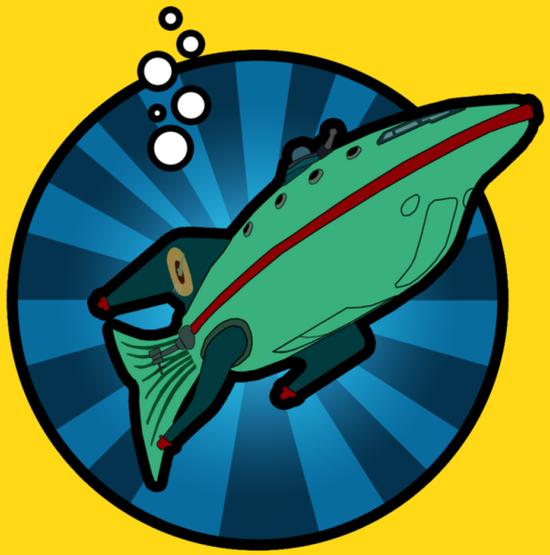
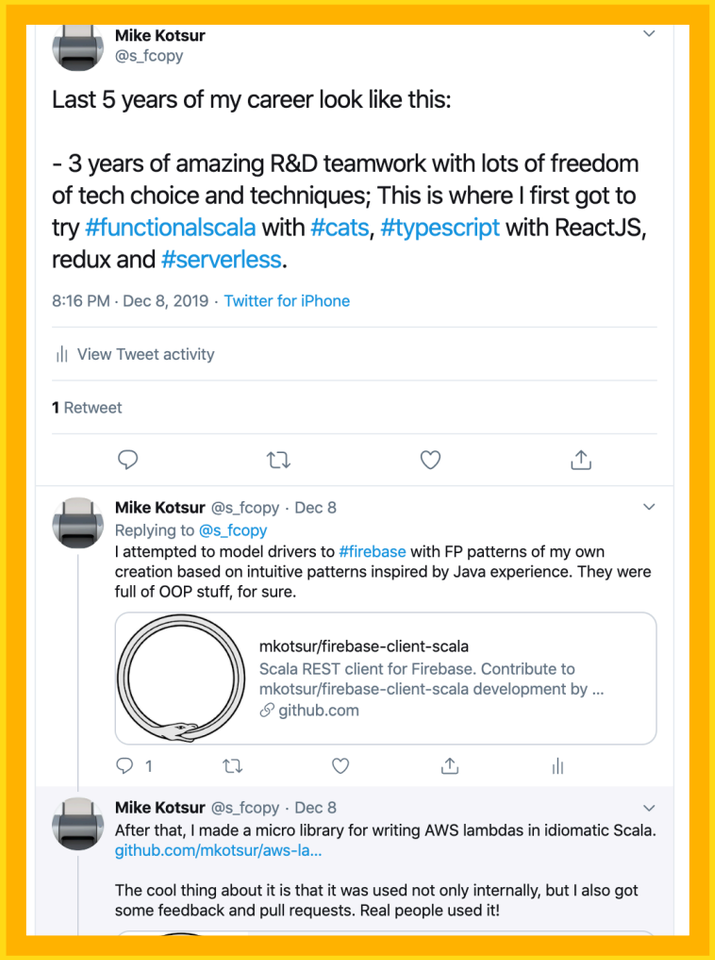
Prototyping the Future with Functional Scala



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

@s_pcopy

medium.com/@mkotsur

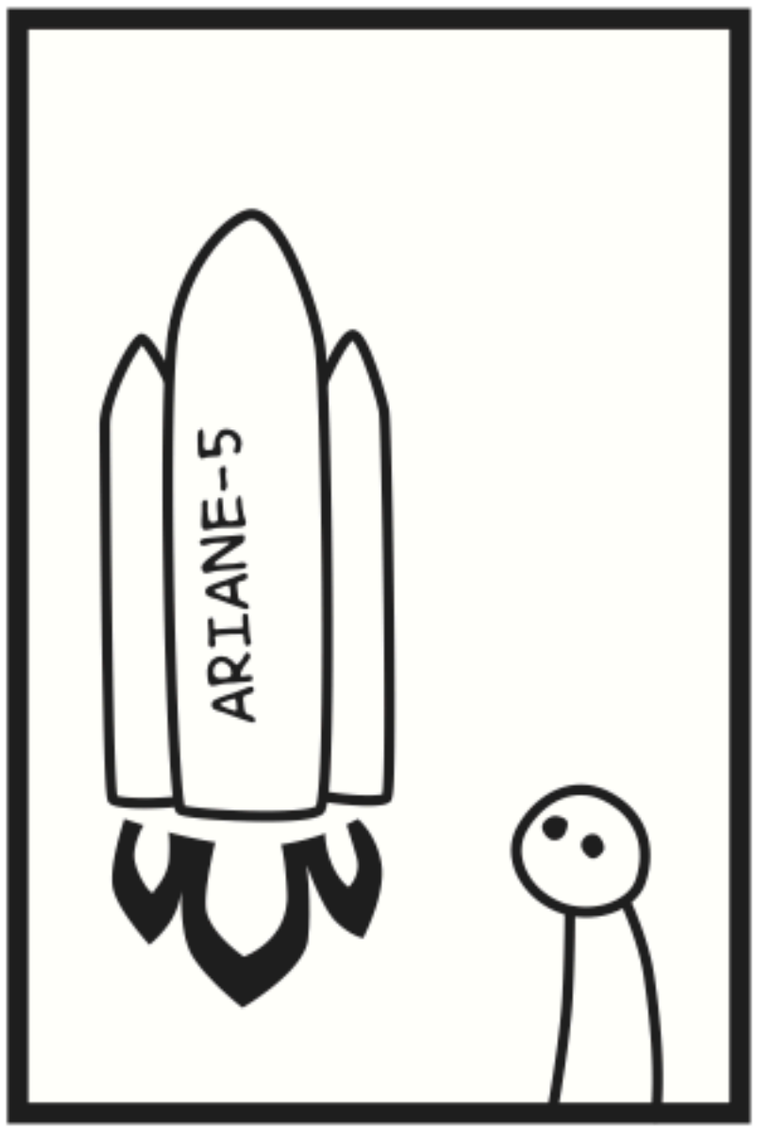
We’ll cover

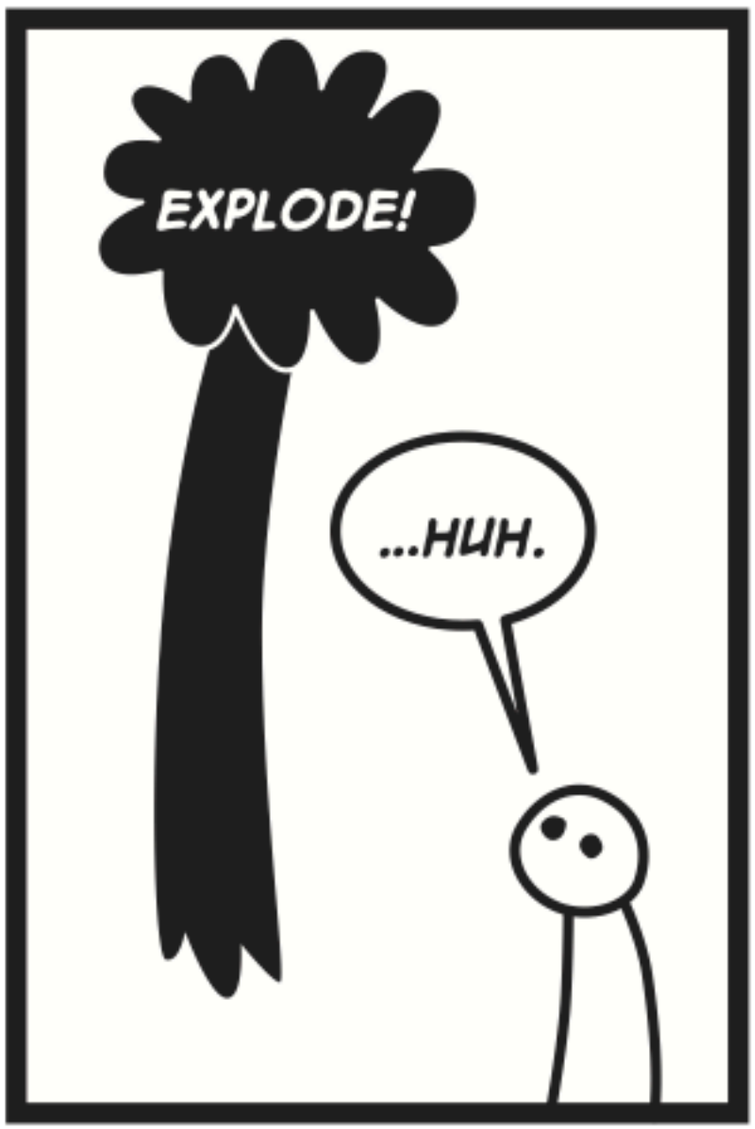
- 🚲 What prototyping is all about;
- ⓪① Data Exchange project challeng
- 🎉☠️ X Lessons I learned;
- 💪 How modern FP Scala libs help.

CGO

Fix lessons total


OH, WELL! I BET THAT CODE NEVER GETS USED, ANYWAY.
>ERROR!<


ARIANE-5

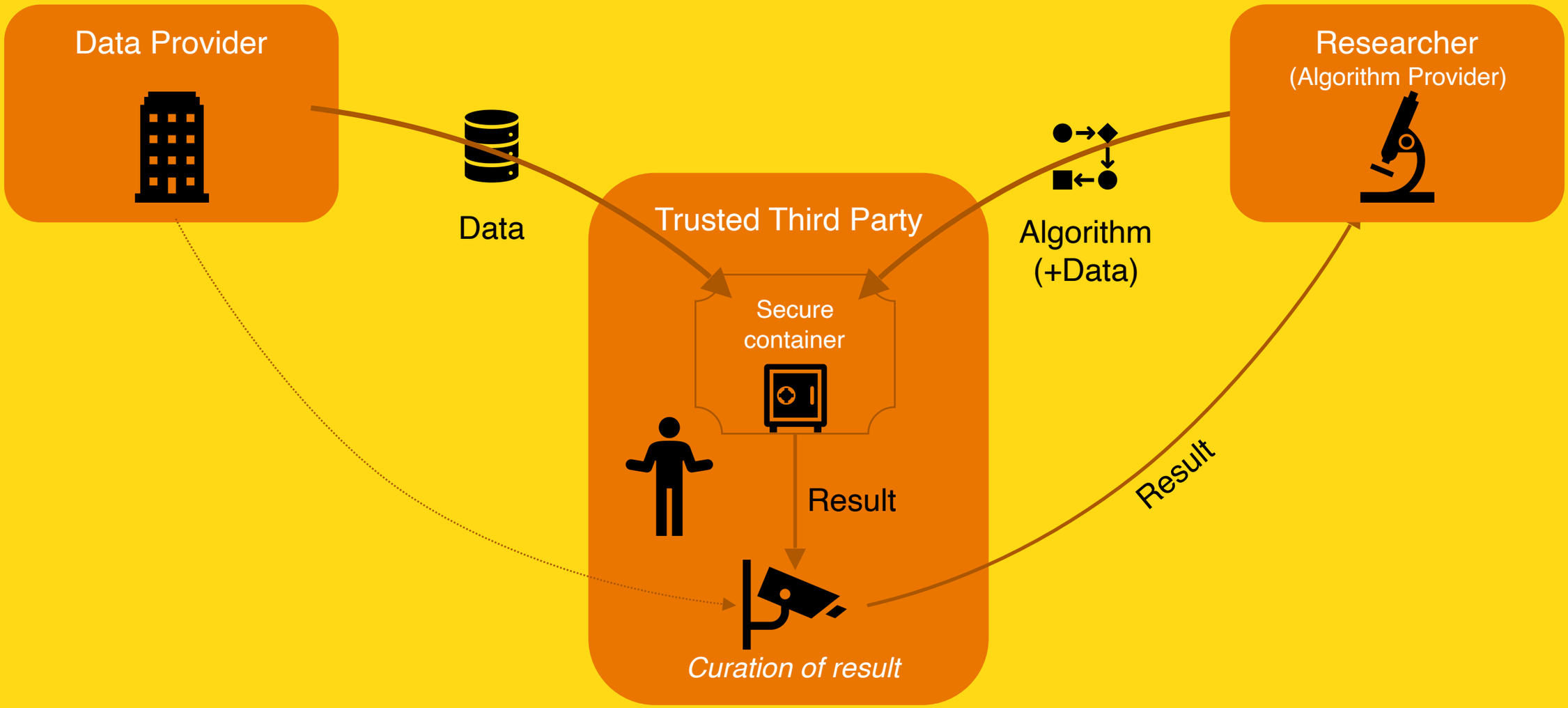

EXPLODE!
...HUH.

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



Lesson 0:

Python is great... in scripts and noteboks

```
>>> 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!
```

Lesson 0:

Python is great... in scripts and noteboks

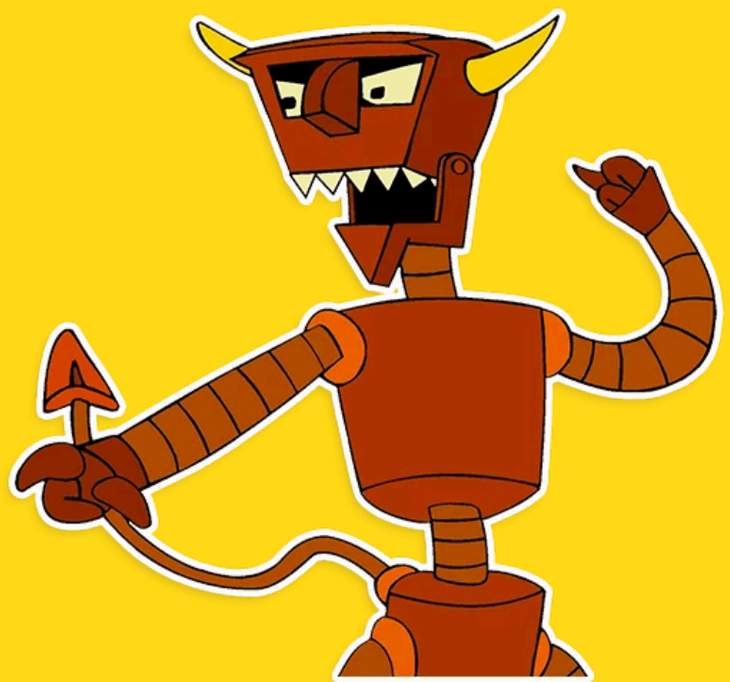
```
>>> a = "some_string"
>>> id(a)
4518396400
>>> id("some" + "_" + "string") # Ids are same 🤪
4518396400


>>> a = "wtf"
>>> b = "wtf"
>>> a is b
True
>>> a = "wtf!"
>>> b = "wtf!"
>>> a is b # WTF indeed 😯
False



some_dict = {}
some_dict[5.5] = "JavaScript"
some_dict[5.0] = "Ruby"

>>> some_dict[5.5]
"JavaScript"
>>> some_dict[5.0] # Shouldn’t it be Python? 🤔
“Python"



 //More: https://github.com/satwikkansal/wtfpython
```

Python/Django issues

- Not thread safe (see GIL[1]);
    - Scaling through processes;

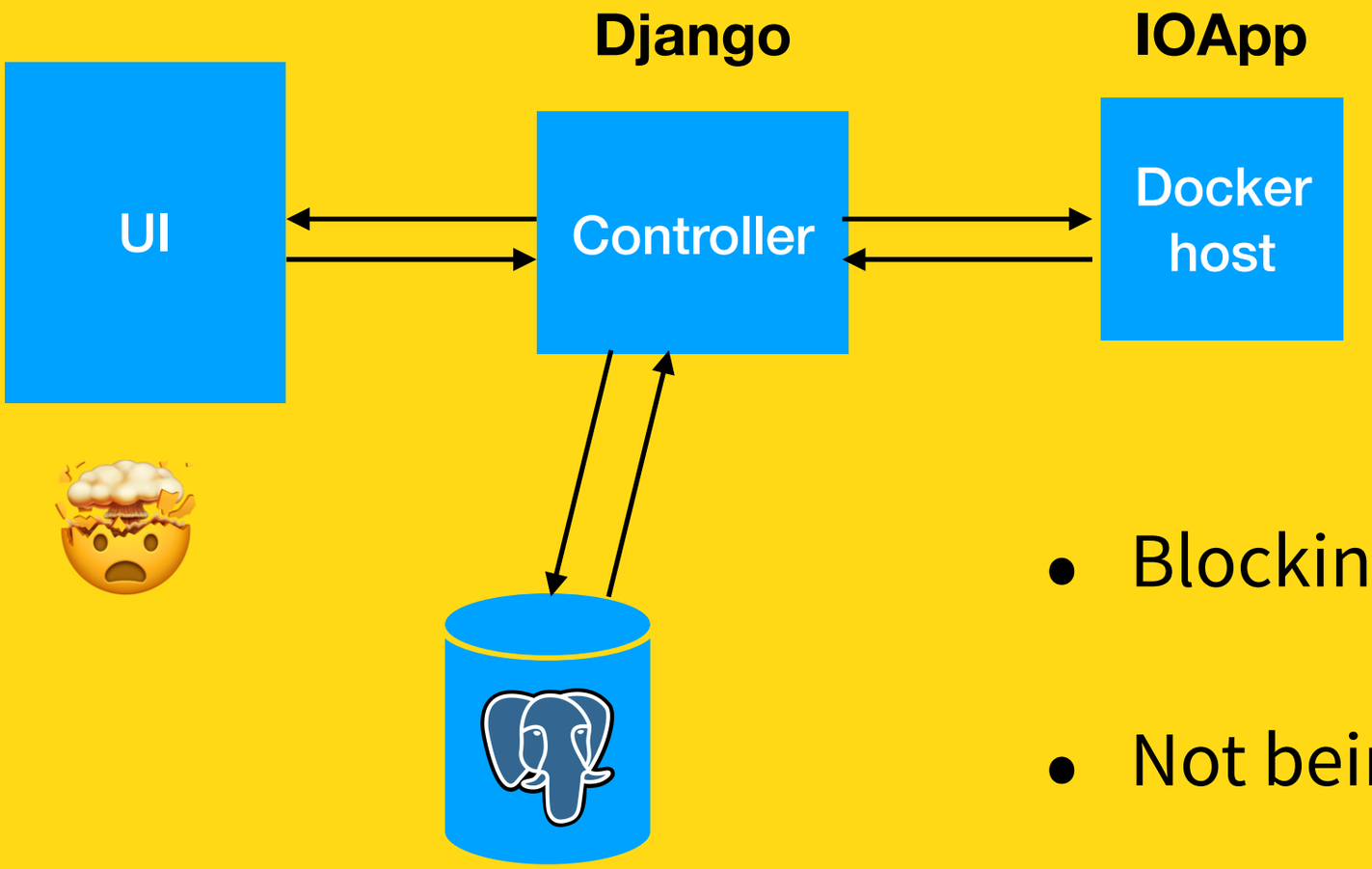
The Twelve-Factor App
#6, #8

- Not type safe;
    - Suboptimal development experience;
    - Difficult domain modelling.



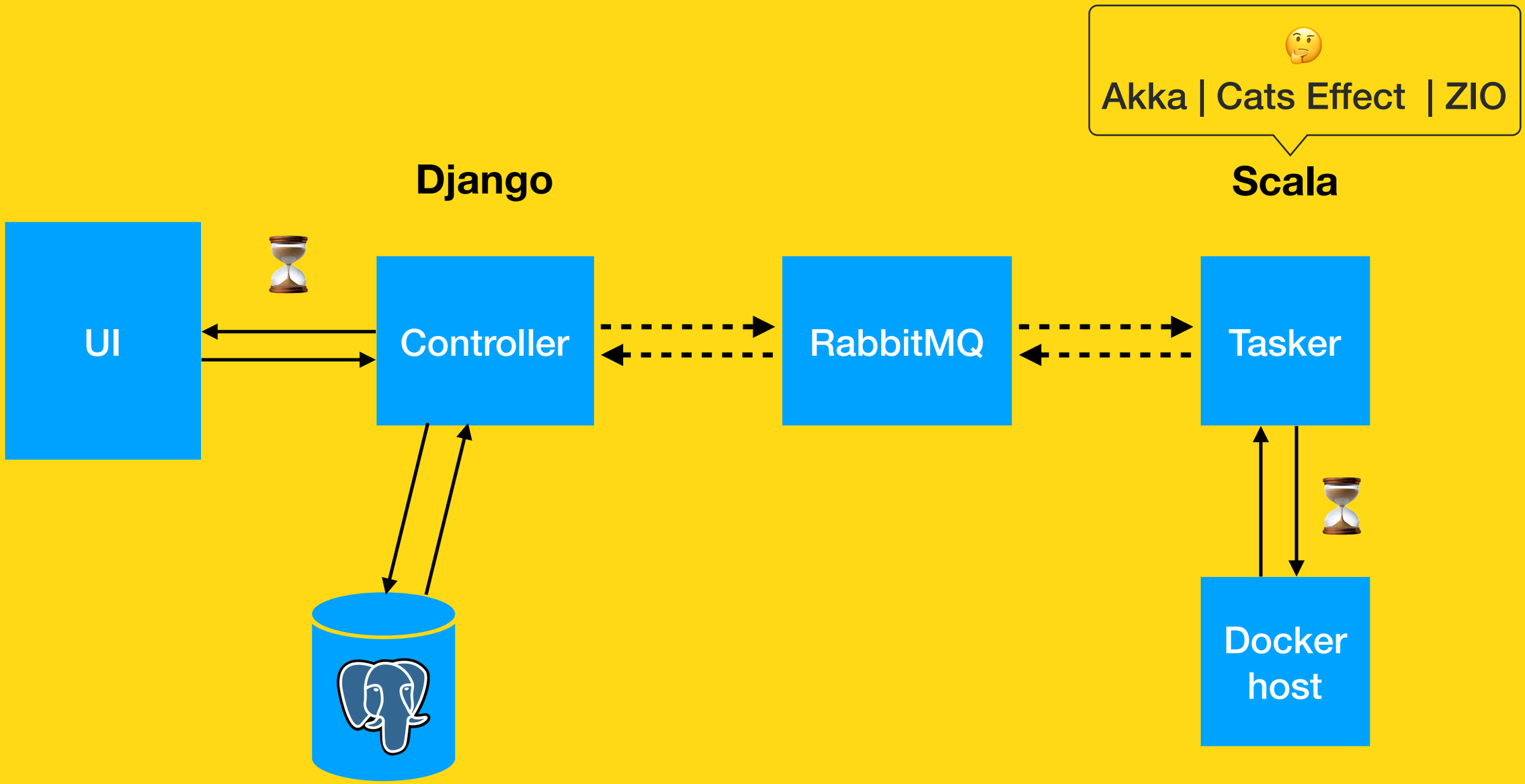
1. Real Python - Global Interpreter Lock

Architecture v0.1



- Blocking requests for 2+ minutes
- Not being able to refresh page
- Results are at the mercy of app server

Architecture v0.2



Lesson 1:

Future[T] is a thing of the past

```scala
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:

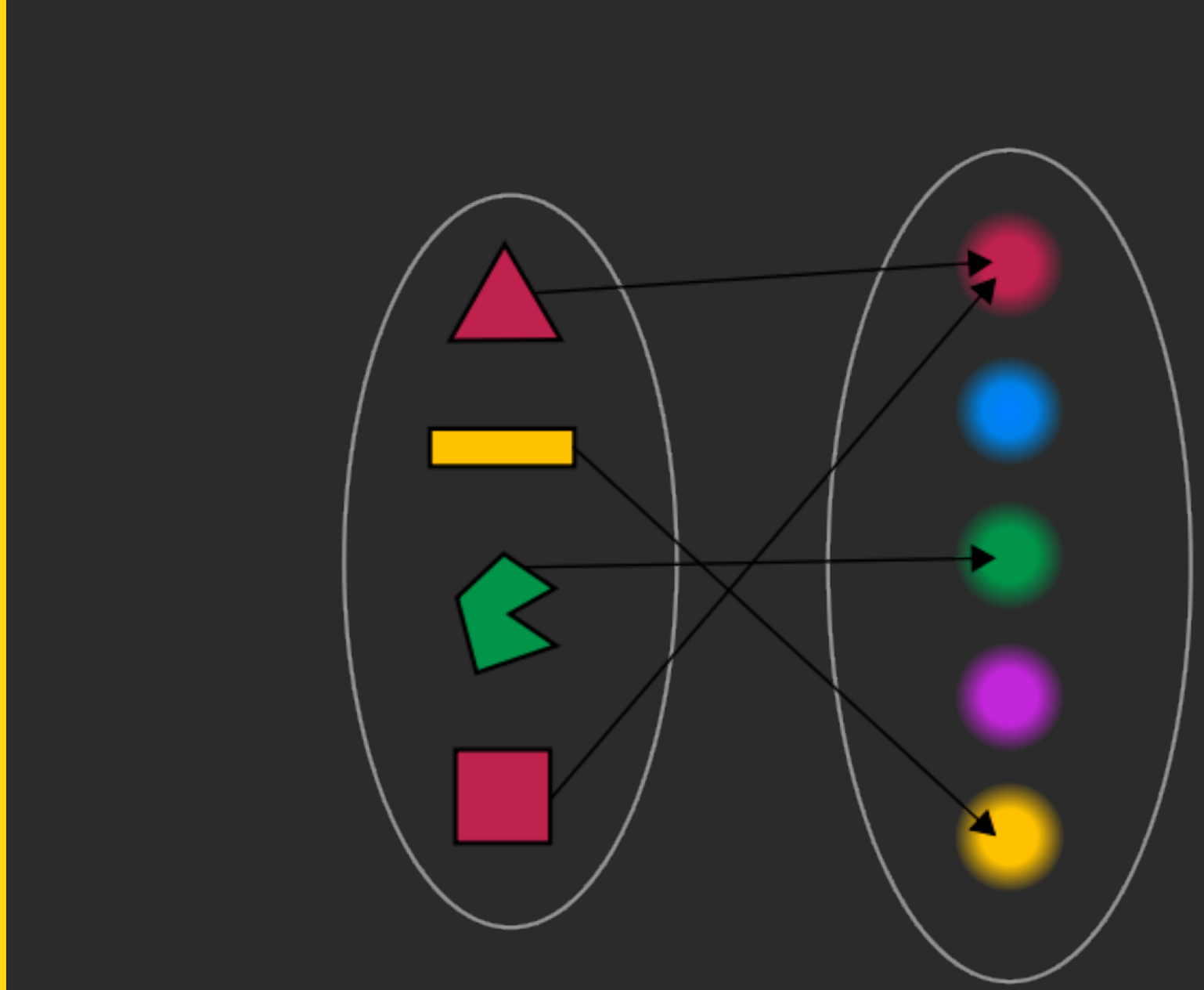
Future[T] is a thing of the past

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:

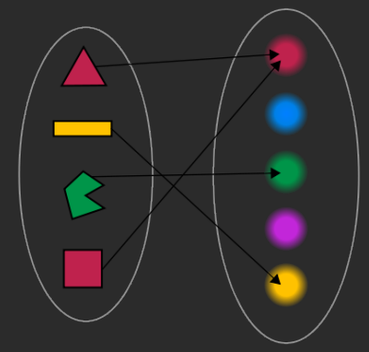
FP in Scala is not **all** or **nothing**



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

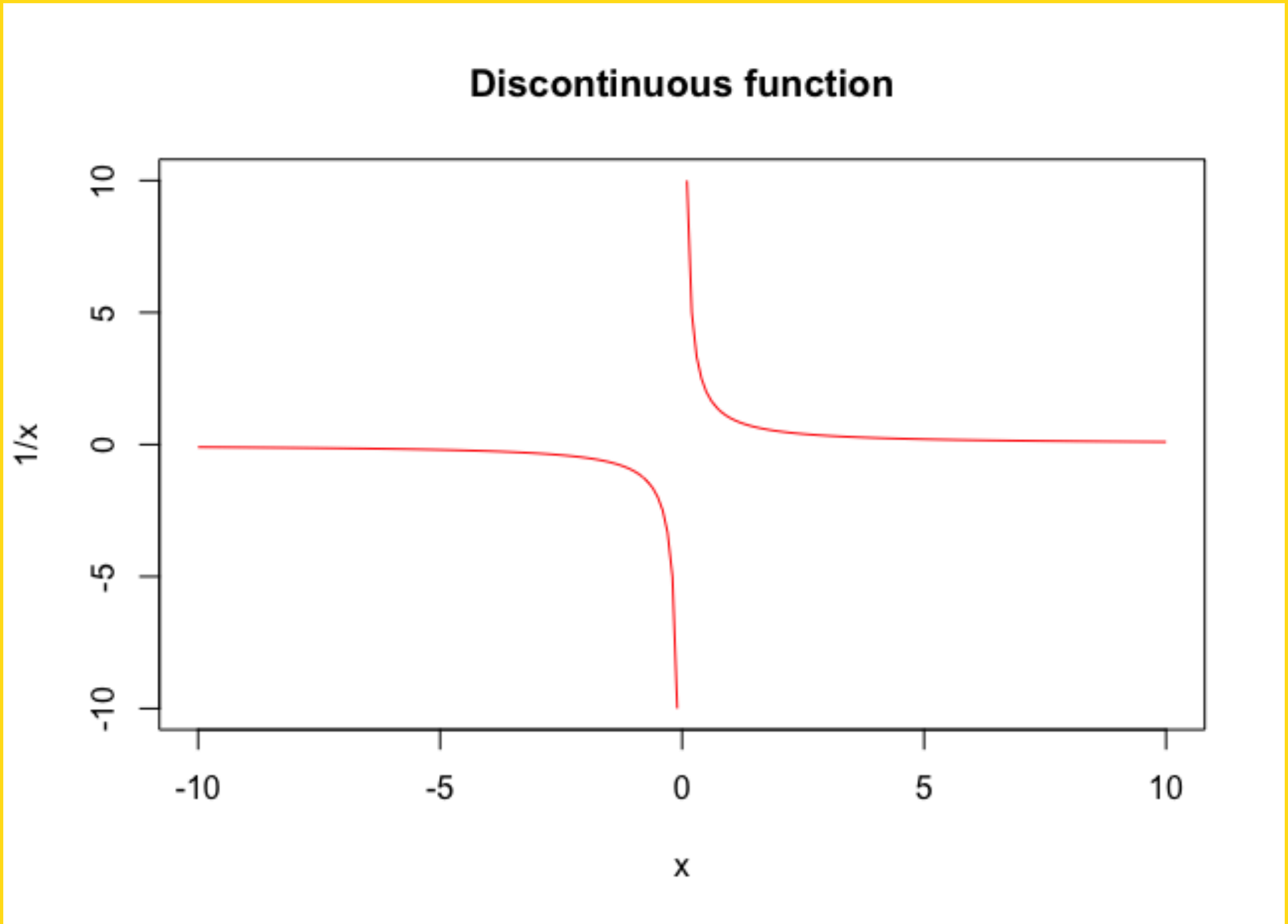
Lesson 2:

FP in Scala is not **all** or **nothing**



```scala
sealed trait Shape
case object Circle extends Shape
case object Square 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.
```

$f(x) = 1/x$



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

$$f: X \to Y$$

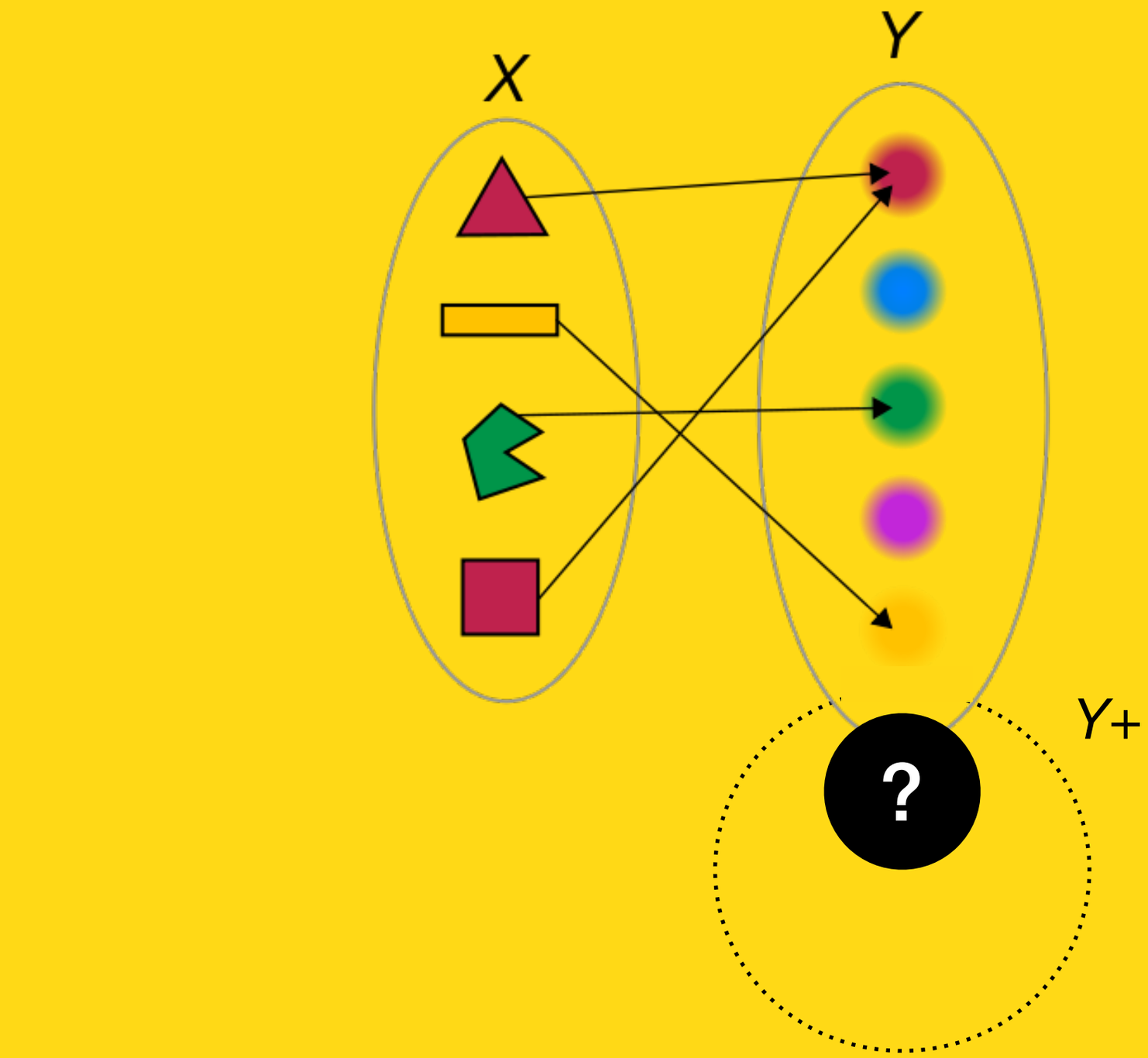
Set X - **domain**; Set Y - **co-domain**;

Fix the operator or explain it

```
X = R / {0}
* or “plug the gap” to
keep it R
```

```
Y ⊇ B
** trait Function1[-T1, +R]
```

Function1[-T1, +R]



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

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

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

Y ||
empty set

Y ||
{<: Throwable}

Y* ||
{<: Throwable}*

* asynchrony

Y and Y*
are disjoint:

```scala
/**
  * NF = Next Future
  */
object NF {}

trait NF[T] {}
```

```scala
package conf2020scalaua.nf

import scala.concurrent.Future

/**
  * NF = Next Future
  */
object NF {
  def apply[T](f: ⇒ Future[T]): NF[T] = ???
}

trait NF[T] {
  private[nf] def asFuture: Future[T]
}
```

```scala
object NFApp {
  type ReturnCode = Int
}

trait NFApp {
  import NFApp._

  // Please override this method!
  def run(args: List[String]): NF[ReturnCode]

  final def main(args: Array[String]): Unit = {
    val resultFuture = this.run(args.toList).asFuture
    val code = Await.result(resultFuture, Duration.Inf)
    System.exit(code)
  }
}
```

```
object NFDemo extends NFApp {
  override def run(args: List[String]): NF[ReturnCode] = {
    NF(Future.successful(0))
  }
}
```

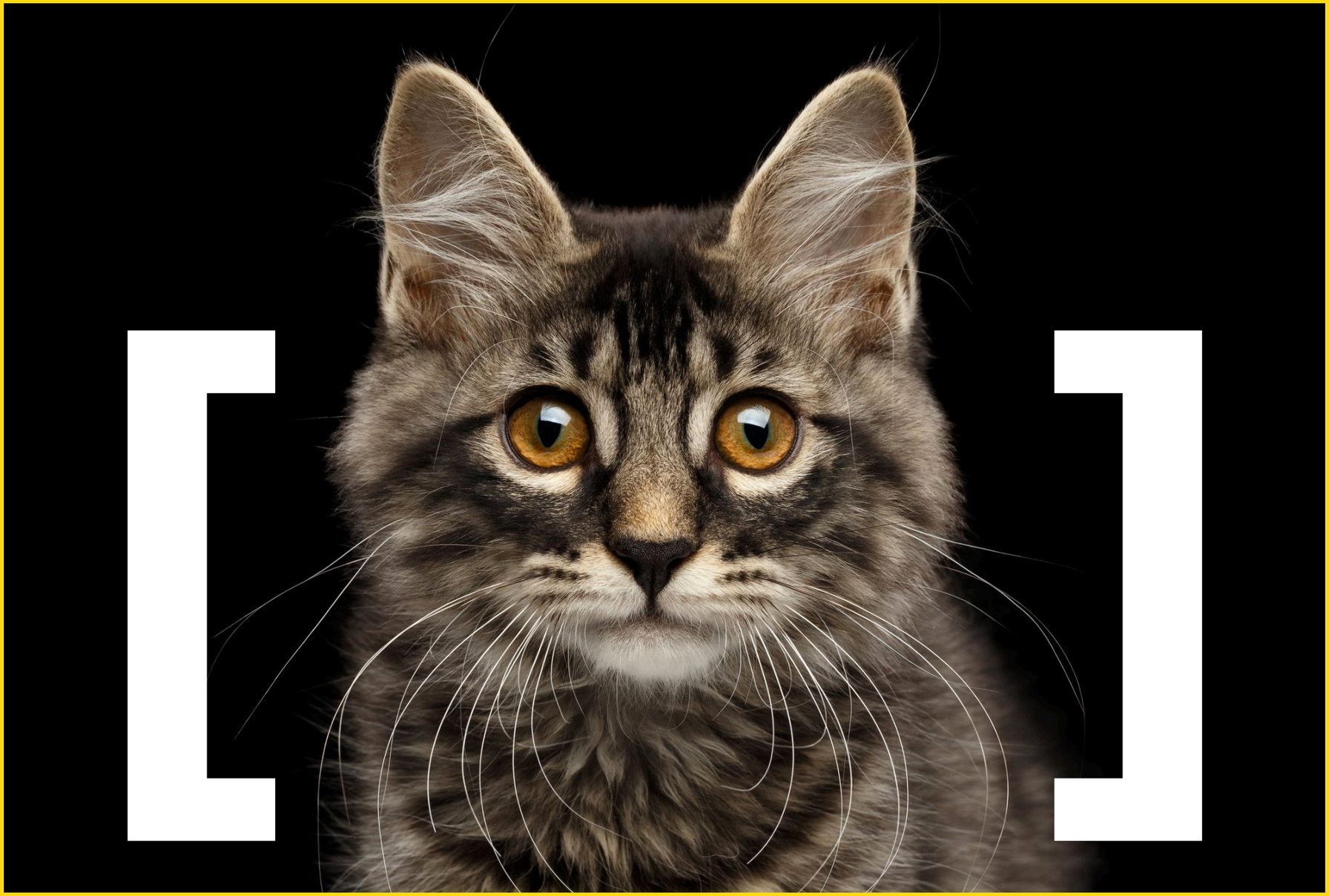
```scala
object NF {
  def apply[T](f: ⇒ Future[T]): NF[T] = new NF(f)
  def sync[T](f: ⇒ T): NF[T] = NF(Future.successful(f))
}
```

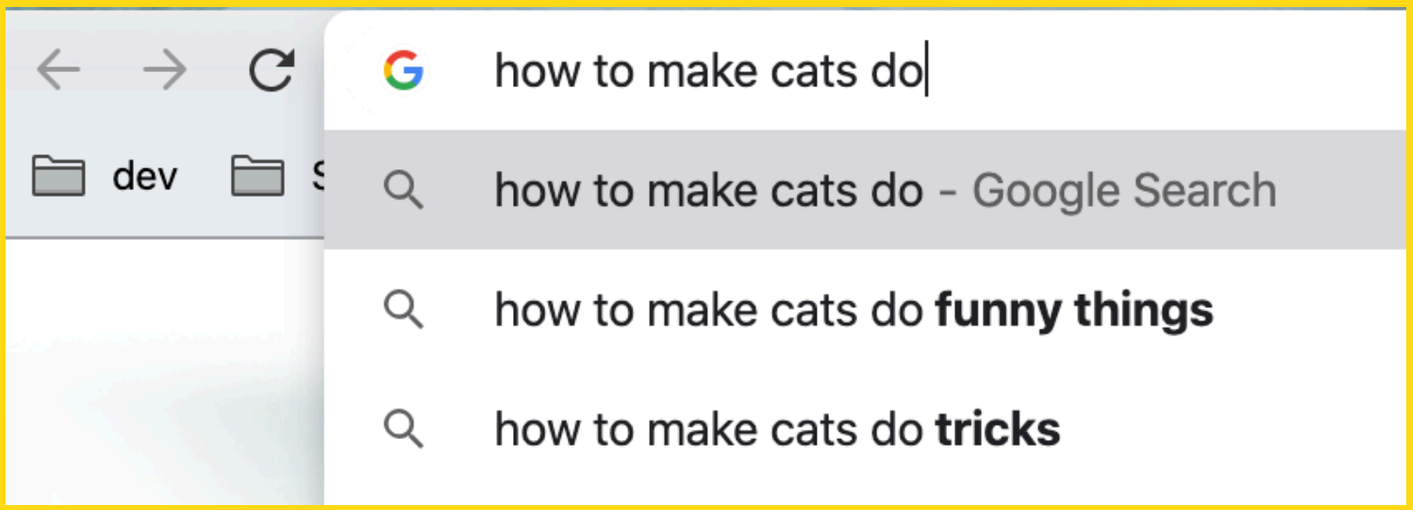
```scala
final class NF[T] private (thunk: ⇒ Future[T]) {
  private[nf] def asFuture: Future[T] = thunk
}
```

```
object NFDemo extends NFApp {
  override def run(args: List[String]): NF[ReturnCode] = {
    NF(Future.successful(0))
  }
}
```

```scala
final class NF[T] private (thunk: => Future[T]) {

  private[nf] def asFuture: Future[T] = thunk

  def flatMap[N](mapper: T => NF[N])(implicit ec:
ExecutionContext): NF[N] =
    NF[N](thunk.flatMap(s => mapper(s).asFuture))

  def map[N](mapper: T => N)(implicit ec:
ExecutionContext): NF[N] =
    NF[N](thunk.map(mapper))

}
```

```scala
object NFDemo extends NFApp {
  override def run(args: List[String]): NF[ReturnCode] = {
    import ExecutionContext.Implicits.global

    val nameUF = NF.sync(StdIn.readLine())

    for {
      _ ← NF.sync(println(s"What is your name?"))
      name ← nameUF
      _ ← NF.sync(println(s"Hello, $name"))
    } yield 0
  }
}
```




how to make cats do
dev
how to make cats do - Google Search
how to make cats do **funny things**
how to make cats do **tricks**

“The Ladder of Functional Programming”
— LambdaConf

Novice	Advanced Beginner	Competent	Proficient	Concepts
Immutable data	ADTs	Generalized ADTs	Codata	High Performance
Second order functions	Pattern matching	Higher-Kinded Types	(Co) Recursion schemes	Kind Polimorphism
Constructing & Destructing	Parametric Polymorphism	Rank-N Types	Advanced optics	Generic Programming
Functional composition	General Recursion	Folds and Unfolds	Dual Abstractions (Comonad)	Type-Level Programming
First-Class functions & Lambdas	Type Classes, instances and Laws	Higher-Order Abstractions	Monad Transformers	Dependent-Types, Singleton Types
	Lower-Order abstractions (Equal,	Basic Optics	Free monads & Extensible Effects	Category Theory
	Referential Transparency, Totality	Efficient Persistent Data Structures	Functional Architecture	Graph Reduction
	Higher-Order Functions	Existential Types	Advanced Functors	Higher-Order Abstract Syntax
	Partial-Application, Currying, Point-Free	Embedded DSLs using combinators	Embedded DSLs using GADTs	Compiler Design for functional languages
			Advanced Monads (Continuation, Logic)	Profound Optics
			Type Families, Functional Dependencies	

https://twitter.com/lambda_conf/status/803695008100466688/photo/1

“The Ladder of Functional Programming”
— LambdaConf

Novice	Advanced Beginner	Competent	Proficient	Concepts
Immutable data	ADTs	Generalized ADTs	Codata	High Performance
Second order functions	Pattern matching	Higher-Kinded Types	(Co) Recursion schemes	Kind Polimorphism
Constructing & Destructing	Parametric Polymorphism	Rank-N Types	Advanced optics	Generic Programming
Functional composition	General Recursion	Folds and Unfolds	Dual Abstractions (Comonad)	Type-Level Programming
First-Class functions & Lambdas	Type Classes, instances and Laws	Higher-Order Abstractions	Monad Transformers	Dependent-Types, Singleton Types
	Lower-Order abstractions (Equal,	Basic Optics	Free monads & Extensible Effects	Category Theory
	Referential Transparency, Totality	Efficient Persistent Data Structures	Functional Architecture	Graph Reduction
	Higher-Order Functions	Existential Types	Advanced Functors	Higher-Order Abstract Syntax
	Partial-Application, Currying, Point-Free	Embedded DSLs using combinators	Embedded DSLs using GADTs	Compiler Design for functional languages
			Advanced Monads (Continuation, Logic)	Profound Optics
			Type Families, Functional Dependencies	

https://twitter.com/lambda_conf/status/803695008100466688/photo/1

“The Ladder of Functional Programming”
— LambdaConf

Novice	Advanced Beginner	Competent	Proficient	Concepts
Immutable data	ADTs	Generalized ADTs	Codata	High Performance
Second order functions	Pattern matching	Higher-Kinded Types	(Co) Recursion schemes	Kind Polimorphism
Constructing & Destructing	Parametric Polymorphism	Rank-N Types	Advanced optics	Generic Programming
Functional composition	General Recursion	Folds and Unfolds	Dual Abstractions (Comonad)	Type-Level Programming
First-Class functions & Lambdas	Type Classes, instances and Laws	Higher-Order Abstractions	Monad Transformers	Dependent-Types, Singleton Types
	Lower-Order abstractions (Equal,	Basic Optics	Free monads & Extensible Effects	Category Theory
	Referential Transparency, Totality	Efficient Persistent Data Structures	Functional Architecture	Graph Reduction
	Higher-Order Functions	Existential Types	Advanced Functors	Higher-Order Abstract Syntax
	Partial-Application, Currying, Point-Free	Embedded DSLs using combinators	Embedded DSLs using GADTs	Compiler Design for functional languages
			Advanced Monads (Continuation, Logic)	Profound Optics
			Type Families, Functional Dependencies	

https://twitter.com/lambda_conf/status/803695008100466688/photo/1

Lesson 3:

IO is good for evolutionary development

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

```
object Jackpot

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

```scala
object SimpleGambleApp extends IOApp {

  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
  }

}
```

```scala
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

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

```scala
private def parGamble: IO[scala.Option[Jackpot.type]] =
  Range(0, 10).map(_ ⇒ gambleGentle).toList.parSequence.flatMap {
    _.count(_.isDefined) match {
      case 0 ⇒ None.pure[IO]
      case _ ⇒ Jackpot.some.pure[IO]
    }
  }
```

List[IO[T]] => IO[List[T]]

Mind the ContextShift ⚠️

https://morbotron.com/

Lesson 5:

Parse, don’t validate

```scala
object MyConf {

  object mymodule {
    val awakeInterval: FiniteDuration = 30.seconds
    val jdbcUrl = sys.env
      .getOrElse("DB_JDBC_URL", "jdbc:postgresql://localhost:5432/mydb")
    val dbUser = sys.env.getOrElse("DB_USER", "user")
    val dbPassword = sys.env.getOrElse("DB_PASSWORD", "")
  }
}
```

Lesson 5:

Parse, don't validate

```scala
case class WatcherConf(db: DbConf,
                       awakeInterval: FiniteDuration)


object WatcherConf {

  implicit def hint[T]: ProductHint[T] =
    ProductHint[T](ConfigFieldMapping(CamelCase, CamelCase))

  def loadF: IO[WatcherConf] =
    ConfigSource.default.at("watcher").loadF[IO,
WatcherConf]

  case class DbConf(jdbcUrl: String, username: String,
password: String)

}
```

Lesson 5:

Parse, don't validate

```
// 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:

Parse, don't validate

```
// 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 x: onfig

```
object queues {
  object todo {
    private val name = "tasker_todo"
    val config =
DeclarationQueueConfig.default(QueueName(nam
e))
    val exchangeConfig =
      DeclarationExchangeConfig.default(
        ExchangeName(name),
        ExchangeType.Direct
      )
    val routingKey = RoutingKey(name)
  }
  object done {
    private val name = "tasker_done"
    val config =
DeclarationQueueConfig.default(QueueName(nam
e))
    val exchangeConfig =
      DeclarationExchangeConfig.default(
        ExchangeName(name),
        ExchangeType.Direct
      )
    val routingKey = RoutingKey(name)
  }
}
```

```
queues {
  todo {
    name = tasker_todo
    exchangeName = taker_todo
    routingKey = taker_todo
  }

  done {
    name = tasker_done
    exchangeName = taker_done
    routingKey = taker_done
  }
}
```

Move all implicit configuration conventions into the config files.

Lesson 4:

At least two thread pools

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

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

Lesson 2:
At least 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

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

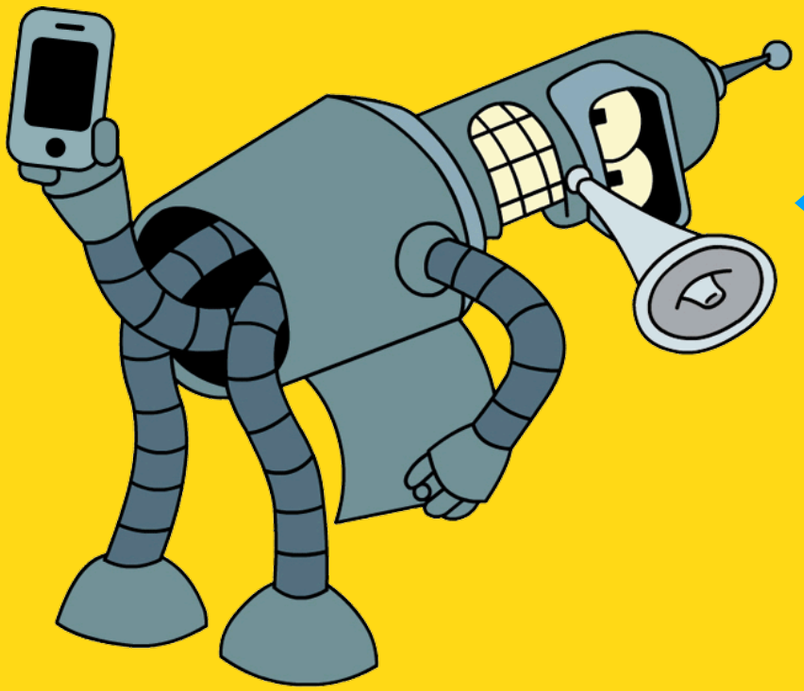
`gambleBig` needs this

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 3:
[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 ()
```



Not much more can I do if there is no docker image on the host 🤷‍♂️

```
val imageWithDeps: IO[ImageId] = ???
```

Lesson 4: Algebraic Data Types

```
object Artifact {

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

}
```

Path on the host of JVM process

Path in the container

Relative path to the file

Lesson 4: Data model FTW

```
object Artifact {

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

}
```

Path on the host of JVM process

Path in the container

Relative path to the file

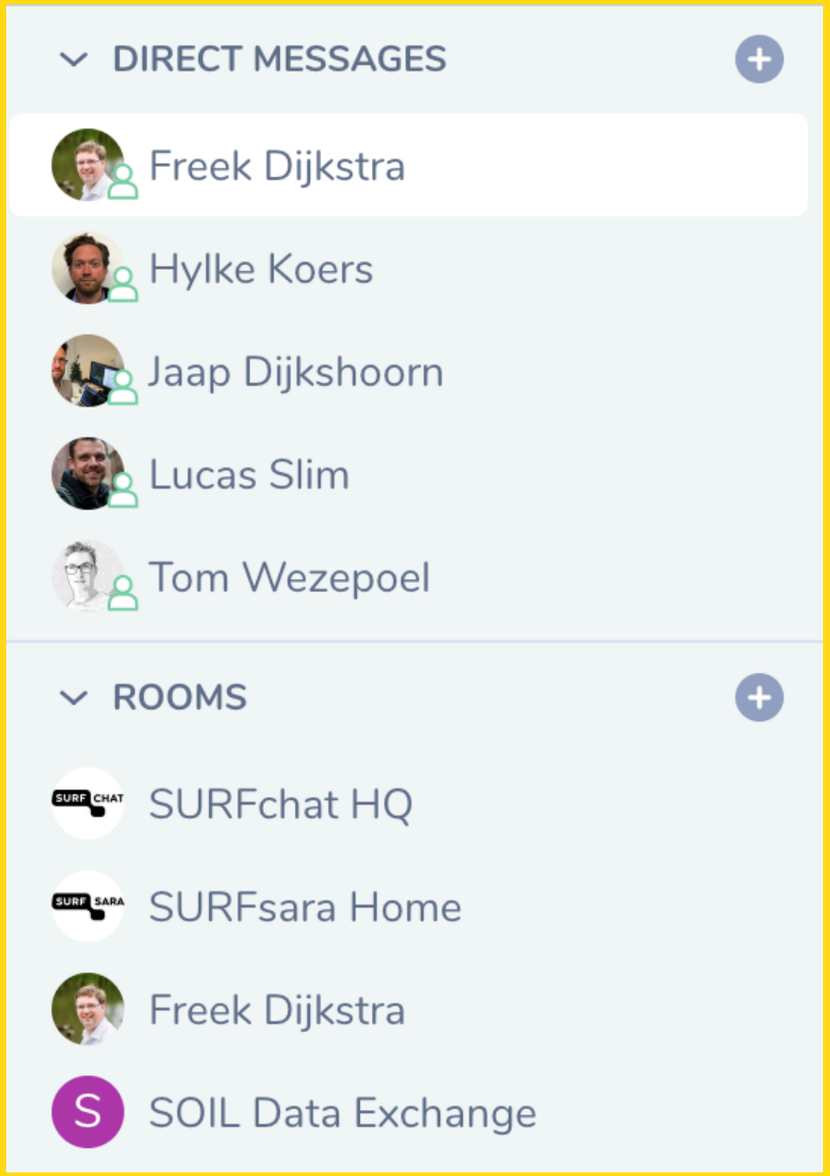
Lesson 5:

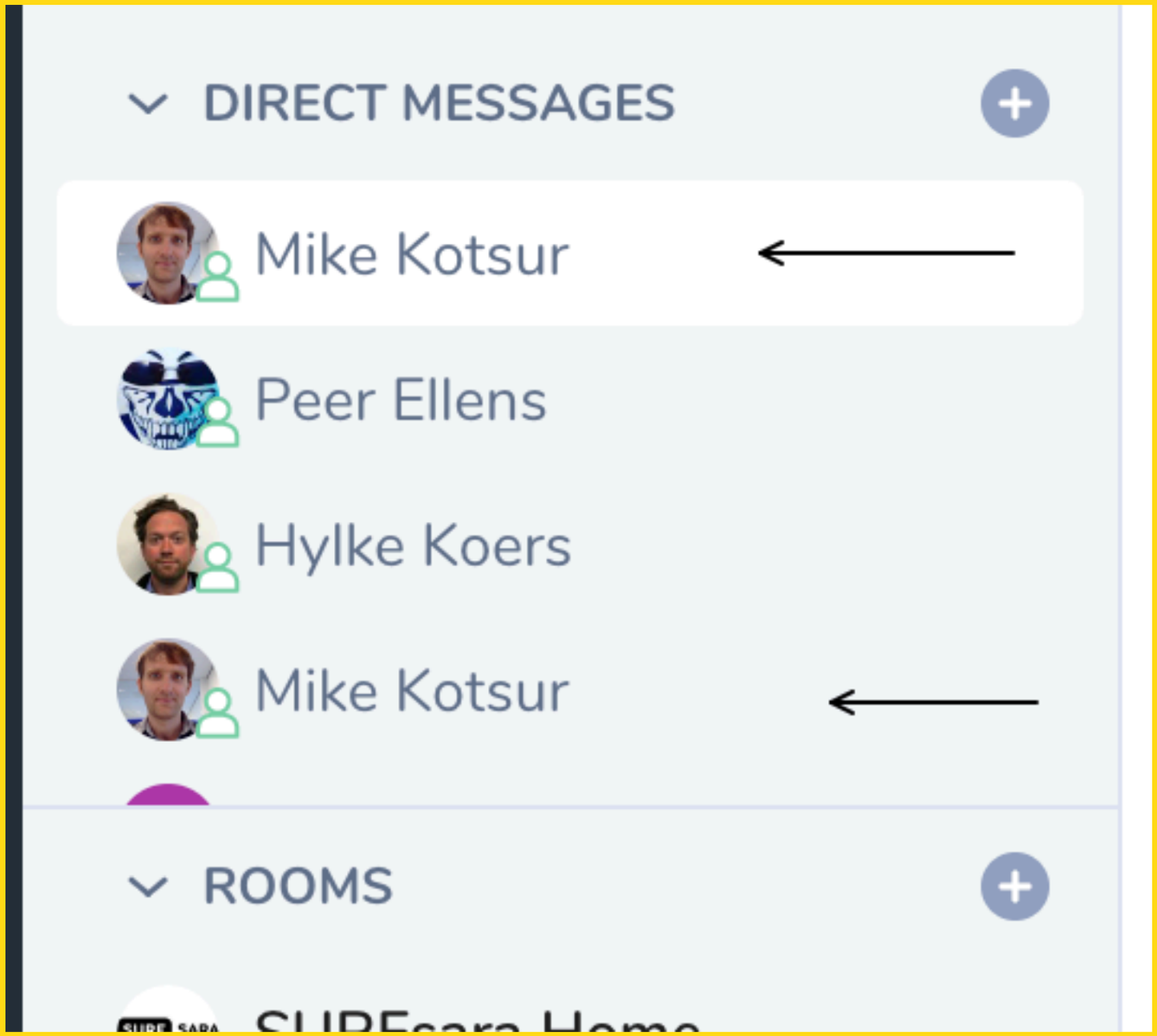
“Make illegal state unrepresentable”

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

type ChatService
trait ChatPane {

  def display()

}
```

?:

“Make illegal state irrepresentable”





Lesson 4:
Speed needs safety

- Resource is a cats-effect data structure that captures the effectful allocation of a resource (acquire), along with its finalizer (release).

- Still not 100% fool-proof, though.

Resource

Acquiring

```
val tempDirResource: Resource[IO, Path] = Resource.make(
  acquire = IO(Files.createTempDirectory("datex_"))
)(
  release = dir ⇒ IO(FileUtils.deleteDirectory(dir.toFile))
)
```

Releasing

Bracket

Acquiring

```
private val serviceIO = IO(new
FileInputStream(conf.credentialsFile)).bracket(is ⇒ IO(
  new Drive.Builder(httpTransport, jsonFactory,
    GoogleCredential.fromStream(is)
      .createScoped(scopes.asJavaCollection))
    .setApplicationName(conf.applicationName)
    .build()
)) {is ⇒ IO(is.close())}
```

Releasing

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!

ContainerEnv

```
def containerEnv(
    startContainerCmd: Messages.StartContainer
): Resource[IO, ContainerEnv] =
  tempDirResource.evalMap { tempHome ⇒
    import cats.implicits._

    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 => IO[V2]

fs2-rabbit

```
QueueResources.rabbitClientRes.use { rabbit ⇒
  rabbit.createConnectionChannel.use { implicit channel ⇒
    import io.circe.generic.auto._

    for {
     _ ← rabbit.declareQueue(queues.todo.config)
      _ ← rabbit.declareExchange(queues.todo.exchangeConfig)
      _ ← rabbit.bindQueue(
        queues.todo.config.queueName,
        queues.todo.exchangeConfig.exchangeName,
        queues.todo.routingKey
      )
      ← rabbit.declareQueue(queues.done.config)
     _ ← rabbit.declareExchange(queues.done.exchangeConfig)
     _ ← rabbit.bindQueue(
       queues.done.config.queueName,
       queues.done.exchangeConfig.exchangeNam
       queues.done.routingKey
     )
     consumer ← rabbit
       .createAutoAckConsumer(queues.todo.config.queueName)(
         channel,
         AmqpCodecs.decoder[StartContainer]
       )
     publisher ← rabbit.createPublisher(
       queues.done.exchangeConfig.exchangeName,
       queues.done.routingKey
     )(channel, AmqpCodecs.encoder[Done])
     _ ← new SecureContainerFlow(consumer, pu
   } yield ExitCode.Success
}
```

consumer:
fs2.Stream[IO, Envelope[DecodedMessage[M1]]

publisher: M2 => IO[Unit]

Defined the rest of the program as a transformation of the stream of messages.

Find a better example

```
consumer

  .map(_.payload)

  .through(AmqpCodecs.filterAndLogErrors(logger))

  .evalTap(msg ⇒ logger.info(s"Processing incoming
message $msg"))

  .evalMap { msg ⇒ … }
```

It should be you that finds the error rather than your colleague

Lesson 6:

There is no escape from **dependency management** problem

```
// More:
// https://gist.github.com/gvolpe/1454db0ed9476ed0189dcc016fd758aa
```

Lesson 5:

There is no escape from **dependency injection**

Lesson 7:
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

Lesson 7:

Introduce FP one step at a time

```
// + Explanation
// + Peer review
```

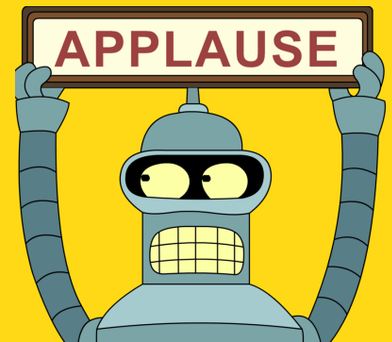
```
// + Explanation
// + Refer to overarching concepts
```

Lessons learned

* FP is very helpful in making steady progress
* Lesson 2
* Lesson 3
* Lesson 4
* Lesson 5
* One more lesson

Scala ecosystem
is great for prototyping!

- Scala - easy data modelling;
- Cats Effect - IO, parSequence(), attempt(), Fibers;
- Resources - well modelled types that became first-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;



Functional Scala makes Reactive paradigm ~~cheap~~ affordable

The Reactive Manifesto published in 2014

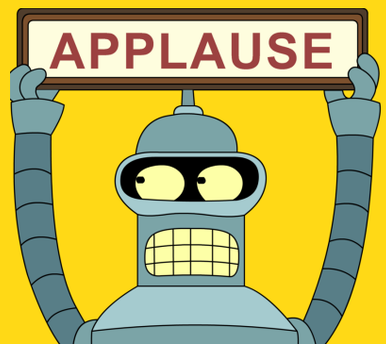
Reactive Systems are:

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively.

The Reactive Manifesto

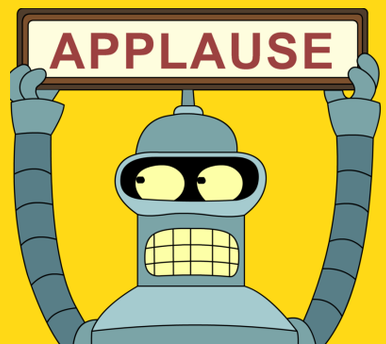
Functional Scala makes Reactive paradigm ~~cheap~~ affordable

In plain terms reactive programming is about non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale vertically (i.e. within the JVM) rather than horizontally (i.e. through clustering).



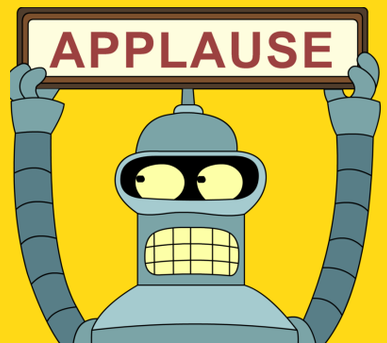
Functional Scala makes Reactive paradigm ~~cheap~~ affordable

Building on the principles of The Reactive Manifesto Akka allows you to write systems that self-heal and stay responsive in the face of failures.



Functional programming is embraced by React

If the React community embraces [hooks], it will reduce the number of concepts you need to juggle when writing React applications. Hooks let you always use functions instead of having to constantly switch between functions, classes, higher-order components, and render props.



Functional Scala makes Reactive paradigm ~~cheap~~ affordable

- **High resolution of events**
- **Asynchronous events**

In a banking application I say “update my avatar” and I don’t wait for the super slow backend systems to execute the transaction, I see something changed in the UI so that I know I did what I had to do an I can close my browser and go to bed. The browser needs to receive an event for that.

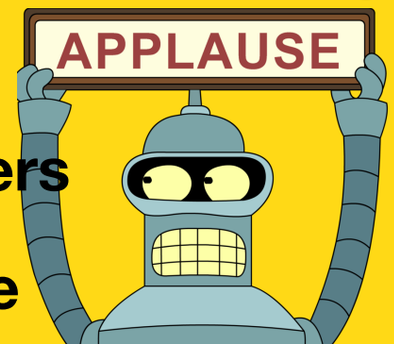
The application quickly reacts on my action

Sending message

When something happened while I wasn’t active (someone sent a message, or edited a file I’m interested in), the information just appears on my screen without me needing to hit F5 and pull this information.

The application quickly reacts the actions of other users

Receiving a message



Thank you! Questions?

aws-lambda-scala

mkotsur

@s_pcopy

More lessons learned

medium.com/@mkotsur

More about prototyping