KIP-270 - A Scala Wrapper Library for Kafka Streams

Status

Current state: *"Under Discussion"*

Discussion thread: *here [Change the link from the KIP proposal email archive to your own email thread]*

JIRA: *here*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

This document outlines a proposal for developing a Scala library as a wrapper over the existing Java APIs for Kafka Streams.

Kafka Streams currently offers Java APIs based on the Builder design pattern, which allows users to incrementally build the target functionality using lower level compositional fluent APIs. The problems of using these APIs from a Scala code are 2 fold:

1. **Additional type annotations -** The Java APIs use Java generics in a way that are not fully compatible with the type inferencer of the Scala compiler. Hence the user has to add type annotations to the Scala code, which seems rather non-idiomatic in Scala.
2. **Verbosity -** In some cases the Java APIs appear too verbose compared to idiomatic Scala.
3. **Type Unsafety -** The Java APIs offer some options where the compile time type safety is sometimes subverted and can result in runtime errors. This stems from the fact that the serdes defined as part of config are not type checked during compile time. Hence any missing serdes can result in runtime errors.

The current work on the Scala wrapper is inspired from the works of Alexis Seigneurin on his Github Repository.

Proposed Changes

Summary

The suggested Scala library is a wrapper over the existing Java APIs for Kafka Streams DSL and addresses the above 3 concerns. It does not attempt to provide idiomatic Scala APIs that one would implement in a Scala library developed from scratch. The intention is to make the Java APIs more usable in Scala through better type inferencing, enhanced expressiveness, and lesser boilerplates.

The library wraps Java Stream DSL APIs in Scala thereby providing:

1. Better type inference in Scala
2. Less boilerplate in application code
3. The usual builder-style composition that developers get with the original Java API
4. Implicit serializers and de-serializers leading to better abstraction and less verbosity
5. Better type safety during compile time

The above points result in an overall improved productivity for development.

This document introduces the Kafka Streams Scala library.

In addition, we received a proposal for an alternate implementation of the same functionality using the type class based approach in Scala. This is the PR currently open in our repository and is based on a fork of our implementation. There has been lots of discussions on the pros and cons of both the approaches.

Dependencies

kafka-streams-scala only depends on the Scala standard library and Kafka Streams.

Sample Usage

The library works by wrapping the original Java abstractions of Kafka Streams within a Scala wrapper object and then using implicit conversions between them. All the Scala abstractions are named identically as the corresponding Java abstraction but they reside in a different package of the library e.g. the Scala class `org.apache.kafka.streams.scala.StreamsBuilder` is a wrapper around `org.apache.kafka.streams.StreamBuilder`, `org.apache.kafka.streams.scala.kstream.KStream` is a wrapper around `org.apache.kafka.streams.kstream.KStream`.

Here's an example of the classic *Word Count* program that uses the Scala builder `StreamBuilder` and then builds an instance of `KStream` using the wrapped API `builder.stream`. Then we reify to a table and get a `KTable`, which, again is a wrapper around Java `KTable`.

The net result is that the following code is structured just like using the Java API, but from Scala and with far fewer type annotations compared to using the Java API directly from Scala. The difference in type annotation usage will be more obvious when we use a more complicated example. The library comes with a test suite of a few examples that demonstrate these capabilities.

Word Count

```
package org.apache.kafka.streams.scala

import org.apache.kafka.streams.scala.kstream._
// brings in scope all necessary implicit serdes
import DefaultSerdes._

val builder = new StreamsBuilder
val textLines = builder.stream[String, String](inputTopic)

val pattern = Pattern.compile("\\W+", Pattern.UNICODE_CHARACTER_CLASS)

val wordCounts: KTable[String, Long] =

 textLines.flatMapValues(v => pattern.split(v.toLowerCase))
   .groupBy((k, v) => v)
   .count()
wordCounts.toStream.to(outputTopic)

val streams = new KafkaStreams(builder.build, streamsConfiguration)

streams.start()
```

In the above code snippet, we don't have to provide any serdes, `Serialized`, `Produced`, `Consumed` or `Joined` explicitly. They will also *not be dependent on any serdes specified in the config* - in fact all serdes specified in the config will be ignored by the Scala APIs. All serdes and `Serialized`, `Produced`, `Consumed` or `Joined` will be handled through *implicit serdes* as discussed later in the document. The complete independence from configuration based serdes is what makes this library completely type-safe - any missing instances of serdes, `Serialized`, `Produced`, `Consumed` or `Joined` will be flagged as a compile time error.

Type Inference and Composition

Here's a sample code fragment using the Scala wrapper library. Compare this example to the Scala code for the same example using the Java API directly in Confluent's repository.

Better type inference

```
// Compute the total per region by summing the individual click counts per region.
// KTable is the Scala abstraction wrapping the Java instance
val clicksPerRegion: KTable[String, Long] =
 userClicksStream
   // Join the stream against the table
   .leftJoin(userRegionsTable,
     (clicks: Long, region: String) =>
       (if (region == null) "UNKNOWN" else region, clicks))

   // Change the stream from <user> -> <region, clicks> to <region> -> <clicks>
   .map((_, regionWithClicks) => regionWithClicks)

   // Compute the total per region by summing the individual click counts per region.
   .groupByKey
   .reduce(_ + _)
```

Implicit Serdes

One of the common complaints of Scala users with the Java API has been the repetitive usage of the serdes in API invocations. Many of the APIs need to take the serdes through abstractions like `Serialized`, `Consumed`, `Produced` or `Joined`. And the user has to supply them every time through the `with` function of these classes.

The library uses the power of *Scala implicits* to alleviate this concern. As a user you can provide implicit serdes or implicit values of `Serialized`, `Joined`, `Consumed` or `Produced` once and make your code less verbose. In fact you can just have the implicit serdes in scope and the library will make the instances of `Serialized`, `Produced`, `Consumed` or `Joined` available in scope.

The library also bundles all implicit serdes of the commonly used primitive types in a Scala module - so just import the module vals and have all serdes in scope. Similar strategy of modular implicits can be sdopted for any user-defined serdes as well.

Here's an example:

Implicit Serdes

```
// DefaultSerdes brings into scope implicit serdes (mostly for primitives)
// that will set up all Serialized, Produced, Consumed and Joined instances.
// So all APIs below that accept Serialized, Produced, Consumed or Joined will
// get these instances automatically
import DefaultSerdes._

val builder = new StreamsBuilder()

val userClicksStream: KStream[String, Long] = builder.stream(userClicksTopic)

val userRegionsTable: KTable[String, String] = builder.table(userRegionsTopic)

// The following code fragment does not have a single instance of Serialized,
// Produced, Consumed or Joined supplied explicitly.
// All of them are taken care of by the implicit serdes imported by DefaultSerdes
val clicksPerRegion: KTable[String, Long] =
  userClicksStream
    .leftJoin(userRegionsTable,
      (clicks: Long, region: String) =>
        (if (region == null) "UNKNOWN" else region, clicks))
    .map((_, regionWithClicks) => regionWithClicks)
    .groupByKey
    .reduce(_ + _)

clicksPerRegion.toStream.to(outputTopic)
```

Quite a few things are going on in the above code snippet that may warrant a few lines of elaboration:

1. The code snippet does not depend on any config defined serdes. In fact any serde defined as part of the config will be ignored
2. All serdes are picked up from the implicits in scope. And `import DefaultSerdes._` brings all necessary serdes in scope.
3. This is an example of compile time type safety that we don't have in the Java APIs
4. The code looks less verbose and more focused towards the actual transformation that it does on the data stream

User-defined Serdes

When the default primitive serdes are not enough and we need to define custom serdes, the usage is exactly the same as above. Just define the implicit serdes and start building the stream transformation. Here's an example with `AvroSerde`:

User defined Serde

```
// domain object as a case class
case class UserClicks(clicks: Long)

// An implicit Serde implementation for the values we want to
// serialize as avro
implicit val userClicksSerde: Serde[UserClicks] = new AvroSerde

// Primitive serdes
import DefaultSerdes._

// And then business as usual ..

val userClicksStream: KStream[String, UserClicks] = builder.stream(userClicksTopic)

val userRegionsTable: KTable[String, String] = builder.table(userRegionsTopic)

// Compute the total per region by summing the individual click counts per region.
val clicksPerRegion: KTable[String, Long] =
 userClicksStream

   // Join the stream against the table.
   .leftJoin(userRegionsTable, (clicks: UserClicks, region: String) => (if (region == null) "UNKNOWN" else
region, clicks.clicks))

   // Change the stream from <user> -> <region, clicks> to <region> -> <clicks>
   .map((_, regionWithClicks) => regionWithClicks)

   // Compute the total per region by summing the individual click counts per region.
   .groupByKey
   .reduce(_ + _)

 // Write the (continuously updating) results to the output topic.
 clicksPerRegion.toStream.to(outputTopic)
```

A complete example of user-defined serdes can be found in a test class within the library.

Scala Version Compatibility

Binary Compatibility

When two versions of Scala are binary compatible, it is safe to compile your project on one Scala version and link against another Scala version at run time

(http://docs.scala-lang.org/overviews/core/binary-compatibility-of-scala-releases.html).

Binary compatibility is a common concern for Scala library authors. Scala releases are always backward and forward binary compatible between minor releases since Scala 2.10.x. This is automatically enforced by use of the Scala Binary Compatibility validation tool (MiMa). However binary compatibility is typically broken across major releases.

Scala major versions 2.11 and 2.12 are not binary compatible due to compiler changes that use several new language features made available in Java 8. Scala 2.13 has not been released yet, but it's anticipated to be binary incompatible with 2.12. The Scala 2.13 release has a central theme of core library changes which will cause incompatibility across libraries compiled using earlier versions of Scala.

If there's a desire MiMa could be used as part of the build and release process to manage binary compatibility for kafka-streams-scala releases inline with Apache Kafka's version policy.

Source Compatibility

Two library versions are Source Compatible with each other if switching one for the other does not incur any compile errors or unintended behavioral changes (semantic errors)

(http://docs.scala-lang.org/overviews/core/binary-compatibility-for-library-authors.html#source-compatibility).

To support multiple major versions of Scala it is necessary to cross build a source compatible project with two or more versions of Scala. This is commonly done between major versions of Scala such as 2.10/2.11 and 2.11/2.12.

Due to fundamental core library changes that will be released in 2.13 (such as the collections redesign effort), it's anticipated source compatibility will be an issue due to the ubiquitous use of collections libraries. It's anticipated that Lightbend will release a compatibility library that allows the library author to preserve source compatibility so that managing multiple code branches won't be necessary. Guides from Lightbend will also be made available to make managing this transition as easy as possible for library authors.

Further Reading

- Binary Compatibility for Library Authors
  http://docs.scala-lang.org/overviews/core/binary-compatibility-for-library-authors.html
- Scala Binary Compatibility validation tool (MiMa)
  https://github.com/lightbend/migration-manager
- Scala 2.13 Roadmap
  https://www.scala-lang.org/news/roadmap-2.13.html

New or Changed Public Interfaces

The Scala abstractions available to the user are the following:

- `KStream`
- `KTable`
- `KGroupedStream`
- `KGroupedTable`
- `StreamsBuilder`
- `SessionWindowedKStream`
- `TimeWindowedKStream`

All of the above abstractions wrap the corresponding Java ones having the same names. However the Scala abstractions reside in a separate package `org.apache.kafka.streams.scala.kstream`. Besides the above ones, the library also has several utility abstractions and modules that the user needs to use for proper semantics. These are:

- `org.apache.kafka.streams.scala.ImplicitConversions:` Module that brings into scope the implicit conversions between the Scala and Java classes
- `org.apache.kafka.streams.scala.DefaultSerdes:` Module that brings into scope the implicit values of all primitive serdes
- `org.apache.kafka.streams.scala.ScalaSerde:` Base abstraction that can be used to implement custom serdes in a type safe way

Migration Plan and Compatibility

N/A

Current Status

The original version of Kafka Streams Scala library is available as an open source project from Lightbend on Github. The current available version is 0.2.0 and has been updated to adopt to the above package structure for `org.apache.kafka`. A PR on Apache Kafka is available. The PR contains the following:

- the library implementation
- changes in `build.gradle` to build the library jar
- tests (1 basic test for WordCount, 2 tests demonstrating usage of implicit serdes)

Rejected Alternatives

None.