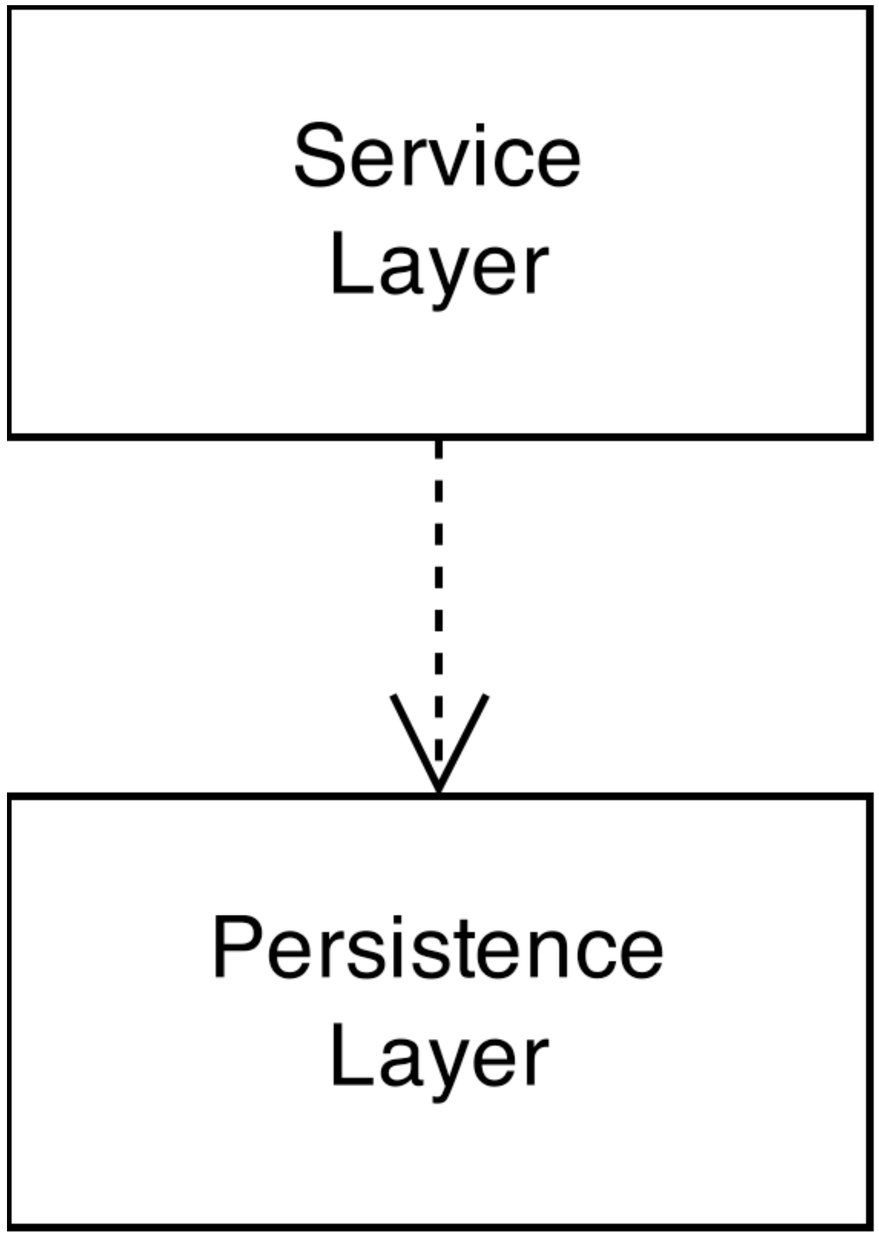
The Reader Monad for Dependency Injection

Jason Arhart (@jarhart)

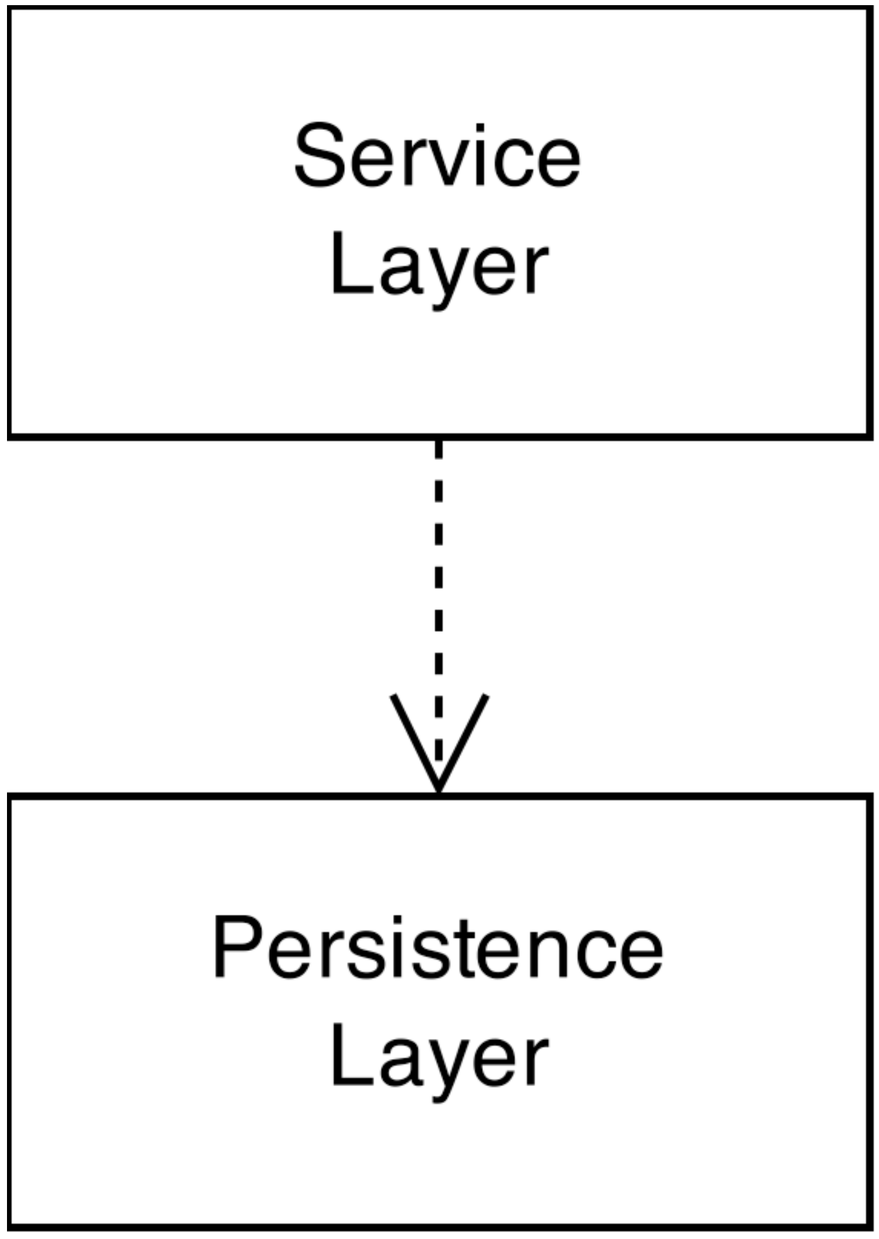
Why Dependency Injection?

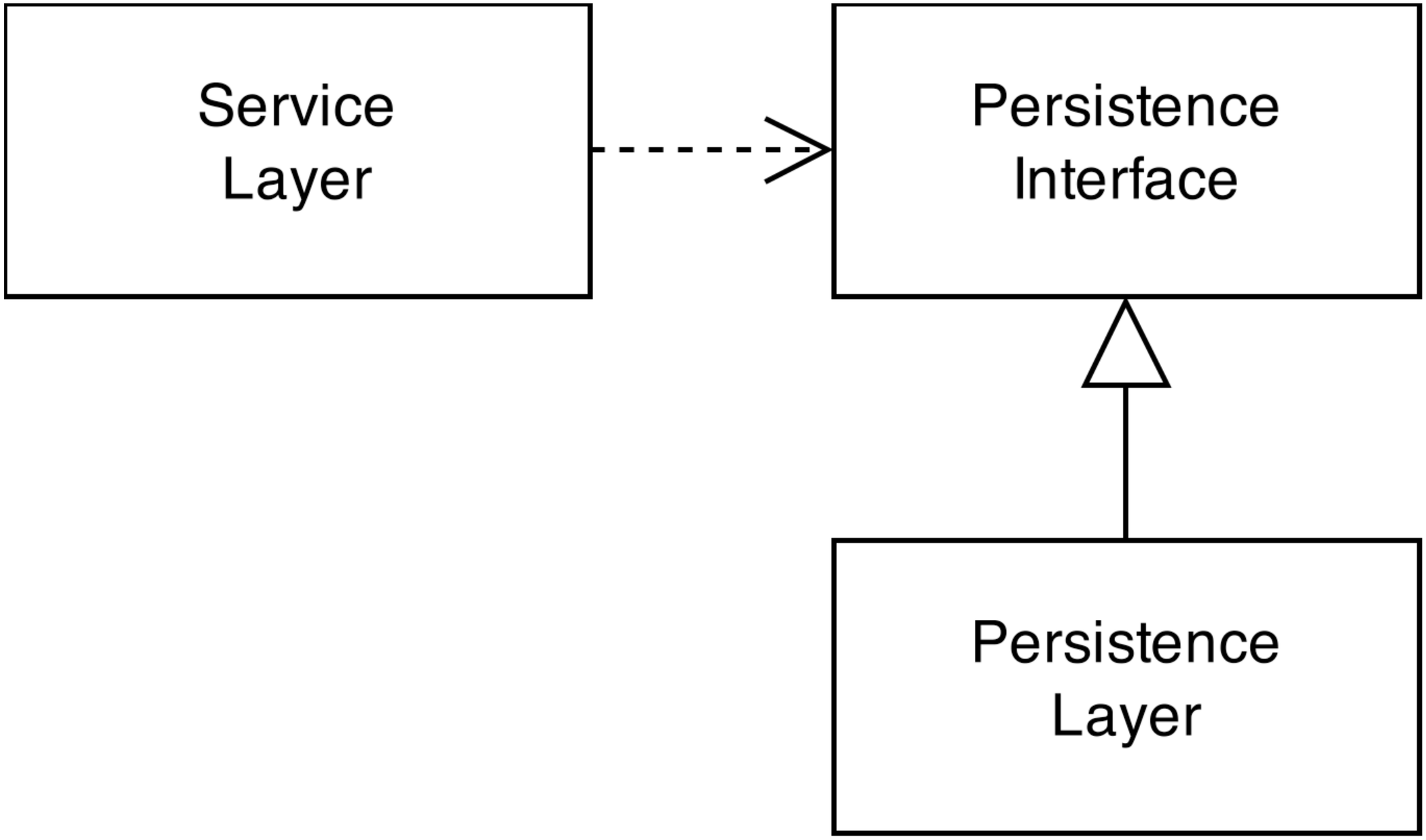

Service
Layer
Persistence
Layer

Dependency Inversion Principle

- High level modules should not depend upon low level modules. Both should depend upon abstractions.

- Abstractions should not depend upon details. Details should depend upon abstractions.


Service
Layer
Persistence
Layer


Service
Layer
Persistence
Interface
Persistence
Layer

How do the high level modules

access the low level modules?

Dependency Injection

Monads in Scala

List

```
val xs = List(1, 2, 5)
```

```
val xs = List(1, 2, 5)

xs map (x => x * x)   // => List(1, 4, 25)
```

```scala
val xs = List(1, 2, 5)

for (x <- xs) yield x * x   // => List(1, 4, 25)
```

```
class List[A] {

  def map[B](f: A => B): List[B]
}
```

```
class List[A] {

  def map[B](f: A => B): List[B]
}
```

```
class List[A] {

  def map[B](f: A => B): List[B]
}
```

```
xs.map(identity) == xs

xs.map(f).map(g) == xs.map(x => g(f(x)))
```

```scala
val xs = List(2, 5)

xs flatMap (x => List(x, x + 1))
  // => List(2, 3, 5, 6)
```

```scala
val xs = List(1, 2)

val ys = List(3, 4)

xs flatMap { x =>
  ys map (y => x * y)
}   // => List(3, 4, 6, 8)
```

```scala
val xs = List(1, 2)

val ys = List(3, 4)

for {
  x <- xs
  y <- ys
} yield x * y
```

```scala
class List[A] {

  def flatMap[B](f: A => List[B]): List[B]
}
```

```
class List[A] {

  def flatMap[B](f: A => List[B]): List[B]
}
```

```scala
class List[A] {

  def flatMap[B](f: A => List[B]): List[B]
}
```

```
xs.flatMap(x => List(f(x))) == xs.map(f)
```

```
List(x).flatMap(f) == f(x)

xs.flatMap(x => List(x)) == xs

xs.flatMap(f).flatMap(g) == xs.flatMap(f(_).flatMap(g))
```

```scala
class List[A] {

  def map[B](f: A => B): List[B]

  def flatMap[B](f: A => List[B]): List[B]
}
```

```
xs.map(identity) == x

xs.map(f).map(g) == xs.map(x => g(f(x)))

xs.flatMap(x => List(f(x))) == xs.map(f)

List(x).flatMap(f) == f(x)

xs.flatMap(List(_)) == xs

xs.flatMap(f).flatMap(g) == xs.flatMap(f(_).flatMap(g))
```

Function

```
val add2: Int => Int = x => x + 2
```

```
val add2: Int => Int = _ + 2

add2(3)    // => 5
```

```
val add2: Int => Int = _ + 2

add2(3)    // => 5
```

```scala
val add2: Int => Int = _ + 2

add2(3)    // => 5
```

```scala
def add(x: Int): Int => Int = _ + x

val add2 = add(2)

add2(3)   // => 5
```

```scala
def add(x: Int): Int => Int = _ + x

val add2 = add(2)

add2(3)   // => 5
```

```scala
def add(x: Int): Int => Int = _ + x

val add2 = add(2)

add2(3)   // => 5
```

```scala
def add(x: Int): Int => Int = _ + x

def multiplyBy(x: Int): Int => Int = _ * x

val f = add(2) andThen multiplyBy(3)

f(3)  // => 15
```

```scala
def add(x: Int): Int => Int = _ + x

def multiplyBy(x: Int): Int => Int = _ * x

val f = add(2) andThen multiplyBy(3)

f(3)  // => 15
```

```scala
def add(x: Int): Int => Int = _ + x

def multiplyBy(x: Int): Int => Int = _ * x

val f = add(2) andThen multiplyBy(3)

f(3)  // => 15
```

```
f.andThen(identity) == f

f.andThen(g).andThen(h) == f.andThen(x => h(g(x)))
```

```scala
def add(x: Int): Int => Int = _ + x

val f = add(2) andThen (_ * 3)

f(3)  // => 15
```

```
def add(x: Int): Int => Int = _ + x

val f = add(2) map (_ * 3)
```

```
def add(x: Int): Int => Int = _ + x

val f = add(2) map (_ * 3)
```

```
error: value map is not a member of Int => Int
       val f = add(2) map (_ * 3)
                      ^
```

The Reader Monad

```scala
case class Reader[A, B](run: A => B) {

  def apply(x: A): B = run(x)

  def map[C](f: B => C): Reader[A, C] =
    Reader(run andThen f)

  def flatMap[C](f: B => Reader[A, C]): Reader[A, C] =
    Reader(x => map(f)(x)(x))
}
```

```scala
import scalaz.Reader

val f = Reader[Int, Int](_ + 2)

val g = f map (_ * 3)

g(3)  // => 15
```

```scala
import scalaz.Reader

val f = Reader[Int, Int](_ + 2)

val g = for (x <- f) yield x * 3

g(3)  // => 15
```

Dependency Injection?

```
def getUser(userId: Int) =
  Reader[UserRepo, User](_.get(userId))
```

```
def getUser(userId: Int) =
  Reader[UserRepo, User](_.get(userId))
```

```scala
def getUser(userId: Int) =
  Reader[UserRepo, User](_.get(userId))

def getEmail(userId: Int) =
  for (user <- getUser(userId))
    yield user.email
```

```scala
def getUser(userId: Int) =
  Reader[UserRepo, User](_.get(userId))

def getSupervisor(userId: Int) =
  for {
    user <- getUser(userId)
    supervisor <- getUser(user.supervisorId)
  } yield supervisor
```

```scala
trait UserRepo {
  def get(userId: Int): User
  def find(email: String): User
  def update(user: User): User
}
```

```scala
object UserRepo {

  def getUser(userId: Int) =
    Reader[UserRepo, User](_.get(userId))

  def findUser(email: String) =
    Reader[UserRepo, User](_.find(email))

  def updateUser(user: User) =
    Reader[UserRepo, User](_.update(user))
}
```

```scala
object UserRepo {

  val userRepo =
    Reader[UserRepo, UserRepo](identity)

  def getUser(userId: Int) =
    userRepo map (_.get(userId))

  def findUser(email: String) =
    userRepo map (_.find(email))

  def updateUser(user: User) =
    userRepo map (_.update(user))
}
```

```scala
object UserRepo {

  val userRepo =
    Reader[UserRepo, UserRepo](identity)

  def getUser(userId: Int) =
    userRepo map (_.get(userId))

  def findUser(email: String) =
    userRepo map (_.find(email))

  def updateUser(user: User) =
    userRepo map (_.update(user))
}
```

```scala
object UserRepo {

  val userRepo =
    Reader[UserRepo, UserRepo](identity)

  def getUser(userId: Int) =
    userRepo map (_.get(userId))

  def findUser(email: String) =
    userRepo map (_.find(email))

  def updateUser(user: User) =
    userRepo map (_.update(user))
}
```

Other Dependencies?

```scala
trait Repositories {
  def userRepo: UserRepo
  def addressRepo: AddressRepo
}
```

```scala
object Repositories {

  val repositories =
    Reader[Repositories, Repositories](identity)

  val userRepo =
    repositories map (_.userRepo)

  val addressRepo =
    repositories map (_.addressRepo)
}
```

```scala
object UserRepo {
  import Repositories.userRepo

  def getUser(userId: Int) =
    userRepo map (_.get(userId))

  def findUser(email: String) =
    userRepo map (_.find(email))

  def updateUser(user: User) =
    userRepo map (_.update(user))
}
```

```
object UserRepo {
  import Repositories.userRepo

  def getUser(userId: Int) =
    userRepo map (_.get(userId))

  def findUser(email: String) =
    userRepo map (_.find(email))

  def updateUser(user: User) =
    userRepo map (_.update(user))
}
```

```scala
object UserRepo {
  import Repositories.userRepo

  def getUser(userId: Int) =
    userRepo map (_.get(userId))

  def findUser(email: String) =
    userRepo map (_.find(email))

  def updateUser(user: User) =
    userRepo map (_.update(user))
}
```

```scala
trait Env {
  def config: Configuration
  def emailService: EmailService
  def repositories: Repositories
}
```

```scala
object Env {
  val env = Reader[Env, Env](identity)

  val config =
    env map (_.config)

  val emailService =
    env map (_.emailService)

  val repositories =
    env map (_.repositories)
}
```

```scala
object Repositories {
  import Env.repositories

  val userRepo =
    repositories map (_.userRepo)

  val addressRepo =
    repositories map (_.addressRepo)
}
```

```
object Repositories {
  import Env.repositories

  val userRepo =
    repositories map (_.userRepo)

  val addressRepo =
    repositories map (_.addressRepo)
}
```

```scala
object UserRepo {
  import Repositories.userRepo

  def getUser(userId: Int) =
    userRepo map (_.get(userId))

  def findUser(email: String) =
    userRepo map (_.find(email))

  def updateUser(user: User) =
    userRepo map (_.update(user))
}
```

```scala
object Env {
  val env = Reader[Env, Env](identity)

  val config =
    env map (_.config)

  val emailService =
    env map (_.emailService)

  val repositories =
    env map (_.repositories)
}
```

```scala
object UserService {

  def getEmail(userId: Int) =
    for {
      user <- UserRepo.getUser(userId)
    } yield user.email

  def findAddress(email: String) =
    for {
      user <- UserRepo.findUser(email)
      address <- AddressRepo.getAddress(user.id)
    } yield address
}
```

```scala
trait ConfigurationComponent {
  def config: Configuration
}

trait EmailServiceComponent {
  def emailService: EmailService
}

trait RepositoriesComponent {
  def repositories: Repositories
}
```

```scala
trait UserRepoComponent {
  def userRepo: UserRepo
}

trait AddressRepoComponent {
  def addressRepo: AddressRepo
}
```

```
trait Env extends
  ConfigurationComponent with
  EmailServiceComponent with
  RepositoriesComponent

trait RepositoriesComponent extends
  UserRepoComponent with
  AddressRepoComponent
```

```
object productionEnv extends Env
  with PlayConfigComponent
  with PlayEmailServiceComponent
  with MongoRepositoriesComponent

object testEnv extends Env
  with MockConfigComponent
  with MockEmailServiceComponent
  with MockRepositoriesComponent
```

Other Monads

```scala
trait UserRepo {
  def get(userId: Int): User
  def find(email: String): User
  def update(user: User): User
}
```

```scala
trait UserRepo {
  def get(userId: Int): Future[User]
  def find(email: String): Future[User]
  def update(user: User): Future[User]
}
```

```
def getEmail(userId: Int) =
  for (user <- getUser(userId))
    yield user.email
```

```scala
def getEmail(userId: Int) =
  for (userFuture <- getUser(userId))
    yield userFuture map (_.email)
```

```scala
def findAddress(email: String) =
  for {
    user <- UserRepo.findUser(email)
    address <- AddressRepo.getAddress(user.id)
  } yield address
```

```scala
def findAddress(email: String) =
  Env.env map { env =>
    for {
      user <- UserRepo.findUser(email).run(env)
      address <- AddressRepo.getAddress(user.id).run(env)
    } yield address
  }
```

```scala
def findAddress(email: String) =
  Env.env map { env =>
    for {
      user <- UserRepo.findUser(email).run(env)
      address <- AddressRepo.getAddress(user.id).run(env)
    } yield address
  }
```

```scala
def findAddress(email: String) =
  Env.env map { env =>
    for {
      user <- UserRepo.findUser(email).run(env)
      address <- AddressRepo.getAddress(user.id).run(env)
    } yield address
  }
```

Monad Transformers

```
def findUser(email: String): Reader[Env, Future[User]]
```

```scala
class Reader[Env, Future[A]] {

  def map[B](f: Future[A] => B):
    Reader[Env, Future[B]]

  def flatMap[B](f: Future[A] => Reader[Future[B]]):
    Reader[Env, Future[B]]
}
```

```scala
class Reader[Env, Future[A]] {

  def map[B](f: Future[A] => B):
    Reader[Env, Future[B]]

  def flatMap[B](f: Future[A] => Reader[Future[B]]):
    Reader[Env, Future[B]]
}
```

```scala
class Something[A] {

  def map[B](f: A => B):
    Something[B]

  def flatMap[B](f: A => Something[B]):
    Something[B]
}
```

```
class Something[A] {

  def map[B](f: A => B):
    Something[B]

  def flatMap[B](f: A => Something[B]):
    Something[B]
}
```

```
def findUser(email: String): Reader[Env, Future[User]]
```

```
def findUser(email: String): ReaderT[Future, Env, User]
```

```scala
class ReaderT[Future, Env, A] {

  def map[B](f: A => B):
    ReaderT[Future, Env, B]

  def flatMap[B](f: A => ReaderT[Future, Env, B]):
    ReaderT[Future, Env, B]
}
```

```scala
class ReaderT[Future, Env, A] {

  def map[B](f: A => B):
    ReaderT[Future, Env, B]

  def flatMap[B](f: A => ReaderT[Future, Env, B]):
    ReaderT[Future, Env, B]
}
```

```scala
def findAddress(email: String) =
  for {
    user <- UserRepo.findUser(email)
    address <- AddressRepo.getAddress(user.id)
  } yield address
```

```scala
import scalaz.ReaderT

val getUser(userId: Int) =
  ReaderT[Future, UserRepo, User] { repo =>
    repo.get(userId))
  }
```

```scala
import scalaz.ReaderT

val getUser(userId: Int) =
  ReaderT[Future, UserRepo, User] { repo =>
    repo.get(userId))
  }
```

```
error: not found: value ReaderT
       ReaderT[Option, Int, Int] { i =>
       ^
```

```scala
import scalaz.Kleisli

val getUser(userId: Int) =
  Kleisli[Future, UserRepo, User] { repo =>
    repo.get(userId))
  }
```

```scala
import scalaz.{Kleisli, ReaderT}

type Query[A] = ReaderT[Future, Env, A]

object Query {

  def apply[A](run: Env => Future[A]): Query[A] =
    Kleisli[Future, Env, A](run)

  def lift[A](reader: Reader[Env, Future[A]]) =
    Query(reader.run)
}
```

```scala
import scalaz.{Kleisli, ReaderT}

type Query[A] = ReaderT[Future, Env, A]

object Query {

  def apply[A](run: Env => Future[A]): Query[A] =
    Kleisli[Future, Env, A](run)

  def lift[A](reader: Reader[Env, Future[A]]) =
    Query(reader.run)
}
```

```scala
import scalaz.{Kleisli, ReaderT}

type Query[A] = ReaderT[Future, Env, A]

object Query {

  def apply[A](run: Env => Future[A]): Query[A] =
    Kleisli[Future, Env, A](run)

  def lift[A](reader: Reader[Env, Future[A]]) =
    Query(reader.run)
}
```

```scala
import scalaz.{Kleisli, ReaderT}

type Query[A] = ReaderT[Future, Env, A]

object Query {

  def apply[A](run: Env => Future[A]): Query[A] =
    Kleisli[Future, Env, A](run)

  def lift[A](reader: Reader[Env, Future[A]]) =
    Query(reader.run)
}
```

```scala
object UserRepo {
  import Repositories.userRepo

  def getUser(userId: Int)(implicit ec: ExecutionContext) =
    Query.lift(userRepo map (_.get(userId)))

  def findUser(email: String)(implicit ec: ExecutionContext) =
    Query.lift(userRepo map (_.find(email)))

  def updateUser(user: User)(implicit ec: ExecutionContext) =
    Query.lift(userRepo map (_.update(user)))
}
```

```scala
object UserService {
  import scalaz.contrib.std.scalaFuture._

  def getEmail(userId: Int)(implicit ec: ExecutionContext) =
    for {
      user <- UserRepo.getUser(userId)
    } yield user.email

  def findAddress(email: String)(implicit ec: ExecutionContext) =
    for {
      user <- UserRepo.findUser(email)
      address <- AddressRepo.getAddress(user.id)
    } yield address
}
```

```scala
object UserService {
  import scalaz.contrib.std.scalaFuture._

  def getEmail(userId: Int)(implicit ec: ExecutionContext) =
    for {
      user <- UserRepo.getUser(userId)
    } yield user.email

  def findAddress(email: String)(implicit ec: ExecutionContext) =
    for {
      user <- UserRepo.findUser(email)
      address <- AddressRepo.getAddress(user.id)
    } yield address
}
```

```scala
object UserService {
  import scalaz.contrib.std.scalaFuture._

  def getEmail(userId: Int)(implicit ec: ExecutionContext) =
    for {
      user <- UserRepo.getUser(userId)
    } yield user.email

  def findAddress(email: String)(implicit ec: ExecutionContext) =
    for {
      user <- UserRepo.findUser(email)
      address <- AddressRepo.getAddress(user.id)
    } yield address
}
```

Dependency Injection?

```scala
import scalaz.Reader

abstract class EnvController(env: Env) extends Controller {

  def run[A](r: Reader[Env, A]): A = r.run(env)

  def run[A](query: Query[A]): Future[A] = query.run(env)
}
```

```scala
import scalaz.Reader

abstract class EnvController(env: Env) extends Controller {

  def run[A](r: Reader[Env, A]): A = r.run(env)

  def run[A](query: Query[A]): Future[A] = query.run(env)
}
```

```scala
import scalaz.Reader

abstract class EnvController(env: Env) extends Controller {

  def run[A](r: Reader[Env, A]): A = r.run(env)

  def run[A](query: Query[A]): Future[A] = query.run(env)
}
```

```scala
class Users(env: Env) extends EnvController(env) {
  import scalaz.contrib.std.scalaFuture._
  import Execution.Implicits.defaultContext

  def show(id: String) = Action.async { request =>
    for (user <- run(UserService.getUser(id)))
      yield Ok(views.html.user(user))
  }
}
```

```scala
class Users(env: Env) extends EnvController(env) {
  import scalaz.contrib.std.scalaFuture._
  import Execution.Implicits.defaultContext

  def show(id: String) = Action.async { request =>
    for (user <- run(UserService.getUser(id)))
      yield Ok(views.html.user(user))
  }
}
```

```scala
object Global extends GlobalSettings {

  private object env extends Env
    with PlayConfigComponent
    with PlayEmailServiceComponent
    with MongoRepositoriesComponent

  override def getControllerInstance[A](c: Class[A]) =
    c.getConstructor(classOf[Env]).newInstance(env)
}
```

routes

```
GET /user/:id   @controllers.Users.show(id: String)
```

routes

```
GET /user/:id   @controllers.Users.show(id: String)
```

Testing

```scala
trait MockRepositoriesComponent extends RepositoriesComponent {
  object repos extends Repos
    with MockUserRepoComponent
    with MockAddressRepoComponent
}

trait MockUserRepoComponent extends UserRepoComponent {
  val userRepo = mock[UserRepo]
}

trait MockAddressRepoComponent extends AddressRepoComponent {
  val addressRepo = mock[AddressRepo]
}
```

```scala
class MockEnv extends Env
  with MockConfigComponent
  with MockEmailServiceComponent
  with MockRepositoriesComponent

trait MockConfigComponent extends ConfigComponent {
  val config = mock[Config]
}

trait MockEmailServiceComponent extends EmailServiceComponent {
  val emailService = mock[EmailService]
}
```

```scala
class MockEnv extends Env
  with MockConfigComponent
  with MockEmailServiceComponent
  with MockRepositoriesComponent

trait MockConfigComponent extends ConfigComponent {
  val config = mock[Config]
}

trait MockEmailServiceComponent extends EmailServiceComponent {
  val emailService = mock[EmailService]
}
```

```scala
trait TestEnv {
  import Helpers._

  def env: Env

  def config = env.config
  def emailSerivce = env.emailSerivce
  def repositories = env.repositories
  def userRepo = repositories.userRepo
  def addressRepo = repositories.addressRepo

  def await[A](query: Query[A]): A =
    Helpers.await(query.run(env))
}
```

```scala
trait TestEnv {
  import Helpers._

  def env: Env

  def config = env.config
  def emailSerivce = env.emailSerivce
  def repositories = env.repositories
  def userRepo = repositories.userRepo
  def addressRepo = repositories.addressRepo

  def await[A](query: Query[A]): A =
    Helpers.await(query.run(env))
}
```

```scala
trait TestEnv {
  import Helpers._

  def env: Env

  def config = env.config
  def emailSerivce = env.emailSerivce
  def repositories = env.repositories
  def userRepo = repositories.userRepo
  def addressRepo = repositories.addressRepo

  def await[A](query: Query[A]): A =
    Helpers.await(query.run(env))
}
```

```scala
class UserServiceSpec extends FreeSpec {

  trait UserServiceTesting extends TestEnv {

    val env = new MockEnv

    val testEmail = // ...
    val testUser = // ...
    val testAddress = // ...
  }

  // ...
}
```

```scala
class UserServiceSpec extends FreeSpec {

  trait UserServiceTesting extends TestEnv {

    val env = new MockEnv

    val testEmail = // ...
    val testUser = // ...
    val testAddress = // ...
  }

  // ...
}
```

```scala
class UserServiceSpec extends FreeSpec {

  trait UserServiceTesting extends TestEnv {

    val env = new MockEnv

    val testEmail = // ...
    val testUser = // ...
    val testAddress = // ...
  }

  // ...
}
```

```scala
class UserServiceSpec extends FreeSpec {

  trait UserServiceTesting extends TestEnv {

    val env = new MockEnv

    val testEmail = // ...
    val testUser = // ...
    val testAddress = // ...
  }

  // ...
}
```

```
"UserService" - {

  "findAddress finds an address" in new UserServiceTesting {

    when(userRepo.findUser(email))
      .thenReturn(testUser)

    when(addressRepo.getAddress(testUser.id))
      .thenReturn(testAddress)

    assert(
      await(UserService.findAddress(testEmail))
        == testAddress
    )
  }
}
```

```
"UserService" - {

  "findAddress finds an address" in new UserServiceTesting {

    when(userRepo.findUser(email))
      .thenReturn(testUser)

    when(addressRepo.getAddress(testUser.id))
      .thenReturn(testAddress)

    assert(
      await(UserService.findAddress(testEmail))
        == testAddress
    )
  }
}
```

```
"UserService" - {

  "findAddress finds an address" in new UserServiceTesting {

    when(userRepo.findUser(email))
      .thenReturn(testUser)

    when(addressRepo.getAddress(testUser.id))
      .thenReturn(testAddress)

    assert(
      await(UserService.findAddress(testEmail))
        == testAddress
    )
  }
}
```

```
"UserService" - {

  "findAddress finds an address" in new UserServiceTesting {

    when(userRepo.findUser(email))
      .thenReturn(testUser)

    when(addressRepo.getAddress(testUser.id))
      .thenReturn(testAddress)

    assert(
      await(UserService.findAddress(testEmail))
        == testAddress
    )
  }
}
```