



IO monad & Error management

From exceptions to Cats MTL

Bonjour!

My name is **Guillaume Bogard**. I'm a Scala Developer @Linkvalue.

I love functional programming, roller-coasters, and Age of Empires.

...

Also mechanical keyboards 

You can follow me on Twitter @bogardguillaume.

Let's talk about IO

Why do we need IO anyway ?

- **IO**s are **programs as values**.

```
def getUser(id: String): IO[User]
```

- They turn impure programs into referentially transparent values.
- They reveal the presence of sneaky side effects 🐍
- They **compose** :

```
def getFavoritePet(user: User): IO[Pet]  
val marksFavoritePet: IO[Pet] = getUser("Mark") flatMap getFavoritePet
```

When things go wrong

Cats effect allows to raise `Throwable`s inside the `IO` context, propagate them across all the `IO` chain, and recover them later.

```
val failedIO: IO[Int] = IO.raiseError(new Exception("Boom"))

failedIO.recoverWith({
  case e => IO {
    logger.error("Something went wrong", e)
    42
  }
}).unsafeRunSync() // => 42
```

Use case : Modeling an authentication flow

I want to authenticate a user using a name and a password. I need to return the user's information.

- I need to find the user information (maybe from a database)
- Check that the password is valid by comparing hashes
- I need to check that the user subscription is valid (maybe through a billing service)
- And the user must not be banned from our service

I could do all that by composing several **IO** together.

```
case object WrongUserName extends RuntimeException("No user with that name")
case object WrongPassword extends RuntimeException("Wrong password")
case class ExpiredSubscription(expirationDate: Date) extends
  RuntimeException("Expired subscription")
case object BannedUser extends RuntimeException("User is banned")

def findUserByName(username: String): IO[User] = ???
def checkPassword(user: User, password: String): IO[Unit] = ???
def checkSubscription(user: User): IO[Unit] = ???
def checkUserStatus(user: User): IO[Unit] = ???

def authenticate(userName: String, password: String): IO[User] =
  for {
    user <- findUserByName(userName)
    _ <- checkPassword(user, password)
    _ <- checkSubscription(user)
    _ <- checkUserStatus(user)
  } yield user
```

```
authenticate("john.doe", "foo.bar")
  .flatMap(user => IO {
    println(s"Success! $user")
  })
  .recoverWith({
    case WrongUserName => IO { /* Do stuff ... */ }
    case WrongPassword => IO { /* Do stuff ... */ }
    case ExpiredSubscription(date) => IO { /* Do stuff ... */ }
    case BannedUser => IO { /* Do stuff ... */ }
    case _ => IO {
      println("Another exception was caught !")
    }
  })
})
```


The issues with Exception

- ✗ Exceptions are invisible
- ✗ They must be explicitly recovered
- ✗ They must be explicitly documented
- ✗ They can be ambiguous :
 - `IO`s only raise and recover `Throwable`s. Type-wise they don't distinguish between `java.util.concurrent.TimeoutException` & `AuthenticationException`.
- ✗ You get no proper exhaustivity check

This is a lie

```
def authenticate(userName: String, password: String): IO[User]
```

The method's signature doesn't convey anything about possible error cases.

Unhandled errors will be propagated across the whole application once the IO is ran like good ol' Java when you forget to `try/catch`.

Requires more reading and more testing.

Make exceptions exceptional again

 Opinionated statements incoming

Domain errors are not exceptions

- Domain errors are documented edge cases that can happen as part of the user experience.
e.g.: The user hasn't paid their subscription
- Exceptions should be reserved for unexpected, purely technical failures.
e.g.: The database server is unreachable.

Exceptions should be propagated to the upper levels of the app and actively monitored.

Modeling errors using an ADT






```
sealed trait AuthenticationError  
case object WrongUserName extends AuthenticationError  
case object WrongPassword extends AuthenticationError  
case class ExpiredSubscription(expirationDate: Date) extends AuthenticationError  
case object BannedUser extends AuthenticationError
```

Using it with **Either**

```
def authenticate(userName: String, password: String): IO[Either[AuthenticationError, User]]
```

What we've achieved

```
def authenticate(userName: String, password: String): IO[Either[AuthenticationError, User]]
```

-  Side effects are obvious
-  Domain errors are visible and cannot be forgotten
-  Technical errors can still be raised and recovered
-  No need to check the documentation for unhandled edge cases
-  We have clearly distinct error families

But we've lost something very important along the way

This does not compose anymore !

✗ This does not compile

```
def findUserByName(username: String): IO[Either[AuthenticationError, User]] = ???
def checkPassword(user: User, password: String): IO[Either[AuthenticationError, Unit]] = ???
def checkSubscription(user: User): IO[Either[AuthenticationError, Unit]] = ???
def checkUserStatus(user: User): IO[Either[AuthenticationError, Unit]] = ???

def authenticate(userName: String, password: String): IO[Either[AuthenticationError, User]] =
  for {
    user <- findUserByName(userName)
    _ <- checkPassword(user, password)
    _ <- checkSubscription(user)
    _ <- checkUserStatus(user)
  } yield user
```

We can't compose many `IO[Either[A, B]]` together. We must handle the errors explicitly.

```
def authenticate(userName: String, password: String): IO[Either[AuthenticationError, User]] =  
  findUserByName(userName).flatMap({  
    case Right(user) => checkPassword(user, password).flatMap({  
      case Right(_) => checkSubscription(user).flatMap({  
        case Right(_) => checkUserStatus(user).map(_._map(_ => user))  
        case Left(err) => IO.pure(Left(err))  
      })  
      case Left(err) => IO.pure(Left(err))  
    })  
    case Left(err) => IO.pure(Left(err))  
  })
```


A dramatic scene from the Transformers movie franchise. On the left, Optimus Prime, a blue and red Autobot, stands tall. On the right, Megatron, a yellow and black Decepticon, is shown in a more dynamic, almost falling or lunging pose. They are positioned in front of the Great Pyramids of Giza under a cloudy, blue sky. The text "Monad Transformers" is centered over the image in a white, sans-serif font.

Monad Transformers

Monad transformers (`OptionT`, `IorT`, `EitherT`, `ReaderT` and `WriterT`) add new behavior, also referred to as an *effect*, to an underlying monad while preserving compositionality.

E.g : `OptionT[F, A]` creates a new monad which adds the effect of absence to a monad `F`.

From `F[Either[A, B]]` to `EitherT[F, A, B]`

- “ `EitherT[F, L, R]` is a light wrapper around `F[Either[A, B]]` that makes it easy to compose `Either`s and `F`s together. ”
- It has bidirectional transformation from/to `F[Either[A, B]]` via the `apply` and `value` methods respectively.

```
val a: IO[Either[AuthenticationError, User]] = ???  
val b: EitherT[IO, AuthenticationError, User] = EitherT(a)
```

`EitherT` forms a compound monad out of two out of some `F` type, and the `Either` monad.

When `F` is a monad, such as `IO`, `EitherT` will also form a monad, making it easy to compose the compound monad using `map` and `flatMap`.

`OptionT` does the same for `Option`.

`IorT` does the same for `Ior` (an inclusive-or relationship between two data types)

Rewriting our authentication method




```
def findUserByName(username: String): EitherT[IO, AuthenticationError, User] = ???
def checkPassword(user: User, password: String): EitherT[IO, AuthenticationError, Unit] = ???
def checkSubscription(user: User): EitherT[IO, AuthenticationError, Unit] = ???
def checkUserStatus(user: User): EitherT[IO, AuthenticationError, Unit] = ???

def authenticate(userName: String, password: String): EitherT[IO, AuthenticationError, User] =
  for {
    user <- findUserByName(userName)
    _ <- checkPassword(user, password)
    _ <- checkSubscription(user)
    _ <- checkUserStatus(user)
  } yield user
```

`EitherT` will short-circuit computation on the first encountered error, a pattern sometimes called *Railway-oriented programming* 🚂

Look how far we've come!

We've met all our goals :

-  Side effects visible
-  `IO`s and errors can be composed, railway style
-  We still get the benefit of having two distinct error channels :
 - Exceptions thrown inside the `IO` for purely technical failures
 - The *Left* of the `Either` for business-related errors

This way we can **fail fast** on technical failures and easily provide good feedback to the user for business edge cases.

We've won, let's have a drink! 🍺🍻

But, wait, what about Cats MTL then ?

Challenges yet to address

- What about nested transformers ? What if I want to model mutable state **and** potential absence for example ?
- What about type inference and expressivity ?
 - Nesting monad transformers requires many type parameters and type lambdas. The more you nest, the worst inference gets!

Can you guess what this code does ?

```
// Retrieves document from a super secure data store
def getDocument: IO[SecretDocument] = ???

type Count = Int
val readSecretDocument: User => EitherT[IO, String, SecretDocument] = {
  val state: StateT[ReaderT[IO, User, *], Count, Either[String, SecretDocument]] =
    StateT[ReaderT[IO, User, *], Int, Either[String, SecretDocument]](currentAttemptsCount =>
      ReaderT[IO, User, (Count, Either[String, SecretDocument])](user =>
        if (currentAttemptsCount >= 3) IO.pure((currentAttemptsCount, Left("Max attempts exceeded")))
        else if (user.isAdmin) getDocument.map(doc => (currentAttemptsCount, Right(doc)))
        else IO.pure((currentAttemptsCount + 1, Left("Access denied")))
      )
    )
  state.run(0).map(_._2).mapF(EitherT(_)).run
}
```

Me neither.

When we need to combine effects (e.g short-circuiting AND mutable state), monad transformers only get us so far.

Scala's inference system can't keep up with nested monad transformers stack, requiring a ridiculous amount of boilerplate to get simple things done.

The idea of Cats MTL

Monad transformers encode some *effect*, e.g. :

- `EitherT` encodes the effect of short-circuiting on error
- `ReaderT` (i.e. `Kleisli`) encode the effect of accessing a read-only value from a context, and producing a value from it

Cats MTL encodes these effects, among others, in **type classes**. It gives the ability to combine effects together, without the drawback of bad inference.

How would one encode the effect of raising errors ?

```
def readSecretDocument[F[_] : Applicative](user: User)
  (implicit F: FunctorRaise[F, String]): F[SecretDocument] =
  if (user.isAdmin) SecretDocument().pure[F]
  else F.raise("Access Denied!")
```

We've turned our `EitherT` into a generic `F[_]` `Applicative`. All we know about this `F[_]` is that there is an instance of `FunctorRaise` defined for it.

We'll need to provide a concrete implementation of `F` to run the program.

What about recovering ?

```
def getDocumentContent[F[_] : Applicative](user: User)
  (implicit A: ApplicativeHandle[F, String]): F[String] =
  readSecretDocument[F](user)
    .map(_.content)
    .handle[String](_ => "Default content")
```

`ApplicativeHandle` extends `FunctorRaise` with the ability to handle errors.

We still get a dedicated channel for technical failures

By adding a context bound on `MonadError`, we can raise `Exceptions` in our IOs, and segregate technical failures from domain errors 🙌🙌

```
def findUserByName[F[_]](name: String)
  (implicit AE: ApplicativeError[F, Throwable]) = {
    AE.raiseError(new RuntimeException("Database not reachable!"))
  }
```

Implementing the **authenticate** method

```
def checkPassword[F[_]](user: User, password: String)
  (implicit F: FunctorRaise[F, AuthenticationError]): F[Unit] =
  F.raise(WrongPassword)

def checkSubscription[F[_]](user: User): F[User] = ???
def checkUserStatus[F[_]](user: User): F[User] = ???

def authenticate[F[_]](userName: String, password: String)
  (implicit F: FunctorRaise[F, AuthenticationError], AE: MonadError[F, Throwable]): F[User] =
  for {
    user <- findUserByName[F](userName)
    _ <- checkPassword[F](user, password)
    _ <- checkSubscription[F](user)
    _ <- checkUserStatus[F](user)
  } yield user
```


Interpreting the program

It's time to provide a concrete implementation for the type classes we used.

```
object Main extends App {  
  type E[A] = EitherT[IO, AuthenticationError, A]  
  authenticate[F]("john.doe", "123456")  
}
```

Summing up

- Shit happens, we need to handle it carefully
- Don't let technical details mess up your domain
 - Monad transformers let you add effects to existing monads, to create more badass monads
 - Cats MTL gives you this without the syntax headache
- Challenge your approach, there are plenty of error management strategies out there! (e.g. ZIO)
Cats MTL is cool, but you might not need it!
- Have fun

Thank you!

Keep calm and curry on 🎸

This talk is on Github :

gbogard/cats-mtl-talk