Implementing a REST service with the high-level EndPoints API of ZIO HTTP

1. Introduction. Specification of our REST-API and some characteristics of its ZIO HTTP implementation.

In this article, we discuss 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> }.*

Accepts two request 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 that requests 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 implementation of this REST service uses ZIO HTTP, a Scala library used for building HTTP servers and clients in a purely functional way, and REDIS, an in-memory data structure store used to implement a cache to speed up the response for an organization previously requested.

ZIO HTTP is based on ZIO, which is, to begin with, an I/O monad for Scala. But the name ZIO is also used to refer to an entire software ecosystem used for **building purely-functional, highly concurrent, asynchronous, efficient, back-end applications in Scala**.

The main purpose of ZIO as an I/O monad is to allow programming with side effects (like I/O operations, or calls to external, fallible, and frequently remote, services) using only pure functions, thus combining the advantages of pure functional programming with the inevitable need for side effects through which our programs can interact with the outside world. For a discussion of the importance of programming with pure functions, and an introduction to the basic concepts of side effect and I/O monad by means of simple examples, the interested readers are invited to read this article).

ZIO HTTP, on its part, is a Scala library powered by ZIO, as just said, and Netty, that aims at **being the defacto solution for writing highly scalable and performant web applications using idiomatic Scala**.

ZIO HTTP inherits from ZIO the power and simplicity of the functional programming paradigm and from Netty the high throughput of an asynchronous, event-driven network application framework. It is a library in continuous evolution that, among its impressive recent improvements, has been provided with an Endpoints API for the high-level definition of endpoints that further simplifies the development of REST servers.

This article discusses the ZIO HTTP implementation of the GitHub-based REST API just defined, explaining the basics of ZIO HTTP and the use of its Endpoints API for the high-level definition of the corresponding endpoint.

To better understand that explanation, some familiarity with the ZIO framework is advisable. Section 2 gives some hints that can be useful to get the desirable basic knowledge of ZIO. It also briefly introduces the fundamental concepts of ZIO HTTP and the modular structure of the implementation of our REST service.

Section 3 is dedicated to a detailed explanation of the modules of the ZIO HTTP implementation of our service. Section 4 gives some numbers about the processing times of that implementation. Section 5 briefly presents three ZIO TEST suites written to test our implementation. Section 6 concludes with an overview of ZIO and ZIO HTTP to the light of the experience acquired while implementing our service, and some references.

2. ZIO as the platform of choice for the implementation of our REST service.

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). Constructing programs using this mechanism in practice means solving a programming problem by first dividing it into simpler parts and then combining the solutions of the parts as a solution of the whole. This is the “divide-and-conquer” strategy that allows developers to build correct software no matter the degree of complexity involved. The implementation of this strategy through the composition of pure functions constitutes the simplest and most powerful way to apply it.

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 developers 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 effectual 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.

The general definition of a ZIO functional effect takes the form of 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.

Another important characteristic of ZIO's functional effects is its approach to error handling, that encourages dealing with E-values with the same attention given to the processing of A-values. Frequently, the full processing capabilities available in functional effect-processing platforms are applied only to the success values returned by functions, especially not side-effecting ones. But many functions used in real-world programming are effectual and can fail in predictable ways. Even some, otherwise pure, functions, can raise exceptions, which makes them actually effectual (take as an example integer division with a zero denominator). For those, and all other foreseeable error possibilities, ZIO provides the E "error channel" with the same functional processing capabilities as the A channel, where the "normal" flow of data proceeds.

ZIO's functional effects provide a simple and powerful way of solving the dependency injection problem using the `R` resource type, which allows to build a side-effecting program using `ZLayer` s, which are software layers used to define the modular structure of a ZIO application. A `ZLayer` stipulates the resources that it needs (expressed explicitly by the `R` type), as well as, if necessary, the logic to automatically manage its life cycle (defined by the functions to be used in order to acquire and release those resources). When the “big ZIO” representing a complete application composed by many layers 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.

To better understand the code presented in this article, it is desirable an understanding of the following ZIO features at a basic level:

- Constructing functional effects using Scala code and smart constructors like `succeed` , `fail` and `attempt` .
- Composing functional effects using the combinators `orElse` , `zip` , `zipLeft` and `zipRight` .
- Transforming functional effects using the higher-order functions `map` , `mapError` , `flatMap` , and the monadic processing of functional effects using a for-comprehension.
- Using the default services provided by ZIO: `Console` , `Clock` , `System` and `Random` .
- Defining and providing `ZLayer` s, as well as safely managing the life cycle of their resources.

All that basic knowledge of ZIO can be acquired very quickly, if needed, following a ZIO tutorial. There are many freely available in the cloud, we suggest to start with those included in ZIO's official documentation.

ZIO HTTP is a library for building type-safe, purely functional HTTP servers and clients using ZIO's functional effects. As such, it inherits all the simplicity and power of ZIO for the pure-functional development of effectual programs in Scala.

The current version of ZIO HTTP (1.0.0-RC4 at the time of writing this article) includes a DSL for the high-level definition of REST endpoints that allows developers to overlook the low level details of defining routes and handlers, freeing them from writing the repetitive and error-prone code required to decode the query parameters, headings and bodies of the HTTP requests and encode the corresponding HTTP responses.

The high-level definition of endpoints makes use of the `Endpoint` smart constructor for specifying a URL and a small amount of auxiliary functions, the more frequently used being `query`, `header`, `in`, `out` and `outErrors`, whose detailed use in our case is explained in Section 3.3. Additionally, the definition makes use of several kinds of annotations and schemas associated to the types of the domain model and the error model of an application, which provides to ZIO HTTP the meta-data needed to support the automatic endpoints implementation and other related high-level services.

Within the ZIO ecosystem, a `Schema[A]` describes the structure of some data type `A`. Schemas model the structure of data types as first class values, so they can be introspected, transformed, and combined using ordinary Scala code. ZIO provides implicit schemas for all standard Scala types, and it is possible to automatically derive a schema for a custom-defined data type `A` by simply calling `DeriveSchema.gen[A]`.

The meta-data provided with the high-level endpoints definition allows the automatic generation of the corresponding Open API documentation and of a safe-type client useful for the testing of a REST-API from the OS prompt and for its incorporation into automation bash (or Scala-cli) scripts. In this way, the endpoints definition and associated schemas and annotations become the "single source of truth" for all the functionalities provided by ZIO HTTP for a given REST-API implementation. Anyone who has had to manually synchronize program code with its documentation will greatly appreciate this feature.

As previously said, the modular structure of a ZIO application is defined in terms of `ZLayer` s. These are software layers implemented as traits that specify the services they provide and the resources they need, and "live" objects that extend those traits providing the logic needed to implement the services and also, if necessary, the logic to safely manage the corresponding resources. The Scala code implementing these traits and live objects is not discussed in detail in this article for all the `ZLayer` s composing our application, but can be examined, if desired, by delving into its source code (the link is provided in Section 6).

Our ZIO application is composed by four `ZLayer` s: `RestCLient`, `RestServer`, `RestServerCache` and `RedisServerClient`.

The `RestCLient` layer uses ZIO HTTP for implementing the access to the GitHub REST API, in order to get the data needed to build the response to a given request.

The logic for building a response is managed by the `RestServer` layer, which also associates that logic to the endpoint serving a request to our service. We use the ZIO HTTP Endpoints API for a high-level definition of that endpoint.

The `RestServerCache` layer is responsible for managing the REDIS cache where the contributions by repository of the previously requested organizations are saved.

Finally, the `RedisServerClient` layer manages the embedded REDIS server and the REDIS client used by the `RestServerCache` layer.

3. The modules of our ZIO HTTP implementation.

3.1 The domain and error models.

The domain model of our application is defined inside the `DomainErrorTypes` standard Scala module (just an object whose members can be imported wherever necessary), mainly by the following code:

```scala
final case class Contributor(
                              @description("Contributor's repository") repo: String,
                              @description("Contributor's name") contributor: String,
                              @description("Contributor's number of contributions") contributions: Int
                            )
object Contributor {
  implicit val jsonCodec: JsonCodec[Contributor] = DeriveJsonCodec.gen[Contributor]
  implicit val schema: Schema[Contributor] = DeriveSchema.gen[Contributor]
}
```

This code shows the annotations that provide meta-data about the elements of our data model, as well as the use of `DeriveSchema.gen` to generate the implicit schemas needed to support the high-level services provided by ZIO HTTP. Besides, this code shows the use of `DeriveJsonCodec.gen` , a function that generates implicit JSON encoders and decoders used behind the scenes, together with all the other high-level supporting devices, to drastically simplify the implementation of a REST-API in Scala using ZIO HTTP.

Another important feature of the `DomainErrorTypes` module regards laying the foundations for the custom error-handling of our application, based on our error model:

```scala
sealed trait ErrorTypeH

object ErrorTypeH {
  final case class OrganizationNotFound() extends ErrorTypeH
  final case class LimitExceeded() extends ErrorTypeH
  final case class UnexpectedError() extends ErrorTypeH

  implicit val organizationNotFoundSchema: Schema[ErrorTypeH.OrganizationNotFound] =
    DeriveSchema.gen[ErrorTypeH.OrganizationNotFound]
  implicit val limitExceededSchema: Schema[ErrorTypeH.LimitExceeded] =
    DeriveSchema.gen[ErrorTypeH.LimitExceeded]
  implicit val unexpectedErrorSchema: Schema[ErrorTypeH.UnexpectedError] =
    DeriveSchema.gen[ErrorTypeH.UnexpectedError]
}
```

This code defines a two-level hierarchy of error types, with root `ErrorTypeH` , which represent the possible ways in which a call to the GitHub REST-API can go wrong, namely:

- GitHub cannot find the requested organization.
- The API rate limit of GitHub has been exceeded.
- GitHub returns an unexpected error status code (an "umbrella" type covering any error situation not explicitly considered).

Note that within the object defining our error model we generate implicit instances of ZIO schemas for the error types, exactly as we did for the types of our domain model.

3.2 The REST client module.

In this module, in charge of making requests to the GitHub REST service, we will discuss in detail two important features . The first one regards managing the pagination of the GitHub responses by means of an auxiliary recursive function with an accumulating parameter. The purpose of this function can be explained 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 Scala code is the following:

```scala
private def processResponseBody[T](url: String)(processPage: BodyType => List[T]):
  ZIO[Client, ErrorTypeH, List[T]] = {
    def processResponsePage(processedPages: List[T], pageNumber: Int): ZIO[Client, ErrorTypeH, List[T]
      getResponseBody(s"$url?page=$pageNumber&per_page=100").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) =>
          ZIO.fail(error)
      }
    }
    processResponsePage(List.empty[T], 1)
  }
```

The `processResponseBody` function is 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 `BodyType` , a type synonym of `String`) to a `List[T]` . The auxiliary recursive function with an accumulating parameter is `processResponsePage` .

As we can see, the returned type of `processResponsePage` (`ZIO[zio.http.Client, ErrorTypeH, List[T]]`) reflects the fact that our ZIO implementation works 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 if needed. In this case, the desired composition is achieved using the `flatMap` higher-order function, 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).

The second important feature of this module regards `getResponseBody` , which returns a functional effect of type `ZIO[Any, Nothing, Either[ErrorTypeH, BodyType]]` . In the result type of this ZIO, `BodyType` is, as previously said, just a type synonym of `String` , `ErrorTypeH` is the root of our hierarchy of error types, and their `Either` combination represents the potentially successful or failed result of the HTTP request made.

Within this function, the different error status codes that the GitHub REST API can return are mapped to subtypes of `ErrorTypeH` , which are propagated through the ZIO error channel for their eventual translation into responses that can help the end user to understand what went wrong.

```
private def getResponseBody(urlS: String): ZIO[Any, Nothing, Either[ErrorTypeH, BodyType]] = {
  def responseBody(body: BodyType, status: Status): Either[ErrorTypeH, BodyType] =
    status match {
      case Status.Ok =>
        Right(body)
      case Status.Forbidden =>
        Left(new ErrorTypeH.LimitExceeded)
      case Status.NotFound =>
        Left(new ErrorTypeH.OrganizationNotFound)
      case _ =>
        Left(new ErrorTypeH.UnexpectedError)
    }
  val token = if (gh_token != null) gh_token else ""
  val url = URL.decode(urlS).toOption.get
  val request = Request(Version.Default, Method.GET, url, Headers("Authorization" -> token))
  val bodyZ = for {
    response <- Client.request(request)
    bodyString <- response.body.asString
  } yield responseBody(bodyString, response.status)
  bodyZ.provide(Client.default, Scope.default).orDie
}
```

Here, `Client.request` is the ZIO HTTP function used to make an HTTP request, it takes as parameter an object that contains the URL, method, headers and body (in our case empty) of the request.

3.3 The HTTP functions of 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).

Using a DSL provided with that purpose, ZIO HTTP allows the following high-level definition of our endpoint:

```
trait EndPoints {
  val gl = paramStr("group-level").optional ?? Doc.p("Grouping level parameter")
  val mc = paramInt("min-contribs").optional ?? Doc.p("Minimum contributions parameter")
  val getContributorsEndPoint =
    Endpoint(Method.GET / "org" / string("organization") / "contributors").
      query(gl).
      query(mc).
      outErrors[ErrorTypeH](
        HttpCodec.error[ErrorTypeH.OrganizationNotFound](Status.NotFound),
        HttpCodec.error[ErrorTypeH.LimitExceeded](Status.Forbidden),
        HttpCodec.error[ErrorTypeH.UnexpectedError](Status.InternalServerError)
      ).
      out[List[Contributor]](Doc.p("List of contributions")) ?? Doc.p("REST-API endpoint")
}
```

Here we make use of the previously mentioned `Endpoint` smart constructor and its auxiliary functions, in the following way:

- The `EndPoint` constructor allows a parametric specification of a URL with fixed path segments (just character strings separated by the `/` combinator) and parametric ones, like `string("organization")` , which defines a path segment in the stipulated position that has to be of the indicated type (`String` in this case) and will be available in the handler code for the endpoint as a parameter with the indicated name (`organization` in this case). There are functions for defining path segments of other types, like, for example, `int(<variable-name>)` , which allows defining a path segment representing an integer value.
- The `query` function allows to stipulate request parameters of our endpoint, using for that purpose variables defined using the functions `paramStr` (for parameters of type `String`) and `paramInt` (for parameters of type `Int`), normally specified also as optional, as in our case, which changes their type from `A` to `Option[A]` .
- The `outErrors` function is used to map the error types defined in our custom error model to selected HTTP error status codes that provide to the end user information about the cause of a failed request.
- The `out` function is used to stipulate the type of the elements of our response bodies, using the types defined in our custom domain model.
- The `in` function (not shown here because our REST calls have empty bodies) can be used in the same way to stipulate the type of the elements of our request bodies, again using the types defined in our custom domain model.

Notice the calls to the `Doc.p` function attached by means of the `??` combinator to diverse parts of our endpoint high-level definition with the purpose of providing to ZIO HTTP additional meta-data related to our endpoint.

Once defined the `EndPoints` trait containing the Endpoints API definition of our REST endpoint, we can extend it for diverse purposes, as in the following code where we substantially define the ZIO HTTP `Handler` for the endpoint of our REST service:

```
object EndPointsServer extends EndPoints {
  def handleGetContributorsEndPoint(organization: Organization,
                                    groupLevel: Option[String],
                                    minContribs: Option[Int]): ZIO[Client, ErrorTypeH, List[Contributo
    ZIO.logSpan("getContributors") {
      contributorsByOrganization(organization,
                                 groupLevel.getOrElse("organization"),
                                 minContribs.getOrElse(0))
    } @@ ZIOAspect.annotated("organization" -> organization,
                             "groupLevel" -> groupLevel.toString
```

```
                                          "groupLevel" -> groupLevel.toString,
                                         "minContribs" -> minContribs.toString)
    }
    val handlerContribsGH3Z: Handler[Client, ErrorTypeH, (Organization, Option[String], Option[Int]), Li
      handler(handleGetContributorsEndPoint _)
    val routesContribsGH3Z: Routes[Client, Nothing] = Routes(getContributorsEndPoint.implement(handlerCo
    val appContribsGH3Z: HttpApp[Client] = routesContribsGH3Z.sandbox.toHttpApp
  }
```

Here:

- We define the function `handleGetContributorsEndPoint` that takes as arguments the variable names (and their corresponding types) used in the `EndPoints` trait for representing the path segments and request parameters of a request (i.e. `organization` , `groupLevel` and `minContribs`) and returns a `ZIO[Client, ErrorTypeH, List[Contributor]]` using in its success and error channels the types defined in our custom domain and error models. The data used to build a response to a given request is returned by the `contributorsByOrganization` function, that will be discussed in Section 3.4.
- We define a `Handler` for our endpoint using the `handler` function that takes as argument the `handleGetContributorsEndPoint` function just discussed.
- Finally, we define a `Routes` instance and an `HttpApp` instance using the standard functions provided by ZIO HTTP for that purpose, which are ultimately based on the handler whose building was just described.
- Note that in this code no mention is made of the JSON encoder necessary to build our response bodies. As said before, thanks to the Endpoints API, all the necessary JSON encoding/decoding is automatically managed by ZIO HTTP. Note, also, that nothing can be seen in this code that explicitly disassembles a request or assembles a response.
- We can also observe the use of more annotations that provide to ZIO HTTP still more meta-data related to our REST service.

The `HttpApp` instance defined in the last line of the preceding code block, is the object that ZIO HTTP needs to launch an HTTP server for our REST-API.

The `EndPoints` trait can also be extended to allow the definition of a ZIO application that generates and prints to the console the Open-API documentation of our REST service:

```
  object EndPointDoc extends ZIOAppDefault with EndPoints {
    val docs = OpenAPIGen.fromEndpoints(getContributorsEndPoint)
    val run = Console.printLine(docs.toJson)
  }
```

3.4 The processing functions of the REST server module.

Until now, the 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.

The fact that this traversable capability in ZIO exists in a sequential and a parallel version (`collectAll` and `collectAllPar` , respectively), will allow us to switch between very different implementations of our endpoint, from the efficiency point of view, by simply using one version instead of the other, as explained below.

The main functionality of the REST server module is implemented as a functional effect with type `ZIO[Client, ErrorTypeH, List[Contributor]]` , as follows:

```
def contributorsByOrganization(organization: Organization, groupLevel: String, minContribs: Int):
  ZIO[Client, ErrorTypeH, List[Contributor]] = for {
    repos <- restClient.reposByOrganization(organization)
    contributorsDetailed <- contributorsDetailedZIOWithCache(organization, repos)
    contributors = groupContributors(organization, groupLevel, minContribs, contributorsDetailed)
  } yield contributors

private def contributorsDetailedZIOWithCache(organization: Organization, repos: List[Repository]):
  ZIO[Client, ErrorTypeH, 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(8)
    // 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, contributorsD
    } 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 its repositories as a `List[Contributor]` , in both cases, of course, as ZIO result types. The core functionality is implemented by `contributorsDetailedZIOWithCache` .

This functionality, make use of `contributorsByRepo` to get from GitHub the contributors of each single repository of the given organization. The calls to this function are made in parallel, applying the traversable capabilities by means of `ZIO.collectAllPar` .

The `reposByOrganization` and `contributorsByRepo` are functional effects provided by `restClient` , both based on the generic capability of `processResponseBody[T]` explained at detail in Section 3.2.

A peculiarity of the ZIO implementation of our REST-API is worth emphasizing once again: 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 strong implications of this trivial change in our code, from the point of view of its run-time efficiency, will be presented in Section 4.

The REST server module also includes the `groupContributors` pure function, used to group by repository the contributions of a given organization. As we can see in the code just shown, it is used as an auxiliary function by `contributorsByOrganization` , which is the effectual function that actually implements our endpoint by returning, as a deferred effect, the `List[Contributor]` that will be transformed by ZIO HTTP into the desired HTTP response.

3.5 A module to encapsulate the Redis services needed by the cache.

One of ZIO's strengths is its comprehensive solution of the dependency injection problem, 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 ZIO application, we decided to use two Java libraries, EmbeddedRedis and Jedis, to show how easily Scala allows to take advantage of the huge amount of functionality available in the Java ecosystem. Using those Java libraries from ZIO was not only almost trivial, but it also allowed to easily avoid the risk of leaking both resources, 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` . 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. 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.

3.6 A module for the services provided by the cache.

Having available the `RedisServerClient` ZIO layer containing the EmbeddedRedis server and Jedis client, we can put the cache services themselves into 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
      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.

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 3.5) 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.

3.7 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 ContribsGH3Z extends ZIOAppDefault {
    override val run = {
      ZIO.serviceWithZIO[RestServer](_.runServer).
        provide(
          RestClientLive.layer,
          RestServerLive.layer,
          RestServerCacheLive.layer,
          RedisServerClientLive.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. Notice, in particular, how the `ZLayer` s are provided as a plain structure, leaving to ZIO the task of automatically infer and manage their hierarchical structure.

4. Efficiency of our ZIO HTTP implementation.

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

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

Sequential

ContribsGH3-Z - Starting REST API call at 13-01-2024 19:02:13 - organization='revelation' ContribsGH3-Z - # of repos=24

ContribsGH3-Z - repo='globalize2' retrieved from GitHub ContribsGH3-Z - repo='revelation.github.com' retrieved from GitHub ... ContribsGH3-Z - repo='ember-rails' retrieved from GitHub ContribsGH3-Z - repo='ey-cloud-recipes' retrieved from GitHub

ContribsGH3-Z - Finished REST API call at 13-01-2024 19:02:32 - organization='revelation' ContribsGH3-Z - Time elapsed from start to finish: 19,15 seconds

Parallel

ContribsGH3-Z - Starting REST API call at 13-01-2024 19:35:06 - organization='revelation' ContribsGH3-Z - # of repos=24

ContribsGH3-Z - repo 'globalize2' retrieved from GitHub ContribsGH3-Z - repo 'almaz' retrieved from GitHub ... ContribsGH3-Z - repo 'rails' retrieved from GitHub ContribsGH3-Z - repo 'ey-cloud-recipes' retrieved from GitHub

ContribsGH3-Z - Finished REST API call at 13-01-2024 19:35:11 - organization='revelation' ContribsGH3-Z - 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:

ContribsGH3-Z - Starting REST API call at 13-01-2024 19:46:11 - organization='revelation' ContribsGH3-Z - # of repos=24

ContribsGH3-Z - repo 'revelation-globalize2' retrieved from cache ContribsGH3-Z - repo 'revelation-revelation.github.com' retrieved from cache ... ContribsGH3-Z - repo 'revelation-ember-rails' retrieved from cache ContribsGH3-Z - repo 'revelation-ey-cloud-recipes' retrieved from cache

ContribsGH3-Z - Finished REST API call at 13-01-2024 19:46:12 - organization='revelation' ContribsGH3-Z - 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 by using a cache implemented as an in-memory Redis data-structure server.

5. Testing the ZIO HTTP implementation of our REST-API.

The source code of the ZIO HTTP implementation of our REST-API also contains ZIO TEST code aimed at testing its most important functions.

Our testing code contains three test applications:

- `GroupContributorsSpec` , for testing the pure function `groupContributors` .
- `ContributorsByOrganizationLiveEnvSpec` , for testing the effectual function `contributorsByOrganization` with the RestServer's live layer whose code has been analyzed in Section 3.4.
- `ContributorsByOrganizationTestEnvSpec` , for testing again `contributorsByOrganization` , but using a RestServer's test layer with predefined offline data for the contributions, instead of the live layer used in the previous test application, which gets the contributions online from GitHub. In that way, it is possible to compare the total number of contributions returned by our endpoint with a total calculated in advance from the predefined data. We cannot do that with the total obtained from GitHub online, because it may change over time with the addition of new contributions.

As testing is not, strictly speaking, part of our present subject, we refer the interested reader to this article, where the testing of our REST-API implementation is discussed extensively, although for a version that does not use the Endpoints high-level API of ZIO HTTP. Notwithstanding this, all the considerations made in that article apply to the present implementation.

6. Main benefits of ZIO and ZIO HTTP. References.

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 fundamental declarative character of pure functional programming.

This advantage is put in the hands of a Scala programmer without resorting to the advanced features of Scala. 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 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.
- **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. ZIO also provides tools for concurrency and parallelism at a higher level, which make the explicit use of fibers unnecessary in most cases, such as, to name a few, `Ref` , to safely share mutable state between fibers, `Semaphore` to control the degree of concurrency of fibers, and `STM` (Software Transactional Memory) that allows performing a group of memory operations as a single atomic operation.
- **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, and is fully integrated with ZIO.
- **Large, continuously evolving, ecosystem.** Composed of official ZIO libraries and community-maintained libraries.

All the mentioned advantages of ZIO are directly inherited by ZIO HTTP. The specific advantages of ZIO HTTP, as in the current version at the time of writing, can be summarized as follows:

- High throughput inherited from the Netty asynchronous (non-blocking) networking framework.
- High-level Endpoints API for the simplified definition and implementation of REST-APIs.
- ZIO schemas and annotations to support the automatic generation not only of the endpoints of a REST service, but also of its Open API documentation.
- Automatic generation of type-safe clients.
- Type-safe HTML basic templating (basic at the moment ...) to create web applications with server-side rendered pages.
- A code base in continuous evolution where great improvements are frequently included.

To close, we provide some references that can be useful to complement the information provided in this article about ZIO and ZIO HTTP:

- Jorge Vasquez' presentation at Functional Scala 2023 about the Endpoints API of ZIO HTTP can be found here.
- The first part of a ZIO HTTP tutorial by Jakub Czuchnowski, presents good examples of custom error handling, using the traditional way of defining endpoints.
- The official documentation of ZIO HTTP.
- The ZIO "bible" Zionomicon, co-written by John A. De Goes, the author of ZIO.

The Scala source code of this article can be found here.