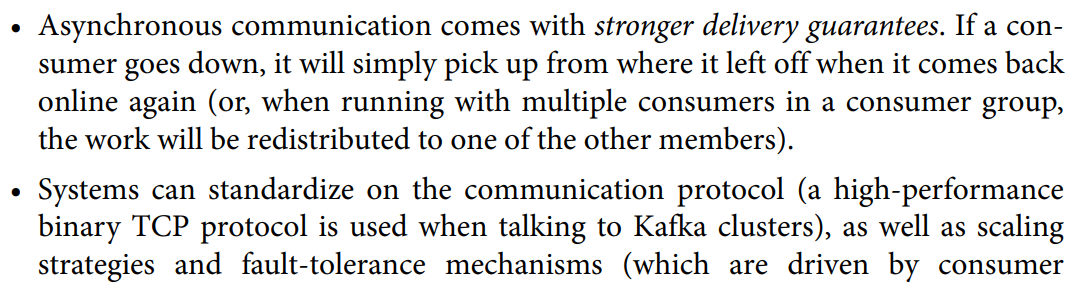
**KakfaStreamExamples:**

[**https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb**](https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb)

1. **Chapter1:**

**In Point to point communication:**

Communication is not replayable. This makes it difficult to reconstruct the state of a system





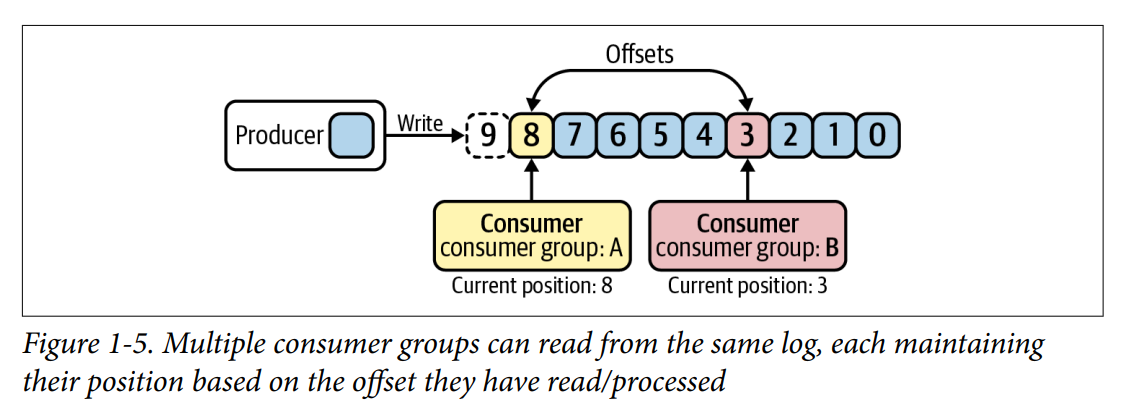
One important difference between the pub/sub model and the client-server model is that communication is not bidirectional in Kafka’s pub/sub model. In other words, streams flow one way. If a system produces some data to a Kafka topic, and relies on another system to do something with the data (i.e., enrich or transform it), the enriched data will need to be written to another topic and subsequently consumed by the original process

Logs are append-only data structures that capture an ordered sequence of events. Let’s examine the italicized attributes in more detail, and build some intuition around logs, by creating a simple log from the command line

1. Logs are Immutable be it add or update.

2. Logs are always by order; not by timestamp BUT the that a record’s position in the log is fixed, and never changes.

Furthermore, while the position of each log entry in the preceding example uses line numbers, Kafka refers to the position of each entry in its distributed log as an offset. Offsets start at 0 and they enable an important behavior: they allow multiple con‐ sumer groups to each read from the same log, and maintain their own positions in the log/stream they are reading from



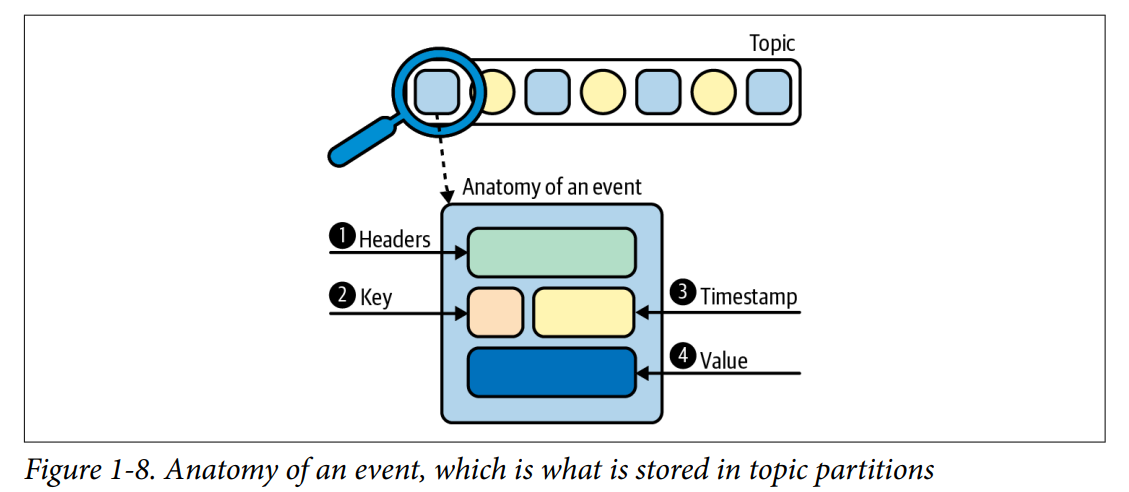
**We have also learned that append-only commit logs are used to model streams in Kafka’s storage layer**

Partitions are individual logs (i.e., the data structures we discussed in the previous section) where data is produced and consumed from. Since the commit log abstrac‐ tion is implemented at the partition level, this is the level at which ordering is guaran‐ teed, with each partition having its own set of offsets. Global ordering is not supported at the topic level, which is why producers often route related records to the same partition.

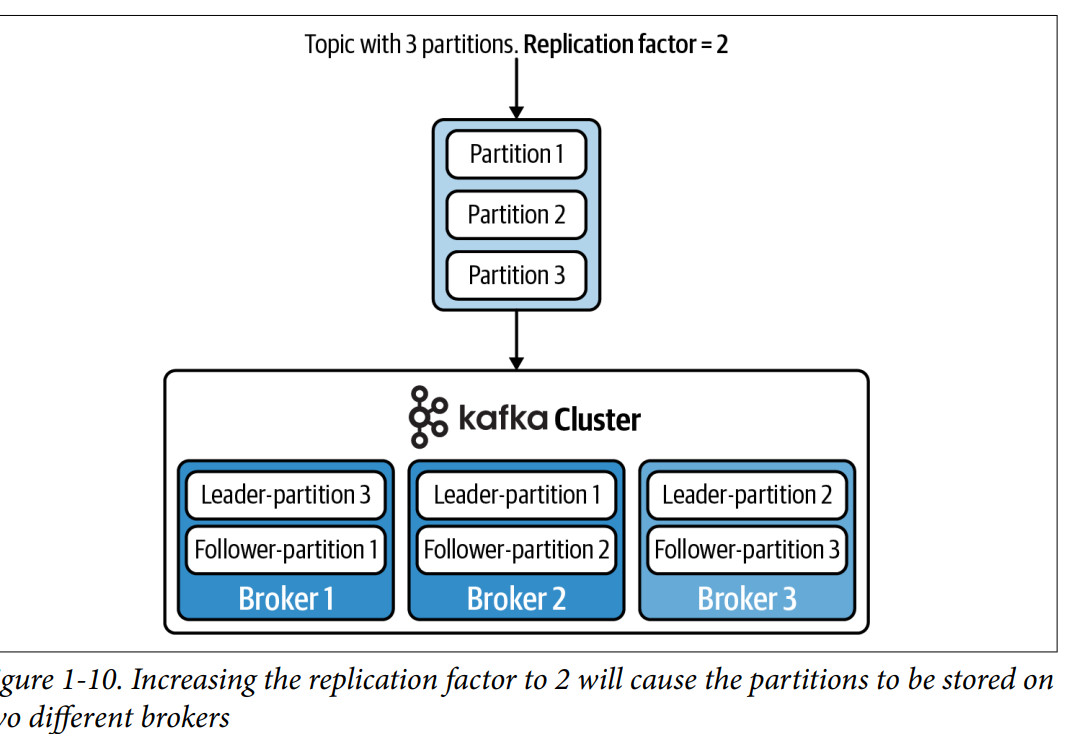
How to make partitioning configurable in kkfa???

An event is a timestamped key-value pair that records something that happened.

An event is stored on topic parttion:

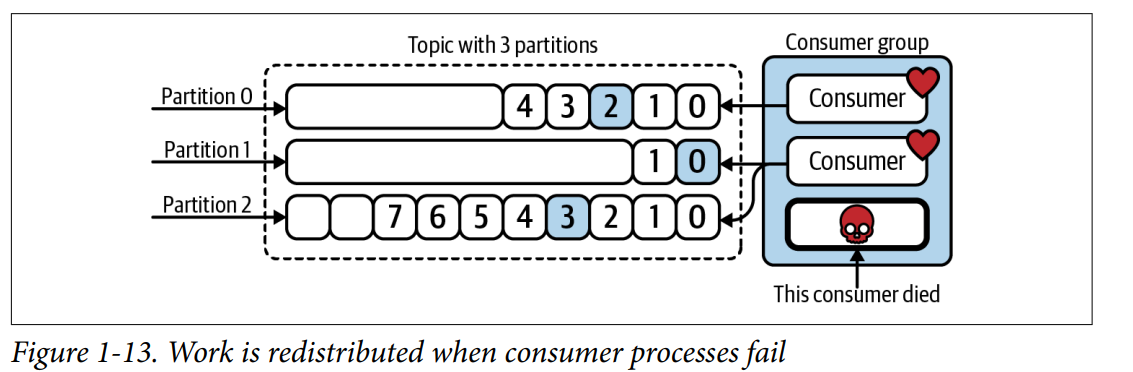


. To achieve fault tolerance and high availability, you can set a replication factor when configuring the topic

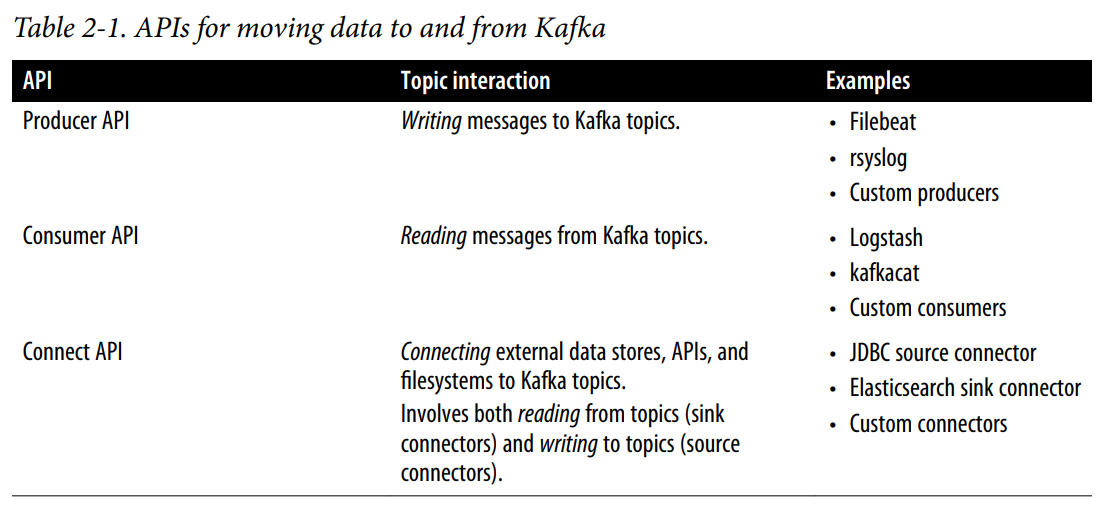


Consumer Group:

For High ThroughPut; Kafka is optimized for high throughput and low latency. To take advantage of this on the consumer side, we need to be able to parallelize work across multiple processes. This is accomplished with consumer groups.



**CHAPTER 2:**



**==**

Kafka Streams is dedicated to helping you process real-time data streams, not just move data to and from Kafka.6 It makes it easy to consume real-time streams of events as they move through our data pipeline, apply data transformation logic using a rich set of **stream processing operators** / as example to show project and primitives, and optionally write new representations of the data back to Kafka

A high-level DSL that looks and feels like Java’s streaming API. The DSL provides a fluent and functional approach to processing data streams that is easy to learn and use. • A low-level Processor API that gives developers fine-grained control when they need it. • **Convenient abstractions for modeling data as either streams or tables. • The ability to join streams and tables, which is useful for data transformation and enrichment. • Operators and utilities for building both stateless and stateful stream processing applications. • Support for time-based operations, including windowing and periodic functions.**

**Make examples to all.**

Scalability:

1. Consumer group
2. d since topics can be expanded by adding more partitions, the amount of work a Kafka Streams application can undertake can be scaled by increasing the number of partitions on the source topics.

Compare to other systems:

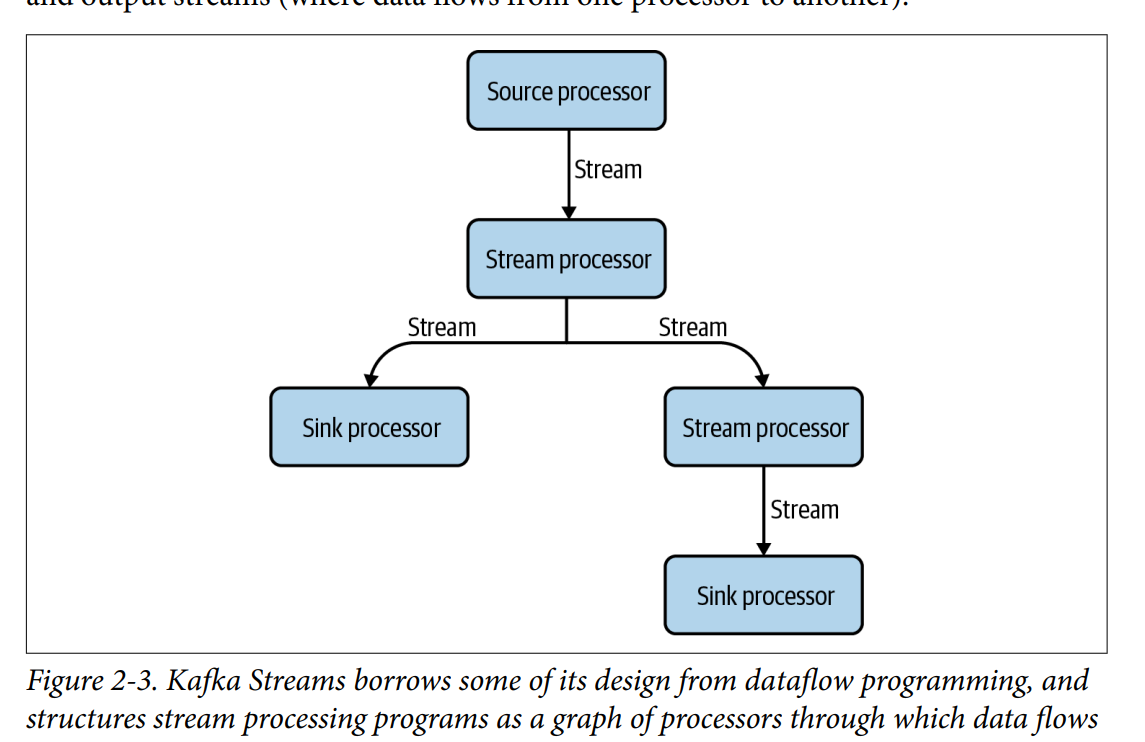
**Deployment Model** Kafka Streams takes a different approach to stream processing than technologies like Apache Flink and Apache Spark Streaming. The latter systems require you to set up a dedicated processing cluster for submitting and running your stream processing pro‐ gram

**Processing Model** Another key differentiator between Kafka Streams and systems like Apache Spark Streaming or Trident is that Kafka Streams implements event-at-a-time processing, so events are processed immediately, one at a time, as they come in. This is considered true streaming and provides lower latency than the alternative approach, which is called micro-batching.

True streaming because; no microbtching; thus low latency

Examples:

Chat infrastructure (Slack), chat bots, virtual assistants • Machine learning pipelines (Twitter) and platforms (Kafka Graphs)



There are three basic kinds of processors in Kafka Streams:

Source processors

Sources are where information flows into the Kafka Streams application. Data is

read from a Kafka topic and sent to one or more stream processors.

Stream processors

These processors are responsible for applying data processing/transformation

logic on the input stream. In the high-level DSL, these processors are defined

using a set of built-in operators that are exposed by the Kafka Streams library,

which we will be going over in detail in the following chapters. Some example

operators are filter, map, flatMap, and join.

Sink processors

Sinks are where enriched, transformed, filtered, or otherwise processed records

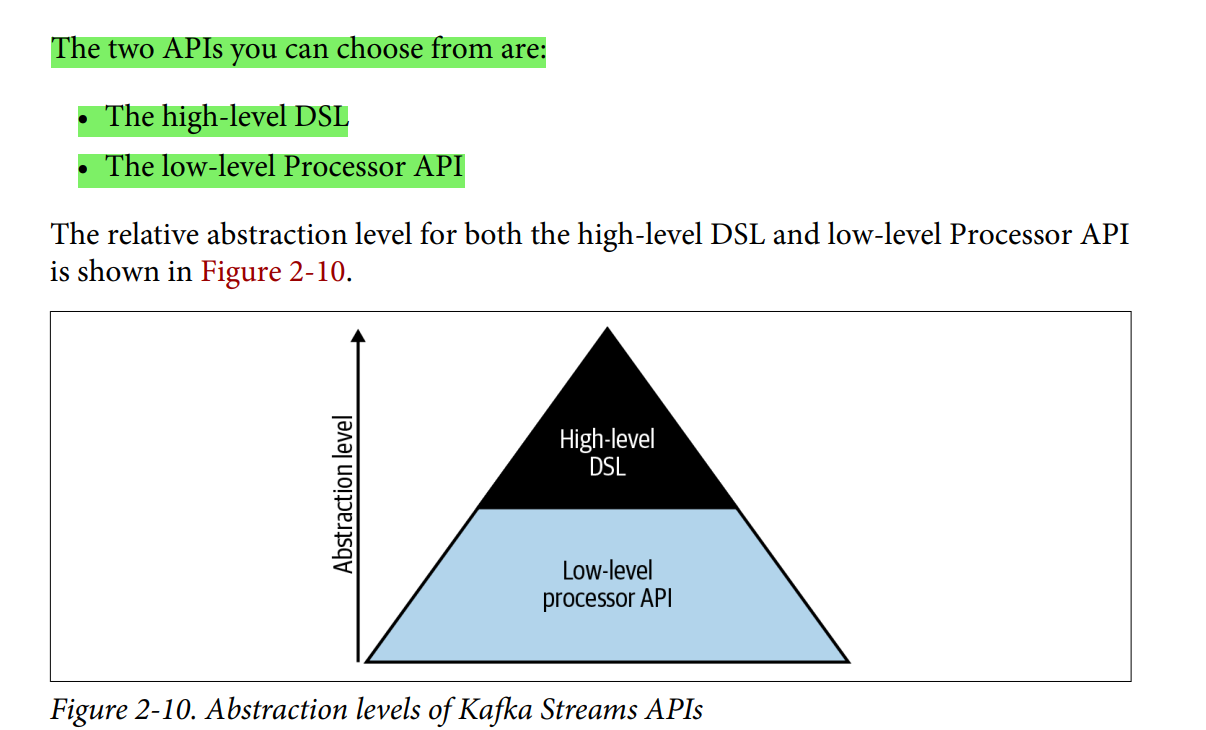
are written back to Kafka, either to be handled by another stream processing

application or to be sent to a downstream data store via something like Kafka

Connect. Like source processors, sink processors are connected to a Kafka topic.

Depth-First Processing Kafka Streams uses a depth-first strategy when processing data. When a new record is received, it is routed through each stream processor in the topology before another record is processed

**The use case that can be showed as project:**



**processor topology involves specifying a set of source and sink processors, which correspond to the top‐ ics your application will read from and write to. However, instead of working with Kafka topics directly, the Kafka Streams DSL allows you to work with different repre‐ sentations of a topic, each of which are suitable for different use cases. There are two ways to model the data in your Kafka topics: as a stream (also called a record stream) or a table (also known as a changelog stream)**

Records that are written to Kafka are immutable, so how is it possible to model data as updates, using a table represen‐ tation of a Kafka topic?

The Kafka stream is visualized as Table; built on client side;:

**The answer is simple: the table is materialized on the Kafka Streams side using a keyvalue store which, by default, is implemented using RocksDB.26 By consuming an ordered stream of events and keeping only the latest record for each key in the clientside key-value store (more commonly called a state store in Kafka Streams terminol‐ ogy), we end up with a table or map-like representation of the data. In other words, the table isn’t something we consume from Kafka, but something we build on the client side.**

What about the other side of the coin (representing a table as a stream)? When view‐ ing a table, you are viewing a single point-in-time representation of a stream. As we saw earlier, tables can be updated when a new record arrives. By changing our view of the table to a stream, we can simply process the update as an insert, and append the new record to the end of the log instead of updating the key

The following list includes a high-level overview of each:

**KStream**

A KStream is an abstraction of a partitioned record stream, in which data is repre‐

sented using insert semantics (i.e., each event is considered to be independent of

other events).

**KTable**

A KTable is an abstraction of a partitioned table (i.e., changelog stream), in which

data is represented using update semantics (the latest representation of a given

key is tracked by the application). Since KTables are partitioned, each Kafka

Streams task contains only a subset of the full table.28

**GlobalKTable**

This is similar to a KTable, except each GlobalKTable contains a complete (i.e.,

unpartitioned) copy of the underlying data.

**CHAPTER 3:**

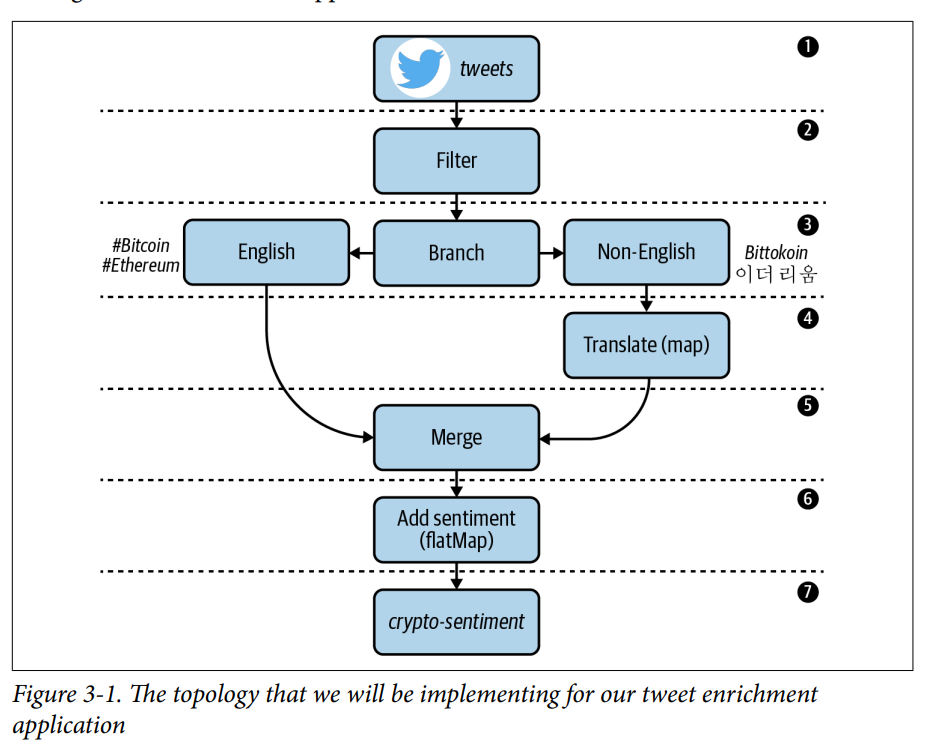
stateless operators that are included in Kafka Streams,

Filtering records • Adding and removing fields • Rekeying records • Branching streams • Merging streams • Transforming records into one or more outputs • Enriching records, one at a time

**Used Both StateLess and stateful Processing:**

Operators are stream processing functions (e.g., filter, map, flatMap, join, etc.) that are applied to events as they flow through your topology. Some operators, like filter, are consid‐ ered stateless because they only need to look at the current record to perform an action (in this case, filter looks at each record individually to determine whether or not the record should be forwarded to downstream processors). Other operators, like count, are stateful since they require knowledge of previous events (count needs to know how many events it has seen so far in order to track the number of messages).

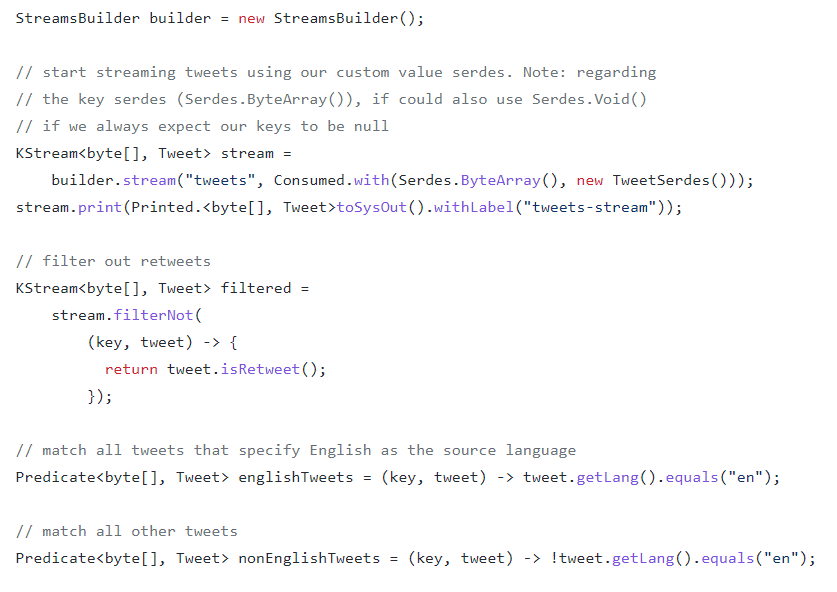
**UseCase; same as equaivalent to football in git:**



In Kafka Streams, serializer and deserializer classes are often combined into a single class called a **Serdes**

**Data Class to define to use for serilaization:**

One feature of Gson (and some other JSON serialization libraries, like Jackson) is that it allows us to convert JSON byte arrays into Java objects. In order to do this, we sim‐ ply need to define a data class, or POJO (Plain Old Java Object), that contains the fields that we want to deserialize from the source object.



**Filtring Data:**

One of the most common stateless tasks in a stream processing application is filtering data. Filtering involves selecting only a subset of records to be processed, and ignor‐ ing the rest

Similrly; we ave from merging, branching Streams.

**Avro Serilaizers: TO DO as project cae:**

When we serialize data using Avro, we have two choices:

• Include the Avro schema in each record.

• Use an even more compact format, by saving the Avro schema in Confluent

Schema Registry, and only including a much smaller schema ID in each record

instead of the entire schema.

Performing 1:1 record transformations using map and mapValues

• Performing 1:N record transformations using flatMap and flatMapValues

• Writing records to output topics using to, through, and repartition

• Serializing, deserializing, and reserializing data using custom serializers, deserial‐

izers, and Serdes implementations

**CHAPTER 5: WINDOWS TIMING:**

event time, ingestion time, and processing time

How to build a custom timestamp extractor for associating events with a particu‐

lar timestamp and time semantic

• How time controls the flow of data through Kafka Streams

• What types of windows are supported in Kafka Streams

• How to perform windowed joins

• How to perform windowed aggregations

• What strategies are available for dealing with late and out-of-order events

• How to use the suppress operator to process the final results of windows

• How to query windowed key-value stores