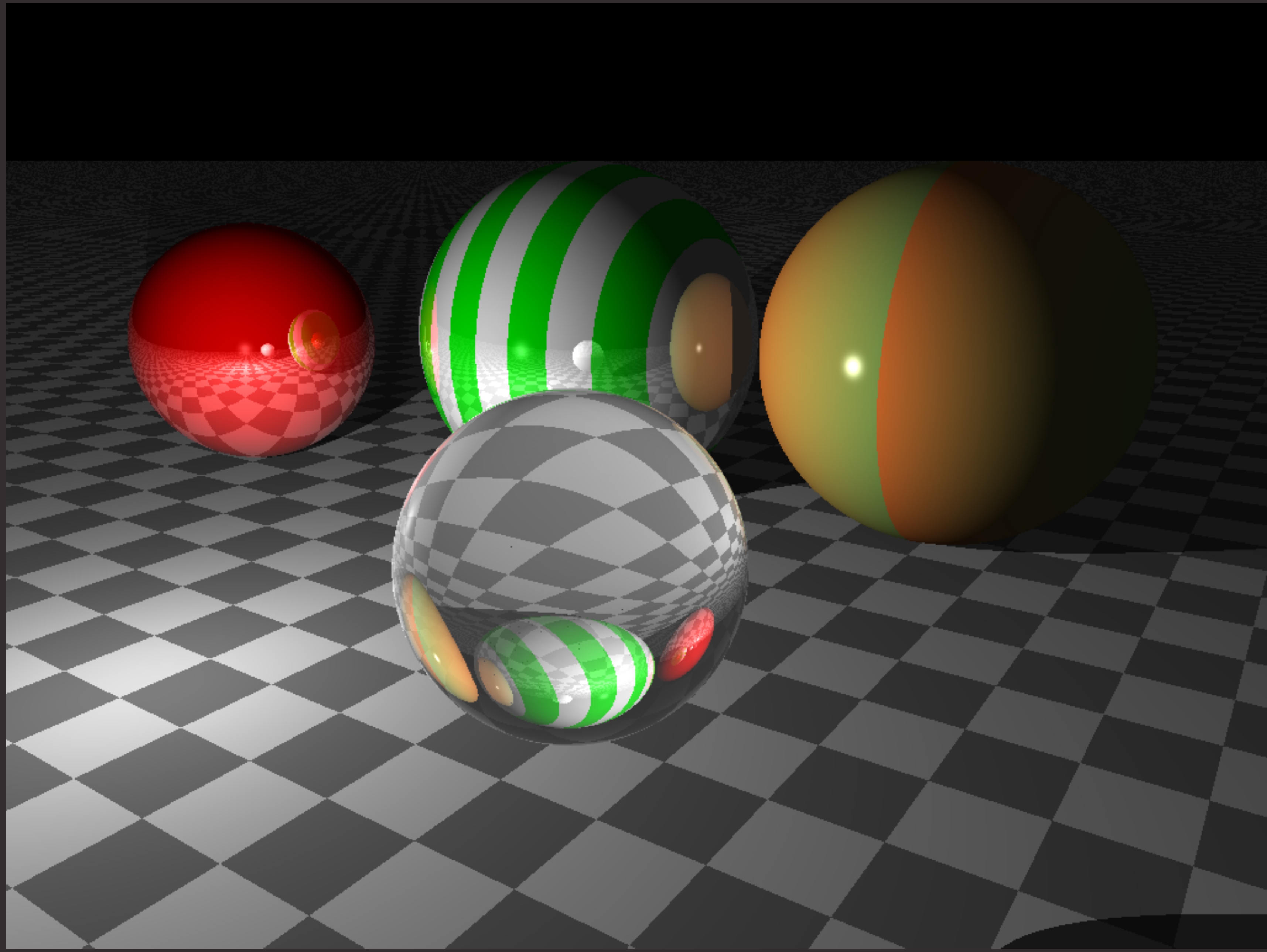
ZLayer

Build a web application

Pierangelo Cecchetto

LambdaConf 2020
18 August 2020



ZIO-101

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

        ⬇

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

        ⬇

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

ZIO-101: Programs

```scala
val prg: ZIO[Console with Random, Nothing, Long] = for {
  n <- random.nextLong                          // ZIO[Random, Nothing, Long]
  _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

ZIO-101: Programs

- ZIO programs are values

```
val prg: ZIO[Console with Random, Nothing, Long] = for {
  n <- random.nextLong                           // ZIO[Random, Nothing, Long]
  _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

ZIO-101: Programs

- ZIO programs are values
- Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {
  n <- random.nextLong                          // ZIO[Random, Nothing, Long]
  _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

ZIO-101: Programs

- ZIO programs are values
- Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {
  n <- random.nextLong                        // ZIO[Random, Nothing, Long]
  _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

ZIO-101: Programs

- ZIO programs are values
- Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {
  n <- random.nextLong                          // ZIO[Random, Nothing, Long]
  _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

ZIO-101: Programs

- ZIO programs are values
- Concurrency based on fibers (green threads)

```scala
val prg: ZIO[Console with Random, Nothing, Long] = for {
  n <- random.nextLong                          // ZIO[Random, Nothing, Long]
  _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

ZIO-101: Programs

- ZIO programs are values
- Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {
  n <- random.nextLong                           // ZIO[Random, Nothing, Long]
  _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

ZIO-101: Programs

- ZIO programs are values
- Concurrency based on fibers (green threads)

```scala
val prg: ZIO[Console with Random, Nothing, Long] = for {
  n <- random.nextLong                        // ZIO[Random, Nothing, Long]
  _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

ZIO-101: R means *requirement*

```
val prg: ZIO[Console with Random, Nothing, Long] = ???

val autonomous: ZIO[Any, Nothing, Long] = ???

val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???
```

ZIO-101: `R` means *requirement*

```scala
val prg: ZIO[Console with Random, Nothing, Long] = ???

val autonomous: ZIO[Any, Nothing, Long] = ???

val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???
```

ZIO-101: `R` means *requirement*

```
val prg: ZIO[Console with Random, Nothing, Long] = ???

val autonomous: ZIO[Any, Nothing, Long] = ???

val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???
```

ZIO-101: `R` means *requirement*

```
val prg: ZIO[Console with Random, Nothing, Long] = ???

val autonomous: ZIO[Any, Nothing, Long] = ???

val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???
```

ZIO-101: Requirements elimination

```
val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???

val provided: ZIO[Any, Nothing, User] =
  getUserFromDb.provide(DBConnection(...))

val user: User = Runtime.default.unsafeRun(provided)
```

ZIO-101: Requirements elimination

```
val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???

val provided: ZIO[Any, Nothing, User] =
  getUserFromDb.provide(DBConnection(...))

val user: User = Runtime.default.unsafeRun(provided)
```

ZIO-101: Requirements elimination

```
val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???

val provided: ZIO[Any, Nothing, User] =
  getUserFromDb.provide(DBConnection(...))

val user: User = Runtime.default.unsafeRun(provided)
```

ZIO-101: Requirements elimination

```
val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???

val provided: ZIO[Any, Nothing, User] =
  getUserFromDb.provide(DBConnection(...))

val user: User = Runtime.default.unsafeRun(provided)
```

ZIO-101: Useful Aliases

```
type IO[+E, +A]    = ZIO[Any, E, A]
type Task[+A]      = ZIO[Any, Throwable, A]
type RIO[-R, +A]   = ZIO[R, Throwable, A]
type UIO[+A]       = ZIO[Any, Nothing, A]
type URIO[-R, +A]  = ZIO[R, Nothing, A]
```

ZIO-101: Useful Aliases

```
type IO[+E, +A]     = ZIO[Any, E, A]
type Task[+A]       = ZIO[Any, Throwable, A]
type RIO[-R, +A]    = ZIO[R, Throwable, A]
type UIO[+A]        = ZIO[Any, Nothing, A]
type URIO[-R, +A]   = ZIO[R, Nothing, A]
```

ZIO-101: Useful Aliases

```
type IO[+E, +A]    = ZIO[Any, E, A]
type Task[+A]      = ZIO[Any, Throwable, A]
type RIO[-R, +A]   = ZIO[R, Throwable, A]
type UIO[+A]       = ZIO[Any, Nothing, A]
type URIO[-R, +A] = ZIO[R, Nothing, A]
```

ZIO-101: Useful Aliases

```
type IO[+E, +A]     = ZIO[Any, E, A]
type Task[+A]       = ZIO[Any, Throwable, A]
type RIO[-R, +A]    = ZIO[R, Throwable, A]
type UIO[+A]        = ZIO[Any, Nothing, A]
type URIO[-R, +A]   = ZIO[R, Nothing, A]
```

ZIO-101: Useful Aliases

```
type IO[+E, +A]    = ZIO[Any, E, A]
type Task[+A]      = ZIO[Any, Throwable, A]
type RIO[-R, +A]   = ZIO[R, Throwable, A]
type UIO[+A]       = ZIO[Any, Nothing, A]
type URIO[-R, +A] = ZIO[R, Nothing, A]
```

ZIO-101: Useful Aliases

```
type IO[+E, +A]     = ZIO[Any, E, A]
type Task[+A]       = ZIO[Any, Throwable, A]
type RIO[-R, +A]    = ZIO[R, Throwable, A]
type UIO[+A]        = ZIO[Any, Nothing, A]
type URIO[-R, +A] = ZIO[R, Nothing, A]
```

ZIO-101: Modules

Example: a module to collect metrics

```
type Metrics = Has[Metrics.Service]
object Metrics {
  trait Service {
    def inc(label: String): IO[Nothing, Unit]
  }

  //accessor method
  def inc(label: String): ZIO[Metrics, Nothing, Unit] =
      ZIO.accessM(_.get.inc(label))
  }
}
```

ZIO-101: Modules

Example: a module to collect metrics

```scala
type Metrics = Has[Metrics.Service]
object Metrics {
  trait Service {
    def inc(label: String): IO[Nothing, Unit]
  }

  //accessor method
  def inc(label: String): ZIO[Metrics, Nothing, Unit] =
      ZIO.accessM(_.get.inc(label))
  }
}
```

ZIO-101: Modules

Example: a module to collect metrics

```
type Metrics = Has[Metrics.Service]
object Metrics {
  trait Service {
    def inc(label: String): IO[Nothing, Unit]
  }

  //accessor method
  def inc(label: String): ZIO[Metrics, Nothing, Unit] =
      ZIO.accessM(_.get.inc(label))
  }
}
```

ZIO-101: Modules

Example: a module for logging

```
type Log = Has[Log.Service]
object Log {
  trait Service {
    def info(s: String): IO[Nothing, Unit]
    def error(s: String): IO[Nothing, Unit]
  }

  //accessor methods...
}
```

ZIO-101: Modules

Write a program using existing modules, i.e. *program to an interface*

```
val prg: ZIO[Metrics with Log, Nothing, Unit] =
  for {
    _ <- Log.info("Hello")
    _ <- Metrics.inc("salutation")
    _ <- Log.info("LambdaConf")
    _ <- Metrics.inc("subject")
  } yield ()
```

ZIO-101: The `Has` data type

`Has[A]` is a dependency on a value of type A

```
val hasLog: Has[Log.Service]           // type Log     = Has[Log.Service]
val hasMetrics: Has[Metrics.Service] // type Metrics = Has[Metrics.Service]
val mix: Log with Metrics = hasLog ++ hasMetrics

//access each service
mix.get[Log.Service].info("Starting the application")
```

ZIO-101: The `Has` data type

`Has[A]` is a dependency on a value of type A

```
val hasLog: Has[Log.Service]               // type Log     = Has[Log.Service]
val hasMetrics: Has[Metrics.Service] // type Metrics = Has[Metrics.Service]
val mix: Log with Metrics = hasLog ++ hasMetrics

//access each service
mix.get[Log.Service].info("Starting the application")
```

ZIO-101: The `Has` data type

```
val mix: Log with Metrics = hasLog ++ hasMetrics

mix.get[Log.Service].info("Starting the application")
```

ZIO-101: The `Has` data type

```
val mix: Log with Metrics = hasLog ++ hasMetrics

mix.get[Log.Service].info("Starting the application")
```

- To the compiler this looks like trait mixin

ZIO-101: The `Has` data type

```scala
val mix: Log with Metrics = hasLog ++ hasMetrics

mix.get[Log.Service].info("Starting the application")
```

- To the compiler this looks like trait mixin
- Plays well with `ZIO[-R, _, _]`

ZIO-101: The `Has` data type

```
val mix: Log with Metrics = hasLog ++ hasMetrics

mix.get[Log.Service].info("Starting the application")
```

- To the compiler this looks like trait mixin
- Plays well with `ZIO[-R, _, _]`
- Is backed by a heterogeneus map `ServiceType -> Service`

ZIO-101: The `Has` data type

```
val mix: Log with Metrics = hasLog ++ hasMetrics

mix.get[Log.Service].info("Starting the application")
```

- To the compiler this looks like trait mixin
- Plays well with `ZIO[-R, _, _]`
- Is backed by a heterogeneus map `ServiceType -> Service`
- Can replace/update services

ZIO-101: ZLayer

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

ZIO-101: `ZLayer`

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

- A recipe to build an `ROut`

ZIO-101: `ZLayer`

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

- A recipe to build an `ROut`
- Backed by `ZManaged`: safe acquire/release

ZIO-101: `ZLayer`

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

- A recipe to build an `ROut`
- Backed by `ZManaged`: safe acquire/release
- `type Layer[+E, +ROut] = ZLayer[Any, E, ROut]`

ZIO-101: `ZLayer`

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

- A recipe to build an `ROut`
- Backed by `ZManaged`: safe acquire/release
- `type Layer[+E, +ROut] = ZLayer[Any, E, ROut]`
- `type ULayer[+ROut]      = ZLayer[Any, Nothing, ROut]`

ZIO-101: `ZLayer`

Construct from value

```scala
val layer: ULayer[UserRepo] =
  ZLayer.succeed(new UserRepo.Service)
```

ZIO-101: `ZLayer`

Construct from function

```
val layer: URLayer[Connection, UserRepo] =
  ZLayer.fromFunction { c: Connection =>
    new UserRepo.Service
  }
```

ZIO-101: ZLayer

Construct from effect

```
import java.sql.Connection

val e: ZIO[Connection, Error, UserRepo.Service]

val layer: ZLayer[Connection, Error, UserRepo] =
  ZLayer.fromEffect(e)
```

ZIO-101: ZLayer

Construct from resources

```
import java.sql.Connection

val connectionLayer: Layer[Nothing, Has[Connection]] =
  ZLayer.fromAcquireRelease(makeConnection) { c =>
     UIO(c.close())
  }
```

ZIO-101: ZLayer

Construct from other services

```
val usersLayer: URLayer[UserRepo with UserValidation, BusinessLogic] =

  ZLayer.fromServices[UserRepo.Service, UserValidation.Service] {
    (repoSvc, validSvc) =>
      new BusinessLogic.Service {
        // use repoSvc and validSvc
      }
    }
```

ZIO-101: ZLayer

Compose horizontally

(*all inputs for all outputs*)

```
val l1: ZLayer[Connection, Nothing, UserRepo]
val l2: ZLayer[Config, Nothing, AuthPolicy]

val hor: ZLayer[Connection with Config, Nothing, UserRepo with AuthPolicy] =
  l1 ++ l2
```

ZIO-101: ZLayer

Compose vertically

(*output of first for input of second*)

```
val l1: ZLayer[Config, Nothing, Connection]
val l2: ZLayer[Connection, Nothing, UserRepo]

val ver: ZLayer[Config, Nothing, UserRepo] =
  l1 >>> l2
```

ZIO-101: ZLayer

Provide required module to a program

```
val p: ZIO[Metrics, Nothing, Unit] = Metrics.inc("LambdaConf")

Metrics.live: ULayer[Metrics]

val runnable: ZIO[Any, Nothing, Unit] = p.provideLayer(Metrics.live)

Runtime.default.unsafeRun(runnable)
```

ZIO-101: ZLayer

Provide required module to a program

```
val p: ZIO[Metrics, Nothing, Unit] = Metrics.inc("LambdaConf")

Metrics.live: ULayer[Metrics]

val runnable: ZIO[Any, Nothing, Unit] = p.provideLayer(Metrics.live)

Runtime.default.unsafeRun(runnable)
```

ZIO-101: ZLayer

Provide required module to a program

```
val p: ZIO[Metrics, Nothing, Unit] = Metrics.inc("LambdaConf")

Metrics.live: ULayer[Metrics]

val runnable: ZIO[Any, Nothing, Unit] = p.provideLayer(Metrics.live)

Runtime.default.unsafeRun(runnable)
```

ZIO-101: ZLayer

Provide required module to a program

```
val p: ZIO[Metrics, Nothing, Unit] = Metrics.inc("LambdaConf")

Metrics.live: ULayer[Metrics]

val runnable: ZIO[Any, Nothing, Unit] = p.provideLayer(Metrics.live)

Runtime.default.unsafeRun(runnable)
```

ZIO-101: ZLayer

ZIO uses ZLayer to provide the basic modules, all bundled in ZEnv

```scala
package object console {
  type Console = Has[Console.Service]
}

val p: URIO[Console, Unit] = zio.console.putStrLn("Hello world")

Runtime.default.unsafeRun(p)
```

Digression

Digression

- What is FP?

Digression

- What is FP?
    - Referential Transparency 👍

Digression

- What is FP?
  - Referential Transparency 👍
  - Immutability 👍

Digression

- What is FP?
  - Referential Transparency 👍
  - Immutability 👍
  - Modularity and composability! 🚀

Digression

- What is FP?
  - Referential Transparency 👍
  - Immutability 👍
  - Modularity and composability! 🚀
- ZLayer is a tool to compose dependency trees of arbitrary complexity, with strong resource management guarantees

Build a simple application

Given: A module that computes a png from scene description[1]

```
case class SceneBundle(world: World, viewFrom: Pt, viewTo: Pt) // a bit simplified

object PngRenderer {

  trait Service {
    def draw(scene: SceneBundle): UIO[Chunk[Byte]]
  }

  def draw(scene: SceneBundle): URIO[PngRenderer, Chunk[Byte]] =
    ZIO.accessM(_.get.draw(scene))

  val live: URLayer[CanvasSerializer with RasteringModule with ATModule, PngRenderer] = ???
}
```

[1] left over from a previous PoC about ZIO modularity

A simple application

A simple application

- Wrap in http layer

A simple application

- Wrap in http layer
- Minimal user management

A simple application

- Wrap in http layer
- Minimal user management
- Users can fetch their scenes after authentication

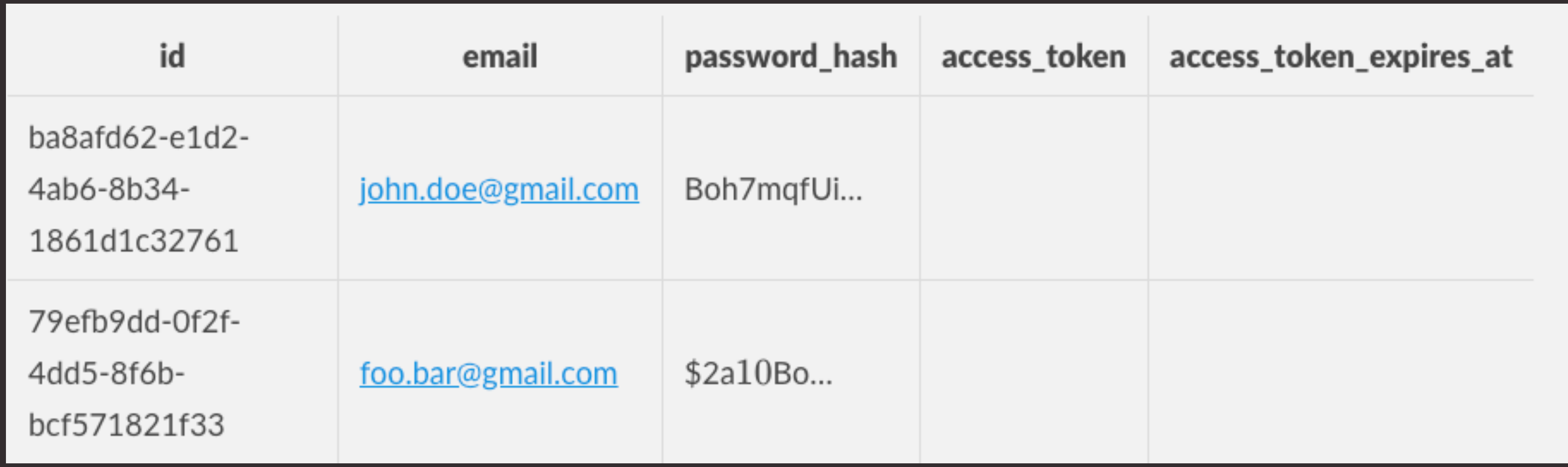
User Management / UserRepo

```
object UsersRepo {

  trait Service {
    def createUser(user: User): IO[DBError, Unit]
    def getUser(userId: UserId): IO[DBError, Option[User]]
    def getUserByEmail(email: Email): IO[DBError, Option[User]]
    def getUserByAccessToken(email: AccessToken): IO[DBError, Option[User]]
    def updatePassword(userId: UserId, newPassword: PasswordHash): IO[DBError, Unit]
    def updateAccessToken(
      userId: UserId, newAccessToken: AccessToken, expiresAt: ZonedDateTime
    ): IO[DBError, Unit]
  }

  /* and accessor methods */
```

User Management / UserRepo

```scala
val live: URLayer[DB.Transactor, UsersRepo] =
  ZLayer.fromService[HikariTransactor[Task], UsersRepo.Service] {
    transactor =>
    new Service {

      def getUser(userId: UserId): IO[DBError, Option[User]] = {
        Queries.getUser(userId)
          .option.transact(transactor)
          .mapError(e =>
            DBError(s"Error fetching user with id = $userId", Some(e))
          )
      }
    }
  }
```

```scala
object Queries {
  def getUser(userId: UserId): Query0[User] =
    sql"""select * from users
         |  where id = ${userId.value}
         """.stripMargin.query[User]
}
```

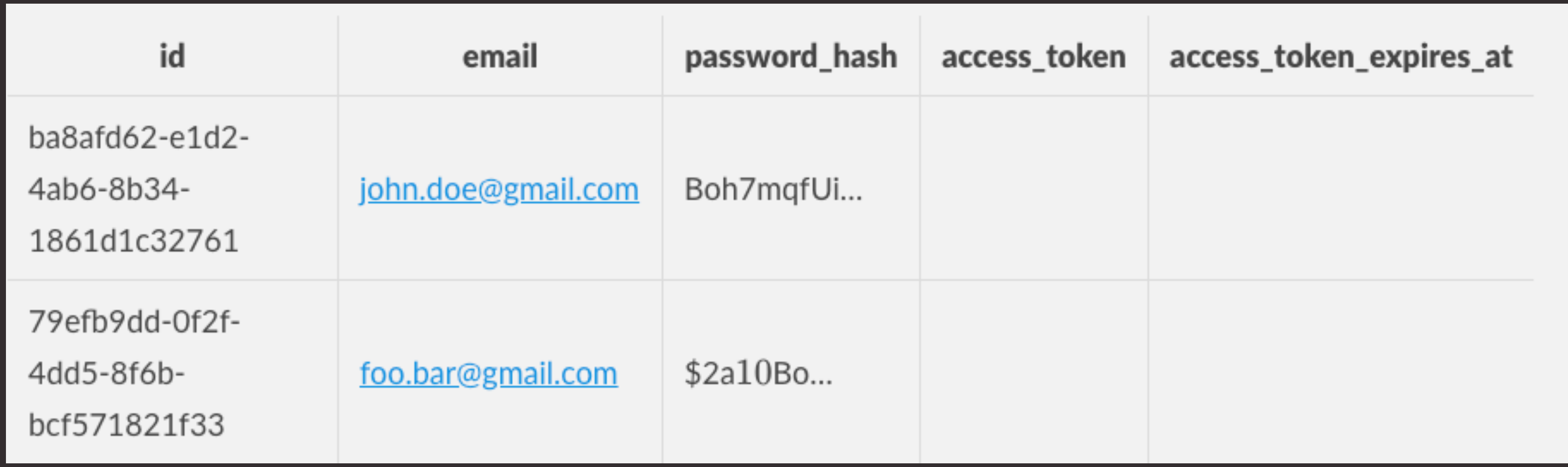


id	email	password_hash	access_token	access_token_expires_at
ba8afd62-e1d2-4ab6-8b34-1861d1c32761	john.doe@gmail.com	Boh7mqfUi...		
79efb9dd-0f2f-4dd5-8f6b-bcf571821f33	foo.bar@gmail.com	$2a10Bo...		

User Management / UserRepo

```
val live: URLayer[DB.Transactor, UsersRepo] =
  ZLayer.fromService[HikariTransactor[Task], UsersRepo.Service] {
    transactor =>
    new Service {

      def getUser(userId: UserId): IO[DBError, Option[User]] = {
        Queries.getUser(userId)
          .option.transact(transactor)
          .mapError(e =>
            DBError(s"Error fetching user with id = $userId", Some(e))
          )
      }
    }
  }
```

```
object Queries {
  def getUser(userId: UserId): Query0[User] =
    sql"""select * from users
         |  where id = ${userId.value}
         """.stripMargin.query[User]
}
```

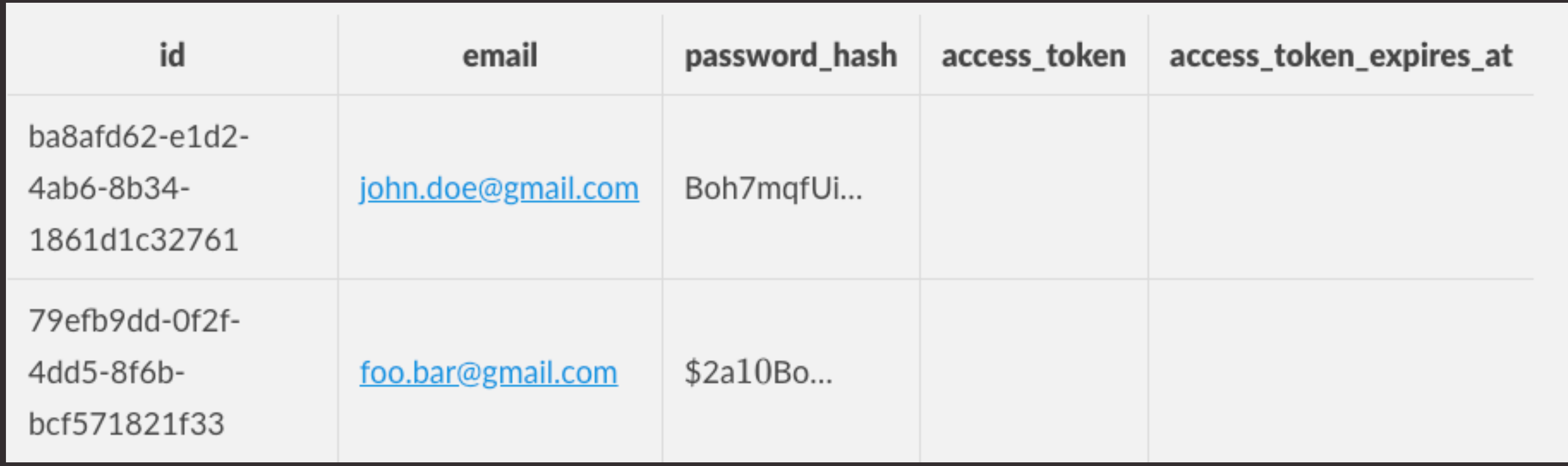


id	email	password_hash	access_token	access_token_expires_at
ba8afd62-e1d2-4ab6-8b34-1861d1c32761	john.doe@gmail.com	Boh7mqfUi...		
79efb9dd-0f2f-4dd5-8f6b-bcf571821f33	foo.bar@gmail.com	$2a10Bo...		

User Management / UserRepo

```scala
val live: URLayer[DB.Transactor, UsersRepo] =
  ZLayer.fromService[HikariTransactor[Task], UsersRepo.Service] {
    transactor =>
    new Service {

      def getUser(userId: UserId): IO[DBError, Option[User]] = {
        Queries.getUser(userId)
          .option.transact(transactor)
          .mapError(e =>
            DBError(s"Error fetching user with id = $userId", Some(e))
          )
      }
    }
  }
```

```scala
object Queries {
  def getUser(userId: UserId): Query0[User] =
    sql"""select * from users
         |  where id = ${userId.value}
         """.stripMargin.query[User]
}
```



id	email	password_hash	access_token	access_token_expires_at
ba8afd62-e1d2-4ab6-8b34-1861d1c32761	john.doe@gmail.com	Boh7mqfUi...		
79efb9dd-0f2f-4dd5-8f6b-bcf571821f33	foo.bar@gmail.com	$2a10Bo...		

User Management / Service

```
object Users {

  case class UserCreated(userId: UserId)
  case class PasswordUpdated(userId: UserId)
  case class LoginSuccess(userId: UserId, accessToken: AccessToken)

  trait Service {
    def createUser(email: Email): IO[APIError, UserCreated]
    def updatePassword(email: Email, newPassword: ClearPassword): IO[APIError, PasswordUpdated]
    def login(userEmail: Email, givenPassword: ClearPassword): IO[APIError, LoginSuccess]
  }
```

User Management / Service

```scala
val live: URLayer[UsersRepo with Logging with Clock, Has[Service]] =
  ZLayer.fromServices[UsersRepo.Service, Logger[String], Clock.Service, Service] { (usersRepo, logger, clock) =>

  new Service {

    def login(userEmail: Email, clearPassword: ClearPassword): IO[APIError, LoginSuccess] =
        for {
          user <- usersRepo.getUserByEmail(userEmail).catchAll(e =>
            logger.throwable("DB error fetching user by email", e) *>
              ZIO.fail(APIError("Couldn't fetch user"))
          ).some.mapError(_ => APIError("User not found"))
          pwdHash    <- user.password.fold[IO[APIError, PasswordHash]](
              ZIO.fail(APIError("Password not set for user, cannot authenticate"))
            )(ZIO.succeed(_))
          newToken   <- createToken(clearPassword, pwdHash)
          now        <- clock.instant
          _ <- usersRepo.updateAccessToken(user.id, newToken, now.atZone(ZoneId.of("UTC")))
                .catchAll { dbErr =>
                  logger.throwable("DB Error updating access token", dbErr)
                    .as(APIError("Could not update access token, you must login again"))
                }
        } yield LoginSuccess(user.id, newToken)
```

User Management / Service

```
val live: URLayer[UsersRepo with Logging with Clock, Has[Service]] =
  ZLayer.fromServices[UsersRepo.Service, Logger[String], Clock.Service, Service] { (usersRepo, logger, clock) =>

  new Service {

    def login(userEmail: Email, clearPassword: ClearPassword): IO[APIError, LoginSuccess] =
        for {
          user <- usersRepo.getUserByEmail(userEmail).catchAll(e =>
            logger.throwable("DB error fetching user by email", e) *>
              ZIO.fail(APIError("Couldn't fetch user"))
          ).some.mapError(_ => APIError("User not found"))
          pwdHash    <- user.password.fold[IO[APIError, PasswordHash]](
              ZIO.fail(APIError("Password not set for user, cannot authenticate"))
            )(ZIO.succeed(_))
          newToken   <- createToken(clearPassword, pwdHash)
          now        <- clock.instant
          _ <- usersRepo.updateAccessToken(user.id, newToken, now.atZone(ZoneId.of("UTC")))
                .catchAll { dbErr =>
                  logger.throwable("DB Error updating access token", dbErr)
                    .as(APIError("Could not update access token, you must login again"))
                }
        } yield LoginSuccess(user.id, newToken)
```

User Management / Service

```scala
val live: URLayer[UsersRepo with Logging with Clock, Has[Service]] =
  ZLayer.fromServices[UsersRepo.Service, Logger[String], Clock.Service, Service] { (usersRepo, logger, clock) =>

  new Service {

    def login(userEmail: Email, clearPassword: ClearPassword): IO[APIError, LoginSuccess] =
        for {
          user <- usersRepo.getUserByEmail(userEmail).catchAll(e =>
            logger.throwable("DB error fetching user by email", e) *>
              ZIO.fail(APIError("Couldn't fetch user"))
          ).some.mapError(_ => APIError("User not found"))
          pwdHash    <- user.password.fold[IO[APIError, PasswordHash]](
              ZIO.fail(APIError("Password not set for user, cannot authenticate"))
            )(ZIO.succeed(_))
          newToken   <- createToken(clearPassword, pwdHash)
          now        <- clock.instant
          _ <- usersRepo.updateAccessToken(user.id, newToken, now.atZone(ZoneId.of("UTC")))
                .catchAll { dbErr =>
                  logger.throwable("DB Error updating access token", dbErr)
                    .as(APIError("Could not update access token, you must login again"))
                }
        } yield LoginSuccess(user.id, newToken)
```

User Management / Service

```
val live: URLayer[UsersRepo with Logging with Clock, Has[Service]] =
  ZLayer.fromServices[UsersRepo.Service, Logger[String], Clock.Service, Service] { (usersRepo, logger, clock) =>

  new Service {

    def login(userEmail: Email, clearPassword: ClearPassword): IO[APIError, LoginSuccess] =
        for {
          user <- usersRepo.getUserByEmail(userEmail).catchAll(e =>
            logger.throwable("DB error fetching user by email", e) *>
              ZIO.fail(APIError("Couldn't fetch user"))
          ).some.mapError(_ => APIError("User not found"))
          pwdHash    <- user.password.fold[IO[APIError, PasswordHash]](
              ZIO.fail(APIError("Password not set for user, cannot authenticate"))
            )(ZIO.succeed(_))
          newToken   <- createToken(clearPassword, pwdHash)
          now        <- clock.instant
          _ <- usersRepo.updateAccessToken(user.id, newToken, now.atZone(ZoneId.of("UTC")))
                .catchAll { dbErr =>
                  logger.throwable("DB Error updating access token", dbErr)
                    .as(APIError("Could not update access token, you must login again"))
                }
        } yield LoginSuccess(user.id, newToken)
```

User Management / Service

```
val live: URLayer[UsersRepo with Logging with Clock, Has[Service]] =
  ZLayer.fromServices[UsersRepo.Service, Logger[String], Clock.Service, Service] { (usersRepo, logger, clock) =>

  new Service {

    def login(userEmail: Email, clearPassword: ClearPassword): IO[APIError, LoginSuccess] =
        for {
          user <- usersRepo.getUserByEmail(userEmail).catchAll(e =>
            logger.throwable("DB error fetching user by email", e) *>
              ZIO.fail(APIError("Couldn't fetch user"))
          ).some.mapError(_ => APIError("User not found"))
          pwdHash    <- user.password.fold[IO[APIError, PasswordHash]](
              ZIO.fail(APIError("Password not set for user, cannot authenticate"))
            )(ZIO.succeed(_))
          newToken   <- createToken(clearPassword, pwdHash)
          now        <- clock.instant
          _ <- usersRepo.updateAccessToken(user.id, newToken, now.atZone(ZoneId.of("UTC")))
                .catchAll { dbErr =>
                  logger.throwable("DB Error updating access token", dbErr)
                     .as(APIError("Could not update access token, you must login again"))
                }
        } yield LoginSuccess(user.id, newToken)
```

User Management / Service

```scala
val live: URLayer[UsersRepo with Logging with Clock, Has[Service]] =
  ZLayer.fromServices[UsersRepo.Service, Logger[String], Clock.Service, Service] { (usersRepo, logger, clock) =>

  new Service {

    def login(userEmail: Email, clearPassword: ClearPassword): IO[APIError, LoginSuccess] =
        for {
          user <- usersRepo.getUserByEmail(userEmail).catchAll(e =>
            logger.throwable("DB error fetching user by email", e) *>
              ZIO.fail(APIError("Couldn't fetch user"))
          ).some.mapError(_ => APIError("User not found"))
          pwdHash   <- user.password.fold[IO[APIError, PasswordHash]](
              ZIO.fail(APIError("Password not set for user, cannot authenticate"))
            )(ZIO.succeed(_))
          newToken  <- createToken(clearPassword, pwdHash)
          now       <- clock.instant
          _ <- usersRepo.updateAccessToken(user.id, newToken, now.atZone(ZoneId.of("UTC")))
                .catchAll { dbErr =>
                  logger.throwable("DB Error updating access token", dbErr)
                    .as(APIError("Could not update access token, you must login again"))
                }
        } yield LoginSuccess(user.id, newToken)
```

Http Layer

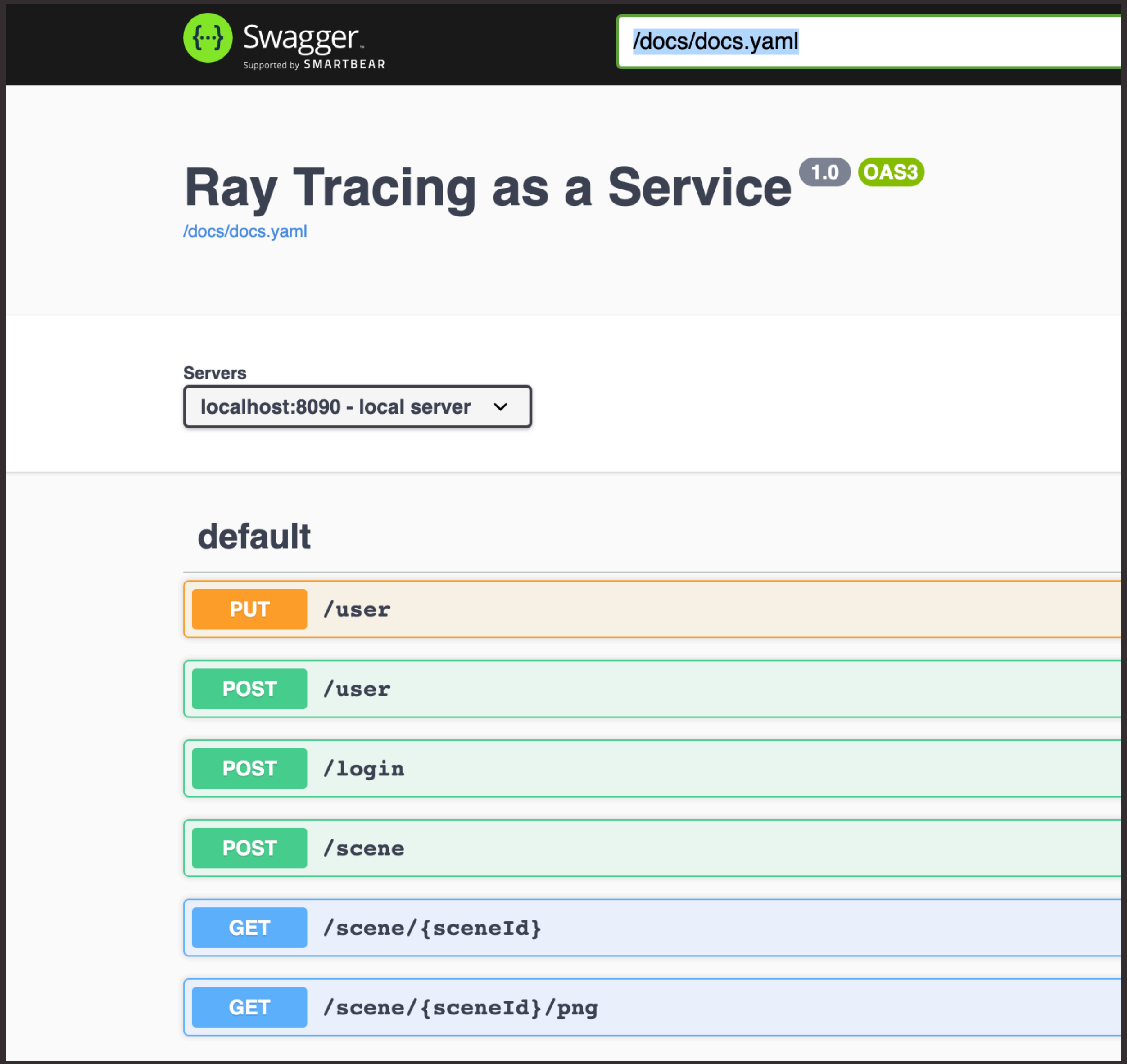
Tapir: **endpoints as values**

```scala
val login: Endpoint[Login, APIError, LoginSuccess, Nothing] =
  endpoint.post.in("login").in(jsonBody[Login]).out(jsonBody[LoginSuccess]).errorOut(jsonBody[APIError])
    .description("Login to obtain an access token")
```

Http Layer

Tapir: OpenAPI **documentation for free**

```
val openApiDocs: OpenAPI = Seq(
  ...
  endpoints.login,
  ...
).toOpenAPI("Ray Tracing as a Service", "1.0")
  .servers(List(Server("localhost:8090").description("local server")))

val docsRoutes: HttpRoutes[Task] = new SwaggerHttp4s(openApiDocs.toYaml).routes[Task]
```



Ray Tracing as a Service 1.0 OAS3

/docs/docs.yaml

Servers

localhost:8090 - local server

default

PUT /user

POST /user

POST /login

POST /scene

GET /scene/{sceneId}

GET /scene/{sceneId}/png

Tapir: Integration with ZIO

```scala
// bind endpoint with module
val loginWithLogic: ZServerEndpoint[Users, Login, APIError, LoginSuccess] =
  endpoints.login.zServerLogic(login =>
    Users.login(login.email, login.password)
  )

//make HttpRoutes for http4s
val loginRoute:    URIO[Users, HttpRoutes[Task]]  = loginWithLogic.toRoutesR
val getSceneRoute: URIO[Scenes, HttpRoutes[Task]] = getSceneWithLogic.toRoutesR

val serve: RIO[Users with Scenes with Logging, Unit] = for {
  allRoutes <- ZIO.mergeAll(List(loginRoute, getSceneRoute))(docsRoutes)(_ <+> _)
  _         <- serveRoutes(allRoutes)
} yield ()
```

Http Layer

Tapir: Integration with ZIO

```
// bind endpoint with module
val loginWithLogic: ZServerEndpoint[Users, Login, APIError, LoginSuccess] =
  endpoints.login.zServerLogic(login =>
    Users.login(login.email, login.password)
  )

//make HttpRoutes for http4s
val loginRoute:    URIO[Users, HttpRoutes[Task]]  = loginWithLogic.toRoutesR
val getSceneRoute: URIO[Scenes, HttpRoutes[Task]] = getSceneWithLogic.toRoutesR

val serve: RIO[Users with Scenes with Logging, Unit] = for {
  allRoutes <- ZIO.mergeAll(List(loginRoute, getSceneRoute))(docsRoutes)(_ <+> _)
  _         <- serveRoutes(allRoutes)
} yield ()
```

Http Layer

Tapir: Integration with ZIO

```
// bind endpoint with module
val loginWithLogic: ZServerEndpoint[Users, Login, APIError, LoginSuccess] =
  endpoints.login.zServerLogic(login =>
    Users.login(login.email, login.password)
  )

//make HttpRoutes for http4s
val loginRoute:    URIO[Users, HttpRoutes[Task]]  = loginWithLogic.toRoutesR
val getSceneRoute: URIO[Scenes, HttpRoutes[Task]] = getSceneWithLogic.toRoutesR

val serve: RIO[Users with Scenes with Logging, Unit] = for {
  allRoutes <- ZIO.mergeAll(List(loginRoute, getSceneRoute))(docsRoutes)(_ <+> _)
  _         <- serveRoutes(allRoutes)
} yield ()
```

Tapir: Integration with ZIO

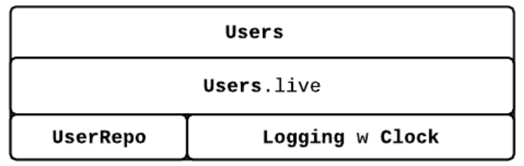
```
// bind endpoint with module
val loginWithLogic: ZServerEndpoint[Users, Login, APIError, LoginSuccess] =
  endpoints.login.zServerLogic(login =>
    Users.login(login.email, login.password)
  )

//make HttpRoutes for http4s
val loginRoute:    URIO[Users, HttpRoutes[Task]]  = loginWithLogic.toRoutesR
val getSceneRoute: URIO[Scenes, HttpRoutes[Task]] = getSceneWithLogic.toRoutesR

val serve: RIO[Users with Scenes with Logging, Unit] = for {
  allRoutes <- ZIO.mergeAll(List(loginRoute, getSceneRoute))(docsRoutes)(_ <+> _)
  _         <- serveRoutes(allRoutes)
} yield ()
```

Putting things together

```scala
val program: ZIO[Users
  with Logging
  with Transactor
  with Scenes, BootstrapError, Unit] =
  for {
    _ <- log.info("Running Flyway migration...")
    _ <- DB.runFlyWay
    _ <- log.info("Flyway migration performed!")
    _ <- serve.mapError(e =>
           BootstrapError("Error starting http server", Some(e))
         )
  } yield ()

override def run(args: List[String]): URIO[zio.ZEnv, ExitCode] =
  program.provideCustomLayer(???)
```

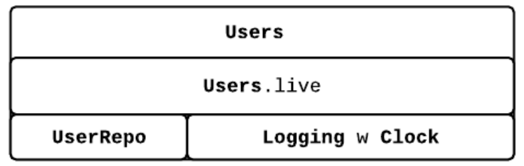
Putting things together

```
val program: ZIO[Users
  with Logging
  with Transactor
  with Scenes, BootstrapError, Unit] = ???

val program: ZIO[Users,
  BootstrapError, Unit] =  ???

Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]
```



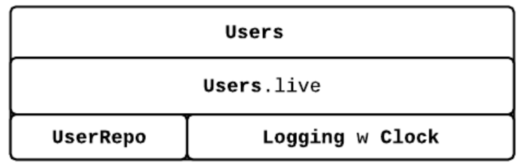
Putting things together

```
val program: ZIO[Users
  with Logging
  with Transactor
  with Scenes, BootstrapError, Unit] = ???

val program: ZIO[Users,
  BootstrapError, Unit] =  ???

Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]
```



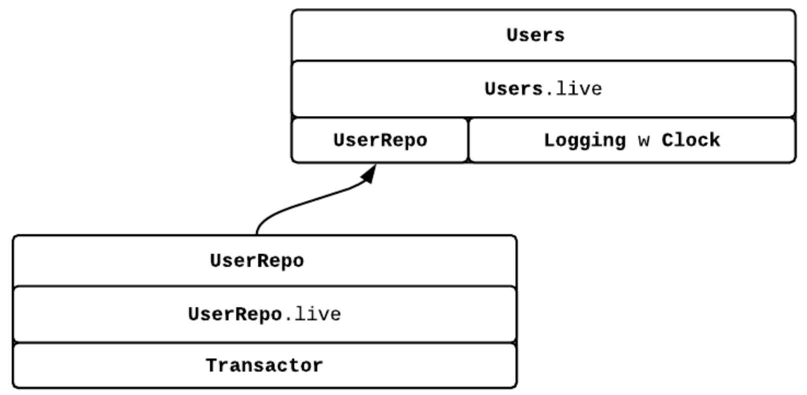
Putting things together

```
val program: ZIO[Users
  with Logging
  with Transactor
  with Scenes, BootstrapError, Unit] = ???

val program: ZIO[Users,
  BootstrapError, Unit] =  ???

Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]
```



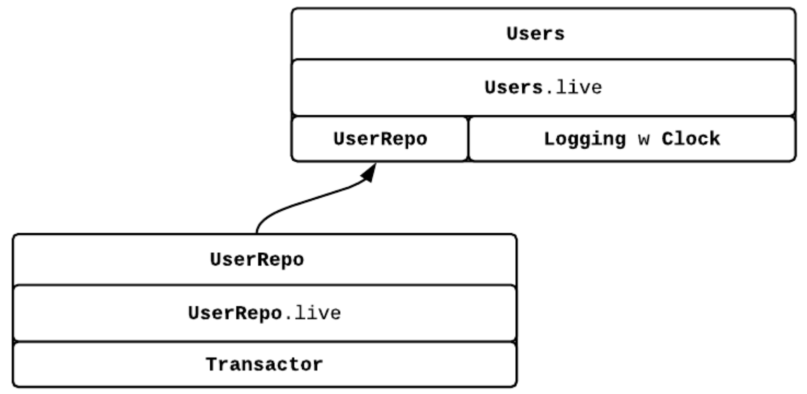
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live
```



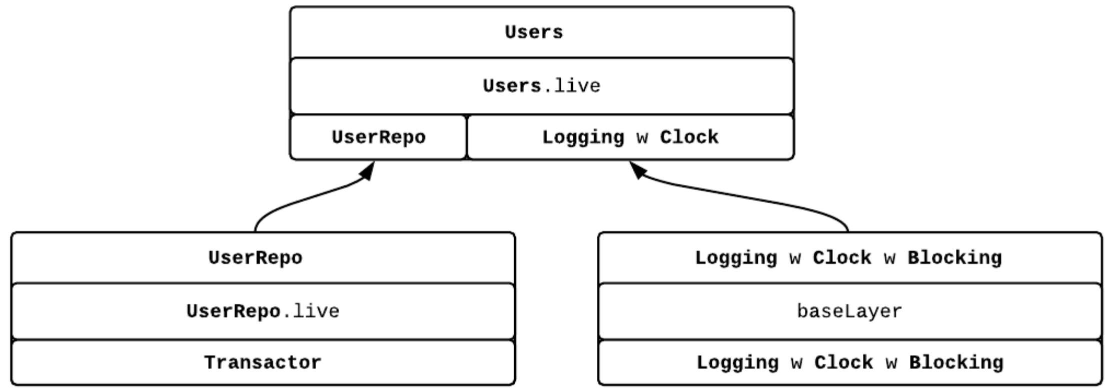
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live
```



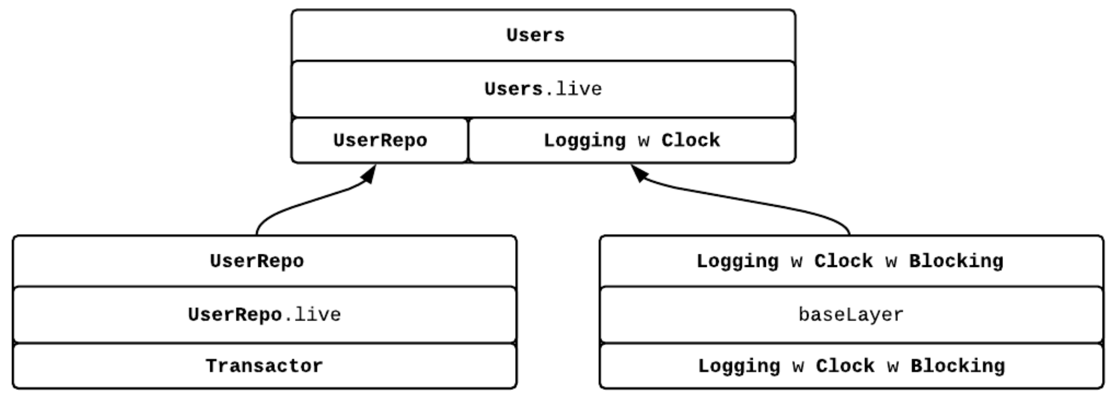
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live
```



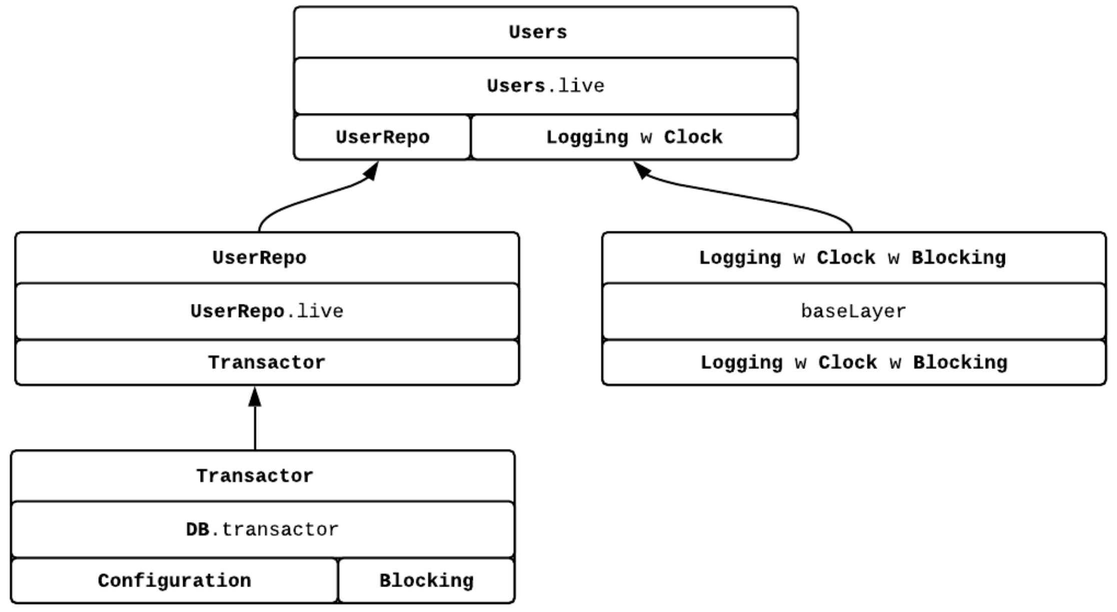
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live
```



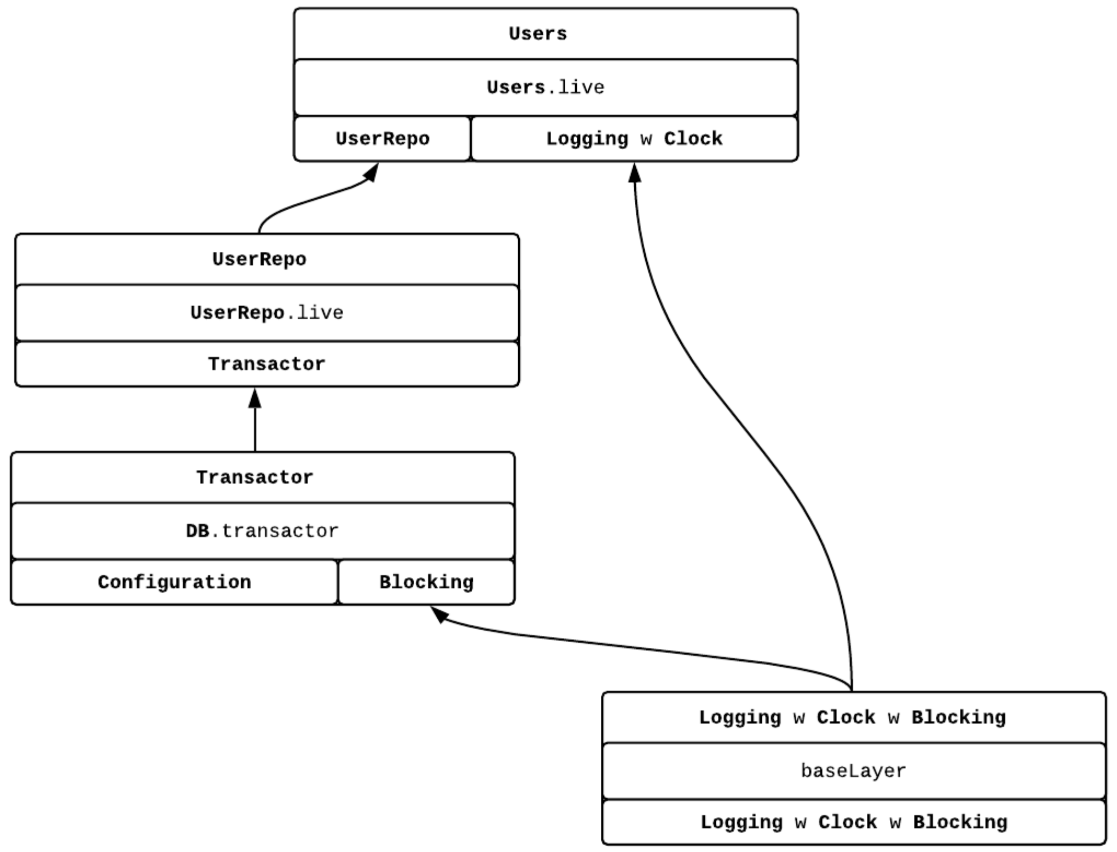
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live

DB.transactor: ZLayer[Blocking with Configuration, DBError, Transactor] = ???
```



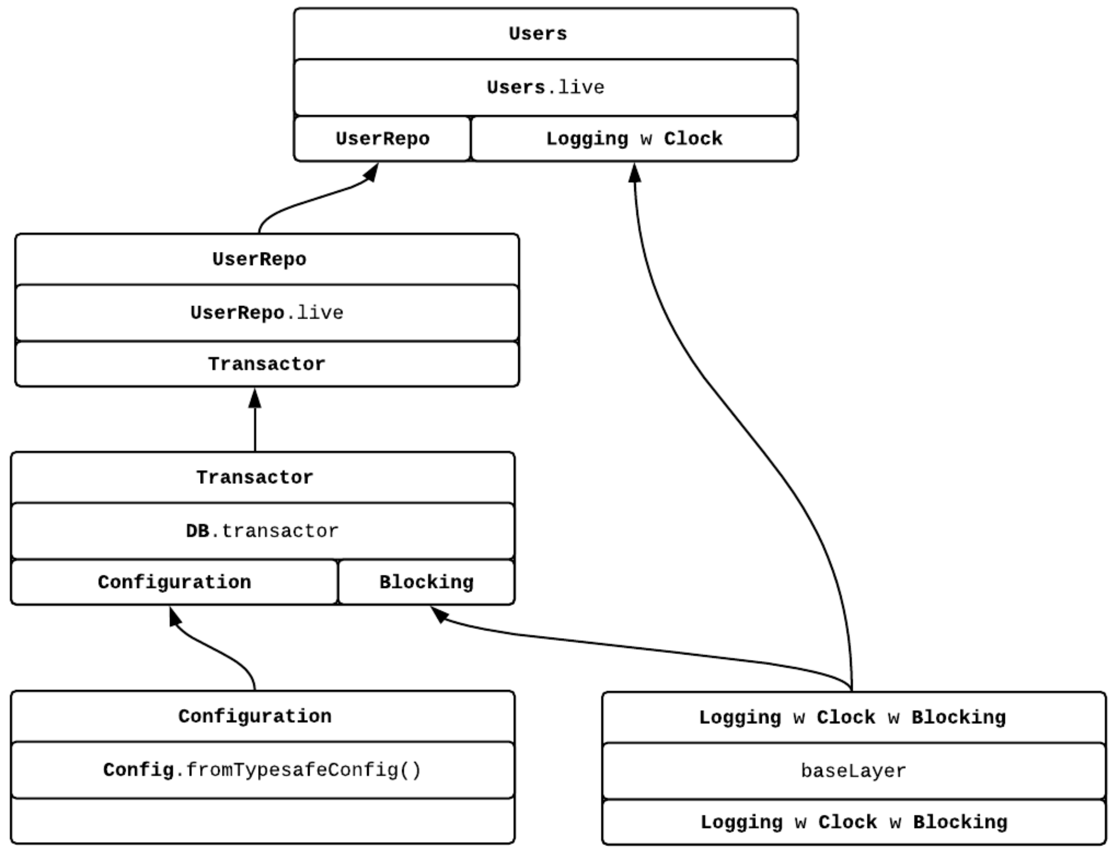
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live

DB.transactor: ZLayer[Blocking with Configuration, DBError, Transactor] = ???
```



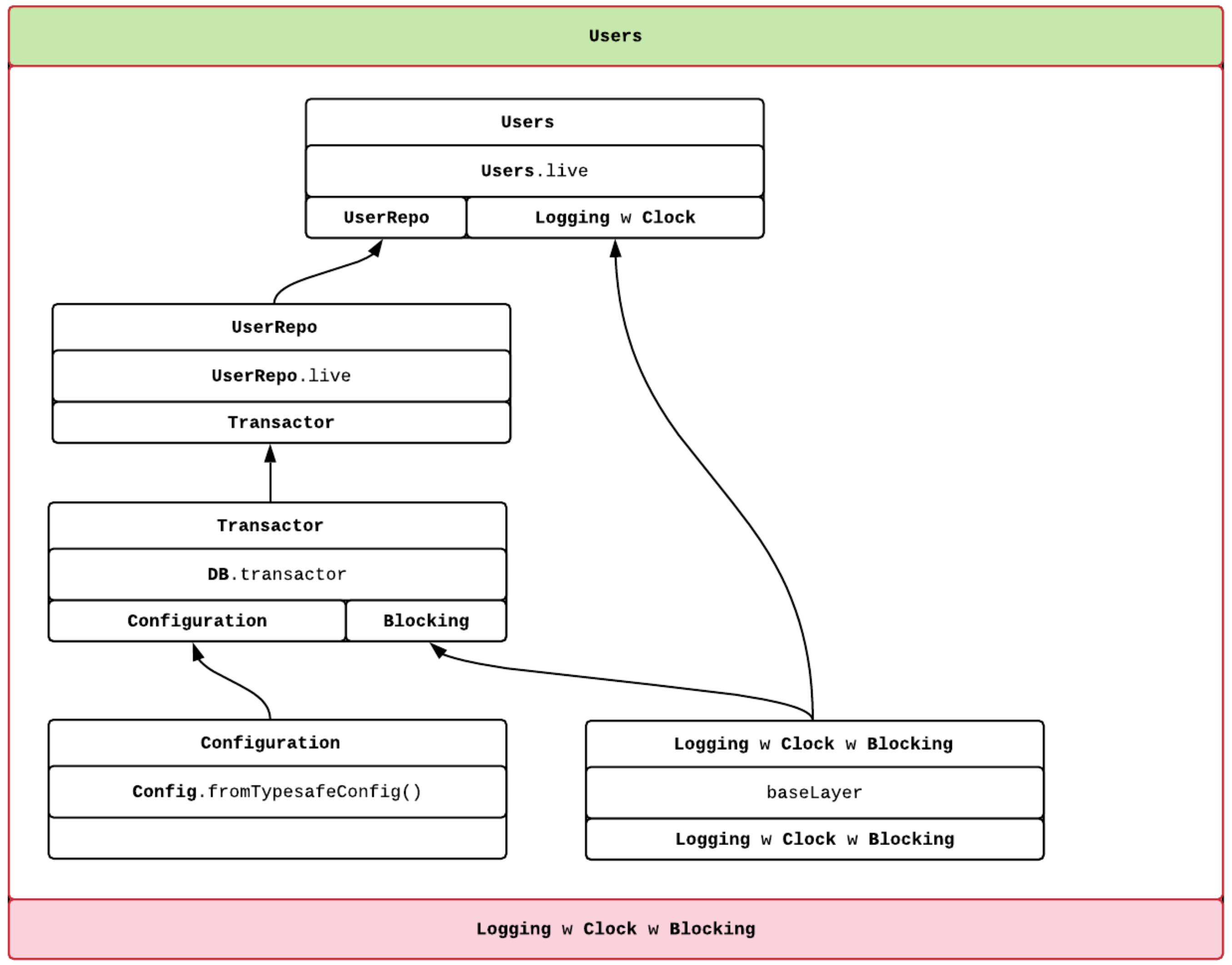
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live

DB.transactor: ZLayer[Blocking with Configuration, DBError, Transactor] = ???

val transactorLayer: ZLayer[Blocking, AppError, Transactor] =
(Config.fromTypesafeConfig() ++ ZLayer.identity[Blocking]) >>> DB.transactor
```



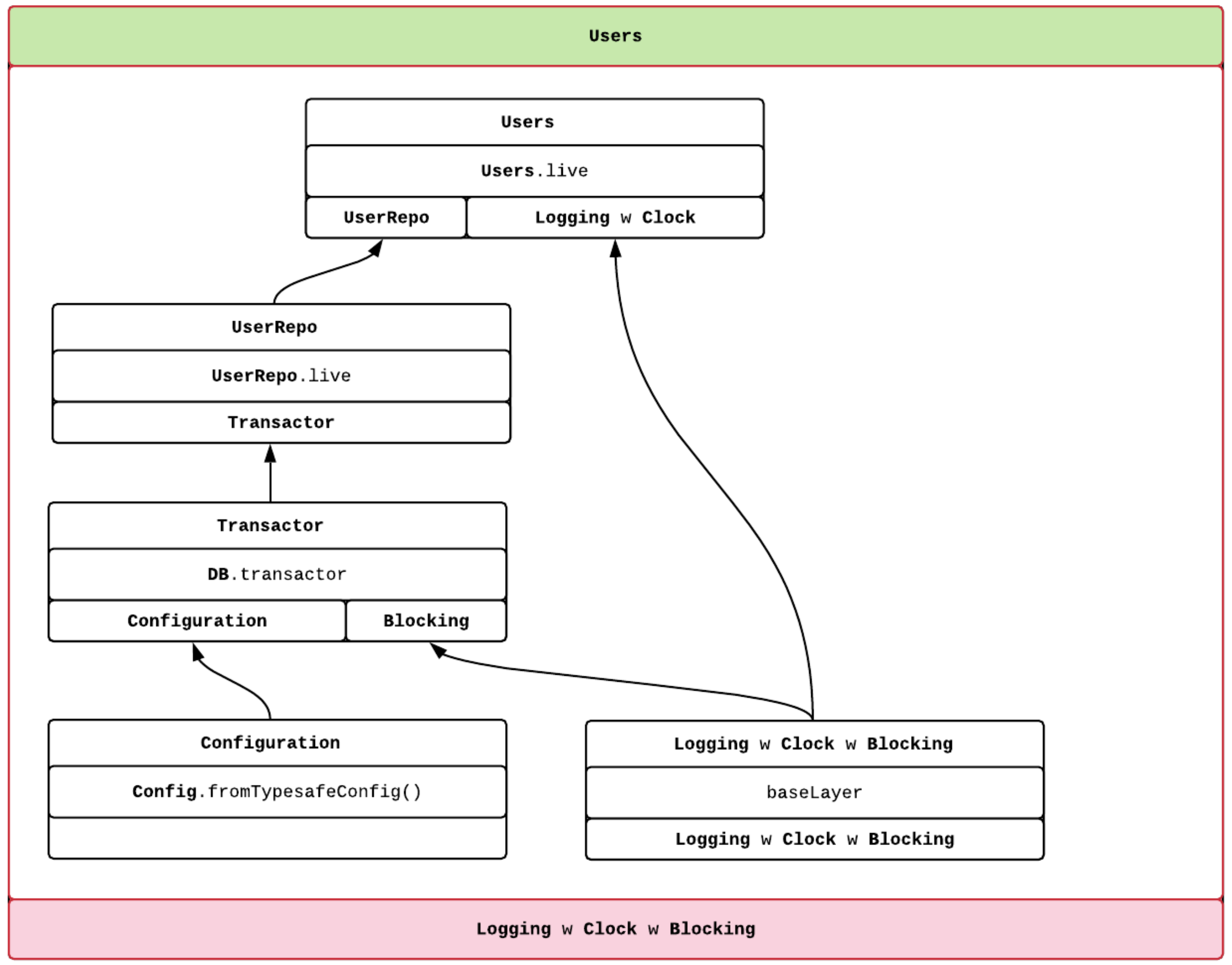
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live

DB.transactor: ZLayer[Blocking with Configuration, DBError, Transactor] = ???

val transactorLayer: ZLayer[Blocking, AppError, Transactor] =
(Config.fromTypesafeConfig() ++ ZLayer.identity[Blocking]) >>> DB.transactor

val fullLayer: ZLayer[AppEnv, AppError, Users] =
(transactorLayer ++ baseLayer) >>> usersLayer

val program: ZIO[Users,
  BootstrapError, Unit] =  ???

val runnable: ZIO[AppEnv,
  AppError, Unit] = program.provideLayer(fullLayer)
```



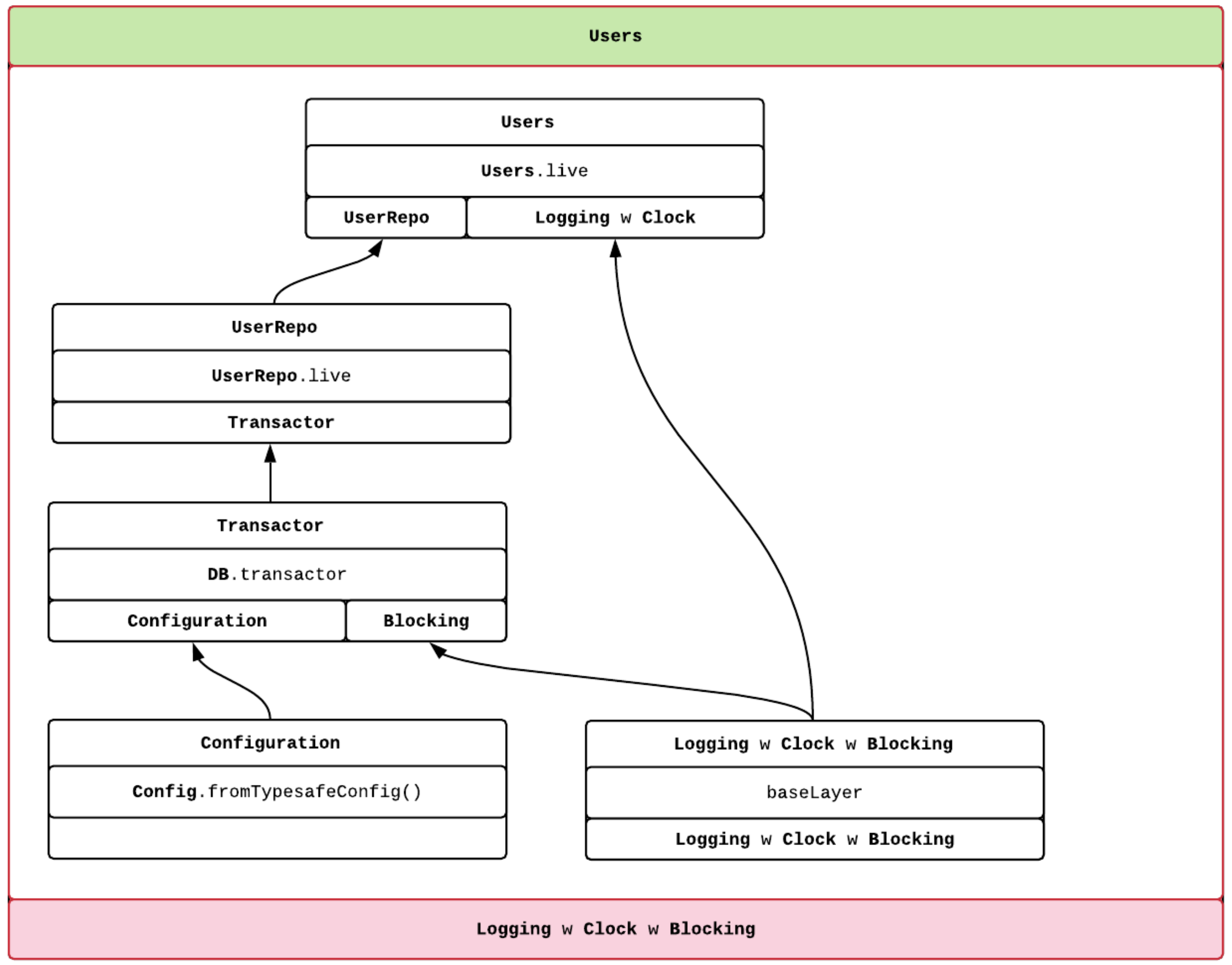
Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live

DB.transactor: ZLayer[Blocking with Configuration, DBError, Transactor] = ???

val transactorLayer: ZLayer[Blocking, AppError, Transactor] =
(Config.fromTypesafeConfig() ++ ZLayer.identity[Blocking]) >>> DB.transactor

val fullLayer: ZLayer[AppEnv, AppError, Users] =
(transactorLayer ++ baseLayer) >>> usersLayer

val program: ZIO[Users,
  BootstrapError, Unit] =  ???

val runnable: ZIO[AppEnv,
  AppError, Unit] = program.provideLayer(fullLayer)
```



Putting things together

```
Users.live: URLayer[UsersRepo
  with Logging
  with Clock, Users]

UsersRepo.live: URLayer[DB.Transactor, UsersRepo] = ???

type AppEnv = Blocking with Clock with Logging
val baseLayer = ZLayer.identity[AppEnv]

val usersLayer: ZLayer[Transactor with AppEnv, AppError, Users] =
(UsersRepo.live ++ baseLayer) >>> Users.live

DB.transactor: ZLayer[Blocking with Configuration, DBError, Transactor] = ???

val transactorLayer: ZLayer[Blocking, AppError, Transactor] =
(Config.fromTypesafeConfig() ++ ZLayer.identity[Blocking]) >>> DB.transactor

val fullLayer: ZLayer[AppEnv, AppError, Users] =
(transactorLayer ++ baseLayer) >>> usersLayer

val program: ZIO[Users,
  BootstrapError, Unit] =  ???

val runnable: ZIO[AppEnv,
  AppError, Unit] = program.provideLayer(fullLayer)
```



Demo time!

Unit testing

Test `Users.live`, mocking dependency on UsersRepo

```
val live: URLayer[UsersRepo with Logging with Clock, Users] = ???

//mock
val userRepo: URLayer[Has[Ref[Map[UserId, User]]], UsersRepo] = ZLayer.fromService (users =>
    new UsersRepo.Service {
      def getUser(userId: UserId): IO[AppError.DBError, Option[User]] =
        users.get.map(_.find(_._1 == userId).map(_._2))
      /* ... */
    })

val usersRepoLayer: ULayer[UsersRepo] = ZLayer.fromEffect(Ref.make(Map(testUser.id -> testUser))) >>> userRepo
val slf4jLogger: ULayer[Logging] = ???

//Test assertion:
(
  for {
    loginOutput <- Users.login(Email("aeinstein@research.com"), ClearPassword("pwd123"))
  } yield assert(loginOutput.userId)(equalTo(testUser.id))
).provideSomeLayer((slf4jLogger ++ usersRepoLayer ++ ZLayer.identity[Clock]) >>> Users.live)
```

Unit testing

Test `Users.live`, mocking dependency on UsersRepo

```
val live: URLayer[UsersRepo with Logging with Clock, Users] = ???

//mock
val userRepo: URLayer[Has[Ref[Map[UserId, User]]], UsersRepo] = ZLayer.fromService (users =>
    new UsersRepo.Service {
      def getUser(userId: UserId): IO[AppError.DBError, Option[User]] =
        users.get.map(_.find(_._1 == userId).map(_._2))
      /* ... */
    })

val usersRepoLayer: ULayer[UsersRepo] = ZLayer.fromEffect(Ref.make(Map(testUser.id -> testUser))) >>> userRepo
val slf4jLogger: ULayer[Logging] = ???

//Test assertion:
(
  for {
    loginOutput <- Users.login(Email("aeinstein@research.com"), ClearPassword("pwd123"))
  } yield assert(loginOutput.userId)(equalTo(testUser.id))
).provideSomeLayer((slf4jLogger ++ usersRepoLayer ++ ZLayer.identity[Clock]) >>> Users.live)
```

Unit testing

Test Users.live, mocking dependency on UsersRepo

```
val live: URLayer[UsersRepo with Logging with Clock, Users] = ???

//mock
val userRepo: URLayer[Has[Ref[Map[UserId, User]]], UsersRepo] = ZLayer.fromService (users =>
    new UsersRepo.Service {
      def getUser(userId: UserId): IO[AppError.DBError, Option[User]] =
        users.get.map(_.find(_._1 == userId).map(_._2))
      /* ... */
    })

val usersRepoLayer: ULayer[UsersRepo] = ZLayer.fromEffect(Ref.make(Map(testUser.id -> testUser))) >>> userRepo
val slf4jLogger: ULayer[Logging] = ???

//Test assertion:
(
  for {
    loginOutput <- Users.login(Email("aeinstein@research.com"), ClearPassword("pwd123"))
  } yield assert(loginOutput.userId)(equalTo(testUser.id))
).provideSomeLayer((slf4jLogger ++ usersRepoLayer ++ ZLayer.identity[Clock]) >>> Users.live)
```

Unit testing

Test Users.live, mocking dependency on UsersRepo

```
val live: URLayer[UsersRepo with Logging with Clock, Users] = ???

//mock
val userRepo: URLayer[Has[Ref[Map[UserId, User]]], UsersRepo] = ZLayer.fromService (users =>
    new UsersRepo.Service {
      def getUser(userId: UserId): IO[AppError.DBError, Option[User]] =
        users.get.map(_.find(_._1 == userId).map(_._2))
      /* ... */
    })

val usersRepoLayer: ULayer[UsersRepo] = ZLayer.fromEffect(Ref.make(Map(testUser.id -> testUser))) >>> userRepo
val slf4jLogger: ULayer[Logging] = ???

//Test assertion:
(
  for {
    loginOutput <- Users.login(Email("aeinstein@research.com"), ClearPassword("pwd123"))
  } yield assert(loginOutput.userId)(equalTo(testUser.id))
).provideSomeLayer((slf4jLogger ++ usersRepoLayer ++ ZLayer.identity[Clock]) >>> Users.live)
```

Unit testing

Test `Users.live`, mocking dependency on UsersRepo

```scala
val live: URLayer[UsersRepo with Logging with Clock, Users] = ???

//mock
val userRepo: URLayer[Has[Ref[Map[UserId, User]]], UsersRepo] = ZLayer.fromService (users =>
    new UsersRepo.Service {
      def getUser(userId: UserId): IO[AppError.DBError, Option[User]] =
        users.get.map(_.find(_._1 == userId).map(_._2))
      /* ... */
    })

val usersRepoLayer: ULayer[UsersRepo] = ZLayer.fromEffect(Ref.make(Map(testUser.id -> testUser))) >>> userRepo
val slf4jLogger: ULayer[Logging] = ???

//Test assertion:
(
  for {
    loginOutput <- Users.login(Email("aeinstein@research.com"), ClearPassword("pwd123"))
  } yield assert(loginOutput.userId)(equalTo(testUser.id))
).provideSomeLayer((slf4jLogger ++ usersRepoLayer ++ ZLayer.identity[Clock]) >>> Users.live)
```

Conclusion - ZLayer

Conclusion - ZLayer

- Dependency graph in the code 💪

Conclusion - ZLayer

- Dependency graph in the code 💪
- Type safety, no magic, full control 🙌

Conclusion - ZLayer

- Dependency graph in the code 💪
- Type safety, no magic, full control 🙌
- Compiler helps to satisfy requirements 🤗

Conclusion - ZLayer

- Dependency graph in the code 💪
- Type safety, no magic, full control 🙌
- Compiler helps to satisfy requirements 🤗
- Resource safety 🦺

Conclusion - ZLayer

- Dependency graph in the code 💪
- Type safety, no magic, full control 🙌
- Compiler helps to satisfy requirements 🤗
- Resource safety 🦺
- Easy to onboard 😊

Thank you!

@pierangelocecc

https://github.com/pierangeloc/ray-tracer-zio