Errors happen. Can we get rid of them using ZIO Test?

1. Introduction.

Edsger W. Dijkstra, a genius from the early days of computer science, who made impressive contributions to the fields of graph algorithms, concurrent processes, nondeterministic programming (among many others), and carried out equational reasoning to its highest peaks, made this logically irrefutable statement:

> Tests can only prove the presence of errors, not their absence.

Reflecting, in a certain sense, a controversial attitude towards testing, such a categorical dictum allows us to draw a fundamental lesson:

Failed tests provide evidence of errors, but successful tests do not constitute proofs of correctness. Tests can help to debug programs, but not to solve the problems associated to the design and development of correct programs. Simple and understandable mental models of our programs are essential for that purpose, and functional programming can contribute a lot in that sense.

Because writing correct programs is not easy, as another quote from Dijkstra emphatically states:

> We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty.

And the only way to deal with high complexity is applying the power of abstraction (Dijkstra again):

> We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called “abstraction”.

Pure functional programming excels in this regard, since abstraction is applied in the simplest and most “natural” way conceivable, by using pure function composition as the fundamental way of building programs.

But embracing the fundamental implication of Dikstra's claims should not prevent us from putting testing into a more practical perspective, considering that we are not verifying logical statements but engineering devices that must abide to a (hopefully) precise and complete specification. And good tests, considered as executable specifications, can be extremely helpful. So much so, that writing tests as a translation of a set of specifications can even be considered the best first step of the software development process, as claimed by test-driven development TDD.

Besides, software testing has become a field in itself and developed concepts and technologies of great value for the exercise of software engineering. One of the most relevant in our opinion, property-based testing, uses test cases not devised "by hand" but generated automatically in a random manner; a technique that, with properly generated test case distributions, can be useful to identify corner cases, as well as to give a broader base of confidence to our tests. Of course, random test cases cannot replace test cases specifically intended to verify a property the tester has in mind (guided, as said before, by a good specification). Instead, the combination of good example-based and random-based tests can give testers the confidence in the tested programs they are looking for.

We are implicitly talking about the tester as someone different from the programmer because programming and testing play clearly different roles in software development. But many times both roles are performed by the same person and, in fact, it is good practice to put programmers in front of tests devised to prove, not as logicians but rather as defendants without presumption of innocence (as "Very Humble Programmers", Dijkstra would say), that their code does not break the corresponding specification. Of course, third-party testing is also very important, to the point that it has given rise to a whole new branch of the software industry, but that's another story.

> As an anecdotal note, Dijkstra was not precisely diplomatic, at times, when expressing the conclusions of his always rigorous thinking. That provoked angry reactions on more than one occasion, but also humorous ones, such as this: "Arrogance in computer science is measured in nano-Dijkstras". A serious accusation of arrogance would clearly be unfair directed to someone who advocated humility as an essential virtue in a programmer when receiving the ACM Turing Award for "... his eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness ..."

And now, what is in essence testing? Testing is all about detecting errors to get rid of them. And since errors, besides being unavoidable, cover a really broad spectrum, it should seem natural to start our discussion of the testing of ZIO applications with an attempt to classify errors from the ZIO perspective.

Our classification attempt should start with the more general, not ZIO-specific, distinction between programming errors and run-time application errors. In other words, errors made by the programmer when writing a program, and errors that will face the program (as foreseen by the programmer) when it runs. This article mainly covers the former, also considering laterally how ZIO helps deal with the latter. As we will see, the classification of errors from the ZIO perspective goes far beyond this first classification step.

In a previous article we wrote about a REST-API implemented with ZIO, whose testing using ZIO Test is our present subject.

In this article, Section 2 introduces the approach to error processing adopted by ZIO and the main characteristics of ZIO Test. Section 3 explains ZIO Test with a little more detail by means of a few selected examples. Section 4 gives an extended example of ZIO Test by applying it to test the implementation of the REST-API just mentioned. Section 5 concludes with some brief remarks and a few references.

> We would like to close this introduction with a side note addressed to the more theoretically inclined readers. A few lines above, we stated that as a basis for a practical approach to testing we should consider that we are not dealing with logical statements but with engineering devices. Actually, a correct program (written in an appropriate functional programming language) **is** a theorem in the theory of types, as the Curry-Howard isomorphism, the isomorphism between proofs and programs, states. You can find a wonderful presentation of this important proof-theoretic result here.

2. Error handling and testing in ZIO.

ZIO 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 effects through which our programs can interact with the outside world. For that purpose, ZIO uses the so-called functional effects, or just ZIO's for brevity, which are in essence immutable data definitions of side effects which can be treated as first-class values to compose them (and functionally process them in general terms), and whose execution is suspended until the so-called "end of the world", the final execution of a side-effecting program. 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).

So, functional effects are delayed effects. As a mean for developing concurrent programs, they are in that sense the exact opposite of futures, whose creation immediately triggers their execution, with no possible way of interacting with them until they have finished executing. Instead, functional effects are implemented using fibers, which are virtual threads that can be explicitly scheduled, interrupted or canceled, besides being implicitly processable, as already said, using all the simple and powerful tools of functional programming. 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 explicit concurrency and parallelism at a higher level, such as mutable references, software transactional memory, etc., which make the direct use of fibers unnecessary in most cases.

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, considering exceptions an extreme case, so to speak, of side effect (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.

Returning to errors, our main subject in this article, a second classification step, this time for the run-time errors, constitutes an essential aspect of the ZIO error model: the distinction made between failures and defects. Defects are the unexpected errors and are treated basically as the "escape doors" of an application. All the others, the foreseeable errors or failures, are instead modeled using the same techniques used to model the successful values, i.e. by defining error types that are used to feed the E "error channel" of functional effects in the same way as the domain-model types feed the A "success channel". These error types are arranged in a structure that sets up the custom error model of a ZIO application.

The defects, or unrecoverable errors, are the unexpected exceptions. They don't show up in the error channel and cannot be processed using the standard error processing capabilities of ZIO. They aren't even caught by `catchAll` , a function that can be used to (re)define the response to all kind of failures in ZIO. To be caught, they require a special `catchAllDefect` function, that basically gives a way to intercept the unrecoverable errors to finish an application with a more intelligible error message than those usually provided by unexpected exceptions. The failures, on the other hand, since they are predictable and classifiable, can be processed using the same techniques applicable to the normal return values of functions.

Regarding testing, ZIO Test features effectual tests as first-class values. Functional effects don't need to be explicitly run in order to be tested (using, for example, `unsafeRun` , as would be the case with other test frameworks). All the standard capabilities of functional effects can be used inside the test code, taking for granted that the functional effects used will be properly executed when their results are needed to evaluate the tests. This tight integration of ZIO Test with ZIO makes testing effectual functions as natural as testing pure ones.

A modular application composed of several layers can be tested using a `provide` functionality, conceptually identical to that available outside the test environment, which allows the replacement of an application layer with another deliberately developed with a specific test in mind.

ZIO Test supports property-based testing. For that purpose, it provides: i) predefined data generators for the standard data types, ii) functions to compose generators that allow to easily build data generators for user-

defined types, and iii) a `check` function, that takes as arguments a certain number of generators, and a property checking function used to apply to the generated values an assertion expressing the property of interest.

If a property being checked fails, the failure is automatically "shrunk" to the smallest failing cases, to ease the diagnosis of the problem. To achieve reproducibility of tests, the generators are deterministic, i.e. running a generator multiple times produces the same sequence of values (whether random or not).

To finish our quick introduction to ZIO Test, let's mention test aspects, which are powerful tools for modifying the behavior of existing individual tests or groups of tests, used to solve cross-cutting issues. Test aspects cover a wide range of purposes and are easily composable.

We did not intend to give in this section a complete overview of ZIO Test. Our purpose was just to present the main concepts and peculiarities of ZIO Test, as needed to understand the code used for testing the ZIO REST API implementation mentioned in Section 1. The following section gives some examples of the use of ZIO Test that were selected with that purpose in mind.

For a complete introduction to ZIO Test and a detailed explanation of all its features, we refer the reader to the official documentation.

3. Characteristics of ZIO Test.

A testing application in ZIO Test is defined as an object that extends the `ZIOSpecDefault` trait and overrides the `spec` method, which corresponds to a specification composed of suites, which in turn are groups of tests that can contain other suites, giving the test application the desired hierarchical structure. That structure is reflected when the test results are displayed, labeling each suite and test with the string used as the first parameter of its definition. Each test verifies an assertion, which closes the Scala code segment that defines it. That code can include pure functions as well as effectual ones.

Assertions in ZIO Test are of two types: i) the classic assertions, defined using the `assert` function for ordinary values and the `assertZIO` function for effectual values, and ii) the smart assertions, defined using the `assertTrue` function, that provides a unified syntax for checking both ordinary and effectual values, and is the recommended way of making assertions with the current version of ZIO Test because of its simplicity and greater readability (although classic assertions are still available in case they were preferable for some specific purpose).

ZIO Test provides a large repertoire of built-in assertions. Their classification based on the result-type can be very useful to find the most suitable one for a specific purpose. For examples of the use of that classification, we refer the reader again to the documentation of ZIO Test.

Assertions can be composed using logical operators like `&&` for conjunction, `||` for disjunction, and `!` for negation.

The following example shows a testing app for a specification consisting of two suites, containing 2 and 3 tests, respectively, that verify some simple assertions for ordinary and effectual values.

```
object AssertionsSpec extends ZIOSpecDefault {
  def spec = suite("AssertionsSpec") (
    suite ("Suite 1 with 2 classic-assertion tests") (
      test("Nested and logically composed assertions") {
        val x = Right(Some(7))
        val y = 1 + 1
        assert(x)(isRight(isSome(equalTo(7)))) && assert(y)(equalTo(2))
      },
      test("Fourth value of a vector is equal to 3") {
        val xs = Vector(1, 3, 5, 7)
        assert(xs)(hasAt(3)(equalTo(7)))
      }
    ),
    suite ("Suite 2 with 3 smart-assertion tests") (
      test("Assertion on ordinary value: 1 + 1 == 2") {
        val x = 1 + 1
        assertTrue(x == 2)
      },
      test("Assertion on ordinary value: fourth value of a vector is equal to 3") {
        val xs = Vector(1, 3, 5, 7)
        assertTrue(xs(3) == 7)
      },
      test("Assertion on effectual value: update successfully increments by one a Ref[Int]") {
        for {
          r <- Ref.make(0)
          _ <- r.update(_ + 1)
          v <- r.get
        } yield assertTrue(v == 1)
      }
    )
  ) @@ TestAspect.sequential
}
```

The first suite of the example shows two tests with classic assertions. The first test exemplifies the use of nested classic assertions and logical operators on assertions. The second test exemplifies the access to the elements of a vector using a classic assertion.

The second suite shows three tests with smart assertions, two applied to ordinary values and one applied to an effectual value. The second smart assertion for an ordinary value verifies the same condition as the second classic assertion of the first suite, clearly showing the advantages of smart assertions regarding clarity of expression. The third smart assertion is applied to an effectual value extracted from a `Ref[Int]`, initialized to zero and incremented by one within a for-comprehension.

A `Ref[A]` is a ZIO that provides a purely-functional reference to a mutable state of type `A`, with functions that describe allocating and modifying the mutable state, rather than modifying the state directly. Our last example test uses the functions: `make` to initialize the state (of type `Int`), `update` to apply a function (of type `Int => Int`) to that state, and `get` to retrieve the updated value of the state, all within a for-comprehension that yields the final value of the state within a smart assertion applied to it. As explained before, all the effectual actions are actually executed only when needed, at the "end of the world", i.e. when we execute the test.

Our first example also shows the use of the test aspect `sequential` applied to the entire spec, with the effect of executing its elements in a strictly sequential way, ensuring the display of the results in the order specified within the test code. Without the use of this aspect, the results would be displayed in a non-deterministic order, since tests are executed, by default, in parallel.

The example test application can be executed by issuing the following command:

```
sbt "testOnly AssertionsSpec"
```

The execution causes the display of the following result at the console:

```
+ AssertionsSpec
  + Suite 1 with 2 classic-assertion tests
    + Nested and logically composed assertions
    + Fourth value of a vector is equal to 3
  + Suite 2 with 3 smart-assertion tests
    + Assertion on ordinary value: 1 + 1 == 2
    + Assertion on ordinary value: fourth value of a vector is equal to 3
    + Assertion on effectual value: update successfully increments by one a Ref[Int]
```

ZIO includes standard services (`Clock` , `Console` , `System` , and `Random`) that can be used within ZIO Test in special versions suited to all kinds of testing purposes. We can, for example, simulate the pass of time with the `TestClock` service, or use the `TestConsole` service to accumulate within a `Vector[string]` the results of `printLine` calls.

The following example shows the use of a smart assertion to verify that the result of executing `printLine` equals the expected value of the `Vector[String]` associated with the `TestConsole` . Note that the `sayHello` function, executed within the test for-comprehension, is defined as acting directly upon the standard `Console` , which is automatically simulated at testing time by `Testconsole` .

```
object HelloWorld {
  def sayHello: ZIO[Any, IOException, Unit] =
    Console.printLine("Hello, World!")
}

import HelloWorld._

object HelloWorldSpec extends ZIOSpecDefault {
  def spec = suite("HelloWorldSpec") (
    test("sayHello correctly displays output") {
      for {
        _ <- sayHello
        output <- TestConsole.output
      } yield assertTrue(output == Vector("Hello, World!\n"))
    }
  )
}
```

As previously said, test aspects are powerful tools for modifying the behavior of already written tests. They encapsulate cross-cutting concerns that can be easily applied after we have resolved the main concerns of our tests, thus allowing us to increase their modularity. Test aspects can be used to modify individual tests, suites, or entire specs, using a convenient `@@` syntax that puts them in clear evidence and also allows them to be easily composed sequentially.

For an example of the composition of test aspects, consider the following test:

```
test("Aspect composition example") {
  assertTrue(1 + 1 == 2)
} @@ TestAspect.jvmOnly @@ TestAspect.repeat(Schedule.recurs(5))
```

This test will be executed only on the JVM, and will be scheduled to repeat 5 times after its first execution.

There are test aspects for a great number of concerns, as, for example:

- environment access (like `jvmOnly`, just exemplified),
- execution strategy (like `sequential`, seen in a previous example),
- execution configuration (like `repeat`, just exemplified),
- stipulation of logic to be executed before, after, and around tests (like `after`, of which an example follows),
- many others that the reader can consult in the documentation of ZIO Test.

The following example presents a test-counting spec, taken from the documentation, that exemplifies the use of the `after` test aspect, as well as the provision to the spec of a `ZLayer` with a fully managed resource. This layer uses a `Ref[Int]` for test-counting purposes, specifying its acquire and release logic by means of the `ZIO.acquireRelease` function.

```
case class Counter(value: Ref[Int]) {
  def inc: UIO[Unit] = value.update(_ + 1)
  def get: UIO[Int] = value.get
}

object Counter {
  val layer =
    ZLayer.scoped(
      ZIO.acquireRelease
        (Ref.make(0).map(Counter(_)) <* ZIO.debug("Counter message - initialized!"))
        (c => c.get.debug("Counter message - number of tests executed"))
    )
  def inc = ZIO.service[Counter].flatMap(_.inc)
}

object CounterSpec extends ZIOSpecDefault {
  def spec = {
    suite("Spec1")(
      test("test1") {
        assertTrue(true)
      } @@ TestAspect.after(Counter.inc),
```

```
      test("test2") {
        assertTrue(true)
      } @@ TestAspect.after(Counter.inc)
    ) +
      suite("Spec2") {
        test("test1") {
          assertTrue(true)
        } @@ TestAspect.after(Counter.inc)
      }
  }.provideShared(Counter.layer)
}
```

In this example, the `after` test aspect is used to trigger an increment-by-one function, defined for the `Ref[Int]` resource managed by a `ZLayer` provided to the spec using the `provideShared` function. Note how the release logic for that resource is used to display the number of tests executed when the test app finishes (and, as a consequence, the resource release function is automatically executed). Note also that the initial value of the resource is set automatically by its acquire function when the test app starts. Both events are notified to the console using ZIO debug messaging functions.

The execution of this test application displays the following output to the console:

```
Counter message - initialized!
+ Spec1
  + test2
  + test1
+ Spec2
  + test1
Counter message - number of tests executed: 3
```

The attentive reader may have noticed that the message for test2 appears in the output before the message for test1, notwithstanding the fact that test1 precedes test2 in the test definition. This happened because this time we didn't apply the `sequential` aspect to our spec.

In the Counter example, the provided layer was acquired and released only once because it was shared between all tests: we provided it to the entire spec. But we can also opt to selectively provide a layer to just one suite, or we can share a layer between different specs defined in the same file or even between specs in different files (if you are interested in the details, please consult the ZIO Test documentation).

Using this same technique, when testing an application composed by several layers we can replace any of them with a layer developed with some specific testing purpose in mind, in addition to, naturally, testing the app with the "live" layers in place. A complete example of using custom layers for specific testing purposes, as well as the live layers themselves, can be found in Section 4, which applies ZIO Test to the REST-API implementation mentioned in Section 1.

As already said, to support property-based testing ZIO Test allows the use of data generators for standard types as well as generators for user-defined types. The general type of a generator is `Gen[R, A]`, where `R`

is the type of the resource needed by the generator and `A` is the type of the generated values. When a generator requires no resources, `R` is specified as `Any`.

There are diverse types of random generators available for all the primitive types, as well as generators that can take as source for the generated values a primitive type generator, a collection of fixed values, a ZIO effect, etc. The combination of generators using for-comprehensions is particularly useful to generate values of compound types. It is also possible to define collection generators for diverse kinds of collections, like lists, sets, maps, etc.

The following example shows generators for 2-tuples and for values of a user defined type (both using for-comprehensions), and a generator for sets of integers.

```
val tupleGen: Gen[Any, (Int, Double)] =
  for {
    a <- Gen.int
    b <- Gen.double
  } yield (a, b)

case class Person(name: String, age: Int)
val userGen: Gen[Any, Person] =
  for {
    name <- Gen.asciiString
    age <- Gen.int(18, 100)
  } yield Person(name, age)

val intSetGen : Gen[Any, Set[Int]] = Gen.setOf(Gen.int)
```

As mentioned in the previous section, ZIO Test provides a function `check` that takes as its first argument one or more generators used to generate a sample of tuples. Those tuples are used to feed the second argument of `check`, which is a function that applies to each tuple an assertion expressing the property to be verified.

In the following example, `check` is used to verify the associative property of integer addition.

```
check(Gen.int, Gen.int, Gen.int) { (x, y, z) =>
  assertTrue(((x + y) + z) == (x + (y + z)))
}
```

Property-based testing can be configured with respect to either the size of the samples or the number of times the property of interest must be verified. The default sample size can be modified using the `samples` test aspect. To specify the number of repetitions of a property verification, we can use the function `checkN` instead of `check`.

A complete example of property-based testing, applied to the REST-API implementation mentioned in Section 1, can be found in the following section.

4. Testing a REST-API implementation.

The ZIO application whose testing is discussed in this section implements a REST service, based on the GitHub REST API v3, that responds to a GET request at 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. The requests accept two parameters: 'group-level' to choose between a repository-based or an organization-based accumulation of the contributions, and 'min-contribs' to set the minimum number of contributions required for a contributor to appear explicitly in the response. The first parameter 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 pseudo-repository “All {org_name} repos” must be 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 the described REST service uses ZIO HTTP, a library for building HTTP servers and clients in a purely functional way using ZIO's functional effects, and a REDIS cache, used to speed up the response to a request for an organization whose contributors were previously requested.

As already said, the modular structure of a ZIO application is defined in terms of `ZLayer`s, which consist of 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.

Our ZIO app 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. These two functions are again implemented using ZIO HTTP.

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

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

The `RedisServerClient` layer is based on two Java libraries, EmbeddedRedis and Jedis, deliberately included to show how easily Scala allows to take advantage of the huge amount of functionality available in the Java ecosystem. The safe use of those libraries in the context of our ZIO application is very simple, thanks to the powerful and easy-to-use tools provided by ZIO to manage the life cycle of the resources needed by a functional effect, no matter if those come from ZIO itself, Scala without ZIO, or even plain old Java as is the case of our Redis services.

The main function of our application is an object that extends the `ZIOAppDefault` trait and overrides the `run` function, providing the live implementations of the layers needed. Those layers are provided to the application, as we can see in the following code:

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

Here it is possible to verify what we said previously regarding the provision of dependencies: notwithstanding the fact that the `provide` function hands over the layers as a flat structure, ZIO is able to automatically infer and manage their hierarchy.

The `RestServer` layer is the core of our application. It provides the functionality specified in the following trait:

```
trait RestServer {
  val runServer: ZIO[Any, Throwable, ExitCode]
  def contributorsByOrganization(organization: Organization, groupLevel: String, minContribs: Int):
    ZIO[zio.http.Client, Throwable, List[Contributor]]
  def groupContributors(organization: Organization,
                        groupLevel: String,
                        minContribs: Int,
                        contributorsDetailed: List[Contributor]):
    List[Contributor]
}
```

Where:

- `runServer` is the function, required by ZIO HTTP, used to map the REST endpoints to the logic that implements them.
- `contributorsByOrganization` and `groupContributors` are the functions that in our case implement that logic, and therefore are the functions that our testing efforts should focus on.

As the `RestServer` trait shows, `groupContributors` is a function that returns a non-effectual value. In fact, it is a pure function used, as it name suggests, to group by repository the contributions of a given organization. It is used as an auxiliary function by `contributorsByOrganization`, which is the effectual function that actually implements our endpoint by building, as a deferred effect, the `List[Contributor]` that will be transformed by ZIO HTTP into the desired HTTP response.

4.1 Testing a pure function.

Our first specification is implemented by the `GroupContributorsSpec` object, whose code follows. It shows the property-based testing of the pure function `groupContributors`, using data generators for user-defined data types.

```
object GroupContributorsSpec extends ZIOSpecDefault {

  val liveEnv = ZLayer.make[RestServer with zio.http.Client](
    zio.http.Client.default,
    RestServerLive.layer,
    RestClientLive.layer,
    RestServerCacheLive.layer,
    RedisServerClientTest.layer
  )

  val repoGen = Gen.elements("Repo1", "Repo2", "Repo3", "Repo4", "Repo5")
  val contributorGen = Gen.elements("Contributor1", "Contributor2", "Contributor3")
  val intGen = Gen.int(1, 999)
  val nrOfListElements = 50
  val nrOfLists = 100 // (default 200)

  val contributionGen = (repoGen <*> contributorGen <*> intGen).map(t => Contributor(t._1, t._2, t._3)
  val contributionListGen = Gen.listOfN(nrOfListElements)(contributionGen)

  def spec =

    suite("RestServer.groupContributors") (

      test("Total contributions equals expected total") {
        check(contributionListGen) { contributionList =>
          for {
            // could fail
            // resL <- RestServer.groupContributors("TestOrg", "organization", 1000, contributionList)
            // cannot fail for max number of contributions and size of list of contributions as config
            resL <- RestServer.groupContributors("TestOrg", "organization", 20000, contributionList)
            expectedTotal = contributionList.map(_.contributions).sum
          } yield assertTrue(resL.head.contributions == expectedTotal)
        }
      },
      test("Contributions by contributor are sorted in descending order") {
        check(contributionListGen) { contributionList =>
          for {
            resL <- RestServer.groupContributors("TestOrg", "organization", 1, contributionList)
            resLContribs: List[Int] = resL.map(_.contributions)
          } yield assert(resLContribs.reverse)(isSorted(Ordering[Int]))

        }
      },
      test("Contributions by the largest contributor equals expected value") {
        check(contributionListGen) { contributionList =>
          for {
            resL <- RestServer.groupContributors("TestOrg", "organization", 1, contributionList)
            largestContributor: String = resL.head.contributor
```

```
            largestContributor: String = resL.head.contributor
            expectedValue = contributionList.groupBy(_.contributor).get(largestContributor).
                            getOrElse(List[Contributor]()).map(_.contributions).sum
          } yield assertTrue(resL.head.contributions == expectedValue)
        }
      }

    ).provideShared(liveEnv) @@ TestAspect.samples(nrOfLists) @@ TestAspect.sequential

}
```

In the code shown, we can clearly see:

- The generation of a number of instances of `List[Contributor]` for some repositories and contributors, to use them as inputs to the `groupContributors` function.
- The use of the `check` function to verify the following properties of the `groupContributors` function, for each generated list of contributors:

1. The total number of contributions is correct when the parameters force the function to generate a single element for the returned `List[Contributor]`, with the contributions grouped together as if they all belonged to a single repository and a single contributor. The details of this special case of grouping can be considered as a complement provided by this test to the specification given at the beginning of Section 4.
2. The returned `List[Contributor]` is sorted in descending order by the number of contributions. This is a feature of our implementation that we can see as "specified" by this test (q.v. our previous mention of test-driven development).
3. The number of contributions from the largest contributor is equal to the expected value.

Looking at the test cases (lists of contributors as already said) we can see that their number is 100 (instead of 200, that ZIO Test takes by default) as determined by the `samples` test aspect. The maximum length of those lists is 50, as determined by the first argument of the `listOfN` function used to generate them. The lists elements are generated by applying the `Contributor` constructor to tuples of repositories, contributors and number of contributions, which in turn are generated using the `<*>` operator to combine generators for the tuple members. The tuple members are generated as elements of the standard `String` and `Int` data types using the functions shown in the code, chosen for its suitability for our purposes among the many available for the corresponding types.

This first test spec can be executed from the shell by issuing the command `sbt "testOnly GroupContributorsSpec"`, with the following result displayed by ZIO Test at the console:

```
+ RestServer.groupContributors
  + Total contributions equals expected total
  + Contributions by contributor are sorted in descending order
  + Contributions by the largest contributor equals expected value
3 tests passed. 0 tests failed. 0 tests ignored.
```

4.2 Testing an effectual function using the live layers.

Our second specification is implemented by the `ContributorsByOrganizationLiveEnvSpec` object, whose code follows. It shows the testing of the effectual function `contributorsByOrganization` using the live environment, composed by all the `ZLayer` s used by our ZIO HTTP application itself.

```
object ContributorsByOrganizationLiveEnvSpec extends ZIOSpecDefault {

  val liveEnv = ZLayer.make[RestServer with zio.http.Client](
    zio.http.Client.default,
    RestServerLive.layer,
    RestClientLive.layer,
    RestServerCacheLive.layer,
    RedisServerClientLive.layer
  )

  def spec =
    suite("RestServer.contributorsByOrganization") (

      suite("With live environment") (
        test("For organization 'xxx' fails with an ErrorType error") {
          assertZIO(
            RestServer.contributorsByOrganization("xxx", "organization", 1000).exit
          )(fails(isSubtype[ErrorType](anything)))
        },
        test("For organization 'revelation' returns List(Contributor(All revelation repos, Other contr
          for {
            respL <- RestServer.contributorsByOrganization("revelation", "organization", 10000)
            resp = respL.head
          } yield assertTrue(respL.length == 1) &&
                  assertTrue(resp.repo == "All revelation repos") &&
                  assertTrue(resp.contributor == "Other contributors")
        }
      ).provideShared(liveEnv)

    ) @@ TestAspect.sequential
}
```

As we can see by the successful execution of this specification, our endpoint:

- Responds to a request for a non-existent organization with an error that belongs to the `ErrorType` enumeration that defines the custom error model of our ZIO HTTP application. The interested reader can see the details of this error model, and its use to feed the error-channel of the functional effects involved, in the source code of this article at the GitHub repository referenced in Section 5.
- Responds properly to a request for the organization "revelation", with request parameters that guarantee the grouping of all the contributions under a single pseudo-repository ("All revelation repos") and a single pseudo-contributor ("Other contributors"). Again, the details of this special case of grouping can be considered as a complement provided by this test to the specification.

Note that the assertions for the second test of this spec stipulate the names of the single repository and the single contributor that summarize all the contributions to the organization in question, but not the total number of those contributions because the live environment gets the necessary information from GitHub online and that total may change over time with the addition of new contributions. To test that our endpoint correctly calculates the total number of contributions, we should be able to replace the live REST client layer, in charge of getting the necessary data online from GitHub, with another layer used to simulate the same functionality by obtaining the data offline from a predefined source, with a total number of contributions known in advance. And that is precisely the purpose of our next, and last, specification.

The results of executing this second test spec from the shell using sbt, as previously shown for the first spec, are the following:

```
+ RestServer.contributorsByOrganization
  + With live environment
    + For organization 'xxx' returns an empty list
    + For organization 'revelation' returns List(Contributor(All revelation repos, Other contributors,
2 tests passed. 0 tests failed. 0 tests ignored.
```

4.3 Testing an effectual function using a test-specific layer to replace a live layer.

Our last specification tests the effectual function `contributorsByOrganization`, using a layer with predefined offline data for the contributions to the "revelation" organization, instead of the live layer used in our previous specification which gets those contributions online from GitHub. In that way, we can compare the total number of contributions returned by our endpoint with the correct total calculated in advance from our own predefined data. This last spec is implemented by the `ContributorsByOrganizationTestEnvSpec` object, whose code follows.

```
object ContributorsByOrganizationTestEnvSpec extends ZIOSpecDefault {

  val testEnv = ZLayer.make[RestServer with zio.http.Client](
    zio.http.Client.default,
    RestServerLive.layer,
    RestClientTest.layer,
    RestServerCacheLive.layer,
    RedisServerClientTest.layer
  )

  def spec =
    suite("RestServer.contributorsByOrganization") (

      suite("With test environment") (
        test("Without cache, returns List(Contributor(All revelation repos, Other contributors, 20399)
          for {
            respL <- RestServer.contributorsByOrganization("revelation", "organization", 10000)
            resp = respL.head
          } yield assertTrue(respL.length == 1) &&
```

```
          } yield assertTrue(respL.length == 1) &&
                  assertTrue(resp.repo == "All revelation repos") &&
                  assertTrue(resp.contributor == "Other contributors") &&
                  assertTrue(resp.contributions == 20399)
        },
        test("With cache, also returns List(Contributor(All revelation repos, Other contributors, 2039
          for {
            respL <- RestServer.contributorsByOrganization("revelation", "organization", 10000)
            resp = respL.head
          } yield assertTrue(respL.length == 1,
                             resp.repo == "All revelation repos",
                             resp.contributor == "Other contributors",
                             resp.contributions == 20399)
        }
      ).provideShared(testEnv)

    ) @@ TestAspect.sequential
}
```

This spec is provided with a test environment `testEnv`, that differs from the live environment `liveEnv`, provided to our previous spec, by the use of a new `RestClientTest` layer instead of the original `RestClientLive` layer. Before presenting some characteristics of that new layer, we can immediately note the following:

- The assertions for the first test stipulate the names of the single pseudo-repository and the single pseudo-contributor that summarize all the contributions to the "revelation" organization, as well as the total number of those contributions, because this time that total can be calculated in advance from the sample data used by the `RestClientTest` layer.
- The second test makes exactly the same assertions as the first one, but the tested function this time doesn't take the data from the `RestClientTest` layer, but from the REDIS cache where the contributions were saved for future use the first time that the "revelation" organization was processed. Besides, it shows the use of `assertTrue` with repeated parameters, to replace a conjunction of single-parameter assertions, for even better readability.

Regarding the implementation of the `RestClientTest` layer, the following observations are in order.

Both, the live and the test layer extend the `RestClient` trait that specifies the functions used, at due time, by the `RestServer` trait to get the data necessary for building the response to a given request.

```
trait RestClient {
  def reposByOrganization(organization: Organization): ZIO[zio.http.Client, Nothing, List[Repository]]
  def contributorsByRepo(organization: Organization, repo: Repository): ZIO[zio.http.Client, Nothing,
}
```

The `reposByOrganization` and the `contributorsByRepo` functions of this trait are implemented in the live layer by processing the results of making calls to the GitHub REST API. In the test layer, instead, those functions are implemented by processing the results of reading json files containing the contributors by organization for a sample of repositories of the "revelation" organization. Those json files were built by

processing with jq, a lightweight and flexible command-line JSON processor, the responses to HTTP requests made with curl to GitHub for some repositories of that organization.

We avoid a more detailed discussion of the `RestClientTest` layer for reasons of space. The interested reader can see the layer's code, as well as the json files used as data source by the layer, at the GitHub repository referenced in Section 5.

The results of executing this last test spec from the shell using sbt are the following:

```
+ RestServer.contributorsByOrganization
 + With test environment
  + Without cache, returns List(Contributor(All revelation repos, Other contributors, 20399))
  + With cache, also returns List(Contributor(All revelation repos, Other contributors, 20399))
2 tests passed. 0 tests failed. 0 tests ignored.
```

5. Conclusion. Recommendations and references.

We conclude with a handful of briefly stated recommendations that we hope can be considered well-supported by the discussion and examples presented in the previous sections.

Use ZIO Test for testing ZIO applications, taking advantage from tests as first class values, simple and powerful assertions, advanced property-based testing, special layers for specific testing purposes, and useful test aspects. Investigate other features of ZIO Test, like, just to mention an example, special techniques for testing concurrent programs (a nice presentation of this subject can be found in the second reference below).

Make good use of the highly customizable error handling functionality provided by ZIO. Although this is a recommendation more related to development than testing, it is presented here due to its close relationship with the topic of errors. You can see it applied to our case, as said before, by examining the source code.

As a final conclusion, the answer to the question posed at the beginning of this article would be: we cannot avoid errors, but we can get rid of them with the help of ZIO and ZIO Test together. The power and simplicity of the functional programming paradigm are essential for achieving this difficult goal.

References:

- A slightly outdated presentation of ZIO Test by Adam Fraser, where most inspiration for this article was found, is still worth watching here.
- Jorge Vasquez' article about STM, which contains an example of testing a concurrent program using multiple threads, can be found here. The text is also a little outdated, but the Scala code has been updated to ZIO 2.
- The first part of a ZIO HTTP tutorial by Jakub Czuchnowski, presents nice examples of error handling using the ZIO error chanel with types defined in a custom error model.
- The official documentation of ZIO Test.

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