Kafkaaa for last ~3 months

What is this talk ?

- Sharing general experience with kafka in IAG for last ~3 months
- How difficult it was to write a type driven code given a kafka platform?
- How can we reduce the bugs, runtime errors ?
- Importance of test, a few practices followed.
- Sharing a few code snippets.

Is kafka a silver bullet ?

I heard "Computer says no" many times by now.

1. Kafka Admin Client

Creating a kafka admin client can block a thread forever!

2. Topic operations

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

3. 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

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. serde

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", "...")`

7. kafka-connect

If the deployed job with the same name for the same source table exists, then it chooses an old offset.

So, Have I lost time ?

Yes, I have.

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

While we can't solving all of the above, we could reduce their probability and other possible hiccups.

What did we really miss?

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 could follow in any programming language - be it Java or Scala.
- Constraints are important. This could be forming a type as early as possible.

And

- If performance is a concern in hot loop, write a referentially transparent unbreakable test!
- A test-case with in-place mutations to test a streaming app with in-place mutation simply negates the purpose of test.
- If running an integration test, then a sbt task for docker-compose up will make functional tests cleaner.
- We will see some code soon .

How about Serdes?

Generic Record 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 ways in fact.

Well, we must be careful though

- We will end up using `KafkaAvroSerializer` directly on types that `maynot` be serializable.
- We will come to know that when job is running!
- Even if we make it running somehow, if the app is huge with low level APIs, things get hard to read/maintain/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 in IAG.

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.
2. If given a resusable function that forms the stream, how will you ensure users are not passing a wrong set of serializer for their generated classes?
3. 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!

```
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, if key is `K`, raw value is `R` and given `f:(K, R) => Either[NonEmptyList[Error], V]`, it seems we can build streams topology.

```
def buildStreams[K, R, V] =
 builder.stream(sourceTopic.asString, Consumed.`with`[K, R](
   HasSerde[K].serde(config), HasSerde[R].serde(config))
 ).map[K, V](
    (key, value) => new KeyValue(key, f(key,value).fold(throw ...)(identity)
  ).to(sourceTopic.copy(type=Type.Standardise)).asString,
    Produced.`with`(HasSerde[K].serde(config),(HasSerde[V]).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 serde instances of output types *somehow* before calling `buildStreams` and this can go wrong.
- Throwing an exception at mapValue?
- Types don't really tell the story. It is a lie!

A bit more less poweful constraint exposed?

Why don't we try a less powerful abstraction as a constraint.

For that, we could rely on a few internals of `avro4s` library:

```
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)
}
```

Let's define a HasSchema

Now let's change the constraints to `HasSchema` instead of `HasSerde`.

```
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 buildStream is having less powerful constraint.

```
def buildStreams[K1 : HasSerde, K : HasSchema, R : HasSerde, V : HasSchema](
  f:(K, R) => Either[NonEmptyList[Error], V], g: K => K1
) =
 builder.stream(sourceTopic.asString, Consumed.`with`[K, R](
   fromGenericAvroSerde[K].serde(config), HasSerde[R].serde(config))
 ).map[K, V](
    (key, value) => new KeyValue(g(key), f(key,value).fold(throw ...)(identity)
  ).to(
    sourceTopic,
    Produced.`with`(
      fromGenericAvroSerde[K].serde(config),
      fromGenericAvroSerde[V]).serde(config))
    )
```

Optional: We can push this further forward

```
// Given a `f:(K, R) => Either[NonEmptyList[Error], V]`
  (key, value) => new KeyValue(key, f(key,value).fold(throw ...)(identity)
```

to

```
  (key, value) => new KeyValue(key, WithPrimaryKey(key, value))
```

WithPrimaryKey?

```
case class WithPrimaryKey[K, R](key: K, value: R)

def impureSerdeFrom[K, R, V](
  serde: HasSerde[V]
)(
  implicit P: ParserFromTo[(K, R), V]
): HasSerde[WithPrimaryKey[K, R]] =
  serde.imap[(K, R)](P.to)(d => P.from(d).fold(
    throw _, identity)
  ).imap({case(a, b) => WithPrimaryKey(a, b)})(t => (t.key, t.value))
}
```

ParseFromTo ?

```
trait ParserFromTo[A, B] {
  def from(c: A): Either[NonEmptyList[ParseError], B]
  def to(b: B): A
}

 def bidirectionalLaw[A: Eq, B](a: A)(implicit ev: ParserFromTo[A, B]): Boolean =
    ParserFromTo[A, B].from(a) match {
      case Right(value) => ParserFromTo[A, B].to(value) === a
      case Left(_) => true
    }

  def identityParserLaw[A : Eq](a: A): Boolean = {
    ParserFromTo[A, A].from(a).fold(_ => false, b => identityParser.to(b) === a)
```

The final buildStream:

```
def buildStreams[K1 : HasSerde, K2 : HasSchema, V1 : HasSerde, V2 : HasSchema](
  f: K1 => K2
)(implicit P: ParseFromTo[(K1, V1), V2]) =
 builder.stream(sourceTopic, Consumed.`with`[K1, V1](
   HasSerde[K1].serde(config), HasSerde[V1].serde(config))
 ).map[K, V](
    (key, value) => new KeyValue(f(key), WithPrimaryKey(key, value)
  ).to(destinationTopic,
    Produced.`with`(
       fromGenericAvroSerde[K2].serde(config),
       WithPrimaryKey.impureSerdeFrom(fromGenericAvroSerde[V2]).serde(config))
    )
```

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 `ParseFromTo`.
- Type says the story: *It reads `K1` and `V1` and converts to `K2`, `V2` given `ParseFromTo[(K1, V1), V2]`, and `HasSchema` of K2 and 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.

Let's talk about test cases !

- The low level streams apps in IAG 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.

An automation App

Automation app dealt with schema registry, kafka-connect, streams, admin clients and so forth. Each component looks like this:

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

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[
    exists(subject) or registerAndVerify(schema, subject)
}
```

What is in conclusion?

- 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
- Controversial: Prefer writing it in any language that has `Either` and `F` that can represent concurrency.
- Above all, let's explain this to client side developers !