Kafkaaa meets FP

What's covered:

- Sharing general experience with kafka for last ~3 months
- How can we reduce the bugs, runtime errors ?
- Importance of test, a few practices followed.
- Sharing a few code snippets that may be interesting.

How often it fails ?

I heard "Computer says no", and it was always late to say no.

A subset of the issues that we faced..

1. Topic operations

Synchronous creation of topics may not work. It's good we check if it exists and if not `wait`!

2. Consumer subscription and assignment

Allows consumers to `subscribe` to a topic and get `assigned` with a predefined set of partitions simultaneously - to form a runtime exception

3. Kafka Admin Client

It seems kafka admin client can block a thread forever!

4. Consumer type

```
typeOf[Consumer unsbuscribed] =:=
  typeOf[Consumer subscribed] =:=
    typeOf[Consumer assigned]
```

Hence we guess the best out of the three states and start polling the topic (that may or may not exist).

5. Serdes

Set the property of serializer to a deserializer class and we get a runtime exception.

Once we fix that, we see `a.serializer` is not an instance of `b.serializer`!

6. Avro serde

```
val client = new CachedSchemaRegistryClient("schema.registry.url")

val serde = new GenericAvroSerde(client)
```

and we see `schema.registry.url` is not set!

Fix: `serde.configure("schema.registry.url", "...")` â šď¸

7. kafka-connect

If the deployed job with the same name for the same source table "existed" ever before, then it chooses an old offset.

8. Any success story ?

Yes, anything with a retry logic.

So, Have I lost time ?

Yes, I have.

I kept learning from mistakes, until I bump into the next one.

While we may not solve all of the above , we could fix some of them and improve the quality to reduce other possible hiccups.

What do we usually lack?

Essentially **abstractions**, **constraints** and **type-driven approach**

- Stay away from kafka and write code only in terms of abstractions. We will see some code soon.
- In fact, this is a general principle that we should try and follow in any programming language - be it Java or Scala.
- Constraints are important and that comes with well typed logic.

And

- If performance is a concern in hot loop, write a referentially transparent unbreakable test!
- A test-case with mutations to test an app with mutations simply negates the purpose of test. Either of them should be true to the world.
- Docker integration tests should work just like other functional tests. No external steps.
- We will see some code soon .

How about Serdes?

Serializing to Avro in Kafka

There are multiple ways to do avro serializion in Kafka.

- Form a `GenericRecord` and use `GenericAvroSerde` explicitly. It works.
- Another way is to generate specific classes that are avro serializable (sbt-avrohugger), and use `KafkaAvroSerializer`
- And there are many other ways.

Well, we must be careful though

- We will end up using `KafkaAvroSerializer` directly on types that `maynot` be serializable.
- We will come to know at runtime.
- Even if we make it run somehow, if the app itself is huge with low level APIs, things get hard to read, maintain, and make-changes!

To add more complexity

Avro serialisation is dependent on few configurations that we must set usng `java.util.Properties`, and we are happy when it works somehow!

A simple standardisation app.

Let's say we are writing a simple abstract KStreams app that does some sort of standardization of data to a defined schema and storing it in Kafka. During the process, we make sure the data is added with Keys, and partitioned as per keys.

You get input as csv, psv, append primary keys into the values, convert values to the schema based on avdl/avsc files and then store in Kafka.

Questions that arise

1. How often do you want to repeat forming a topology for standardising different datasets. DRY principle can be broken but not always.
2. If given a resusable function that forms the stream, how will you ensure users are not passing a wrong set of serializers and deserializers for their generated classes?
3. How do we ensure generated classes are legit?
4. If we expose the generic code that builds the topology, how can we make sure, users of the abstract function are driven by types.

First thing we did!

Serdes are the most important bit, and some of them depends on `KafkaConfig` (a config that has all kafka details).

Instances can discard KafkaConfig if not used

```
trait HasSerde[A] {
  def serde: KafkaConfig => Serde[A]
}
```

Given `HasSerde[A]`, `f:A => B` and `g: B => A`, we can get `HasSerde[B]`.

Was that an `Invariant` functor ?

```
implicit def invariantHasSerde: Invariant[HasSerde] = new Invariant[HasSerde] {
  override def imap[A, B](fa: HasSerde[A])(f: A => B)(g: B => A): HasSerde[B] =
    config =>
      val ser = new Serializer[A] {
        def serialise(topic: String, data: B): Array[Byte] =
          fa.serde(config).serializer.serialize(topic, g(data))
      }

      val deser = new Deserializer[A] {
        override def deserialise(topic: String, data: Array[Byte]) =
          f(fa.serde(config).deserializer().deserialize(topic, data))
      }

      Serde.from(ser, deser)
}
```

Let's begin solution and discover contraints

Let say, given:

- Input key is `K1`, Raw value is `V1`
- Output key is `K2`, Output value is `V2`
- `f:(K1, V1) => Either[NonEmptyList[Error], V2]`, and `g: V2 => K2`

We can build streams topology.

```
def buildStreams[K1, V1, V2] =
 builder.stream(sourceTopic.asString, Consumed.`with`[K1, V1](
   HasSerde[K1].serde(config), HasSerde[V1].serde(config))
 ).map[K2, V2](
    (key, value) => new KeyValue(key, f(key,value).fold(throw ...)(identity)
  ).to(sourceTopic.copy(type=Type.Standardise)).asString,
    Produced.`with`(HasSerde[K2].serde(config),(HasSerde[V2]).serde(config)))
```

What are the constraints we discovered?

i.e, `def buildStreams[K: HasSerde, V : HasSerde, R : HasSerde]`

What are the issue with these constraints ?

```
K : HasSerde
V : HasSerde
R : HasSerde
```

- This says, user has to create `HasSerde` instances of output types *somehow* before calling `buildStreams` and this can go wrong.
- Throwing an exception at mapValue? This is terrible.
- End of the day types don't really tell the story, it isn't type-driven code.

A bit more less poweful constraint exposed?

Why don't we try a less powerful abstraction as a constraint (which is then exposed to the user)

For that, we could rely on a few internals of a library called `avro4s`, and develop something as given below. Explanation later:

```
def fromGenericAvroSerde[B: HasSchema : Encoder : Decoder]: HasSerde[B] = {
  val recordFormat = RecordFormat.apply[B](HasSchema[B].getSchema)

  val genericRecordSerde = new HasSerde[GenericRecord] {
    override def serde: KafkaConfig => Serde[GenericRecord] = config => {
      val schemaRegistryClient = new CachedSchemaRegistryClient(List(config.
      new GenericAvroSerde(schemaRegistryClient)
    }
  }

  genericRecordSerde.imap(recordFormat.from)(recordFormat.to)
}
```

That was a quick jump to some complex-looking code?

Well, all that we did there is:

- We got `HasSerde[B]` from `HasSerde[GenericRecord]` as `HasSerde` is invariant.
- We got `HasSerde[GenericRecord]` using `Encoder` and `Decoder` of `B` (where `Encoder` and `Decoder` are type-classes in avro4s, and we love type classes, don't we?)

Let's define a HasSchema

Now let's change the constraints to `HasSchema` instead of `HasSerde`. `HasSchema` is a less powerful abstraction. We expose only that to the user.

```
trait HasSchema[A] {
  def getSchema: Schema
}

object HasSchema {
  def apply[A](implicit ev:  HasSchema[A]): HasSchema[A] = ev

  def instance[A](schema: Schema): HasSchema[A] = new HasSchema[A] {
    override def getSchema: Schema = schema
  }
}
```

Now buildStreams is having less powerful constraint.

We got rid of `HasSerde` from two places. That's our first win..

```
def buildStreams[K1 : HasSerde, K2 : HasSchema, V1 : HasSerde, V2 : HasSchema](
  f:(K1, V1) => Either[NonEmptyList[Error], V2], g: V2 => K2
) =
 builder.stream(sourceTopic.asString, Consumed.`with`[K1, V1](
   fromGenericAvroSerde[K].serde(config), HasSerde[R].serde(config))
 ).map[K1, V1](
    (key, value) => {

      val newValue = f(key, value)

      new KeyValue(
        g(newValue.fold(throw...)(identity), //Haha throw exception..Good luck
        newValue.fold(throw ...)(identity)
      )
    }
  ).to(
    sourceTopic,
    Produced.`with`(
      fromGenericAvroSerde[K].serde(config),
      fromGenericAvroSerde[V]).serde(config))
    )
```

Optional: We can push this further forward

Did we see a horrible throw exception somewhere? yes we did!
Let's wrap the new key and value into another type whose serialization can fail.
As easy as that!

```
val newValue = f(key, value)
  new KeyValue(
    g(newValue.fold(throw...)(identity), //Haha throw exception..Good luck
    newValue.fold(throw ...)(identity)
  )
```

to

```
// what is this KV? We will see what that is - dont worry!
val newKeyValue = KV.newKeyValue(key)(value)

new KeyValue(
  TransformedKey(newKeyValue.map(_._1)),
  TransformedValue(newKeyValue.map(_._2))
)
```

KV?

`KV` was `KeyValueDerivation` typeclass.

It simply explains the to and fro directions between K1, V1, K2 and V2.

```
trait KeyValueDerivation[K1, V1, K2, V2] {
  def newValue: K1 => V1 => Either[NonEmptyChain[ParseError], V2]
  def newKey: V2 => K2
  def newKeyValue: K1 => V1 => Either[NonEmptyChain[ParseError], (K2, V2)] =
    k1 => v1 => newValue(k1)(v1).map(t => (newKey(t), t))
}
```

You might note that, satisfying this pure interface (oh ! type class) requires to accumulate errors on the left - pretty much forcing you to handle errors when going from `K1,V1` to `V2` and accumulate all of them.

Exposing this abstraction is safe. It might make the users angry but they end up writing only safe code (unless they really, really want to cheat)

Lets's define Serdes for TransfromedKey and Transformed Value

Pushing errors to serialization!

```
final case class Transformed[A](value: Either[NonEmptyChain[ParseError], A])

// Yea..we pushed as hard as possible to push the errors to the very end
// and in Kafka, at the serialization phase..
// In fact, it is documented on how to customise things for errors
// during serialization.
object TransformedValue {
  def impureSerdeFrom[V](serde: HasSerde[A]): HasSerde[TransformedValue[A]] =
    serde.imap[TransformedValue[V]](
    t => TransformedValue(t.asRight))(
      b => b.value.fold(errors => throw errors.asThrowable, identity)
  )
}
```

The final buildStream:

```
// code has skipped a few unnecessary details like properties setting etc
def buildStreams[K1: HasSerde, K2: HasSchema, V1: HasSerde, V2: HasSchema](
  config: KafkaConfig,
  sourceTopic: TopicName,
)(implicit KV: KeyValueDerivation[K1, V1, K2, V2]) =

  builder.stream(
    sourceTopic.asString, Consumed.`with`[K1, V1](
    HasSerde[K1].serde(config), HasSerde[V1].serde(config))
  ).map[Transformed[K2], Transformed[V2]](
    (key, value) => {
       val newKeyValue = KV.newKeyValue(key)(value)
       new KeyValue(
         Transformed(newKeyValue.map(_._1)),
         Transformed(newKeyValue.map(_._2)))
         }
    ).to(
      sourceTopic.copy(type=trans), // remember, no string operations here :)
      Produced.`with`(outputKeySerde, outputValueSerde)
    )
```

What we achieved ?

- Users of buildStreams don't make assumptions, but only implement the constraints - less powerful constraints.
- If compiles, it works with better confidence than passing magic strings of serde classes.
- Better separation of concerns. Users of `buildStreams` need to worry about only `KeyValueDerivation`.
- Type says the story: It reads `K1` and `V1` and converts to `K2`, `V2`.
- If not happy with `avro4s` way of doing things, you can directly use `KafkaAvroStreams` instead of `fromGenericRecord` function with the `HasSchema` constraint in place.
- Obviously, Less number of lines of code.
- Easy to extend, fix things, as compiler tells the story mostly!

Let's talk about test cases - this was a different usecase.

- The low level streams apps makes use of GenericRecord extensively.
- This resulted in Random creation of GenericRecord in test cases.
- The valid GenericRecord is in fact created by mutating a random generic record created using confluent's function.
- In short, we ended up having `GenericRecord` somehow!

A simple test case

- Let's say we have a streaming app that has internal state.
- The state is updated with some value in headers field in the input record.
- The streaming app skips the record that has a state value less than the previous one.
- The streaming app processes the record that has a state value less greater than the previous record.

For reasons, the streaming app is complex and is **dealing with `GenericRecord`**.

Let's see simplest approach.

```
implicit val genGenericRecord: Arbitrary[GenericRecord] =
  for {
    seed <- arbitrary[Long]
    headerRecord =
      new Generator(
        header.value,
        new Random(seed)).generate().asInstanceOf[GenericRecord]
    columnRecord =
      new Generator(
      column.value,
      newRandom(seed)).generate().asInstanceOf[GenericRecord]
  } yield
  new GenericRecordBuilder(schema.value)
    .set("header", headerRecord)
    .set("columns", columnRecord).build()
```

And we make a valid generic-record from an invalid

```
val validGenericRecord =
  for {
    genericRecord <- arbitrary[GenericRecord]
    value <- arbitrary[Long]

    _ = record.get("headers")
     .asInstanceOf[GenericRecord]
     .put("state", value)

    a <- arbitrary[String]
    _ = record.get("columns")
        .asInstanceOf[GenericRecord]
        .put("name", a)

  .....
  .....
} yield genericRecord
```

And we bumped into a never passing test

```
val flow =

for {
 genericRecord <- validGenericRecord

 currentState =
   genericRecord.get("headers").asInstanceOf[GenericRecord].get("state")

 newGenericRecord <-
   genericRecord.get("headers")
   .asInstanceOf[GenericRecord]
   .put("state", currentState + 1)

} yield ()

...streams
.convert((genericRecord, newGenericRecord))
  must_=== (genericRecord, newGenericRecord))
```

This never passes because only `newGenericRecord` exists by the time stream is called.

What is going wrong ?

- We never thought of giving a good start to the problem.
- We straight away dealt with writing statements.
- It is always late if we don't get it right in the beginning - resulting in a totally unreadable code getting over issues that's not quite intuitive for any developer.

How about this approach ?

- To begin with valid generic record and invalid generic record can be two separate types `ValidRecord` and `InvalidRecord`
- Lets create a better `put` method for generic record as a syntax - every mutation results in a new generic record, and accepts only a coproduct of types.
- Let's create a function that accepts `f: Field => Part[GenericRecord] => Part[GenericRecord]`, and mutates `GenericRecord` using the better put methods.

A better put in GenericRecord in tests.

```
type Out = Cop[Int :: String :: Long :: Double :: TNil]

implicit class GenericRecordOps(rec: GenericRecord) {
  def putSafe(key: String, value: GenericRecord \/ Out): GenericRecord = {

    val newRec = new GenericRecordBuilder(rec.asInstanceOf[Record]).build()

    newRec.put(key, value match {
      case \/-(String(ss))    => ss
      case \/-(Int(ss))       => ss
      case \/-(Long(ss))      => ss
      case \/-(Double(ss))    => ss
      case -\/(ss)              => ss
    }) |> (_ => newRec)

}
```

A snippet that deals with mutation of generic record

```
type MutationF[F[_], S, A] = StateT[F, S, A]

def mutateGenericRecord[F[_] : Monad, P[_], R](
  g: Field => P[GenericRecord] => F[P[GenericRecord]],
)(
   implicit
     G: Section[P, GenericRecord],
     Iso: R <=> GenericRecord
  ): MutationF[F, P, Unit] = ???
```

`F` can be `scalacheck.Gen`. This allows the users to be in the context of `Gen` while creating a function that deals with mutating a generic record.

`Section` says, we can fetch `P` from `GenericRecord`, where `P` itself is a generic-record. They are `<=> (isomorphic)` that represents the `asIntanceOf` before.

Now the arbitraries look better:

```
implicit val arbitrary: Arbitrary[InvalidRecord] = ???

// Users need to worry about only this function to create a valid record
val f: Field => Header[GenericRecord] =>Gen[Header[GenericRecord]] =
  field => headerRecord =>
      if (field.name() === "state") {
        Gen.posNum[Long].map(long => (headerRecord.putSafe("state", Long(long)))
      } else headerRecord


val g: Field => Column[GenericRecord] =>Gen[Column[GenericRecord]] = ???

val toValid =
  for {
    _ <- mutateGenericRecord(f)
    _ <- mutateGenericRecord(g)
  } yield ()


def genOfValidRecord: Gen[ValidRecord] =
  Arbitrary[InvalidRecord].flatMap(invalidRecord => toValid.run(invalidRecord))
```

Now the final test case works without much change in code

```
val flow =

for {
 validRecord <- arbitary[ValidRecord]

 currentState =
   Section[Header, ValidRecord].value(validRecord).get("state")

 newGenericRecord <-
   mutateGenericRecord(_ => header =>
     Gen.const(currentState + 1).map(long => header.putSafe(Long(long))
   ) run validRecord

} yield ()

...

streams
  .convert((genericRecord, newGenericRecord))
    must_=== (genericRecord, newGenericRecord))
```

Final tagless for an automation

In IAG, we had to deal with an automation that sticks together various components that with schema registry, kafka-connect, streams, admin clients and so forth. Each component looks like this:

```
type Action[F[_], A] = Kleisli[EitherT[F, SchemaRegistryError, ?], SchemaRegistryCl

trait SchemaRegistryOp[F[_]] {
  def exists(subject: Subject): Action[F, Schema]
  def register(schema: Schema, subject: Subject): Action[F, Unit]
  def registerAndVerify(schema: Schema, subject: Subject)(implicit B: Monad[F]) =
    register(schema, subject) *> exists(subject)
  def registerIfNotExists(schema: Schema, subject: Subject)(implicit F: Monad[F]) =
    exists(subject) or registerAndVerify(schema, subject)
}
```

If interested, we will have a quick glance at them.

Why finally tagless for an automation of kafka ?

- It was easy to reason about the flow.
- Development time of the logic flow was quicker as we hardly dealt with kafka when writing them.
- The presence of effect `F` can easily represent concurrency effect to improve performance by making a few operations unblocking.
  It is so easy to reason about as `IO.async(callBack => doRest and callBack)` instead of `IO.apply(doRest)`.
- The presence of `Either` all over the tagless, made sure we take care of possible errors as much as we could.

PS: More time was spent on simulating the entire flow in docker environment. However we made sure they are written like any other functional tests without much moving parts.

To conclude:

- Nothing actually stops us from striving to write a good piece of software.
- If the underlying APIs are burning our head, it might probably the best situation to use more type-safe code.
- If bad things are done for reasons, make sure we do good things in test package.
- Find abstractions, constraints and form a type driven approach.
- Tagless final wins many times
- Prefer writing it in any language that has `Either` and `F` that can represent concurrency.