Introduction to Functional Architectures

A commit-by-commit dissection of a real application

Anatolii Kmetiuk

Contents

1. Preface

1.1 Why this book was written

Whenever a concept sticks around in an applied discipline, it means it is useful in solving some particular problem. Hence the best way to learn applied disciplines is by focusing on the problems and studying solutions to them only once the problems are fully understood. Unfortunately, there is not many problem-focused learning materials in the functional programming world.

For example, I often see discussions about "what a Monad is", but much more rare people discuss "what problem a Monad solves". This is also true for many functional programming libraries, which in the Scala world are currently assembled under the Typelevel[1] umbrella. Although this organization aims[2] to remove barriers for the newcomers to enter purely functional programming world, I still see many confusion about "why need Cats" or "what Shapeless is good for".

This book aims to solve this problem of confusion. Growing from the belief that one must study applied concepts starting from the problems they solve, this book follows the development of an application designed according to the functional style. Here is how this book is different from other resources:

- The application development is followed commit-by-commit, starting from the very first one. Every chapter has one or more commits it discusses. Any commit introduces a certain change to the code base, and this book analyzes why and how these changes were made.
- The application in question and its development are real. In the sense that they were not developed for the sake of this book only. The application is a video streaming server - so that you can launch a server and access the file system of the host computer via browser over the local network. You can play any mp4 file via HTML5 player over the local network. The motivation to develop this application was precisely my own need to stream videos hosted on my computer to my tabled, and the absence of satisfactory solutions to the problem (either too slow, or proprietary, or some other issue).
- Since the application is "real", there will be unintended bugs and typos in the code. But instead of editing them out with a rebase or other git technique, I viewed them as just another

[1] http://typelevel.org/
[2] https://youtu.be/RGFZ0fT_Pzw

kind of problem. As you recall, this entire book is focused on learning solutions by learning problems they solve - so more problems, more content to learn! Concretely, if you make a typo, maybe your compiler could have discovered it? Or maybe your code is not DRY enough, so that you changed it in one place, but forgot to change the other one? So, instead of editing out these typos and bugs and making a clean, sterile commit sequence, I preserved them and described why they happened and how to avoid them. Indeed, we all are going to apply this knowledge in the real world, and the real world is not a sterile, bug-free place where everything happens according to the text book. Who will I be lying to if I try to create an "ideal" commit sequence - this never happens in the real world anyway.

- The principle made in this book is "functional programming last". We won't be using functional techniques right away, rather we will start with tried and working imperative solutions. Indeed, if you are reading this book, you are probably an experienced imperative programmer looking to see what functional programming is all about. You worked on lots of problems, and you know that imperative programming works just great solving them. If the existing solutions work, that's the job of the functional programming to prove its value, not of the imperative programming. Concretely, this means that if we need to do an I/O streaming, we won't be going to FS2[3], but to Apache IO[4]. Only when we start to encounter problems with the existing solutions that we may switch to the functional ones.
- Technical chapters. As in any real-life application, not all of our commits will be about learning new technologies. Some of them will be dedicated to refactoring and architectural improvement. Hence certain chapters covering these commits are purely technical. More often than not they will merely mention the refactoring was done without going into too much details - just so that sudden changes in the code don't surprise you in the later chapters.

1.2 Structure

This book consists of four parts:

- **Inception** - covers the beginning of the life of the application. The project setup, the early design decisions - basically the setting for what happens next.
- **Server Side** - covers the server side of the application. In process, we will cover what you can do with implicit conversions in Scala, the motivation for pure functional design, and some patterns of functional programming: Functional Onion Architecture and the Type Classes.
- **Client Side** - covers the client side developments. In process, we will focus in more details on why we need effect types (`F[A]`), how they naturally lead to Monads and to libraries like Cats.
- **Video Streaming** - covers the implementation of the video browsing and streaming capability. Here, we will learn about more advanced functional programming techniques: a real-world scenario for when you want to use Applicative, Travers and Monad Transformers.

The parts consist of chapters. Every chapter is bound to a certain commit or a set of commits and aims to introduce a certain concept used in these commits. Usually chapters consist of the following sections:

- **Motivation** describes the problem we are facing. Why we had the need to change something at all.
- **Solution**. once we understood the problem from the Motivation section, we will discuss its solution in this section. Here, we discuss it on the conceptual level, the level of planning.
- **Implementation**. This is where we turn our Solution into code.

[3]https://github.com/functional-streams-for-scala/fs2
[4]https://commons.apache.org/io/

1.3 Conventions

1.3.1 Remarks

At the beginning of each chapter, you'll usually see remarks as follows:

M Motivation in one sentence

C ff3d4d

The "M", or Motivation, remark describes the motivation of the chapter in one sentence. The essential idea behind what we will do. It will be useful to keep in mind this idea throughout the chapter, use it as a "beacon" to know what it is all about. Also it can be useful if you want to quickly repeat the content covered in the book, or gain a birds-eye view on how the application evolves.

The "C", or Commit, remark states the commits covered in this chapter. The commits are clickable, and will take you to the GitHub page with the diff in question.

1.3.2 Code Listings

The code listings used in the book come in two flavors: *conventional listings* and *diffs*.

Here's a typical conventional source:

Listing 1.1: `MainJVM.scala`, `ec72db`

```scala
def main(args: Array[String]): Unit = {
  val server = HttpServer.create(new InetSocketAddress(8080), 0)

  def serveFile(path: String, contentType: String = "text/html"): HttpHandler = { e: HttpExchange =>
    val file   = new File(s"assets/$path")
    val fileIs = FileUtils.openInputStream(file)
    val os     = e.getResponseBody
    try {
      e.sendResponseHeaders(200, 0)
      IOUtils.copy(fileIs, os)
      os.close()
    } finally {
      os.close()
      fileIs.close()
    }
  }

  server.createContext("/", serveFile("html/index.html"))
  server.createContext("/js/application.js", serveFile("js/application.js", "text/javascript"))
  server.createContext("/js/application.js.map", serveFile("js/application.js.map", "text/plain"))

  server.start()
}
```

The title of the listing follows a convention and contains the following:

- The name of the file the listing was taken from. It is clickable and will take you to the GitHub page of that file as it was during the commit we are currently on.
- The commit this file belongs to. If you click the commit, you will be taken to its diff page.

And here is a typical diff listing:

Listing 1.2: `MainJS.scala`, `055f4e`, `@@ -1,9 +1,11 @@`

```
 package functionalstreamer

 import scala.scalajs.js.JSApp
+import org.scalajs.dom.{document, window}

 object MainJS extends JSApp {
-  def main(): Unit = {
-    println("Hello World")
+  def main(): Unit = window.onload = { _ =>
+    val placeholder = document.getElementById("body-placeholder")
+    placeholder.innerHTML = "Hello World from JS"
   }
 }
```

It follows the Unified Diff Format[5]. This format describes changes to the sources in terms of what is removed and what is added by the commit.

In the title, you can see one more element added, the `@@ -1,9 +1,11 @@`. This essentially means, "we removed 9 lines starting from the line 1 from the file before the commit, and inserted 11 lines starting from the line 1 to the file after the commit, as specified in the listing that follows".

In the listing, you can see which lines exactly are removed and which are added by the commit. Whatever is prefixed by "-" is removed from the original file by the commit, and what is prefixed by "+" - is added.

In this example, we have added one more import, removed the original definition of the `main` method and provided a new one.

If a listing lacks a file name or a commit in its title, this means it does not belong to the code base and most probably is presented as a thought experiment.

1.4 Obtaining and working with the sources

The GitHub page of the application we will be discussing is as follows: https://github.com/functortech/functional-streamer. Here is how we will set up our environment[6]:

- In order to download the repository, run `git clone https://github.com/functortech/functional-stre` Then, `cd` to the directory it was cloned into by running `cd functional-streamer` command.
- To navigate between commits, run `git checkout <commit>`, for example, `git checkout 055f4e`.
- To find out which commit you are currently on, run `git status`.
- If you accidentally modify the code and want to reset it to the condition as it was in the commit you are on, run `git reset -hard`.
- If you've got files that are untracked by Git and that fail you the compilation, you can get rid of them via `git clean -f`[7]. This normally should not happen.

[5]http://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html#Detailed-Unified
[6]The only prerequisites to set everything up are up-to-date versions of Git and SBT
[7]https://stackoverflow.com/q/61212

- We will be running and compiling the project from the SBT console. To enter the SBT console, run `sbt`. This book was not written against any IDE, so if you want to use a particular IDE, you should consult its documentation.
- To compile and start the program, run `functionalstreamerJVM/reStart` (this command will be explained later in the Inception part).

1.5 How to read this book

1.5.1 Philosophy

This book should not be perceived as a conventional textbook. Rather, think of it as of a story. A story usually tells us a sequence of events that happened in someone's life. Similarly, this book tells a story of the life of the video streaming server application. The chapters are written in chronological order, so at the beginning of the book you'll learn about the early commits in life of the application, and the closer you move to the end, the later commits you will be learning about.

While reading a conventional story, you may think of the motivation of the events in the lives of people involved. You may think why they acted this way or another, and how would you have acted in their place. While thinking on the stories you read, you gain experience that you can use in your real life. Similarly, this book will make you think on the course the application develops over time. That's exactly how you will gain more experience from it.

1.5.2 Algorithm

The key is to follow what is happening during each commit and why. Therefore, it is highly recommended that you have the list of commits[8] in front of you at all times, and whenever you see a reference to a commit in the book, you should see where this commit belongs in the list. Just to see the big picture.

To gain hands-on experience, I recommend to also follow the commits in your local clone of the repository. Whenever you see a commit, it is advised that you checkout it locally, compile, run and experiment with it. The way you can navigate through commits is described in the section on obtaining the sources.

This information should be sufficient for you to be all set for the journey. It is my hope that it will be insightful and fun for you!

[8] https://github.com/functortech/functional-streamer/commits/master

I Inception

2. Functional Streamer, the Application

M A minimalistic video streaming server.

C `ff3d4d`, `ec72db`, `055f4e`

2.1 Motivation

Functional Streamer was born out of a personal need to be able to stream videos from my computer to the tablet via local Wi-Fi network. The existing solutions for that were either too slow or did not do exactly what I wanted.

The idea of Functional Streamer is to have a minimalistic, local YouTube: a server with access to your file system that you can browse and where you can watch your videos.

2.2 Solution

We need a server that is capable of serving static files and streaming videos.

Another desirable feature is that the streaming site should be a single-page web application (SPA) that communicates with the server via a JSON-based HTTP API. The motivation behind an SPA is as follows:

- It keeps the server-side specialized on the logic and the client-side - on representation.
- The JSON API opens the possibility to implement native mobile applications to use the service with.

2.3 Setting

Since we are building an SPA, we need a project that has a web server and a JavaScript client. For this reason, we set up the project as a Scala.js one via the first commit, `ff3d4d`.

Next we need to implement a server-side capability to serve pages. We try to solve this task with minimal effort. For this purpose, we will start from the most simple technologies: Sun's standard `HttpServer`[1] for listening to HTTP events and Apache Commons IO to handle streaming:

[1]https://docs.oracle.com/javase/8/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/package-summary.html

Listing 2.1: `MainJVM.scala`, `ec72db`

```scala
import java.io.File
import java.net.InetSocketAddress
import com.sun.net.httpserver.{HttpServer, HttpHandler,
  HttpExchange}

import org.apache.commons.io.{IOUtils, FileUtils}
```

HttpServer is extremely easy to bootstrap with only several lines of code. And Apache Commons IO can turn ten lines of stream management code into just one.

Listing 2.2: `MainJVM.scala`, `ec72db`

```scala
def main(args: Array[String]): Unit = {
  val server = HttpServer.create(new InetSocketAddress(8080),
     0)

  def serveFile(path: String, contentType: String = "text/html
     "): HttpHandler = { e: HttpExchange =>
    val file   = new File(s"assets/$path")
    val fileIs = FileUtils.openInputStream(file)
    val os     = e.getResponseBody
    try {
      e.sendResponseHeaders(200, 0)
      IOUtils.copy(fileIs, os)
      os.close()
    } finally {
      os.close()
      fileIs.close()
    }
  }

  server.createContext("/", serveFile("html/index.html"))
  server.createContext("/js/application.js", serveFile("js/
     application.js", "text/javascript"))
  server.createContext("/js/application.js.map", serveFile("js
     /application.js.map", "text/plain"))

  server.start()
}
```

This code is straightforward in a manner typical for Java. We first create the server, then instruct it to listen to several request URLs and respond with the corresponding files. Since all these endpoints do the same thing - read a file and respond with it - the logic to read the file is abstracted into a separate method. And yes, doing `os.close()` twice is an unintended typo.

We also add the static file to serve, `index.html`.

Next, since we are building an SPA, it is a good idea to implement a rudimentary client-side capability to set different views. Just so that we can have something to start with when we move to implementing the client-side.

Listing 2.3: `index.html`, 055f4e, `@@ -25,7 +25,7 @@`

```
      </a>
    </div>
    <div class="ui divider"></div>
-   <div>Hello World</div>
+   <div id="body-placeholder"></div>
  </div>

  </body>
```

Listing 2.4: `MainJS.scala`, 055f4e, `@@ -1,9 +1,11 @@`

```
 package functionalstreamer

 import scala.scalajs.js.JSApp
+import org.scalajs.dom.{document, window}

 object MainJS extends JSApp {
-  def main(): Unit = {
-    println("Hello World")
+  def main(): Unit = window.onload = { _ =>
+    val placeholder = document.getElementById("body-placeholder")
+    placeholder.innerHTML = "Hello World from JS"
   }
 }
```

If you run the program with `functionalstreamerJVM/reStart`[2], you should be able to navigate to localhost:8080.

[2]`reStart` comes from the Revolver plugin for SBT. It allows to run the program in a separate JVM. Here, the motivation is that we should be able to re-start and re-compile the project frequently. But a running server, obviously, blocks the main thread, preventing us from running SBT commands. Revolver allows us to run the server while retaining the access to the SBT console at the same time. You can learn more about Revolver from its GitHub page: https://github.com/spray/sbt-revolver

3. DSL

M Hide technical details and focus on the important when defining the server.

C a23941

3.1 Motivation

There are certain things with this Java-style implementation that are not particularly likable:

- The way `HttpHandlers` are bound to paths. For every path you need to call a `createContext` method on the server, quite a bit of boilerplate.
- `HttpHandlers` are bound to a certain request URL represented as a `String`. However, what if we want more fine-grained filtering? For example, for the purposes of AJAX, we may want to have an endpoint that handles only POST requests. With the current implementation, we would probably need to have an `if-else` clause in every `HttpHandler` to check for an appropriate method, which is also ugly.

So the problem we are facing here is that the important details are hidden amidst a lot of technical code (also a typical scenario for Java). What we need, hence, is a convenient DSL - a Domain-Specific Language, that would describe what we need the way we want it, while hiding technical details.

3.2 Solution

We have a situation where different code should be executed based on the type of the request that arrives. Differentiation based on the kind of some value is a job for a `match` statement, or its cousin, `PartialFunction`.

Instead of having a bunch of calls to `createContext`, we may have a single partial function that matches the URL and the method of the request, and executes a corresponding handler. We want something as follows:

Listing 3.1: `MainJVM.scala`, a23941

```scala
def main(args: Array[String]): Unit = {
  val server = createServer(8080) {
    case e @ GET -> "/"                  => serveFile(e, "html
       /index.html"  )
    case e @ GET -> "/js/application.js" => serveFile(e, "js/
       application.js")
  }
  server.start()
}
```

3.3 Implementation

How do we implement this DSL? A DSL involves a set of types to represent what is important and some operations on these types.

3.3.1 Types

We should define the types we will describe the server with.

Listing 3.2: `ServerAPI.scala`, a23941

```scala
// Types we will use to describe the Server
type Handler = PartialFunction[HttpExchange, Unit]
type Path    = String

sealed trait Method
case object GET  extends Method
case object POST extends Method
```

A `Handler` is a `PartialFunction` with its domain being some subset of all `HttpExchanges`. A `PartialFunction` implies that we will be using `case` statements to match on the `HttpExchange`, and that not all `HttpExchanges` may be handled by a given handler.

3.3.2 High-level API

Now, we need to define what we can do with these types and we can create instances of them.

Server Creation

The function to create the server can be implemented as follows:

Listing 3.3: `ServerAPI.scala`, a23941

```scala
def createServer(port: Int)(handler: Handler): HttpServer = {
  val server = HttpServer.create(new InetSocketAddress(port),
     0)
  val nativeHandler: HttpHandler = toNativeHandler(handler)
  server.createContext("/", nativeHandler)
  server
}
```

`createServer` accepts a port on which to listen and the handler. It then converts the DSL type of `Handler` to the native Sun's `HttpHandler` and binds it to the root path. This way, the path-matching logic is delegated to the `Handler`, not the Sun's native `HttpServer`.

Here, createServer is a high-level abstraction of some lower-level logic. Notice the friction that starts to appear between the high- and the low-level APIs: the need to convert the higher-level Handler to the lower-level HttpHandler, so that the lower-level API can understand it. To address it, a conversion function, toNativeHandler is defined.

Request

We are able to represent the handlers with the partial functions defined on HttpExchange, but we are unable to extract the HTTP method and the URL yet, as in the desired main method (see Listing 3.1). What we need here is an extractor for the HttpExchange type:

Listing 3.4: ServerAPI.scala, a23941

```scala
object -> {
  def unapply(exchange: HttpExchange): Option[(Method, Path)]
     =
    toMethod(exchange.getRequestMethod).map { _ -> exchange.
       getRequestURI.getPath }
}
```

Since the name of the object consists of symbols and its unapply method returns a pair, we will be able to pattern-match on it in an infix operator style.

Note also that we face the friction between the two APIs once again. This time, we need to convert the request method from a low-level String to a high-level Method. As previously, we do that via a conversion method.

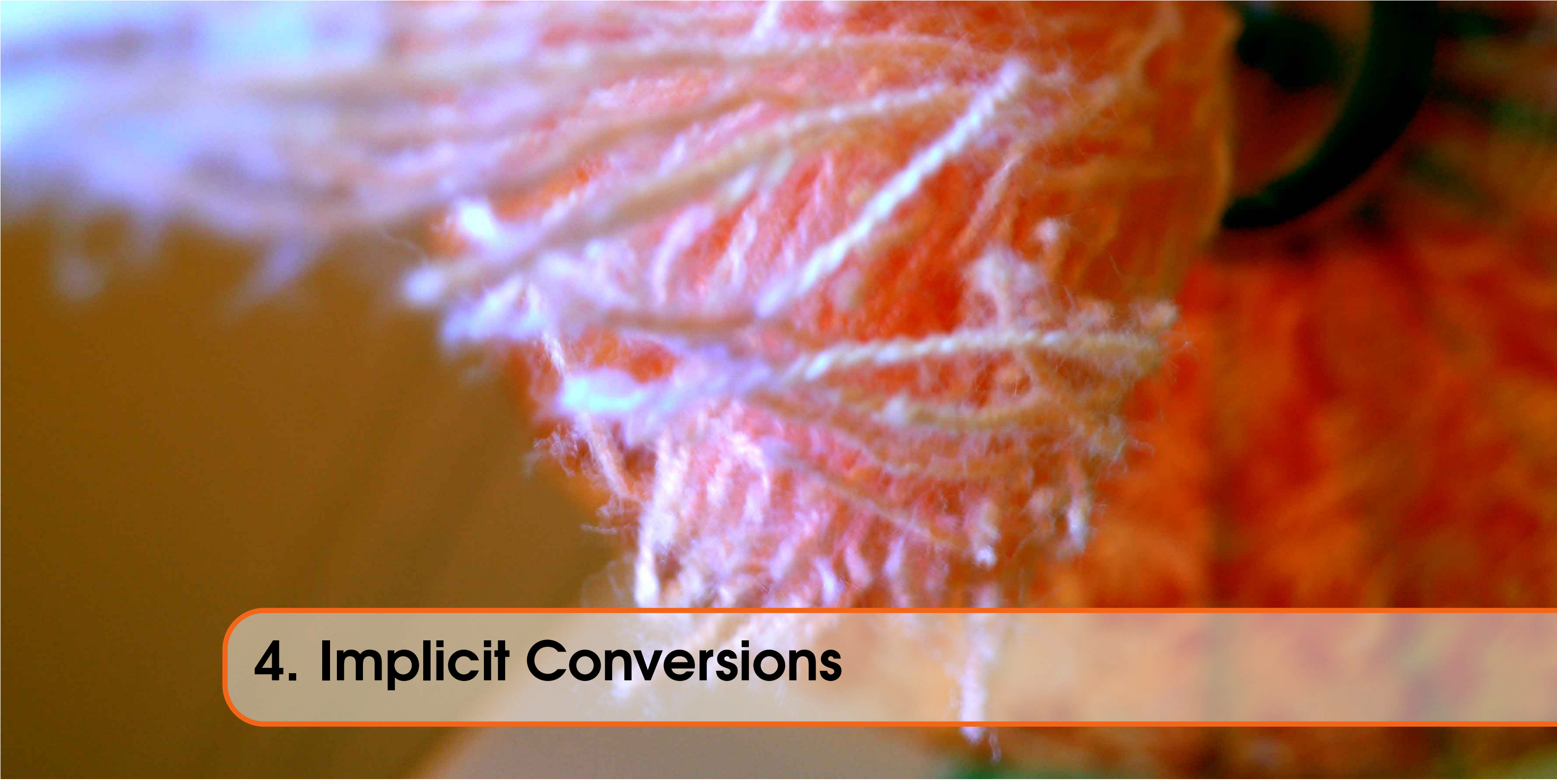
3.4 Conclusion

We now have two APIs: the low-level ones that comes with the Sun's HttpServer, and the higher-level one we have just built. The lower-level API focuses on the technical aspects of the server, while the higher-level DSL focuses on the task at hand that we are implementing.

The only thing to notice at this stage is the friction that appears between the two APIs. We need to communicate between them at some point, and hence the need to convert values from one API language to the other one.

II

Server Side



4. Implicit Conversions

M Hide the technical detail of converting (for example, from a high-level API to a low-level one)

C `ca11ff`

4.1 Refactoring: Modularisation

M Technical: Refactoring

`ServerAPI.scala` grew a bit larger and it contains logic concerning different aspects of the high-level API. So it makes sense to split it into several files and settle them to a separate package:

- `Method.scala` contains the `sealed trait Method` and its subclasses, because it is a good practice to keep each class in a separate file.
- `package.scala`, the package object, contains the high-level API types (`Handler` and `Path`, as well as the conversion logic between the two APIs - the extractor object `->`).
- `ServerAPI.scala` contains only the API that is supposed to be called by the user. `createServer` and `serveFile` ended up here.

4.2 Motivation

In the previous chapter, we have encountered the friction in the place where the two APIs meet. We solved it in a standard way: by encapsulating the conversion logic in separate methods. Can Scala do better?

4.3 Solution

When we need to convert one type to another but the conversion details feel too technical, Scala has *implicit conversions* to offer.

Implicit conversions are ordinary methods defined with the `implicit` keyword. They accept a single argument of a type that needs to be converted. They return some other type - the one we are converting to.

So essentially they are just like the conversion methods we used in the previous chapter, but with the `implicit` keyword.

When the compiler encounters an expression of some type in place where an expression of another type is expected (e.g. `Handler` passed to a method that expects `HttpHandler`), it tries to find an implicit conversion in scope that is capable of converting it to the right type. It looks at the signatures: If we have some type `A` in place where `B` is expected, the compiler will look for an implicit conversion method that accepts `A` and returns `B` - `A => B`. If it finds one, it applies it implicitly.

We can redefine the previous implementation of the `createServer` method (see Listing 3.3) as follows:

Listing 4.1: `ServerAPI.scala`, `ca11ff`

```scala
def createServer(port: Int)(handler: Handler): HttpServer = {
  val server = HttpServer.create(new InetSocketAddress(port),
     0)
  server.createContext("/", handler)
  server
}
```

4.4 Implementation

Now, let us define an implicit conversion to convert from `Handler` to `HttpHandler` implicitly:

Listing 4.2: `package.scala`, `ca11ff`

```scala
implicit def toNativeHandler(handler: Handler): HttpHandler =
   { exchange: HttpExchange =>
  if (handler.isDefinedAt(exchange)) handler(exchange)
  else {
    exchange.sendResponseHeaders(404, 0)
    val os = exchange.getResponseBody()
    try IOUtils.write("Not found", os)
    finally os.close()
  }
}
```

In fact, all we had to do is to add an `implicit` keyword to the already existing conversion method! Finally, the compiler will apply it automatically for us whenever this conversion is needed (provided you imported it first).

4.5 Conclusion

Since the need to convert values from one type to another is so frequent, Scala has a built-in solution to help with that. Whenever the you need to perform a conversion and the details of it feel too technical or clutter your code, you can use implicit conversions to hide them.

5. Rich Wrappers

M Method injection into already defined classes

C ca11ff

5.1 Motivation

We have another point of friction: the one where we have to convert the request method from `String` to `Method` with a `toMethod` call. Can we also prepend `implicit` to that method and get rid of its call from `->.unapply`? So that `unapply` looks as follows:

```
def unapply(exchange: HttpExchange): Option[(Method, Path)] =
  exchange.getRequestMethod.map { _ -> exchange.getRequestURI.
     getPath }
```

We can try:

```
[error] functionalstreamer/server/package.scala:13: type
   mismatch;
[error]  found   : String
[error]  required: ?{def map: ?}
[error] Note that implicit conversions are not applicable
   because they are ambiguous:
[error]  both method augmentString in object Predef of type (x
   : String)scala.collection.immutable.StringOps
[error]  and method toMethod in package server of type (str:
   String)Option[functionalstreamer.server.Method]
[error]  are possible conversion functions from String to ?{
   def map: ?}
[error]       exchange.getRequestMethod.map { _ -> exchange.
   getRequestURI.getPath }
[error]                         ^
[error] one error found
```

Oops! At the position `exchange.getRequestMethod` of type `String` is encountered, the compiler expects something with the `map` method (`required: ?{def map: ?}`) - since we try to call that method on that object.

5.1.1 Two implicit conversions in scope

Normally, `String` does not have a `map` method. However, Scala provides us with an implicit conversion from `String` to the `StringOps`[1] class, that defines collection methods on `String`. Also we have just defined an implicit conversion from `String` to `Option[Method]` (`toMethod`), and `Option` also defines the `map` method. Two or more applicable implicit conversions in scope are illegal in Scala, the compiler does not know which one to choose. Hence an error.

5.1.2 Implicits can be dangerous

Another reason not to introduce that conversion from `String` to `Option[Method]` is that `String` is a very abundant type. The more you use a type that has implicit conversions, the harder it is to trace where they are applied. It can lead to the conversion being applied accidentally, without your knowledge, in the wrong place, which may result in unexpected behaviour.

5.2 Solution

5.2.1 Essence

`StringOps` is a *rich wrapper* for the `String` class. A rich wrapper is a pattern that is possible due to Scala's implicit conversions. Essentially, rich wrappers allow you to inject methods into already defined classes.

5.2.2 Details

A rich wrapper over some type `A` used to inject a method `f` into it includes two things:

- A class that has the `f` method and a reference to the value of type `A` it wraps.
- An implicit conversion from `A` to that class.

This way, when we call `f` on `A`, the compiler implicitly converts that `A` to the wrapper that has the `map` method.

In our case, `StringOps` is a rich wrapper over the type `String` used to inject `map` into it.

5.2.3 Application

We can use the Rich Wrapper pattern to inject a `toMethod` method into a `String`. This way, at least, we will get rid of a pair of parentheses and will be able to perform the conversion in a more OOP way:

`package.scala`, `ca11ff`

```scala
object -> {
  def unapply(exchange: HttpExchange): Option[(Method, Path)]
    =
    exchange.getRequestMethod.toMethod.map { _ -> exchange.
      getRequestURI.getPath }
}
```

Compare this to the original implementation (see Listing 3.4). Arguably, the new implementation is more readable.

[1] http://www.scala-lang.org/api/current/scala/collection/immutable/StringOps.html

5.3 Implementation

We can perform this injection as follows:

package.scala, ca11ff

```scala
implicit class MethodString(str: String) {
  def toMethod: Option[Method] = str.toLowerCase match {
    case "get"  => Some(GET)
    case "post" => Some(POST)
    case _      => None
  }
}
```

The `implicit class` construct is a shorthand that exists specifically with rich wrappers in mind. Here is what is going on here:

- An `implicit class` defines an ordinary class with a single constructor argument. However, that constructor is implicit conversion itself. It converts the constructor parameter type to the class this constructor creates.
- We then define some methods in the body of that class that can work on that constructor argument.
- The effect is that whenever the compiler sees the call to these methods on a value for which an ‘implicit class‘ is defined, it will find and perform the conversion to that class.

5.4 Conclusion

The *rich wrapper* pattern allows you to inject methods in classes and types without having control over their definition. It is possible due to the implicit conversions the Scala compiler supports.

6. Refactoring: Error Handling

M Technical: Adding the capability to specify error handlers

C 55e6f4 to f8e0b0

6.1 Responding with `String`s

In the `toNativeHandler` method (see Listing 4.2), we have defined how to respond when handler can not handle the request. However, we already have the logic to respond with files (`serveFile` method, unchanged since Listing 2.2) in the `ServerAPI.scala`. It is only reasonable to encapsulate the logic to return `String`s the same way. This is what `55e6f4` does:

package.scala, 55e6f4, `@@ -23,11 +22,6 @@ package object server`

```

    implicit def toNativeHandler(handler: Handler):
       HttpHandler = { exchange: HttpExchange =>
      if (handler.isDefinedAt(exchange)) handler(exchange)
-     else {
-       exchange.sendResponseHeaders(404, 0)
-       val os = exchange.getResponseBody()
-       try IOUtils.write("Not found", os)
-       finally os.close()
-     }
+     else ServerAPI.serveString(exchange, "Not Found", 404)
    }
  }
```

ServerAPI.scala, 55e6f4, @@ -29,2 +29,9 @@ trait ServerAPI

```
    }
+
+   def serveString(e: HttpExchange, str: String, responseCode
    : Int): Unit = {
+     e.sendResponseHeaders(404, 0)
+     val os = e.getResponseBody()
+     try IOUtils.write(str, os)
+     finally os.close()
+   }
  }
```

6.2 Error Handling

The error handling logic should be customizable by the end user. It should not be hard-coded into the server, but some reasonable default should be provided.

To address this, we can pass the error handler in the same place where the ordinary handlers are passed - in the `createServer` (see Listing 4.1) method.

However, if the ordinary handlers are partial functions, the error handler must be a total function. More precisely, the error handler must be able to handle the *absolute complement*[1] of the domain of the handler partial function. In other words, everything not handled by the request handler, must be handled by the error handler.

So first, we redefine our handlers:

package.scala, f8e0b0, @@ -3,5 +3,6 @@ package functionalstreamer

```
  import com.sun.net.httpserver.{HttpHandler, HttpExchange}

  package object server {
-   type Handler = PartialFunction[HttpExchange, Unit]
-   type Path    = String
+   type PartialHandler = PartialFunction[HttpExchange, Unit]
+   type TotalHandler   = HttpExchange => Unit
+   type Path           = String
```

[1] https://en.wikipedia.org/wiki/Complement_(set_theory)

Then we modify the createServer method accordingly:

ServerAPI.scala, f8e0b0, @@ -8,12 +8,17 @@ import org.apache.commons.io.IOUtils, FileUtils

```
 object ServerAPI extends ServerAPI
 trait ServerAPI {
-  def createServer(port: Int)(handler: Handler): HttpServer
   = {
+  def createServer(port: Int)(handler: PartialHandler,
   errorHandler: TotalHandler = defaultErrorHandler):
   HttpServer = {
     val server = HttpServer.create(new InetSocketAddress(
        port), 0)
-    server.createContext("/", handler)
+    server.createContext("/", { e: HttpExchange =>
+      if (handler.isDefinedAt(e)) handler(e)
+      else errorHandler(e)
+    })
     server
   }

+  val defaultErrorHandler: TotalHandler = serveString(_, "
   Not Found", 404)
+
   def serveFile(e: HttpExchange, path: String, contentType:
      String = "text/html"): Unit = {
     val file   = new File(s"assets/$path")
     val fileIs = FileUtils.openInputStream(file)
```

The conversion from the handler to an HttpHandler with an error fallback handler is done only once, in the createServer method. So we can to abolish the implicit conversion altogether:

package.scala, f8e0b0, @@ -19,9 +20,4 @@ package object server

```
      case _       => None
    }
  }
-
-  implicit def toNativeHandler(handler: Handler):
   HttpHandler = { exchange: HttpExchange =>
-    if (handler.isDefinedAt(exchange)) handler(exchange)
-    else ServerAPI.serveString(exchange, "Not Found", 404)
-  }
 }
```

Also notice how Java 8 functional interfaces allow us to write the HttpHandler as a Scala lambda, without the new HttpHandler {...} syntax.

7. Purity. Functional Onion Architecture.

M Side effects are the unknown information for a caller of a side-effecting function. Sometimes their logic repeats. The Onion architecture DRYs the side effect logic and removes the unknown component from the application logic.

C 55e6f4 to 72ec5c

7.1 Motivation

7.1.1 Bugs

If you read the commits, you have probably noticed something ugly. In the `serveString` method, we forgot to set the response code from the argument of the function:

ServerAPI.scala, f8e0b0, @@ -30,6 +35,6 @@ trait ServerAPI

```

   def serveString(e: HttpExchange, str: String, responseCode
     : Int): Unit = {
-    e.sendResponseHeaders(404, 0)
+    e.sendResponseHeaders(responseCode, 0)
     val os = e.getResponseBody()
     try IOUtils.write(str, os)
     finally os.close()
```

This happened because, as you recall, we just copy-pasted the string serving logic from the error handler into that function during 55e6f4:

package.scala, 55e6f4, @@ -23,11 +22,6 @@ package object server

```diff

    implicit def toNativeHandler(handler: Handler):
      HttpHandler = { exchange: HttpExchange =>
      if (handler.isDefinedAt(exchange)) handler(exchange)
-     else {
-       exchange.sendResponseHeaders(404, 0)
-       val os = exchange.getResponseBody()
-       try IOUtils.write("Not found", os)
-       finally os.close()
-     }
+     else ServerAPI.serveString(exchange, "Not Found", 404)
    }
  }
```

Other two ugly things are present in the serveFile method:

ServerAPI.scala, f8e0b0

```scala
def serveFile(e: HttpExchange, path: String, contentType:
   String = "text/html"): Unit = {
  val file   = new File(s"assets/$path")
  val fileIs = FileUtils.openInputStream(file)
  val os     = e.getResponseBody
  try {
    e.sendResponseHeaders(200, 0)
    IOUtils.copy(fileIs, os)
    os.close()
  } finally {
    os.close()
    fileIs.close()
  }
}
```

First, we forgot to set the contentType header. Second, there is that os.close() duplication type we mentioned earlier. Believe it or not, these bugs are not staged. This is a real-world scenario.

Why did these happen, and what can we do to prevent them and the likes?

7.1.2 Problems

Non-DRY code

Don't Repeat Yourself[1] principle tells us not to write the same logic twice. serveFile and serveString methods do a very close thing:

- They set the request metadata, such as the headers and the response code.
- They perform a response by writing something from one stream to another.

Static files and raw strings are not the only cases of responses - we will also need to respond with JSON and video streams. If we don't follow the DRY principle and write the same logic many times, we increase the opportunity for these kind of bugs.

[1]https://en.wikipedia.org/wiki/Don't_repeat_yourself

Side effects

The first solution that comes to mind is to encapsulate the common logic above into a separate method:

```scala
def serveFile(e: HttpExchange, path: String, contentType:
   String = "text/html"): Unit = {
  val file    = new File(s"assets/$path")
  respond(e, () => FileUtils.openInputStream(file),
     contentType, 200)
}

def serveString(e: HttpExchange, str: String, responseCode:
   Int): Unit =
  respond(e, () => IOUtils.toInputStream(str, "utf8"), "text/
     plain", responseCode)

def respond(e: HttpExchange, isGen: () => InputStream,
  path: String, contentType: String, responseCode: Int): Unit
     = ???
```

Notice how these methods together with the `Handler` type return `Unit`. This means we expect some side effects to happen in these functions.

We can define a *side effect* as some set of instructions that is executed inside a function so, that it affects the environment outside the scope of this function. In the case of the above methods, `HttpExchange` which is passed to them from the outside world is modified and is written to - hence side effects.

A *side effecting function* has two aspects to it:

- The known: its return type (in our case, `Unit`). It is known, because it is clearly manifested in the signature.
- The unknown: its side effects (in our case, the procedure of responding to a request). Typically most side effects are not manifested by the functions. An exception is Java exceptions: in Java, a method that throws exceptions (also a side effect, since it has a potential of disrupting the program flow outside the method's scope) must declare so in the signature. But this argument is irrelevant for Scala.

So with the above solution, when the `Handlers` return `Unit` and are side effecting, we do not know what happens inside them at their call site.

What can go wrong here is that there is no guarantee people will call our `respond` method. Some people may not know we implemented it, some people may want to reinvent the wheel and continue writing the responses themselves. There is still no guarantee the `respond` method above is the only place such a logic will be concentrated.

This is all because we are explicitly defining a `Handler` as a function that performs arbitrary side effects. We can come up with a better definition.

7.2 Solution

7.2.1 Purity

A *pure function* is a function that does not produce any side effects and the result of which is determined only by its input.

In contrast to the side effecting functions, the pure ones have only one aspect to them: their return type is defined. All the information about what happened inside the function is stored in the value that it returns.

This has obvious the benefit of reducing the mental load of needing to track the side effects.

Making our handlers pure with respect to the response side effects eliminates the problem of the unknown aspect coming with the side effecting functions.

7.2.2 Functional Onion Architecture

The solution to the problem of DRY-ing the effects comes with the *functional onion architecture*[2]. This is an adaptation of the ordinary onion architecture[3]

The idea is that your application has a layered structure. The layers closer to the "core" of the onion - the inner layers - must not produce any side effects and be pure. Their job is only to compute the *description* of the operation to be done, but not actually do it.

After the description of the task to be done is computed, it is passed to the outermost layer of the onion. This is the only layer that is allowed to produce side effects. It executes the description passed to it from the inner layers and produces side effects in process.

The benefit here is that the side effects become concentrated in a single place - the outer layer. And, hence, are DRY, easy to modify, test and debug.

Another benefit is that all the other layers now are pure - meaning they do not have the unknown component of the side effects to it. This means it is easier to manipulate them: you no longer need to worry about, say, canceling a side effect of a function when you want to reverse the computation done in it.

Yet another benefit is that, as a rule, side effects happen during runtime. When you eliminate the side effects from your business logic, you suddenly have an increased capability to catch possible bugs on compile time via type-level techniques.

A few words should be said about the benefits for testability. It is much easier to test the business logic of a pure core than that of a side effecting program. If your logic is pure, you do not need to start a side-effecting server with the infrastructure required (e.g. a database or a proper file system). All you need to do for testing is to run the pure function and analyze its output. This is so because you know for sure that these functions do not do anything but produce the result they return. The only thing that needs to be tested against the live environment is the topmost thin side-effecting layer. It is usually much smaller than the business logic layer and contains only low level operations: e.g. how to write to a database, how to read a file. In contrast, business logic contains higher-level things (e.g. how to log in a user) expressed in the language the thin side-effecting layer can interpret.

7.2.3 Putting it all together

In our case, the "onion" in question (or, rather, what we want to turn into onion) is the request handler. Its *business logic* is to determine what to respond with - which input stream to read from - and what metadata should be set for the response - the content type and the response code. Its *side effects* is the process of actually writing the response into the output stream after setting the metadata.

To turn this into an onion, we need to separate the two layers: the business logic and the side effects.

[2]http://degoes.net/articles/modern-fp-part-2
[3]http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

7.3 Implementation

7.3.1 Handlers

First, let us define the class with which we will describe the response:

Response.scala, 72ec5c

```scala
package functionalstreamer.server

import java.io.InputStream

case class Response(
  payload: () => InputStream
, contentType: String = "text/plain"
, responseCode: Int    = 200
)
```

Next, make the handlers return an instance of this class:

package.scala, 72ec5c, @@ -5,4 +5,4 @@

```diff
 package object server {
-  type PartialHandler = PartialFunction[HttpExchange, Unit]
-  type TotalHandler   = HttpExchange => Unit
+  type PartialHandler = PartialFunction[HttpExchange,
   Response]
+  type TotalHandler   = HttpExchange => Response
   type Path           = String
```

The previous definition said: "We expect you to execute arbitrary side when a request arrives". The current definition says: "We expect you to describe how you want to respond when a request arrives".

7.3.2 Processing

Next, we need to define the way the `Response` should be processed:

ServerAPI.scala, 72ec5c

```scala
def createServer(port: Int)(handler: PartialHandler,
   errorHandler: TotalHandler = defaultErrorHandler):
   HttpServer = {
  val server = HttpServer.create(new InetSocketAddress(port),
     0)

  server.createContext("/", { e: HttpExchange =>
    val Response(payloadIsGenerator, contentType, responseCode
       ) = handler.applyOrElse(e, errorHandler)

    // Write the content type in the headers
    val headers = e.getResponseHeaders
    headers.put("Content-Type" , List(contentType).asJava)

    // Get the payload and response body streams
    val is = payloadIsGenerator()
    val os = e.getResponseBody

    try {
      e.sendResponseHeaders(responseCode, 0) // Send the
         status code
      IOUtils.copy(is, os)                   // Write the
         payload
    } finally {
      is.close()
      os.close()
    }
  })

  server
}
```

Now, there is only one place for us to forget to write the headers, the response code, and to close the streams.

7.3.3 Response methods

Finally, we can rewrite our serveFile and serveString as follows:

ServerAPI.scala, 72ec5c

```
def serveFile(path: String, contentType: String = "text/html")
    : Response =
  Response( () => FileUtils.openInputStream(new File(s"assets/
     $path")), contentType, 200 )

def serveString(str: String, responseCode: Int): Response =
  Response( () => IOUtils.toInputStream(str, defaultEncoding),
      "text/plain", responseCode )
```

These methods now do not need to have a reference to HttpExchange to perform their logic. Hence, it is easier to call them when defining the responses:

MainJVM.scala, 72ec5c, @@ -6,8 +6,8 @@

```
  object MainJVM {
    def main(args: Array[String]): Unit = {
      val server = createServer(8080) {
-       case e @ GET -> "/"                  => serveFile(e, "
    html/index.html"  )
-       case e @ GET -> "/js/application.js" => serveFile(e, "
    js/application.js")
+       case GET -> "/"                  => serveFile("html/
    index.html")
+       case GET -> "/js/application.js" => serveFile("js/
    application.js", "application/javascript")
      }
      server.start()
    }
```

8. Type Classes

M Interface injection

C 72ec5c to 49259c

8.1 Motivation

Let us have a closer look at how the two functions we use to generate responses changed:

ServerAPI.scala, 72ec5c

```scala
def serveFile(path: String, contentType: String = "text/html"): Response =
  Response( () => FileUtils.openInputStream(new File(s"assets/$path")), contentType, 200 )

def serveString(str: String, responseCode: Int): Response =
  Response( () => IOUtils.toInputStream(str, defaultEncoding), "text/plain", responseCode )
```

The only useful thing these functions do now is encapsulating the logic to open the streams. The creation of Response itself is easy, we do not need to encapsulate that. Hence, we should re-define them to do that only, without Response creation.

The methods in question may look something as follows:

```scala
def stream(f: File): () => InputStream = ???
def stream(s: String): () => InputStream = ???
```

We can also use Rich Wrappers to inject them into the File and String, so that to call them with an OOP syntax.

Normally, when two or more types support the same set of operations, this is described via an interface:

```scala
trait Streamable {
  def stream: () => InputStream
}
```

The benefit here is *polymorphism*. In order to call `stream` on an object, it is enough to know that it implements `Streamable` interface. Hence, better abstraction of our application logic:

```scala
def f(target: Streamable): Unit = {
  target.stream
}
```

The problem is, how do we define an interface-like construct to enable such a polymorphism? Let us have a look at the two cases: the `def stream[T](obj: T): () => InputStream` and the one where this method is injected via Rich Wrappers into multiple types.?

8.2 Solution

8.2.1 Without Rich Wrappers

For the polymorphism, we need to declare the polymorphic operations in question. If we declare it as in the trait above, though, we won't be able to make the `File` and `String` types implement the interface.

The next best thing to do is not to store the operations in question in the class definitions themselves, but as separate objects:

`Streamable.scala`, 49259c

```scala
trait Streamable[A] {
  def stream(a: A): () => InputStream
}
```

An instance of `Streamable[A]` is a bundle of certain operations supported by `A`. Just a convenient place to store them together.

Now, whenever we need to call `obj.stream` on `obj: A`, all we need to know about `A` is that there is a `Streamable[A]` for this type. "We know there is a `Streamable[A]` instance for `A`" can be expressed with implicit values:

```scala
def f[T](target: T)(implicit streamable: Streamable[T]): Unit
   = {
  streamable.stream(target)
}
```

What you can see above is called an *implicit argument*. Every method or constructor, implicit or explicit, can optionally have the last argument group declared as `implicit`. This means that you don't need to pass them explicitly (obviously), but the compiler will look they up for you in the implicit scope. All you need to do is to put the desired values for them on scope by importing them.

The code above is polymorphic on `T`. All we need to know about `T` is that there is a `Streamable[T]` object, which is a bundle of certain operations for it.

`Streamable[T]` is called a *type class*. You can think about it as a trait defining an entire class of types that support a certain operation. Type classes are interfaces of functional programming.

8.2.2 With Rich Wrappers

We have previously seen how the Rich Wrapper pattern can be used to inject new methods into classes. We can use it again, to inject entire interfaces. Type classes, that is:

`Streamable.scala`, 49259c

```scala
implicit class StreamableOps[A](a: A)(implicit typeclass:
   Streamable[A]) {
```

```
  def stream: () => InputStream = typeclass.stream(a)
}
```

Here, we see an example of an implicit argument to a constructor.

`StreamableOps`, hence, is a rich wrapper, polymorphic on the type `A`. It is capable of injecting its methods in any type `A` as long as a `Streamable[A]` is present in scope.

For other types, the `Streamable[A]` will not be found and hence it will not be possible to call the conversion. The logic of the injected `stream` method is delegate do `Streamable[A]`.

With this Rich Wrapper in scope, we can write the code as follows:

```
def f[T](target: T)(implicit streamable: Streamable[T]): Unit
    = {
  target.stream
}
```

The polymorphism on `T` is the same as without Rich Wrappers, but now we have the nice OOP syntax.

8.2.3 Putting it all together: the Type class pattern

The approach described in the two sections above forms a pattern. As a rule, a type class is composed of:

- A trait with one or more type arguments (the types the type class is defined for) and the operations on these types.
- A companion object with some convenience methods and default implementations. A common convenience method is `def apply[T] = implicitly[Typeclass[T]]`, where `Typeclass` is the type class in question. This way, you can easily get instances of the type class via writing `Typeclass[T]` instead of `implicitly[Typeclass[T]]`. `implicitly` is a method defined in `scala.Predef` (which is imported automatically by the compiler) that searches an implicit value of the given type in the implicit scope.
- The syntactic sugar defined either as an `implicit class`, or an ordinary class (trait) and an implicit conversion from `T` to that class, provided there is a type class implementation for `T`: `def toOps[T](implicit tc: Typeclass[T]) = ???`

8.3 Implementation

Next step is to inject `Streamable` interface into `String` and `File` by implementing type class instances for them:

`Streamable.scala`, 49259c

```
object Streamable {
  implicit val streamableFile: Streamable[File] = f =>
    () => FileUtils.openInputStream(f)

  implicit val streamableString: Streamable[String] = str =>
    () => IOUtils.toInputStream(str, defaultEncoding)
}
```

Since the trait contains only one method and we are working in Java 8, we can implement the entire trait with a simple lambda. This lambda will be interpreted as that single method's implementation by Java 8, and the trait will be created.

After we implemented that, we can rewrite the server handler as follows:

MainJVM.scala, 49259c, @@ -6,9 +10,13 @@

```
object MainJVM {
+  implicit class AssefFileString(str: String) {
+    def assetFile: File = new File(s"assets/$str")
+  }
+
   def main(args: Array[String]): Unit = {
     val server = createServer(8080) {
-      case GET -> "/"                       => serveFile("html/index.html")
-      case GET -> "/js/application.js" => serveFile("js/application.js", "application/javascript")
+      case GET -> "/"                       => Response("html/index.html"  .assetFile.stream, "text/html"              )
+      case GET -> "/js/application.js" => Response("js/application.js".assetFile.stream, "application/javascript")
     }
     server.start()
   }
}
```

And the default error handler becomes:

ServerAPI.scala, 49259c, @@ -41,7 +40,1 @@

```
- val defaultErrorHandler: TotalHandler = _ => serveString("Not Found", 404)
-
- def serveFile(path: String, contentType: String = "text/html"): Response =
-   Response( () => FileUtils.openInputStream(new File(s"assets/$path")), contentType, 200 )
-
- def serveString(str: String, responseCode: Int): Response =
-   Response( () => IOUtils.toInputStream(str, defaultEncoding), "text/plain", responseCode )
+ val defaultErrorHandler: TotalHandler = _ => Response("Not Found".stream, responseCode = 404)
```

We have removed the unnecessary now `serveFile` and `serveString` methods.

8.4 Implicit Scope

The rules for the implicit scope are complex, but here are some rules of thumb on what ends up on the implicit scope[1]:

- Imported implicit values.
- Implicit values defined in the companion objects of the target type. For example, if you need an implicit value of type `A[B[C[D]]]`, the companions of `A`, `B`, `C` and `D` will be searched. This

[1]More information on how the implicits are looked up can be found here: https://github.com/milessabin/export-hook/blob/master/README.md

should explain why we defined the `Streamable` implementations under the `Streamable` companion object.
- If you can call an implicit value or conversion without specifying its fully qualified name, most probably it is on the implicit scope.

8.5 Conclusion

Rich Wrappers allow you to inject methods into types. Type classes allow you to inject interfaces.

Sometimes you want several classes to support the same set of methods. This rises a natural desire to call them polymorphically.

In OOP, if several classes define the same set of methods, they are abstracted to an interface.

In case the methods are not defined by the classes but are injected with Rich Wrappers, polymorphism is achieved via type classes.

In essence, type class instances are objects used as bundles of methods supported by some types. You can also have them in plain Java (e.g. `Comparator` that defines an operation of comparison over some type). In Scala, the implicits mechanism allows to almost completely hide the fact that your are using them, hence they are used much wider in this language than in Java.

III

Client Side

9. Ajax with Circe

M Technical: Set up the basic Ajax capability

C 27b0b0 to 7a3188

Our server looks good enough. It is time to start the work on the client side. Since we are building a single-page application, the first step for the client-side should be Ajax implementation. We will start from the simplest Ajax protocol possible - the echo protocol, where the server should respond with the same message as the client sent it.

9.1 Protocol

A natural way to model the communication protocol is the case classes:

Protocol.scala, 27b0b0

```scala
package functionalstreamer

case class EchoReq (str: String)
case class EchoResp(str: String)
```

We will also need the way to convert these case classes to JSON String. Circe[1], the JSON library for Scala, is convenient for these purposes, so we will include it in our build:

build.sbt, 27b0b0

```scala
, libraryDependencies ++= Seq(
    "com.lihaoyi" %%% "scalatags" % ScalaTags

  , "io.circe"    %%% "circe-core"    % Circe
  , "io.circe"    %%% "circe-generic" % Circe
  , "io.circe"    %%% "circe-parser"  % Circe
  )
```

[1]https://circe.github.io/circe/

Circe give us two methods: `decode[T](x: String): Either[CirceError, T]` - to decode a string to some case class `T`, and `toJson`, which is injected to all the case classes and sealed traits to convert them to JSON strings.

All this comes for free, you do not need to implement anything. This is achieved via type classes and rich wrappers we have seen previously. Case classes are analysed on compile time and their JSON representation is inferred via it.

9.2 Client side

The idea is that the client should send an Ajax request to a certain server endpoint, wait for a response and update the page accordingly.

First we need to do some Circe imports:

MainJS.scala, 27b0b0

```scala
import io.circe.parser.decode
import io.circe.generic.auto._, io.circe.syntax._  // Implicit
    augmentations & type classes
```

As you recall, type classes and rich wrappers rely on implicits extensively. These, and the capabilities to infer JSON representations of classes on compile time (also implicits and probably some macros) are imported here.

After that, it is easy to implement the required behavior using standard ScalaJS capabilities:

MainJS.scala, 27b0b0

```scala
def main(): Unit = window.onload = { _ =>
  val placeholder = document.getElementById("body-placeholder"
    )

  val req = EchoReq("Hello from Ajax")

  Ajax.post(url = "/api", data = req.asJson.noSpaces)
    .map     { req => decode[EchoResp](req.responseText) }
    .flatMap {
      case Right(resp) => Future.successful(resp)
      case Left (err ) => Future.failed    (err )
    }
    .onComplete {
      case Success(EchoResp(str)) => placeholder.innerHTML =
         str
      case Failure(err: AjaxException) => placeholder.
         innerHTML = err.xhr.responseText
      case Failure(err) => placeholder.innerHTML = s"Unknown
         error: ${err.toString}"
    }
}
```

`Ajax.post` sends a post request to the endpoint and returns a `Future` of response. We than `map` the `Future` to extract and decode the JSON response into a case class.

A catch here is that Circe's `decode` method may fail, hence it returns an `Either` as was discussed above. If we try to handle `Either` from `onComplete`, we'll end up with a nested match statement: first we need to match for Ajax errors that `Future` may contain, then - for Circe errors the successful `Future`'s result may contain.

To avoid this spaghetti, we `flatMap` the `Future`, failing it if the underlying `Either` contains an error. Since we are effectively converting an `Either` to a `Future` in that `flatMap`, we can as well inject a convenience method into `Either`:

package.scala, 7a3188

```scala
import scala.concurrent.Future

package object functionalstreamer {
  implicit class EitherToFuture[A <: Throwable, B](e: Either[A, B]) {
    def toFuture: Future[B] = e match {
      case Right(b) => Future.successful(b)
      case Left (e) => Future.failed    (e)
    }
  }
}
```

Then we can simplify the client code as follows:

MainJS.scala, 7a3188, @@ -20,11 +19,7 @@

```diff
    Ajax.post(url = "/api", data = req.asJson.noSpaces)
-     .map     { req => decode[EchoResp](req.responseText) }
-     .flatMap {
-       case Right(resp) => Future.successful(resp)
-       case Left (err ) => Future.failed    (err )
-     }
+     .map(_.responseText).map(decode[EchoResp]).flatMap(_.toFuture)
      .onComplete {
        case Success(EchoResp(str)) => placeholder.innerHTML = str
        case Failure(err: AjaxException) => placeholder.innerHTML = err.xhr.responseText
        case Failure(err) => placeholder.innerHTML = s"Unknown error: ${err.toString}"
      }
```

9.3 Server side

Finally, we need to add the Ajax endpoint to the server handler:

MainJVM.scala, 27b0b0

```scala
case e @ POST -> "/api" =>
  val req = IOUtils.toString(e.getRequestBody, defaultEncoding)

  val respOrError: Either[CirceError, Response] =
    decode[EchoReq](req)
```

```
      .map { case EchoReq(str) =>
        Response(EchoResp(s"Echo response: $str").asJson.
          noSpaces.stream, application.json)
      }

  respOrError match {
    case Right(resp) => resp
    case Left (err ) =>
      Response(s"Error occurred while parsing JSON request: ${
         err.toString}".stream, responseCode = 400)
  }
```

The code above:

- Reads the request body - a request JSON.
- Decodes it into the model case class.
- If the decoding went fine, responds with the response created based on the request. Otherwise, respond with an error.

Notice that you can call `map` on `Either`. `map` works on `Right` (which contains successful result by default in most libraries). If this `Either` is `Left`, it is unaffected by `map`. `flatMap` behaves the same way.

10. Effect Types

M Theory. From a function, one should return not only the value computed by it, but also the information about what happened inside - if this information is relevant.

10.1 Motivation

Circe's `decode[A](x: A)` returns `Either[Error, A]`. ScalaJS's `Ajax.post` returns `Future[XMLHttpRequest]`[1]. Why do they all return a result wrapped in some higher-kinded type?

Higher-kinded types are often used to encapsulate side effects that happen during function execution.

10.1.1 Side effects revisited

Let us recall our definition of side effects: these are instructions that are executed in a function that can affect the environment outside the function.

Also recall how we separated the result of function execution into two aspects: the known (the returned value) and the unknown (the side effects that happen withing). The unknown part is undesirable. Hence the motivation to eliminate it by making functions pure. A pure function's execution result is described only by its return type, since the side effects are absent.

Another way to think about the effects is as of "surprises" the function may have. A pure function discloses full information on what it did in its return type, no surprises. With the impure function, the story is different. Here are some examples of the surprises it may have:

- Exceptions. You can not tell a function throws them by its return type. In Java, functions must declare that they throw exceptions, but not in Scala. If a function throws an exception and you do not expect it, it may crash your application. You become surprised.
- Optionality - when a function does not return a value under certain circumstances. Java APIs usually return `null` in such cases. You can not guess such a function by its return type. If it returns `null`, this may cause `NullPointerExceptions` when you try to use that value outside the function. And you become surprised.
- Computations on another thread. When you call a function that creates a new thread with a computation and immediately returns, you most probably need to register a callback

[1] https://www.scala-js.org/api/scalajs-dom/0.9.0/#org.scalajs.dom.ext.Ajax$

somewhere to collect the result. This is not be obvious if the function returns a `Unit`. If you don't know what it does, you don't register the callback, the program goes wrong and you become surprised.

10.1.2 Encapsulating side effects with higher-kinded types

The trick with higher-kinded types is to declare what happened within a function in the value it returns. Here is how the above effects can be encapsulated in Scala higher-kinded types:

- Exceptions - `Try[A]` or `Either[T <: Throwable, A]`. `A` is the value the function originally returns. The fact that it is under `Try` or `Either` means not only `A` is returned, but also some data about how the computation went. You can compare it to how Java declares its exceptions with `throws` in a method's signature. `throws` is a construct specifically introduced for exceptions, however, and higher-kinded types can be generalised to any effect. In any case, if you get a `Try` or `Either` from a function, you no longer can use underlying `A` without acknowledging the fact that the computation may have gone wrong. You are no longer surprised.
- Optionality - `Option[A]`. If the function is not supposed to return a value, it returns `None`, otherwise - `Some[A]`. Again, you can not access `A` without acknowledging the possibility of `None`, and are no longer surprised.
- Async computations - `Future[A]`. As opposed to `Unit`, you clearly see what happened: an async computation resulting in `A`. Also you know where to set the callback - on the `Future`.

The general pattern here is that pure functions return `F[A]` instead of `A`. `A` is the result we are computing, and `F` - the effect type - contains the data of what happened during the function execution.

In you program this way, you will encounter a lot of functions of the form `A => F[B]`.

10.2 Intuition

10.2.1 What is `F[A]`?

It is important to keep in mind that higher-kinded types is a very abstract notion. However, most of the times (but not always!) you will be correct when thinking of them as of structures composed of `A`.

Sometimes the structures are interesting by themselves (as in `List`s or `Map`s), other times they provide supplementary data about how the computation went (as in `Either[Error, A]`).

In case of representing effects, `F[A]` is a structure that contains the result `A`, as well as the data about the computation written in `F`.

Again, this intuition is to be used with caution. Not all `F[A]` can be understood as structures, if it does not work for you, you should seek another analogy.

10.2.2 What is `A => F[B]`?

The same thing as `A => B`, but it writes all the effects that happen in the process of execution into structure `F`.

10.2.3 What about the Onion architecture?

The Onion architecture prevents functions from invoking any effects, just asks them for a description of what they want to do. It executes this description from a single place.

Effect types, in contrast, execute side effects from the function, but they are no longer unknown to the user. They are fully described by the returned structure `F`.

Some things to keep in mind when deciding where you need an Onion, effect types or a mixture of both:

- Onion DRYs effect execution, and hence introduces the dependency on the layer that executes effects. May not always be desirable.
- Some effects may have too little in common or are handled too easily (like `Either` or `Option`, where every case's handling is different). Onion may introduce an overhead of an additional layer.
- Some effects can not be easily by an effect type because of their nature. For example, a database transaction, or a process of writing to a file system. You can capture the effect of errors during this operation, but how do you encapsulate the effect of writing itself? Remember that we consider side effects as something that modifies the environment beyond the scope of the function. In case of an exception or an error, you can eliminate such a modification via `Either` (the application will no longer crush under a runtime exception). But writing to the outside world is a modification outside the scope of the function by definition. So probably Onion is more appropriate in this case: the functions should compute what they need to write, but the actual writing is done in a single, strictly specified place.

10.3 Conclusion

Impure functions do not disclose full information about what happened within to the user. They only disclose the result of their computation. Pure functions disclose the result, as well as the data about what happened during the computation.

11. Monads

M Philosophy of purity combined with modularity results in many functions of type `A => F[B]`. Monads help to execute `A => F[B]` and `B => F[C]` sequentially.

C 7a3188 to e88224

11.1 Motivation

11.1.1 Problem: Concrete

Following this logic and seen the example of Circe, we may see our extensive usage of Apache Commons IO (which is a Java API) under a new angle. It is side effectful.

For example, take this line from the server code:

`MainJVM.scala`, 7a3188

```scala
val req = IOUtils.toString(e.getRequestBody, defaultEncoding)
```

According to the javadoc[1], `IOUtils.toString` throws `IOException` and `NullPointerException`. We do not know how the underlying Sun's server is implemented, so it is quite possible that one of these can occur. We will be surprised when suddenly the server becomes inaccessible due to a runtime exception crushing the application.

We can make it safer by following the Functional paradigm: not only return whatever needs to be computed, but also the data of whether an exception happened. We can do this with errors similarly to how Circe does it - via `Either`:

`MainJVM.scala`, 9de0d8, `@@ -26,8 +28,12 @@`

```scala
      case e @ POST -> "/api" =>
-       val req = IOUtils.toString(e.getRequestBody,
      defaultEncoding)
-
-       val respOrError: Either[CirceError, Response] =
-         decode[EchoReq](req)
-           .map { case EchoReq(str) =>
```

[1] https://commons.apache.org/proper/commons-io/javadocs/api-2.5/org/apache/commons/io/IOUtils.html

```
-         Response(EchoResp(s"Echo response: $str").asJson.
noSpaces.stream, application.json)
-       }
+   val reqOrError: Either[Throwable, String] =
+     Try { IOUtils.toString(e.getRequestBody,
defaultEncoding) }.toEither
+
+   val respOrError: Either[Throwable, Response] =
+     reqOrError.flatMap { req =>
+       val decodedOrError: Either[CirceError, EchoReq] =
decode[EchoReq](req)
+       decodedOrError
+         .map { case EchoReq(str) =>
+           Response(EchoResp(s"Echo response: $str").asJson
.noSpaces.stream, application.json)
+         }
+     }
```

However, `respOrError` depends on the value computed by `reqOrError`. Two computations, both have an error effect, one of them depends on the result of the other. How do we combine them?

11.1.2 Problem: General

If we make all our functions pure, they will all have a form of `A => F[B]`. Given that most applications are modular, we'll end up with many such functions. Since an application is a sum of its modules, we'll need to combine these functions: call `B => F[C]` on the result of `A => F[B]` for example.

How do we do that?

11.2 Solution

11.2.1 `flatMap`

The answer is `flatMap`. Given an `F[A]` and a function `A => F[B]`, `flatMap` is capable of running it on the result of the first `F` to produce `F[B]`.

For example, if `type F[A] = Either[Error, A]`, `flatMap` is capable of running `A => Either[Error, B]` on `Either[Error, A]`'s result to produce `Either[Error, B]`. Remember that `Either` is right-biased: `map` and `flatMap` run on its right side, while ignoring the left one.

If there is a function `A => B`, you can run it on `F[A]` via `map`.

11.2.2 Monads

The majority of the effect types `F[_]` you encounter are so-called *Monads*. A `Monad` is something that has a `flatMap` function similar to the one we saw in `Either` and `Future`. Also a `Monad` has a way to lift any value `A` to a monad `F[A]`: `def pure[A](a: A): F[A])`.

A `flatMap` is defined as follows:

```
def flatMap[A, B](fa: F[A], f: A => F[B]): F[B]
```

Given a monad `F[A]` and a function that takes its result type and computes another monad, `flatMap` is capable to run that function `f` on the result of `fa`.

Another way to view `flatMap` is to see what happens if we define it as follows:

```
def flatMap[A, B](f: A => F[B]): F[A] => F[B]
```

Essentially that is the same code as above: `flatMap` takes a function `f` and returns a function that takes a monad `F[A]` to create `F[B]`. But if you view it this way, `flatMap` "lifts" a function `f: A => F[B]` to its monadic version `fm: F[A] => F[B]`:

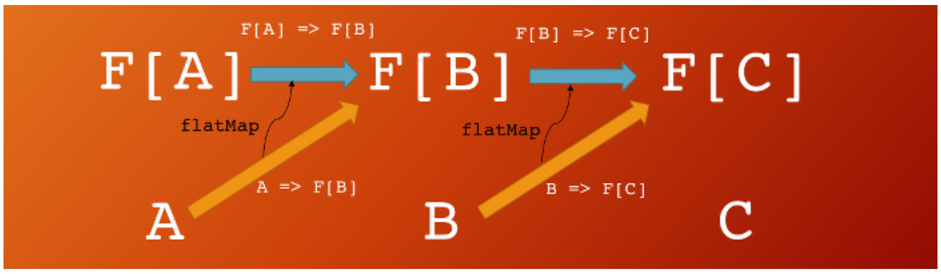


Figure 11.1: FlatMap as a function lift

So, if you have a bunch of functions of the shape `A => F[B]`, you can turn them all into `F[A] => F[B]` and compose them as ordinary functions.

An important catch here is that the monads are used for *sequential*, *dependent* computations: if you have `A => F[B]` and `B => F[C]` and you want to compose them, `B => F[C]` depends on the result of `A => F[B]` and can not be computed until this result is known. This will become important when we will be covering the `Applicative` type class.

11.2.3 Monad type class

Some types may define `flatMap` and `map`, and some may not (by the way, if you have `flatMap: (A => F[B]) => (F[A] => F[B])` and `pure: A => F[A]` - the definition of `Monad` - it is easy to define `map: (A => B) => (F[A] => F[B])`. Compose `pure` and `A => B` to get `A => F[B]`). For the types that do not define them but would benefit from them, it is a common practice to inject them via the type class technique discussed previously.

11.3 Intuition

11.3.1 What is F(A) => F(B)?

It is like `A => F[B]`, but the input is assumed to be computed by another computation with possible side effects. These effects are written into `F`.

Hence, `F[A] => F[B]` must do just the same thing as `A => F[B]` (compute `B` and write side effects to `F`), plus one more: describe how to deal with the input `F` when computing the output `F`. You need to merge the information about side effects of the input computation with the information you will create when computing `F[B]`. Some examples:

- If `F` is `Either` and the input is `Left` (error), we do not need to perform the computation, just return the error. If it is `Right`, we need to extract the result and run the computation. The resulting `Either` thus contains the input computation error, or the output computation error, or the result.
- If `F` is `Future`, we need to wait for the result of the input first. The resulting `Future` thus is a combination of the time you need to wait for the input and the output to be computed.

11.4 Implementation

11.4.1 Naive

We can use `flatMap` to help our problem as follows:

MainJVM.scala, 9de0d8

```scala
val respOrError: Either[Throwable, Response] =
  reqOrError.flatMap { req =>
    val decodedOrError: Either[CirceError, EchoReq] = decode[
       EchoReq](req)
    decodedOrError
      .map { case EchoReq(str) =>
        Response(EchoResp(s"Echo response: $str").asJson.
           noSpaces.stream, application.json)
      }
  }
```

Preventing exceptions is nice, but we have ended up with spaghetti code that is hard to read here. Nested `flatMap` and `map` are not nice, but it sounds like in functional programming, every new `A => F[A]` function you compose adds a new layer of nesting.

11.4.2 Monadic flow

Fortunately, Scala mitigates this with `for`:

MainJVM.scala, e88224

```scala
case e @ POST -> "/api" =>
  val responseOrError: Either[Throwable, Response] = for {
    req     <- Try { IOUtils.toString(e.getRequestBody,
       defaultEncoding) }.toEither
    decoded <- decode[EchoReq](req)
    response = Response(EchoResp(s"Echo response: ${decoded.
       str}").asJson.noSpaces.stream, application.json)
  } yield response

  responseOrError match {
    case Right(resp) => resp
    case Left (err ) =>
      Response(s"Error occurred while parsing JSON request: ${
         err.toString}".stream, responseCode = 400)
  }
```

In Scala, `for` is just a syntactic sugar that translates to `flatMap` and `map` calls. The rewriting above is equivalent to the previous version.

So, with `for` you are able to write your ordinary sequential code with each line containing a separate statement, while working under the monad. And, hence, not only computing the end result, but also collecting the data about how the computation went in process.

12. Cats

M *A library of type classes. Mostly needed to help with higher kinded types:* `F[A]` *and* `A => F[B]`*.*

C `e88224`

12.1 Motivation

There is a method called `merge` injected in `Either` by `MergableEither`[1] rich wrapper. Given an `Either[A, A]`, it returns `A` whichever `A` it contains - left or right.

We could replace the `match` statement with it when handling the API requests from the server side:

`MainJVM.scala`, 588767, `@@ -34,5 +37,3 @@`

```
-  responseOrError match {
-    case Right(resp) => resp
-    case Left (err ) =>
-      Response(s"Error occurred while parsing JSON request:
   ${err.toString}".stream, responseCode = 400)
-  }
+  responseOrError.leftMap { err =>
+    Response(s"Error occurred while parsing JSON request: ${
   err.toString}".stream, responseCode = 400)
+  }.merge
```

`Either` represents a result of a computation of `Response` value that can result in errors. The idea is to map the error (the `Left` side) to a response to get `Either[Response, Response]`. And then merge it.

The problem is, we do not have `leftMap` defined on `Either`. Out of the box, we can only `map` its right hand side.

[1] http://www.scala-lang.org/api/current/scala/util/Either$$MergeableEither.html

12.2 Solution

12.2.1 Cats - the library for Functional Programming

Let us recall two ideas we have discussed recently:

- Type classes are interfaces you can inject into existing types without the need to be in control of their definition.
- Since functional programming relies on `flatMap` extensively to compose pure effectful functions[2] on the type level, a very popular use case for the type classes is a `Monad`.

If type classes are not dependent on a code you may want to inject them in, would it not make sense if someone implemented a library of type classes?

If the `Monad` type class is so useful for functional programming with higher-kinded types, maybe there are others? If there are others, maybe there are libraries of type classes for functional programming, to simplify handling these `F[A]`'s?

Turns out, there are. One of them is Cats[3]. It defines many type classes, including the `Monad`[4], that you can inject into your types. Also it provides implementations of these type classes for the standard types, such as `Either`.

12.2.2 Functor

A fancy name for something with a `map: (A => B) => (F[A] => F[B])` method defined on it is *Functor*.

12.2.3 Bifunctor

This works well for `F[A]`, but if you have a type with two holes - `F[A, B]`, such as our `Either` - you may want a *Bifunctor*. A `Bifunctor`[5] is a fancy name for something with two holes where you can map both of them:

```scala
def bimap[A, B, C, D](fab: F[A, B])(f: (A) => C, g: (B) => D):
    F[C, D]
```

It is easy to derive `map` and `leftMap` from `bimap` by providing the `identity` function to the hole you are not interested in.

Now let us inject `Bifunctor` into `Either` and make the code above work:

`MainJVM.scala`, 588767

```scala
import cats.instances.either.catsStdBitraverseForEither  //
   Type class for Bifunctor (which is a superclass of
   Bitraverse we are importing)
import cats.syntax.bifunctor.toBifunctorOps              //
   Implicit augmentation of types for which Bifunctor is
   available with Bifunctor operations
```

As you recall, to inject a type class you need an instance of it and the rich wrapper for the syntax. Both of them should be on the implicit scope.

Cats stores the rich wrappers for its type classes in the `cats.syntax` package, and the instances of the type classes for certain standard types in the `cats.instances` package. It is easy to find the ones you need via scaladoc[6].

[2]"pure effectful" is used in the sense that all the effects that happen in the function are fully reflected in its return value.

[3]The other one is ScalaZ, which gradually declines in popularity.

[4]http://static.javadoc.io/org.typelevel/cats-core_2.12/0.9.0/cats/Monad.html

[5]http://static.javadoc.io/org.typelevel/cats-core_2.12/0.9.0/cats/functor/Bifunctor.html

[6]http://javadoc.io/doc/org.typelevel/cats-core_2.12/0.9.0

After you've added these imports to the `MainJVM.scala`, the code with the `leftMap` should work.

IV

Video Streaming

13. More Onions

M Technical: Refactor the client-side Ajax input-output infrastructure to the Onion Architecture

C 5b6cb98 to d9a35c

We will want to support many types of AJAX requests. Hence, it is reasonable to decouple their processing from the server handler:

MainJVM.scala, 5b6cb98 to d9a35c, @@ -43,2 +45,4 @@

```
-  def handleApi(request: EchoReq): EchoResp =
-    EchoResp(s"Echo response: ${request.str}")
+  def handleApi(request: APIRequest): Either[Throwable,
   APIResponse] = request match {
+    case EchoReq(str) => Right(EchoResp(s"Echo response: $str
   "))
+    case _ => Left(ServerError(s"Unknown JSON API request:
   $request"))
+  }
```

Notice that we allow errors to happen by specifying `Either` as return type. This costs us nothing since we are using the monadic flow under `Either` in the endpoint handler. The handler itself can be modified as follows:

MainJVM.scala, 5b6cb98 to d9a35c, @@ -30,16 +31,19 @@

```
   case e @ POST -> "/api" =>
     (for {
       req       <- Try { IOUtils.toString(e.getRequestBody,
          defaultEncoding) }.toEither
-      decoded   <- decode[EchoReq](req)
-      respJson   = handleApi(decoded)
+      decoded   <- decode[APIRequest](req)
+      respJson <- handleApi(decoded)
```

At the client side, we will be calling the AJAX API extensively. Hence, we need to abstract the API calling logic from the processing of its results.

Further, the processing of the results involve manipulation of the page. This is an effect. We can draw parallels with what we have seen on the server side. There, we had a need to produce effects from request handlers, saw the dangers of this approach and decided to use the Onion architecture. It is a good idea to apply it here too, since the situations are similar.

ClientOperation.scala, d9a35c

```scala
package functionalstreamer

sealed trait ClientOperation
case class RenderString(str: String) extends ClientOperation
```

First, we define the types describing our effects. So far, we only need to render a String. Next, we apply the Onion architecture to the client side code:

MainJS.scala, d9a35c

```scala
object MainJS extends JSApp {
  def placeholder = document.getElementById("body-placeholder")

  def main(): Unit = window.onload = { _ =>
    ajax(EchoReq("Hello from Ajax")).onComplete(renderResponse)
  }

  def handleApi(response: APIResponse): Either[Throwable, ClientOperation] = response match {
    case EchoResp(str) => Right(RenderString(str))
    case _ => Left(ClientError(s"Can not handle $response"))
  }

  def ajax(request: APIRequest): Future[ClientOperation] =
    for {
      response  <- Ajax.post(url = "/api", data = request.asJson.noSpaces)
      respText   = response.responseText
      decoded   <- decode[APIResponse](respText).toFuture
      operation <- handleApi(decoded).toFuture
    } yield operation

  def renderResponse(response: Try[ClientOperation]): Unit = response match {
    case Success(RenderString(str)) => placeholder.innerHTML = str

    case Failure(err: AjaxException) => placeholder.innerHTML = s"Ajax exception: ${err.xhr.responseText}"
    case Failure(err) => placeholder.innerHTML = s"Unknown error: ${err.toString}"
  }
}
```

Note how `ajax` method computes the operation to be performed in a `Future`, but does not do anything. The logic to handle the operation and produce effects is concentrated under `renderResponse`.

14. Browsing the Directories

M Technical: Implementing the capability to browse file system via Ajax

C `d118ba` to `f48f9c`

Now we are ready for the next step: let us view the file system from the web application.

On the client side, let us show the file system us a list of files and folders. Folders should be clickable. On click, the contents of that folder should be retrieved and the page should display that content.

14.1 Basics

First, let us define the model and the protocol:

`Model.scala`, `f48f9c`

```scala
package functionalstreamer

case class FileModel(path: String, name: String, tpe: FileType)

sealed trait FileType
object FileType {
  case object Directory extends FileType
  case object Misc      extends FileType
}
```

`Protocol.scala`, `f48f9c`

```scala
package functionalstreamer

sealed trait APIRequest
case class DirContentsReq(path: String) extends APIRequest

sealed trait APIResponse
```

```
case class DirContentsResp(contents: List[FileModel], parent: Option[FileModel]) extends APIResponse
```

Now let us define the server-side handler for that request:

`MainJVM.scala`, `d118ba` to `f48f9c`, `@@ -45,5 +63,11 @@`

```
    def handleApi(request: APIRequest): Either[Throwable, APIResponse] = request match {
-     case EchoReq(str) => Right(EchoResp(s"Echo response: $str"))
+     case DirContentsReq(path) =>
+       for {
+         contents       <- path.file.contents
+         contentsPaths   = contents.map(_.toModel)
+         maybeParent     = path.file.parent.map(_.toModel.copy(name = ".."))
+       } yield DirContentsResp(contentsPaths, maybeParent)
+
      case _ => Left(ServerError(s"Unknown JSON API request: $request"))
    }
```

We have injected a bunch of convenience methods in the `File` class:

`MainJVM.scala`, `f48f9c`

```
implicit class FileAPI(file: File) {
  def contents: Either[Throwable, List[File]] = Try { file.listFiles }.toEither
    .filterOrElse(null !=, ServerError(s"Error occurred while retrieving the contents of the file: $file"))
    .map(_.toList)

  def toModel = FileModel(file.getAbsolutePath, file.getName, file.tpe)

  def tpe: FileType = file match {
    case _ if file.isDirectory => FileType.Directory
    case _                     => FileType.Misc
  }

  def parent: Option[File] = Some(file.getParentFile).filter(null !=)
}
```

14.2 Client Side

Now we need to render the response on the client side, from `handleApi` method. We expect that we will also render the contents of the files in the future, so it is convenient to encapsulate the rendering logic into a separate method `view`.

MainJS.scala, d118ba to f48f9c, @@ 21,4 23,4 @@

```diff
   def handleApi(response: APIResponse): Either[Throwable, ClientOperation] = response match {
-    case EchoResp(str) => Right(RenderString(str))
+    case resp @ DirContentsResp(files, parent) => view(resp).map(RenderTag)
     case _ => Left(ClientError(s"Can not handle $response"))
   }
```

We have added a new RenderTag client operation. We will be using Scalatags[1] to describe the HTML views, so we need a separate operation to describe that.

And here is the view itself:

MainJS.scala, f48f9c

```scala
def view(x: Any): Either[ClientError, HtmlTag] = x match {
  case DirContentsResp(files, parent: Option[FileModel]) =>
    for {
      filesViews <- files.map(view)
        .foldLeft[Either[ClientError, List[HtmlTag]]](Right(Nil)) {
          (listOrError, nextOrError) => for {
            list <- listOrError
            next <- nextOrError
          } yield list :+ next
        }

      maybeParentView <- parent.map(view) match {
        case Some(either) => either.map(Some(_))
        case None         => Right(None)
      }
      listItems = (maybeParentView ++ filesViews).map { f => li(f) }.toList
    } yield ul(listItems)

  case FileModel(path, name, FileType.Directory) =>
    Right( button(onclick := ajaxCallback(DirContentsReq(path)))(name) )

  case FileModel(path, name, _) => Right(p(name))

  case _ => Left(ClientError(s"Can not render view: $x"))
}
```

MainJS.scala, f48f9c

```scala
def ajaxCallback(request: APIRequest): () => Unit =
  () => ajax(request).onComplete(renderResponse)
```

We have built it in a modular manner: every fragment of the view is rendered by a separate case clause, so it becomes easier to compose larger fragments from the smaller ones.

[1]https://github.com/lihaoyi/scalatags

We have also build the `view` with the possibility errors in mind: e.g. when a view for something was not found. We do not want the errors to be silently logged to the browser console.

Now, looking at the spaghetti under `DirContentsResp` you probably regretted you have ever entertained the idea of using functional programming in your projects. Hang on.

Let us first see what is going on under `DirContentsResp`.

14.3 Directory Rendering

14.3.1 Contents

We have a list of files to render as an unordered list, and the link to the parent directory on top, if it is present. Each file can be rendered by a separate call to `view` that returns an `Either`. The rendering of `DirContentsResp` itself should return `Either`.

When we need to produce a monad out of the values of several others, it is a familiar situation. We need to combine them with a monadic flow `for`.

However, the model of the directory contents is a `List`. We can `map` it to the views, but then we will get `List[Either[Error, File]]`. And we do not know how to combine a `List` of monads under a monadic flow - not in the nice way.

The next best thing to do is to compute an `Either[Error, List[HtmlTag]]`. In an ordinary monadic flow, we have every monad's result stored in a certain variable (e.g. `a <- b` extracts the result of `b` to `a`). If we compute an `Either[Error, List[HtmlTag]]`, we won't have every `HtmlTag` referenced by a separate variable, but we will have a variable for the `List[HtmlTag]`. Good enough. The code below does just that - it turns `List[Either[Error, HtmlTag]]` into an `Either[Error, List[HtmlTag]]`:

MainJS.scala, f48f9c

```scala
filesViews <- files.map(view)
  .foldLeft[Either[ClientError, List[HtmlTag]]](Right(Nil)) {
    (listOrError, nextOrError) => for {
      list <- listOrError
      next <- nextOrError
    } yield list :+ next
  }
```

14.3.2 Parent

Same thing happens with `parent`. It is an `Option[FileModel]` that can be mapped to an `Option[Either[Error, HtmlTag]]`. As previously, we are working under `Either` so we can't have that. And, as previously, the next best thing is to swap the two effects - `Option` and `Either`:

MainJS.scala, f48f9c

```scala
maybeParentView <- parent.map(view) match {
  case Some(either) => either.map(Some(_))
  case None         => Right(None)
}
listItems = (maybeParentView ++ filesViews).map { f => li(f)
   }.toList
```

Then it is straightforward to construct a HTML list and return it under the `Either` monad.

15. Applicative

M Combine pure, effectful independent computations. As opposed to `Monad`, that is used to combine dependent computations.

C 855922

15.1 Motivation

15.1.1 Concrete

Obviously the code from the previous chapter is horrible. Aside from looking bad, the error reporting is not satisfactory. We are working from a monadic flow based on `for` (sugar for `flatMap`). In case of `Either`, `flatMap` terminates at first `Left` encountered. Meaning if there are several view errors, only one error at a time will get reported. This is not satisfactory, however, since by our design views are composed of smaller views that are independent one from another. And hence, we would like to have errors for all the views at once, not just the first one.

15.1.2 General

Remember how we emphasized that `Monads` are good to compose *dependent*, *sequential* computations. `A => F[B]` and `B => F[C]`, where the second function can not be called without the result of the first one.

In our case, this is not so. The views for each fragment of the page are computed *independently*: you do not need the result of rendering one file in a list to render another one.

Only once you have computed the views for all the fragments in an independent manner, you use them to assemble the total view. Fragment views are independent one from another, the total view is dependent on all the fragments.

So the monadic solution we have is suboptimal here, since it solves a different problem and places an unnecessary constraint (dependency one on another) on our fragment views.

15.2 Solution

15.2.1 Applicative

Applicative is a fancy name for a type class that allows to zip several higher-kinded types. Given `a: F[A]` and `b: F[B]`, you can produce `ab: F[(A, B)]` out of them, as follows:.

```
val c: F[(A, B)] = (a |@| b).tupled
```

You can afterwards `map` this tuple to compute something based on the combined result of `a` and `b`.

`a` and `b` are computed independently. Then you can make another computation depend on their result via `map`ping the combined version of the two.

15.2.2 Technical stuff

There's a bit more going on under the hood:

- Technically, it is `Cartesian` type class that builds tuples. `Cartesian` is defined by `def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]`. However, how do you build a 3-tuple of based on that definition? So clearly `Cartesian` on its own is a bit weak. The companion object of it defines the logic to construct n-ary tuples, but only in presence of some other type class, `Invariant`.
- `Applicative` is a more powerful version of `Cartesian`. Defined by `def ap[A, B](ff: F[(A) => B])(fa: F[A]): F[B]` and `def pure[A](x: A): F[A]`, not only can it confuse people with the purpose of these two methods, but also use the higher level of abstractions gained by them to produce higher-arity tuples. If you are interested, the best way to see how it is done is to try doing that on your own, say for an `Option`. First try constructing a 3-typle from three `Option`s via bare `product` (and only via it, you are not allowed to use `map` or `flatMap`, since they are a part of a `Monad`), then via `ap`, and you will see why we need `Applicative`.

This is a technical stuff. The important thing about the `Applicatives` is that they allow to zip results of independent effect types into a type of tuple. As opposed to a Monad, that is used to describe sequential computations, where the subsequent computations depend on the results of the previous ones.

15.3 Intuition

15.3.1 What does it mean to zip `F[A]` and `F[B]`?

Say, `((a: F[A]) |@| (b: F[B])).tupled: F[(A, B)]`.

Think of `F` as a structure containing the infromation about the effects that happened during the computation of `a` and `b`. Can be an error (`Either`) an optionality (`Option`), or a delay (`Future`). When we zip two such effects, two things happen:

- Their results are combined into a tuple: `A, B => (A, B)`.
- Their wrapping effect types, `F`'s, get composed in some way. It is reasonable: we feed two `F`'s as an input, but get one `F` as an output, so we must be combining the two inputs in some way.

How does it make sense to combine effects? This is individual to the effect in question.

- `Option`: if at least one of the two is `None`, the result effect is `None`.
- `Future`: a future that waits for all the futures you are zipping to finish, then produces the tuple with their results.
- `Either`: if all the `Either`s are `Right`, produce a `Right` with the tuple. If at least one is `Left`, produce `Left` with a combination of the `Left`'s payloads.

15.4 Implementation

In our case, we need to map several `Either`s to the resulting `Either` of HTML list, while logging all the errors in all the `Either`s.

It turns out that the standard Cats' implementation of Applicative for `Either` does the same thing as the `Monad` does: if at least one `Either` is `Left`, take the first `Left` encountered.

Fortunately, we can write our own implementation:

`typeclasses.scala`, 855922

```scala
package functionalstreamer

import cats.{Applicative, Monoid}
import cats.syntax.monoid._

object typeclasses {
  implicit def applicativeEither[A: Monoid, ?]: Applicative[
      Either[A, ?]] = new Applicative[Either[A, ?]] {
    def pure[B](x: B): Either[A, B] = Right(x)

    def ap[B, C](ff: Either[A, B => C])(fa: Either[A, B]):
        scala.util.Either[A, C] = (ff, fa) match {
      case (Right(f), Right(b)) => Right(f(b))
      case (Left (a), Right(b)) => Left(a)
      case (Right(b), Left (a)) => Left(a)
      case (Left(a1), Left(a2)) => Left(a1 |+| a2)
    }
  }
}
```

First, some clarifications:

- `implicit def applicativeEither[A: Monoid, ?]` is a typo. Should be `implicit def applicativeEither[A: Monoid]`.
- `?` is a syntax that becomes available due to the Kind projector[1][2] plugin. `F[A, ?]` is the same as `type FA[X] = F[A, X]; FA`. That is, when we need to partially specify some holes in the type in question.
- `A: Monoid` syntax means that `A` should have a `Monoid` type class to it. `implicit def applicativeEither[A: Monoid]` translates directly to `implicit def applicativeEither[A](implicit somerandomname: Monoid[A])`.
- A Monoid is a type class. It is defined by `def combine(x: A, y: A): A` and `def empty: A`. So `Monoid[A]` is just a fancy way to say that you can combine two `A`s in some way and have a default value of `A`. For example, one implementation of `Monoid[Int]` can `combine` two `Int`s by adding them, and an `empty` of `Int` is `0`.

`Applicative` is defined on some higher-kinded type `F[_]`. It means you can not define `Applicative[Int]` (because `Int` does not have any holes in its type definition), but you can define an `Applicaive[Option]` (since `Option[A]` has one hole - `A`). An intuition is that `Applicative` works on structures, not individual types.

Applicative is defined on `F[_]` and not `F[A]`, because the holes of `F` are known on the level of `def ap[A, B](ff: F[(A) => B])(fa: F[A]): F[B]`. It is reasonable: would be weird if we could zip `Option[Int]` with `Option[String]`, but not some arbitrary `Option[A]` and `Option[B]` - given how easily we can create a tuple of `(A, B)` from any `A` and `B`, and the fact that the structure - `Option` - is the same in both cases.

In other words, we need polymorphism in `ap`.

[1]GitHub: https://github.com/non/kind-projector

[2]More information: http://underscore.io/blog/posts/2016/12/05/type-lambdas.html

This is why we need the `?` syntax: `Applicative` needs `F[_]`, but `Either[_, _]` has two holes. So we need to partially apply that type definition to some arbitrary type `A` on the left, and specify the other hole with `?`. In other words, we need to specify which hole we are defining the `Applicative` for.

15.4.1 Inside `ap`

`ap` is defined as:

```scala
def ap[A, B](ff: F[(A) => B])(fa: F[A]): F[B]
```

In case of zipping `F[A]` with `F[B]`, you use `A` in a computation that produces `F[(A, B)]`. In case of `ap`, you use `A` in an arbitrary computation `A => B`. So essentially it is similar to zipping, just more powerful.

We argued above that zipping two effect types involves zipping their result types into a tuple and combining the effect structures themselves. In case of `ap`, we need to do the same thing, except that we need to combine `A => B` and `A` not by producing a tuple but by applying the function to the value. It is equally easy to do, so does not change the big picture.

How do we compose `Either` effects? If both of them are `Right`, compose their results. If at least one is `Left`, we need to produce `Left` with the combination of all the `Left` results.

This is where `Monoid` comes into play. It allows to combine the two `A`s, whichever they are, as long as we know that a `Monoid` is defined for them.

This is a nice polymorphism similar to the Java interface one. In Java style, you would probably define an interface (say `Combinable`) and use it in place of `A`: `F[Combinable]`. Except that you can not inject interfaces into existing classes, but you can do so with the type class technique.

15.4.2 Application

With that type class in place (don't forget to import it!) we can combine the directory contents view and the parent button view via applicatives:

MainJS.scala, 855922

```scala
case DirContentsResp(files, parent: Option[FileModel]) =>
  val filesViewsEither = files.map(view)
    .foldLeft[Either[ClientError, List[HtmlTag]]](Right(Nil))
       {
      (listOrError, nextOrError) => for {
        list <- listOrError
        next <- nextOrError
      } yield list :+ next
    }

  val maybeParentViewEither = parent.map(view) match {
    case Some(either) => either.map(Some(_))
    case None         => Right(None)
  }

  (filesViewsEither |@| maybeParentViewEither).map { (
     filesViews, maybeParentView) =>
    val listItems = (maybeParentView ++ filesViews).map { f =>
        li(f) }.toList
    ul(listItems)
  }
```

We have replaced the monadic flow, `for`, with `|@|`. Then we mapped the two `Either`'s results to get a combined result under `Either`.

15.5 Conclusion

`Monads` are needed to compose sequential, dependent computations: `F[A] => F[B] => F[C] => F[D]`.

`Applicatives` are needed to combine independent computations:
`(F[A], F[B], F[C]) => F[(A, B, C)]`.

Notice how in the `Monad` case, every subsequent `F` can not be computed without the previous `F`. In the `Applicative` case, all the `F`'s are computed independently (more precisely, we do not describe how to compute them; they are already computed) and can be passed in a single input block.

16. Traverse

M *Combine collections of pure, effectful independent computations. An* `Applicative` *generalized to collections.*

C 855922 to 5d29ac

16.1 Motivation

We have combined two `Either` into an `Either` of tuple. But in the same code, we also combine a `List` of `Either` into an `Either` of a `List`. A `List` is just a large tuple if you think about it, so if we can build one, we can probably build the other.

16.2 Solution

For this purposes, we have the *Traverse* type class. This is how its `traverse` method of `Traverse[F[_]]` is defined:

```scala
def traverse[G[_], A, B](fa: F[A])(f: (A) => G[B])(implicit
   arg0: Applicative[G]): G[F[B]]
```

More confusing signatures!

Here is some intuition:

- `F[A]` is a structure of some kind. A collection, like a `List[A]`.
- `G[B]` is an effect of some kind. Like `Either[Error, B]` - a result of an error-prone computation.
- `A => G[B]` is the error-prone computation in question. For example, `view: Any => Either[Error, HtmlTag]` runs on `Any` to compute a HTML view of that `Any`, but may fail with an error - for example, when view is not defined for that input.

If you map `F[A]` with `A => G[B]`, you will get `F[G[B]]` - a collection of values together with the data on the side effects they were computed with. However, it makes more sense to have a collection of pure values with the combined data on how all of the computations went. All the data in one structure.

if you can combine effects, you can achieve that. Then, the collection of values under a separate effect each turns into a collection of values under a single combined effect.

In our case, a `List[Either[Error, HtmlTag]]` becomes an `Either[Error, List[HtmlTag]]`.

16.3 Implementation

Here is how to do that with `traverse`:

`MainJS.scala`, 5d29ac

```
case DirContentsResp(files, parent: Option[FileModel]) =>
  (files.traverse(view) |@| parent.traverse(view)).map { (
     filesViews, maybeParentView) =>
    ul( (maybeParentView ++ filesViews).map { f => li(f) }.
       toList ) }
```

17 lines got reduced to just 2 - is that not beautiful?

17. Browsing the Videos

M *Technical: Implementing the video browsing capability*

C `784c4b` to `89b44e`

IMPORTANT: `784c4b` introduces a hard-to-track bug specific to Circe. If you checkout that commit and do a clean compile (that is `sbt all:clean` and then `sbt compile`), you'll get a cryptically sounding compile time error. However, if you do incremental compilation (that is you already have the previous commit compiled, there is a chance that you will not get the error. This error is due to the fact that Circe heavily employs type-level programming and macros to do its job, and they are error prone. In other words, Circe's `decode` method messes up the compilation. This kind of errors is inevitable in cutting-edge libraries, and chances are that by the time you will be reading this book there is already a newer version of Circe where this bug is fixed.

The cure for the bug is to insert a `FileType.Parent` statement (yes, you got it right, a single line that only mentions that singleton object) at the top of the `MainJVM` object. The trick is to have all the subclasses of whatever trait you are decoding with `decode` mentioned prior to the `decode` call. This unfortunate bug will be fixed in two commits, in `d0eabe`.

Now that we have the navigation through the file system, let us implement the ability to watch individual video files.

Long story short, we do that via detecting video files by extensions and making them clickable. On click, we send an Ajax request to the server requesting the URL of the video stream of that file. When this URL arrives, we use HTML5 `video` tag to display the video.

All this is done the same way the directory retrieval works. If you are interested in details, you can have a look at the commits: `784c4b`.

One thing deserves our attention though. Many directories contain multiple video files. And it is convenient to have the "Previous" and "Next" buttons to navigate to neighboring files from a video file view. So we include the information about the neighbors on the server side in the response:

MainJVM.scala, 784c4b

```
case VideoReq(path) =>
  for {
    parent        <- path.file.parentModel.toEither
    maybePrevious <- path.file.leftNeighborOfType (Set(
       FileType.Video)).map(_.map(_.toModel))
    maybeNext     <- path.file.rightNeighborOfType(Set(
       FileType.Video)).map(_.map(_.toModel))
    streamPath     = s"/video/$path"
  } yield VideoResp(path.file.getName, streamPath, parent,
     maybePrevious, maybeNext)
```

`leftNeighborOfType` and `rightNeighborOfType` look for the first file of a given type to the left and right of the target file respectively. You may have already spotted something weird about them: we end their lines with `.map(_.map(_.toModel))`, which is confusing.

In the next chapter, we will see what exactly the problem here is and how to solve it.

18. Monad Transformers

M Diversity in effects: you may have not only `F[_]`, but also `G[_]`. Monad transformers are needed to stack them – `F[G[_]]` – painlessly.

C 784c4b to 14b95b

18.1 Motivation

MainJVM.scala, 89b44e

```scala
def leftNeighbor  = neighbor(_ => true) { case f :: `file` :: Nil => f }
def rightNeighbor = neighbor(_ => true) { case `file` :: f :: Nil => f }
```

MainJVM.scala, 89b44e

```scala
private[this] def neighbor(filter: File => Boolean)(predicate: PartialFunction[List[File], File]): Either[Throwable, Option[File]] =
  parent.traverse(_.contents).map { mCts: Option[List[File]] =>
    mCts.flatMap(_.filter(filter).sliding(2, 1).collectFirst(predicate))
  }
```

These three methods are defined in the `FileAPI` rich wrapper that injects some convenience methods into `File`.

The `left` and `right` methods are both defined in terms of the `neighbor` method. The `neighbor` method and does the following:

- Looks ugly.
- Obtains the parent `File` of the file we are injecting the methods into.
- Obtains the contents of the parent.
- Filters them by a given predicate (used to take into account only certain extensions; if you are watching a video, you do not want to get a file that is not supported when you press `Next`).

- Runs `sliding(2, 1)` on them to get a sliding window of size 2 on this collection. A window of such a size is enough to determine whether two files are neighbors.
- Uses `collectFirst` to locate a window with the current file and its required neighbor (left or right). `collectFirst` returns an `Option`, so a neighbor might not exist.

Up until now whenever we've been working with monads, we were working with a single monad type: `Either`. `neighbor` is so ugly, because it operates on more than one effect:

- `parent` returns an `Option[File]`
- `contents` needed to obtain the contents of the parent returns `Either[Throwable, List[File]]`
- `collectFirst` returns again an `Option[File]`

A naive way to go would be as follows:

```scala
parent.map { p: Option[File] =>
  p.contents.map { fs: Either[Throwable, List[File]] =>
    fs.filter(filter).sliding(2, 1).collectFirst(predicate)
  }
}
```

This however returns `Option[Either[Throwable, Option[File]]]`. Very confusing, but it should be more clear if you apply the following intuition:

- Higher kinded types, such as `Option` and `Either`, store effects a function produces.
- In a situation where you have two effects nested, say `F[G[A]]`, you can read it as "`G` after `F`, with the result of `A`".

Hence, `Option[Either[Throwable, Option[File]]]` can be described as `File`, the computation of which has effects of optionality (maybe there is no file after all?), possibly producing an error (say, no access rights to the file system), optionality (again!).

Now we feel two forces:

- How can we have two optionality effects? They would be better off merged into one.
- Our entire application works with monadic flows under `Either` monad, but `neighbor` returns an `Option`. We will not be able to integrate it in our monadic flows this way.

18.2 Solution

18.2.1 Hack

One solution may be to:

- Swap the first two effects, `Option` and `Either`, with `traverse`. We will get `Either[Throwable, Option[Option[File]]]`.
- Use `flatMap` instead of `map` to flatten the second option: `Either[Throwable, Option[File]]`.

This is what the code above does. But it is still ugly.

18.2.2 Problem in-depth

What we witnessed above is a method with nested effects. It emerged as a result of composition of two other methods, each of which produce different effects.

In functional programming we denote a function with effects as `A => F[B]`. We have many such functions due to modularisation. Also it is easy to see that effects can be different. Since the final application is the result of composition of these functions, hitting the problem of different stacked effects, `F[G[_]]`, was inevitable.

We have already seen the it is possible to get to the underlying result `A` via nesting `map`. However, we have also seen how `map` produces duplicate effects: if you have `A => F[B]`, then `B => G[C]`, then `C => F[D]` and try to compose them with a `map`, you will get `F[G[F[D]]]` as an output. If not for `G`, the second `F` would have been merged with the first one with a `flatMap`. In general, you don't want two different structures F to contain two different aspects of the same effect - you want to merge this data into a single F.

We know how to work with single effects. How do we work with composed effects?

18.2.3 Monad Transformers

The solution is to treat `F[G[A]]` as a single effect, a single monad. Then, a `flatMap` on such a monster would work directly on `A`, as opposed to `G[A]`. By definition, this `flatMap` should also return `F[G[B]]`.

When you want to treat stacked monads as a single monad, you want *Monad Tranformers*. These are monads built with stackability in mind. Here is an example of a monad transformer for an `Option` defined by Cats:

```scala
final case class OptionT[F[_], A](value: F[Option[A]])
```

Notice the following things about them:

- They usually are named after the monad they allow stacking upon, plus the letter `T` at the end.
- They usually accept an extra type parameter: `F[_]` in this case. It is specified by the programmer and denotes the type we are stacking on top of this monad.

You can also get a hint of what it really is: `value: F[Option[A]]` means that it is just an `Option` wrapped in `F`. So in our case, we may want to use `Either[Throwable, ?]` (remember, we need one hole but have two!) as `F` to get `value: Either[Throwable, Option[A]]`.

The benefit of `OptionT` is that it has a `Monad` instance defined for it in the companion object:

```scala
implicit def catsDataMonadForOptionT[F[_]](implicit F0: Monad[
  F]): Monad[OptionT[F, ?]]
```

So, it is a monad that works on the second type parameter of `OptionT` - which is `A` in `F[Option[A]]`! Precisely what we need to use monadic flow on stacked effects.

18.2.4 Lifting to monad transformers

A word of caution: `flatMap` takes a function of type `A => OptionT[F, B]`, virtually `A => F[Option[B]]`.

`F[Option[A]]` is contained in `OptionT[F, A]` and is virtually the same as it, but not completely. You can not assign `F[Option[A]]` to a variable of type `OptionT[F, A]`. So, before using it, first you need to create `OptionT` out of it.

Also, if you have `A => F[B]` or `A => Option[B]`, you need to lift their result types to `OptionT[F, B]`. This other effect is either `F[Option[A]]`, or `Option[A]`, or `F[A]`.

Lifting `F[Option[A]]` is straightforward, you just need to pass it to `OptionT` case class constructor.

The two other cases have separate methods in the `OptionT` companion:

- `fromOption` - lifts `Option[A]` to `OptionT[F, A]`.
- `liftF` - lifts `F[A]` to `OptionT[F, A]`.

18.3 Implementation

With this, let us simplify our `neighbor` code.

First, let us import the lift functions:

`MainJVM.scala`, 14b95b

```scala
import cats.data.OptionT, OptionT.{fromOption => liftOpt,
  liftF}
```

Next, we define the F effect type for convenience:

MainJVM.scala, 14b95b

```scala
type Error[A] = Either[Throwable, A]
```

Error would otherwise be expressed as Either[Throwable, ?], which is more verbose. Finally, let us apply use the monad transformers and the monadic flow:

MainJVM.scala, 14b95b

```scala
private[this] def neighbor(filter: File => Boolean)(predicate:
    PartialFunction[List[File], File]): OptionT[Error, File] =
  for {
    p         <- liftOpt[Error](parent)
    contents  <- liftF  [Error, List[File]](p.contents)
    neighbour <- liftOpt[Error]( contents.filter(filter).
       sliding(2, 1).collectFirst(predicate) )
  } yield neighbour
```

Also, note how we avoid nested maps in the request handler:

MainJVM.scala, 14b95b, @@ -97,8 +102,8 @@

```diff
   case VideoReq(path) =>
   for {
     parent        <- path.file.parentModel.toEither
-    maybePrevious <- path.file.leftNeighborOfType (Set(
    FileType.Video)).map(_.map(_.toModel))
-    maybeNext     <- path.file.rightNeighborOfType(Set(
    FileType.Video)).map(_.map(_.toModel))
+    maybePrevious <- path.file.leftNeighborOfType (Set(
    FileType.Video)).map(_.toModel).value
+    maybeNext     <- path.file.rightNeighborOfType(Set(
    FileType.Video)).map(_.toModel).value
     streamPath     = s"/video/$path"
   } yield VideoResp(path.file.getName, streamPath, parent,
      maybePrevious, maybeNext)
```

19. Streaming the Videos

M Technical: Implementing the video streaming capability

C e79da9

Finally, we can implement the video streaming as required by HTML5 video player. It involves being able to handle Range request headers and respond with corresponding byte ranges from the video file. We implement that as yet another case clause in the server request handler. At a glance, the implementation looks as follows:

MainJVM.scala, e79da9

```scala
case e @ GET -> videoPath(path) =>
  (for {
    file      <- Some(new URLCodec().decode(path).file).filter
       (_.exists).toEither
    available <- file.size
    range     <- e.getRequestHeaders.get("Range").asScala.head
       .rangeHeaderWithCeiling(available)
    (from, to) = range
    length     = to - from + 1
  } yield Response(
      payload      = file.stream
    , contentType  = video.mp4
    , responseCode = 206
    , headers      = Map(
        "Accept-Ranges"  ->  "bytes"
      , "Content-Range"  -> s"bytes $from-$to/$available")
    , writeMethod  = Some { (is, os) => IOUtils.copyLarge(is,
       os, from, length) }
    )
  )
  .leftMap { e => Response(s"Error occurred: ${e.toString}".
     stream, responseCode = 400) }
  .merge
```

There are quite a bit of other technical details which we will not be focusing on.

Probably the most interesting detail to note is that with the streaming, we are starting to feel the force of our server being synchronous. It responds to requests from a single thread, so if response involves some heavy operation (such as writing a chunk of a video file), other requests will have to wait a long time.

Due to the fact that we are using the Onion architecture and hence the response logic is concentrated in one place, it is easy to make it multithreaded by wrapping the entire thing in `Future`:

`ServerAPI.scala`, e79da9, `@@ -14,20 +17,24 @@`

```
    def createServer(port: Int)(handler: PartialHandler,
       errorHandler: TotalHandler = defaultErrorHandler):
       HttpServer = {
      val server = HttpServer.create(new InetSocketAddress(
         port), 0)

-     server.createContext("/", { e: HttpExchange =>
-       val Response(payloadIsGenerator, contentType,
    responseCode) = handler.applyOrElse(e, errorHandler)
+     server.createContext("/", (e: HttpExchange) => Future {
+       val Response(payloadIsGenerator, contentType,
    responseCode, extraHeaders, writeMethod) = handler.
    applyOrElse(e, errorHandler)
```

Quite a bit of other things got changed there though: `Response` is now able to accept additional headers (`Accept-Ranges` and `Content-Range` are required to be returned for the stream). Also, we can now specify an optional `writeMethod` that describes how to write the `InputStream` into the response `OutputStream`. We need that because the solution for large files is implemented separately by Apache Commons IO.

20. Afterword

It is my hope that this book gave you a good idea of why functional programming is a thing and how it may be useful in practice. If you really want to learn to apply the concepts discussed here in practice, there is only one way to do so. Practice. No amount of theory and observation will develop the necessary skill and intuition.

So, what further steps will it be a good idea for you to take?

- Pick up a project where you can afford making mistakes and having large delays.
- Resolve to use only the functional style in this project. This means, no mutations and no side effects, at least not in the business logic.
- Stick to this resolution. You will encounter problems with time. Look for the solutions. With each problem you will solve, you will gain more and more experience in functional programming.
- Where to look for the solutions? Probably the fastest way is to ask on Gitter[1] of Cats (or whatever other technique-based library you are using).

I wish you fun and exciting journey into the functional world, and remember: Practice makes perfect!

[1] https://gitter.im/typelevel/cats