How to use ZLayers in a ZIO 2.0 Application

By Ignacio Gallego Sagastume
Hivemind Technologies (https://hivemindtechnologies.com/)

October, 2023

Abstract

The *ZIO* framework is frequently used to build Scala applications. In the newest version of the framework, *ZIO 2.0*, there are some modifications introduced to improve the usability, simplicity, readability and practicality of the applications that you can build with it.
In this article, we will explain and define what *ZIO Layers (ZLayers)* are, and we will explore the different mechanisms that you can use to construct and use them. For that matter, we will provide a fully functional ZIO application, with a layered architecture.

Applying the "Service Pattern" **[2]** for Dependency Injection (DI) **[3]**, and using *ZLayers* will help us to organise and structure the application for better reuse and maintainability. Also, we will take advantage of the different compilation errors that you get when you do not provide the correct dependencies on the different application components.

Introduction

The *ZIO* framework has gained popularity and attention recently **[5]**, because of the benefits that *Functional Programming* (**FP**, see **[4])** brings to the table, and improved performance compared to other side-effect solutions. If you are new to *ZIO*, you can start with the *ZIO online quickstart guides and tutorials* on **[6]**, and also with article series on **[7]**.

However, as the *ZIO framework* is being actively improved and developed, sometimes the syntax of function definitions provided by ZIO changes, making some articles and code examples found in Github repositories no longer valid with the new versions of the framework. For example, from version *ZIO 1.0* to *ZIO 2.0*, there are some breaking changes regarding syntax of *Environment* and *ZLayers*. This article will provide a full example of a layered application, with a fixed in-memory database layer, a repository layer and a service layer, together with a set of tests for each layer.

Let's start defining and introducing *ZIO* and the concept of *ZLayers* and how this can help in our Scala **[8]** applications.

ZIO Effects

Let's first start with ZIO effects.

As you might know, a pure *FP* application cannot have side effects, because mathematical functions are total and deterministic, and always return the same result when provided the same input. This would make impossible to write in a console for example as a statement, as we normally would do in an imperative language like Java for example:

```
public static void main(String[] args) {
   System.out.println(“Hello world !”);
}
```

Or equivalently in Scala:

```
final def main(args: Array[String]): Unit = {
  println(“Hello world !”)
}
```

Using pure FP, we would have to redefine the main entry point of the application to run a side effect for writing in the console. Using ZIO, we would have to replace the return type (*Void* or *Unit*) to a type indicating that the function will do a side effect when it is run:

```
object Main extends ZIOAppDefault {
 override def run: ZIO[Any, IOException, Unit] =
   for {
     _ <- Console.printLine("Hello world !")
   } yield ()
}
```

The definition of the ZIO datatype:

```
ZIO[-R, +E, +A]
```

specifies an environment ***R***, an error channel ***E*** and a success return type ***A***.

In this last example, we are redefining the entry point of the application (the main method) to return a *ZIO* effect that has **no requirements for the environment (*Any*** type**)**, **can fail with** an *IOException* and returns ***Unit*** as its **success value**.

The main change is the way of thinking, now we are returning a *ZIO value* (the unit value in this case) instead of a side-effectful computation of the first example (***println***). Now the ***run()*** method function is a pure function, in the mathematical sense.

A practical mental model to think about the ZIO data type, that describes a side-effectful computation is:

```
R => Either[E, A]
```

That is, a function that, given the environment ***R***, returns either an error ***E*** or a success value ***A***.

For a more detailed explanation, see the references section.

What are ZLayers and why should we use them?

In the scaladoc documentation in the *ZLayer* type, we can find a good starting point:

*"A **ZLayer[E, A, B]** describes how to build one or more services in your application. Services can be injected into effects via **ZIO.provide**. Effects can require services via **ZIO.service**. **Layer** can be thought of as recipes for producing bundles of services, given their dependencies (other services).*
Construction of services can be effectful and utilize resources that must be acquired and safely released when the services are done being utilized.
By default layers are shared, meaning that if the same layer is used twice the layer will only be allocated a single time.
Because of their excellent composition properties, layers are the idiomatic way in ZIO to create services that depend on other services.".

Let's try to explain this concept. The *ZLayer* data type specifies three type parameters like a ZIO effect:

```
class ZLayer[-RIn, +E, +ROut]
```

- First, the ***RIn*** type parameter is the data type for the value required to define the *ZLayer* (its dependencies). Is like the environment for the *ZLayer*. *ZLayers* can depend on other *ZLayers.*
- Then, the ***E*** type parameter is the error channel. Specifies the error type that can occur when allocating resources or running computations needed to define the *ZLayer*.
- Finally, the ***ROut*** type parameter specifies the success channel, similar to a *ZIO* effect. This type specifies the service or the value to be generated by the *ZLayer*, when provided the needed dependencies (that are specified by the first parameter).

The same mental model used for the *ZIO* data type can be used here:

```
RIn => Either[E, ROut]
```

Given the requirements ***RIn***, the function will return either an error ***E*** or a success value ***ROut***.

Let's see an example:

```
case class Config(databaseParameters: DatabaseParameters)

object Config {
  private val myDBParams: DatabaseParameters = ...

  val live: ULayer[Config]                   =
    ZLayer.succeed(
      Config(databaseParameters = myDBParams),
    )
}
```

Imagine that your application has a configuration class ***Config***, that specifies the databaseParameters (like maximum number of open connections, database username and password, etc.).

In the companion object for class ***Config***, you define a value for those parameters. Then, as you have all the requirements for defining a ***Config*** value, you can define a *ZLayer* (using the service pattern **[2]**) by calling ***ZLayer.succeed()***. This method will **always** return a *ZLayer* (**it cannot fail**).

The data type:

```
ULayer[Config]
```

Is equivalent to:

```
ZLayer[Any, Nothing, Config]
```

That is, a *ZLayer* that has **no requirements (*Any*** in the requirements), and that will always return a ***Config*** success value. Also, the computation **cannot fail**, as there is a ***Nothing*** in the error channel (this is because there does not exist an instance of the type ***Nothing*** that we could possibly *return* or *throw*).

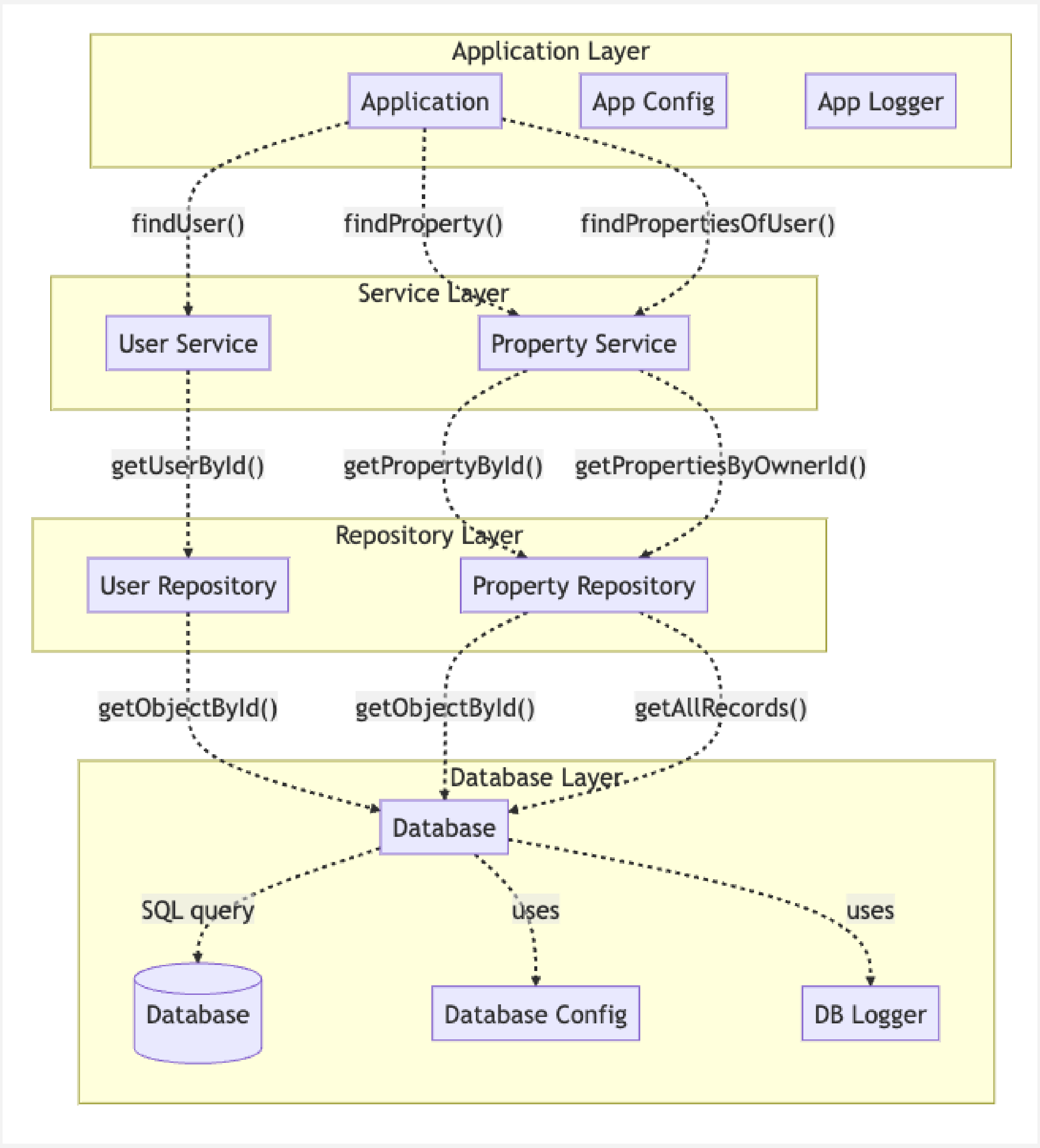
That's it! We have specified our first service in a *ZIO Layer*.

The *ZIO ZLayer* data type lets you specify the dependencies of your ZIO application explicitly in a type signature, and, as we will see later, *ZIO* will provide compile-time errors that will help you to build all the application tree for production or test classes in a safe way. However, there are other mechanisms and frameworks that address this same *DI* problem **[3]** in a good way. So, it's up to you which framework you choose to build and run your *ZIO* application.

The Architecture of the Sample Application

The same application is a classic example of two different repositories or collections of elements like *users or clients* that can have zero, one or more *properties* (that is, taxable objects, like houses, cars or boats). For some operations the application will use only one of these two collections, while in other cases we have to collect related elements of both collections. The first case is for example when we have to query one *user*, implemented by the operation ***findUser(userId: Int)***. For the second scenario, we have for example the operation ***findPropertiesOfUser(userId: Int)***, for which we have to retrieve first the *user* with the specified ***userId***, and also all their related *properties*.
The application is divided into four different layers of logically related components: the application, service, repository, and database layers.

The following architecture diagram illustrates the main components of the application:



Application Layer

The ***Application Layer*** is concerned about the main logic of the whole system. It uses the different components to react to the user requirements and combine the results of the different services and data.

Service Layer

The ***Service Layer*** defines the different related components to implement the user needs, like the ***User Service*** and the ***Property Service***. The ***User Service*** will define methods for reading, updating and deleting ***Users*** (models) at the highest level of abstraction. In a similar way, the ***Property Service*** will group the operations related to *property models* that a *person or user* can possess (houses, cars, boats, etc.).

Repository Layer

The ***Repository Layer*** defines collections of items, in this case ***Users*** and ***Properties*** of those users. The operations care about collecting and filtering items in a lower level of abstraction.

Database Layer

The ***Database Layer*** language is concerned with *Records*, *Keys*, *Queries*, etc. in the lowest level of abstraction. The *database* can trigger and perform queries in the database store, that could be a *relational* database (like Oracle, PostgreSQL, MySQL) or *NoSQL* database (like MongoDB, MariaDB or Cassandra) or even one or more system files.

The Scala code base for the sample application can be found at **[9]**.

Dependencies between components or services

From the previous architecture diagram you can see that some components *depend* or *delegate responsibility* on others in order to function properly.

For example, let's start with the functional logger trait defined in the sample application:

```
trait Logger {
 def log(message: String): UIO[Unit]
}
```

The ***Logger*** trait specifies an abstract ***log*** method that prints a message in the console as a *ZIO* side effect.

This method should return a value of the type ***UIO[Unit]***. *ZIO* defines the following type alias:

```
type UIO[+A]       = ZIO[Any, Nothing, A]
```

This kind of side effect **does not require anything** from the environment (***Any*** in the first *ZIO* type channel), **can never fail (*Nothing*** in the error type channel**)**, and always succeeds with an ***A,*** bound to the ***Unit*** type in this case.

The implementation of the ***Logger*** trait, ***LoggerImpl***, *needs* or *depends on* a ***Console*** object in order to print the message in it:

```
case class LoggerImpl(console: Console) extends Logger {
 override def log(message: String): UIO[Unit] =
   console.printLine(s"INFO: $message").ignore
}
```

We can specify this *dependency* in the ***LoggerImpl*** constructor, as we can see in this last piece of code. To implement the ***log()*** method, we use the ***printLine()*** method of the ***zio.Console*** type. Please note the ".ignore" at the end of the line, that will ignore the ***IOException*** errors that can occur when printing a string into the console.

Using the *Service Pattern* **[2]**, we can provide a default implementation for the ***Logger*** trait into the ***Logger*** companion object, in the ***live*** value (you can use a ***lazy val*** if you prefer). This will define a new ***ZLayer*** instance:

```
object Logger {
 val live: URLayer[Console, Logger]= ZLayer {
   for {
     console <- ZIO.service[Console]
     impl    <- ZIO.succeed(LoggerImpl(console))
   } yield impl
 }
}
```

In this ***live*** value we specify a *ZLayer* that requires only a ***Console*** instance in order to build and return a ***Logger*** instance. The ***URLayer*** is another type alias:

```
type URLayer[-RIn, +ROut] = ZLayer[RIn, Nothing, ROut]
```

As you can see, there is a ***Nothing*** type in the error channel, so the construction of this ZLayer **can never fail**.

We are creating a *ZLayer* instance with the ***ZLayer.apply()*** method, that is a synonym of the ***ZLayer.fromZIO()*** method, that will take a *ZIO* instance value as parameter.

In the next section we are going to explain the line that takes the ***Console*** instance from the *Environment* of the *ZLayer*.

Later in this article, we will modify this ***Logger.live()*** method to use another *ZLayer* creation method that will allocate resources before returning the created ***LoggerImpl*** instance.

Accessing the Environment

In this section, we are going to examine the creation of a ZLayer that defines two different *dependencies* in the requirements of the *ZLayer*. As in the previous example, we are going to access the environment in order to build the success value of the *ZLayer*.

Let’s take the example of the ***Database*** trait:

```
trait Database {
 def getObjectById(id: Int, table: TableName): IO[DatabaseException,
Option[Record]]
 def getAllRecords(table: TableName): IO[DatabaseException, List[Record]]
}
```

We have the following implementation, that requires the dependencies in the constructor:

```
class DatabaseImpl(parameters: DatabaseParameters, logger: Logger) extends
Database
```

We are specifying here that this ***DatabaseImpl*** implementation requires some database configuration parameters and a ***Logger*** instance to work or function properly. This implementation will call some methods (or *delegate behaviour*) in those instances.

Again, using the *Service Pattern* **[2]**, we can create a *ZLayer* for the ***Database*** trait:

```
object Database {
 val live: URLayer[Logger with Config, Database] = ZLayer {
   for {
     logger       <- ZIO.service[Logger]
     config       <- ZIO.service[Config]
     databaseImpl <- ZIO.succeed(DatabaseImpl(config.databaseParameters,
logger))
   } yield databaseImpl
 }
}
```

In the type signature of the ***live*** value, we can see that we are requiring two things: a ***Logger*** instance and a ***Config*** instance. We can compose different type requirements using the keyword *“**with**”* as many times as needed for the dependencies we have. For example, if we require three values of types **A**, **B** and **C** respectively, we can specify ***A with B with C*** in the environment channel of the *ZLayer*. Also, we are returning, as a success value, a ***Database*** instance.

With the first line in the for comprehension, using ***ZIO.service[Logger]***, we are *accessing the environment* and extracting the already provided instance of the ***Logger*** class. Analogously with the following line, but accessing the ***Config*** instance this time.

Finally, we create and return a new ***Database*** implementation instance using the calculated values of previous lines (the parameters and the database errors logger).

Different ways of creating ZLayers

Up to now, we have seen 2 ways of creating a *ZLayer* instance. Let’s sum up and add more ways of creating *ZLayers* in this section.

From a simple value

We can create a *ZLayer* from a simple object value, using for example a ***Config*** instance:

```
val testConfigZLayer: ULayer[Config] = ZLayer.succeed(testConfig)
```

This way, the construction of the *ZLayer* will **not require any dependency,** will **never fail**, and **always succeed** with a ***Config*** instance, as we can see in the type signature.

From a ZIO value

You can create a *ZLayer* from a *ZIO* value (or even a more complex *ZIO* computation), using the ***ZLayer.apply()*** or the ***ZLayer.fromZIO()*** methods, that are equivalent.

For example, in the same way as we did for the ***Database*** trait, we can do for the ***UserRepository*** trait, that requires a ***Logger*** and a ***Database*** as dependencies:

```
object UserRepository {
 val live: URLayer[Logger with Database, UserRepository] =
   ZLayer { // apply == fromZIO
     for {
       logger   <- ZIO.service[Logger]
       database <- ZIO.service[Database]
     } yield UserRepositoryImpl(database, logger)
   }
}
```

Now, we can add two more ways in the following subsections.

From a function

You can create a ZLayer from a (*non-ZIO)* function that take the *ZLayer requirements* as parameters, using the ***ZLayer.fromFunction()*** construct:

```
def liveFromFunction: URLayer[Console, Logger] =
 ZLayer.fromFunction((console: Console) => LoggerImpl(console))
```

Here we are providing a simple lambda function (*anonymous function expression*), but we could use a ***def*** definition to provide a more complex function logic.

Creating a *ZLayer* that allocates resources

Sometimes we need to allocate resources and release them when constructing a *ZLayer* instance. For this use case, we are going to provide another ***Logger*** implementation that allocates resources.

Let's suppose that we need to count the number of lines that the logger prints in the console, and we need to print this number before we destroy the allocated resources and the ***Logger*** instance. We can then write the following ***Logger*** implementation:

```scala
case class LoggerWithLineCounter(console: Console, counter: Ref[Int]) extends
Logger {

 def initializer: UIO[Unit] =
   console.printLine("Initializing logger instance ...").ignore

 override def log(message: String): UIO[Unit] =
   for {
     _ <- counter.update(_ + 1)
     _ <- console.printLine(s"INFO: $message").ignore
   } yield ()

 def finalizer: UIO[Unit] =
   for {
     lines <- counter.get
     _       <- console.printLine(s"Total printed lines: $lines").ignore
   } yield ()
}
```

To use this implementation we can reimplement the ***live*** value, but this time using the ***ZLayer.scoped()*** construct:

```scala
object Logger {
 val liveWithLineCounter: URLayer[Console, Logger] = ZLayer.scoped {
   for {
     ref     <- Ref.make(0)
     console <- ZIO.service[Console]
     impl    <- ZIO.succeed(LoggerWithLineCounter(console, ref))
     _       <- impl.initializer
     _       <- ZIO.addFinalizer(impl.finalizer)
   } yield impl
 }
}
```

Here, we define a series of steps:

1. First, we create a ***zio.Ref*** value initialised to *0*.
2. After that, we get the console instance from the environment
3. We create the ***Logger*** instance (implemented by the ***LoggerWithLineCounter*** class)
4. Call the initializer
5. And lastly, we add a finalizer that will be called when the instance is destroyed and the allocated resources are freed.

This is a very simple example, but demonstrates a pattern that we can use for more complex use cases.

Providing the layers to build a ZIO effect

We have seen so far how to define layers based on a set of requirements, and how to access the environment or requirements of a *ZLayer* for instantiating and working with those services.

Let's focus now on how to use all the defined layers and integrate them to create a *ZIO* effect that can be run as a test or as an entire application.

Suppose that we have the following application ***Config*** object value:

```
val testConfig: Config                  = Config.testConfig(...)
```

Then, we can create a *ZLayer* for the ***Config*** dependency, using ***ZLayer.succeed()***:

```
val testConfigZLayer: ULayer[Config] = ZLayer.succeed(testConfig)
```

We can create also a *ZLayer* for the console dependency, using the value ***Console.ConsoleLive*** from the ***zio*** package:

```
val consoleZLayer: ULayer[Console]= ZLayer.succeed(Console.ConsoleLive)
```

Now, using the ***ZIO.service[T]*** method, we can create a *ZIO* effect that, when provided a ***Database***, will return a ***Database***:

```
val dbTestURIO: URIO[Database, Database] = ZIO.service[Database]
```

Finally, the ***ZIO.provide()*** method, we can transform the last effect ***dbTestURIO*** (that required a context) to a *ZIO* effect that does not require any context (note the change in the resulting type, from ***URIO*** to ***UIO***):

```
val databaseUIO: UIO[Database] = dbTestURIO.provide(consoleZLayer,
testConfigZLayer, Logger.live, Database.live)
}
```

Here, using the ***live*** method (or values) defined before, we are providing *all the requirements* that are needed in order to create the ***Database*** dependency:

1. The ***Console*** layer (used by the ***Logger*** layer requirement)
2. The ***Config*** layer (for the unit test in this case)
3. The ***Logger*** *ZLayer*
4. And lastly, the ***Database*** layer

Now, we are ready to use this ***databaseUIO*** *ZIO* effect in any test or application. For example, we can define a ***ZIO Test*** like this:

```
val test1: Spec[Any, DatabaseException] = test("returns Alonzo Church when
getObjectById is executed") {
 for {
   db     <- databaseUIO
   record <- db.getObjectById(1, TableName.Users)
 } yield assertAlonzoChurch(record)
}

private def assertAlonzoChurch(value: Option[Record]): TestResult =
 assert(value)(isSome(equalTo(DatabaseImpl.alonzoChurch)))
```

Similarly, we can create *ZIO* effects for other components of the application, like the ***UserService*** or the ***PropertyService*** for example. See the provided Github code repository for more details.

Provide all except one dependency

In some cases, for example when working with *Scopes*, you can provide some of the layers, leaving one dependency still to be provided. For example, we can leave out the ***Scope*** dependency as follows:

```
dbTestURIO.provideSome[Scope](consoleZLayer, testConfigZLayer,
Logger.liveWithLineCounter, Database.live)
```

In this case, all the dependencies except for the ***Scope*** have been provided.

Useful compilation errors and diagram generation

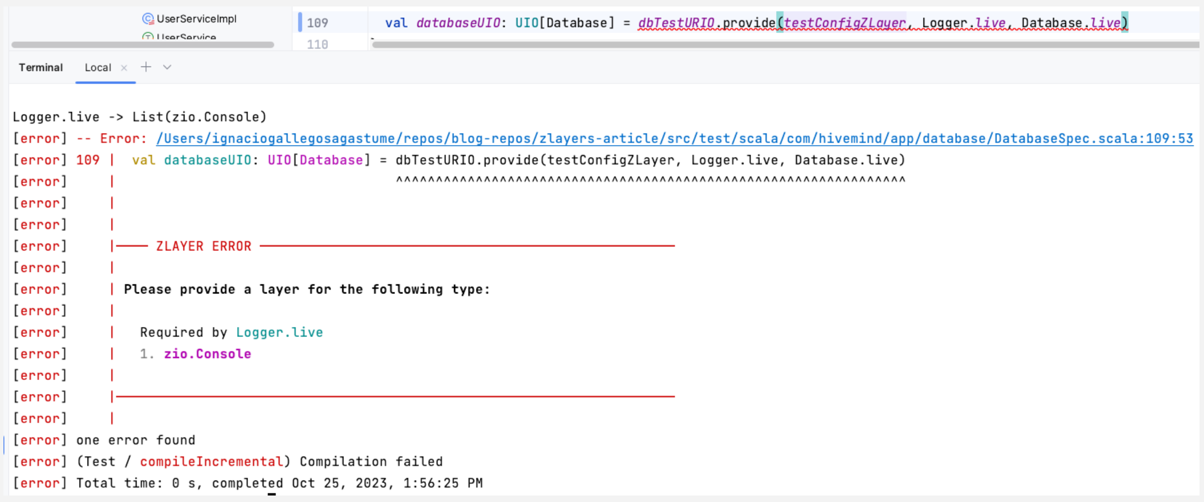
In very complex cases, the *ZIO 2.0* framework helps us to build the application dependency graph with some helpful compilation time error messages. Let's examine some use cases using the Database and PropertyService components in the sample application (see reference **[9]**).

Not providing enough layers

Let's take the last definition of the *ZIO* effect ***databaseUIO***, but provide less arguments in the ***ZIO.provide()*** function, for example, removing the console layer:

```
dbTestURIO.provide(testConfigZLayer, Logger.live, Database.live)
```

The *sbt* compiler then shows the following compilation error in line *109*:

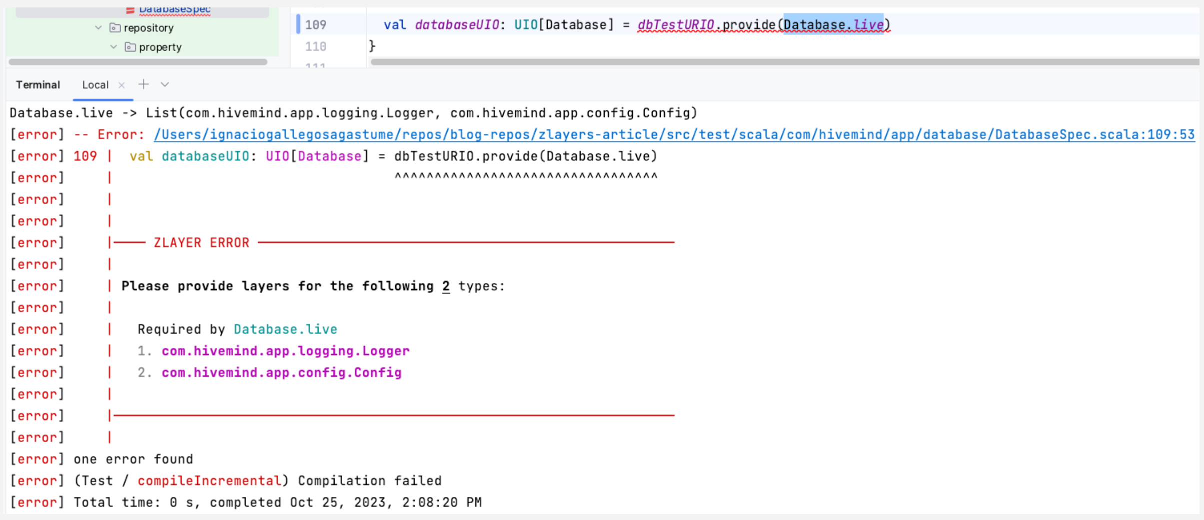


```
val databaseUIO: UIO[Database] = dbTestURIO.provide(testConfigZLayer, Logger.live, Database.live)
```

```
Logger.live -> List(zio.Console)
[error] -- Error: /Users/ignaciogallegosagastume/repos/blog-repos/zlayers-article/src/test/scala/com/hivemind/app/database/DatabaseSpec.scala:109:53
[error] 109 |  val databaseUIO: UIO[Database] = dbTestURIO.provide(testConfigZLayer, Logger.live, Database.live)
[error]     |                                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[error]     |
[error]     |
[error]     |──── ZLAYER ERROR ────────────────────────────────────────────────────
[error]     |
[error]     | Please provide a layer for the following type:
[error]     |
[error]     |   Required by Logger.live
[error]     |   1. zio.Console
[error]     |
[error]     |──────────────────────────────────────────────────────────────────────
[error]     |
[error] one error found
[error] (Test / compileIncremental) Compilation failed
[error] Total time: 0 s, completed Oct 25, 2023, 1:56:25 PM
```

The error shows that for the ***Logger*** dependency, we need a ***zio.Console***.

Now, if we remove also the test and logger layers, leaving only the ***Database*** *ZLayer*:



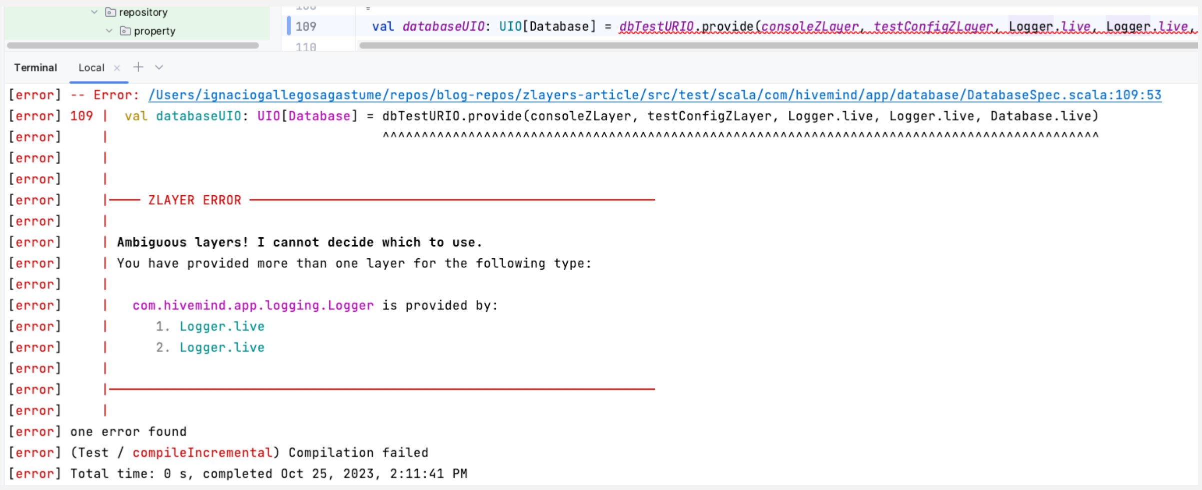
```
val databaseUIO: UIO[Database] = dbTestURIO.provide(Database.live)
}
```

```
Database.live -> List(com.hivemind.app.logging.Logger, com.hivemind.app.config.Config)
[error] -- Error: /Users/ignaciogallegosagastume/repos/blog-repos/zlayers-article/src/test/scala/com/hivemind/app/database/DatabaseSpec.scala:109:53
[error] 109 |  val databaseUIO: UIO[Database] = dbTestURIO.provide(Database.live)
[error]     |                                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[error]     |
[error]     |
[error]     |──── ZLAYER ERROR ────────────────────────────────────────────────────
[error]     |
[error]     | Please provide layers for the following 2 types:
[error]     |
[error]     |   Required by Database.live
[error]     |   1. com.hivemind.app.logging.Logger
[error]     |   2. com.hivemind.app.config.Config
[error]     |
[error]     |──────────────────────────────────────────────────────────────────────
[error]     |
[error] one error found
[error] (Test / compileIncremental) Compilation failed
[error] Total time: 0 s, completed Oct 25, 2023, 2:08:20 PM
```

As you can see in the picture, to satisfy the ***Database*** requirements, we need a ***Logger*** and a ***Config*** instance.

Providing more layers than needed

Also, if we provide more than it is needed (for example two ***Logger*** instances):



```
repository
property
val databaseUIO: UIO[Database] = dbTestURIO.provide(consoleZLayer, testConfigZLayer, Logger.live, Logger.live,
```

Terminal Local

```
[error] -- Error: /Users/ignaciogallegosagastume/repos/blog-repos/zlayers-article/src/test/scala/com/hivemind/app/database/DatabaseSpec.scala:109:53
[error] 109 |  val databaseUIO: UIO[Database] = dbTestURIO.provide(consoleZLayer, testConfigZLayer, Logger.live, Logger.live, Database.live)
[error]     |                                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[error]     |
[error]     |
[error]     |─── ZLAYER ERROR ──────────────────────────────────────────────
[error]     |
[error]     | Ambiguous layers! I cannot decide which to use.
[error]     | You have provided more than one layer for the following type:
[error]     |
[error]     |   com.hivemind.app.logging.Logger is provided by:
[error]     |      1. Logger.live
[error]     |      2. Logger.live
[error]     |
[error]     |──────────────────────────────────────────────────────────────
[error]     |
[error] one error found
[error] (Test / compileIncremental) Compilation failed
[error] Total time: 0 s, completed Oct 25, 2023, 2:11:41 PM
```

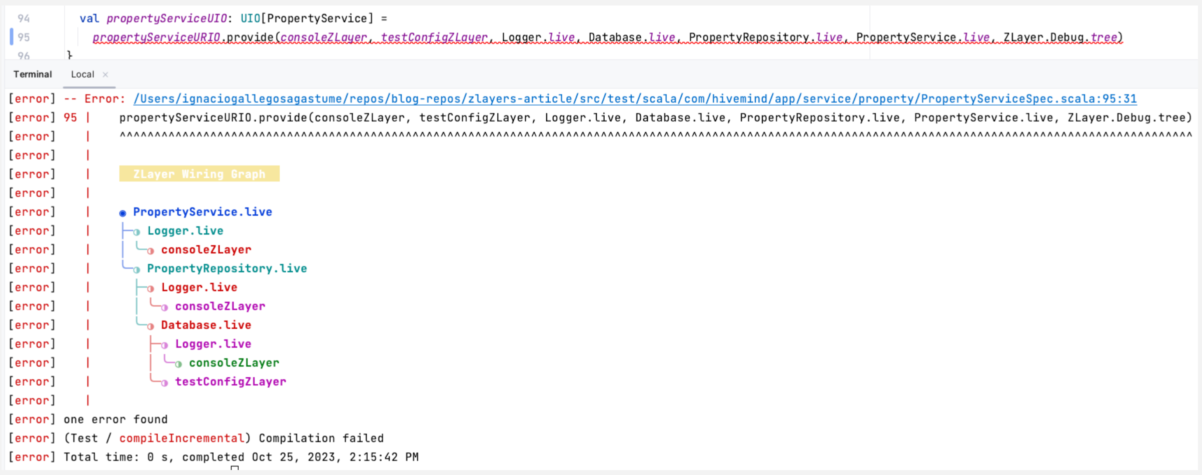
The error is very helpful and indicates the exact problem.

Debugging the dependency graph

If you need to check and debug the dependency graph, you have two options:

1. Using ***ZLayer.Debug.tree*** as dependency in the ***provide*** function
2. Using ***ZLayer.Debug.mermaid*** as dependency in the ***provide*** function

In the first option, there will be a compile time error in the console that will show the dependency graph with nice colours for you to check:

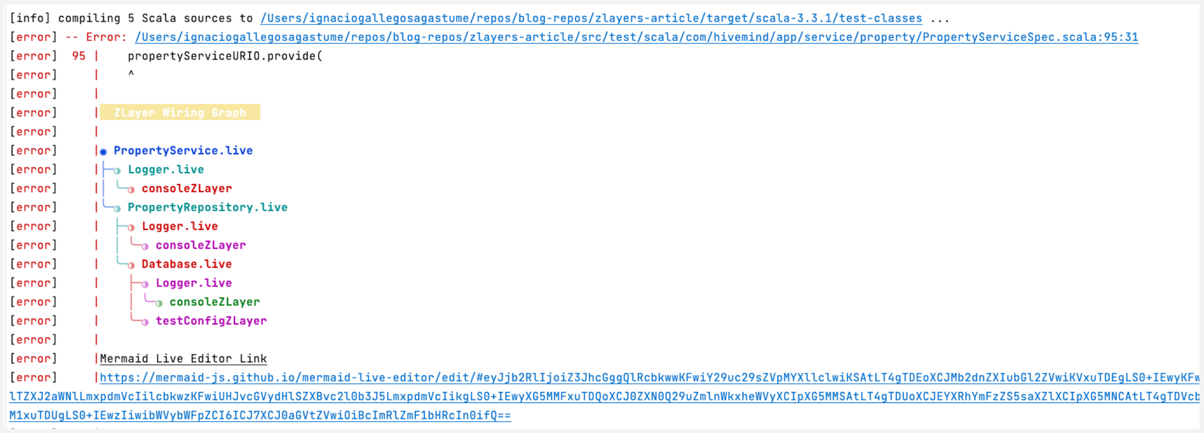


```
val propertyServiceUIO: UIO[PropertyService] =
  propertyServiceURIO.provide(consoleZLayer, testConfigZLayer, Logger.live, Database.live, PropertyRepository.live, PropertyService.live, ZLayer.Debug.tree)
}
```

Terminal Local

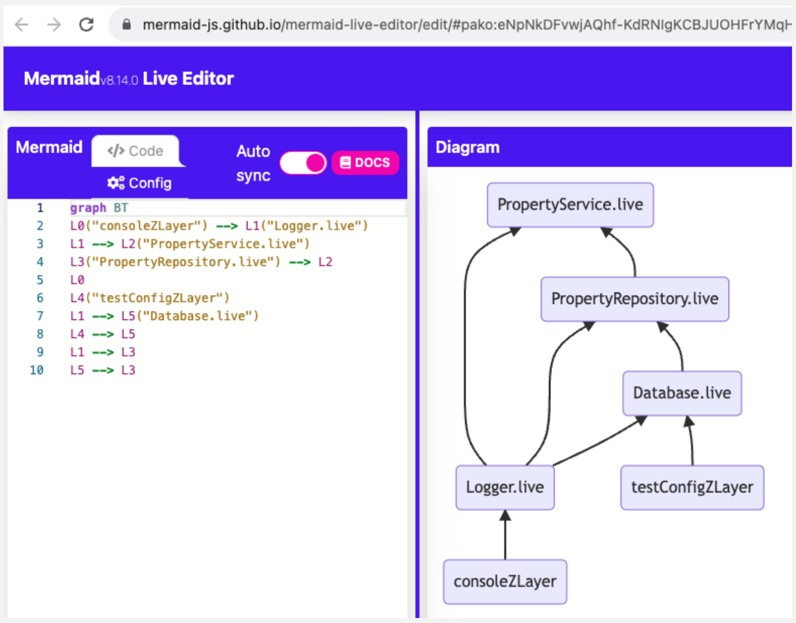
```
[error] -- Error: /Users/ignaciogallegosagastume/repos/blog-repos/zlayers-article/src/test/scala/com/hivemind/app/service/property/PropertyServiceSpec.scala:95:31
[error] 95 |    propertyServiceURIO.provide(consoleZLayer, testConfigZLayer, Logger.live, Database.live, PropertyRepository.live, PropertyService.live, ZLayer.Debug.tree)
[error]    |    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[error]    |
[error]    |     ZLayer Wiring Graph
[error]    |
[error]    |    ◉ PropertyService.live
[error]    |    ├─◑ Logger.live
[error]    |    │ ╰─◑ consoleZLayer
[error]    |    ╰─◑ PropertyRepository.live
[error]    |      ├─◑ Logger.live
[error]    |      │ ╰─◑ consoleZLayer
[error]    |      ╰─◑ Database.live
[error]    |        ├─◑ Logger.live
[error]    |        │ ╰─◑ consoleZLayer
[error]    |        ╰─◑ testConfigZLayer
[error]    |
[error] one error found
[error] (Test / compileIncremental) Compilation failed
[error] Total time: 0 s, completed Oct 25, 2023, 2:15:42 PM
```

Using the second option, you will get a compile time error with the graph and a link to a mermaid diagram with all the dependencies:



```
[info] compiling 5 Scala sources to /Users/ignaciogallegosagastume/repos/blog-repos/zlayers-article/target/scala-3.3.1/test-classes ...
[error] -- Error: /Users/ignaciogallegosagastume/repos/blog-repos/zlayers-article/src/test/scala/com/hivemind/app/service/property/PropertyServiceSpec.scala:95:31
[error]  95 |    propertyServiceURIO.provide(
[error]     |    ^
[error]     |
[error]     | ZLayer Wiring Graph 
[error]     |
[error]     |◉ PropertyService.live
[error]     |├─◑ Logger.live
[error]     |│ ╰─◑ consoleZLayer
[error]     |╰─◑ PropertyRepository.live
[error]     |  ├─◑ Logger.live
[error]     |  │ ╰─◑ consoleZLayer
[error]     |  ╰─◑ Database.live
[error]     |    ├─◑ Logger.live
[error]     |    │ ╰─◑ consoleZLayer
[error]     |    ╰─◑ testConfigZLayer
[error]     |
[error]     |Mermaid Live Editor Link
[error]     |https://mermaid-js.github.io/mermaid-live-editor/edit/#eyJjb2RlIjoiZ3JhcGggQlRcbkwwKFwiY29uc29sZVpMYXllclwiKSAtLT4gTDEoXCJMb2dnZXIubGl2ZVwiKVxuTDEgLS0+IEwyKFw
lTZXJ2aWNlLmxpdmVcIilcbkwzKFwiUHJvcGVydHlSZXBvc2l0b3J5LmxpdmVcIikgLS0+IEwyXG5MMFxuTDQoXCJ0ZXN0Q29uZmlnWkxheWVyXCIpXG5MMSAtLT4gTDUoXCJEYXRhYmFzZS5saXZlXCIpXG5MNCAtLT4gTDVcb
M1xuTDUgLS0+IEwzIiwibWVybWFpZCI6ICJ7XCJ0aGVtZVwiOiBcImRlZmF1bHRcIn0ifQ==
```

If you follow the link, it will open the browser on the ***mermaid-js.github.io*** website, with a diagram like the following:



```
graph BT
L0("consoleZLayer") --> L1("Logger.live")
L1 --> L2("PropertyService.live")
L3("PropertyRepository.live") --> L2
L0
L4("testConfigZLayer")
L1 --> L5("Database.live")
L4 --> L5
L1 --> L3
L5 --> L3
```

This nice feature hopefully will help you to understand and debug some dependency problems that you may encounter while developing your application.

Also, it is a good starting point to add some documentation about the architecture of your application, that you can include in the ***README.md*** (*markdown)*, using the following syntax:

~~~
## Architecture of the Application

```mermaid
<!-- Your diagram here -->
```
~~~

For more information about mermaid syntax, check the following website:
https://mermaid.js.org/syntax/flowchart.html

Putting it all together

In this section, we are going to show a final example of a *ZIO* 2.0 application that makes use of the service layer to query users and properties, and print them in the console.

We can start by creating an *object* that extends the ***ZIOAppDefault*** trait:

```
object Main extends ZIOAppDefault {
```

Then we override the ***run()*** method of ***ZIOAppDefault***, and *provide all* the required dependencies for the application to work. This will create a *ZIO* effect that does not require a particular context, can fail with ***IOException*** and finishes with a *Unit* value:

```
 override def run: ZIO[Any, IOException, Unit] =
   myAppLogic.provide(
     Logger.live,
     ZLayer.succeed(zio.Console.ConsoleLive),
     PropertyService.live,
     PropertyRepository.live,
     UserService.live,
     UserRepository.live,
     Database.live,
     Config.live,
     // ZLayer.Debug.mermaid // (uncomment to show a dependency graph in a
diagram)
     // ZLayer.Debug.tree    // (uncomment to show a dependency graph in
console)
   )
```

One thing to notice is that the ***Config.live*** method will return a *ZLayer* with a default configuration of the database parameters. These default parameters indicate that the implementation must *not* fail and will always return valid records from the database.

Also, we use a value ***myAppLogic*** that we haven't defined yet. Let's define that in the following piece of code:

```
val myAppLogic: ZIO[Logger with UserService with PropertyService,
IOException, Unit] =
 for {
   logging         <- ZIO.service[Logger]
   userService     <- ZIO.service[UserService]
   propertyService <- ZIO.service[PropertyService]
   _               <- logging.log("Starting application ...",
HivemindLogLevel.INFO)
   _               <- logging.log("Looking for user with id=1")
   alonzo          <- userService
                        .findUser(1)
                        .mapError(err => IOException(s"UserService failed
...", err))
                        .flatMap(ZIO.fromOption)
                        .orElseFail(IOException(s"Did not find user with id
1"))
   _                  <- logging.log(s"Found user '${alonzo.name}
${alonzo.surname}'")
   _                  <- logging.log(s"Looking for ${alonzo.name}'s properties")
   properties         <- propertyService
                        .findPropertiesOfUser(alonzo.id)
                        .mapError(err => IOException(s"PropertyService ...",
err))
   propertiesAsStr   = properties.map(property => s"{ id: ${property.id},
kind: ${property.kind}, price: €${property.price} }").mkString("\n")
   _                  <- logging.log(s"User ${alonzo.name} has the following
properties:\n$propertiesAsStr")
   _                  <- logging.log("Ending application")
 } yield ()
```

Note the *ZIO* datatype of ***myAppLogic*** in the first line: we are requiring a ***Logger*** and the two services to work. Then, in the first three lines of the *for-comprehension* we are extracting those dependencies and creating references to use them.

In this example, we are searching for the user with id *1*, handling the errors, and printing the result in the console. Then, we are using the ***PropertyService*** to find all the properties of that user, and also printing them in the console.

If we run this *non-failing* program, it will produce the following output:

```
INFO: Starting application ...
INFO: Looking for user with id=1
INFO: Query finished successfully.
INFO: Found user 'Alonzo Church'
INFO: Looking for Alonzo's properties
INFO: Query finished successfully.
INFO: Query finished successfully.
INFO: User Alonzo has the following properties:
{ id: 1, kind: Car, price: €17800 }
{ id: 2, kind: House, price: €230500 }
{ id: 3, kind: Boat, price: €180000 }
INFO: Ending application

Process finished with exit code 0
```

We can see a query success output for looking at a user, then two more queries to find the properties of a user. This is because the implementation of ***findPropertiesOfUser()*** method first searches for the user, and then for the properties, accessing the database two times. See the implementation for more details in **[9]**.

Testing error handling

If you want to see errors printed in the console, try to change the ***Config.live()*** dependency in the ***run()*** method to return a config with database parameters with a different ***outcome*** value. For example, in:

```
case class DatabaseParameters(
 databaseName: String,
 databasePassword: String,
 maxConnections: Int,
 outcome: DatabaseLayerExecutionOutcome,
)
```

You can choose different values for the ***outcome***, as the *enum* type indicates:

```
enum DatabaseLayerExecutionOutcome {
 case FinishWithoutErrors, RaiseConnectionClosedError, RaiseTimeoutError,
RaiseQueryExecutionError
}
```

This will indicate the database to fail with different errors. For example, if we change the ***outcome*** value in the database parameters to be a ***RaiseConnectionClosedError*** value, this will be the output:

```
INFO: Starting application ...
INFO: Looking for user with id=1
ERROR: The connection closed unexpectedly in the database layer.
ERROR: A connection exception occurred in the repository layer.
ERROR: A connection exception occurred in the service layer.
timestamp=2023-11-02T08:22:42.522742Z level=ERROR thread=#zio-fiber-1
message="" cause="Exception in thread "zio-fiber-4" java.io.IOException: Did
not find user with id 1
```

As you can see, the error has been handled and logged by the different layers, and propagated with an ***IOException*** to the main thread, as the ***myAppLogic*** indicates.

Final remarks & conclusions

Once you understand the *ZLayers* basic concepts, the data type is not difficult to handle and use. When working with large applications with complex dependency graphs, the debug and error messages can be very helpful.

Also, the *ZLayer* type parameters can detect problems at compile time, something that other frameworks can’t provide.

The *ZLayers* data type provides a safe and trustworthy way to inject and check dependencies, in contrast with other frameworks or with configuring the dependency injection of your application by hand.

References

1) ***”ZLayers in ZIO 2.0 are a totally different BEAST!”*** by **DevInsideYou** Youtube channel video: https://www.youtube.com/watch?v=3ScqDZp9X3c
2) ***“The Five Elements of Service Pattern”***, on **ZIO website online documentation**: https://zio.dev/reference/service-pattern/
3) ***“Inversion of Control Containers and the Dependency Injection pattern”***, by **Martin Fowler**, online reference: https://martinfowler.com/articles/injection.html
4) ***“What is functional programming why it matters for your business?”***, by **Łukasz Kuczera**, Scalac blog online: https://scalac.io/blog/benefits-of-functional-programming/
5) ***“Why Developers Should Pay Attention to ZIO in 2023”***, by **Daria Karasec**, Scalac blog online: https://scalac.io/blog/why-developers-should-pay-attention-to-zio-in-2023/
6) ***“ZIO Quickstart Guides and tutorials”,*** on **ZIO website online documentation**: https://zio.dev/guides/
7) ***“Getting Started with ZIO, Part 1”***, by **Software Mill**, online article: https://softwaremill.com/getting-started-with-zio-part-1/
8) ***The Scala Programming Language official website***: https://www.scala-lang.org/
9) ***Article Sample Application Github code repository***: https://github.com/HivemindTechnologies/zlayers-article