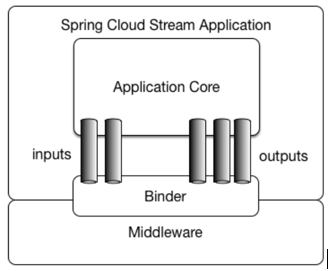
**Using Spring Cloud Streams for Apache Kafka**

Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in Kafka clusters. It combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology.

Spring Cloud Stream is a framework for building message-driven microservice applications. The application communicates with the outside world through input and output channels injected into it by Spring Cloud Stream. Channels are connected to external brokers through middleware-specific Binder implementations.



Spring Cloud Stream Kafka binder provides implementation of Spring Cloud Stream binding API for Apache Kafka, allowing transparent connectivity of Spring Cloud Stream microservices to Apache Kafka destinations (e.g., topics).

This post will discuss how to use Spring Cloud Streams to write a Kafka Streams application.

We will use a sample service that computes the top trending social media post. Results will be continuously updated and can be queried interactively using REST API.

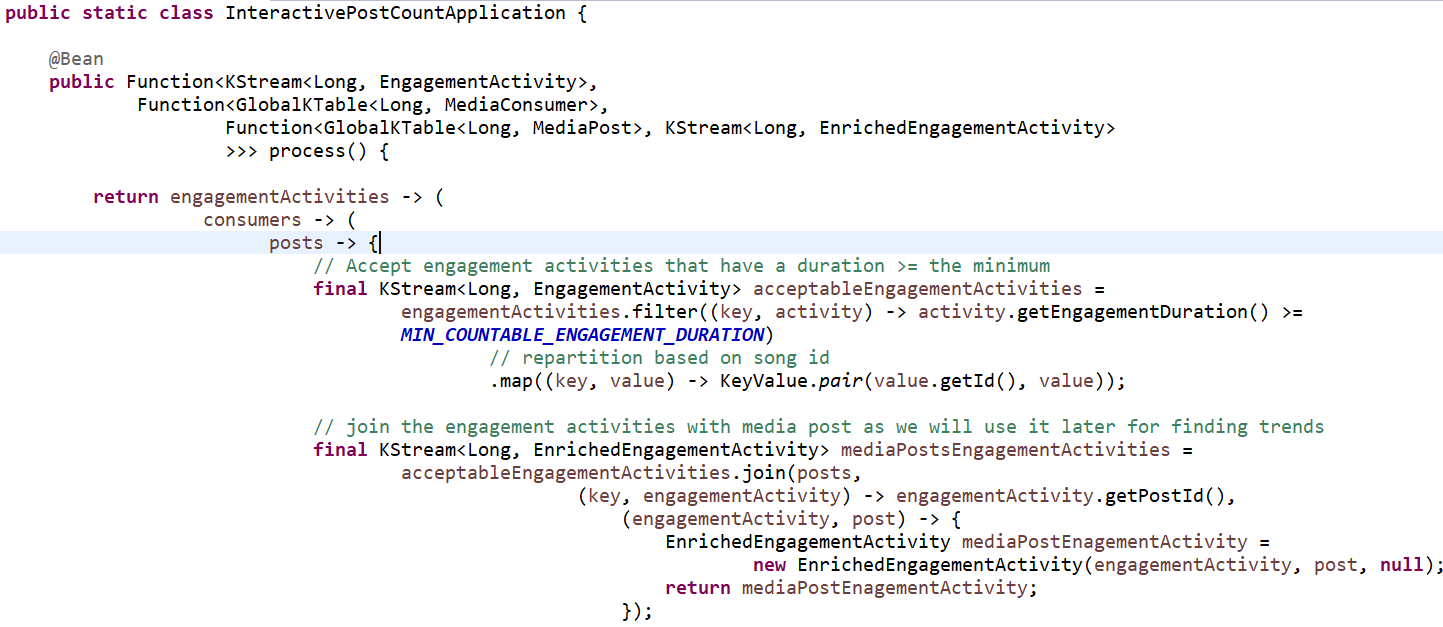
Create a Spring Starter Project with Cloud Stream and Spring for Apache Kafka dependencies.



We need an event generator to generate following events: -

1. Social Media Post creation.
2. Social Media Consumer profile creation.
3. Consumer’s engagement activity for media posts

Now, let’s look at our streams application code: -



* we provide a java.util.function.Function bean where we encapsulate our application’s logic through a lambda expression.
* The Process takes three inputs. Therefore, we start with a Function, but then, on the outbound of this first function, we provide another Function or Consumer until we exhaust our inputs.
* Our first input is the Kstream of Engagement activities. Our second input is a GlobalKTable of Media Consumers. And our third input is a GlobalKTable of Media posts.
* Behind the scenes, the Kafka Streams binder for Spring Cloud Stream will take care of creating the Kafka Streams topology, connecting to a Kafka Cluster, binding to a topic and consuming data from that Kafka topic, which is bound as **KStream**, and **GlobalKTables** in this case.
* Binder provides default values to all the applicable configuration properties for building Kafka Streams. For example: -
  + **Cluster information**: *localhost:9092*. If that is not the case, you can override that by using configuration properties available through Spring Cloud Stream.
  + **Application ID**: By default, the binder will generate an application ID and assign it to the processor. It uses the function bean name as a prefix. For example, if we have a consumer as above, the binder will generate the application ID as process-applicationId.

User provided values override the default values.

* + **Topics to consume from**: For the above processor, we can provide the topic to consume, as follows: -

spring.cloud.stream.bindings.process-in-0: destination: engagement-activity-topic

spring.cloud.stream.bindings.process-in-1: destination: consumer-topic

spring.cloud.stream.bindings.process-in-2: destination: post-topic

we are saying that, for the function bean (process) and its first input (in-0), it shall be bound to a Kafka topic named *engagement-activity-topic*. If you don’t provide an explicit destination like this, the binder assumes that you are using a topic that is the same as the binding name (*process-in-0*, in this case).

* + Output topic: The output topic can be configured as below:

spring.cloud.stream.bindings.process-out-0: destination: enriched-engagement-activity-topic

* **Serialization and Deserialization (Serdes)**: Kafka Streams uses a Serde class to serialize/deserialize records. It is a wrapper around a deserializer on the inbound and a serializer on the outbound. Normally, we need to tell Kafka Streams what Serde to use for each consumer. Binder, however, infers this information by using the parametric types provided as part of Kafka Streams.

The types that are inferred by the binder are those for which Kafka Streams provides out of the box Serde implementations. These are those types:

Integer

Long

Short

Double

Float

Byte[]

UUID

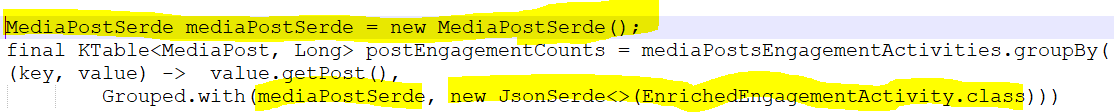
String

If your KStream, KTable, or GlobalKTable have these as the types for the key and the value, you don’t need to provide any special Serde configuration.

If the types are not from one of these, you can provide a bean of type Serde<T>, and, if the generic type T matches with the actual type, the binder will delegate that as the Serde.

If the binder cannot match any Serde, it will fall back to the JsonSerde implementation provided by Spring for Apache Kafka project. You must make sure that the classes are JSON-friendly.

Serde used inside the actual business logic: Kafka Streams has several API methods that need access to Serde objects. For example, look at the method calls groupBy in the Function example processor.



Any Serde required by your business logic implementation needs to be provided by the application.

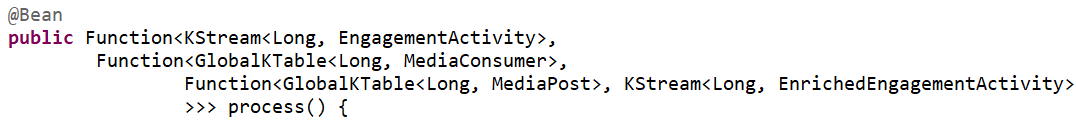
**Named State Stores**

When you have the need to maintain state in the application, Kafka Streams lets you materialize that state information into a named state store. There are several operations in Kafka Streams that require it to keep track of the state such as count, aggregate, reduce, various windowing operations, and others.

**Consuming data as KTable or GlobalKTable**

Kafka Streams binder-based applications can bind to destinations as KTable or GlobalKTable. GlobalKTable is a special table type, where you get data from all partitions of an input topic, regardless of the instance that it is running. By contrast, a KTable gives you only data from the respective partitions of the topic that the instance is consuming from.

Look at the signature of our Function process:

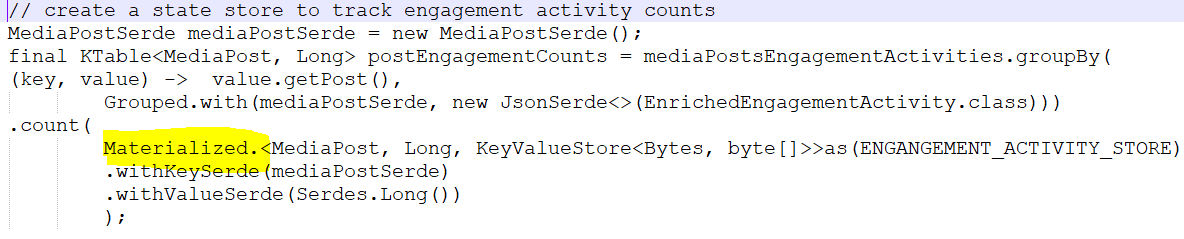


This function has three input bindings, one KStream, two GlobalKTable. Kafka Streams lets you materialize tables consumed like these into named state stores, given that these tables are based on a primary key. You can use the binding level property to materialize them into named state stores along with consumption. The following examples show how to do so:



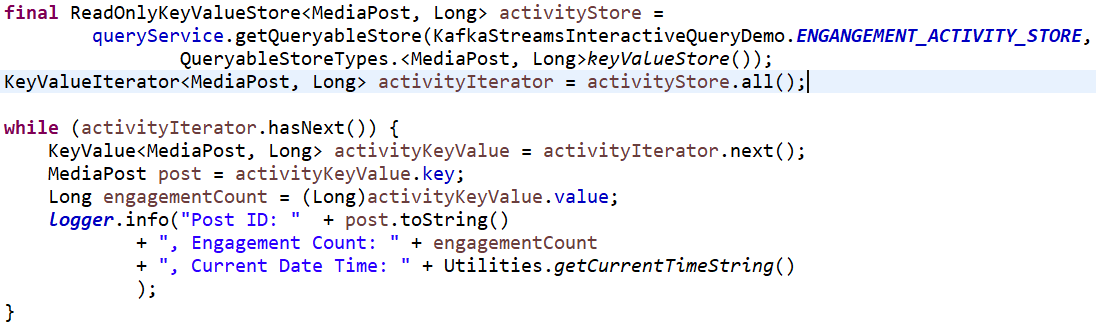
**Kafka Streams DSL operations materialized into state stores**

There are various methods in the Kafka Streams high-level DSL, which returns table types such as count, aggregate, and reduce. There are other operations that use state stores to keep track of information. For example, the various join method calls in KStream, although they return a KStream type, internally use state stores to keep the joined data. In summary, when Kafka Streams lets you materialize data either as a table or stream, it is materialized into a state store, much like data stored in a database table.



**Interactive queries to query data from state stores**

Kafka Streams lets you interactively query the data in the state store in real time as live stream processing is going on. The binder provides abstractions around this feature to make it easier to work with interactive queries. InteractiveQueryService is a basic API that the binder provides to work with state store querying.



If there are multiple instances of the kafka streams application running, then before we can query them interactively, we need to identify which application instance hosts the key. InteractiveQueryService API provides methods for identifying the host information.

In order for this to work, we must configure the property application.server as below:





Source Code