# KIP-513: Distinguish between Key and Value serdes in scala wrapper library for kafka streams

## Status

**Current state**: *"Under Discussion"*

**Discussion thread**: *here*

**JIRA**: *here [Change the link from KAFKA-1 to your own ticket]*

## Motivation

In Scala it is a common practice to define/derive all typeclass instances in one place and after that use them in all other places.

Assume I have a `org.apache.kafka.common.serialization.Serde[T]` typeclass and I want to define all instances in some common place. Unfortunately, if you use the confluent schema registry library, there is no easy way to do that. It happens because that library wants you to specify whether or not the given Serde is for a key.
Since the scala kafka-streams library does not have any distinction on the type level between Serdes for keys and Serdes for values, I had to create a workaround.
The idea behind it is very simple: If we cannot introduce a type level distinction on Serdes, lets introduce that distinction on the model level and try to do that as lightweight as possible.

**Workaround**

```
import org.apache.kafka.common.serialization.Serde
import org.apache.kafka.streams.scala.StreamsBuilder
import org.apache.kafka.streams.scala.kstream.KStream
import org.apache.kafka.streams.scala.ImplicitConversions._
import io.confluent.kafka.streams.serdes.avro.GenericAvroSerde

import scala.collection.JavaConverters._

object Workaround {
  trait IsKey
  type Key[T] = T with IsKey

  implicit class RichT[T](value: T) {
    def asKey: Key[T] = value.asInstanceOf[Key[T]]
  }
}

object SerdesSupport {
  import Workaround.Key

  // I don't want to go too deep into implementation details,
  // so lets define some dummy type class called AlmostSerde and assume that,
  // if I have an instance of that type class for type T and GenericAvroSerde, I can create instance of Serde
for T
  class AlmostSerde[T]()
  implicit def deriveAlmostSerde[T]: AlmostSerde[T] = ??? // assume that we can derive AlmostSerde instances
for all T
  private def createSerde[T](as: AlmostSerde[T], gas: GenericAvroSerde): Serde[T] = ???

  implicit def specificSerdeForKey[T >: Null](implicit as: AlmostSerde[T]): Serde[Key[T]] = {
    val genericSerde = new GenericAvroSerde()
    val configuration = Map[String, Any]().asJava
    val isSerdeForKey = true
    genericSerde.configure(configuration, isSerdeForKey)
    createSerde(as, genericSerde)
  }

  implicit def specificSerde[T >: Null](implicit as: AlmostSerde[T]): Serde[T] = {
    val genericSerde = new GenericAvroSerde()
    val configuration = Map[String, Any]().asJava
    val isSerdeForKey = false
    genericSerde.configure(configuration, isSerdeForKey)
    createSerde(as, genericSerde)
  }

}

object TestApp extends App {
  case class A(x: Int)
  case class B(y: String)

  import Workaround._
  import SerdesSupport._

  val builder = new StreamsBuilder()

  val streamAtoB: KStream[Key[A], B] = builder.stream("topic")
  val streamBtoA: KStream[Key[B], A] = streamAtoB.map((a, b) => b.asKey -> a)

}
```

Such solution will do the job, but it requires some boilerplate and makes the whole logic of a program a bit unclear.

# Public Interfaces

I want to use different Serdes for keys and for values. This means that I will have to change more or less all the public function signatures of the scala wrapper library, where we take Serde as a parameter.

# Proposed Changes

I think it will be enough to do the same trick, that I did for 'labeling' models, but for Serdes.

**Solution**

```
import org.apache.kafka.common.serialization.Serde
import org.apache.kafka.streams.scala.StreamsBuilder
import org.apache.kafka.streams.scala.kstream.{Consumed, KStream}
import io.confluent.kafka.streams.serdes.avro.GenericAvroSerde

import scala.collection.JavaConverters._

object Solution {

  trait IsKeySerde
  type KeySerde[T] = Serde[T] with IsKeySerde

  trait IsValueSerde
  type ValueSerde[T] = Serde[T] with IsValueSerde

  //example of the changed implicit conversion from the org.apache.kafka.streams.scala.ImplicitConversions
  implicit def consumedFromSerdeNew[K, V](implicit keySerde: KeySerde[K], valueSerde: ValueSerde[V]): Consumed
[K, V] =
    Consumed.`with`[K, V]

}

object OldBehavior {
  import Solution._
  implicit def asKeySerde[T](serde: Serde[T]): KeySerde[T] = serde.asInstanceOf[KeySerde[T]]
  implicit def asValueSerde[T](serde: Serde[T]): ValueSerde[T] = serde.asInstanceOf[ValueSerde[T]]
}

object SerdesSupport {
  import Solution._
  import OldBehavior._

  // I don't want to go too deep into implementation details,
  // so lets define some dummy type class called AlmostSerde and assume that,
  // if I have an instance of that type class for type T and GenericAvroSerde, I can create instance of Serde
for T
  class AlmostSerde[T]()
  implicit def deriveAlmostSerde[T]: AlmostSerde[T] = ??? // assume that we can derive AlmostSerde instances
for all T
  private def createSerde[T](as: AlmostSerde[T], gas: GenericAvroSerde): Serde[T] = ???

  implicit def specificSerdeForKey[T >: Null](implicit as: AlmostSerde[T]): KeySerde[T] = {
    val genericSerde = new GenericAvroSerde()
    val configuration = Map[String, Any]().asJava
    val isSerdeForKey = true
    genericSerde.configure(configuration, isSerdeForKey)
    createSerde(as, genericSerde)
  }

  implicit def specificSerdeForValue[T >: Null](implicit as: AlmostSerde[T]): ValueSerde[T] = {
    val genericSerde = new GenericAvroSerde()
    val configuration = Map[String, Any]().asJava
    val isSerdeForKey = false
    genericSerde.configure(configuration, isSerdeForKey)
    createSerde(as, genericSerde)
  }
```

```
}

object TestApp extends App {
  case class A(x: Int)
  case class B(y: String)

  import Solution._
  import SerdesSupport._

  val builder = new StreamsBuilder()

  val streamAtoB: KStream[A, B] = builder.stream("topic")
  val streamBtoA: KStream[B, A] = streamAtoB.map((a, b) => b -> a)
}
```

## Compatibility, Deprecation, and Migration Plan

In theory we do not need any changes from the library's users side. We just need to put implicit rules from the OldBehavior object into the correct place, such that these rules will have the lowest priority while the implicit resolution process.

## Rejected Alternatives

Honestly speaking, I have nothing to write here. I am open for your suggestions