Introduction to Functional Architectures

A commit-by-commit dissection of a real application

Anatolii Kmetiuk

FUNCTORHUB.COM

The template used to create this book is credited to www.latextemplates.com.

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`

```
10 def main(args: Array[String]): Unit = {
11   val server = HttpServer.create(new InetSocketAddress(8080),
       0)
12
13   def serveFile(path: String, contentType: String = "text/html
       "): HttpHandler = { e: HttpExchange =>
14     val file   = new File(s"assets/$path")
15     val fileIs = FileUtils.openInputStream(file)
16     val os     = e.getResponseBody
17     try {
18       e.sendResponseHeaders(200, 0)
19       IOUtils.copy(fileIs, os)
20       os.close()
21     } finally {
22       os.close()
23       fileIs.close()
24     }
25   }
26
27   server.createContext("/", serveFile("html/index.html"))
28   server.createContext("/js/application.js", serveFile("js/
       application.js", "text/javascript"))
29   server.createContext("/js/application.js.map", serveFile("js
       /application.js.map", "text/plain"))
30
31   server.start()
32 }
```

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

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`

```
3  import java.io.File
4  import java.net.InetSocketAddress
5  import com.sun.net.httpserver.{HttpServer, HttpHandler,
       HttpExchange}
6
7  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`

```
10 def main(args: Array[String]): Unit = {
11   val server = HttpServer.create(new InetSocketAddress(8080),
         0)
12
13   def serveFile(path: String, contentType: String = "text/html
         "): HttpHandler = { e: HttpExchange =>
14     val file   = new File(s"assets/$path")
15     val fileIs = FileUtils.openInputStream(file)
16     val os     = e.getResponseBody
17     try {
18       e.sendResponseHeaders(200, 0)
19       IOUtils.copy(fileIs, os)
20       os.close()
21     } finally {
22       os.close()
23       fileIs.close()
24     }
25   }
26
27   server.createContext("/", serveFile("html/index.html"))
28   server.createContext("/js/application.js", serveFile("js/
         application.js", "text/javascript"))
29   server.createContext("/js/application.js.map", serveFile("js
         /application.js.map", "text/plain"))
30
31   server.start()
32 }
```

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 @@`

```
25        </a>
26      </div>
27      <div class="ui divider"></div>
28  -   <div>Hello World</div>
29  +   <div id="body-placeholder"></div>
30    </div>
31
32    </body>
```

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

```
1   package functionalstreamer
2
3   import scala.scalajs.js.JSApp
4  +import org.scalajs.dom.{document, window}
5
6   object MainJS extends JSApp {
7  -  def main(): Unit = {
8  -    println("Hello World")
9  +  def main(): Unit = window.onload = { _ =>
10 +    val placeholder = document.getElementById("body-
       placeholder")
11 +    placeholder.innerHTML = "Hello World from JS"
12     }
13  }
```

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

```
6  def main(args: Array[String]): Unit = {
7    val server = createServer(8080) {
8      case e @ GET -> "/"                    => serveFile(e, "html
           /index.html"  )
9      case e @ GET -> "/js/application.js" => serveFile(e, "js/
           application.js")
10   }
11   server.start()
12 }
```

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

```
11 // Types we will use to describe the Server
12 type Handler = PartialFunction[HttpExchange, Unit]
13 type Path    = String
14
15 sealed trait Method
16 case object GET  extends Method
17 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

```
44 def createServer(port: Int)(handler: Handler): HttpServer = {
45   val server = HttpServer.create(new InetSocketAddress(port),
         0)
46   val nativeHandler: HttpHandler = toNativeHandler(handler)
47   server.createContext("/", nativeHandler)
48   server
49 }
```

`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`

```
21 object -> {
22   def unapply(exchange: HttpExchange): Option[(Method, Path)]
        =
23     toMethod(exchange.getRequestMethod).map { _ -> exchange.
         getRequestURI.getPath }
24 }
```

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.

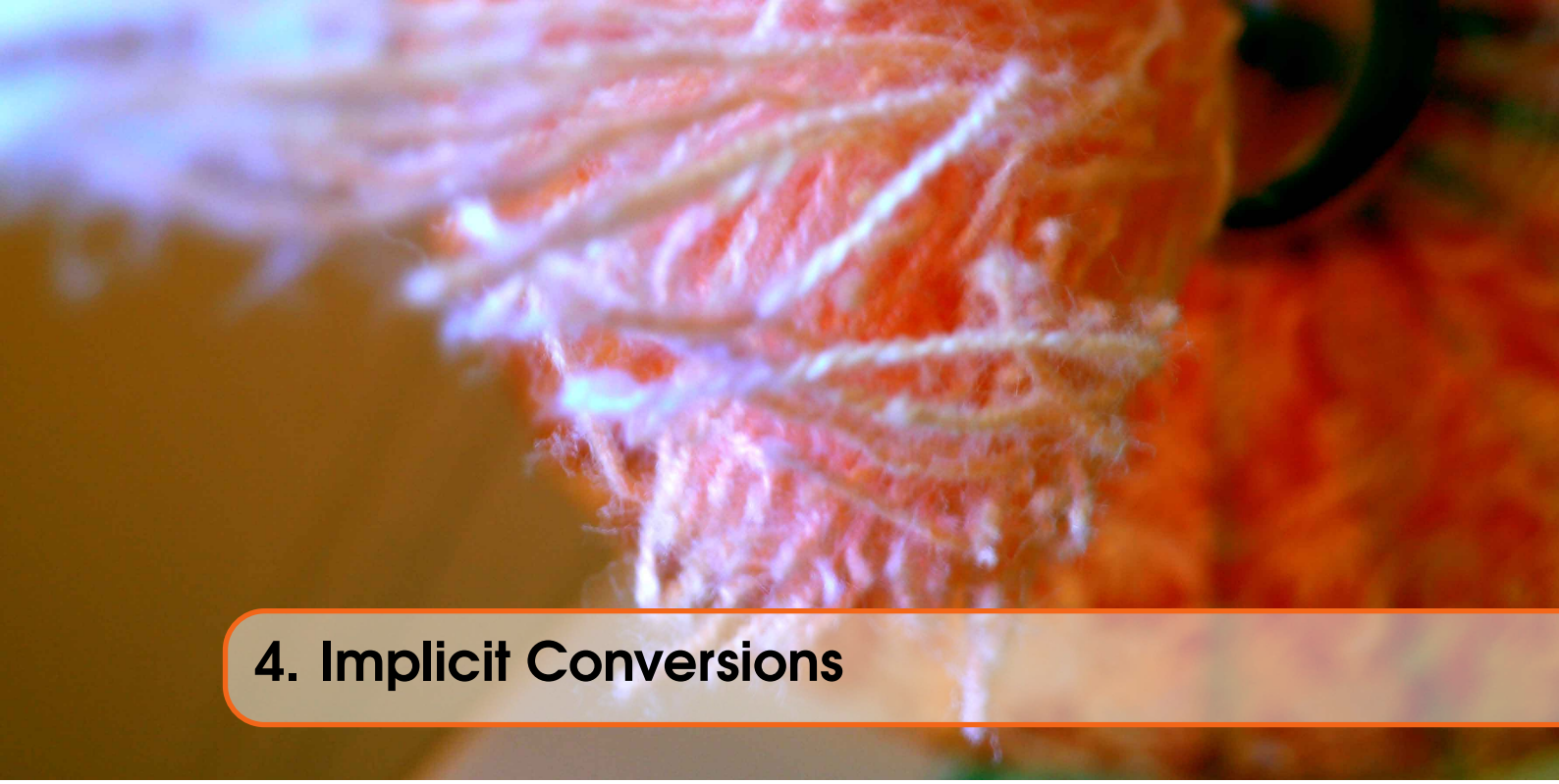
Note also that we face the friction between the two APIs once again. This time, we need to convert the request method from a low-level `String` to a high-level `Method`. As previously, we do that via a conversion method.

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`

```
11 def createServer(port: Int)(handler: Handler): HttpServer = {
12   val server = HttpServer.create(new InetSocketAddress(port),
       0)
13   server.createContext("/", handler)
14   server
15 }
```

4.4 Implementation

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

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

```
24 implicit def toNativeHandler(handler: Handler): HttpHandler =
     { exchange: HttpExchange =>
25   if (handler.isDefinedAt(exchange)) handler(exchange)
26   else {
27     exchange.sendResponseHeaders(404, 0)
28     val os = exchange.getResponseBody()
29     try IOUtils.write("Not found", os)
30     finally os.close()
31   }
32 }
```

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.