTIONAL PROGRAMMING, FROM BUENOS AIRES, ARGENTINA. HE HAS BEEN WRITING CODE SINCE 2005, USING SCALA PROFESSIONALLY SINCE 2014 AND HASKELL SINCE 2017. ACTIVE OPEN-SOURCE SOFTWARE CONTRIBUTOR.