(IM)PRACTICAL FUNCTIONAL PROGRAMMING

ADOPTING FP IN INDUSTRY

WHY THIS TALK?

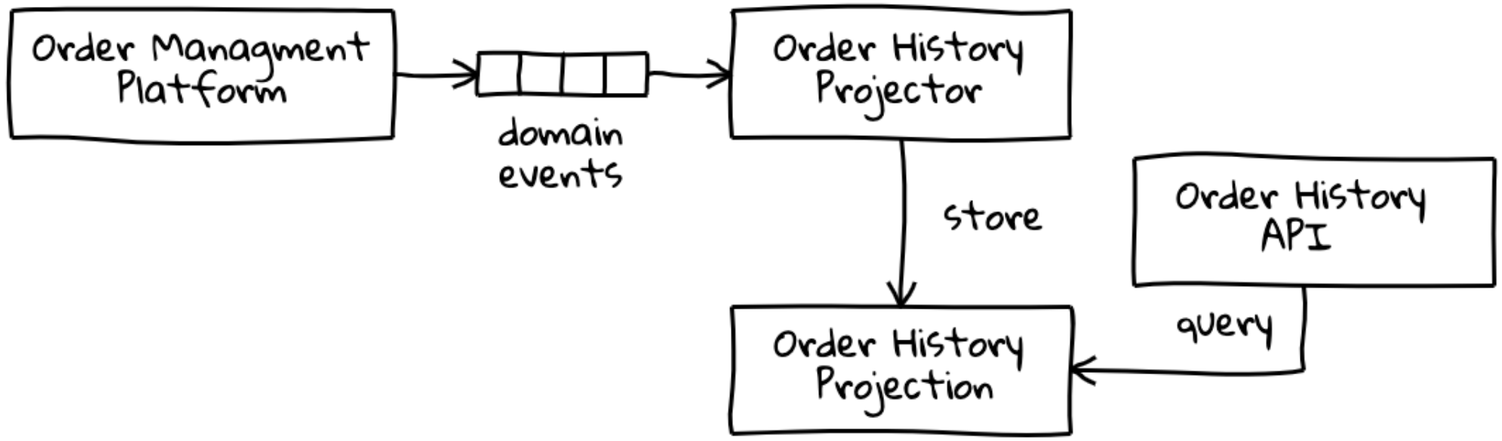
HOW MANY TIMES HAVE YOU HEARD:

- FP is too hard
- FP is not pragmatic
- FP is not suited *to deliver value to the business*

AGENDA

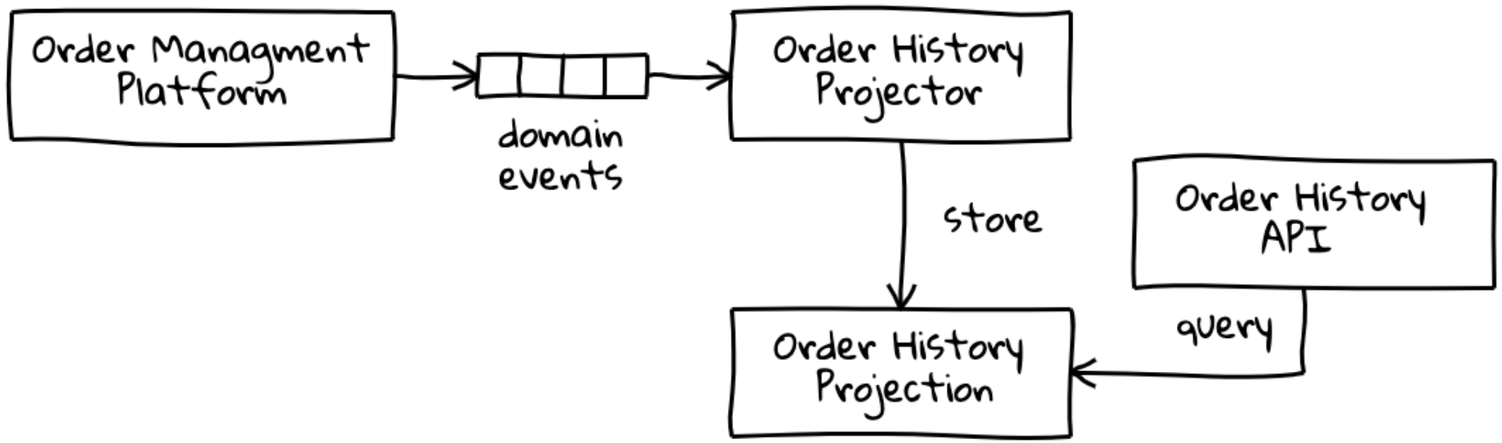
- A sample architecture
- Introduce a bunch of building blocks
- Design architecture components

SAMPLE ARCHITECTURE: *Order History Service*



- Let's assume we are provided with domain events from an Order Management Platform (e.g. OrderCreated, OrderShipped, etc..), via a RabbitMQ broker
  - We need to build an Order History Service

ORDER HISTORY SERVICE: *components*



- a component which projects a (read) model, in a MongoDB collection
- so that an HTTP service can query the collection returning orders

DISCLAIMER

Our focus here is NOT on the System Architecture

We'll just put our attention on implementing an architecture component (the projector) adopting Functional Programming principles

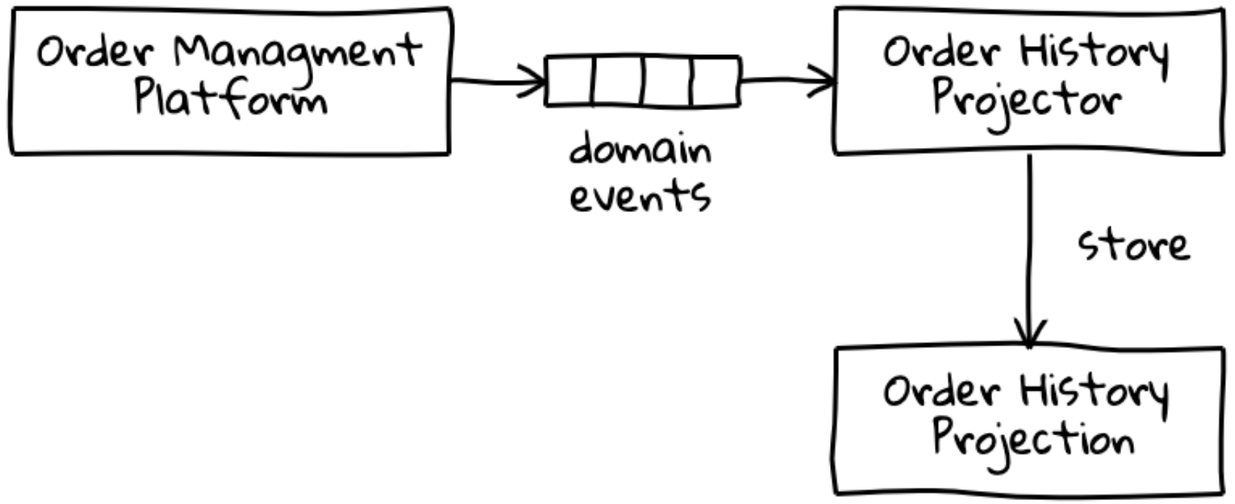
SPOILER

We WON'T be using:

- `var`
- `throw`
- methods returning `Unit`
- poorly typed definitions (`Any`, `Object`, etc...)
- low level concurrency mechanisms (`Thread`, `Actor`, etc..)

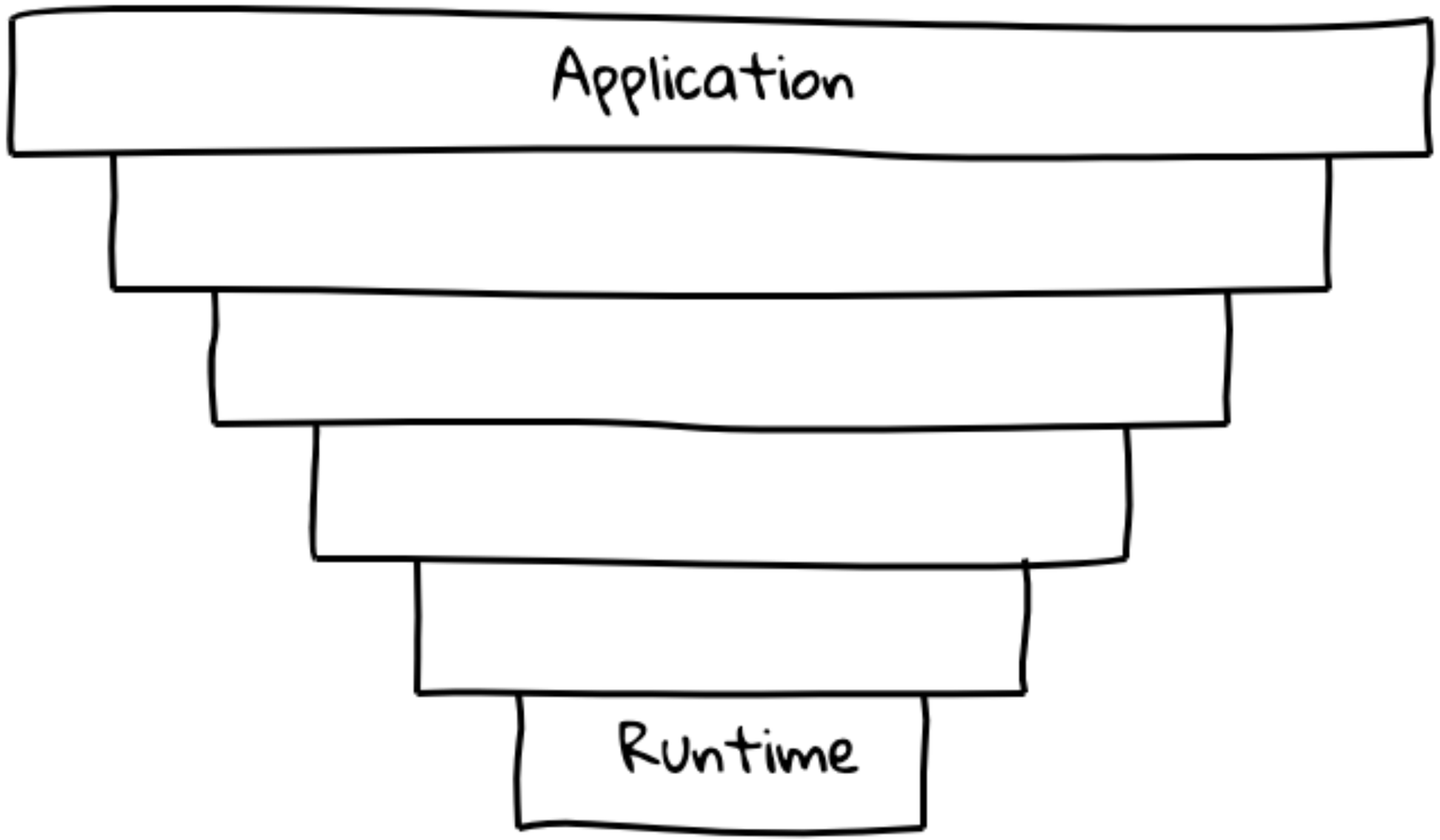
LET'S START

BUILDING A PROJECTOR



- *Consume* a stream of events from a RabbitMQ queue
- *Persist* a read model to a MongoDB collection

HOW TO FILL THE ABSTRACTION GAP?



PROJECTOR APPLICATION

1. read a bunch of configs from the env
2. interact with a RabbitMQ broker
   2.1 open a connection
   2.2 receive a Stream of events from the given queue
3. interact with a MongoDB cluster
   3.1 open a connection
   3.2 store the model to the given collection

1. READ A BUNCH OF CONFIGS FROM THE ENV

```
object Mongo {
  case class Auth(username: String, password: String)
  case class Config(auth: Auth, addresses: List[String], /*...*/)

  // reading from env variables
  lazy val config: Config = {
      val user       = System.getenv("MONGO_USERNAME")
      val password   = System.getenv("MONGO_PASSWORD")
      //...reading other env vars ... //
      Config(Auth(user, password), endpoints, port, db, collection)
  }
}
```

HOW TO TURN AN API TO BE FUNCTIONAL™?

HOW TO TURN AN API TO BE FUNCTIONAL™?

In most cases:

wrap the impure types/operations,
only *expose* a safer version of its operations

CAN FP HELP US WITH *I/O* OPERATIONS?

CAN FP HELP US WITH *side-effects*?

INTRODUCING IO

A DATA TYPE FOR *encoding effects* AS PURE VALUES

INTRODUCING IO

- enable capturing and controlling actions - a.k.a effects - that your program wishes to perform within a resource-safe, typed context with seamless support for concurrency and coordination
- these effects may be asynchronous (callback-driven) or synchronous (directly returning values); they may return within microseconds or run infinitely.

INTRODUCING IO

A value of type `IO[A]` is a computation that, when evaluated, can perform *effects* before either

- yielding exactly one **result**: a value of type `A`
- raising a **failure** (`Throwable`)

IO VALUES

- are pure and immutable
- represents just a description of a side effectful computation
- are not evaluated (suspended) until the *end of the world*
- respects referential transparency

REFERENTIAL TRANSPARENCY

An expression may be replaced by its value (or anything having the same value)
without changing the result of the program

WHY DO WE NEED REFERENTIAL TRANSPARENCY IF IT WORKS ANYWAY?

```
def askInt(): Future[Int] =
  Future(println("Please, give me a number:"))
    .flatMap(_ => Future(io.StdIn.readLine().toInt))

def askTwoInt(): Future[(Int, Int)] =
  for {
    x <- askInt()
    y <- askInt()
  } yield (x , y)

def program(): Future[Unit] =
  askTwoInt()
    .flatMap(pair => Future(println(s"Result: ${pair}")))
```

```
> Output:
> Please, give me a number:
> 4
> Please, give me a number:
> 7
> Result: (4,7)
```

WHY DO WE NEED REFERENTIAL TRANSPARENCY IF IT WORKS ANYWAY?

```scala
def askInt(): Future[Int] =
  Future(println("Please, give me a number:"))
    .flatMap(_ => Future(io.StdIn.readLine().toInt))

def askTwoInt(): Future[(Int, Int)] =
  val sameAsk = askInt()
  for {
    x <- sameAsk
    y <- sameAsk
  } yield (x , y)

def program(): Future[Unit] =
  askTwoInt()
    .flatMap(pair => Future(println(s"Result: ${pair}")))
```

```
> Output:
> Please, give me a number:
> 4
> Result: (4,4)
```

We just wanted to reduce duplication through an extract var![1]

[1] Example gently stolen from my dear friend https://github.com/matteobaglini/onion-with-functional-programming

REFERENTIAL TRANSPARENCY

- code easier to reason about
- code easier to refactor
- code easier to compose

- we're already used to referential transparency since our math lessons!
- we're already using a lot of data types in a referential transparent manner (e.g. `List`, `Option`, `Try`, `Either`)!

HOW TO TURN AN API TO BE FUNCTIONAL™?

In most cases:

wrap the impure types/operations,
only *expose* a safer version of its operations
which will need to be referential transparent

IO AND COMBINATORS

```
object IO {
  def delay[A](a: => A): IO[A]
  def pure[A](a: A): IO[A]
  def raiseError[A](e: Throwable): IO[A]
  def sleep(duration: FiniteDuration): IO[Unit]
  def async[A](k: /* ... */): IO[A]
  ...
}
```

```
class IO[A] {
  def map[B](f: A => B): IO[B]
  def flatMap[B](f: A => IO[B]): IO[B]
  def *>[B](fb: IO[B]): IO[B]
  ...
}
```

COMPOSING SEQUENTIAL EFFECTS

```scala
val ioInt: IO[Int] =
  IO.delay { println("hello") }
    .map(_ => 1)

val program: IO[Unit] =
 for {
    i1 <- ioInt
    _  <- IO.sleep(i1.second)
    _  <- IO.raiseError( // not throwing!
            new RuntimeException("boom!"))
    i2 <- ioInt //comps is short-circuted
 } yield ()
```

```
> Output:
> hello
> <...1 second...>
> RuntimeException: boom!
```

WE ARE PRACTICAL

1. READ A BUNCH OF CONFIGS FROM THE ENV, MADE FUNCTIONAL™

```scala
object Mongo {
  case class Auth(username: String, password: String)
  case class Config(auth: Auth, addresses: List[String], /*...*/)

  object Config {
    // a delayed computation which read from env variables
    val load: IO[Config] =
      for {
        user     <- IO.delay(System.getenv("MONGO_USERNAME"))
        password <- IO.delay(System.getenv("MONGO_PASSWORD"))
        //...reading other env vars ... //
      } yield Config(Auth(user, password), endpoints, port, db, collection)
  }
}
```

COMPOSING EFFECTS

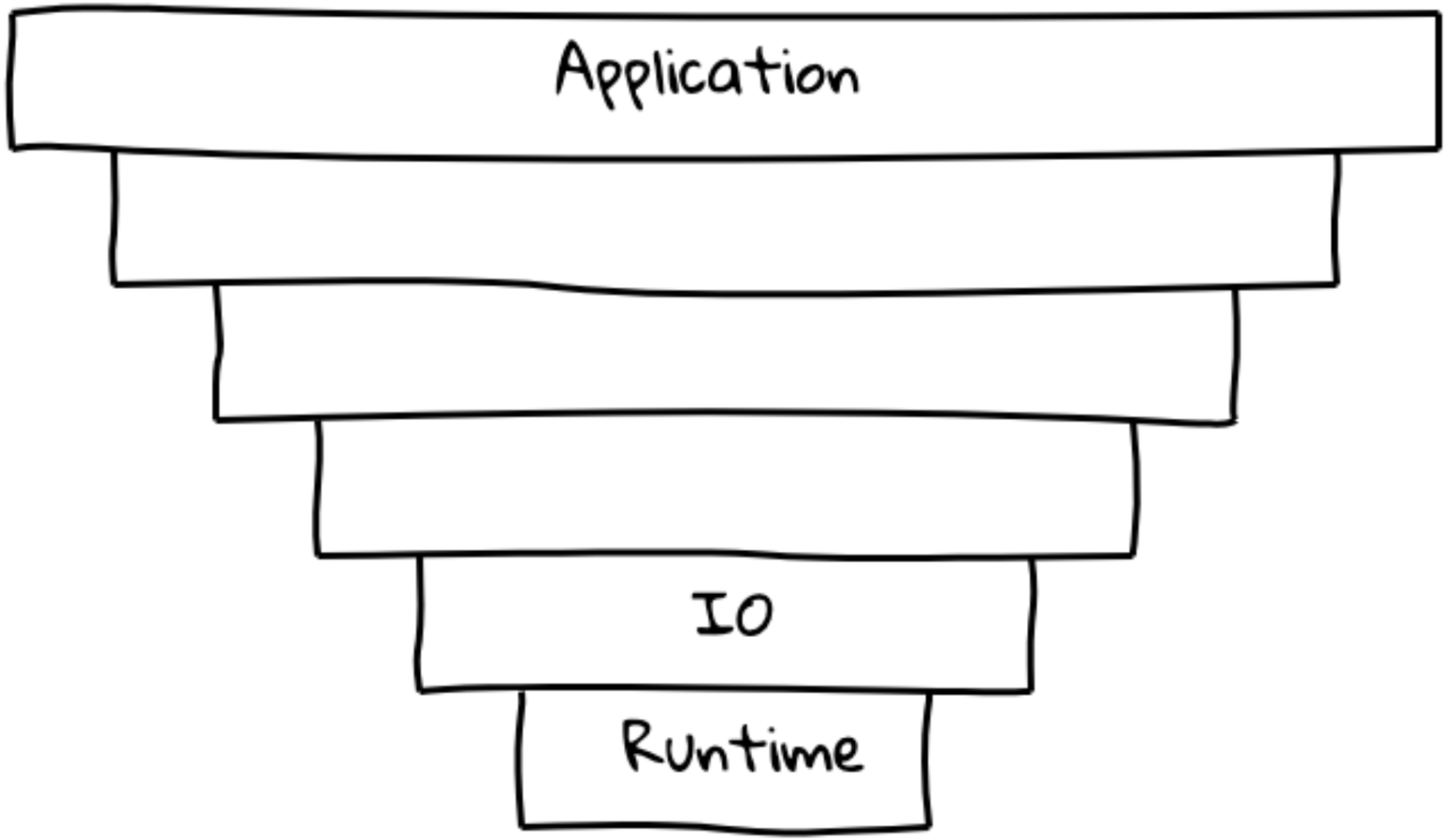
```
val ioOps =
 for {
    mongoConfig  <- Mongo.Config.load
    rabbitConfig <- Rabbit.Config.load
    // TODO use configs to do something!
 } yield ()
```

HOW IO VALUES ARE EXECUTED?

If IO values are just a description of effectful computations which can be composed and so on...

Who's gonna *run* the suspended computation then?

HOW TO FILL THE ABSTRACTION GAP?



END OF THE WORLD

- `IOApp` describes a main which executes an `IO`
- as the single entry point to a *pure* program.

```
object OrderHistoryProjectorApp extends IOApp.Simple {
  override def run: IO[Unit] =
    for {
      mongoConfig  <- Mongo.Config.load
      rabbitConfig <- Rabbit.Config.load
      // TODO use configs to start the main logic!
    } yield ()
}
```

PROJECTOR APPLICATION

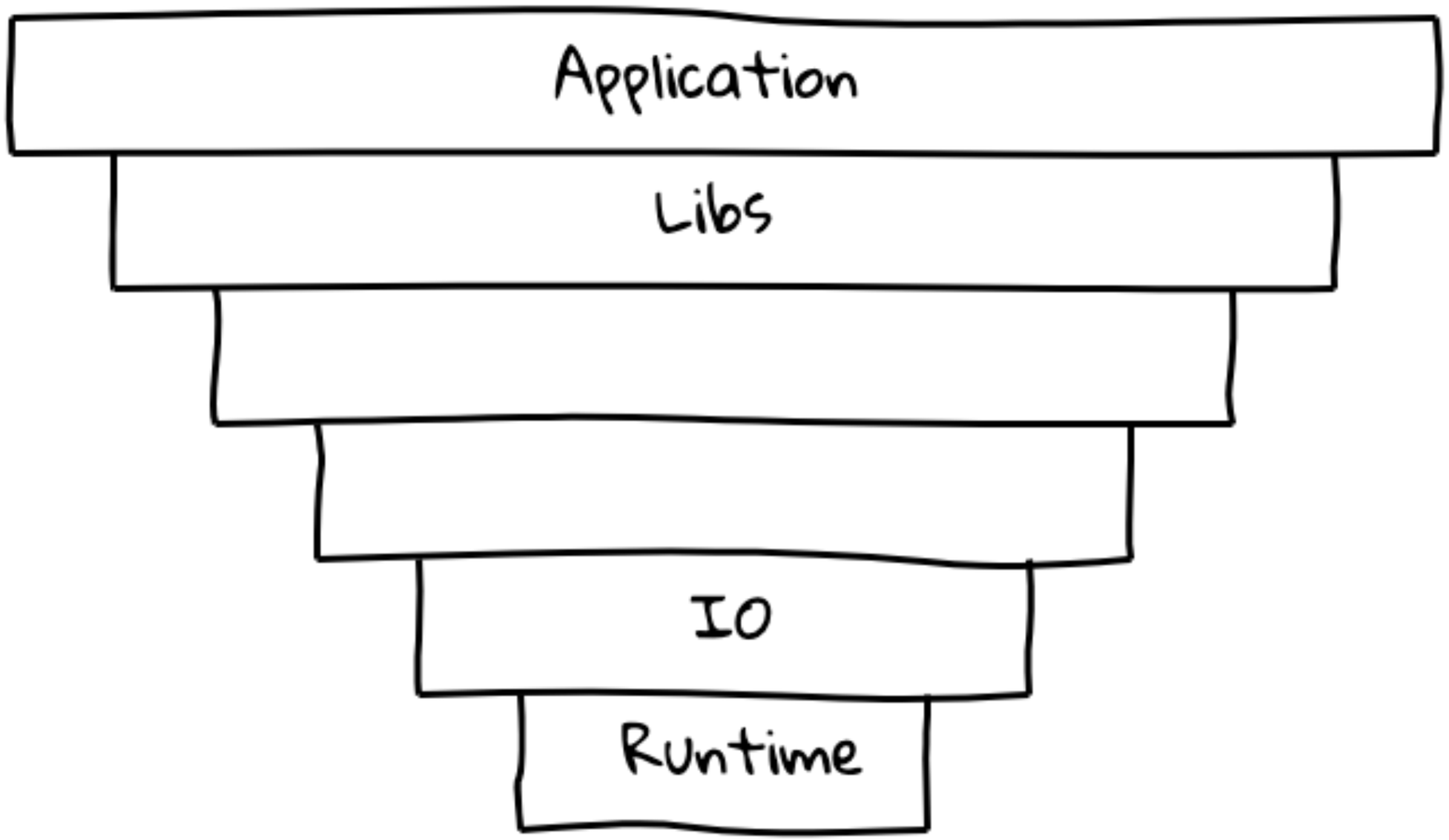
1. ~~read a bunch of configs from the env~~
2. interact with a RabbitMQ broker
   2.1 open a connection
   2.2 receive a Stream of events from the given queue
3. interact with a MongoDB cluster
   3.1 open a connection
   3.2 store the model to the given collection

2. INTERACT WITH A RABBITMQ BROKER

Using `fs2-rabbit` lib which:

- provides a purely functional api
- let me introduce you a bunch of useful data types

HOW TO FILL THE ABSTRACTION GAP?



2.1. INTERACT WITH A RABBITMQ BROKER

OPEN A CONNECTION

```
val channel: Resource[AMQPChannel] =
  for {
    rabbitClient <- RabbitClient.resource(config)
    channel      <- rabbitClient.createConnectionChannel
  } yield channel
```

RESOURCE?

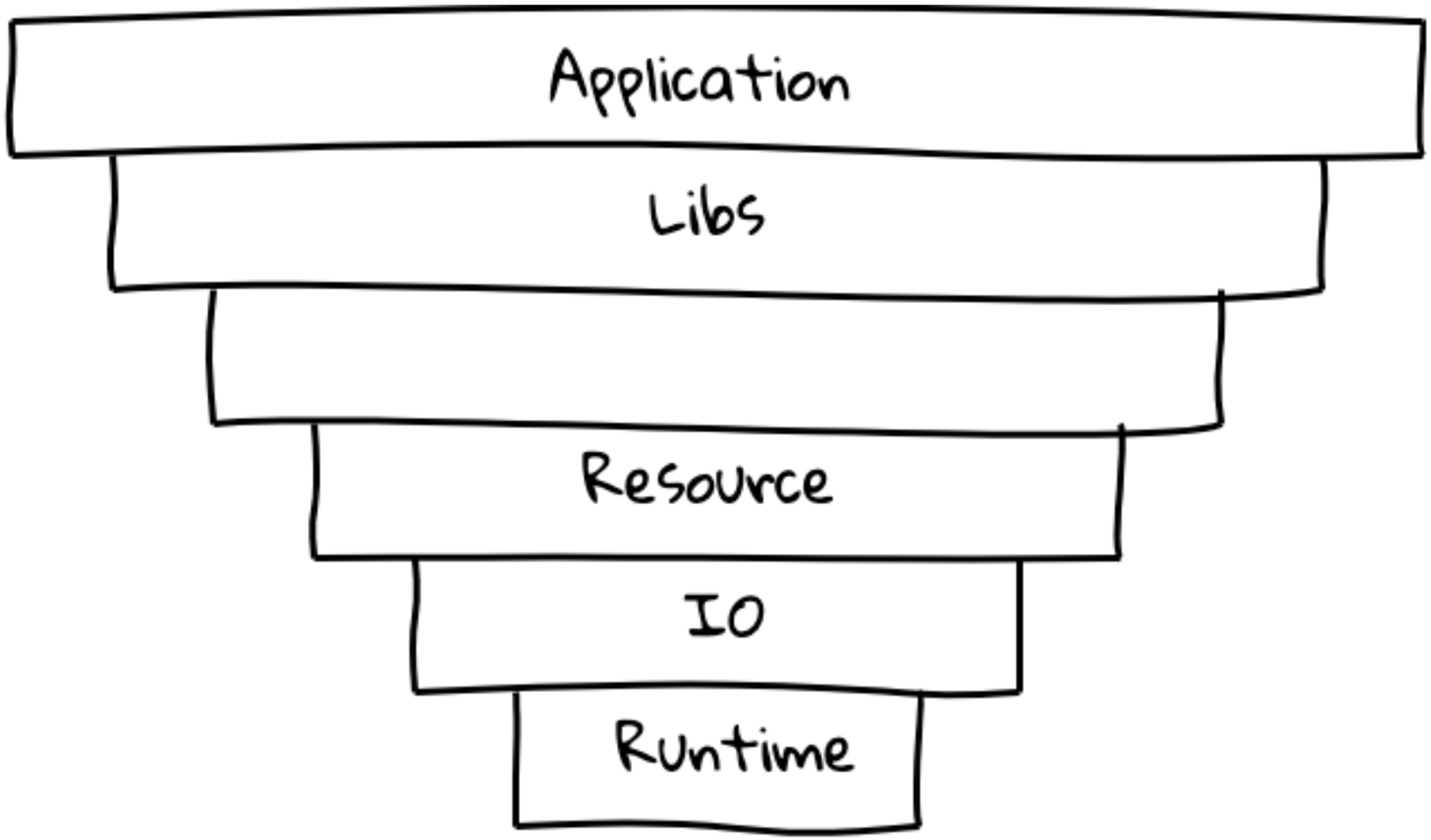
INTRODUCING RESOURCE

EFFECTFULLY ALLOCATES AND RELEASES A RESOURCE

EXTREMELY HELPFUL TO WRITE CODE THAT:

- doesn't leak
- handles properly terminal signals (e.g. `SIGTERM`) by default (no need to register a shutdown hook)
- do the right thing™ by design
- avoid the need to reboot a container every once in a while :)

HOW TO FILL THE ABSTRACTION GAP?



INTRODUCING RESOURCE

```
object Resource {
  def make[A](
    acquire: IO[A])(
    release: A => IO[Unit]): Resource[A]
}

class Resource[A] {
  def use[B](f: A => IO[B]): IO[B]

  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]
  ...
}
```

NB: not actual code, just a simplification sticking with IO type

MAKING A RESOURCE

```
def mkResource(s: String): Resource[String] = {
  val acquire =
    IO.delay(println(s"Acquiring $s")) *> IO.pure(s)

  def release(s: String) =
    IO.delay(println(s"Releasing $s"))

  Resource.make(acquire)(release)
}
```

USING A RESOURCE

```
val r: Resource[(String, String)] =
  for {
    outer <- mkResource("outer")
    inner <- mkResource("inner")
  } yield (outer, inner)

r.use { case (a, b) =>
  IO.delay(println(s"Using $a and $b"))
} // IO[Unit]
```

```
Output:
> Acquiring outer
> Acquiring inner
> Using outer and inner
> Releasing inner
> Releasing outer
```

USING A RESOURCE

```
val sessionPool: Resource[MySessionPool] =
  for {
    connection <- openConnection()
    sessions   <- openSessionPool(connection)
  } yield sessions

sessionPool.use { sessions =>
  // use sessions to do whatever things!
}
```

```
Output:
> Acquiring connection
> Acquiring sessions
> Using sessions
> Releasing sessions
> Releasing connection
```

GOTCHAS:

- Nested resources are released in reverse order of acquisition
  - Easy to lift an `AutoClosable` to `Resource`, via `Resource.fromAutoclosable`
  - Every time you need to use something which implements `AutoClosable`, you should really be using `Resource`!
- You can lift any `IO[A]` into a `Resource[A]` with a no-op release via `Resource.eval`

WHY NOT SCALA.UTIL.USING?

- not composable (no `map`, `flatMap`, etc...)
  - no support for properly handling effects

WE ARE PRAGMATIC

2.1. INTERACT WITH A RABBITMQ BROKER

```
object Rabbit {
  //...
  def consumerFrom(
      config: Fs2RabbitConfig,
      decoder: EnvelopeDecoder[IO, Try[OrderCreatedEvent]]
    ): Resource[(Acker, Consumer)] =
      for {
        rabbitClient <- RabbitClient.resource(config)
        channel      <- rabbitClient.createConnectionChannel
        (acker, consumer) <- Resource.eval(
                               rabbitClient.createAckerConsumer(
                                 queueName = QueueName("EventsFromOms"),
                                 channel = channel,
                                 decoder = decoder
                               ) // IO[(Acker, Consumer)]
                             )
      } yield (acker, consumer)

  type Acker    = AckResult => IO[Unit]
  type Consumer = Stream[AmqpEnvelope[Try[OrderCreatedEvent]]]
}
```

I HEAR YOU...

```
type Consumer =
  Stream[AmqpEnvelope[Try[OrderCreatedEvent]]]
```

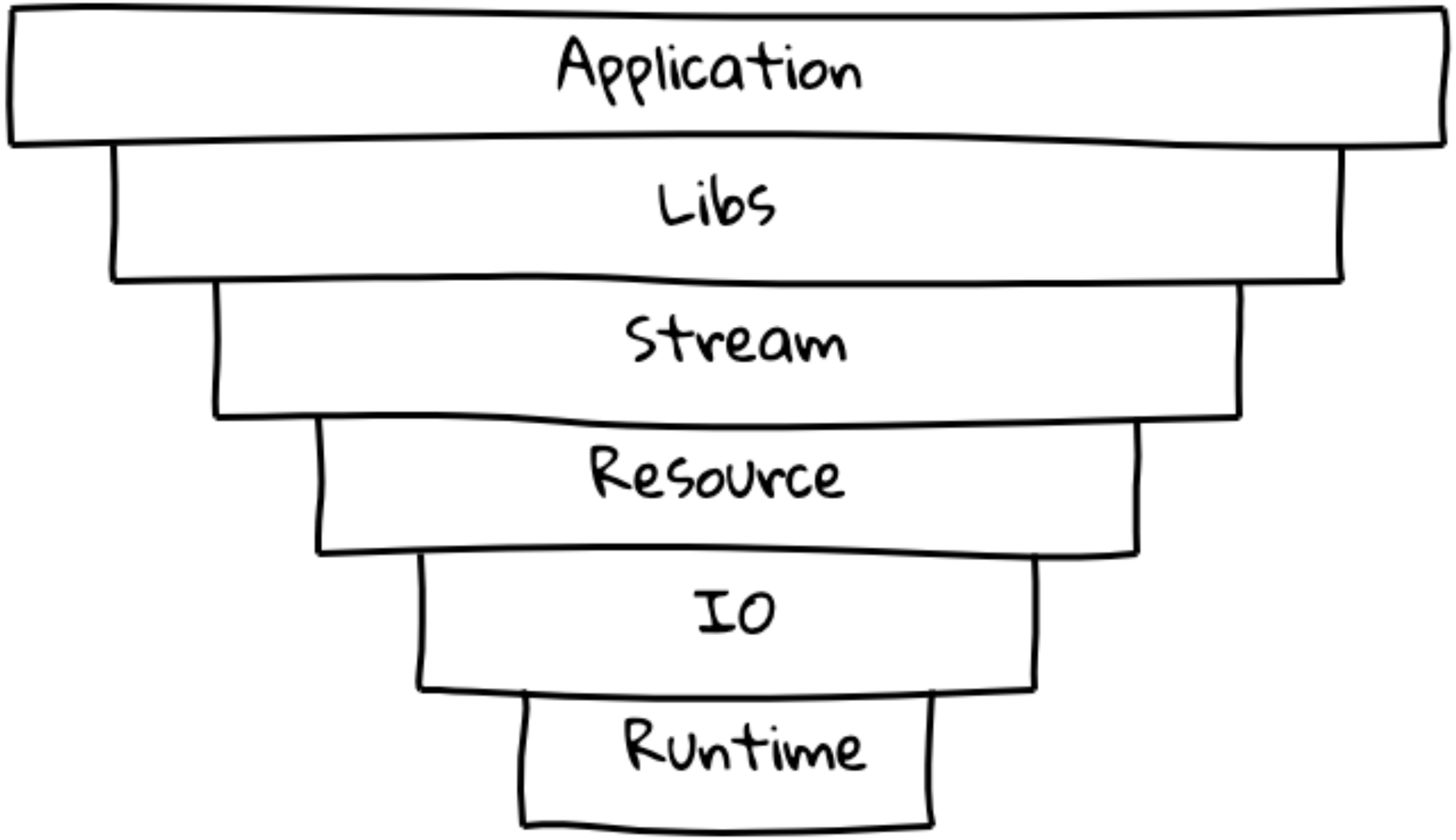
INTRODUCING STREAM

A SEQUENCE OF EFFECTFUL COMPUTATION

INTRODUCING STREAM

- Simplify the way we write concurrent streaming consumers
- Pull-based, a consumer pulls its values by repeatedly performing pull steps

HOW TO FILL THE ABSTRACTION GAP?



INTRODUCING STREAM

A stream producing output of type `O` and which may evaluate `IO` effects.

```
object Stream {
  def emit[A](a: A): Stream[A]
  def emits[A](as: List[A]): Stream[A]
  def eval[A](f: IO[A]): Stream[A]
  ...
}

class Stream[O]{
  def evalMap[O2](f: O => IO[O2]): Stream[O2]
  ...
  def map[O2](f: O => O2): Stream[O2]
  def flatMap[O2](f: O => Stream[O2]): Stream[O2]
}
```

NB: not actual code, just a simplification sticking with IO type

INTRODUCING STREAM

A sequence of effects...

```
Stream(1,2,3)
  .repeat
  .evalMap(i => IO.delay(println(i))
  .compile
  .drain
```

WE DELIVER

CONSUMING A STREAM

```
class OrderHistoryProjector(consumer: Consumer, acker: Acker, logger: Logger) {
  val project: IO[Unit] =
    consumer.evalMap { envelope =>
      envelope.payload match {
        case Success(event) =>
          logger.info("Received: " + envelope) *>
            acker(AckResult.Ack(envelope.deliveryTag))
        case Failure(e) =>
          logger.error(e)("Error while decoding") *>
            acker(AckResult.NAck(envelope.deliveryTag))
      }
    }
     .compile.drain
}
```

PROJECTOR APPLICATION

1. ~~read a bunch of configs from the env~~
2. ~~interact with a RabbitMQ broker~~
   - 2.1 ~~open a connection~~
   - 2.2 ~~receive a Stream of events from the given queue~~
3. interact with a MongoDB cluster
   - 3.1 open a connection
   - 3.2 store the model to the given collection

3. INTERACT WITH A MONGODB CLUSTER

Using `mongo4cats`, a thin wrapper over the official mongodb driver, which is exposes purely functional apis

3.1 OPEN A CONNECTION

```
object Mongo {
  ...
  def collectionFrom(conf: Config): Resource[MongoCollection[Order]] = {
    val clientSettings = ??? // conf to mongo-scala-driver settings

    for {
      client     <- MongoClient.create(settings.build())
      db         <- Resource.eval(client.getDatabase(conf.databaseName))
      collection <- Resource.eval(db.getCollectionWithCirceCodecs[Order](conf.collectionName))
    } yield collection
  }
}
```

PROJECTOR APPLICATION

1. ~~read a bunch of configs from the env~~
2. ~~interact with a RabbitMQ broker~~
   - 2.1 ~~open a connection~~
   - 2.2 ~~receive a Stream of events from the given queue~~
3. interact with a MongoDB cluster
   - 3.1 ~~open a connection~~
   - 3.2 store the model to the given collection

3.2 STORE THE MODEL TO THE GIVEN COLLECTION

```
class EventRepository(collection: MongoCollection[Order]) {
  def store(event: OrderCreatedEvent): IO[Unit] =
    collection
      .insertOne(
        Order(
          orderNo = OrderNo(event.id),
          company = Company(event.company),
          email = Email(event.email),
          lines = event.lines.map(...)
        )
      )
}
```

3.2 STORE THE MODEL TO THE GIVEN COLLECTION

```
class OrderHistoryProjector(eventRepo: EventRepository,
                            consumer: Consumer,
                            acker: Acker,
                            logger: Logger) {
  val project: IO[Unit] =
    consumer.evalMap { envelope =>
      envelope.payload match {
        case Success(event) =>
          logger.info("Received: " + envelope) *>
            eventRepo.store(event) *>
              acker(AckResult.Ack(envelope.deliveryTag))
        case Failure(e) =>
          logger.error(e)("Error while decoding") *>
            acker(AckResult.NAck(envelope.deliveryTag))
      }
    }
     .compile.drain
}
```

PROJECTOR APPLICATION

1. ~~read a bunch of configs from the env~~
2. ~~interact with a RabbitMQ broker~~
   - 2.1 ~~open a connection~~
   - 2.2 ~~receive a Stream of events from the given queue~~
3. ~~interact with a MongoDB cluster~~
   - 3.1 ~~open a connection~~
   - 3.2 ~~store the model to the given collection~~

WIRING

How to achieve separation of concerns and have a good modularity?

WIRING

Constructor Injection!

- JVM application lifecycle is not so complex
- *No need for magic*, each dependency can be explicitly injected
- Acquiring/releasing resources should be handled as an effect

INTRODUCING CONSTRUCTOR INJECTION

HOW *not to suffer* WHILE INJECTING DEPENDENCIES

CONSTRUCTOR INJECTION

- a class with a *private constructor*
- a companion object with a `fromX/make` method (*smart constructor*)

1. taking dependencies as input
2. usually returning `IO/Resource` of the component class

WIRING - CONSTRUCTOR INJECTION

```
class OrderHistoryProjector private (
  eventRepo: EventRepository,
  consumer: Consumer,
  acker: Acker,
  logger: Logger) {
  ...
}

object OrderHistoryProjector {
  def fromConfigs(mongoConfig: Mongo.Config,
                  rabbitConfig: Rabbit.Config
  ): Resource[OrderHistoryProjector] = ...
}
```

WIRING - CONSTRUCTOR INJECTION

```
object OrderHistoryProjector {
  def fromConfigs(
    mongoConfig: Mongo.Config,
    rabbitConfig: Fs2RabbitConfig
  ): Resource[OrderHistoryProjector] =
    for {
      (acker, consumer) <- Rabbit.consumerFrom(rabbitConfig, eventDecoder)
      collection        <- Mongo.collectionFrom(mongoConfig)
      repo               = EventRepository.fromCollection(collection)
      logger            <- Resource.eval(Slf4jLogger.create)
    } yield new OrderHistoryProjector(repo, consumer, acker, logger)
}
```

WIRING - MAIN

```
object OrderHistoryProjectorApp extends IOApp.Simple {

  def run: IO[Unit] =
    for {
      mongoConfig  <- Mongo.Config.load
      rabbitConfig <- Rabbit.Config.load

      _ <- OrderHistoryProjector
            .fromConfigs(mongoConfig, rabbitConfig) // acquire the needed resources
            .use(_.project) // start to process the stream of events

    } yield ()
}
```

ALL DONE!

THERE'S A LOT MORE TO TALK ABOUT

- How to handle concurrency, execution contexts, blocking ops?
  - How to track and handle errors?
  - Do we really need advanced techniques?

CONCLUSIONS

- a production-ready component in under 300 LOC
- only 3 main datatypes: `IO`, `Resource`, `Stream`
- no variables, no mutable state
- not even the M word!
- *I could have written almost the same code in Kotlin, Swift or.. Haskell!*

REFERENCES

https://github.com/AL333Z/fp-in-industry
https://typelevel.org/cats-effect/
https://fs2.io/
https://fs2-rabbit.profunktor.dev/

THANKS

I'VE BEEN LYING TO YOU

STREAM, RESOURCE, RABBITCLIENT AND MONGOCOLLECTION ARE POLYMORPHIC IN THE EFFECT TYPE!

In all the slides I always omitted the additional effect type parameter!

- `Resource[F, A]`
- `Stream[F, A]`
- `RabbitClient[F]`
- `MongoCollection[F]`

POLYMORPHISM IS GREAT, BUT COMES AT A (LEARNING) COST!