Migrating a REST-API implementation from Akka to ZIO

1. Introduction. Statement of the problem and a summary of the characteristics of the starting Akka-Http solution.

In a series of posts started with ContribsGH more than two years ago, we explained in detail diverse solutions to a software development problem consisting of:

The implementation of a REST service based on the GitHub REST API v3 that responds to a GET request at port 8080 and endpoint `/org/{org_name}/contributors` *with a list of the contributors and the total number of their contributions to the repositories of the GitHub organization given by the* `org_name` *part of the URL, in the following JSON format: { "repo": <repository_name>, "contributor": <contributor_login>, "contributions": <no_of_contributions> }.*

The requests to the service must accept two parameters, one to choose between a repository-based or an organization-based accumulation of the contributions, and another to establish a minimum number of contributions for a contributor to appear explicitly in the response. The parameter names are "group-level" and "min-contribs", respectively; the first accepts the values "repo" or "organization" (default), the second any integer value (default 0).

For an API call requesting a grouping at the organization level, the "repo" attribute of the response JSON records must have the value "All {org_name} repos" (a pseudo-repository used to represent the entire organization). In a similar way, the pseudo-contributor "Other contributors" must be used to accumulate the contributions under the value of the "min-contribs" parameter (within each repository or within the entire organization, depending on the value of the "group-level" parameter).

The structure of the problem asked for a solution composed of three modules: a REST server to implement the endpoint of the service, a REST client to get the necessary data from GitHub, and a processing module to make calls to the REST client and transform the data retrieved from GitHub to the format required by the REST server.

A series of Scala solutions to the problem was developed, using what at the time we considered a good mix of Scala libraries: the Lift web framework for implementing the REST server and to quickly develop a web client for the service, and Spray for implementing the REST client.

The solution series started with a synchronous version of the processing module that made calls to the REST client sequentially, each starting after the previous one had finished. Subsequent versions included: i) a more

efficient solution using Scala futures to asynchronously make the REST client calls, ii) a version using Akka typed actors instead of futures, and iii) an enhanced futures-based version with a Redis cache. If you are interested, you can find in the above-mentioned sequence of posts a detailed explanation of the solutions we have just briefly described.

Alas, none of our solutions was purely functional. We used many functional programming idioms to make our code simpler and clearer, but here and there we had side-effecting code intermixed with our functional code, making it impure. As a consequence, we lost the property of referential transparency, which is at the base of: i) safe function composition, the foundation for writing correct and maintainable functional programs, and ii) equational reasoning, based on the substitution of equals by equals, a simple and powerful way to think about programs in order to evaluate them, prove their correctness, design them, transform them, etc. These are too important things to lose due to impurity.

By the time we developed those impure, hybrid solutions, there were already good options for the development of a purely functional solution to our problem, but no compelling reasons to choose one above the others. Instead, since the arrival of its second more stable and mature version, ZIO clearly seemed to us the best option. ZIO has been defined as an ecosystem for building purely-functional, highly concurrent, asynchronous, efficient, back-end applications in Scala. As such, it was the ideal tool for taking the next step in our journey in search of the best solution to the problem posed. To use ZIO there was no need, of course, to throw away what was in good shape in our previous solutions: we could preserve the good while removing the bad (the ugly was taken away at the very beginning by choosing Scala). So, we developed a purely functional solution to our problem in the context of the migration of one of our previous solutions to ZIO.

Among the solution available, the one using Scala futures and a Redis cache was selected as starting point for the migration to ZIO, because of its efficiency and the simplicity of its code. However, instead of using it as it was when the corresponding post was published, we preferred to first update it using Akka-Http (the successor to Spray) to implement both the REST server and the REST client. That allowed us to start the migration from an updated and further simplified program, albeit one without the Lift web client (which was irrelevant anyway for our migration from Akka to ZIO). The library for building Http apps, used to substitute Akka-Http for the migration, was naturally ZIO-Http.

The code of the updated Akka-Http solution using Scala futures and a Redis cache can be downloaded from ContribsGH2-PC.

The code of the solution using ZIO-Http and a Redis cache can be downloaded from ContribsGH2-Z.

This post explains in detail the migration of the Scala application just described, from Akka, a framework based on the actor message-passing paradigm for building highly concurrent applications, to ZIO, a framework used for building highly concurrent applications using the functional-programming paradigm in a pure way.

To better understand that explanation, even a cursory introduction to those frameworks would be helpful. That is the purpose of Section 2. Section 3 offers a preliminary overview of ZIO, presenting only the features necessary to understand Section 4, which is dedicated to a detailed explanation of the changes demanded by the migration from Akka to ZIO. Section 5 gives some numbers about the processing times of the ZIO solution

to our programming problem. Section 6 gives another overview of ZIO, this time from a higher point of view, to the light of the experience acquired while developing the ZIO solution. Section 7 concludes with a summary evaluation of the ZIO solution.

If you are familiar with ZIO, even at a basic level, or prefer to read another introduction to it, you can skip Section 3. If you find the explanation of the migration given in Section 4 too detailed, you can proceed directly to the following sections, where you will find a summary explanation of the Akka to ZIO migration process and a general overview of the conceptual and practical advantages of the resulting program and the ZIO platform.

2. The ZIO platform as an alternative to the Akka platform for the implementation of our REST service.

The Akka platform for the development of concurrent applications is based on the mechanism of message-passing between actors, which are distributed objects with adjustable behavior and protected state. A system is a collection of collaborating actors. The message-passing is done using only actor-references (actor instances are never accessed directly). Actors may have mutable state. This does not imply the inconsistency risks typical of shared state in concurrent systems, because the state is completely encapsulated within the actors. Message processing happens in a “single-threaded illusion” with no concerns for the developer regarding locks, synchronization or any other low-level concurrency primitives. The behavior of an actor is defined as the way it responds to the messages it receives. During the processing of a message, an actor can send messages to other actors, create new actors, and change its internal state and/or its behavior.

Actor systems are a powerful tool for the development of highly concurrent applications, as the success of Akka in that field confirms. Unfortunately, actor systems are not easy to design and implement, nor are they easy to maintain or evolve. The problem with actors is that they are not composable, which makes them hard to design and understand. That is so because the behavior of an (untyped) actor is defined as a partial function of type `Any => Unit`, i.e. a function that receives anything and returns nothing, a kind of blackbox that can only be used for executing a side-effecting action. Functions of this type exclude compositionality, which is based on the coincidence of the output type of one function with the input type of another. With typed actors the input type can be more specific, but the output type remains `Unit`, thus still excluding compositionality.

In contrast, the ZIO platform for the development of concurrent applications is based on the composition of pure functions (functions in the mathematical sense of the term, i.e. without any kind of side effects), which is the simplest and most powerful compositional mechanism, intimately associated to the simplest possible program evaluation mechanism, the substitution model, which stands in sharp contrast with the convoluted models associated with the evaluation of imperative (including actor-based) programs.

In Akka-Http actors are, so to say, behind the scenes, so our recent observation does not apply to the program we selected for ZIO migration, where the parallelism of the calls to the GitHub REST API is achieved using Scala futures. Instead, another program of our series, where that parallelism is achieved explicitly using Akka actors, was ruled out for the ZIO migration precisely because, being equivalent in terms of efficiency, was much more complex in the structure of its Scala code. To give the reader an idea, just the comparison in terms

of lines of code gives for the actors-based solution a count of around 10 times that for the futures-based solution.

But futures also have their problems. First and foremost, when a program creates a future, it starts executing immediately and, from that moment until the moment when the future ends, the program cannot interact with it. To interact with a future, a program must define callbacks for its success/failure, or await for its end blocking the thread that started it. It is possible to compose futures through their callbacks, but that quickly leads to something eloquently known as "callback hell". Besides, without going into much detail, futures are really hard to test. All this discouraging panorama gets a little better when futures are used through for comprehensions; but that does not alter essentially the intrinsic disadvantages of futures just mentioned. In stark contrast, ZIO asynchronous computing works using fibers, which are virtual threads that can be explicitly manipulated in many ways, as well as being implicitly composable through the use of abstractions at a higher level, known as ZIO functional effects.

Functional effects in ZIO are not actual effects, but blueprints or immutable data descriptions of effects, that can be composed in many useful ways before being thrown, at the so-called "end of the world", to an interpreter that actually runs them. As a consequence, effects in ZIO become first-class values, which allows a programmer to write declarative Scala programs with effects, widening the fundamental declarative character of pure functional programming in Scala. The implications of this feature for the readability and ease of maintenance and evolution of effectful programs are enormous, as are the cognitive consequences of separating the definition of an effect from its execution when conceiving, designing, and implementing a solution to a programming problem of an intrinsic asynchronous/concurrent nature. Finally, testing ZIO programs can be quite effective, thanks to the availability, within ZIO Test, of assertions, property-based testing, data generators, mocked resources, etc., all inside a test environment with effects as first class values that leverages the full power of ZIO.

The modular structure of the program we selected for ZIO migration did not change fundamentally when we converted it from Spray/Lift to Akka-Http. Such structure did not change in essence with the migration to ZIO either, with only one important exception related to the Redis cache, originally managed inside the REST server module. That functionality was extracted to two ZIO layers, the first to perform the caching operations themselves, and the second to encapsulate the needed Redis server/client services inside a ZIO layer, in order to present Redis to the other layer as a fully managed resource (details in a further section of this article).

Consequently, the modular structure of the Akka program, originally composed of the REST client, REST server and processing modules, gave rise to a modular structure of the ZIO program composed of the following ZIO layers: RestClient, RestServer, RestServerCache and RedisServerClient. The RestServer module, alleviated from all the duties related to the cache, acquired instead all the logic previously put on the processing module. On closer examination, this structure, almost imposed by the structure of the ZIO layers' requirements, is clearly superior, having as it has a better separation of concerns, and being managed as it is by the services "magically" provided by ZIO for this purpose (i.e. without the need for the programmer to define the hierarchical relationship between layers, which is automatically inferred and managed by ZIO).

3. A brief introduction to ZIO effects and the basic operations on them.

A ZIO effect, not really an effect but instead a blueprint of an effect as mentioned before, has the `ZIO[R, E, A]` parameterized type, where the type parameters represent:

- R, the resource type i.e. the type of the resource (or resources) required to execute the effect,
- E, the error type i.e. the type of the error returned by the effect in case of a failed execution, and
- A, the result type i.e. the type of the value returned by the effect in case of a successful execution.

In terms of its type parameters, a ZIO effect (also called functional effect or just ZIO in what follows) can be conceived as a function of type `R => Either[E, A]`. This characterization can be useful as a basic first approach to understanding ZIO effects, but it is strictly speaking a gross simplification (to start with, ZIO effects are not functions, but models of complex resourceful computations that can be asynchronous or concurrent).

ZIO allows the construction of functional effects from values and from effectful code in many ways, the most common being:

- `succeed`, a smart constructor that allows to use any value of type A to create a successful ZIO effect,
- `fail`, another smart constructor which allows to use any value of type E to create a failed ZIO effect,
- `attempt`, a function that takes as argument any segment of effectful Scala code and returns a `ZIO[Any, Throwable, A]` where A is the type of the value returned by the code segment.

ZIOs can be processed in many ways. In what follows we will present some of the operators that can be used to transform or combine ZIOs, to ease the comprehension of the code discussed in later sections of this article. The informal use of the terms "transform" and "combine" should be interpreted in the context of the processing of immutable values: strictly speaking, for example, a transform operator does not directly transform the return value of a ZIO, but returns a second ZIO which will apply the required transformation to the value returned by the first ZIO, when the corresponding side effects are executed. In what follows we will freely use the terms "transform" (or "convert") and "combine" (or "compose") applied to functional effects, assuming that interpretation.

For space reasons, we will limit ourselves only to the most important operators, starting with those so widely used in functional programming that can be considered absolutely fundamental:

- `map`, allows to apply a function of type `A => B` to a ZIO with return type A, converting it into a ZIO of return type B. We express this capability of functional effects saying that they are functors on their return type.
- `mapError`, in a similar way, allows to apply a function of type `E1 => E2` to a ZIO of error type E1, converting it into a ZIO of error type E2. That makes functional effects also functors on their error type, or bifunctors, to express briefly the coexistence of this capability with the previous one.
- Both map operators return functional effects that model the execution of a functional effect followed by the passing of its result (successful or failed, depending on the case) to a function that transforms it.

- `flatMap`, returns a functional effect that models the execution of a ZIO of return type A followed by the passing of its result to a function that takes an A and returns a ZIO of return type B.
- The `map` and `flatMap` operators together, form the basis of the Scala `for` comprehension, widely used as an intuitively clear representation of the composition of functional effects (in other terms, `flatMap` adds a monadic capability to the functorial capabilities of ZIO effects).

A few examples may help to clarify the concepts presented so far:

```
// appMap
val zio1 = ZIO.succeed("This is a string")
val zio2 = zio1.map(_.toUpperCase)
val zio3 = ZIO.fail("Boom!")
val appMap =
  for {
    val1 <- zio1
    _ <- Console.printLine(s"zio1 succeeds with value ${val1}")
    val2 <- zio2
    _ <- Console.printLine(s"zio2 succeeds with value ${val2}")
    err <- zio3.mapError(_.toUpperCase())
    _ <- Console.printLine(s"zio3 fails with error ${err}!")
  } yield ()

// appEcho
val appEcho = Console.readLine("Echo ... please type something\n").
              flatMap(line => Console.printLine(line))

// appHello
val appHello =
  for {
    _ <- Console.printLine("Hello! What is your name?")
    name <- Console.readLine
    _ <- Console.printLine(s"Hello, ${name}, welcome to ZIO!")
  } yield ()
```

Here we define some ZIOs, process them in different ways, and define three ZIO example applications that can be run as normal Scala apps by defining them inside an object that extends `ZIOAppDefault` and overrides the `run` function.

The first ZIO example application, `appMap`, shows how `map` can transform the result value of a functional effect. The success values of zio1 and zio2, retrieved inside a for comprehension, are displayed using `Console`, a service included with ZIO that provides functions to perform simple I/O operations on the input/output channels of a standard console application (more about ZIO included services in a moment). Within the same for comprehension, the error value of zio3, a failed ZIO, is also transformed, this time using `mapError`. But the subsequent `printLine` is not executed, because the failure of zio3 ends the execution of the entire application, as it always happens when a ZIO fails while a sequence of ZIOs is being executed inside a for comprehension. Instead, an error message is displayed: `... level=ERROR thread=#zio-fiber-0 ... cause="Exception in thread "zio-fiber-4" java.lang.String: BOOM!`, that informs us about the fiber executing the failed ZIO and the cause of the failure (in this case the failure value that defined zio3, converted to upper case by `mapError`).

The second example ZIO app, `appEcho`, shows how `flatMap` can be used to communicate two ZIOs executed in sequence. The third, `appHello`, implements basically the same logic, this time expressed as a for comprehension which naturally reads almost like an equivalent imperative program.

Continuing with our presentation of ZIO operators, we now discuss some important combinators for functional effects:

- `orElse`, allows to execute an alternative ZIO when, and if, the main ZIO fails. The value returned by the composed ZIO is that of the main ZIO, if it succeeds, and that of the alternative one, if it fails.
- `zip`, compose two ZIOs by executing both and returning the 2-tuple of their returned values. If one of the two ZIOs fails, the entire composed ZIO fails.
- `zipRight`, the same as zip, but returns only the value returned by the second ZIO.
- `zipLeft`, again the same as zip, but returns instead only the value returned by the first ZIO.
- `zip` is intrinsically less powerful than `flatMap`, because, even if it also allows the sequential execution of two ZIOs, the second ZIO is entirely independent of the first one (i.e. its execution does not depend on the value returned by the first). That should allow the parallel execution of the zipped ZIOs, and in fact a parallel version of `zip` exists, called, unsurprisingly, `zipPar`.

Here we have a couple of examples using the combinators just presented:

```
import zio._
import java.io.IOException

val helloWorld = ZIO.succeed(print("Hello ")).zipRight(ZIO.attempt(print("World!")))
val helloWorld2 = Console.print("Hello ").zipRight(Console.print("World!"))

val primaryOrBackupData: ZIO[Any, IOException, String] =
  ZIO.readFile("primary.data").orElse(ZIO.readFile("backup.data"))
```

The first two examples print "Hello World!" to the console as the result of executing two ZIOs in sequence, discarding the value of the first one (of type `Unit`, a useless value anyway) and returning the value of the second (again, the useless value of the `Unit` type). The difference between them lies in the fact that the first uses `succeed` and `attempt` to transform Scala side-effecting code into functional effects, while the second uses directly functional effects provided by `zio.Console`. The third example shows a typical use of `orElse` to provide a fallback to an IO operation that could fail.

Continuing with our overview, let's mention four different default services provided by ZIO, that allow us to include in any application some common side-effecting functionality: `Console`, `Clock`, `System` and `Random`. The first, `Console`, was already mentioned and simple examples given of its use. Regarding the others:

- `Clock`, provides functionality related to time and scheduling. For example, you can use `Clock.instant` to get the current time, or apply `Clock.delay` to a ZIO to get a delayed functional effect.
- `System`, provides functionality to get system and environment variables.
- `Random`, provides a random number generator.

As an example of the use of these services, the following Scala segment, extracted in a simplified form from the code of our program, allows us to report in a LogBack logger information about the time consumed by a call to our REST-API:

```
for {
  initialInstant <- Clock.instant
  _ <- ZIO.succeed(logger.info
       (s"Starting ContribsGH2-Z REST API call at ${sdf.format(Date.from(initialInstant))}"))
  _ <- ZIO.succeed(logger.info("Working ..."))
  finalInstant <- Clock.instant
  _ <- ZIO.succeed(logger.info
       (s"Finished ContribsGH2-Z REST API call at ${sdf.format(Date.from(finalInstant))}"))
  _ <- ZIO.succeed(logger.info
       (f"Time elapsed: ${Duration.between(initialInstant, finalInstant).toMillis / 1000.0}%3.2f secon
} yield ()
```

The configuration and creation of the LogBack logger was kept unaltered in the migration from Akka-Http to ZIO-Http. To use it, we just embedded the corresponding logger calls inside `ZIO.succeed`.

To finish our overview, let's discuss briefly the main concepts related to the definition and management of the modular structure of a ZIO application.

`ZLayer`s are software layers used to define the modular structure of a ZIO application, stipulating for any layer the resources that it needs and the services it provides, as well as, if needed, the logic to automatically manage the life cycle of the layer using functions to acquire and release its resources. When the "big ZIO" representing a complete application is defined, a `provide` function can be used to hand it over the needed layers as a flat structure, leaving to ZIO the task of infer and automatically manage the hierarchy of those layers.

A standard simple protocol (used in our program) can be used to create a `ZLayer`. First, define a `trait` containing the signatures of the services provided. Then, define a "live" `case class` containing the code that implements those services. Finally, in the accompanying object of that case class define a smart creator for it, using either `ZLayer.fromFunction` applied to the default constructor of the case class, or `ZLayer.scoped` applied to a ZIO that specifies both, the function to be used to acquire an instance of the layer, and the function appropriate to release it. Examples of both kinds of smart creator will be given in sections 4.4 and 4.5.

4. The changes made to the modules of our Akka-Http implementation to migrate them to ZIO.

4.1 Changes to the REST server module.

The main goal of this module is to define the endpoint of our REST service in terms of URL, HTTP method and HTTP parameters, associating a request thus defined with the appropriate response. Besides that, this

module just launches the HTTP server in charge of handling the request/response cycles (trivial code not shown).

The corresponding Scala code in the case of Akka-Http, using a DSL provided to that end, is:

```
val route: Route = {
  get {
    path("org" / Segment / "contributors") { organization: String =>
      parameters("group-level".optional, "min-contribs".optional) { (glo, mco) =>
        val gl = glo.getOrElse("organization")
        val mc = mco.getOrElse("0").toIntOption.getOrElse(0)
        complete(contributorsByOrganization(organization, gl, mc))
      }
    }
  }
}
```

As we can see, the data returned by the function `contributorsByOrganization` (which makes part of the processing module) is used to build the response associated to a given request.

Providing a DSL with the same purpose, although syntactically diverse, ZIO-Http allows us to define our endpoint using the following code:

```
val contribsGH2ZApp: Http[zio.http.Client, Nothing, Request, Response] = Http.collectZIO[Request] {
  case req@(Method.GET -> !! / "org" / organization / "contributors") =>
    val glS = req.url.queryParams.get("group-level").getOrElse(Chunk[String]()).asString
    val gl = if (glS.trim == "") "organization" else glS
    val mcS = req.url.queryParams.get("min-contribs").getOrElse(Chunk[String]()).asString
    val mc = mcS.toIntOption.getOrElse(0)
    contributorsByOrganization(organization, gl, mc).orDie.map(l => Response.json(l.toJson))
  }
```

Seen from the outside, we could say that the ZIO code is only syntactically different from the Akka code. We even also have a `contributorsByOrganization` function with the duty of returning the data needed to generate the response. But behind the scenes, we have Akka actors in action in one case and ZIO effects in the other. That explains the seemingly just nominal differences in terms of the handled types, which are instead very important as we explain now step by step.

To start with, `contributorsByOrganization` in the Akka case returns directly a `List[Contributor]`, while in the ZIO case it returns, quite predictably, a functional effect, whose type, `ZIO[zio.http.Client, Throwable, List[Contributor]]`, has a `List[Contributor]` as the result of a ZIO that requires a `zio.http.Client` to be run and can fail with a `Throwable`. This ZIO is composed in the rest of the program with other **functional** effects in the way appropriate to implement our REST service. Those effects are finally executed by the ZIO runtime, at the very edge of our program (the already mentioned "end of the world") which is the only place where **real** effects occur.

Next, the `orDie` function applied to the return value of `contributorsByOrganization` in the ZIO case, is used to convert it to an "infallible" ZIO (a ZIO with error type `Nothing`). This results, **at runtime**, in the killing of the fiber executing the ZIO if it throws an error.

Finally, the result of `contributorsByOrganization` as an infallible ZIO is mapped into a `zio.htttp.Response`. This last step is required to finish the provision of a transformation expected by `Http.collectZIO[Request]`, the function that eventually implements the endpoint of our REST service.

4.2 Changes to the REST client module.

In this module, in charge of making requests to the GitHub REST service, the most important change regards managing the pagination of the GitHub responses using an auxiliary recursive function with an accumulating parameter. The purpose of this function was explained in detail in the third post of our series ContribsGH-P

In a few words, before returning the response to a GitHub request, composed eventually of multiple pages, the body contents of those pages are accumulated in one of the parameters of an auxiliary function that is recursively called until a page with an empty body is returned by GitHub (signaling the end of the paginated response).

The corresponding code in the case Akka-Http is:

```
def processResponseBody[T](url: String) (processPage: BodyString => List[T]): List[T] = {

  def processResponsePage(processedPages: List[T], pageNumber: Int): List[T] = {
    val eitherPageBody = getResponseBody(s"$url?page=$pageNumber&per_page=100")
    eitherPageBody match {
      case Right(pageBody) if pageBody.length > 2 =>
        val processedPage = processPage(pageBody)
        processResponsePage(processedPages ++ processedPage, pageNumber + 1)
      case Right(_) =>
        processedPages
      case Left(error) =>
        logger.info(s"processResponseBody error - $error")
        processedPages
    }
  }

  processResponsePage(List.empty[T], 1)
}
```

Which is replaced, in the case ZIO-Http, with the following:

```
def processResponseBody[T](url: String) (processPage: BodyString => List[T]):
  ZIO[zio.http.Client, Nothing, List[T]] = {

  def processResponsePage(processedPages: List[T], pageNumber: Int):
    ZIO[zio.http.Client, Nothing, List[T]] = {
    getResponseBody(s"$url?page=$pageNumber&per_page=100").orDie.flatMap {
```

```
        case Right(pageBody) if pageBody.length > 2 =>
          val processedPage = processPage(pageBody)
          processResponsePage(processedPages ++ processedPage, pageNumber + 1)
        case Right(_) =>
          ZIO.succeed(processedPages)
        case Left(error) =>
          logger.info(s"processResponseBody error - $error")
          ZIO.succeed(processedPages)
      }
    }

    processResponsePage(List.empty[T], 1)
  }
```

These functions are generic on the type `T` of the elements of the list returned, thanks to its second parameter `processPage`, a function that converts the body of a response (of type `BodyString`, a type synonym of `String`) to a `List[T]`. The name of the auxiliary recursive function with an accumulating parameter is `processResponsePage` in both cases.

As we can see, the difference in returned type between the versions of `processResponsePage` (`List[T]` in the Akka case and `ZIO[zio.http.Client, Nothing, List[T]]` in the ZIO case), is similar to that discussed in the previous section. The cause, again, is the fact that the ZIO version work with functions that do not return directly the result of a side effect, but instead a functional effect representing that result, which can be composed with other functional effects as desired. In this case, the desired composition is achieved using the `flatMap` combinator, which feeds the result of the first ZIO to a function that returns the second ZIO (a partial function that, either ends the recursion using `ZIO.succeed` with the accumulated result, or recursively calls `processResponsePage` to continue the accumulation process). Before passing the functional effect returned by `getResponseBody` to `flatMap`, the `orDie` function is applied to it, making it an infallible ZIO as required by the signature of `processResponsePage`.

For its part, `getResponseBody` returns a functional effect of type `ZIO[zio.http.Client, Throwable, Either[BodyString, ErrorString]]`, where the `Throwable` error type is inherited from the error type of the ZIO returned by `zio.http.Client.request`. This function is used to make a Http request taking parameters that represent the necessary URL, method, headers and body. In the result type of `getResponseBody`, the `BodyString` and `ErrorString` types are both synonyms of `String` and their `Either` combination represents the potentially successful or failed result of the Http request made.

4.3 Changes to the processing module.

Until now, the functorial and monadic capabilities of functional effects have been sufficient to transform and combine ZIOs in any way we might need. As powerful as they are, however, we now need to add another one: the traversable capability, which unifies the concept of mapping over a container. It will be implemented through combinators that can be used to transform a collection of ZIOs into the ZIO of a collection.

Those combinators will be used in the ZIO-Http solution to our problem, exactly as we use `Future.sequence` in the Akka-Http solution to transform a `List[Future[T]]` into a `Future[List[T]]`. The use of the

traversable capability of Scala futures implemented by `Future.sequence` is explained in detail in the third post of our series ContribsGH-P

The fact that this traversable capability in the ZIO case exists in a sequential and a parallel version (`collectAll` and `collectAllPar` , respectively), will allow us to mimic not one but two of our previous solutions by simply using one version instead of the other, as explained below.

The functionality of the Akka version processing module, which in the ZIO version resides inside the RestServer module as said before, becomes a functional effect with type `ZIO[zio.http.Client, Throwable, List[Contributor]]` , implemented as follows (with its logging code commented out, to highlight its most relevant code):

```
def contributorsByOrganization(organization: Organization, groupLevel: String, minContribs: Int):
  ZIO[zio.http.Client, Throwable, List[Contributor]] = for {
    // report start time using LogBack as explained in Section 3
    repos <- restClient.reposByOrganization(organization)
    contributorsDetailed <- contributorsDetailedZIOWithCache(organization, repos)
    contributors = groupContributors(organization, groupLevel, minContribs, contributorsDetailed)
    // report end time using LogBack as explained in Section 3
  } yield contributors

def contributorsDetailedZIOWithCache(organization: Organization, repos: List[Repository]):
  ZIO[zio.http.Client, Throwable, List[Contributor]] = {
  val (reposUpdatedInCache, reposNotUpdatedInCache) = repos.
       partition(restServerCache.repoUpdatedInCache(organization, _))
  val contributorsDetailed_L_1: List[List[Contributor]] =
      reposUpdatedInCache.map { repo =>
        restServerCache.retrieveContributorsFromCache(organization, repo)
      }
  val contributorsDetailed_L_Z_2 =
    reposNotUpdatedInCache.map { repo =>
      restClient.contributorsByRepo(organization, repo)
    }
  // retrieve contributors by repo in parallel
  val contributorsDetailed_Z_L_2 = ZIO.collectAllPar(contributorsDetailed_L_Z_2).withParallelism(4)
  // retrieve contributors by repo sequentially
  //val contributorsDetailed_Z_L_2 = ZIO.collectAll(contributorsDetailed_L_Z_2)
  for {
    contributorsDetailed_L_2 <- contributorsDetailed_Z_L_2
    _ <- ZIO.succeed(restServerCache.
         updateCache(organization, reposNotUpdatedInCache, contributorsDetailed_L_2))
  } yield (contributorsDetailed_L_1 ++ contributorsDetailed_L_2).flatten
}
```

Here, the first function shown, `contributorsByOrganization` , consists of a for comprehension that retrieves the values of the functional effects `reposByOrganization` and `contributorsDetailedZIOWithCache` , the first used for getting from GitHub the repositories of a given organization as a `List[Repository]` , and the second to get the contributors of a given organization and repositories as a `List[Contributor]` , in both cases, of course, as ZIO result types. The core functionality, implemented by `contributorsDetailedZIOWithCache` , takes the place of `contributorsDetailedFutureWithCache` used in the Akka version.

Both, the Akka and the ZIO version of this functionality, make use of `contributorsByRepo` to get from GitHub the contributors of each single repository of the given organization. In both cases, the calls to this function are made in parallel, using futures in the Akka version and functional effects in the ZIO version, and applying the corresponding traversable capabilities by means of `Future.sequence` and `ZIO.collectAllPar`, respectively. Again, the difference between the Akka version and the ZIO version of `contributorsByRepo` resides in the fact that the ZIO version returns a `List[Contributor]` indirectly, as the result of a functional effect.

The `reposByOrganization` and `contributorsByRepo` are functional effects provided by `restClient`, both based on the generic capacity of `processResponseBody[T]` explained at detail in Section 4.2.

A peculiarity of the ZIO version is worth emphasizing: the `collectAllPar` function used to traverse the functional effects executing them in parallel, exists also in a non-parallel version, `collectAll`, which also traverse the collection of ZIOs (i.e. converts it into the ZIO of a collection), but executing the involved functional effects sequentially. As a consequence, we can have sequential and parallel versions of our ZIO program just using alternatively `collectAll` or `collectAllPar`. The implications of this trivial change in our code, from the point of view of the efficiency of the ZIO solution to our programming problem, will be presented in Section 5.

4.4 A new module to encapsulate the Redis services needed by the cache.

One of ZIO's strengths is its comprehensive solution of the dependency injection problem, fully integrated in the ZIO platform, starting with the very definition of what a functional effect is. Indeed, the first type parameter of the ZIO type represents the resource (or resources) needed for a ZIO to be executed. Besides, the modular structure of a ZIO application is defined in terms of `ZLayer` s, which specify the services provided and resources needed, and can also include all the logic needed to safely manage the involved resources.

As part of our exploration of ZIO in the context of a migration from Akka, we decided to maintain two Java libraries, EmbeddedRedis and Jedis, deliberately included in the original version to show how easily Scala allows to take advantage of the huge amount of functionality available in the Java ecosystem. Surprisingly, not only was migrating our code using those Java libraries to ZIO almost trivial, but we even improved its use in the ZIO version, thanks to the powerful and extremely easy-to-use tools provided to manage the life cycle of the resources needed by a functional effect, no matter if those come from ZIO, Scala without ZIO, or even plain old Java as is the case of our Redis services.

The acquire-release services provided by ZIO to safely manage a functional effect are easily put at work by using `ZIO.acquireRelease`, as part of the second way to define a `ZLayer` described in Section 3. This function takes as parameters the ZIOs to be executed for acquiring and releasing a resource, returning a "scoped resource" that becomes a `ZLayer` after being fed to the function `ZLayer.scoped`. The resulting layer will afterwards be delivered to the main ZIO of our program to provide the necessary Redis services, together with the services provided by the other layers of our program.

All this translates to the following Scala code:

```
trait RedisServerClient {
  val redisServer: RedisServer
  val redisClient: Jedis
}

case class RedisServerClientLive() extends RedisServerClient {
  val redisServer = new RedisServer(6379)
  redisServer.start()
  val redisClient = new Jedis()
}

object RedisServerClientLive {
  def releaseRSCAux(rsc: RedisServerClientLive): Unit = {
    rsc.redisClient.flushAll()
    rsc.redisClient.close()
    rsc.redisServer.stop()
  }
  def acquireRSC: ZIO[Any, Nothing, RedisServerClientLive] = ZIO.succeed(new RedisServerClientLive())
  def releaseRSC(rsc: RedisServerClientLive): ZIO[Any, Nothing, Unit] = ZIO.succeed(releaseRSCAux(rsc)

  val RedisServerClientLive = ZIO.acquireRelease(acquireRSC)(releaseRSC)
  val layer = ZLayer.scoped(RedisServerClientLive)
}
```

Here we can clearly see the use of `ZIO.acquireRelease` and `ZLayer.scoped`, preceded by the definition of the needed trait and the live case class with accompanying object implementing it (q.v. Section 3). The acquisition effect (`acquireRSC`) just creates an instance of RedisServer, starts it and creates an instance of Jedis. The releasing effect (`releaseRSC`) flushes the Redis server, closes the Jedis client, and stops the server. Both effects are guaranteed to be run without interruptions. For the releasing effect, this translates into a guarantee that our program will not leak resources, even if it stops working as a consequence of an unmanaged error or an interruption.

4.5 A new module to encapsulate the services provided by the cache.

As mentioned in the last paragraph of Section 2, the original REST server module contained all the code needed for administering the Redis cache, including the management of the Redis server and client themselves. Having extracted from that module the EmbeddedRedis server and Jedis client to a separate ZIO layer, `RedisServerClient`, we can do another refactoring, this time regarding the cache services themselves, which can be extracted to another ZIO layer, `RestServerCache`, that takes as a resource the `RedisServerClient` layer.

The following Scala code shows how this can be achieved, showing in detail only one of the services provided by the new layer:

```
trait RestServerCache {
  def repoUpdatedInCache(org:Organization, repo: Repository): Boolean
  def retrieveContributorsFromCache(org:Organization, repo: Repository): List[Contributor]
  def updateCache(organization: Organization, reposNotUpdatedInCache: List[Repository],
                  contributors_L: List[List[Contributor]]): Unit
}

case class RestServerCacheLive(redisServerClient: RedisServerClient) extends RestServerCache {
  def retrieveContributorsFromCache(org:Organization, repo: Repository): List[Contributor] = {
    val repoK = buildRepoK(org, repo)
    val res = redisServerClient.redisClient.
              lrange(repoK, 1, redisServerClient.redisClient.llen(repoK).toInt - 1).asScala.toList
    logger.info(s"repo '$repoK' retrieved from cache, # of contributors=${res.length}")
    res.map(s => stringToContrib(repo, s))
  }
}

object RestServerCacheLive {
  val layer = ZLayer.fromFunction(RestServerCacheLive(_))
}
```

As we can see, the live case class that implements the services of the `RestServerCache` layer, has a constructor that takes as an argument an instance of the `RedisServerClient` layer, which is used to access its services. That instance is automatically provided at runtime by ZIO, as specified in the companion object of the live case class by means of the expression `fromFunction(RestServerCacheLive(_))`, used to build the layer calling precisely that constructor, as part of the first way to define a `ZLayer` described in Section 3.

The expression `redisServerClient.redisClient.lrange(repoK, 1, redisServerClient.redisClient.llen(repoK).toInt - 1)` uses `redisClient` (one of the services provided by the `RedisServerClient` layer, as explained in Section 4.4) to access a Redis list containing the contributions of a repository previously stored in the cache. That list, in the form of a Scala `List[String]`, is mapped to a `List[Contributor]` using an auxiliary function. The Redis list itself is stored under the key `repok`, built using another auxiliary function.

4.6 The modular structure of the complete application.

The provision of all the layers needed by our complete ZIO application is done when overriding the `run` function of `ZIOAppDefault`, as follows:

```
object ContribsGH2Z extends ZIOAppDefault {
  override val run = {
    ZIO.serviceWithZIO[RestServer](_.runServer).
      provide(
        RestClientLive.layer,
        RestServerLive.layer,
        RestServerCacheLive.layer,
        RestServerClientLive.layer
        )
```

```
    }
  }
```

Here we can see how the layers of our ZIO program, RestClient, RestServer, RestServerCache and RedisServerClient, are provided to its main function `RestserverLive.runServer`. We hope that the detailed presentation made of the core code segments of those layers, provide strong support for our earlier statement about the advantages of the ZIO modular structure over the original one.

5. Characteristics of the ZIO implementation in terms of efficiency

To make a simple comparison of the sequential and parallel versions of our ZIO REST server, the former implemented using `ZIO.collectAll` and the later using instead `ZIO.collectAllPar` as explained before, we executed both for the organization "revelation".

Please note, if you try these examples on Windows and receive a 500 HTTP response (internal server error), probably the Defender Firewall is blocking the embedded Redis server. Simply starting a new sbt session should solve the problem.

The following lines show part of a trace of the executions, displayed by the programs to the logger console.

Sequential

[INFO] ContribsGH2-Z.log - Starting ContribsGH2-Z REST API call at 30-05-2023 06:52:13 - organization='revelation' [INFO] ContribsGH2-Z.log - # of repos=24

[INFO] ContribsGH2-Z.log - repo='globalize2', # of contributors=6 [INFO] ContribsGH2-Z.log - repo='revelation.github.com', # of contributors=3 ... [INFO] ContribsGH2-Z.log - repo='ember-rails', # of contributors=100 [INFO] ContribsGH2-Z.log - repo='ey-cloud-recipes', # of contributors=66

[INFO] ContribsGH2-Z.log - repo 'revelation-globalize2' stored in cache [INFO] ContribsGH2-Z.log - repo 'revelation-revelation.github.com' stored in cache ... [INFO] ContribsGH2-Z.log - repo 'revelation-ember-rails' stored in cache [INFO] ContribsGH2-Z.log - repo 'revelation-ey-cloud-recipes' stored in cache

[INFO] ContribsGH2-Z.log - Finished ContribsGH2-Z REST API call at 30-05-2023 06:52:32 - organization='revelation' [INFO] ContribsGH2-Z.log - Time elapsed from start to finish: 19,15 seconds

Parallel

[INFO] ContribsGH2-Z.log - Starting ContribsGH2-Z REST API call at 29-05-2023 05:35:06 - organization='revelation' [INFO] ContribsGH2-Z.log - # of repos=24

[INFO] ContribsGH2-Z.log - repo='globalize2', # of contributors=6 [INFO] ContribsGH2-Z.log - repo='almaz', # of contributors=4 ... [INFO] ContribsGH2-Z.log - repo='rails', # of contributors=47 [INFO] ContribsGH2-Z.log - repo='ey-cloud-recipes', # of contributors=66

[INFO] ContribsGH2-Z.log - repo 'revelation-globalize2' stored in cache [INFO] ContribsGH2-Z.log - repo 'revelation-revelation.github.com' stored in cache ... [INFO] ContribsGH2-Z.log - repo 'revelation-ember-rails' stored in cache [INFO] ContribsGH2-Z.log - repo 'revelation-ey-cloud-recipes' stored in cache

[INFO] ContribsGH2-Z.log - Finished ContribsGH2-Z REST API call at 29-05-2023 05:35:11 - organization='revelation' [INFO] ContribsGH2-Z.log - Time elapsed from start to finish: 4,48 seconds

As we can see, the parallel version took about a quarter of the time of the sequential one. The times shown are for a laptop with 4 Intel I7 cores.

The trace of a second call to our service using the parallel version with the same parameters and organization, shows:

[INFO] ContribsGH2-Z.log - Starting ContribsGH2-Z REST API call at 29-05-2023 05:36:11 - organization='revelation' [INFO] ContribsGH2-Z.log - # of repos=24

[INFO] ContribsGH2-Z.log - repo 'revelation-globalize2' retrieved from cache, # of contributors=6 [INFO] ContribsGH2-Z.log - repo 'revelation-revelation.github.com' retrieved from cache, # of contributors=3 ... [INFO] ContribsGH2-Z.log - repo 'revelation-ember-rails' retrieved from cache, # of contributors=100 [INFO] ContribsGH2-Z.log - repo 'revelation-ey-cloud-recipes' retrieved from cache, # of contributors=66

[INFO] ContribsGH2-Z.log - Finished ContribsGH2-Z REST API call at 29-05-2023 05:36:12 - organization='revelation' [INFO] ContribsGH2-Z.log - Time elapsed from start to finish: 1,23 seconds

Here we can see that the elapsed time is again reduced to approximately a quarter of the previous one. This shows the improvement in efficiency achieved from using a cache implemented as an in-memory Redis data-structure server.

6. Main benefits of ZIO.

From a conceptual point of view, programming highly concurrent applications with ZIO using pure functions and immutable variables makes life easier for Scala developers, as it greatly simplifies the mental model of the programs they write. This is the biggest ZIO conceptual advantage: the ability to handle effects as values to get a higher level of declarative programming in Scala, widening the way opened by the use of functions as values. ZIO functional effects are immutable data values, they can be transformed and combined to generate new functional effects in very powerful ways, and the logic that we apply to reason about them is the same

that is used to reason about any pure values in functional programs: the extremely simple and effective logic based on the substitution of equals by equals.

All this power is put in the hands of a Scala programmer without resorting to the advanced features of the Scala type system (like higher-kinded types or type classes) or sophisticated FP design patterns (like monad transformers or tagless final), which in the alternatives to ZIO represent cognitive barriers that Scala beginners cannot easily overcome. All obstacles to the development of highly concurrent applications using only the fundamentals of pure functional programming, are deliberately avoided by ZIO.

ZIO also has clear advantages from a more practical/technological point of view, which we will try to summarize below.

- **Resource-safety.** The automatic management of the lifetime of resources, even in the presence of unexpected errors, considerably simplifies the development of application that don't leak them.
- **Error handling.** ZIO provides first-class support for typed errors. The static handling of standard and custom-defined errors allows to perform error analysis at compile-time, a practice that can have a very positive impact on development productivity. ZIO provides a variety of other resources related to error-handling that we don't have space to discuss here. We suggest the reader to explore them in the ZIO documentation, to have a better image of an area in which ZIO particularly excels as compared to other effect systems.
- **Concurrency based on fibers.** Fibers are lightweight virtual threads that can be executed, scheduled, examined, interrupted and cancelled explicitly, giving to a programmer looking for a low level of concurrency control a tool of practically unlimited power. Concurrency using fibers performs much better than directly using JVM threads, since they are implemented in a way that allows a single JVM thread to execute many fibers. ZIO also provides tools for concurrency and parallelism at a higher level, which make the explicit use of fibers unnecessary in most cases.
- **Higher level tools for concurrency.** ZIO provides synchronization primitives and concurrent data structures that greatly simplify the development of concurrent applications, such as:
  - `Ref` to safely share mutable state between fibers,
  - `Semaphore` to control the degree of concurrency of fibers,
  - `Queue` to distribute work among multiple fibers, and many others.
  - Besides, ZIO provides `STM` (Software Transactional Memory) that allows performing a group of memory operations as a single atomic operation. STM guarantees three of the well-known ACID properties of database transactions: atomicity(A), consistency(C) and isolation(I) (not durability(D), that doesn't make sense for memory operations).
- **Streaming capabilities.** ZIO Stream is an effectful and resourceful streaming library that provides the traditional source, sink and transform kinds of elements as pure functional, fully composable, components. It supports backpressure through a pull-based mechanism and is fully integrated with ZIO.
- **Large, continuously evolving, ecosystem.** Composed of official ZIO libraries and community-maintained libraries.

7. A summary evaluation of the ZIO solution to our problem.

We start this concluding section with a summary of the changes needed for our migration from Akka-Http to ZIO-Http.

In many cases they were simple changes, determined by the need to have intermediate values of some data flows available "indirectly" as the result of functional effects. Besides, several changes were guided by type-checking error messages issued by the Scala compiler, highlighting another important characteristic of ZIO: its type structure has been carefully designed to get a type-inference process which significantly improves usability.

Despite implying a major paradigm shift, the migration to ZIO did not cause a drastic refactoring of the code of the Akka solution to our problem, as discussed at length in Section 4. Without fear of exaggeration, we could affirm that a good understanding of the basic principles of ZIO was the only important prerequisite to perform a smooth and flawless migration.

A summary of the changes made, module by module, follows:

- **Changes to the REST server module.** Apart from the refactoring mentioned below, just obvious changes caused by the differences between the DSL's used to implement a REST endpoint, and trivial changes of the kind "return result indirectly by means of a functional effect".
- **Changes to the REST client module.** Again, mostly changes implied by the differences between DSL's (this time those used to make a request to another REST service) and changes imposed by the use of functional effects, with only one important addition: the changes made to a recursive function with an accumulating parameter, a simple adaptation of a basic recursive algorithm to make it work with the return values of functional effects.
- **Changes to the processing module.** The main change in this case was the use of `ZIO.collectAllPar` to traverse a collection of ZIOs converting them to a ZIO of a collection, in a way conceptually equivalent to the use of `Future.sequence` in the Akka version.
- **Refactorings for a better modular structure.** Taking advantage of the tools provided by ZIO to automatically administer the resource flow between layers and safely manage their lifecycles, we extracted from the REST server module the Redis services provided by a couple of Java libraries, as well as the caching services based on them. Those refactorings resulted in a better modular structure as explained in detail in Section 4.

Based on all of our discussion above, we could conclude that **our ZIO solution to the software development problem stated in Section 1 has the conceptual advantages derived from the use of pure functional effects, as well as the practical benefits derived from the advanced concurrency/parallelism technology used by ZIO**.

However, there should be more to do with ZIO and ZIO-Http to better solve this problem. After all, our ZIO solution is the result of the migration from a previous non-ZIO solution. Could it be further transformed into a more ZIO idiomatic solution? Could that effort be a good test of the maintainability of a ZIO-Http program? Could we take advantage of the recently announced refactoring of ZIO-Http to get an even better ZIO solution? We hope to give a categorical positive answer to all these questions in a future post, but perhaps it would be better first to write a little more about the conceptual and practical advantages of a clear separation between the definition of an effectful program and the execution of its effects.

Finally, some links to great resources available for learning ZIO:

- Courses, Rock The JVM, Developer Inside You.
- Articles, ebooks and presentations, Scalac.
- ZIO's official site, with many useful guides and good reference material.
- The "bible" Zionomicon, co-written by John A. De Goes, the author of ZIO.