Stream Processing

With Kafka, scalaz, fs2 – experience report

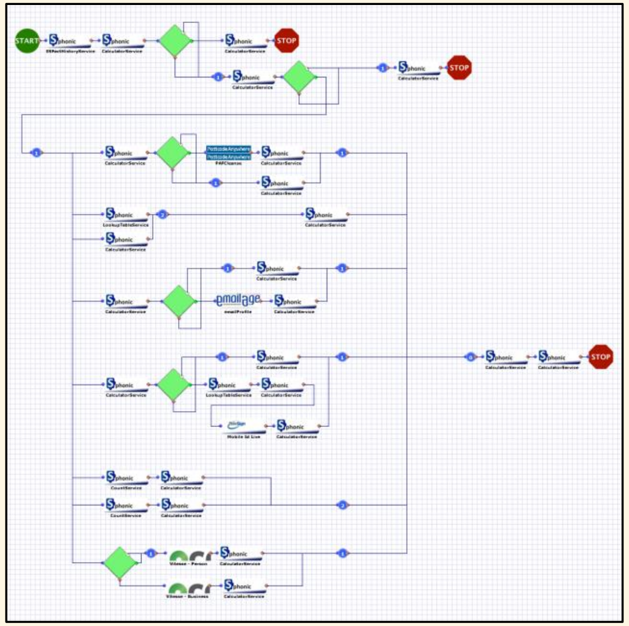
jannmueller@sphonic.com

9 December 2016

What we do

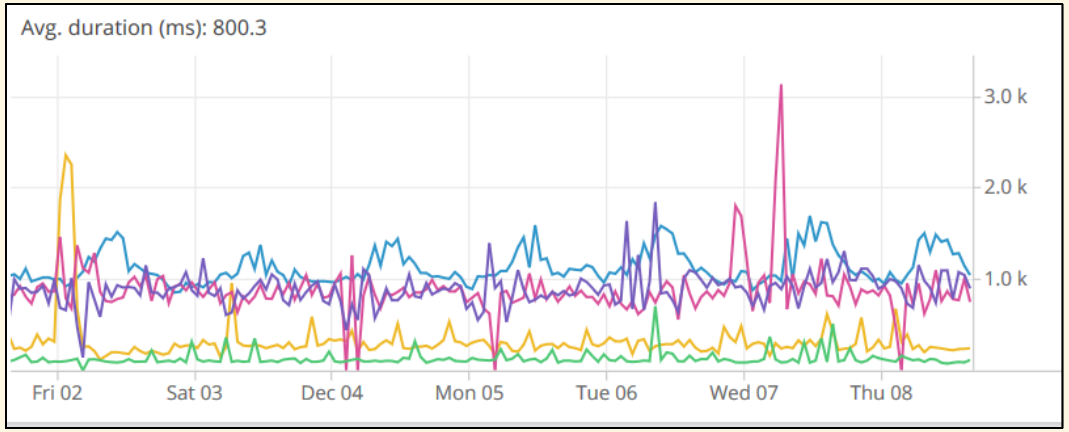
SPHONIC

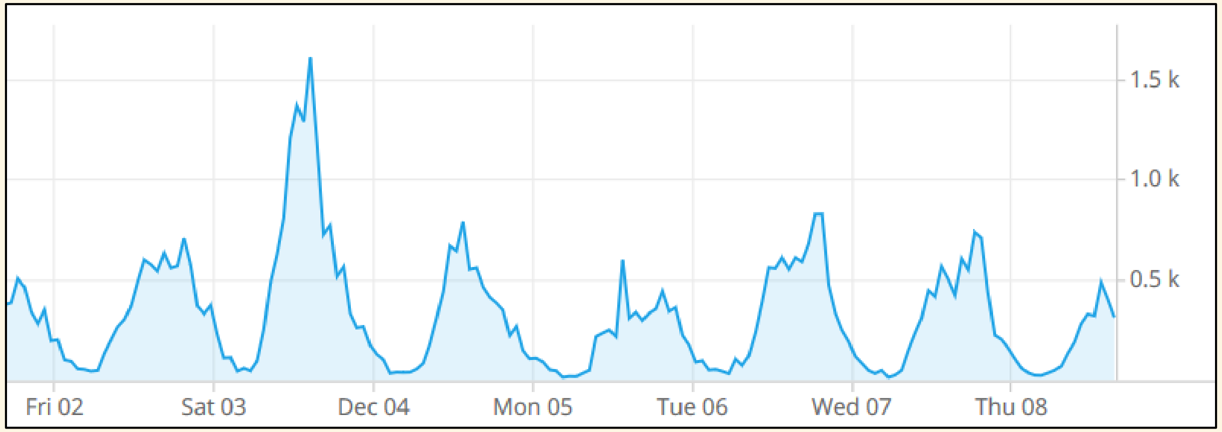
- Fraud risk management
  - Address & age verification, anti-money laundering etc.
  - Connect to ~60 3rd party APIs
  - Post-processing: data mining, analytics
- Custom workflows for each client
- Graphical editor + expression DSL



Performance & Scale

- Relatively low tps (<10 on average), high variance
  - Batches
  - Seasonal/time-of-day
- Latency highly dependent on API calls
  - Most time is spent waiting for other services

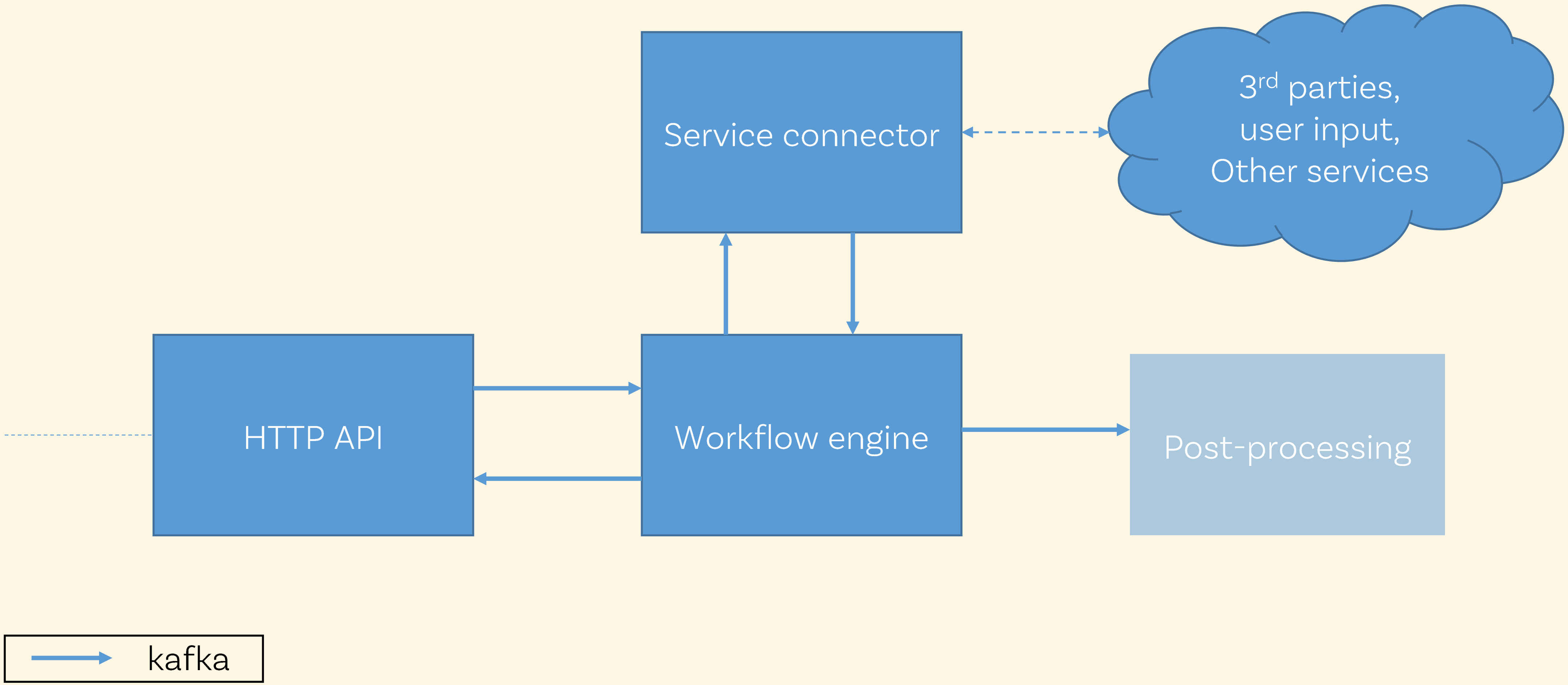




Scala @ Sphonic

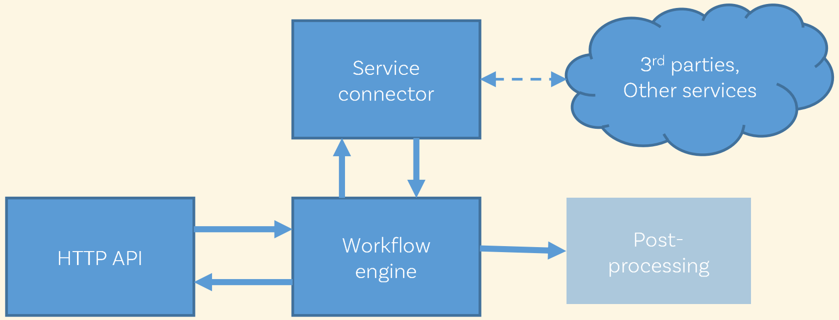
- Current workflow engine:
  - Monolithic, hard to debug, does not scale etc.
  - BUT post-transaction processing already based on kafka
- We did a proof-of-concept for 1 client
  - fs2, scalaz, finagle, finch, algebird, …
- Good results, but workflow is hard-coded
  - To generalise it, we need a stream-based workflow execution engine
  - *That's what the rest of this talk is about*

Workflows as streams
Architecture



Workflows as streams

All side effects are events



- Long-running transactions!
- Persistence!
- Debugging!
- Upgrades!
- Scalacheck for workflows!

Workflows as streams
Input & Output format

- A transaction is a stream of `APIResponses`

```
case class APIResponse(value: String)
```

Workflows as streams

Input & Output format

- A transaction is a stream of `APIResponses`
- The execution engine produces a stream of `APIRequest[Unit]`s and a final result `R`

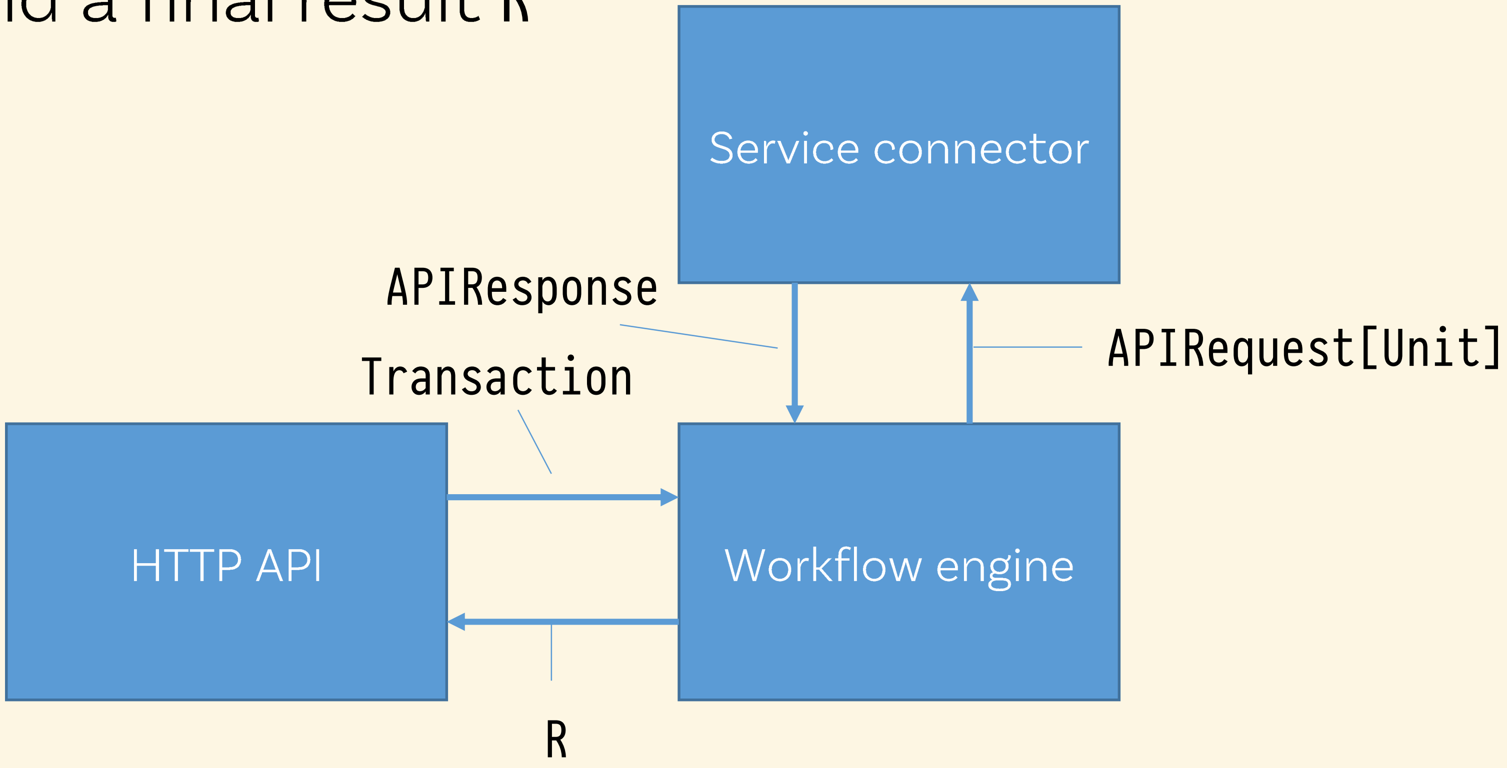
```
case class APIRequest[S](vendorName: String, payload: String, value: S)
```

- Note that $String \times String \times Unit \simeq String \times String$, but we need the S parameter later on

Workflows as streams

Input & Output format

- A transaction is a stream of `APIResponses`
- The execution engine produces a stream of `APIRequest[Unit]`s and a final result `R`



Workflows as streams

Input & Output format

- A transaction is a stream of `APIResponses`
- The execution engine produces a stream of `APIRequest[Unit]`s and a final result `R`
- Conceptually:

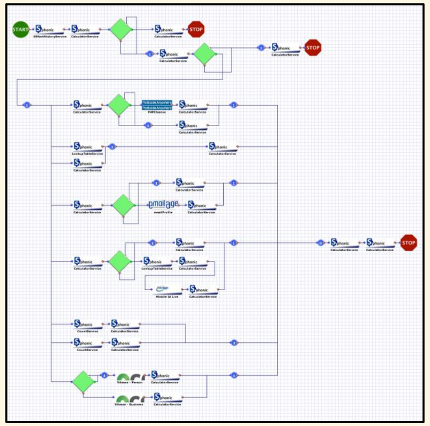
```
type Workflow = Transaction ⇒ Seq[APIResponse] ⇒ Seq[Either[APIRequest[Unit], R]]
```

Workflows as streams

Finite state machines (Moore)

```
trait FSM[-E, +S] {
  def apply(input: E): FSM[E, S]
  def getState: S
}
```

- Resumable / partial execution
- BUT Cannot combine easily
  - Need some combinators to interpret our workflows



Workflow DSL

What is out there

- Streams/iteratees
  - Many implementations: Machines (Haskell), scalaz-stream, …
  - Can be resumed
  - Do not support backtracking
- Parser combinators
  - Backtracking
  - Cannot be resumed

- Also, how can we do requests (to external services)?

Workflow DSL

Solution

- Let's build a hybrid

```
trait Plan[K, O, M[_], A] {
  def map[B](f: A ⇒ B)(implicit M: Functor[M]): Plan[K, O, M, B]
  def flatMap[B](f: A ⇒ Plan[K, O, M, B])(implicit M: Functor[M]): Plan[K, O,
   M, B]
  def mapOutputs[P](f: O ⇒ P)(implicit M: Functor[M]): Plan[K, P, M, A]
  def mapInputs[J](f: J ⇒ K)(implicit M: Functor[M]): Plan[J, O, M, A]
  def orElse(planB: ⇒ Plan[K, O, M, A])(implicit M: Functor[M]): Plan[K, O,
  M, A]
  def foldMap[B](f: O ⇒ B)(implicit B: Monoid[B]): B
  def nextInput(implicit M: Functor[M]): Seq[Input[M[Unit]]]
  def feed(input: Input[K])(implicit A: Applicative[M]): M[Plan[K, O, M, A]]
  def try(implicit M: Functor[M]): Plan[K, O, M, A]
}
```

Workflow DSL

Solution

```scala
object Plan {
  def done[K, O, M[_], A](a: A): Plan[K, O, M, A] = ???
  def write[K, O, M[_], A](y: O, plan: Plan[K, O, M, A]): Plan[K, O, M, A] = ???
  def await[K, O, M[_], A](await: M[K ⇒ Plan[K, O, M, A]]): Plan[K, O, M, A] =
  ???
  def fail[K, O, M[_], A](msg: String): Plan[K, O, M, A] = ???
  def parallel[K, O, M[_], A](left: Plan[K, O, M, A], right: Plan[K, O, M, A]):
  Plan[K, O, M, (A, A)] = ???
}
```

Parallel Branches

- Branches of a workflow can run in parallel
- In parallel.feed we need to know which branch to run

```
def parallel[K, O, M[_], A](left: Plan[K, O, M, A], right: Plan[K, O, M, A])
```

Parallel Branches

- Branches of a workflow can run in parallel
- In parallel.feed we need to know which branch to run
- So we encode the branch in the input type:

```scala
sealed trait Input[A] { def get: A }
case class L[A](inner: Input[A]) extends Input[A]
case class R[A](inner: Input[A]) extends Input[A]
case class M[A](a: A) extends Input[A]
```

Parallel Branches

- Branches of a workflow can run in parallel
- In parallel.feed we need to know which branch to run
- So we encode the branch in the input type

```
def parallel[K, O, M[_], A](left: Plan[K, O, M, A], right: Plan[K, O, M, A]):
Plan[K, O, M, (A, A)] = new Plan[K, O, M, (A, A)] {

  def feed(input: Input[K])(implicit A: Applicative[M]): M[Plan[K, O, M, (A, A)]
  ] = input match {
    case L(i) => ??? // feed left branch
    case R(i) => ??? //feed right branch
  }
}
```

Parallel Branches

- Branches of a workflow can run in parallel
- In parallel.feed we need to know which branch to run
- So we encode the branch in the input type
- And wrap the inputs in parallel.nextInputs

```
def parallel[K, O, M[_], A](left: Plan[K, O, M, A], right: Plan[K, O, M, A]):
Plan[K, O, M, (A, A)] = new Plan[K, O, M, (A, A)] {

  def nextInput(implicit M: Functor[M]): Seq[Input[M[Unit]]] =
    left.nextInput.map(L(_)) ++ right.nextInput.map(R(_))

}
```

Requesting Input

- `await` needs to specify the request parameters
  - Which service to call, arguments, etc.

```
def await[K, O, M[_], A](await: M[K => Plan[K, O, M, A]]): Plan[K, O, M, A]
```

Requesting Input

- `await` needs to specify the request parameters
  - Which service to call, arguments, etc.
- In `nextInput` we return `M[Unit]`

```
def await[K, O, M[_], A](await: M[K => Plan[K, O, M, A]]): Plan[K, O, M, A] =
new Plan[K, O, M, A] {

  def nextInput(implicit F: Functor[M]): Seq[Input[M[Unit]]] =
    M(F.map(await)(_ => ())) :: Nil

}
```

Requesting Input

- `await` needs to specify the request parameters
  - Which service to call, arguments, etc.
- In `nextInput` we return `M[Unit]`
- In `feed` we unwrap the innermost `Input` value

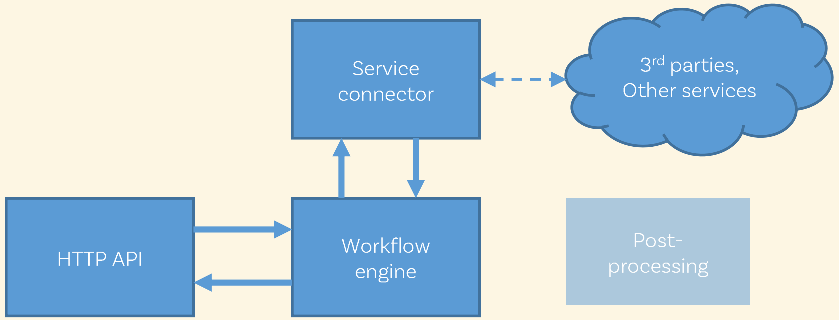
```
def await[K, O, M[_], A](await: M[K ⇒ Plan[K, O, M, A]]): Plan[K, O, M, A] =
new Plan[K, O, M, A] {

  def feed(input: Input[K])(implicit A: Applicative[M]): M[Plan[K, O, M, A]] =
    input match {
      case M(i) ⇒ A.map(await)({ ff ⇒ ff(i) })
  }
}
```

Requesting Input

`Functor[Input]`

- The service connector implements `Input[M[Unit]] => Input[K]`



- This means that `K` should be able to express failed requests
  - We want to handle most failures in the workflow language
  - For example, retries can be done without a round trip to the workflow engine, but failover (vendor down) should be managed in the workflow

Workflow DSL

Generating a finite state machine

- Run a plan on a sequence of inputs
- Collect partial results along the way

```
object Plan {
  def toFSM[K, O, M[_]](plan: Plan[K, O, M, Nothing])(
    implicit M: Applicative[M], MM: Monoid[O]):
     FSM[Input[K], (O, Option[Input[M[Unit]]])]
}
```

Workflow DSL

Generating a finite state machine

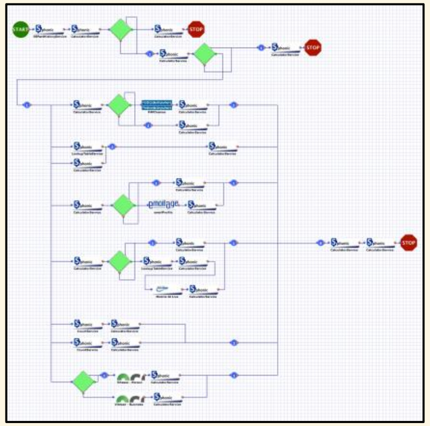
- Run a plan on a sequence of inputs
- Collect partial results along the way
- To get a `Plan[K, O, M, Nothing]`:

```
def finish[K, O, M[_], A](plan: Plan[K, O, M, A]): Plan[K, O, M, Nothing] =
  plan.flatMap({ _ => Plan.fail("") })
```

Workflow DSL

Using the language

- We store workflows as ASTs and interpret them using the `Plan` combinators
- But we can also write workflows in Scala directly (embedded DSL)



Workflow DSL

Using the language

- We store workflows as ASTs and interpret them using the `Plan` combinators
- But we can also write workflows in Scala directly (embedded DSL)
- The output type should have an associative operation to enable intermediate results – `scalaz.Semiring[O]`
  - For example:

```scala
type Facts = Map[String, String] // not really
```

Workflow DSL

Using the language

- We store workflows as ASTs and interpret them using the `Plan` combinators
- But we can also write workflows in Scala directly (embedded DSL)
- The output type should be monoidal to enable intermediate results – `scalaz.Monoid[O]`
- If there are no reasonable append/zero operations, use `Last[O]`

Workflow DSL
Conveniences

```
object Plans {

  def emit[K, O, M[_], A](output: O)(implicit M: Functor[M]): Plan[K, O, M,
  Unit] =
    Plan.write(output, Plan.done(()))

  def request[O](vendor: String, payload: String): Plan[APIResponse, O,
  APIRequest, String] =
    Plan.await(APIRequest(vendor, payload, { r: APIResponse => r.value })

  def verifyAddress[O](address: Address): Plan[APIResponse, O, APIRequest,
  Boolean] = Plans
    .request[O]("addressCheck", address.toString)
    .map(_ == "OK")

  def verifyAge[O](person: Identity): Plan[APIResponse, O, APIRequest, Boolean]
  = Plans
    .request[O]("ageCheck", person.toString)
    .map({ s: String => s.toInt > 18 })

}
```

Workflow DSL

Example

```
object Workflows {

  def identityCheck(i: Identity): Plan[APIResponse, Facts, APIRequest, Unit] =
  for {
    _ ← Plans.emit(i.toFacts)
    (addressOK, ageOK) ← Plan.parallel(
      Plans.verifyAddress(i.address),
      Plans.verifyAge(i))
    result = addressOK && ageOK
    _ ← Plans.emit(Map("result" → result.toString))
  } yield ()

}
```

Working with workflows

Property tests

- Define case classes & arbitrary instances for each kind of input data

```
case class GermanAccount(account: Identity)
case class FrenchAccount(account: Identity)
```

Working with workflows

Property tests

- Define case classes & arbitrary instances for each kind of input data

```scala
trait WorkflowGen extends ScalaCheck {

  implicit val arbGermanAcc: Arbitrary[GermanAccount] = Arbitrary {
    for {
      FirstName(firstName) ← arbitray[FirstName]
      LastName(lastName) ← arbitray[LastName]
    } yield GermanAccount(Identity(firstName, lastName, "GER"))
  }

}
```

Working with workflows

Property tests

- Define case classes & arbitrary instances for each kind of input data

```scala
class WorkflowSpec extends Specification with WorkflowGen {

  "workflow" should {

    "Deny German accounts <18y" in prop {
      (acc: GermanAccount, ic: IdentityCheck, v: Under18AgeCheck) =>
        val result = Workflows
          .verification(acc.account)
          .toFSM
          .feedAll(ic.get :: v.get :: Nil)
          .getState

        result.get("result") === Some("fail")
    }

  }

}
```

Working with workflows

Upgrade running transactions to a new version

- Try to parse all events for that transaction using the new version
- Revert to old version on failure
  - This is safe because it doesn't have cause any side effects

```
def upgrade[K, O, M[_], A](oldWorkflow: Plan[K, O, M, A], newWorkflow: Plan[K, O, M, A]) =
  Plan.attempt(newWorkflow).orElse(oldWorkflow)
```

Pieces of the puzzle

Running the workflow engine on kafka

- Serialise to/from JSON
  - Because it can be derived automatically with circe
  - Alternatively, use binary format eg. thrift

```
def encodeJson[F[_], V](implicit e: CirceEncoder[V]): Pipe[F, V, String] =
  pipe.lift({ v: V ⇒ e(v).toString })
```

```
def decodeJson[F[_], V](implicit e: CirceDecoder[V]): Pipe[F, String, Either
[String, V]] =
  pipe.lift({ s: String ⇒ decode[V](s).toEither.left.map(_.toString) })
```

Pieces of the puzzle

Running the workflow engine on kafka

- FS2 kafka stream
  - ConsumerConnector (kafka) gives 1 iterator for each partition
  - Create 1 stream per partition and join with fs2.concurrent.join

```scala
def kafka[F[_], K, V](
  conn: ConsumerConnector,
  topic: String,
  keyDecoder: Decoder[K],
  valueDecoder: Decoder[V],
  nPartitions: Int = 1)(implicit F: Async[F]): Stream[F, Message[Option[K], V]]
```

Pieces of the puzzle

Running the workflow engine on kafka

- FS2 kafka stream
  - Write to another topic when done

```
def kafkaSink[F[_], K, V](producer: Producer[K, V], sync: Boolean = false)
(implicit F: Async[F]): Sink[F, ProducerRecord[K, V]] = { s: Stream[F,
ProducerRecord[K, V]] ⇒
  s.evalMap({ pr: ProducerRecord[K, V] ⇒
    F.delay({
      if (sync) producer.send(pr).get() else producer.send(pr)
      ()
    })
  })
}
```

Pieces of the puzzle

Running the workflow engine on kafka

- FS2 pipeline to…
    1. Fetch from kafka
    2. Deserialise records
    3. Keep track of intermediate state for all running transactions
        1. RocksDB, but RAM would work just as well
    4. Feed new records to the relevant transaction (or initialise if not present)
    5. Check output
        1. If `Some(APIRequest)`, write to API request topic
        2. If `None`, write results to results topic and empty cache

Caveats

- Make sure to not run out of memory
  - Cache everything with RocksDB, no state in RAM
  - Or: Sliding window
- Futures
  - Three different implementations (scalaz, twitter-util, scala)
  - So we need 6 conversions

References

- http://hackage.haskell.org/package/machines-0.6
- http://comonad.com/reader/wp-content/uploads/2009/08/A-Parsing-Trifecta.pdf

Additional material

Caveats

- Make sure to not run out of memory
  - Cache everything with RocksDB, no state in RAM
  - Or: Sliding window
    - Scalaz FingerTree for querying a time series on O(log(n))

```scala
case class TimeSeriesEvent[S](timestamp: DateTime, measure: S)

case class TimeSeriesValue[S](
  from: DateTime,
  to: DateTime,
  count: Int,
  payload: S
)

implicit def semigroupInstance[S](implicit s: Semigroup[S]) =
  new Semigroup[TimeSeriesValue[S]] { ... }


implicit def reducerInstance S](implicit s: Semigroup[S]) =
  UnitReducer[TimeSeriesEvent[S], Option[TimeSeriesValue[S]]]({ t: T ⇒
    Some(TimeSeriesValue(e.timestamp(t), e.timestamp(t), 1, e.measure(t)))
  })
```

```scala
FingerTree[Option[TimeSeriesValue[S]], T]
```

Kafka connector

```scala
def kafka[F[_], K, V](conn: ConsumerConnector, topic: String,
  keyDecoder: Decoder[K], valueDecoder: Decoder[V], nPartitions: Int = 1)
  (implicit F: Async[F]): Stream[F, Message[Option[K], V]] = {

    val streams = conn.createMessageStreams(
      Map(topic -> nPartitions),
      keyDecoder,
      valueDecoder
    )(topic)

    val procs: List[Stream[F, Message[Option[K], V]]] = streams.map { i =>
      Stream.unfoldEval(i.iterator) { k =>
        F.delay {
          k.hasNext.option {
            val next = k.next
            Message(Option(next.key()), next.message()) -> k
          }
        }
      }
    }

    val merged = fs2.concurrent.join(nPartitions)(Stream.emits(procs))

    merged.onFinalize(F.delay({ conn.shutdown() }))
  }
```