How to turn an API into Functional Programming

Lessons learned while filling the gap

Who am I?

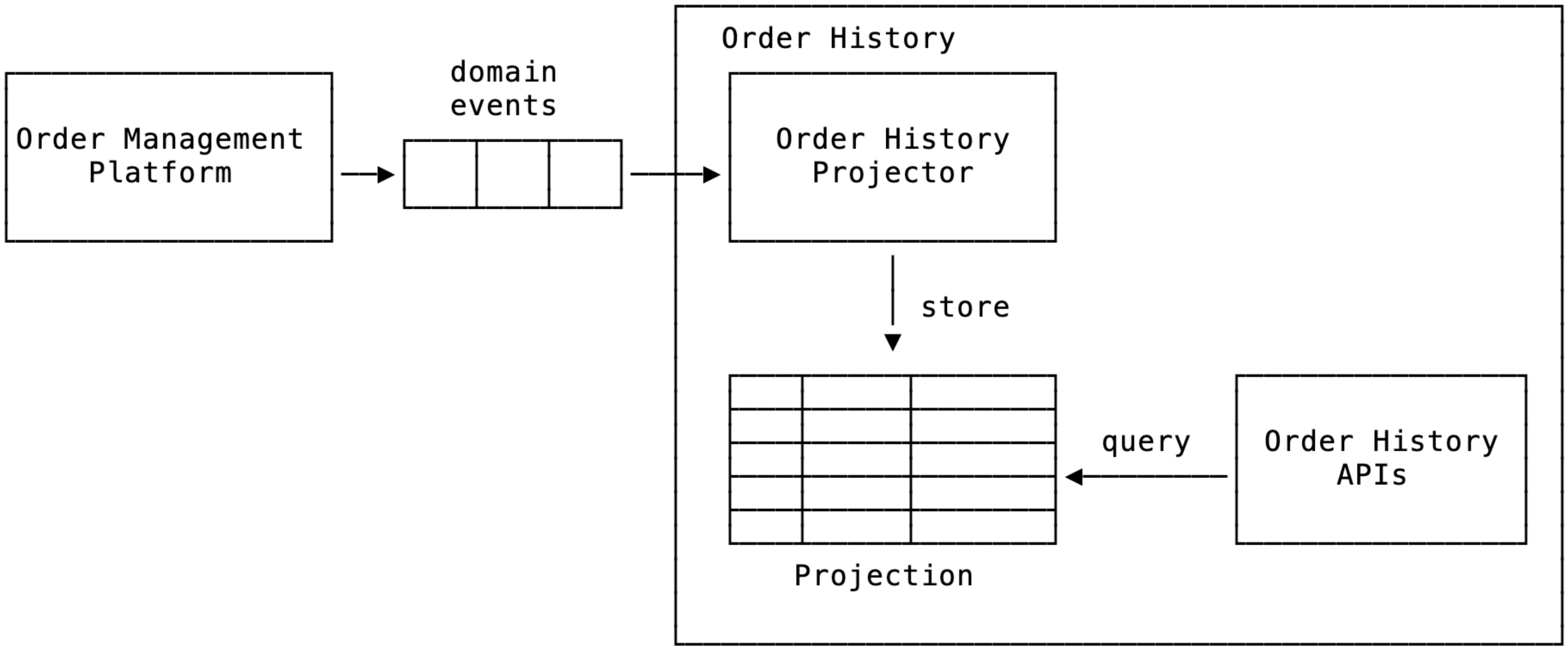
@al333z

- Senior Software Engineer @ Moneyfarm
- Several years in the scala/typelevel ecosystem
- Member of *@FPinBO*

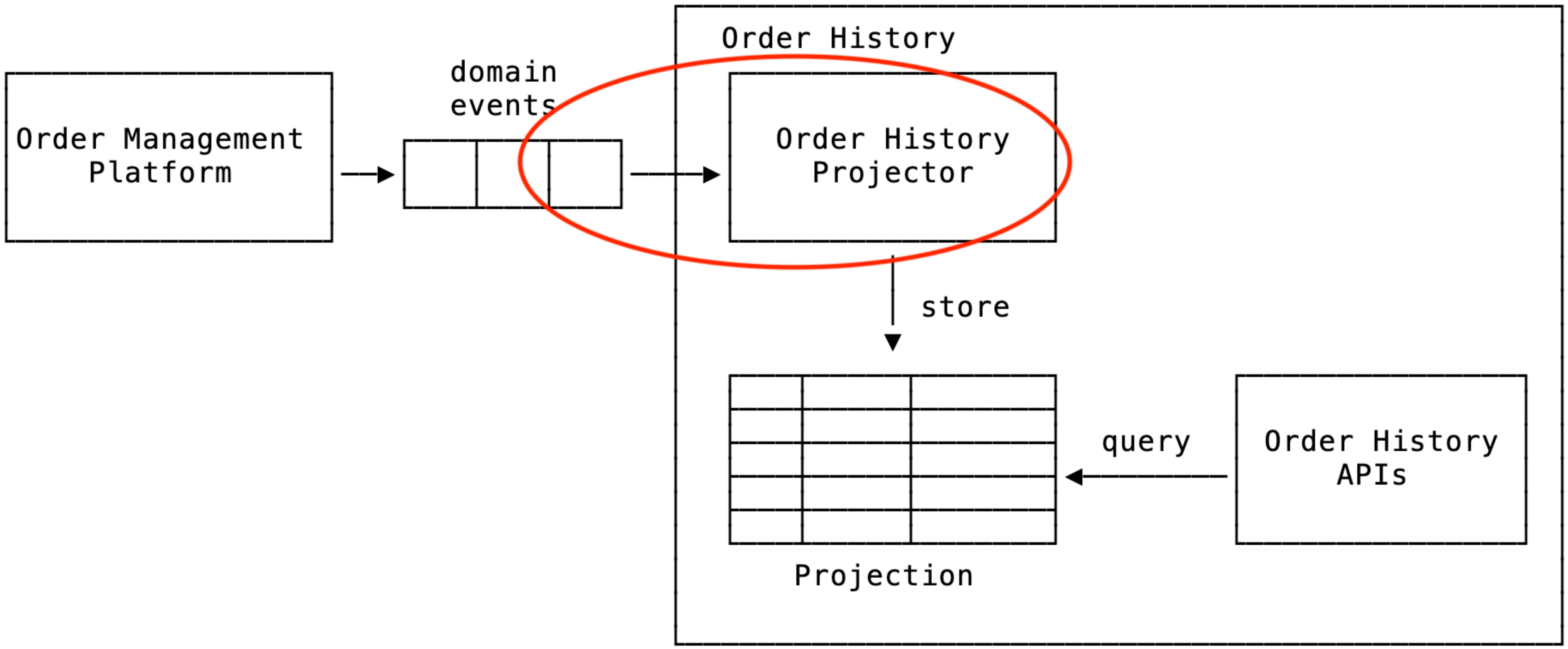
Agenda

- a sample use case, a reference **library to wrap**
- designing library apis
  - introduce a bunch of **building blocks**
  - **refine** edges, **evaluate** alternatives, **iterate**

A sample architecture



A sample architecture



Disclaimer

Our focus here is ***NOT*** on building the coolest library doing the coolest thing ever.

We'll just put our attention on ***designing a set of APIs*** which wraps an existing lib written in the *good old imperative way*, using Pure Functional Programming and the Typelevel stack.

A reference library

Java Message Service a.k.a. JMS

- can be used to facilitate the sending and receiving of messages between enterprise software systems, whatever it means enterprise!
- a bunch of Java interfaces
- each provider offers an implementation (e.g. IBM MQ, ActiveMQ, RabbitMQ, etc…)

Why JMS?

- ~~old~~ stable enough (born in 1998, latest revision in 2015)
- its apis are a **good testbed for sketching a purely functional wrapper**
- found pretty much nothing about (no FP-like bindings…)

Let's start

with a bottom-up approach

A look at the beast: receiving

```
public void receiveMessage(ConnectionFactory connectionFactory, String queueName){
   try (
     JMSContext context = connectionFactory.createContext(Session.SESSION_TRANSACTED);
   ){
      Queue queue = context.createQueue(queueName);
      JMSConsumer consumer = context.createConsumer(queue);
      Message msg = consumer.receive();
      // ... do something useful ...
      context.commit();
   } catch (JMSRuntimeException ex) {
      // ...
   }
}
```

What can we do to improve them these APIs?

- evaluate what is the **design which better supports our intent**
  - **prevent** the developer using our lib from doing **wrong things** (e.g. unconfirmed messages, deadlocks, etc…) by design
  - offering a **high-level** set of APIs

Receiving

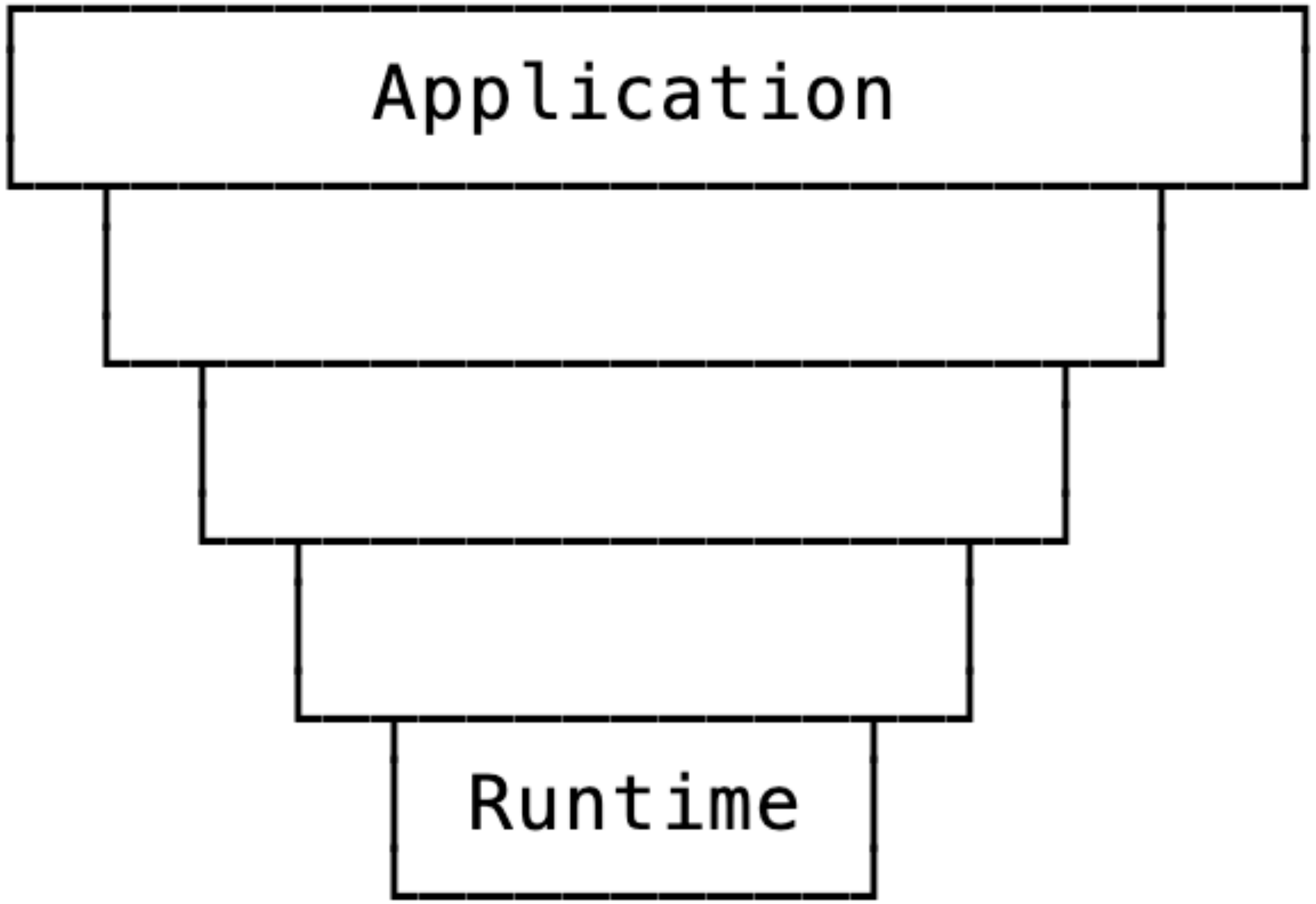
```
public void receiveMessage(ConnectionFactory connectionFactory, String queueName){
   try (
     JMSContext context = connectionFactory.createContext(Session.SESSION_TRANSACTED);
   ){
      Queue queue = context.createQueue(queueName);
      JMSConsumer consumer = context.createConsumer(queue);
      Message msg = consumer.receive();
      // ... do something useful ...
      context.commit();
   } catch (JMSRuntimeException ex) {
      // ...
   }
}
```

Let's start from the low level stuff...

- how to handle **side-effects**?
- how to handle the **resource lifecycle**?

Let's see how FP con help us in doing the right thing™!

The abstraction gap



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

Building effects

```scala
object IO {
  def delay[A](a: => A): IO[A]
  def raiseError[A](e: Throwable): IO[A]
  def async[A](k: /* ... */): IO[A]
  def blocking[A](/* ... */): IO[A]
  ...
}

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

Composing 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 ()
```

Running effects

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

Introducing IO

Building effects

```scala
object IO {
  def delay[A](a: => A): IO[A]
  def raiseError[A](e: Throwable): IO[A]
  def async[A](k: /* ... */): IO[A]
  def blocking[A](/* ... */): IO[A]
  ...
}

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

Composing 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 ()
```

Running effects

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

Introducing IO

Building effects

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

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

Composing effects

```
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 ()
```

Running effects

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

Introducing IO

Building effects

```scala
object IO {
  def delay[A](a: => A): IO[A]
  def raiseError[A](e: Throwable): IO[A]
  def async[A](k: /* ... */): IO[A]
  def blocking[A](/* ... */): IO[A]
  ...
}

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

Composing 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 ()
```

Running effects

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

Introducing IO

Building effects

```scala
object IO {
  def delay[A](a: => A): IO[A]
  def raiseError[A](e: Throwable): IO[A]
  def async[A](k: /* ... */): IO[A]
  def blocking[A](/* ... */): IO[A]
  ...
}

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

Composing 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 ()
```

Running effects

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

Introducing IO

Building effects

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

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

Composing effects

```
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 ()
```

Running effects

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

Introducing IO

Building effects

```scala
object IO {
  def delay[A](a: => A): IO[A]
  def raiseError[A](e: Throwable): IO[A]
  def async[A](k: /* ... */): IO[A]
  def blocking[A](/* ... */): IO[A]
  ...
}

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

Composing 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 ()
```

Running effects

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

Introducing IO

Building effects

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

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

Composing effects

```
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 ()
```

Running effects

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

Introducing IO

Building effects

```scala
object IO {
  def delay[A](a: => A): IO[A]
  def raiseError[A](e: Throwable): IO[A]
  def async[A](k: /* ... */): IO[A]
  def blocking[A](/* ... */): IO[A]
  ...
}

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

Composing 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 ()
```

Running effects

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

Introducing IO

Building effects

```scala
object IO {
  def delay[A](a: => A): IO[A]
  def raiseError[A](e: Throwable): IO[A]
  def async[A](k: /* ... */): IO[A]
  def blocking[A](/* ... */): IO[A]
  ...
}

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

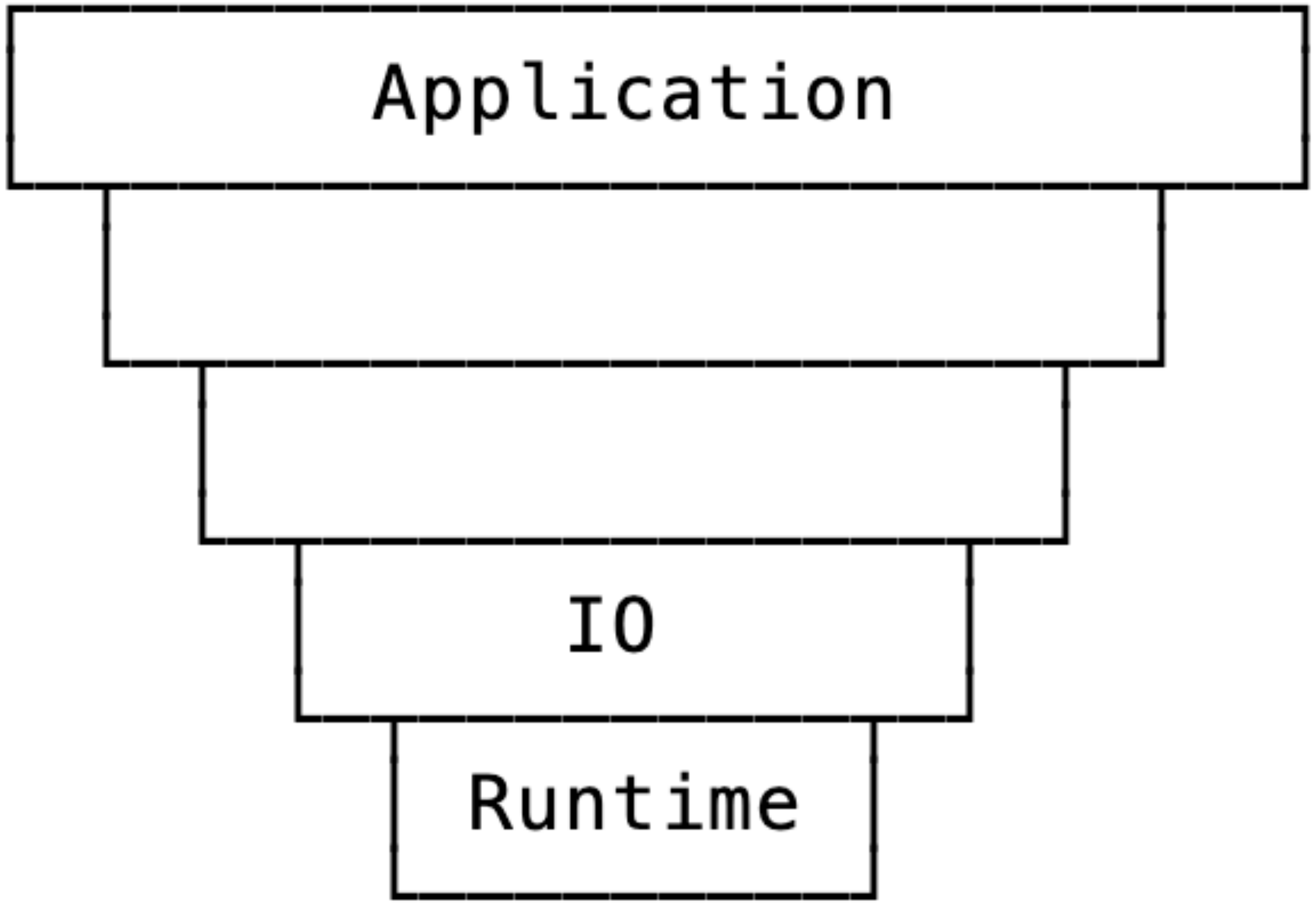
Composing effects

```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 ()
```

Running effects

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

How to fill the abstraction gap?



JmsContext - 1st iteration

```
sealed abstract class JmsContext(private[lib] val raw: javax.jms.JMSContext) {

  def createQueue(queue: QueueName): IO[JmsQueue] =
    IO.delay(new JmsQueue(raw.createQueue(queue.value)))

  def makeJmsConsumer(queueName: QueueName): IO[JmsMessageConsumer] =
    for {
      destination <- createQueue(queueName)
      consumer    <- IO.delay(raw.createConsumer(destination.wrapped))
    } yield new JmsMessageConsumer(consumer)
}

class JmsTransactedContext private[lib] (
  override private[lib] val raw: javax.jms.JMSContext) extends JmsContext(raw)
```

- handle JMSRuntimeException ✅
- handle the resource lifecycle ❌

JmsContext - 1st iteration

```
sealed abstract class JmsContext(private[lib] val raw: javax.jms.JMSContext) {

  def createQueue(queue: QueueName): IO[JmsQueue] =
    IO.delay(new JmsQueue(raw.createQueue(queue.value)))

  def makeJmsConsumer(queueName: QueueName): IO[JmsMessageConsumer] =
    for {
      destination <- createQueue(queueName)
      consumer    <- IO.delay(raw.createConsumer(destination.wrapped))
    } yield new JmsMessageConsumer(consumer)
}

class JmsTransactedContext private[lib] (
  override private[lib] val raw: javax.jms.JMSContext) extends JmsContext(raw)
```

- handle JMSRuntimeException ✅
- handle the resource lifecycle ❌

JmsContext - 1st iteration

```scala
sealed abstract class JmsContext(private[lib] val raw: javax.jms.JMSContext) {

  def createQueue(queue: QueueName): IO[JmsQueue] =
    IO.delay(new JmsQueue(raw.createQueue(queue.value)))

  def makeJmsConsumer(queueName: QueueName): IO[JmsMessageConsumer] =
    for {
      destination <- createQueue(queueName)
      consumer    <- IO.delay(raw.createConsumer(destination.wrapped))
    } yield new JmsMessageConsumer(consumer)
}

class JmsTransactedContext private[lib] (
  override private[lib] val raw: javax.jms.JMSContext) extends JmsContext(raw)
```

- handle JMSRuntimeException ✅
- handle the resource lifecycle ❌

JmsContext - 1st iteration

```
sealed abstract class JmsContext(private[lib] val raw: javax.jms.JMSContext) {

  def createQueue(queue: QueueName): IO[JmsQueue] =
    IO.delay(new JmsQueue(raw.createQueue(queue.value)))

  def makeJmsConsumer(queueName: QueueName): IO[JmsMessageConsumer] =
    for {
      destination <- createQueue(queueName)
      consumer    <- IO.delay(raw.createConsumer(destination.wrapped))
    } yield new JmsMessageConsumer(consumer)
}

class JmsTransactedContext private[lib] (
  override private[lib] val raw: javax.jms.JMSContext) extends JmsContext(raw)
```

- handle JMSRuntimeException ✅
- handle the resource lifecycle ❌

How to handle the lifecycle of a 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

Introducing Resource

Building resources

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

  def fromAutoCloseable[A <: AutoCloseable](
    acquire: IO[A]): Resource[A]
}

class Resource[A] {
  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]

  def use[B](f: A => IO[B]): IO[B]
  ...
}
```

Composing resources

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

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

Using resources

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

NB: not actual code, Resource is polymorphic in the effect type

Introducing Resource

Building resources

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

  def fromAutoCloseable[A <: AutoCloseable](
    acquire: IO[A]): Resource[A]
}

class Resource[A] {
  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]

  def use[B](f: A => IO[B]): IO[B]
  ...
}
```

Composing resources

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

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

Using resources

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

NB: not actual code, Resource is polymorphic in the effect type

Introducing Resource

Building resources

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

  def fromAutoCloseable[A <: AutoCloseable](
    acquire: IO[A]): Resource[A]
}

class Resource[A] {
  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]

  def use[B](f: A => IO[B]): IO[B]
  ...
}
```

Composing resources

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

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

Using resources

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

NB: not actual code, Resource is polymorphic in the effect type

Introducing Resource

Building resources

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

  def fromAutoCloseable[A <: AutoCloseable](
    acquire: IO[A]): Resource[A]
}

class Resource[A] {
  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]

  def use[B](f: A => IO[B]): IO[B]
  ...
}
```

Composing resources

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

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

Using resources

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

NB: not actual code, Resource is polymorphic in the effect type

Introducing Resource

Building resources

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

  def fromAutoCloseable[A <: AutoCloseable](
    acquire: IO[A]): Resource[A]
}

class Resource[A] {
  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]

  def use[B](f: A => IO[B]): IO[B]
  ...
}
```

Composing resources

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

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

Using resources

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

NB: not actual code, Resource is polymorphic in the effect type

Introducing Resource

Building resources

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

  def fromAutoCloseable[A <: AutoCloseable](
    acquire: IO[A]): Resource[A]
}

class Resource[A] {
  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]

  def use[B](f: A => IO[B]): IO[B]
  ...
}
```

Composing resources

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

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

Using resources

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

NB: not actual code, Resource is polymorphic in the effect type

Introducing Resource

Building resources

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

  def fromAutoCloseable[A <: AutoCloseable](
    acquire: IO[A]): Resource[A]
}

class Resource[A] {
  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]

  def use[B](f: A => IO[B]): IO[B]
  ...
}
```

Composing resources

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

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

Using resources

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

NB: not actual code, Resource is polymorphic in the effect type

Introducing Resource

Building resources

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

  def fromAutoCloseable[A <: AutoCloseable](
    acquire: IO[A]): Resource[A]
}

class Resource[A] {
  def map[B](f: A => B): Resource[B]
  def flatMap[B](f: A => Resource[B]): Resource[B]

  def use[B](f: A => IO[B]): IO[B]
  ...
}
```

Composing resources

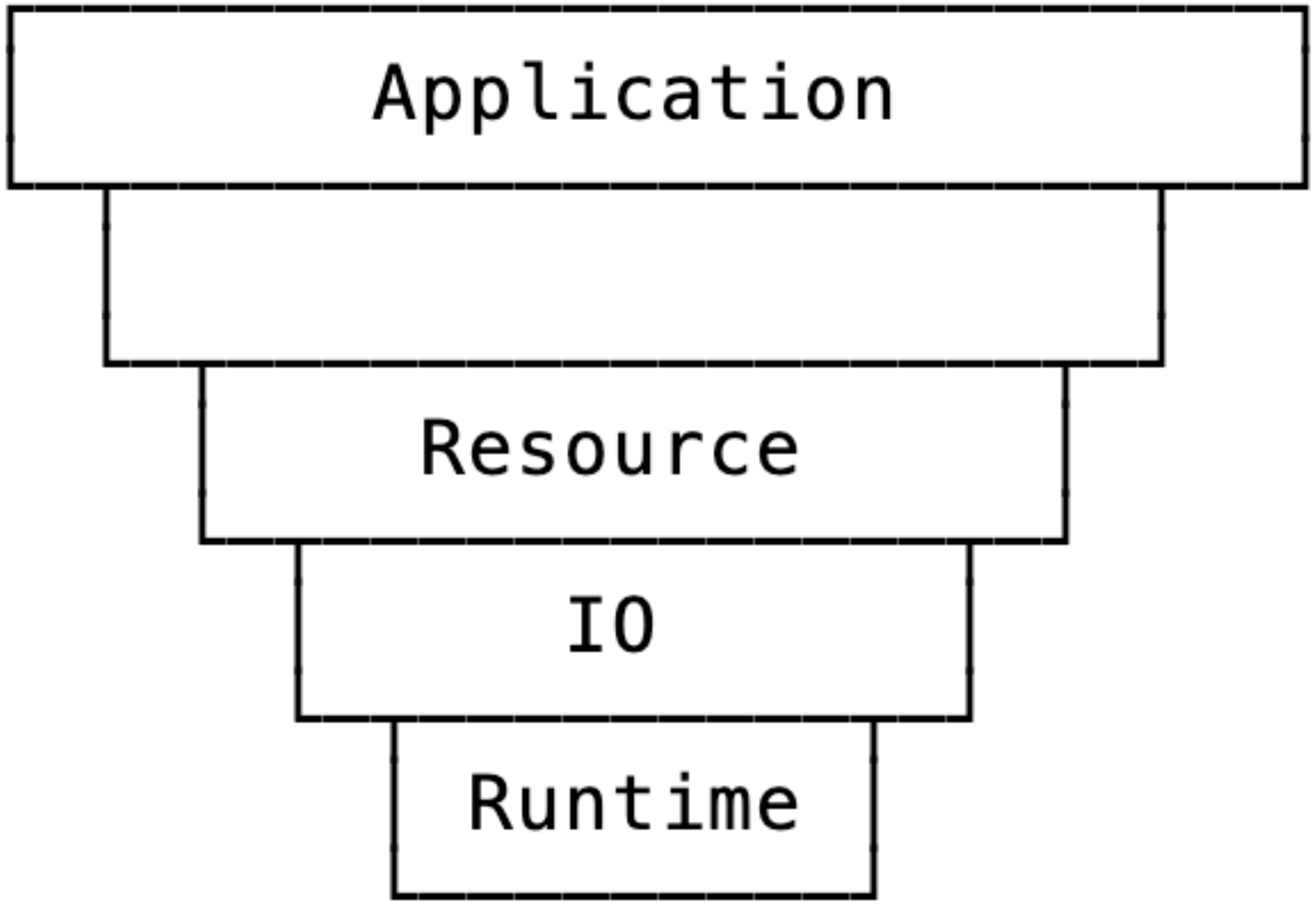
```scala
val sessionPool: Resource[MySessionPool] =
  for {
    connection <- openConnection()
    sessions   <- openSessionPool(connection)
  } yield sessions

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

Using resources

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

NB: not actual code, Resource is polymorphic in the effect type

How to fill the abstraction gap?



JmsContext - 2nd iteration

```scala
sealed abstract class JmsContext(private[lib] val raw: javax.jms.JMSContext) {

  def createQueue(queue: QueueName): IO[JmsQueue] =
    IO.delay(new JmsQueue(raw.createQueue(queue.value)))

  def makeJmsConsumer(queueName: QueueName): Resource[IO, JmsMessageConsumer] =
    for {
      destination <- Resource.eval(createQueue(queueName))
      consumer    <- Resource.fromAutoCloseable(
        IO.delay(raw.createConsumer(destination.wrapped)))
    } yield new JmsMessageConsumer(consumer)
}

class JmsTransactedContext private[lib] (
  override private[lib] val raw: javax.jms.JMSContext) extends JmsContext(raw)
```

- handle JMSRuntimeException ✅
- handle the resource lifecycle ✅

JmsContext - 2nd iteration

```scala
sealed abstract class JmsContext(private[lib] val raw: javax.jms.JMSContext) {

  def createQueue(queue: QueueName): IO[JmsQueue] =
    IO.delay(new JmsQueue(raw.createQueue(queue.value)))

  def makeJmsConsumer(queueName: QueueName): Resource[IO, JmsMessageConsumer] =
    for {
      destination <- Resource.eval(createQueue(queueName))
      consumer    <- Resource.fromAutoCloseable(
        IO.delay(raw.createConsumer(destination.wrapped)))
    } yield new JmsMessageConsumer(consumer)
}

class JmsTransactedContext private[lib] (
  override private[lib] val raw: javax.jms.JMSContext) extends JmsContext(raw)
```

- handle JMSRuntimeException ✅
- handle the resource lifecycle ✅

JmsContext - 2nd iteration

```scala
sealed abstract class JmsContext(private[lib] val raw: javax.jms.JMSContext) {

  def createQueue(queue: QueueName): IO[JmsQueue] =
    IO.delay(new JmsQueue(raw.createQueue(queue.value)))

  def makeJmsConsumer(queueName: QueueName): Resource[IO, JmsMessageConsumer] =
    for {
      destination <- Resource.eval(createQueue(queueName))
      consumer    <- Resource.fromAutoCloseable(
        IO.delay(raw.createConsumer(destination.wrapped)))
    } yield new JmsMessageConsumer(consumer)
}

class JmsTransactedContext private[lib] (
  override private[lib] val raw: javax.jms.JMSContext) extends JmsContext(raw)
```

- handle JMSRuntimeException ✅
- handle the resource lifecycle ✅

How to receive?

JmsMessageConsumer

```scala
class JmsMessageConsumer private[lib] (
  private[lib] val wrapped: javax.jms.JMSConsumer
) {
  val receive: IO[JmsMessage] =
    for {
      recOpt <- IO.delay(Option(wrapped.receiveNoWait()))
      rec    <- recOpt match {
        case Some(message) => IO.pure(new JmsMessage(message))
        case None          => receive
      }
    } yield rec
}
```

- only exposing `receive`, which is an `IO` value which:
  - ***repeats*** a **check-and-receive** operation (`receiveNoWait()`) till a message is ready
  - ***completes*** the IO with the message read

JmsMessageConsumer

```scala
class JmsMessageConsumer private[lib] (
  private[lib] val wrapped: javax.jms.JMSConsumer
) {
  val receive: IO[JmsMessage] =
    for {
      recOpt <- IO.delay(Option(wrapped.receiveNoWait()))
      rec    <- recOpt match {
        case Some(message) => IO.pure(new JmsMessage(message))
        case None          => receive
      }
    } yield rec
}
```

- only exposing `receive`, which is an `IO` value which:
  - ***repeats*** a **check-and-receive** operation (`receiveNoWait()`) till a message is ready
  - ***completes*** the IO with the message read

JmsMessageConsumer

```
class JmsMessageConsumer private[lib] (
  private[lib] val wrapped: javax.jms.JMSConsumer
) {
  val receive: IO[JmsMessage] =
    for {
      recOpt <- IO.delay(Option(wrapped.receiveNoWait()))
      rec    <- recOpt match {
        case Some(message) => IO.pure(new JmsMessage(message))
        case None          => receive
      }
    } yield rec
}
```

- only exposing `receive`, which is an `IO` value which:
  - ***repeats*** a **check-and-receive** operation (`receiveNoWait()`) till a message is ready
  - ***completes*** the IO with the message read

JmsMessageConsumer

```
class JmsMessageConsumer private[lib] (
  private[lib] val wrapped: javax.jms.JMSConsumer
) {
  val receive: IO[JmsMessage] =
    for {
      recOpt <- IO.delay(Option(wrapped.receiveNoWait()))
      rec    <- recOpt match {
        case Some(message) => IO.pure(new JmsMessage(message))
        case None          => receive
      }
    } yield rec
}
```

- only exposing `receive`, which is an `IO` value which:
    - ***repeats*** a **check-and-receive** operation (`receiveNoWait()`) till a message is ready
    - ***completes*** the IO with the message read

JmsMessageConsumer

```scala
class JmsMessageConsumer private[lib] (
  private[lib] val wrapped: javax.jms.JMSConsumer
) {
  val receive: IO[JmsMessage] =
    for {
      recOpt <- IO.delay(Option(wrapped.receiveNoWait()))
      rec    <- recOpt match {
        case Some(message) => IO.pure(new JmsMessage(message))
        case None          => receive
      }
    } yield rec
}
```

- only exposing `receive`, which is an `IO` value which:
  - ***repeats*** a **check-and-receive** operation (`receiveNoWait()`) till a message is ready
  - ***completes*** the IO with the message read

JmsMessageConsumer - alternative

```scala
class JmsMessageConsumer private[lib] (
  private[lib] val wrapped: javax.jms.JMSConsumer,
  private[lib] val pollingInterval: FiniteDuration
) {

  val receive: IO[JmsMessage] =
    for {
      recOpt <- IO.blocking(Option(wrapped.receive(pollingInterval.toMillis)))
      rec    <- recOpt match {
        case Some(message) => IO.pure(new JmsMessage(message))
        case None          => receive
      }
    } yield rec
}
```

- pretty much the same as the former
- leveraging `receive(timeout)` and wrapping the blocking operation in `IO.blocking`

JmsMessageConsumer - alternative

```scala
class JmsMessageConsumer private[lib] (
  private[lib] val wrapped: javax.jms.JMSConsumer,
  private[lib] val pollingInterval: FiniteDuration
) {

  val receive: IO[JmsMessage] =
    for {
      recOpt <- IO.blocking(Option(wrapped.receive(pollingInterval.toMillis)))
      rec    <- recOpt match {
        case Some(message) => IO.pure(new JmsMessage(message))
        case None          => receive
      }
    } yield rec
}
```

- pretty much the same as the former
- leveraging `receive(timeout)` and wrapping the blocking operation in `IO.blocking`

A nearly working example

```
object SampleConsumer extends IOApp.Simple {
  override def run: IO[Unit] = {
    val jmsConsumerRes = for {
      jmsContext <- ??? // A Resource[JmsContext] instance for a given provider
      consumer   <- jmsContext.makeJmsConsumer(queueName)
    } yield consumer

    jmsConsumerRes
      .use(consumer =>
        for {
          msg     <- consumer.receive
          textMsg <- IO.fromTry(msg.tryAsJmsTextMessage)
          _       <- logger.info(s"Got 1 message with text: $textMsg. Ending now.")
        } yield ()
      )
  }
}
```

- IOApp describes a *main* which **executes** an IO
- It's the single *entry point* to a **pure** program (a.k.a. *End of the world*).
- It **runs** (interprets) the effects described in the IO!

Adding support for a provider

```scala
object ibmMQ {
  // ...
  def makeJmsTransactedContext(config: Config): Resource[IO, JmsTransactedContext] =
    for {
      context <- Resource.fromAutoCloseable(IO.delay {
        val connectionFactory: MQConnectionFactory = new MQConnectionFactory()
        connectionFactory.setTransportType(CommonConstants.WMQ_CM_CLIENT)
        connectionFactory.setQueueManager(config.qm.value)
        connectionFactory.setConnectionNameList(hosts(config.endpoints))
        connectionFactory.setChannel(config.channel.value)
        connectionFactory.createContext(
          username.value,
          config.password,
          javax.jms.Session.SESSION_TRANSACTED // support for at-least-once
        )
      })
    } yield new JmsTransactedContext(context)
}
```

That's it!

DONE:

- **effects** handled respecting *referential transparency*
- **resources** get acquired and released in order, the user **can't leak** them
- the **business logic** is made by ***pure functions***

TODO:

- still **low level**
- how to specify message confirmation?
- what if the user needs to implement a **never-ending concurrent message consumer**?

Switching to top-down

- Let's evaluate how we can model an api for a **never-ending message consumer**!

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

AtLeastOnceConsumer - 1st iteration

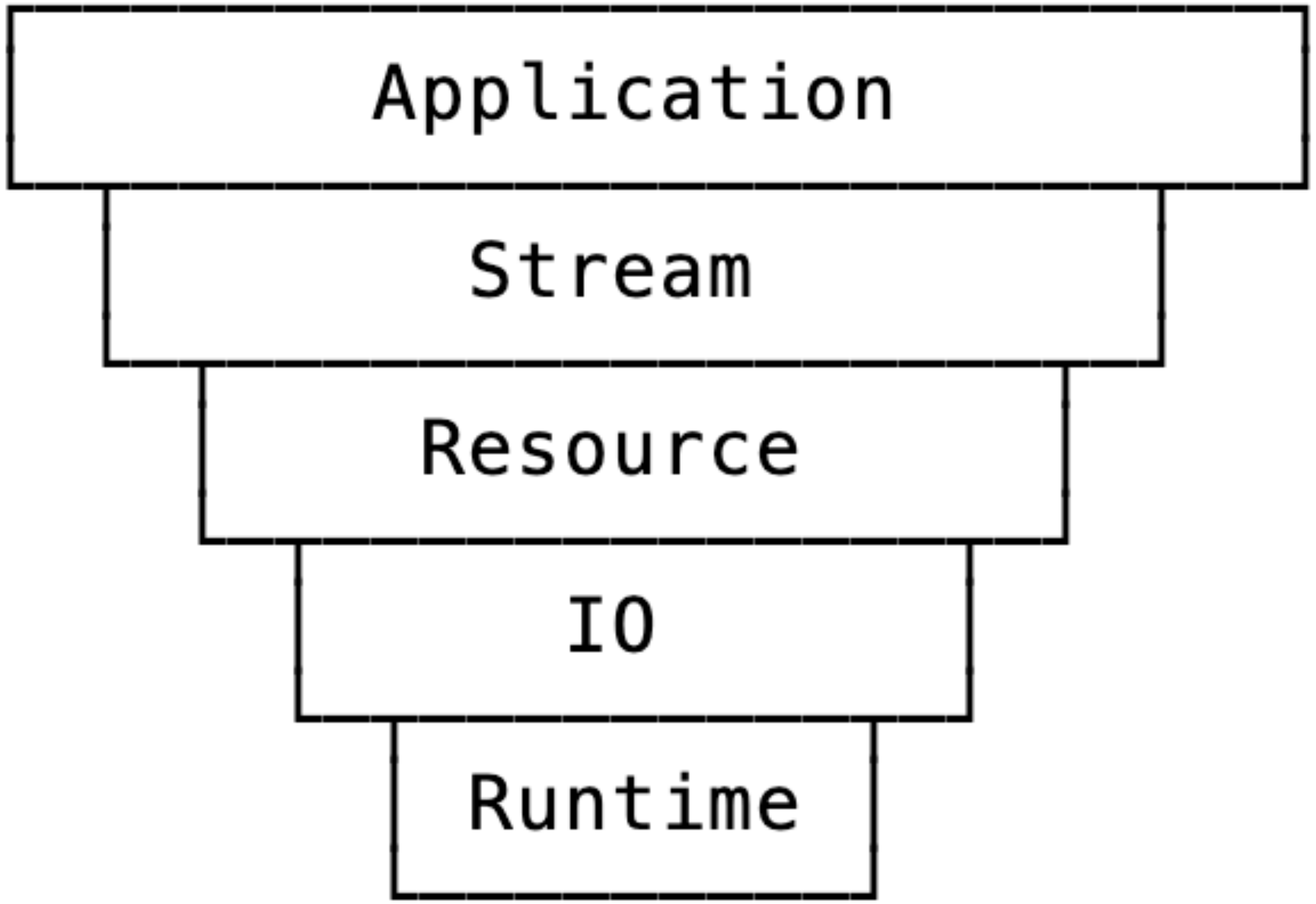
```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

Introducing Stream

A *sequence* of effectful computation

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

NB: not actual code, just a simplification sticking with the 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 map[O2](f: O => O2): Stream[O2]
  def flatMap[O2](f: O => Stream[O2]): Stream[O2]
  ...
  def evalMap[O2](f: O => IO[O2]): Stream[O2]
  def repeat: Stream[O]
}
```

NB: not actual code, just a simplification sticking with the 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 map[O2](f: O => O2): Stream[O2]
  def flatMap[O2](f: O => Stream[O2]): Stream[O2]
  ...
  def evalMap[O2](f: O => IO[O2]): Stream[O2]
  def repeat: Stream[O]
}
```

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

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

- Inspired by fs2-rabbit

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

- Inspired by fs2-rabbit

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

- Inspired by fs2-rabbit

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

- Inspired by fs2-rabbit

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

- Inspired by fs2-rabbit

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

- Inspired by fs2-rabbit

AtLeastOnceConsumer - 1st iteration

```scala
object AtLeastOnceConsumer {

  sealed trait CommitAction
  object CommitAction {
    case object Commit   extends CommitAction
    case object Rollback extends CommitAction
  }

  type Committer = CommitAction => IO[Unit]
  type Consumer  = Stream[IO, JmsMessage]

  def make(context: JmsTransactedContext,
           queueName: QueueName): Resource[IO, (Consumer, Committer)] = {
    val committer = (txRes: CommitAction) =>
      txRes match {
        case CommitAction.Commit   => IO.blocking(context.raw.commit())
        case CommitAction.Rollback => IO.blocking(context.raw.rollback())
      }
    val buildStreamingConsumer = (consumer: JmsMessageConsumer) =>
      Stream.eval(consumer.receive).repeat

    context
      .makeJmsConsumer(queueName)
      .map(buildStreamingConsumer)
      .map(consumer => (consumer, committer))
  }
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use {
          case (consumer, committer) =>
            consumer.evalMap { msg =>
              // whatever business logic you need to perform
              logger.info(msg.show) >>
                committer(CommitAction.Commit)
            }
            .compile.drain
        }
}
```

- Inspired by fs2-rabbit

AtLeastOnceConsumer - 1st iteration

- all **effects** are expressed in the types (`IO`, etc...) ✅
- **resource lifecycle** handled via `Resource` ✅
- messages in the queue are exposed via a `Stream` ✅

AtLeastOnceConsumer - 1st iteration

But…

- what happens if the client **forget to `commit`/`rollback`**?

```
consumer.evalMap { msg =>
  logger.info(msg.show)
}
```

- what happens if the client **`commit`/`rollback` multiple times** the same message?

```
consumer.evalMap { msg =>
  committer(CommitAction.Commit) >>
    committer(CommitAction.Rollback)
}
```

- what happens if the client **evaluates the stream multiple times**?

```
consumer.evalMap{ ... } ++
  consumer.evalMap{ ... }
```

- how to **support concurrency**?

Can we do better?

- Let's think how is the API we'd like to expose...
- And evaluate how to actually implement that!

AtLeastOnceConsumer - 2nd iteration

Ideally...

```
consumer.handle { msg =>
  for {
    _ <- logger.info(msg.show)
    _ <- ??? // ... actual business logic...
  } yield TransactionResult.Commit
}
```

- `handle` should be provided with a function `JmsMessage => IO[TransactionResult]`
  - **lower chanches for the client to do the wrong thing**!
- if errors are raised in the handle function, this is a bug and the program will terminate without confirming the message
- **errors regarding the business logic should be handled inside the program, reacting accordingly** (ending with either a commit or a rollback)

AtLeastOnceConsumer - 2nd iteration

```scala
class AtLeastOnceConsumer private[lib] (
  private[lib] val ctx: JmsContext,
  private[lib] val consumer: JmsMessageConsumer
) {

  def handle(
    runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
      consumer.receive
        .flatMap(runBusinessLogic)
        .flatMap {
          case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
          case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
        }
        .foreverM
}

object AtLeastOnceConsumer {
  sealed trait TransactionResult
  object TransactionResult {
    case object Commit   extends TransactionResult
    case object Rollback extends TransactionResult
  }

  def make(
    context: JmsTransactedContext,
    queueName: QueueName): Resource[IO, AtLeastOnceConsumer] =
      context.makeJmsConsumer(queueName).map(consumer =>
        new AtLeastOnceConsumer(context, consumer))
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

AtLeastOnceConsumer - 2nd iteration

```scala
class AtLeastOnceConsumer private[lib] (
  private[lib] val ctx: JmsContext,
  private[lib] val consumer: JmsMessageConsumer
) {

  def handle(
    runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
      consumer.receive
        .flatMap(runBusinessLogic)
        .flatMap {
          case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
          case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
        }
        .foreverM
}

object AtLeastOnceConsumer {
  sealed trait TransactionResult
  object TransactionResult {
    case object Commit   extends TransactionResult
    case object Rollback extends TransactionResult
  }

  def make(
    context: JmsTransactedContext,
    queueName: QueueName): Resource[IO, AtLeastOnceConsumer] =
      context.makeJmsConsumer(queueName).map(consumer =>
        new AtLeastOnceConsumer(context, consumer))
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

AtLeastOnceConsumer - 2nd iteration

```scala
class AtLeastOnceConsumer private[lib] (
  private[lib] val ctx: JmsContext,
  private[lib] val consumer: JmsMessageConsumer
) {

  def handle(
    runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
      consumer.receive
        .flatMap(runBusinessLogic)
        .flatMap {
          case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
          case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
        }
        .foreverM
}

object AtLeastOnceConsumer {
  sealed trait TransactionResult
  object TransactionResult {
    case object Commit   extends TransactionResult
    case object Rollback extends TransactionResult
  }

  def make(
    context: JmsTransactedContext,
    queueName: QueueName): Resource[IO, AtLeastOnceConsumer] =
      context.makeJmsConsumer(queueName).map(consumer =>
        new AtLeastOnceConsumer(context, consumer))
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

AtLeastOnceConsumer - 2nd iteration

```scala
class AtLeastOnceConsumer private[lib] (
  private[lib] val ctx: JmsContext,
  private[lib] val consumer: JmsMessageConsumer
) {

  def handle(
    runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
      consumer.receive
        .flatMap(runBusinessLogic)
        .flatMap {
          case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
          case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
        }
        .foreverM
}

object AtLeastOnceConsumer {
  sealed trait TransactionResult
  object TransactionResult {
    case object Commit   extends TransactionResult
    case object Rollback extends TransactionResult
  }

  def make(
    context: JmsTransactedContext,
    queueName: QueueName): Resource[IO, AtLeastOnceConsumer] =
      context.makeJmsConsumer(queueName).map(consumer =>
        new AtLeastOnceConsumer(context, consumer))
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

AtLeastOnceConsumer - 2nd iteration

```scala
class AtLeastOnceConsumer private[lib] (
  private[lib] val ctx: JmsContext,
  private[lib] val consumer: JmsMessageConsumer
) {

  def handle(
    runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
      consumer.receive
        .flatMap(runBusinessLogic)
        .flatMap {
          case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
          case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
        }
        .foreverM
}

object AtLeastOnceConsumer {
  sealed trait TransactionResult
  object TransactionResult {
    case object Commit   extends TransactionResult
    case object Rollback extends TransactionResult
  }

  def make(
    context: JmsTransactedContext,
    queueName: QueueName): Resource[IO, AtLeastOnceConsumer] =
      context.makeJmsConsumer(queueName).map(consumer =>
        new AtLeastOnceConsumer(context, consumer))
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
      .use(consumer =>
        consumer.handle { msg =>
          for {
            _ <- logger.info(msg.show)
            _ <- ??? // ... actual business logic...
          } yield TransactionResult.Commit
        }
      )
}
```

AtLeastOnceConsumer - 2nd iteration

```scala
class AtLeastOnceConsumer private[lib] (
  private[lib] val ctx: JmsContext,
  private[lib] val consumer: JmsMessageConsumer
) {

  def handle(
    runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
      consumer.receive
        .flatMap(runBusinessLogic)
        .flatMap {
          case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
          case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
        }
        .foreverM
}

object AtLeastOnceConsumer {
  sealed trait TransactionResult
  object TransactionResult {
    case object Commit   extends TransactionResult
    case object Rollback extends TransactionResult
  }

  def make(
    context: JmsTransactedContext,
    queueName: QueueName): Resource[IO, AtLeastOnceConsumer] =
      context.makeJmsConsumer(queueName).map(consumer =>
        new AtLeastOnceConsumer(context, consumer))
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

AtLeastOnceConsumer - 2nd iteration

- all **effects** are expressed in the types (`IO`, etc...) ✅
- **resource lifecycle** handled via `Resource` ✅
- not exposing messages to `Stream` anymore, **it made things harder to get the design right**!
- the client is ~~forced~~ guided to do the right thing™ ✅

Still, *concurrency* is not there yet...

Supporting concurrency

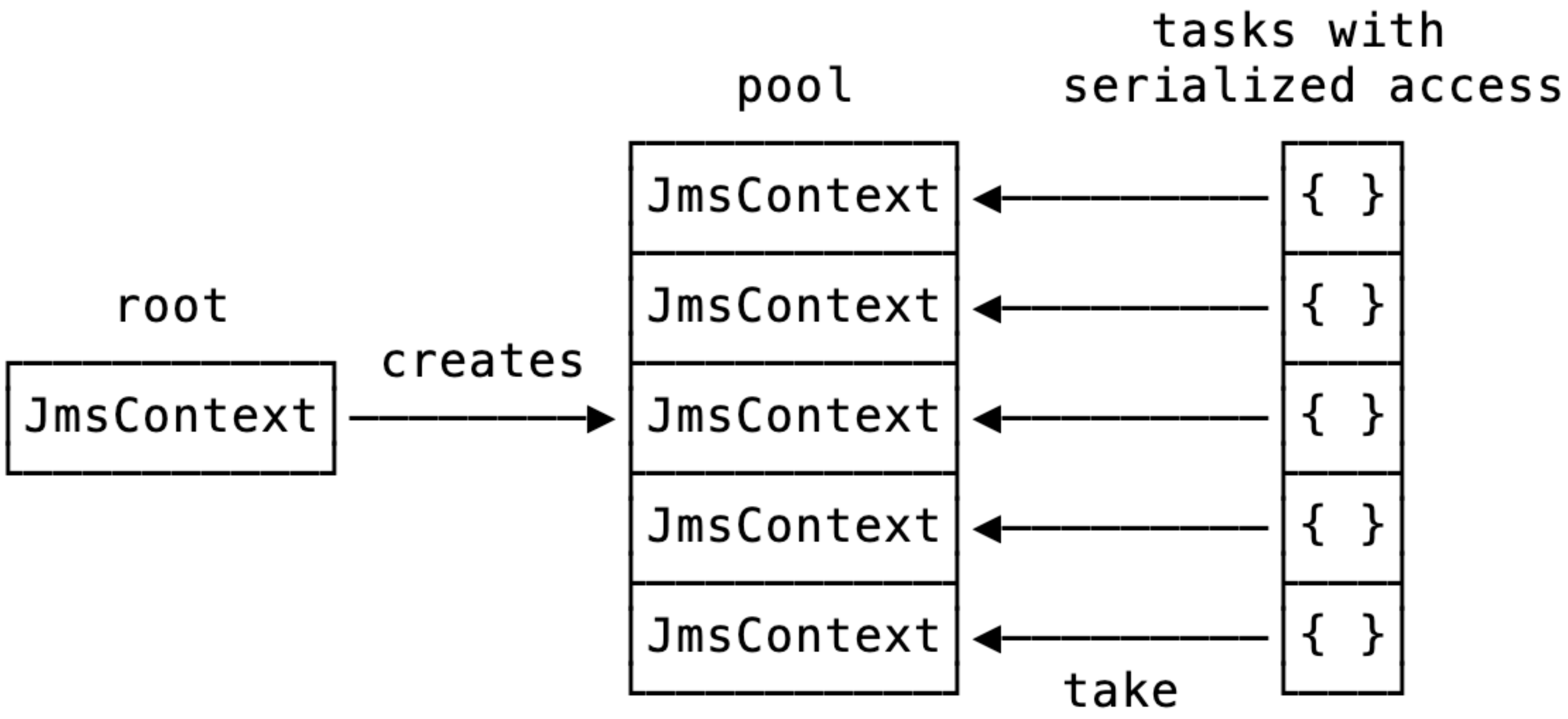
back to bottom-up…

- A `JMSContext` is the main interface in the simplified JMS API introduced for JMS 2.0.
- In terms of the JMS 1.1 API a `JMSContext` should be thought of as representing both a `Connection` and a `Session`
- A **connection** represents a physical link to the JMS server and a **session** represents a **single-threaded context** for sending and receiving messages.
- Applications which require **multiple sessions** to be created on the same connection should:
  - create a root contenxt using the `createContext` methods on the `ConnectionFactory`
  - then use the `createContext` method on the root context to create additional contexts instances that use the same connection
  - all these `JMSContext` objects are application-managed and must be closed when no longer needed by calling their close method.
- **JmsContext is not thread-safe!**

Ref: https://docs.oracle.com/javaee/7/api/javax/jms/JMSContext.html

Supporting concurrency

back to bottom-up...



AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from `typelevel/keypool`

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from `typelevel/keypool`

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from typelevel/keypool

AtLeastOnceConsumer - 3rd iteration

```
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from typelevel/keypool

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from typelevel/keypool

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from `typelevel/keypool`

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from typelevel/keypool

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from typelevel/keypool

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from typelevel/keypool

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from typelevel/keypool

AtLeastOnceConsumer - 3rd iteration

```scala
object AtLeastOnceConsumer {

  def make(
    rootContext: JmsTransactedContext,
    queueName: QueueName,
    concurrencyLevel: Int
  ): Resource[IO, AtLeastOnceConsumer] =
    Pool.Builder(
      for {
        ctx      <- rootContext.makeTransactedContext
        consumer <- ctx.makeJmsConsumer(queueName)
      } yield (ctx, consumer)
    )
    .withMaxTotal(concurrencyLevel)
    .build
    .map(pool => new AtLeastOnceConsumer(pool, concurrencyLevel))
}

class AtLeastOnceConsumer private[lib] (
  private[lib] val pool: Pool[IO, (JmsContext, JmsMessageConsumer)],
  private[lib] val concurrencyLevel: Int
) {

  def handle(runBusinessLogic: JmsMessage => IO[TransactionResult]): IO[Nothing] =
    IO.parSequenceN(concurrencyLevel) {
        pool.take.use { res =>
          val (ctx, consumer) = res.value
          for {
            message <- consumer.receive
            txRes   <- runBusinessLogic(message)
            _ <- txRes match {
              case TransactionResult.Commit   => IO.blocking(ctx.raw.commit())
              case TransactionResult.Rollback => IO.blocking(ctx.raw.rollback())
            }
          } yield ()
        }
      }
      .foreverM
}
```

```scala
object Demo extends IOApp.Simple {

  override def run: IO[Unit] =
    jmsTransactedContextRes.flatMap(ctx =>
      AtLeastOnceConsumer.make(ctx, queueName, 5))
        .use(consumer =>
          consumer.handle { msg =>
            for {
              _ <- logger.info(msg.show)
              _ <- ??? // ... actual business logic...
            } yield TransactionResult.Commit
          }
        )
}
```

- Using a proper resource pool, from typelevel/keypool

AtLeastOnceConsumer - 3rd iteration

- all **effects** are expressed in the types (`IO`, etc...) ✅
- **resource lifecycle** handled via `Resource` ✅
- the client is ~~forced~~ guided to do the right thing™ ✅
- **concurrency** ✅

We came a long way...

- We used a bunch of **data types** (`IO`, `Resource`, `Stream`)
- We used a bunch of **common operators** (`map`, `flatMap`)
- We wrote a little code, **iteratively improving the design**
- We achieved what we needed: a fully functioning functional minimal lib

References

- https://github.com/AL333Z/fp-lib-design
- https://github.com/typelevel/cats-effect
- https://github.com/fp-in-bo/jms4s
- https://github.com/profunktor/fs2-rabbit

Thanks

Bonus
Tagless final

- Couldn't fit in 45 minutes :)
- The actual lib is written with Tagless Final.

Bonus
Other ecosystems?

- Not worth it, for very different reasons.
  - Lightbend stack: not as composable as the FP counterpart, side-effects, missing referential transparent abstractions for effects.
  - ZIO: I just don't like their rhetoric.