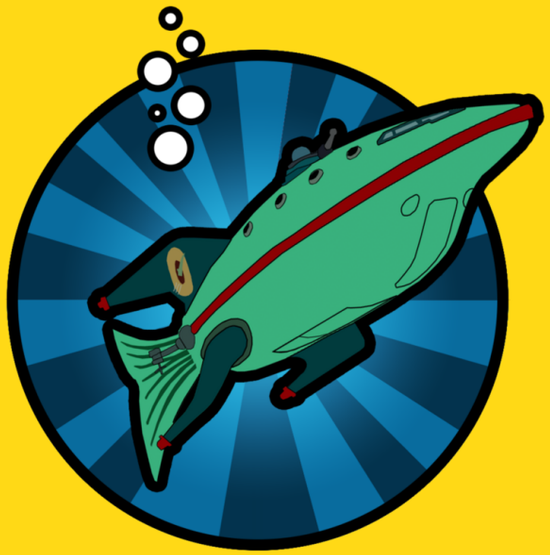
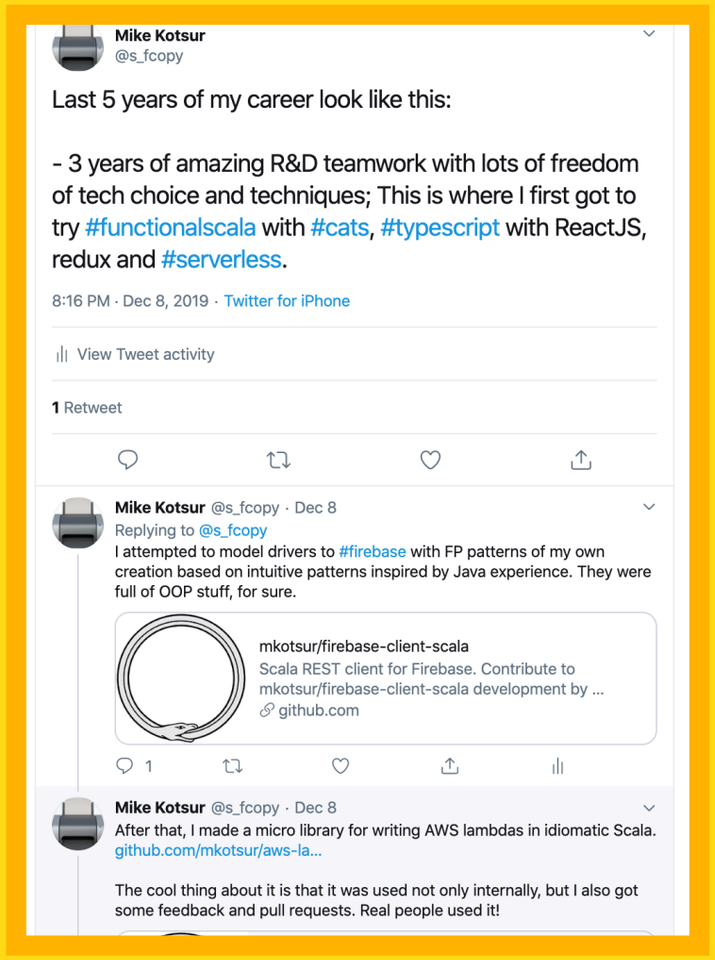
Prototyping the Future with Functional Scala



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;
- 🅾️1️⃣ Data Exchange project challenges;
- 🎉☠️ 5 Lessons I learned;
- 💪 How modern FP Scala libs help.

Programs [should] solve much more problems than it creates.

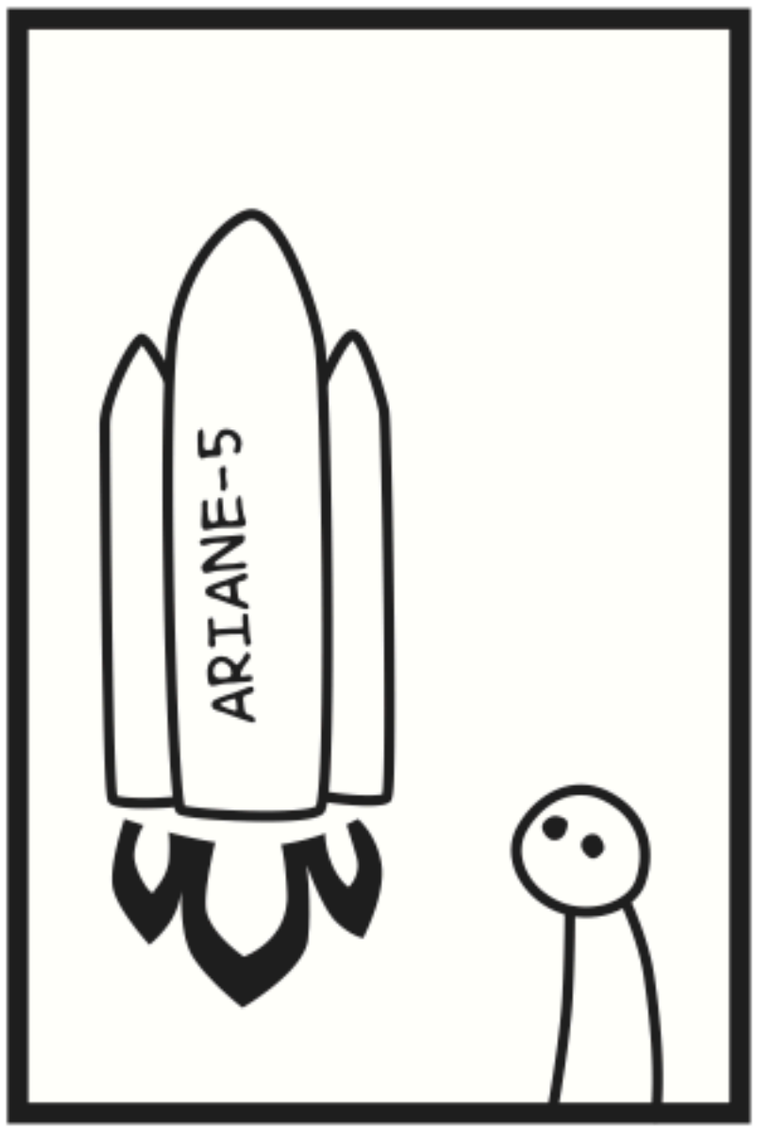
Material world

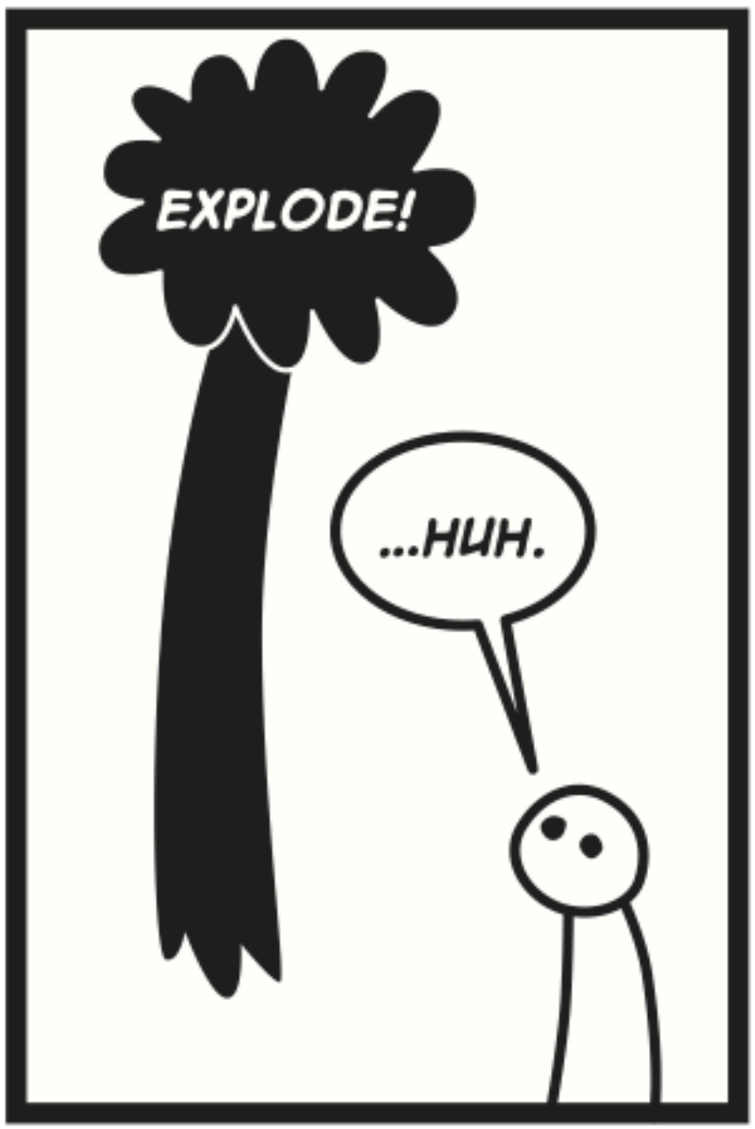
Creating

Software

Running


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


ARIANE-5

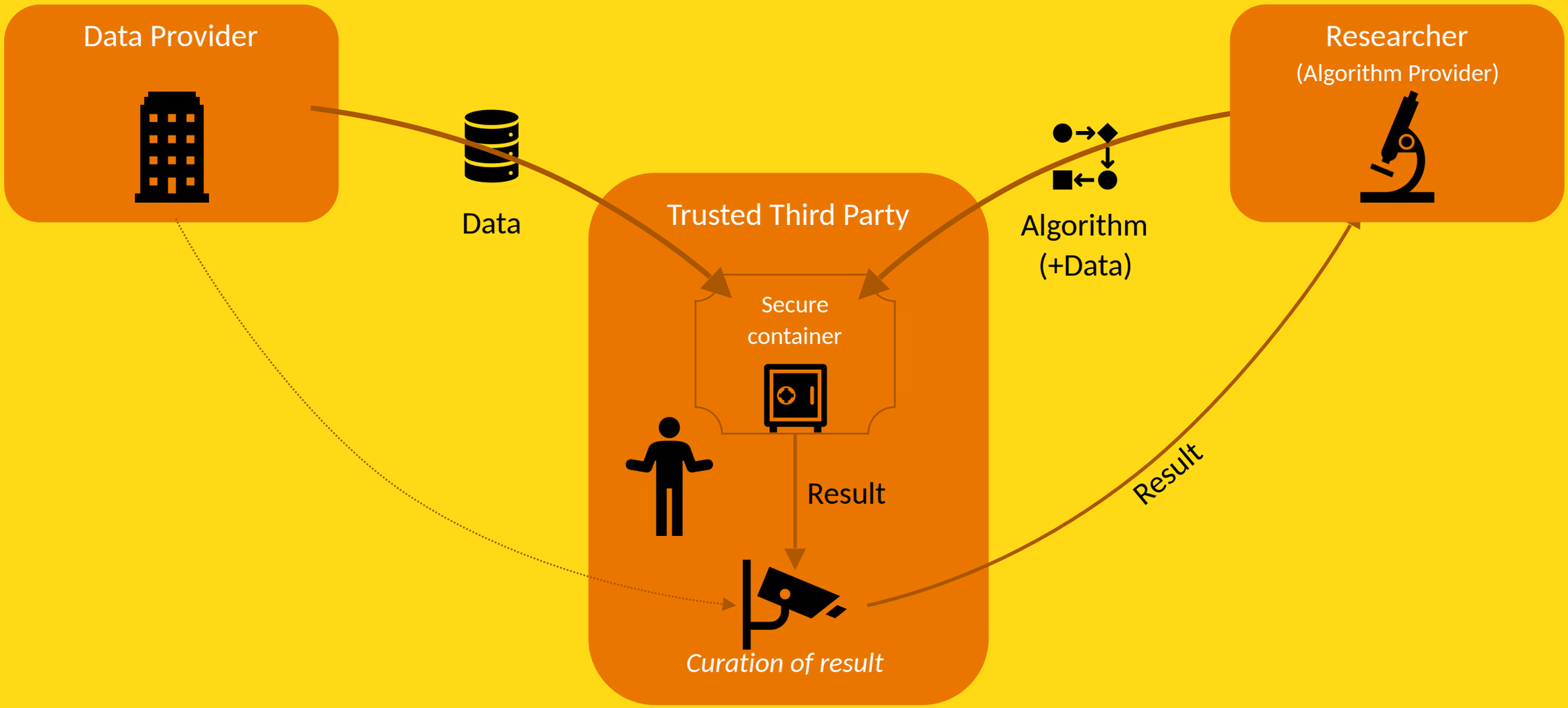

EXPLODE!
...HUH.

Software Prototyping

When you come to someone who's never heard of you before and ask for money, it's better you have a good story and good picture of what they can get in return.

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

Data Exchange

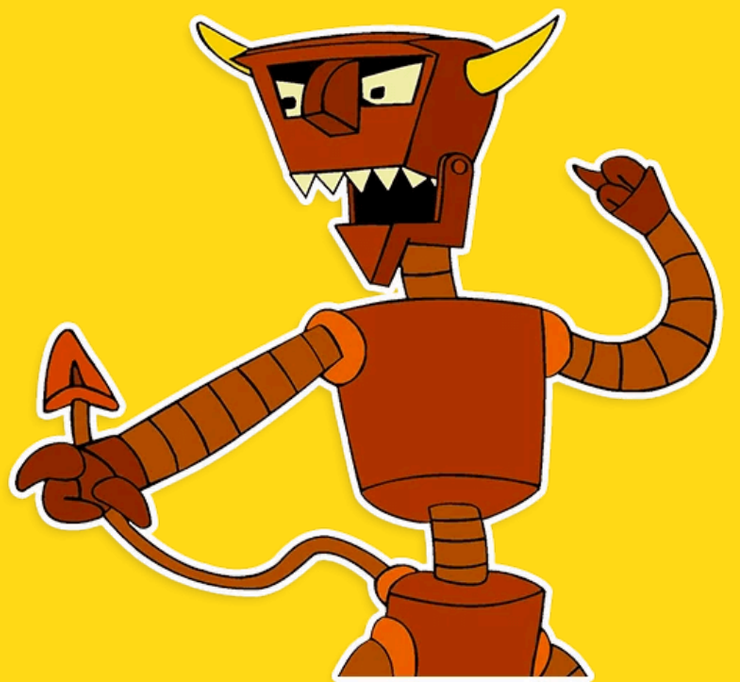


Python/Django

- 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

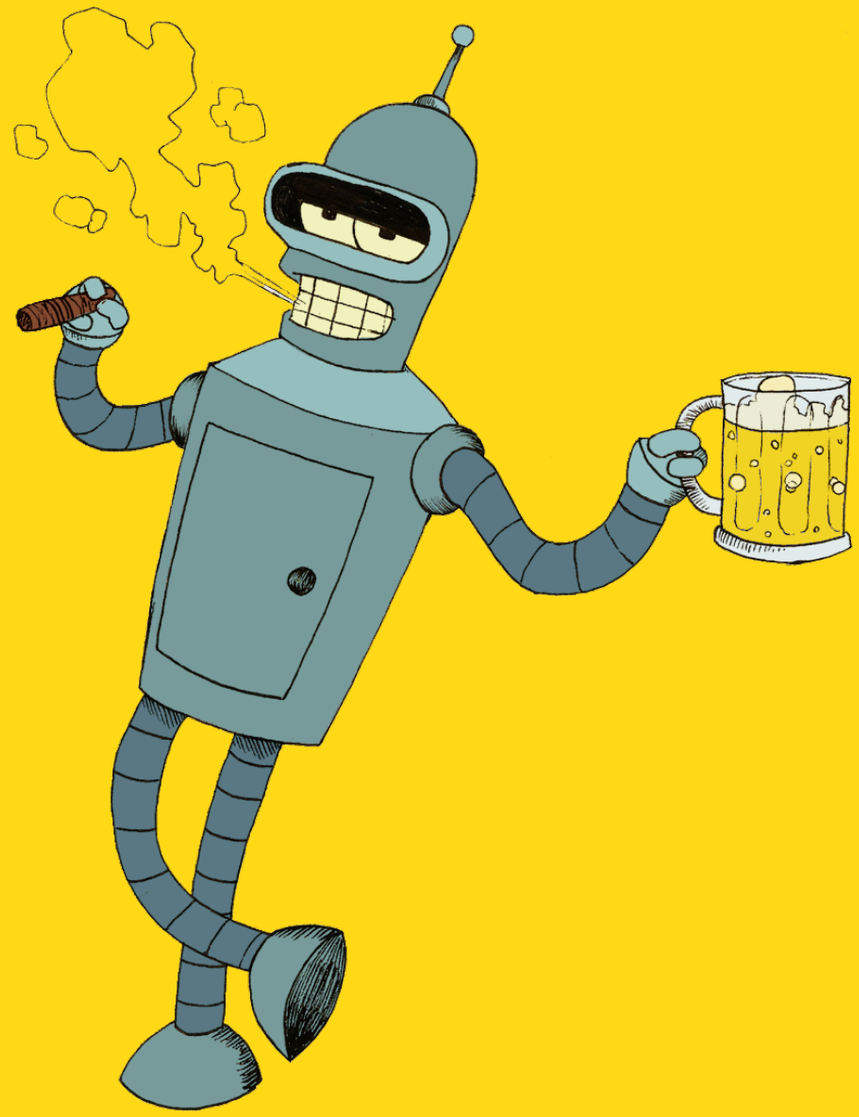
VI. Processes

- The app is executed in the execution environment as one or more processes.

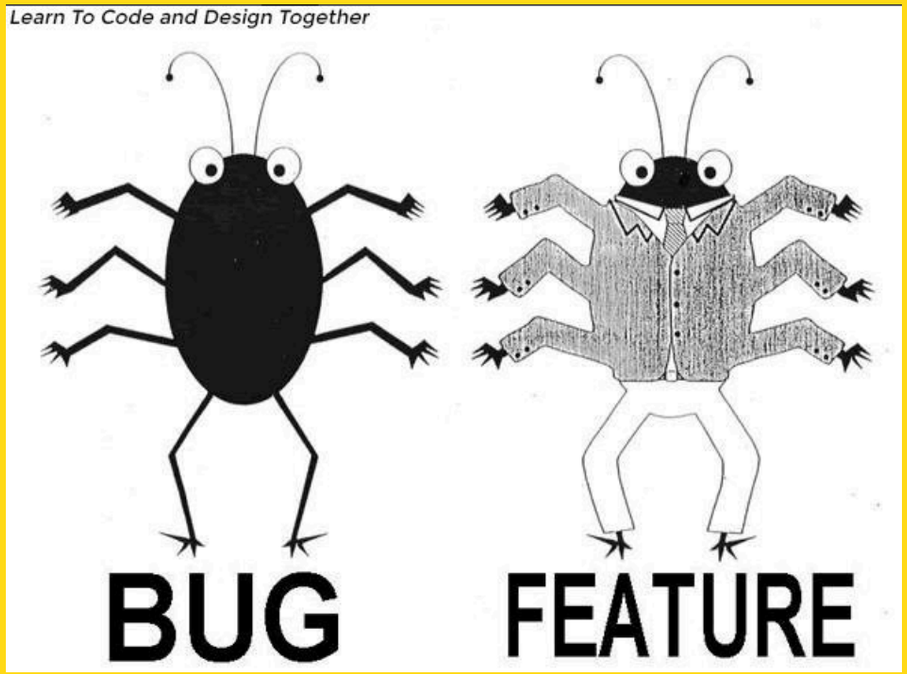
- Trying to make as much **Feedback => Improvement** iterations as possible within limited time;

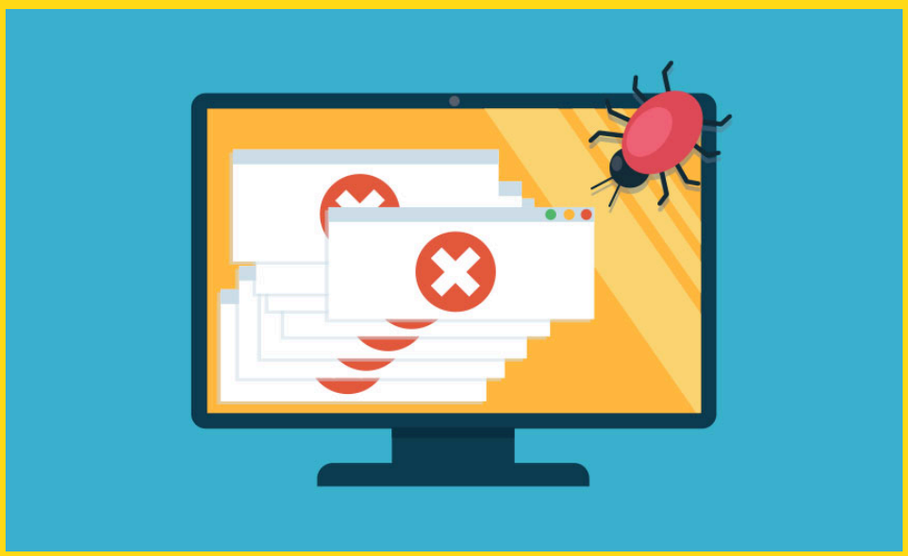
- Dynamic equilibrium: quality / speed / technical maturity.

Language doesn’t matter?

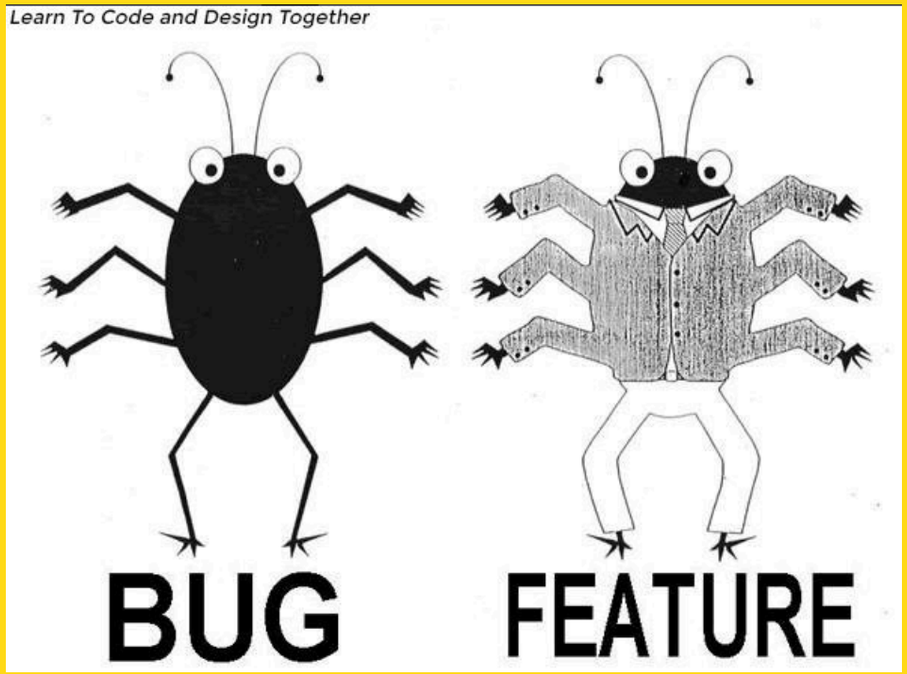


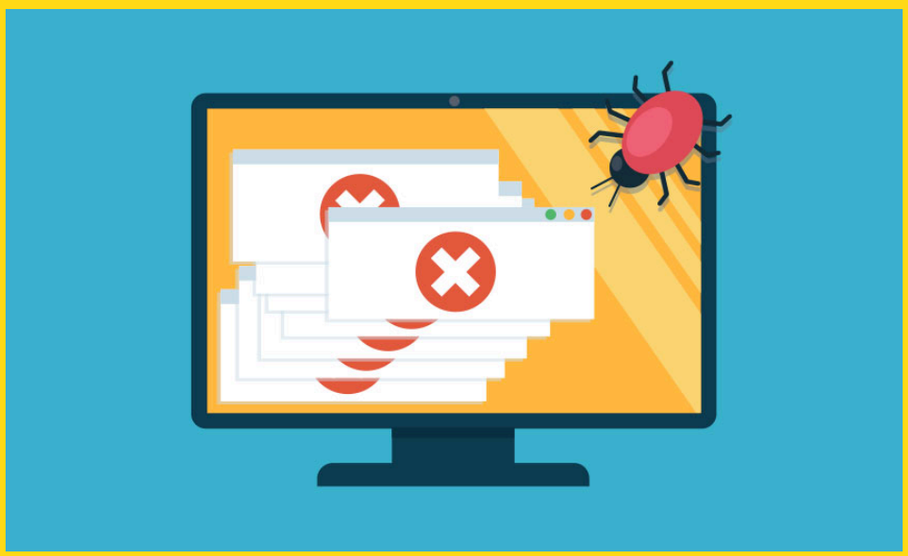
val softwareAntiUtopia =



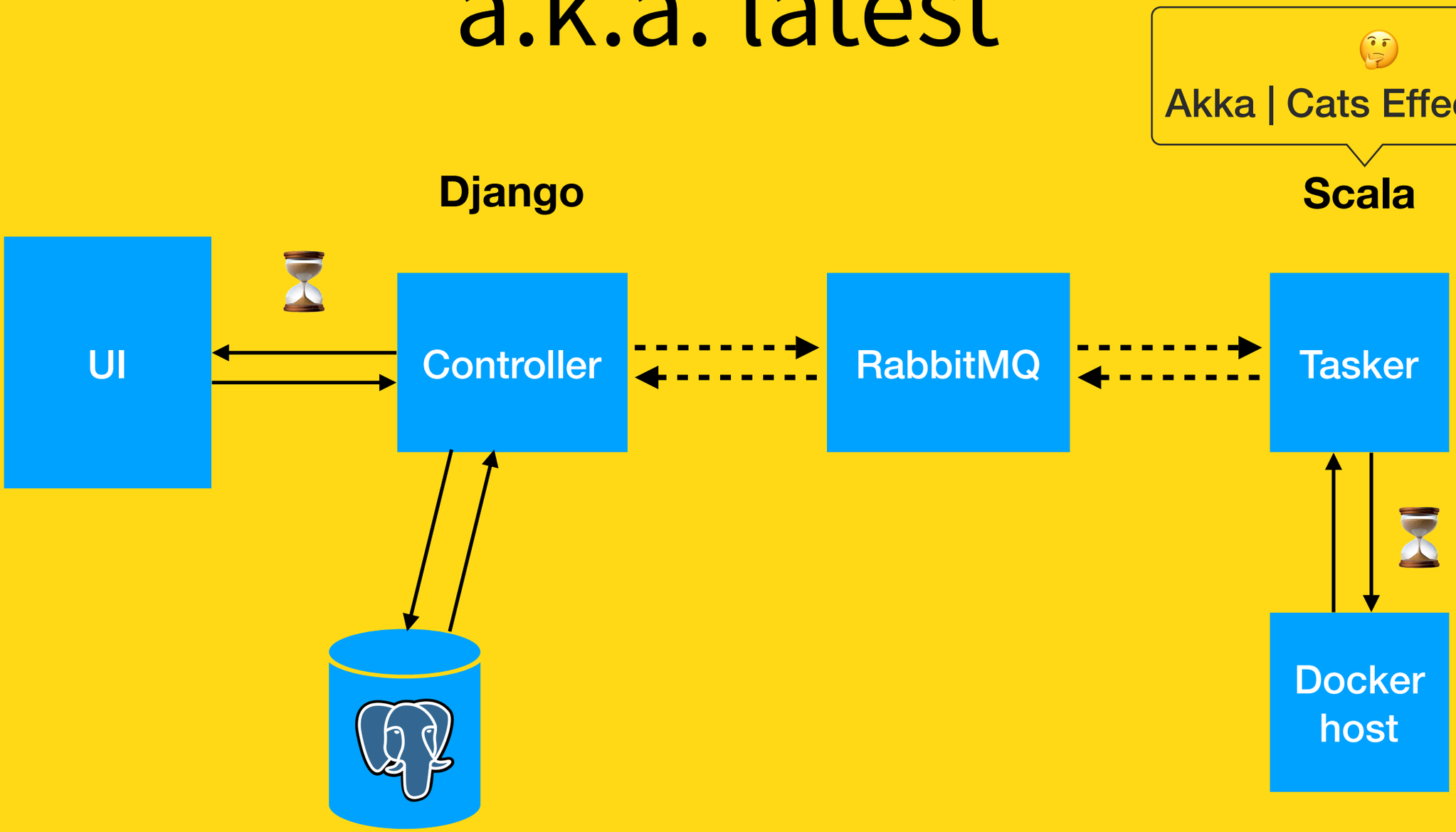


val softwareAntiUtopia =





Architecture v0.2
a.k.a. latest



0. Future is useful, but feels weird for FP

```
val result: User = DB.Users.byId(42)
```

It should be gone, for the simple reason that it's not possible to get the result instantly from a database or a service: the network is in-between, and the network is unreliable. It should be wrapped into some form of asynchronous effect, such as *Future* or even better: *F[_]*.

Reason #1: it generally doesn't force us to handle the possibility of a failure, and even if it does - it needs to be done in an intrusive and verbose way.

Reason #2: if we move this line up or down in our program, the behaviour may change significantly because of the execution time that this call takes.

Reason #3: if we happen to want to parallelise this code and something else, it's tricky.

Reason #4: if we happen to want to parallelise this code and something else, it's tricky.

-

Lesson 0:
FP is not all or nothing

“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

There are odd things with the future

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

Parallel

```scala
for {
  i <- Future { 1 }
  j <- Future { 2 }
} yield i + j
```

Sequential

Eagerly evaluated

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

def doCheapFirst[T](activities: List[Activity[T]]): Future[T] = ???

def doImportantFirst[T](activities: List[Activity[T]]): Future[T] = ???
```

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

def doCheapFirst[T](activities:
List[Activity[T]]): Future[T] = ???

def doImportantFirst[T](activities:
List[Activity[T]]): Future[T] = ???
```

Lesson 1:
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


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

Simple

Effect encoded as a pure value

```scala
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
}
```

Program blows up in a “controlled” way

Fancier

```scala
def gambleGentle: IO[Option[Jackpot]] =
  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!

Fancier

Defining new in terms of old 📖

```scala
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

Fanciest

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

```
private def gambleBig: 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]
    }
  }
```

Mind the ContextShift ⚠️

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

```
Future {
  println("Asd")
}(?: ExecutionContext)
```

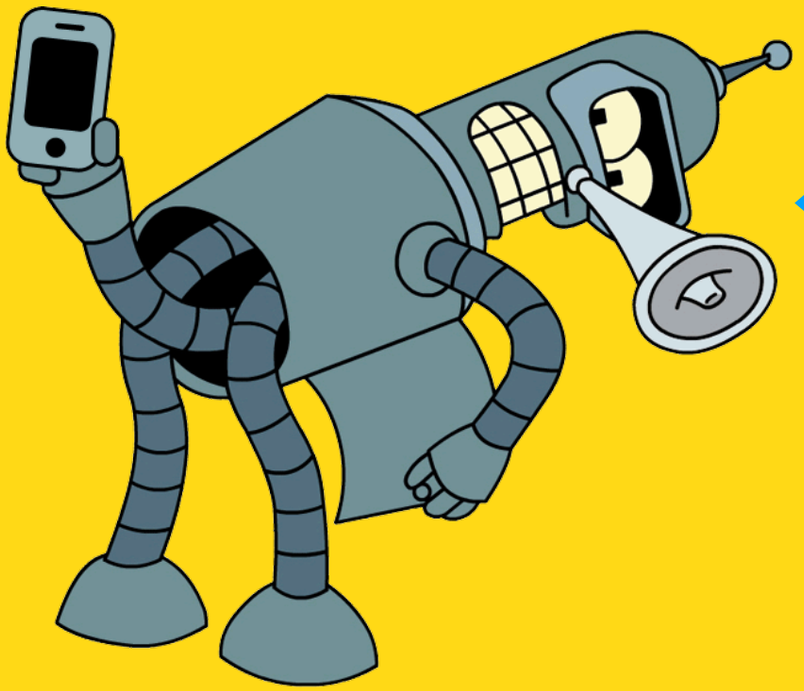
More about thread pools:

- 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 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.

Temp directory

Acquiring resource

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

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

```scala
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]

Lesson 5:
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

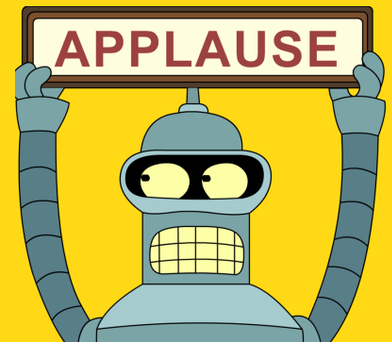
Uses implicit ContextShift

Final lesson:
Do your homework

- Don’t pitch FP or Scala, pitch a solution
- Seek agreement on:
  - importance of the problem;
  - importance of the benefits FP provides.
- Make a commitment;
- Show results often.

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

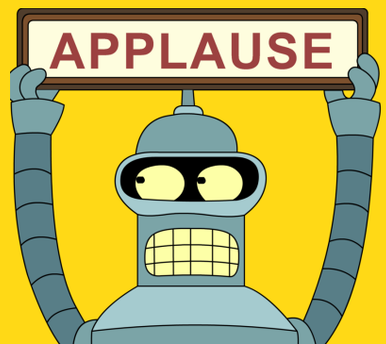
Reactive Systems are:

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

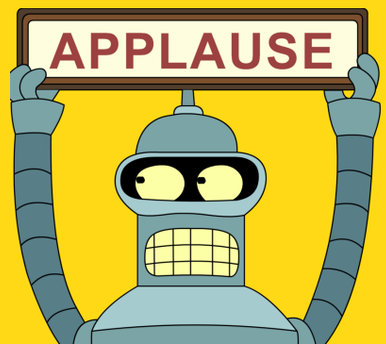
Functional Scala makes Reactive paradigm ~~cheap~~ affordable

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).



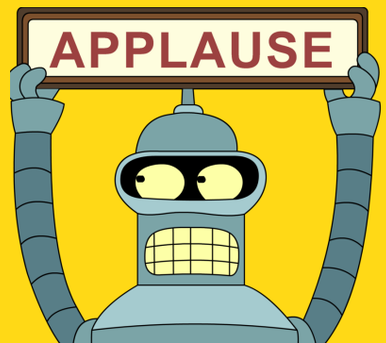
Functional Scala makes Reactive paradigm ~~cheap~~ affordable

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.

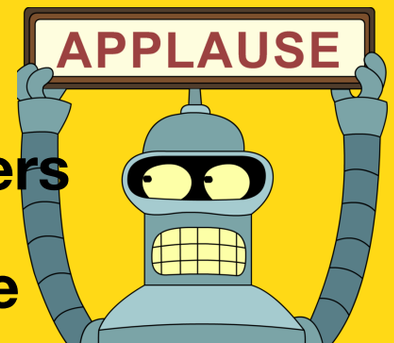
The application quickly reacts on my action

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

Lesson X:
The Config Effect

From the clean code perspective, this is better than:
1. Hardcoding strings in the place they're used
2. Not providing a fallback

But it's definitely not ideal:

1. Mixed values and logic
2. Imperative boilerplate
3. No validation

```
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 X:
The Config Effect

```
import pureconfig.generic.auto._
import pureconfig.module.catseffect.syntax._


object AppConf {

  def loadF: IO[AppConf] =
    ConfigSource.default.loadF[IO, AppConf]

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

}

case class AppConf(db: DbConf)
```

```
db {
  jdbcUrl = "jdbc:postgresql://localhost:5432/mydb"
  jdbcUrl = ${DB_JDBC_URL}

  user = "user"
  user = ${DB_USER}

  password = ""
  password = ${DB_PASSWORD}
}
```

That’s what you can do with Pureconfig

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 X:
Time to Refactor

```scala
object MyConf {

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

From the clean code perspective, this is better than:
1. Hardcoding strings in the place they're used
2. Not providing a fallback

But it's definitely not ideal:

1. Mixed values and logic
2. Imperative boilerplate
3. No validation