(IM)PRACTICAL FUNCTIONAL PROGRAMMING

ADOPTING FP IN INDUSTRY

WHO AM I

@AL333Z

SOFTWARE ENGINEER

MEMBER OF @FPINBO λ fp bo

RUNNER

WHY THIS TALK?

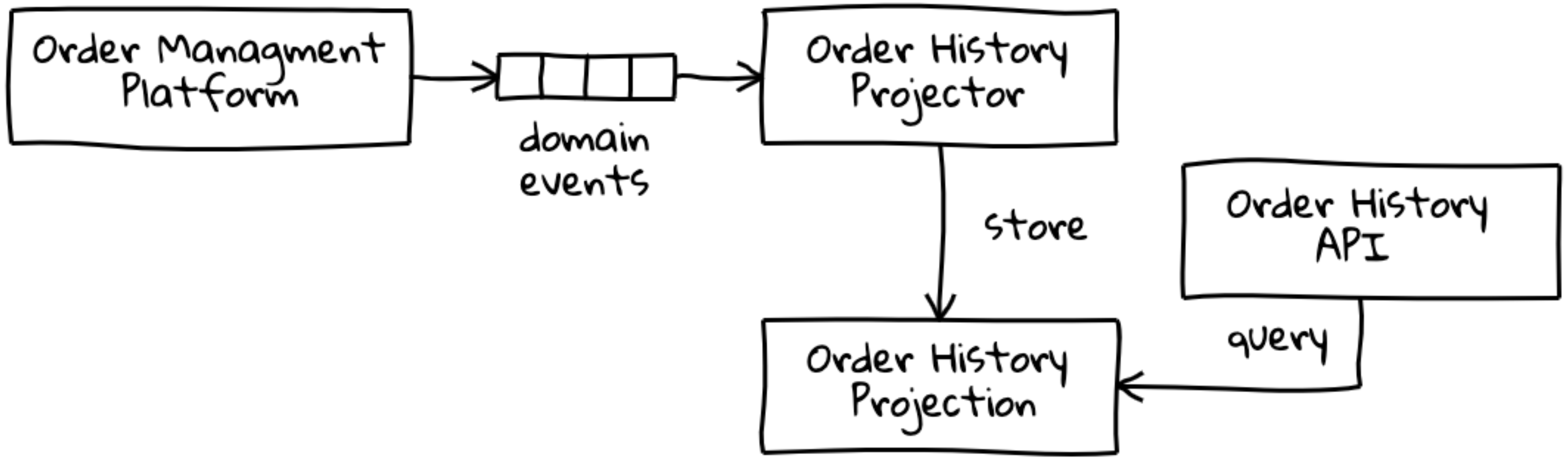
HOW MANY TIMES HAVE YOU HEARD:

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

AGENDA

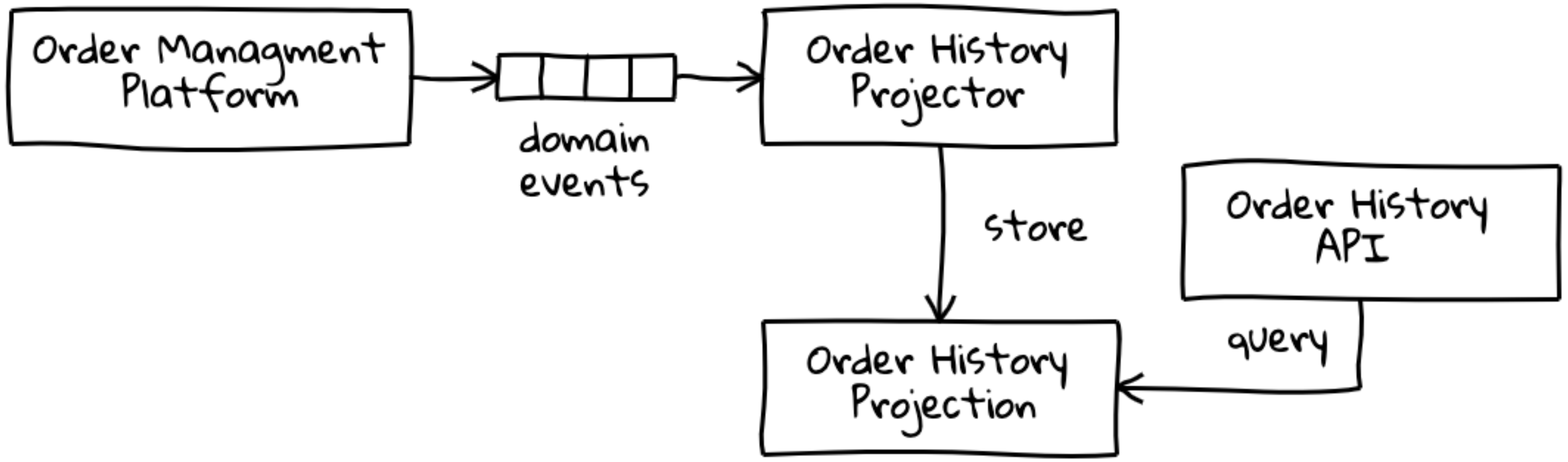
- A sample architecture
- Introduce a bunch of building blocks
- Design architecture components

SAMPLE ARCHITECTURE: ORDER HISTORY SERVICE



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

ORDER HISTORY SERVICE: COMPONENTS



- a component which projects a model, in a MongoDB collection
- so that an HTTP service can queries 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) using Pure Functional Programming, in Scala

WHY SCALA

WHY SCALA

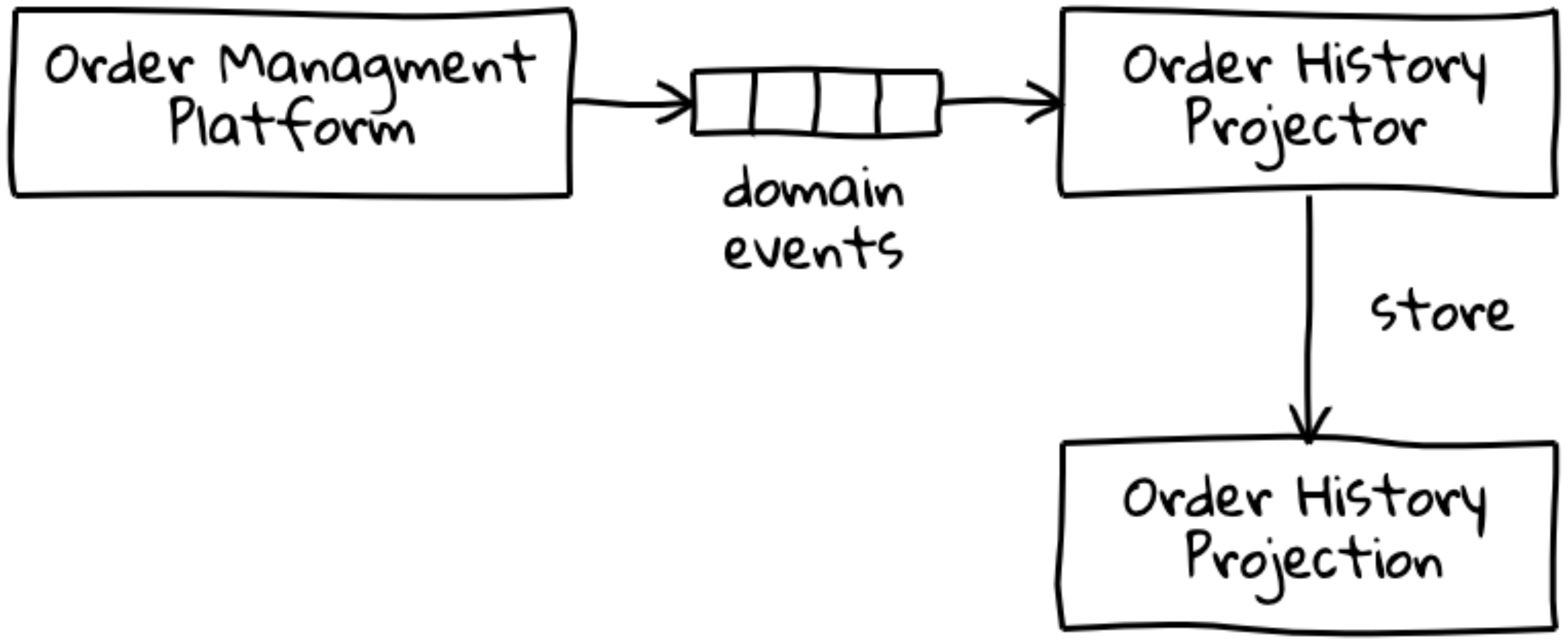
I KNOW SCALA

WHY SCALA

- immutability, ADTs
- higher-kinded types + implicits -> typeclasses
- DSL-friendly
- *mature ecosystem* of FP libs (cats, cats-effects, fs2, circe, http4s, etc..)

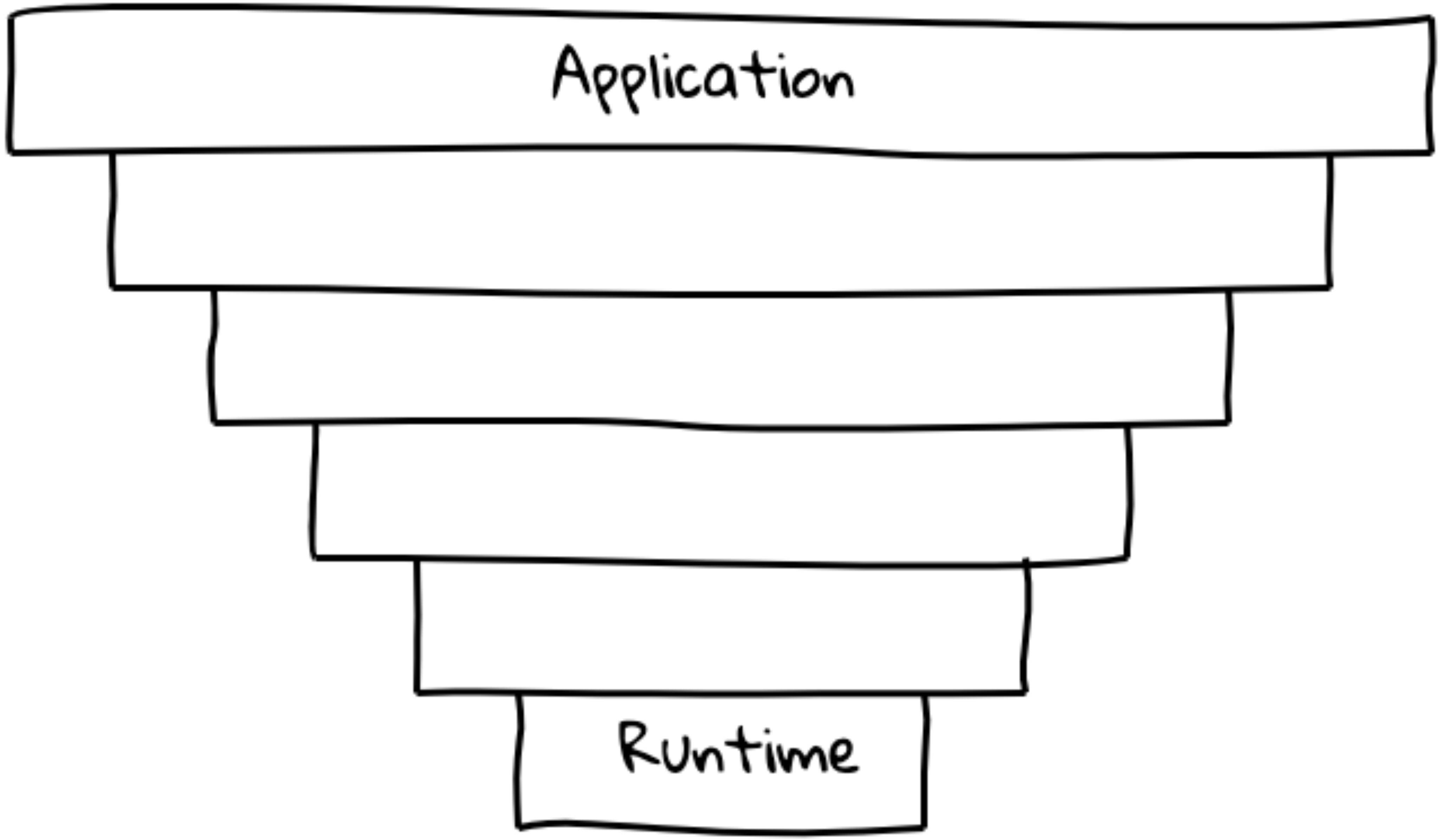
LET'S START

BUILDING A PROJECTOR



- *Consume* a stream of events from a RabbitMQ queue
- *Persist* a 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

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

INTRODUCING IO

A DATA TYPE FOR *encoding effects* AS PURE VALUES

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**

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*

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]
  ...
}

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]
  ...
}
```

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]
  ...
}

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]
  ...
}
```

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]
  ...
}

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]
  ...
}
```

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]
  ...
}

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

```
val ioInt: IO[Int] = IO.delay{ println("hello"); 1 }

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

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

COMPOSING SEQUENTIAL EFFECTS

```
val ioInt: IO[Int] = IO.delay{ println("hello"); 1 }

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

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

COMPOSING SEQUENTIAL EFFECTS

```
val ioInt: IO[Int] = IO.delay{ println("hello"); 1 }

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

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

COMPOSING SEQUENTIAL EFFECTS

```
val ioInt: IO[Int] = IO.delay{ println("hello"); 1 }

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

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

COMPOSING SEQUENTIAL EFFECTS

```
val ioInt: IO[Int] = IO.delay{ println("hello"); 1 }

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

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

COMPOSING SEQUENTIAL EFFECTS

```
val ioInt: IO[Int] = IO.delay{ println("hello"); 1 }

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

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

WE ARE PRACTICAL

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], /*...*/)

  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)
  }
}
```

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], /*...*/)

  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)
  }
}
```

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], /*...*/)

  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)
  }
}
```

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], /*...*/)

  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

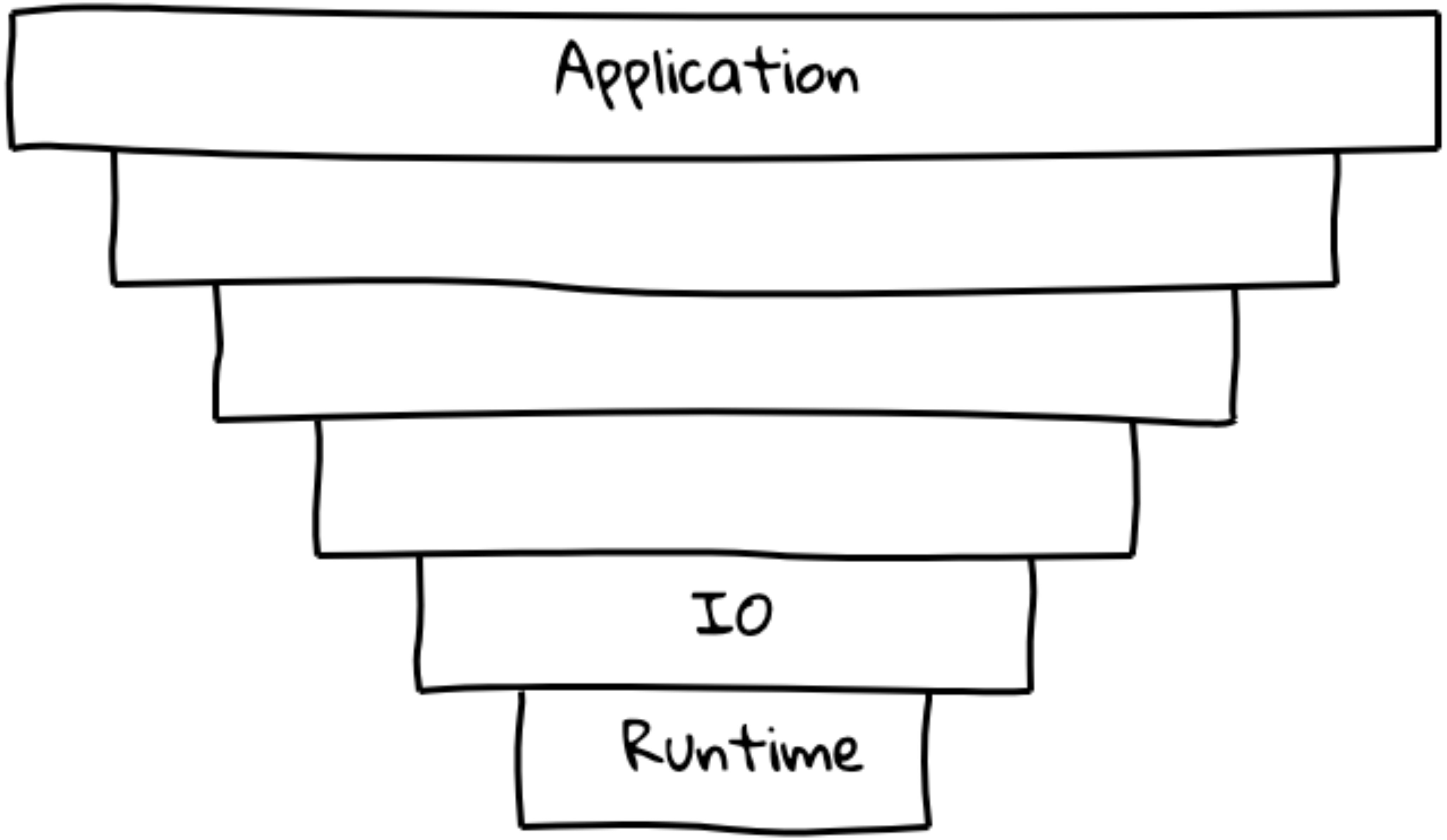
```
val ioOps =
 for {
    mongoConfig  <- Mongo.Config.load
    rabbitConfig <- Rabbit.Config.load
    // TODO use configs to do something!
 } yield ()
```

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 {
  override def run(args: List[String]): IO[ExitCode] =
    for {
      mongoConfig  <- Mongo.Config.load
      rabbitConfig <- Rabbit.Config.load
      // TODO use configs to start the main logic!
    } yield ExitCode.Success
}
```

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 {
  override def run(args: List[String]): IO[ExitCode] =
    for {
      mongoConfig  <- Mongo.Config.load
      rabbitConfig <- Rabbit.Config.load
      // TODO use configs to start the main logic!
    } yield ExitCode.Success
}
```

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

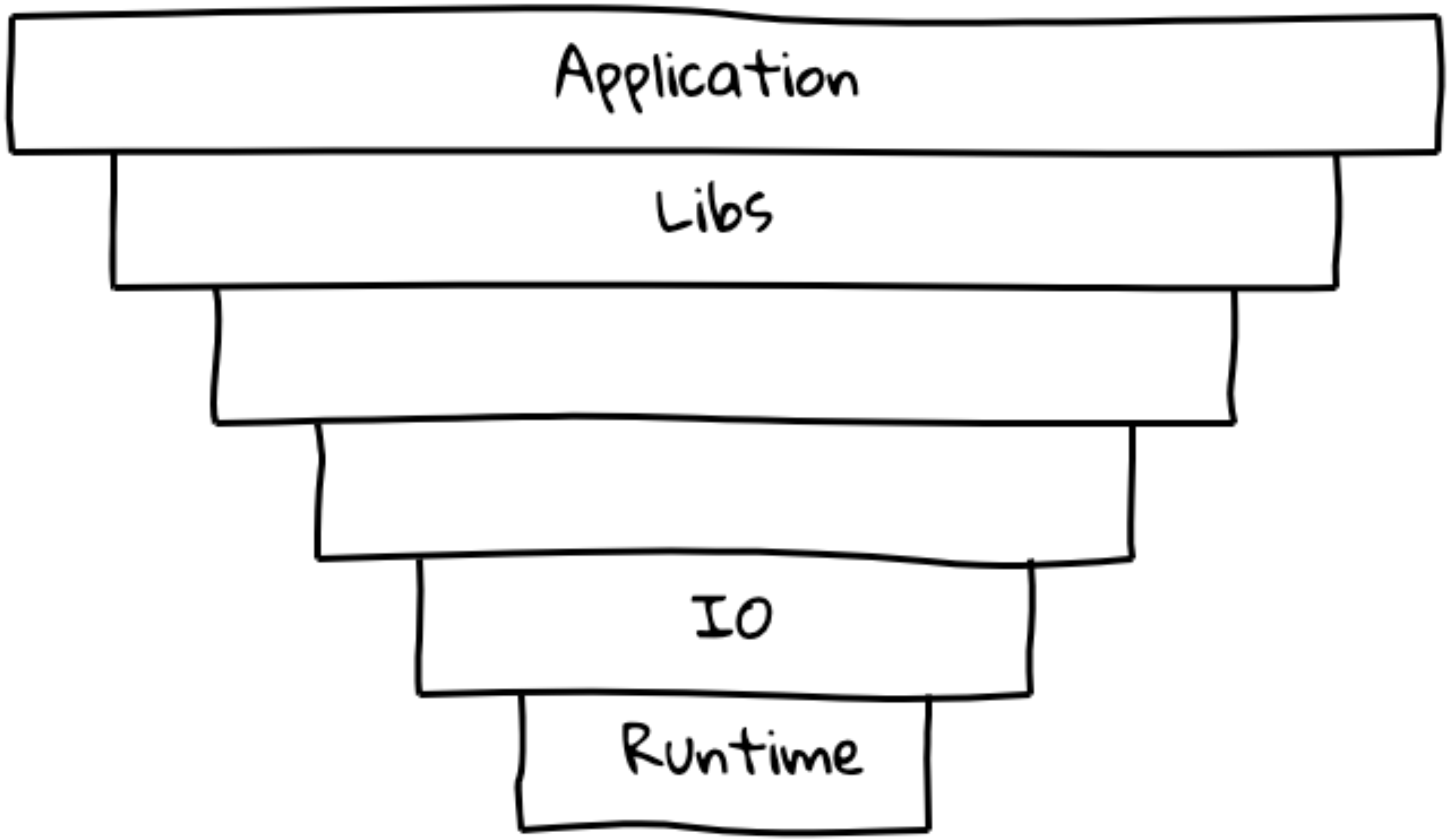
   3.2 store the model to the given collection

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 client: Fs2Rabbit = Fs2Rabbit(config)

val channel: Resource[AMQPChannel] = client.createConnectionChannel
```

Resource?

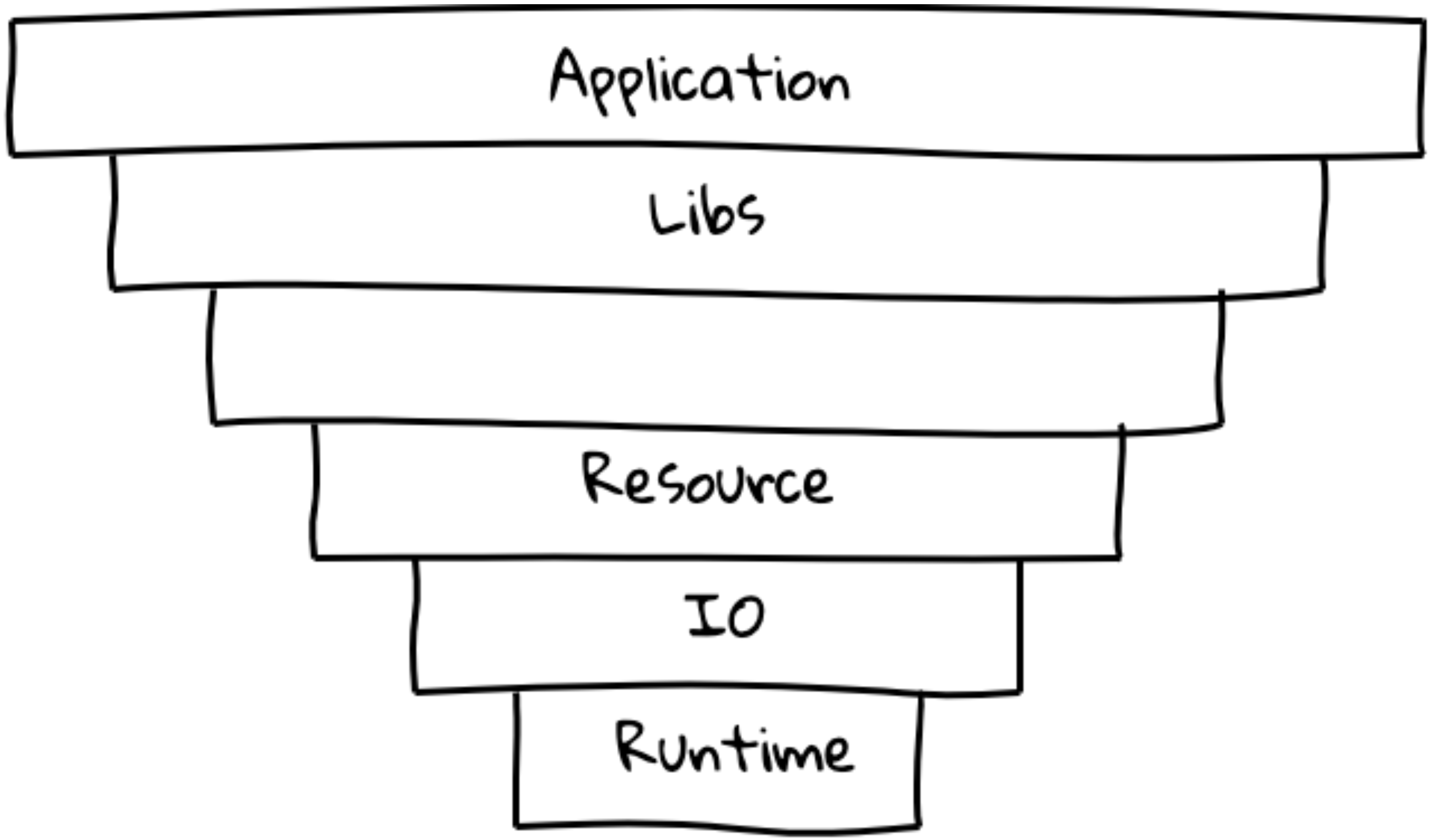
INTRODUCING RESOURCE

EFFECTFULLY ALLOCATES AND RELEASES A RESOURCE

EXTREMELY HELPFUL TO WRITE CODE THAT:

- doesn't leak
- handles properly terminal signals

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

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

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)
}
```

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)
}
```

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)
}
```

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)
}
```

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 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 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 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 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
```

GOTCHAS:

- Nested resources are released in reverse order of acquisition
    - Easy to lift an `AutoClosable` to `Resource`, via `Resource.fromAutoclosable`
- You can lift any `IO[A]` into a `Resource[A]` with a no-op release via `Resource.liftF`

WE ARE PRAGMATIC

2.1. INTERACT WITH A RABBITMQ BROKER

```
val client: Fs2Rabbit = Fs2Rabbit(config)

val rabbitDeps: Resource[(Acker, Consumer)] = for {
  channel <- client.createConnectionChannel // resource opening a connection to a channel
  (acker, consumer) <- Resource.liftF( // lift an IO which creates the consumer
    client.createAckerConsumer[Try[OrderCreatedEvent]](
      queueName = QueueName("EventsFromOms"),
      basicQos = BasicQos(0, 10))(
      channel = channel,
      decoder = decoder
    )
  )
} yield (acker, consumer)

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

2.1. INTERACT WITH A RABBITMQ BROKER

```
val client: Fs2Rabbit = Fs2Rabbit(config)

val rabbitDeps: Resource[(Acker, Consumer)] = for {
  channel <- client.createConnectionChannel // resource opening a connection to a channel
  (acker, consumer) <- Resource.liftF( // lift an IO which creates the consumer
    client.createAckerConsumer[Try[OrderCreatedEvent]](
      queueName = QueueName("EventsFromOms"),
      basicQos = BasicQos(0, 10))(
      channel = channel,
      decoder = decoder
    )
  )
} yield (acker, consumer)

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

2.1. INTERACT WITH A RABBITMQ BROKER

```scala
val client: Fs2Rabbit = Fs2Rabbit(config)

val rabbitDeps: Resource[(Acker, Consumer)] = for {
  channel <- client.createConnectionChannel // resource opening a connection to a channel
  (acker, consumer) <- Resource.liftF( // lift an IO which creates the consumer
    client.createAckerConsumer[Try[OrderCreatedEvent]](
      queueName = QueueName("EventsFromOms"),
      basicQos = BasicQos(0, 10))(
      channel = channel,
      decoder = decoder
    )
  )
} yield (acker, consumer)

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

2.1. INTERACT WITH A RABBITMQ BROKER

```scala
val client: Fs2Rabbit = Fs2Rabbit(config)

val rabbitDeps: Resource[(Acker, Consumer)] = for {
  channel <- client.createConnectionChannel // resource opening a connection to a channel
  (acker, consumer) <- Resource.liftF( // lift an IO which creates the consumer
    client.createAckerConsumer[Try[OrderCreatedEvent]](
      queueName = QueueName("EventsFromOms"),
      basicQos = BasicQos(0, 10))(
      channel = channel,
      decoder = decoder
    )
  )
} yield (acker, consumer)

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

2.1. INTERACT WITH A RABBITMQ BROKER

```
val client: Fs2Rabbit = Fs2Rabbit(config)

val rabbitDeps: Resource[(Acker, Consumer)] = for {
  channel <- client.createConnectionChannel // resource opening a connection to a channel
  (acker, consumer) <- Resource.liftF( // lift an IO which creates the consumer
    client.createAckerConsumer[Try[OrderCreatedEvent]](
      queueName = QueueName("EventsFromOms"),
      basicQos = BasicQos(0, 10))(
      channel = channel,
      decoder = decoder
    )
  )
} yield (acker, consumer)

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

I HEAR YOU...

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

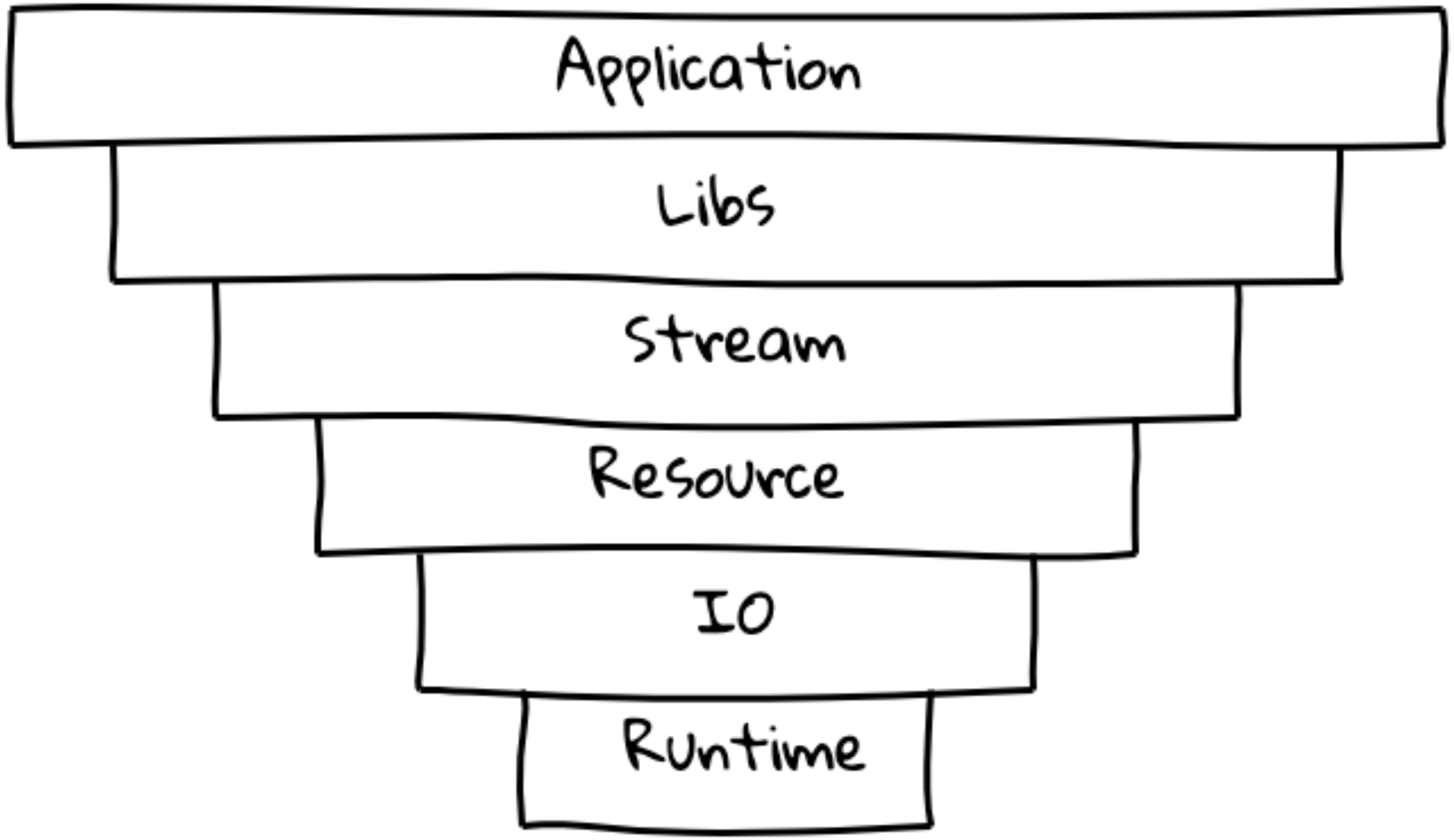
INTRODUCING STREAM

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 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 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
}
```

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
}
```

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
}
```

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
}
```

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 the official `mongo-scala-driver`, which is **not** exposing purely functional apis..

HOW TO TURN AN API TO BE FUNCTIONAL^{TM}?

HOW TO TURN AN API TO BE FUNCTIONAL™?

In most cases:

- *wrap* the **impure type** so that its operations are no more reachable
- only *expose* a **safer** version of its operations

"WRAP THE CRAP"

```
class Collection(
  private val wrapped: MongoCollection[Document]) {

  def insertOne(document: Document): IO[Unit] =
    wrapped
      .insertOne(document)
      .toIO // <- extension method converting to IO!
      .void
}
```

"WRAP THE CRAP"

```
class Collection(
  private val wrapped: MongoCollection[Document]) {

  def insertOne(document: Document): IO[Unit] =
    wrapped
      .insertOne(document)
      .toIO // <- extension method converting to IO!
      .void
}
```

"WRAP THE CRAP"

```
class Collection(
  private val wrapped: MongoCollection[Document]) {

  def insertOne(document: Document): IO[Unit] =
    wrapped
      .insertOne(document)
      .toIO // <- extension method converting to IO!
      .void
}
```

3.1 OPEN A CONNECTION

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

    for {
      client    <- Resource.fromAutoCloseable(IO.defer(MongoClient(clientSettings)))
      unsafeCol =  client.getDatabase(conf.databaseName)
                         .getCollection(conf.collectionName)
    } yield new Collection(unsafeCol)
  }
}
```

3.1 OPEN A CONNECTION

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

    for {
      client    <- Resource.fromAutoCloseable(IO.defer(MongoClient(clientSettings)))
      unsafeCol =  client.getDatabase(conf.databaseName)
                         .getCollection(conf.collectionName)
    } yield new Collection(unsafeCol)
  }
}
```

3.1 OPEN A CONNECTION

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

    for {
      client    <- Resource.fromAutoCloseable(IO.defer(MongoClient(clientSettings)))
      unsafeCol =  client.getDatabase(conf.databaseName)
                         .getCollection(conf.collectionName)
    } yield new Collection(unsafeCol)
  }
}
```

3.1 OPEN A CONNECTION

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

    for {
      client    <- Resource.fromAutoCloseable(IO.defer(MongoClient(clientSettings)))
      unsafeCol =  client.getDatabase(conf.databaseName)
                         .getCollection(conf.collectionName)
    } yield new Collection(unsafeCol)
  }
}
```

3.1 OPEN A CONNECTION

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

    for {
      client    <- Resource.fromAutoCloseable(IO.defer(MongoClient(clientSettings)))
      unsafeCol =  client.getDatabase(conf.databaseName)
                         .getCollection(conf.collectionName)
    } yield new Collection(unsafeCol)
  }
}
```

3.1 OPEN A CONNECTION

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

    for {
      client    <- Resource.fromAutoCloseable(IO.defer(MongoClient(clientSettings)))
      unsafeCol =  client.getDatabase(conf.databaseName)
                         .getCollection(conf.collectionName)
    } yield new Collection(unsafeCol)
  }
}
```

3.1 OPEN A CONNECTION

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

    for {
      client    <- Resource.fromAutoCloseable(IO.defer(MongoClient(clientSettings)))
      unsafeCol =  client.getDatabase(conf.databaseName)
                         .getCollection(conf.collectionName)
    } yield new Collection(unsafeCol)
  }
}
```

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: Collection) {
  def store(event: OrderCreatedEvent): IO[Unit] =
    collection.insertOne( // using safe ops
      Document(
        "id"      -> event.id,
        "company" -> event.company,
        "email"   -> event.email,
        "lines" -> event.lines.map(line => ...)
      )
    )
}
```

3.2 STORE THE MODEL TO THE GIVEN COLLECTION

```
class EventRepository(collection: Collection) {
  def store(event: OrderCreatedEvent): IO[Unit] =
    collection.insertOne( // using safe ops
      Document(
        "id"      -> event.id,
        "company" -> event.company,
        "email"   -> event.email,
        "lines" -> event.lines.map(line => ...)
      )
    )
}
```

3.2 STORE THE MODEL TO THE GIVEN COLLECTION

```
class EventRepository(collection: Collection) {
  def store(event: OrderCreatedEvent): IO[Unit] =
    collection.insertOne( // using safe ops
      Document(
        "id"      -> event.id,
        "company" -> event.company,
        "email"   -> event.email,
        "lines" -> event.lines.map(line => ...)
      )
    )
}
```

3.2 STORE THE MODEL TO THE GIVEN COLLECTION

```
class EventRepository(collection: Collection) {
  def store(event: OrderCreatedEvent): IO[Unit] =
    collection.insertOne( // using safe ops
      Document(
        "id"      -> event.id,
        "company" -> event.company,
        "email"   -> event.email,
        "lines" -> event.lines.map(line => ...)
      )
    )
}
```

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
}
```

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
}
```

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?

WIRING

Constructor Injection!

- JVM application lifecycle is not so complex
- `IO`, `IOApp`, `Resource`, `Stream` are handling properly termination events

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 deps as input
2. usually returning `IO/Resource` of the component class

My view of Constructor Injection for effectful applications

WIRING - CONSTRUCTOR INJECTION

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

object OrderHistoryProjector {
  def fromConfigs(mongoConfig: Mongo.Config,
                  rabbitConfig: Fs2RabbitConfig
  ): Resource[OrderHistoryProjector] = ...
}
```

WIRING - CONSTRUCTOR INJECTION

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

object OrderHistoryProjector {
  def fromConfigs(mongoConfig: Mongo.Config,
                  rabbitConfig: Fs2RabbitConfig
  ): Resource[OrderHistoryProjector] = ...
}
```

WIRING - CONSTRUCTOR INJECTION

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

object OrderHistoryProjector {
  def fromConfigs(mongoConfig: Mongo.Config,
                  rabbitConfig: Fs2RabbitConfig
  ): Resource[OrderHistoryProjector] = ...
}
```

WIRING - CONSTRUCTOR INJECTION

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

object OrderHistoryProjector {
  def fromConfigs(mongoConfig: Mongo.Config,
                  rabbitConfig: Fs2RabbitConfig
  ): Resource[OrderHistoryProjector] = ...
}
```

WIRING - CONSTRUCTOR INJECTION

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

CONSTRUCTOR INJECTION

- *No magic*, each dependency is explicitly injected
- Acquiring/releasing resources is handled as an **effect**

MAIN

```scala
object OrderHistoryProjectorApp extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    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 ExitCode.Success
}
```

MAIN

```
object OrderHistoryProjectorApp extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    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 ExitCode.Success
}
```

ALL DONE!

CONCLUSIONS

- a production-ready component in under 300 LOC
- only 3 main datatypes: `IO`, `Resource`, `Stream`
- no variables, no mutable state
- no ivory tower
- *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 AND FS2RABBIT ARE POLYMORPHIC IN THE EFFECT TYPE!

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

- `Resource[F, A]`
- `Stream[F, A]`
- `Fs2Rabbit[F]`

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