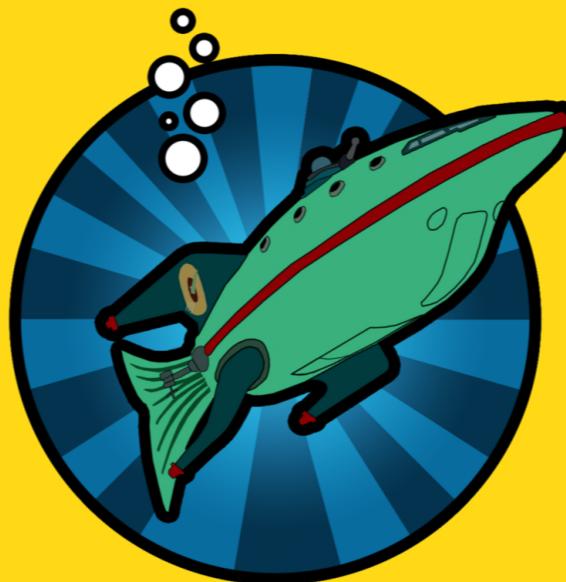


Prototyping the Future with Functional Scala



Mike Kotsur
ScalaUA 2020



“Becomes very happy up in the mountains or writing nice and stable code in a functional language without runtime bugs.”



[mkotsur](https://github.com/mkotsur)



[@s_fcopy](https://twitter.com/@s_fcopy)



medium.com/@mkotsur



Mike Kotsur @s_fcopy
Last 5 years of my career look like this:

- 3 years of amazing R&D teamwork with lots of freedom of tech choice and techniques; This is where I first got to try **#functionalscala** with **#cats**, **#typescript** with ReactJS, redux and **#serverless**.

8:16 PM · Dec 8, 2019 · Twitter for iPhone

View Tweet activity

1 Retweet

Q T L H U

Mike Kotsur @s_fcopy · Dec 8
Replying to @s_fcopy
I attempted to model drivers to **#firebase** with FP patterns of my own creation based on intuitive patterns inspired by Java experience. They were full of OOP stuff, for sure.

 mkotsur/firebase-client-scala
Scala REST client for Firebase. Contribute to mkotsur/firebase-client-scala development by ...
github.com

Q 1 T L H U

Mike Kotsur @s_fcopy · Dec 8
After that, I made a micro library for writing AWS lambdas in idiomatic Scala.
github.com/mkotsur/aws-la...

The cool thing about it is that it was used not only internally, but I also got some feedback and pull requests. Real people used it!

We'll cover

- 🚴 What prototyping is all about;
- ⚡ Data Exchange project challenges;
- 🎉💀 9 Lessons learned;
- 💪 How modern FP Scala libs help.

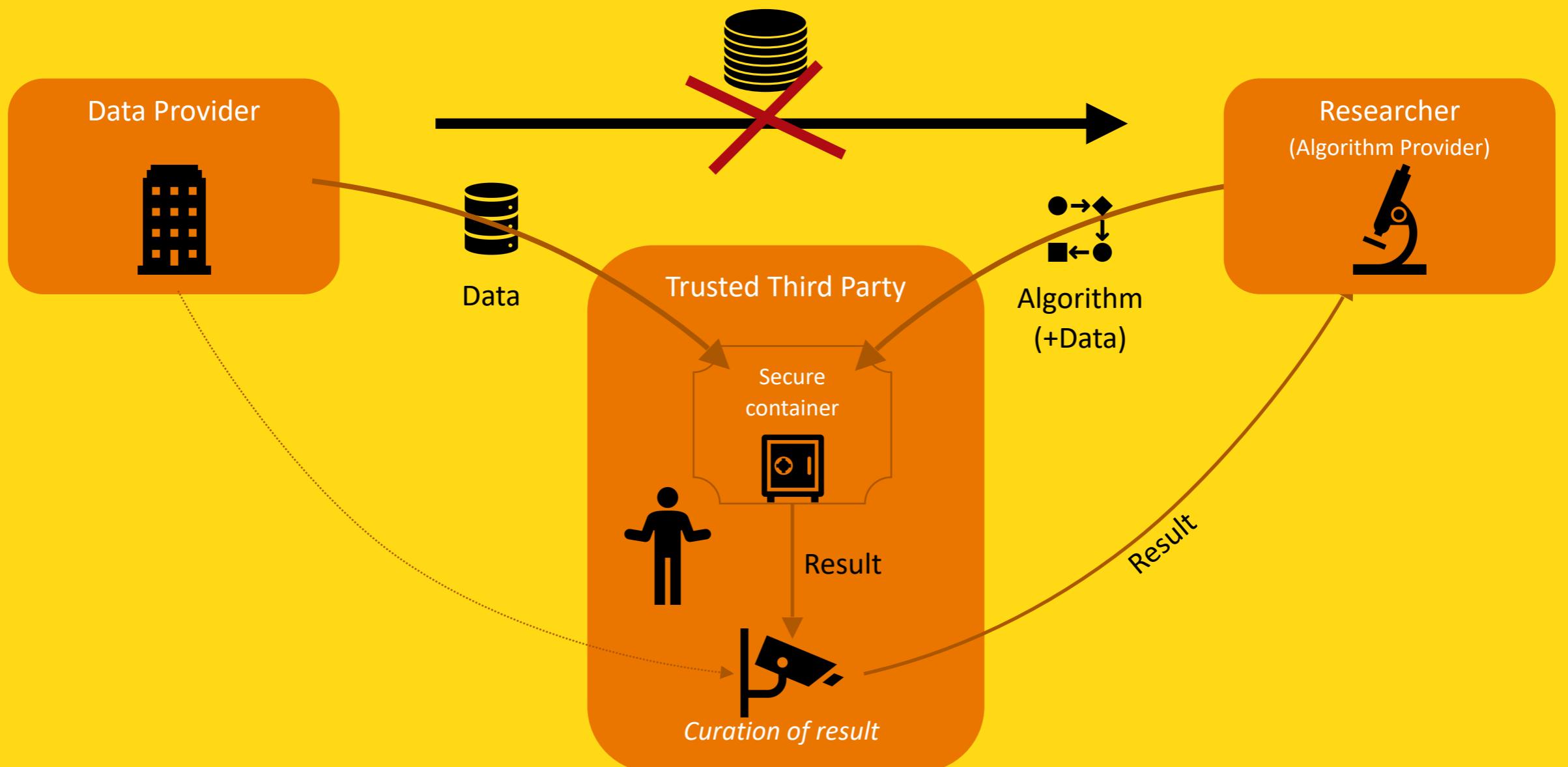


Software Prototyping

- Testing a business idea with the minimal functionality;
- Trying to make as much **Feedback => Improvement** iterations as possible within limited time;
- Difficult tech choices: quality / speed.



Data Exchange



More info:

<https://www.surf.nl/en/data-exchange-trusted-data-sharing>



Lesson 0: Python is great... in scripts and notebooks



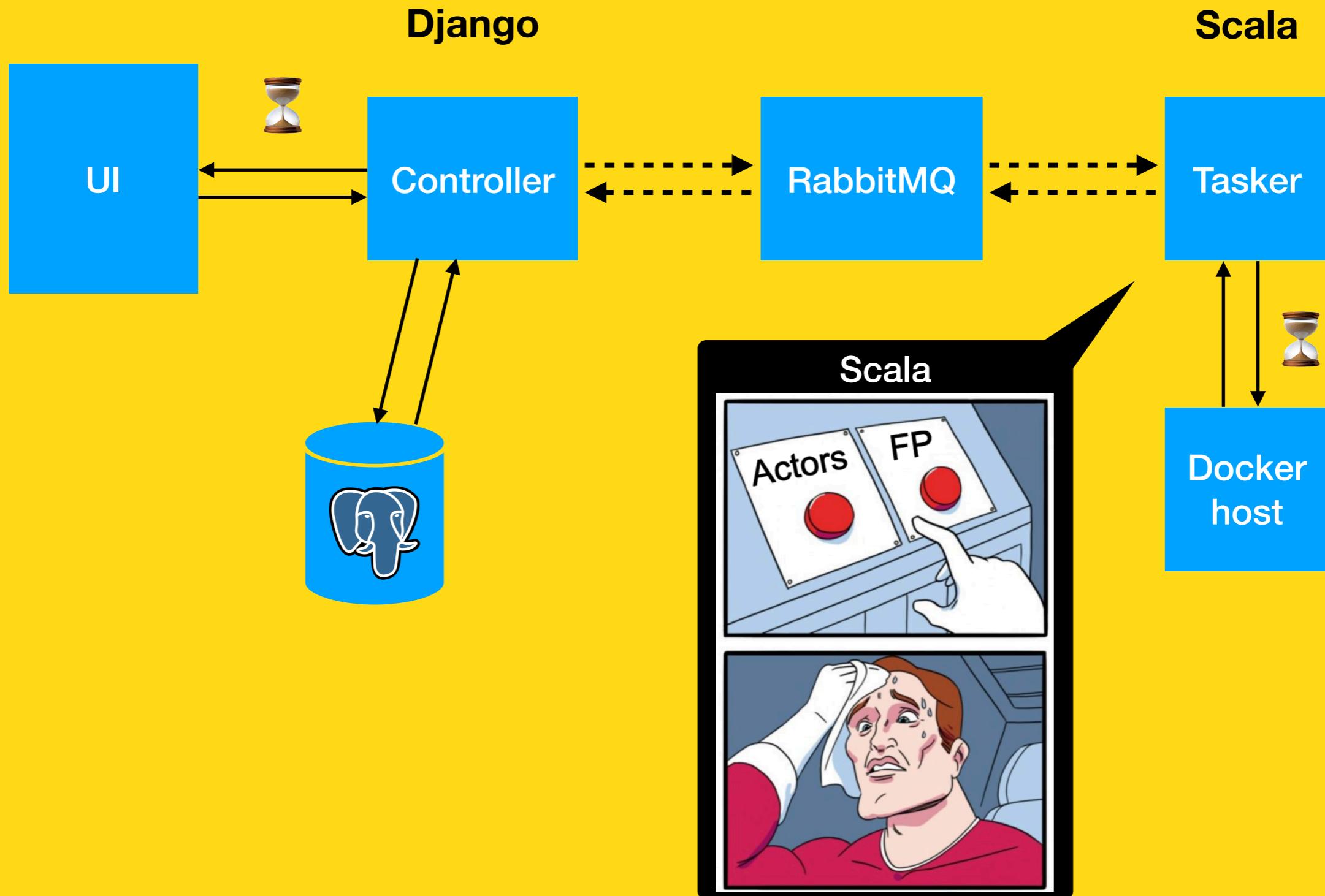
```
>>> import this  
The Zen of Python, by Tim Peters
```

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.

Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way
to do it.
Although that way may not be obvious at first unless you're
Dutch.
Now is better than never.
Although never is often better than **right** now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good
idea.
Namespaces are one honking great idea -- let's do more of
those!

Architecture v0.2



Lesson 1:

Future[T]

is a thing of the past



```
val f0: Future[Int] = Future { ??? }
def times2(i: Int): Int = ???

// times2(f0) doesn't compile

f0.map(times2)(ec)

f0.onComplete {
  case Failure(exception) => ???
  case Success(value)      => ???
}

f0.recover {
  case e: Throwable => ???
}

Await.ready(f0, Duration.Inf)

// Do other things
```

Lesson 1:

Future[T]

is a thing

of the

past

```
val f: Future[Int] = Future { 1 }
val g: Future[Int] = Future { 2 }
for {
    i ← f
    j ← g
} yield i + j
```

```
for {
    i ← Future { 1 }
    j ← Future { 2 }
} yield i + j
```

Lesson 1:

Future[T]

is a thing of the past

Eagerly evaluated

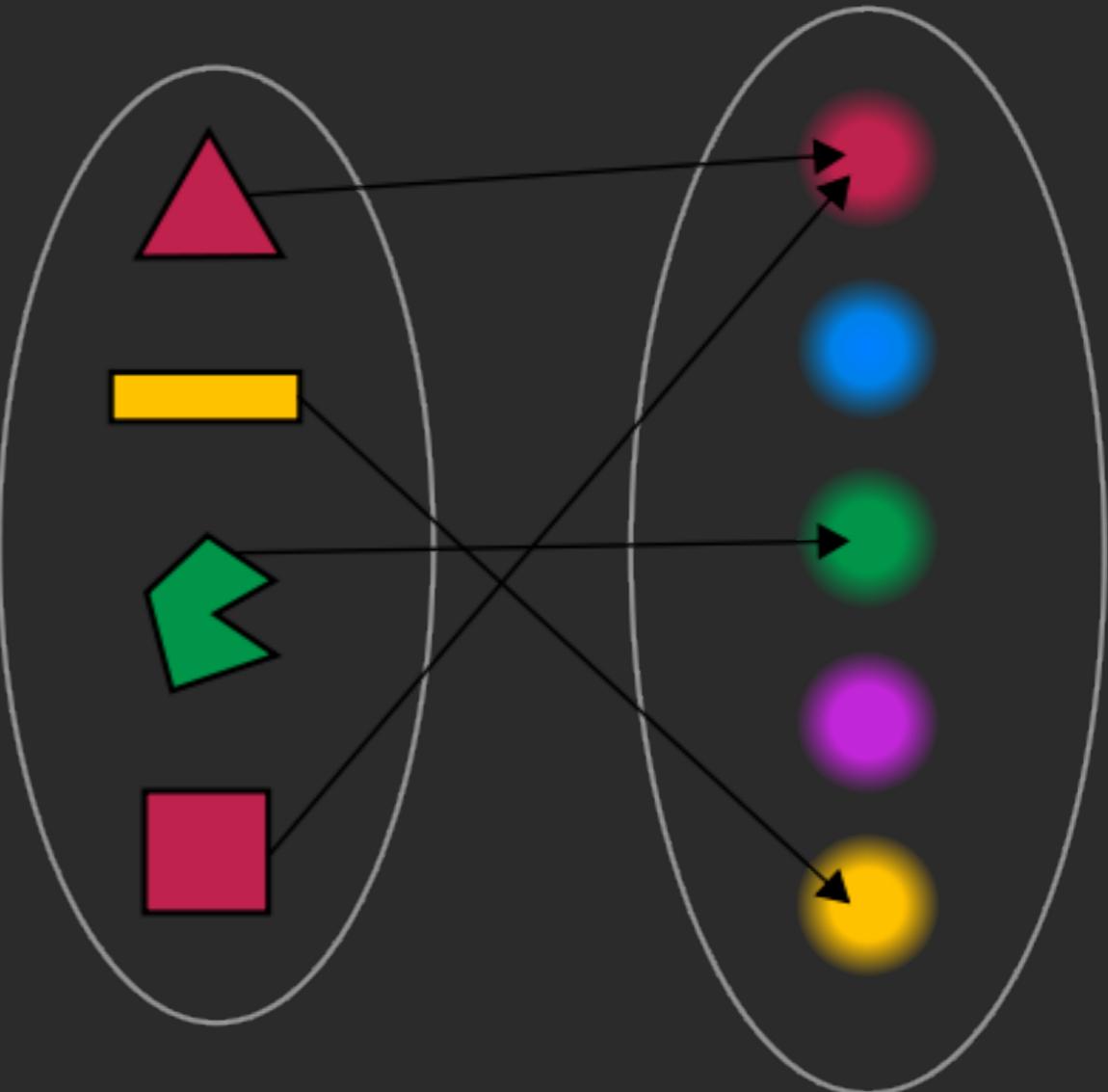
```
type Name = String
type Price = Int
type Importance = Int
type Activity[T] = (Name, Price, Importance, Future[T])

def doCheapFirst[T](aa: List[Activity[T]]) = ???
def doImportantFirst[T](aa: List[Activity[T]]) = ???
```

Lesson 2:

FP

is not
all or
nothing



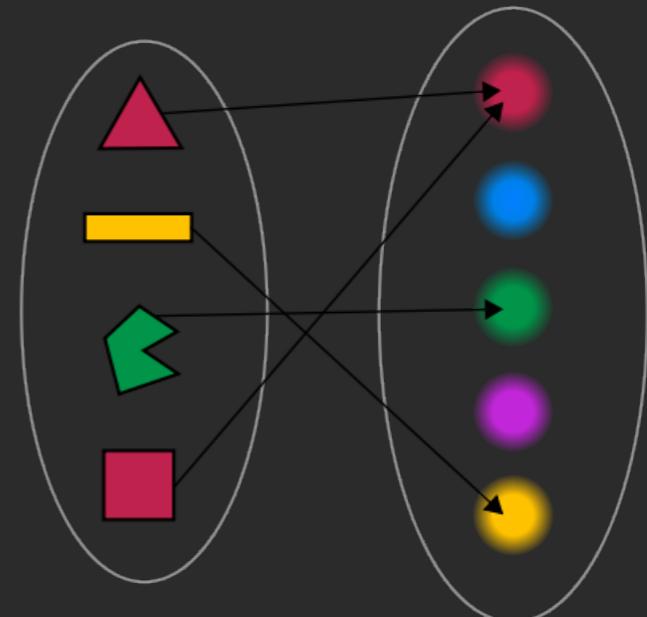
```
// Math function:  
// - relation between sets that;  
// - associates to every element of a first set;  
// - exactly one element of the second set.
```

```
sealed trait Shape
case object Circle extends Shape
case object Polygon extends Shape
case object Rectangle extends Shape
case object Hexagon extends Shape
```

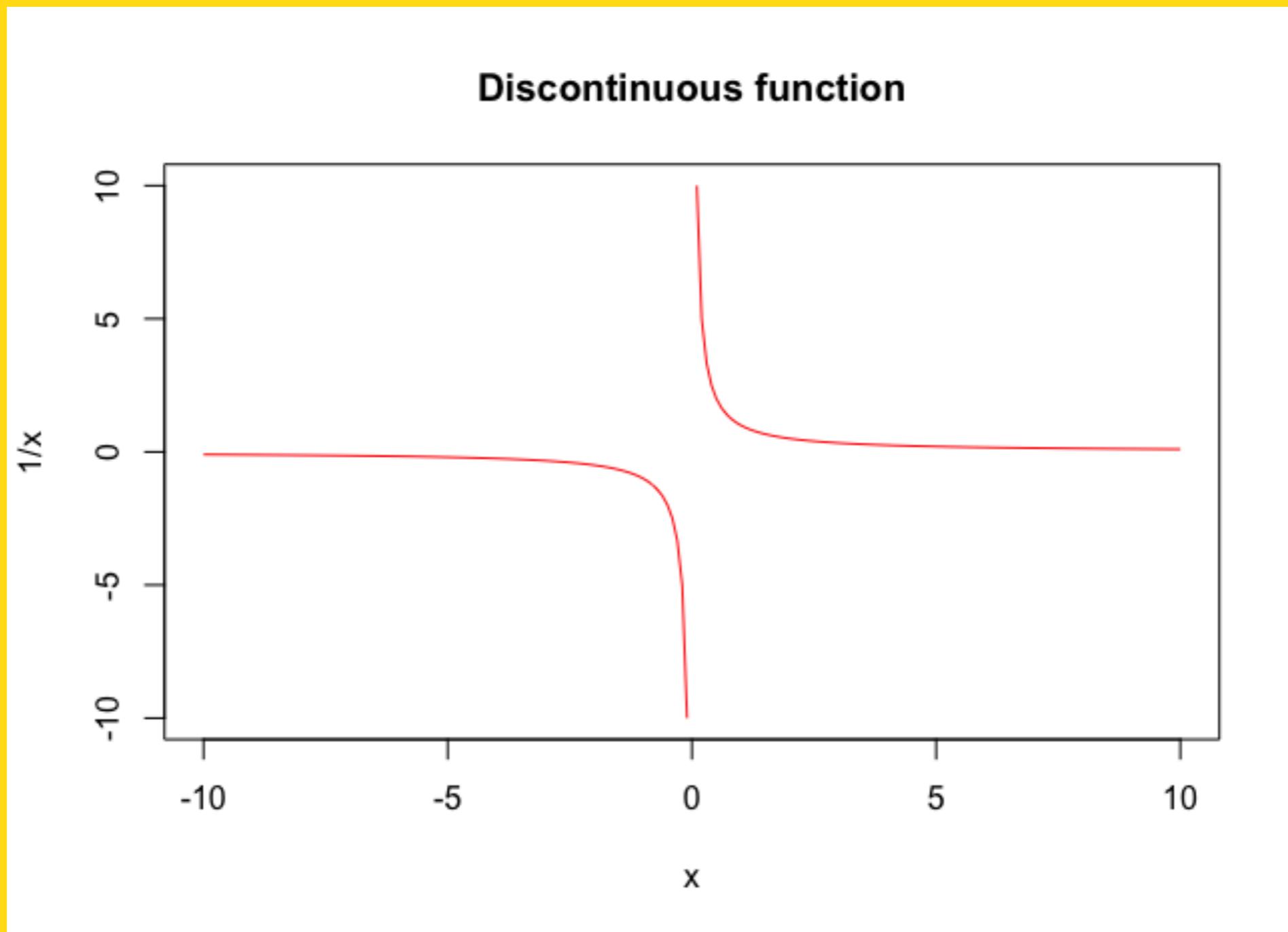
```
sealed trait Color
case object Red extends Color
case object Yellow extends Color
case object Green extends Color
case object Purple extends Color
case object Blue extends Color
```

```
def shapeColour(shape: Shape): Color = ???
```

```
// CS pure function:
// - return value is the same for the same args;
// - its evaluation has no side effects.
```



$$f(x) = 1/x$$



$$f(x) = 1/x$$

$$f: X \rightarrow Y$$

Set X - **domain**; Set Y - **co-domain**;

$$X = \mathbb{R} / \{0\}$$

* or “plug the gap” to
keep it \mathbb{R}

$$Y \supseteq \mathbb{R} / \{0\}$$

** trait Function1[-T1, +R]

```
// Not pure!
def getUserName(uid: String): String
```

-

```
// User may not exist
def getUserName(uid: String): Option[String]
```

- ||
nothing

```
// + there may be an error
def getUserName(uid: String):
    Try[Option[String]]
```

- ||
{<: Throwable}

```
// There may be an error
// + it may not exist
// + we don't know when it's done
def getUserName(uid: String):
    Future[Option[String]]
```

- ||
{<: Throwable} *

* asynchrony



Functional Effect. A functional effect is an **immutable data type** that **describes** (or models) the computation of one or more values, where the computation may require an **additional feature** like optionality, logging, access to context (like configuration), errors, state, or input/output. Using effect-specific operations, functional effects can be transformed

– John A De Goes, [A Glossary of Functional Programming](#)

cats-effect

IO,
SyncIO

A data type for encoding side effects as pure values, capable of expressing both synchronous and asynchronous computations.

Resource

Effectfully allocates and releases a resource.

Fiber

It represents the (pure) result of an Async data type (e.g. IO) being started concurrently and that can be either joined or canceled.

Timer

It is a scheduler of tasks.

...



Lesson 3:

IO is good for evolutionary development



```
object Jackpot

def javaGamble: Jackpot.type = {
    Thread.sleep(1000)
    if (System.currentTimeMillis() % 2 == 0) {
        Jackpot
    } else {
        throw new RuntimeException(
            "Sorry, no jackpot this time :-( "
        )
    }
}
```

Case: a java library, which does what you need, but not how you want it to be done.

Unwrap at the
“end of the world”

```
object SimpleGambleApp extends IOApp {
```

```
Delay[+A](thunk: () => A) extends IO[A]
```

```
(=> T) => IO[T]
```

```
private def gamble: IO[Jackpot.type] = IO(javaGamble)
```

```
override def run(args: List[String]): IO[ExitCode] = {  
    for {  
        _ ← IO(println("Started"))  
        jackpot ← gamble  
        _ ← IO(println(jackpot))  
    } yield ExitCode.Success  
}
```

IO is a monad, hence
`flatMap` and `for`.

}

```
object GentleGambleApp extends IOApp {  
  
    private def gamble: IO[Jackpot.type] = IO(javaGamble)  
  
    def gambleGentle: IO[Option[Jackpot.type]] =  
        gamble.attempt.map {  
            case Right(x) => x.some  
            case Left(_)   => None  
        }  
  
    override def run(args: List[String]): IO[ExitCode] = {  
        for {  
            _ ← IO(println("Started"))  
            jackpot ← gambleGentle  
            _ ← IO(println(jackpot))  
        } yield ExitCode.Success  
    }  
}
```

Let combinators do
their thing 🎉

Always
get a result!

Defining new in terms of
old 📖

```
private def gambleUntilWin(attempts: Int = 100): IO[Jackpot.type] =  
  gambleGentle.flatMap {  
    case Some(x) =>  
      x.pure[IO]  
    case None if attempts > 0 =>  
      IO(println(s"Retrying. ${attempts - 1} attempts left")) *>  
        gambleUntilWin(attempts - 1)  
    case None => IO.raiseError(new RuntimeException("Out of attempts"))  
  }
```

Recursion
without stack

Or use `cats-retry` lib

<https://typelevel.org/cats-effect/datatypes/io.html>

```
private def parGamble: IO[scala.Option[Jackpot.type]] =  
List.fill(10)(gambleGentle).parSequence.flatMap {  
_.count(_.isDefined) match {  
  case 0 => None.pure[IO]  
  case _ => Jackpot.some.pure[IO]  
}  
}
```

List[IO[T]] ⇒ IO[List[T]]

Lesson 4:

Keep your configs pure



```
object MyConf {  
  
    object mymodule {  
        val awakeInterval: FiniteDuration = 30.seconds  
        val jdbcUrl = sys.env  
            .getOrElse("DB_JDBC_URL",  
                      "jdbc:postgresql://xxxxx")  
        val dbUser = sys.env.getOrElse("DB_USER", "user")  
        val dbPassword = sys.env.getOrElse("DB_PASS", "")  
    }  
}  
  
watcher {  
  
    awakeInterval = 30 seconds  
  
    db {  
        jdbcUrl = "jdbc:postgresql://localhost:5432/surfsara"  
        jdbcUrl = ${DB_JDBC_URL}  
  
        user = "surfsara"  
        user = ${DB_USER}  
  
        password = ""  
        password = ${DB_PASSWORD}  
    }  
}
```

```
case class WatcherConf(db: DbConf,  
                      awakeInterval: FiniteDuration)  
  
object WatcherConf {  
  
  case class DbConf(jdbcUrl: String,  
                    username: String,  
                    password: String)  
  
}
```

```
case class WatcherConf(db: DbConf,  
                      awakeInterval: FiniteDuration)  
  
object WatcherConf {  
  
  case class DbConf(jdbcUrl: String,  
                    username: String,  
                    password: String)  
  
  implicit def hint[T]: ProductHint[T] =  
    ProductHint[T](ConfigFieldMapping(CamelCase, CamelCase))  
  
  def loadF: IO[WatcherConf] =  
    ConfigSource.default.at("watcher").loadF[IO, WatcherConf]  
}
```

```
// build.sc
object watcher extends ScalaModule with ScalafmtModule {
    override def moduleDeps = Seq(researchdrive)
}
```

```
// Loading configs per module
conf ← WatcherConf.loadF
rdConf ← ResearchDriveConf.loadF
da ← ResearchDriveDA.create(rdConf)
_ ← scheduleWatcher(conf, da)
```

Lesson 5:

At least two thread pools



```
val f = Future {  
    Thread.sleep(1000)  
} (? ExecutionContext)
```

```
val io = IO {  
    Thread.sleep(1000)  
42  
}
```

```
import cats.implicits._  
List.fill(10)(io).parSequence(? ContextShift)
```

Two thread pools

1. Async code

- Usually fixed thread pool, e.g. `ExecutionContext.global`
- Default: amount of CPUs

2. Blocking tasks that need to hold the thread

`parGamble`
needs
this

- Cached unbounded thread pool.
- In cats-effect normally wrapped into
`Blocker(bc: ExecutionContext)`

... or what?

- Fixed for blocking tasks may cause deadlock;
- Cached for async code will cause performance issues by spawning too many threads too quickly;
- You may see the problem only in production.

More about thread pools:

- <https://typelevel.org/cats-effect/concurrency/basics.html>
- <http://degoes.net/articles/zio-threads>
- <https://docs.scala-lang.org/overviews/core/futures.html>

Lesson 6:

[Un]recoverable errors

```
private def verifyETag(path: WebdavPath, expectedETag: ETag):  
IO[Boolean] = ???  
  
for {  
    eTagValidOrError ← verifyETag(WebdavPath(codePath), eTag).attempt  
    doneMsg ← eTagValidOrError match {  
        case Right(true) ⇒ processMessage(msg)  
        case Right(false) ⇒ TaskProgress.rejectedEtag(taskId).pure[IO]  
        case Left(ex) ⇒ TaskProgress.rejected(taskId, ex).pure[IO]  
    }  
    _ ← logger.info(s"The state of Done message is ${doneMsg.state}")  
    _ ← publisher(doneMsg)  
} yield ()
```

Lesson 7: Data model FTW

```
object Artifact {
```

Path on the host of JVM process

```
case class Location(localHome: Path,  
                    containerHome: Path,  
                    userPath: String)
```

Path in the container

Relative path to the file

```
}
```



Lesson 8:

Make
illegal state
un-
representable



```
// Resource is a data structure that
// captures the effectful allocation (acquire),
// along with its finalizer (release).
```

```
abstract class Resource[F[_], A] {
  def use[B](f: A ⇒ F[B]): F[B]
}
```

```
val ec = Resource
  .make(IO(Executors.newCachedThreadPool()))(
    tp => IO(tp.shutdown())
  )
  .evalMap(executor =>
IO(ExecutionContext.fromExecutor(executor)))
```

// You can't use it nor here

```
ec.use { implicit ec =>
  // ... only here
  IO(Future(42))
}
```

// ... neither here

ContainerEnv

How do I know if all files are in place and safe to use?

```
case class ContainerEnv(algorithm: Algorithm,  
                        input: InputData,  
                        output: OutputData)
```



Model it as a Resource!

```
def containerEnv(  
    startContainerCmd: Messages.StartContainer  
): Resource[IO, ContainerEnv] =  
  tempDirResource.evalMap { tempHome =>  
    import cats.implicitly._  
  
    val algorithmLocation = ???  
    val inputLocation = ???  
    val outputLocation = ???  
    val downloads = ???  
  
    for {  
      _ ← Webdav.downloadToHost(downloads)  
      algorithm ← Artifact.algorithm(algorithmLocation)  
      input ← Artifact.data(inputLocation)  
      output ← Artifact.output(outputLocation)  
    } yield ContainerEnv(algorithm, input, output)  
  }
```

Applies an effectful transformation to the allocated resource.

$V1 \Rightarrow IO[V2]$

Lesson 9:

Module sandwiching

```
object Apps extends IOApp {  
  
    override def run(args: List[String]): IO[ExitCode] =  
        for {  
            f1 ← App1.run(args).start  
            f2 ← App2.run(args).start  
            e1 ← f1.join  
            e2 ← f2.join  
        } yield ExitCode(e1.code + e2.code)  
}
```

Start execution of the source suspended in the `IO` context

Uses implicit ContextShift



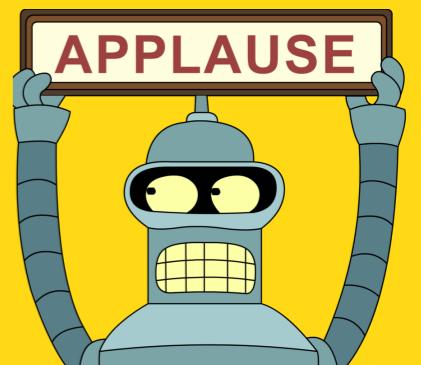
Lessons learned

- * Future[T] is a thing of the past
- * FP is not all or nothing
- * IO is good for evolutionary development
- * Keep your configs pure
- * At least two thread pools
- * [Un]-recoverable errors
- * Data model FTW
- * Make illegal state unrepresentable



Scala ecosystem is great for prototyping!

- Scala - easy data modelling;
- Cats Effect - IO, parSequence(), attempt(), Fibers;
- Resources - well modelled types that became first-class citizen and can be easily passed around;
- Streams - easy and natural integration with a message broker and memory efficiency for large chunks of data;



Thank you ! Questions?

aws-lambda-scala

 [mkotsur](#)

 [@s_pcropy](#)

 [medium.com/@mkotsur](#)

More about
prototyping

More lessons
learned

