Chapter 1: Shopping Cart project

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

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

Integration tests

In this last section, we will see why integration tests are also essential. But first, let's be clear: What do integration tests mean, exactly?

We can interpret them in many ways. The usual meaning refers to starting up our entire application to be tested against all the external components, which in our case are PostgreSQL, Redis, and the remote Payment client.

However, this is tedious, and the benefits don't justify the cost of having such an exclusive testing environment.

A good approach is to test external interpreters in isolation. For example, we could test the Postgres interpreters in a single test suite, and the Redis interpreters in another test suite. If we have a real test payment client, we could also test that. In this case, we don't, so we are going to move forward with the first two.

Resource allocation

When we have to deal with resources that must be shared across tests such as a Postgres connection, we realize we have a problem since Scalatest is not very friendly with purely functional resource management.

Our only hope is to wait for the release of a purely functional testing library such as Flawless[2], or give the work-in-progress Kallikrein[3] a try. Other than that, we are left alone in side-effects land. Though, we can try to hide this in the same way we are hiding the call to `unsafeToFuture()` in our tests.

We need to define a new test suite that extends `PureTestSuite` and mixes-in the `BeforeAndAfterAll` trait from Scalatest.

```
trait ResourceSuite[A] extends PureTestSuite with BeforeAndAfterAll {

  def resources: Resource[IO, A]

  // ... more here ...

}
```

It is parameterized on the resource type `A`, and it defines an abstract method `resources`. Next, we need some private *mutable* variables and a *latch* (backed by a `Deferred`).

[2] https://github.com/kubukoz/flawless
[3] https://github.com/tek/kallikrein

Chapter 8: Assembly

We have come a long way. Having turned business requirements into technical specifications, we then designed our system using purely functional programming and finally wrote property-based tests.

Now is the time to put all the pieces together, and here is where I would like to quote one of my favorite song-writers that says:

> I know the pieces fit
>
> *Maynard James Keenan*

In this chapter, we are going to assemble our application, and we will ultimately spin up our HTTP server, connect to our database, and start serving requests.

- `ask` allows us to access the context.
- `reader` is a shortcut for `ask.map(f)`.

Let's see an example. First, we need some datatypes to represent a context.

```
final case class Foo(value: String)
final case class Bar(value: Int)
final case class Ctx(foo: Foo, bar: Bar)
```

Next, we define a few handy type aliases.

```
type HasFoo[F[_]] = ApplicativeAsk[F, Foo]
type HasBar[F[_]] = ApplicativeAsk[F, Bar]
type HasCtx[F[_]] = ApplicativeAsk[F, Ctx]
```

With all this in place, we can write functions as follows:

```
def p1[F[_]: Console: FlatMap: HasCtx]: F[Unit] =
  F.ask.flatMap(ctx => F.putStrLn(ctx))
```

It accesses the current context, and it prints it out to the console.

We can now materialize our program using `Kleisli` (also known as `ReaderT`).

```
val ctx = Ctx(Foo("foo"), Bar(123))

p1[Kleisli[IO, Ctx, *]].run(ctx) // IO[Unit]
```

We could also materialize it using `IO` directly, but it would require us to either write a rather *hacky* `ApplicativeAsk[IO, Ctx]` instance, or to use a Ref-backed instance provided by Meow MTL.

Both ways of acquiring such instance are effectful, reason why this technique is discouraged by purists, as something alike wouldn't be possible in language with global coherence of typeclasses like Haskell.

Usually, we do it anyway, having an understanding of its trade-offs. If we were to use Monad Transformers instead, we would be introducing more boilerplate (type inference tends to be limited), as well as a performance penalty (nested bind calls).

```
Ref.of[IO, Ctx](ctx).flatMap { ref =>
  ref.runAsk { implicit ioCtxAsk =>
    p1[IO]
  }
}
```

Let's now look at a program that wouldn't compile.

```
  userLens[A].get(a).name

case class Ctx(user: User, id: String)

userName(Ctx(User("Oleg"), "0x42")) // Oleg
```

It is worth mentioning that the lenses defined by Meow MTL are Shapeless' lenses, and the prisms are custom ones defined in the library. If you are looking for a library that derives classy optics using Monocle, I recommend checking out the Sbt classy plugin[10] or the Tofu library[11] that has an interop module.

Configuration

We can now try to apply this technique in our application. Let's start with resources, which require `HttpClientConfig`, `PostgreSQLConfig`, and `RedisConfig`.

Our current implementation takes in an `AppConfig`.

```
def make[F[_]: ConcurrentEffect: ContextShift: Logger](
  cfg: AppConfig
): Resource[F, AppResources[F]] = { ... }
```

This is for convenience, to avoid having three different parameters, which becomes boilerplatey. Ideally, we should share as little information as it is needed.

So let's create two handy type aliases to use `ApplicativeAsk` to access context instead of manually passing arguments.

```
type HasAppConfig[F[_]]       = ApplicativeAsk[F, AppConfig]
type HasResourcesConfig[F[_]] = ApplicativeAsk[F, ResourcesConfig]
```

Our new `ResourcesConfig` datatype is part of `AppConfig` and is defined as follows:

```
case class ResourcesConfig(
    httpClientConfig: HttpClientConfig,
    postgreSQL: PostgreSQLConfig,
    redis: RedisConfig
)
```

We can now modify our smart constructor's signature.

```
def make[
    F[_]: ConcurrentEffect: ContextShift: HasResourcesConfig: Logger
]: Resource[F, AppResources[F]] = { ... }
```

[10] https://github.com/cb372/sbt-classy
[11] https://github.com/TinkoffCreditSystems/tofu

Meow MTL can give us an `ApplicativeAsk[IO, A]` instance if we have a `Ref` holding the context `A`. So let's define a function that loads the configuration and creates a mutable reference with our `AppConfig`.

```
val configLoader: IO[Ref[IO, AppConfig]] =
  config.load[IO].flatMap(Ref.of[IO, AppConfig])
```

We can now modify our entry point as follows:

```
import com.olegpy.meow.effects._

override def run(args: List[String]): IO[ExitCode] =
  configLoader.flatMap(_.runAsk { implicit ioAsk =>
    loadResources[IO] { cfg => res =>
       restOfTheProgram
    }
  })
```

Once again, Meow MTL helped us gain in performance and ergonomics.

What are the benefits?

In this tiny example, we have seen `ApplicativeAsk`'s usage, but what are the benefits of using it compared to plain function arguments? This is a great question. In fairness, one can choose either approach and it would be fine.

Using plain arguments, a program may look as follows:

```
def program[F[_]: Concurrent]: F[Unit] =
  makeContext[F].flatMap { ctx =>
    p1(ctx)
  }

def p1[F[_]: FlatMap](ctx: AppCtx): F[Unit] =
  p2(ctx.foo) >> p3(ctx.bar)

def p2[F[_]: FlatMap](foo: Foo): F[Unit] =
  p4(foo.t1) >> p5(foo.t2)

def p3[F[_]: FlatMap](bar: Bar): F[Unit] =
  p6(bar.t1) >> p7(bar.t2)
```

Every small program takes in the piece of context it needs and nothing more, potentially hiding delicate information from other functions. It does get a bit cumbersome, though, especially when we are talking about a medium to big size application. However, since

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

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

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

Categories

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

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

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

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

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

Items

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

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

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

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

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

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

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

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

Orders

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

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

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

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

We have some new entities here.

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

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

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

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

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

Categories

Our Category routes is fairly similar to the Brand routes.

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

  private[routes] val prefixPath = "/categories"

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

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

}
```

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

Users

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

  private[routes] val prefixPath = "/auth"

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

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

    }

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

}
```

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

Brands Admin

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