

Reactor Kafka Reference Guide

Rajini Sivaram, Mark Pollack, Oleh Dokuka, Gary Russell

1.3.24-SNAPSHOT

Introduction

Chapter 1. Overview

1.1. Apache Kafka

[Kafka](#) is a scalable, high-performance distributed messaging engine. Low latency, high throughput messaging capability combined with fault-tolerance have made Kafka a popular messaging service as well as a powerful streaming platform for processing real-time streams of events.

Apache Kafka provides three main APIs:

- Producer/Consumer API to publish messages to Kafka topics and consume messages from Kafka topics
- Connector API to pull data from existing data storage systems to Kafka or push data from Kafka topics to other data systems
- Streams API for transforming and analyzing real-time streams of events published to Kafka

1.2. Project Reactor

[Reactor](#) is a highly optimized reactive library for building efficient, non-blocking applications on the JVM based on the [Reactive Streams Specification](#). Reactor based applications can sustain very high throughput message rates and operate with a very low memory footprint, making it suitable for building efficient event-driven applications using the microservices architecture.

Reactor implements two publishers [Flux<T>](#) and [Mono<T>](#), both of which support non-blocking back-pressure. This enables exchange of data between threads with well-defined memory usage, avoiding unnecessary intermediate buffering or blocking.

1.3. Reactive API for Kafka

[Reactor Kafka](#) is a reactive API for Kafka based on Reactor and the Kafka Producer/Consumer API. Reactor Kafka API enables messages to be published to Kafka and consumed from Kafka using functional APIs with non-blocking back-pressure and very low overheads. This enables applications using Reactor to use Kafka as a message bus or streaming platform and integrate with other systems to provide an end-to-end reactive pipeline.

Chapter 2. Motivation

2.1. Functional interface for Kafka

Reactor Kafka is a functional Java API for Kafka. For applications that are written in functional style, this API enables Kafka interactions to be integrated easily without requiring non-functional asynchronous produce or consume APIs to be incorporated into the application logic.

2.2. Non-blocking Back-pressure

The Reactor Kafka API benefits from non-blocking back-pressure provided by Reactor. For example, in a pipeline, where messages received from an external source (e.g. an HTTP proxy) are published to Kafka, back-pressure can be applied easily to the whole pipeline, limiting the number of messages in-flight and controlling memory usage. Messages flow through the pipeline as they are available, with Reactor taking care of limiting the flow rate to avoid overflow, keeping application logic simple.

2.3. End-to-end Reactive Pipeline

The value proposition for Reactor Kafka is the efficient utilization of resources in applications with multiple external interactions where Kafka is one of the external systems. End-to-end reactive pipelines benefit from non-blocking back-pressure and efficient use of threads, enabling a large number of concurrent requests to be processed efficiently. The optimizations provided by Project Reactor enable development of reactive applications with very low overheads and predictable capacity planning to deliver low-latency, high-throughput pipelines.

2.4. Comparisons with other Kafka APIs

Reactor Kafka is not intended to replace any of the existing Kafka APIs. Instead, it is aimed at providing an alternative API for reactive event-driven applications.

2.4.1. Kafka Producer and Consumer APIs

For non-reactive applications, Kafka's Producer/Consumer API provides a low latency interface to publish messages to Kafka and consume messages from Kafka.

Applications using Kafka as a message bus using this API may consider switching to Reactor Kafka if the application is implemented in a functional style.

2.4.2. Kafka Connect API

[Kafka Connect](#) provides a simple interface to migrate messages from an external data system (e.g. a database) to one or more Kafka topics. Using existing connectors, this migration can be performed without writing any new code.

Applications using the connector API may consider using Reactor Kafka if a reactive API is available

for the external data system and transformations are required for the data. When transformations involve other I/O (e.g. to obtain additional information from another database), a reactive pipeline benefits from end-to-end non-blocking back-pressure provided by Reactor. Messages from/to different Kafka partitions can be processed in parallel, improving throughput by avoiding blocking for I/O. The pull model in Reactor controls the pace of messages flowing through the pipeline, enabling efficient use of threads and memory without the need for overflow handling in the application.

2.4.3. Kafka Streams API

[Kafka Streams](#) provides lightweight APIs to build stream processing applications that process data stored in Kafka using standard streaming concepts and transformation primitives. Using a simple threading model, the streams API avoids the need for back-pressure. This model works well in cases where transformations do not involve external interactions.

Reactor Kafka is useful for streams applications which process data from Kafka and use external interactions (e.g. get additional data for records from a database) for transformations. In this case, Reactor can provide end-to-end non-blocking back-pressure combined with better utilization of resources if all external interactions use the reactive model.

Chapter 3. Getting Started

3.1. Requirements

You need Java JRE installed (Java 8 or later).

You need [Apache Kafka](#) installed (1.0.0 or later). Kafka can be downloaded from [kafka.apache.org/downloads.html](#). Note that the Apache Kafka client library used with Reactor Kafka should be 2.0.0 or later and the broker version should be 1.0.0 or higher.

3.2. Quick Start

This quick start tutorial sets up a single node Zookeeper and Kafka and runs the sample reactive producer and consumer. Instructions to set up multi-broker clusters are available [here](#).

3.2.1. Start Kafka

If you haven't yet downloaded Kafka, download Kafka version [2.0.0](#) or higher.

Unzip the release and set KAFKA_DIR to the installation directory. For example,

```
> tar -zxf kafka_2.11-2.0.0.tgz -C /opt
> export KAFKA_DIR=/opt/kafka_2.11-2.0.0
```

Start a single-node Zookeeper instance using the Zookeeper installation included in the Kafka download:

```
> $KAFKA_DIR/bin/zookeeper-server-start.sh $KAFKA_DIR/config/zookeeper.properties >
/tmp/zookeeper.log &
```

Start a single-node Kafka instance:

```
> $KAFKA_DIR/bin/kafka-server-start.sh $KAFKA_DIR/config/server.properties >
/tmp/kafka.log &
```

Create a Kafka topic:

```
> $KAFKA_DIR/bin/kafka-topics.sh --zookeeper localhost:2181 --create --replication
-factor 1 --partitions 2 --topic demo-topic
Created topic "demo-topic".
```

Check that Kafka topic was created successfully:

```
> $KAFKA_DIR/bin/kafka-topics.sh --zookeeper localhost:2181 --describe
Topic: demo-topic    PartitionCount:2      ReplicationFactor:1 Configs:
Topic: demo-topic    Partition: 0          Leader: 0             Replicas: 0          Isr: 0
Topic: demo-topic    Partition: 1          Leader: 0             Replicas: 0          Isr: 0
```

3.2.2. Run Reactor Kafka Samples

Download and build Reactor Kafka from github.com/reactor/reactor-kafka/.

```
> git clone https://github.com/reactor/reactor-kafka
> cd reactor-kafka
> ./gradlew jar
```

Set **CLASSPATH** for running reactor-kafka samples. **CLASSPATH** can be obtained using the classpath task of the samples sub-project.

```
> export CLASSPATH=`./gradlew -q :reactor-kafka-samples:classpath`
```

Sample Producer

See github.com/reactor/reactor-kafka/blob/main/reactor-kafka-samples/src/main/java/reactor/kafka/samples/SampleProducer.java for sample producer code.

Run sample producer:

```
> $KAFKA_DIR/bin/kafka-run-class.sh reactor.kafka.samples.SampleProducer
Message 2 sent successfully, topic-partition=demo-topic-1 offset=0
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 3 sent successfully, topic-partition=demo-topic-1 offset=1
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 4 sent successfully, topic-partition=demo-topic-1 offset=2
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 6 sent successfully, topic-partition=demo-topic-1 offset=3
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 7 sent successfully, topic-partition=demo-topic-1 offset=4
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 10 sent successfully, topic-partition=demo-topic-1 offset=5
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 11 sent successfully, topic-partition=demo-topic-1 offset=6
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 12 sent successfully, topic-partition=demo-topic-1 offset=7
timestamp=13:33:16:717 GMT 30 Nov 2016
Message 13 sent successfully, topic-partition=demo-topic-1 offset=8
timestamp=13:33:16:717 GMT 30 Nov 2016
Message 14 sent successfully, topic-partition=demo-topic-1 offset=9
timestamp=13:33:16:717 GMT 30 Nov 2016
Message 16 sent successfully, topic-partition=demo-topic-1 offset=10
timestamp=13:33:16:717 GMT 30 Nov 2016
Message 17 sent successfully, topic-partition=demo-topic-1 offset=11
timestamp=13:33:16:717 GMT 30 Nov 2016
Message 20 sent successfully, topic-partition=demo-topic-1 offset=12
timestamp=13:33:16:717 GMT 30 Nov 2016
Message 1 sent successfully, topic-partition=demo-topic-0 offset=0
timestamp=13:33:16:712 GMT 30 Nov 2016
Message 5 sent successfully, topic-partition=demo-topic-0 offset=1
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 8 sent successfully, topic-partition=demo-topic-0 offset=2
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 9 sent successfully, topic-partition=demo-topic-0 offset=3
timestamp=13:33:16:716 GMT 30 Nov 2016
Message 15 sent successfully, topic-partition=demo-topic-0 offset=4
timestamp=13:33:16:717 GMT 30 Nov 2016
Message 18 sent successfully, topic-partition=demo-topic-0 offset=5
timestamp=13:33:16:717 GMT 30 Nov 2016
Message 19 sent successfully, topic-partition=demo-topic-0 offset=6
timestamp=13:33:16:717 GMT 30 Nov 2016
```

The sample producer sends 20 messages to Kafka topic `demo-topic` using the default partitioner. The partition and offset of each published message is output to console. As shown in the sample output above, the order of results may be different from the order of messages published. Results are delivered in order for each partition, but results from different partitions may be interleaved. In the sample, message index is included as correlation metadata to match each result to its corresponding message.

Sample Consumer

See github.com/reactor/reactor-kafka/blob/main/reactor-kafka-samples/src/main/java/reactor/kafka/samples/SampleConsumer.java for sample consumer code.

Run sample consumer:

```
> $KAFKA_DIR/bin/kafka-run-class.sh reactor.kafka.samples.SampleConsumer
Received message: topic-partition=demo-topic-1 offset=0 timestamp=13:33:16:716 GMT 30
Nov 2016 key=2 value=Message_2
Received message: topic-partition=demo-topic-1 offset=1 timestamp=13:33:16:716 GMT 30
Nov 2016 key=3 value=Message_3
Received message: topic-partition=demo-topic-1 offset=2 timestamp=13:33:16:716 GMT 30
Nov 2016 key=4 value=Message_4
Received message: topic-partition=demo-topic-1 offset=3 timestamp=13:33:16:716 GMT 30
Nov 2016 key=6 value=Message_6
Received message: topic-partition=demo-topic-1 offset=4 timestamp=13:33:16:716 GMT 30
Nov 2016 key=7 value=Message_7
Received message: topic-partition=demo-topic-1 offset=5 timestamp=13:33:16:716 GMT 30
Nov 2016 key=10 value=Message_10
Received message: topic-partition=demo-topic-1 offset=6 timestamp=13:33:16:716 GMT 30
Nov 2016 key=11 value=Message_11
Received message: topic-partition=demo-topic-1 offset=7 timestamp=13:33:16:717 GMT 30
Nov 2016 key=12 value=Message_12
Received message: topic-partition=demo-topic-1 offset=8 timestamp=13:33:16:717 GMT 30
Nov 2016 key=13 value=Message_13
Received message: topic-partition=demo-topic-1 offset=9 timestamp=13:33:16:717 GMT 30
Nov 2016 key=14 value=Message_14
Received message: topic-partition=demo-topic-1 offset=10 timestamp=13:33:16:717 GMT 30
Nov 2016 key=16 value=Message_16
Received message: topic-partition=demo-topic-1 offset=11 timestamp=13:33:16:717 GMT 30
Nov 2016 key=17 value=Message_17
Received message: topic-partition=demo-topic-1 offset=12 timestamp=13:33:16:717 GMT 30
Nov 2016 key=20 value=Message_20
Received message: topic-partition=demo-topic-0 offset=0 timestamp=13:33:16:712 GMT 30
Nov 2016 key=1 value=Message_1
Received message: topic-partition=demo-topic-0 offset=1 timestamp=13:33:16:716 GMT 30
Nov 2016 key=5 value=Message_5
Received message: topic-partition=demo-topic-0 offset=2 timestamp=13:33:16:716 GMT 30
Nov 2016 key=8 value=Message_8
Received message: topic-partition=demo-topic-0 offset=3 timestamp=13:33:16:716 GMT 30
Nov 2016 key=9 value=Message_9
Received message: topic-partition=demo-topic-0 offset=4 timestamp=13:33:16:717 GMT 30
Nov 2016 key=15 value=Message_15
Received message: topic-partition=demo-topic-0 offset=5 timestamp=13:33:16:717 GMT 30
Nov 2016 key=18 value=Message_18
Received message: topic-partition=demo-topic-0 offset=6 timestamp=13:33:16:717 GMT 30
Nov 2016 key=19 value=Message_19
```

The sample consumer consumes messages from topic **demo-topic** and outputs the messages to

console. The 20 messages published by the Producer sample should appear on the console. As shown in the output above, messages are consumed in order for each partition, but messages from different partitions may be interleaved.

3.2.3. Building Reactor Kafka Applications

To build your own application using the Reactor Kafka API, you need to include a dependency to Reactor Kafka.

For gradle:

```
dependencies {  
    compile "io.projectreactor.kafka:reactor-kafka:1.3.24-SNAPSHOT"  
}
```

For maven:

```
<dependency>  
    <groupId>io.projectreactor.kafka</groupId>  
    <artifactId>reactor-kafka</artifactId>  
    <version>1.3.24-SNAPSHOT</version>  
</dependency>
```

Chapter 4. Additional Resources

4.1. Getting help

If you are having trouble with Reactor Kafka, we'd like to help.

Report bugs in Reactor Kafka at github.com/reactor/reactor-kafka/issues.

Reactor Kafka is open source and the code and documentation are available at github.com/reactor/reactor-kafka.

4.2. Resources

- [Reactor Kafka on github](#)
- [Apache Kafka](#)
- [Project Reactor](#)
- [Reactor Core](#)
- [Reactive Streams Specification](#)
- [Understanding Reactive types](#)
- [Lite Rx API Hands-on](#)
- [Reactor by Example](#)

Reference Documentation

Chapter 5. Reactor Kafka API

5.1. Overview

This section describes the reactive API for producing and consuming messages using Apache Kafka. There are two main interfaces in Reactor Kafka:

1. `reactor.kafka.sender.KafkaSender` for publishing messages to Kafka
2. `reactor.kafka.receiver.KafkaReceiver` for consuming messages from Kafka

Full API for Reactor Kafka is available in the [javadocs](#).

The project uses [Reactor Core](#) to expose a "Reactive Streams" API.

5.2. Reactive Kafka Sender

Outbound messages are sent to Kafka using `reactor.kafka.sender.KafkaSender`. Senders are thread-safe and can be shared across multiple threads to improve throughput. A `KafkaSender` is associated with one `KafkaProducer` that is used to transport messages to Kafka.

A `KafkaSender` is created with an instance of sender configuration options `reactor.kafka.sender.SenderOptions`. Changes made to `SenderOptions` after the creation of `KafkaSender` will not be used by the `KafkaSender`. The properties of `SenderOptions` such as a list of bootstrap Kafka brokers and serializers are passed down to the underlying `KafkaProducer`. The properties may be configured on the `SenderOptions` instance at creation time or by using the setter `SenderOptions#producerProperty`. Other configuration options for the reactive `KafkaSender` like the maximum number of in-flight messages can also be configured before the `KafkaSender` instance is created.

The generic types of `SenderOptions<K, V>` and `KafkaSender<K, V>` are the key and value types of producer records published using the `KafkaSender` and corresponding serializers must be set on the `SenderOptions` instance before the `KafkaSender` is created.

```
Map<String, Object> producerProps = new HashMap<>();
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class);
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);

SenderOptions<Integer, String> senderOptions =
    SenderOptions.<Integer, String>create(producerProps) ①
        .maxInFlight(1024); ②
```

- ① Specify properties for underlying `KafkaProducer`
- ② Configure options for reactive `KafkaSender`

Once the required options have been configured on the options instance, a new `KafkaSender` instance can be created with the options already configured in `senderOptions`.

```
KafkaSender<Integer, String> sender = KafkaSender.create(senderOptions);
```

The `KafkaSender` is now ready to send messages to Kafka. The underlying `KafkaProducer` instance is created lazily when the first message is ready to be sent. At this point, a `KafkaSender` instance has been created, but no connections to Kafka have been made yet.

Let's now create a sequence of messages to send to Kafka. Each outbound message to be sent to Kafka is represented as a `SenderRecord`. A `SenderRecord` is a Kafka `ProducerRecord` with additional correlation metadata for matching send results to records. `ProducerRecord` consists of a key/value pair to send to Kafka and the name of the Kafka topic to send the message to. Producer records may also optionally specify a partition to send the message to or use the configured partitioner to choose a partition. Timestamp may also be optionally specified in the record and if not specified, the current timestamp will be assigned by the Producer. The additional correlation metadata included in `SenderRecord` is not sent to Kafka, but is included in the `SendResult` generated for the record when the send operation completes or fails. Since results of sends to different partitions may be interleaved, the correlation metadata enables results to be matched to their corresponding record.

A `Flux<SenderRecord>` of records is created for sending to Kafka. For beginners, [Lite Rx API Hands-on](#) provides a hands-on tutorial on using the Reactor classes `Flux` and `Mono`.

```
Flux<SenderRecord<Integer, String, Integer>> outboundFlux =  
    Flux.range(1, 10)  
        .map(i -> SenderRecord.create(topic, partition, timestamp, i, "Message_" + i,  
i));
```

The code segment above creates a sequence of messages to send to Kafka, using the message index as correlation metadata in each `SenderRecord`. The outbound Flux can now be sent to Kafka using the `KafkaSender` created earlier.

The code segment below sends the records to Kafka and prints out the response metadata received from Kafka and the correlation metadata for each record. The final `subscribe()` in the code block requests upstream to send the records to Kafka and the response metadata received from Kafka flow downstream. As each result is received, the record metadata from Kafka along with the correlation metadata identifying the record is printed out to console by the `onNext` handler. The response from Kafka includes the partition to which the record was sent as well as the offset at the which the record was appended, if available. When records are sent to multiple partitions, responses arrive in order for each partition, but responses from different partitions may be interleaved.

```

sender.send(outboundFlux) ①
    .doOnError(e-> log.error("Send failed", e)) ②
    .doOnNext(r -> System.out.printf("Message #%d send response: %s\n", r
    .correlationMetadata(), r.recordMetadata())) ③
    .subscribe(); ④

```

- ① Reactive send operation for the outbound Flux
- ② If Kafka send fails, log an error
- ③ Print metadata returned by Kafka and the message index in `correlationMetadata()`
- ④ Subscribe to trigger the actual flow of records from `outboundFlux` to Kafka.

See github.com/reactor/reactor-kafka/blob/main/reactor-kafka-samples/src/main/java/reactor/kafka/samples/SampleProducer.java for the full code listing of a sample producer.

5.2.1. Error handling

```

public SenderOptions<K, V> stopOnError(boolean stopOnError);

```

`SenderOptions#stopOnError()` specifies whether each send sequence should fail immediately if one record could not be delivered to Kafka after the configured number of retries or wait until all records have been processed. This can be used along with `ProducerConfig#ACKS_CONFIG` and `ProducerConfig#RETRIES_CONFIG` to configure the required quality of service.

```

<T> Flux<SenderResult<T>> send(Publisher<SenderRecord<K, V, T>> outboundRecords);

```

If `stopOnError` is false, a success or error response is returned for each outgoing record. For error responses, the exception from Kafka indicating the reason for send failure is set on `SenderResult` and can be retrieved using `SenderResult#exception()`. The Flux fails with an error after attempting to send all records published on `outboundRecords`. If `outboundRecords` is a non-terminating Flux, send continues to send records published on this Flux until the result Flux is explicitly cancelled by the user.

If `stopOnError` is true, a response is returned for the first failed send and the result Flux is terminated immediately with an error. Since multiple outbound messages may be in-flight at any time, it is possible that some messages are delivered successfully to Kafka after the first failure is detected. `SenderOptions#maxInFlight()` option may be configured to limit the number of messages in-flight at any time.

5.2.2. Send without result metadata

If individual results are not required for each send request, `ProducerRecord` can be sent to Kafka without providing correlation metadata using the `KafkaOutbound` interface. `KafkaOutbound` is a fluent interface that enables sends to be chained together.

```
KafkaOutbound<K, V> send(Publisher<? extends ProducerRecord<K, V>> outboundRecords);
```

The send sequence is initiated by subscribing to the Mono obtained from `KafkaOutbound#then()`. The returned Mono completes successfully if all the outbound records are delivered successfully. The Mono terminates on the first send failure. If `outboundRecords` is a non-terminating Flux, records continue to be sent to Kafka unless a send fails or the returned Mono is cancelled.

```
sender.createOutbound()
    .send(Flux.range(1, 10)
        .map(i -> new ProducerRecord<Integer, String>(topic, i, "Message_" +
i))) ①
    .then() ②
    .doOnError(e -> e.printStackTrace()) ③
    .doOnSuccess(s -> System.out.println("Sends succeeded")) ④
    .subscribe(); ⑤
```

- ① Create `ProducerRecord` Flux. Records are not wrapped in `SenderRecord`
- ② Get the `Mono` to subscribe to for starting the message flow
- ③ Error indicates failure to send one or more records
- ④ Success indicates all records were published, individual partitions or offsets not returned
- ⑤ Subscribe to request the actual sends

Multiple sends can be chained together using a sequence of sends on `KafkaOutbound`. When the Mono returned from `KafkaOutbound#then()` is subscribed to, the sends are invoked in sequence in the declaration order. The sequence is cancelled if any of the sends fail after the configured number of retries.

```
sender.createOutbound()
    .send(flux1) ①
    .send(flux2)
    .send(flux3)
    .then() ②
    .doOnError(e -> e.printStackTrace()) ③
    .doOnSuccess(s -> System.out.println("Sends succeeded")) ④
    .subscribe(); ⑤
```

- ① Sends `flux1`, `flux2` and `flux3` in order
- ② Get the `Mono` to subscribe to for starting the message flow sequence
- ③ Error indicates failure to send one or more records from any of the sends in the chain
- ④ Success indicates successful send of all records from the whole chain
- ⑤ Subscribe to initiate the sequence of sends in the chain

Note that in all cases the retries configured for the `KafkaProducer` are attempted and failures returned by the reactive `KafkaSender` indicate a failure to send after the configured number of retry

attempts. Retries can result in messages being delivered out of order. The producer property `ProducerConfig#MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION` may be set to one to avoid re-ordering.

5.2.3. Threading model

`KafkaProducer` uses a separate network thread for sending requests and processing responses. To ensure that the producer network thread is never blocked by applications while processing results, `KafkaSender` delivers responses to applications on a separate scheduler. By default, this is a single threaded pooled scheduler that is freed when no longer required. The scheduler can be overridden if required, for instance, to use a parallel scheduler when the Kafka sends are part of a larger pipeline. This is done on the `SenderOptions` instance before the `KafkaSender` instance is created using:

```
public SenderOptions<K, V> scheduler(Scheduler scheduler);
```

5.2.4. Non-blocking back-pressure

The number of in-flight sends can be controlled using the `maxInFlight` option. Requests for more elements from upstream are limited by the configured `maxInFlight` to ensure that the total number of requests at any time for which responses are pending are limited. Along with `buffer.memory` and `max.block.ms` options on `KafkaProducer`, `maxInFlight` enables control of memory and thread usage when `KafkaSender` is used in a reactive pipeline. This option can be configured on `SenderOptions` before the `KafkaSender` is created. Default value is 256. For small messages, a higher value will improve throughput.

```
public SenderOptions<K, V> maxInFlight(int maxInFlight);
```

5.2.5. Closing the KafkaSender

When the `KafkaSender` is no longer required, the `KafkaSender` instance can be closed. The underlying `KafkaProducer` is closed, closing all client connections and freeing all memory used by the producer.

```
sender.close();
```

5.2.6. Access to the underlying `KafkaProducer`

Reactive applications may sometimes require access to the underlying producer instance to perform actions that are not exposed by the `KafkaSender` interface. For example, an application might need to know the number of partitions in a topic in order to choose the partition to send a record to. Operations that are not provided directly by `KafkaSender` like `send` can be run on the underlying `KafkaProducer` using `KafkaSender#doOnProducer`.

```
sender.doOnProducer(producer -> producer.partitionsFor(topic))
    .doOnSuccess(partitions -> System.out.println("Partitions " + partitions))
    .subscribe();
```

User provided methods are executed asynchronously. A `Mono` is returned by `doOnProducer` which completes with the value returned by the user-provided function.

5.3. Reactive Kafka Receiver

Messages stored in Kafka topics are consumed using the reactive receiver `reactor.kafka.receiver.KafkaReceiver`. Each instance of `KafkaReceiver` is associated with a single instance of `KafkaConsumer`. `KafkaReceiver` is not thread-safe since the underlying `KafkaConsumer` cannot be accessed concurrently by multiple threads.

A receiver is created with an instance of receiver configuration options `reactor.kafka.receiver.ReceiverOptions`. Changes made to `ReceiverOptions` after the creation of the receiver instance will not be used by the `KafkaReceiver`. The properties of `ReceiverOptions` such as a list of bootstrap Kafka brokers and de-serializers are passed down to the underlying `KafkaConsumer`. These properties may be configured on the `ReceiverOptions` instance at creation time or by using the setter `ReceiverOptions#consumerProperty`. Other configuration options for the reactive `KafkaReceiver` including subscription topics must be added to options before the `KafkaReceiver` instance is created.

The generic types of `ReceiverOptions<K, V>` and `KafkaReceiver<K, V>` are the key and value types of consumer records consumed using the receiver and corresponding de-serializers must be set on the `ReceiverOptions` instance before the `KafkaReceiver` is created.

```
Map<String, Object> consumerProps = new HashMap<>();
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "sample-group");
consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeserializer
    .class);
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer
    .class);

ReceiverOptions<Integer, String> receiverOptions =
    ReceiverOptions.<Integer, String>create(consumerProps)           ①
        .subscription(Collections.singleton(topic));                ②
```

① Specify properties to be provided to `KafkaConsumer`

② Topics to subscribe to

Once the required configuration options have been configured on the options instance, a new `KafkaReceiver` instance can be created with these options to consume inbound messages. The code block below creates a receiver instance and creates an inbound Flux for the receiver. The underlying `KafkaConsumer` instance is created lazily later when the inbound Flux is subscribed to.

```
Flux<ReceiverRecord<Integer, String>> inboundFlux =  
    KafkaReceiver.create(receiverOptions)  
        .receive();
```

The inbound Kafka Flux is ready to be consumed. Each inbound message delivered by the Flux is represented as a `ReceiverRecord`. Each receiver record is a `ConsumerRecord` returned by `KafkaConsumer` along with a committable `ReceiverOffset` instance. The offset must be acknowledged after the message is processed since unacknowledged offsets will not be committed. If commit interval or commit batch size are configured, acknowledged offsets will be committed periodically. Offsets may also be committed manually using `ReceiverOffset#commit()` if finer grained control of commit operations is required.

```
inboundFlux.subscribe(r -> {  
    System.out.printf("Received message: %s\n", r);           ①  
    r.receiverOffset().acknowledge();                          ②  
});
```

① Prints each consumer record from Kafka

② Acknowledges that the record has been processed so that the offset may be committed

5.3.1. Error handling

Since in reactive streams an error represents a terminal signal, any error signal emitted in the inbound Flux will cause the subscription to be cancelled and effectively cause the consumer to shut down. This can be mitigated by using the `retry()` operator (or `retryWhen` for finer grained control), which will ensure that a new consumer is created:

```
Flux<ReceiverRecord<Integer, String>> inboundFlux =  
    KafkaReceiver.create(receiverOptions)  
        .receive()  
        .retryWhen(Retry.backoff(3, Duration.of(10L, ChronoUnit.SECONDS)));
```

Any errors related to the event processing rather than the `KafkaConsumer` itself should be handled as close to the source as possible and should ideally be prevented from propagating up to the inbound Flux. This is to ensure that the `KafkaConsumer` doesn't get restarted unnecessarily due to unrelated application errors.

5.3.2. Subscribing to wildcard patterns

The example above subscribed to a single Kafka topic. The same API can be used to subscribe to more than one topic by specifying multiple topics in the collection provided to `ReceiverOptions#subscription()`. Subscription can also be made to a wildcard pattern by specifying a pattern to subscribe to. Group management in `KafkaConsumer` dynamically updates topic assignment when topics matching the pattern are created or deleted and assigns partitions of matching topics to available consumer instances.

```
receiverOptions = receiverOptions.subscription(Pattern.compile("demo.*")); ①
```

① Consume records from all topics starting with "demo"

Changes to `ReceiverOptions` must be made before the receiver instance is created. Altering the subscription deletes any existing subscriptions on the options instance.

5.3.3. Manual assignment of topic partitions

Partitions may be manually assigned to the receiver without using Kafka consumer group management.

```
receiverOptions = receiverOptions.assignment(Collections.singleton(new TopicPartition(topic, 0))); ①
```

① Consume from partition 0 of specified topic

Existing subscriptions and assignments on the options instance are deleted when a new assignment is specified. Every receiver created from this options instance with manual assignment consumes messages from all the specified partitions.

5.3.4. Controlling commit frequency

Commit frequency can be controlled using a combination of commit interval and commit batch size. Commits are performed when either the interval or batch size is reached. One or both of these options may be set on `ReceiverOptions` before the receiver instance is created. If commit interval is configured, at least one commit is scheduled within that interval if any records were consumed. If commit batch size is configured, a commit is scheduled when the configured number of records are consumed and acknowledged.

Manual acknowledgement of consumed records after processing along with automatic commits based on the configured commit frequency provides at-least-once delivery semantics. Messages are re-delivered if the consuming application crashes after message was dispatched but before it was processed and acknowledged. Only offsets explicitly acknowledged using `ReceiverOffset#acknowledge()` are committed. Note that acknowledging an offset acknowledges all previous offsets on the same partition. All acknowledged offsets are committed when partitions are revoked during rebalance and when the receive Flux is terminated.

Applications which require fine-grained control over the timing of commit operations can disable periodic commits and explicitly invoke `ReceiverOffset#commit()` when required to trigger a commit. This commit is asynchronous by default, but the application may invoke `Mono#block()` on the returned `Mono` to implement synchronous commits. Applications may batch commits by acknowledging messages as they are consumed and invoking `commit()` periodically to commit acknowledged offsets.

```
receiver.receive()
    .doOnNext(r -> {
        process(r);
        r.receiverOffset().commit().block();
    });
```

Note that committing an offset acknowledges and commits all previous offsets on that partition. All acknowledged offsets are committed when partitions are revoked during rebalance and when the receive Flux is terminated.

Starting with version 1.3.12, when a rebalance occurs due to group member changes, the rebalance is delayed until records received from the previous poll have been processed. This is controlled by two `ReceiverOptions` - `maxDelayRebalance` (default 60s) and `commitIntervalDuringDelay` (default 100ms). While the delay is in process, any offsets available for committal will be committed every `commitIntervalDuringDelay` milliseconds. This allows orderly completion of processing the records that have already been received. `maxDelayRebalance` should be less than `max.poll.interval.ms` to avoid a forced rebalance due to a non-responsive consumer.

5.3.5. Out of Order Commits

Starting with version 1.3.8, commits can be performed out of order and the framework will defer the commits as needed, until any "gaps" are filled. This removes the need for applications to keep track of offsets and commit them in the right order. Deferring commits increases the likelihood of duplicate deliveries if the application crashes while deferred commits are present.

To enable this feature, set the `maxDeferredCommits` property of `ReceiverOptions`. If the number of deferred offset commits exceeds this value, the consumer is `pause()`d until the number of deferred commits is reduced by the application acknowledging or committing some of the "missing" offsets.

```
ReceiverOptions<Object, Object> options = ReceiverOptions.create()
    .maxDeferredCommits(100)
    .subscription(Collections.singletonList("someTopic"));
```

The number is an aggregate of deferred commits across all the assigned topics/partitions.

Leaving the property at its default `0` disables the feature and commits are performed whenever called.

5.3.6. Auto-acknowledgement of batches of records

`KafkaReceiver#receiveAutoAck` returns a `Flux` of batches of records returned by each `KafkaConsumer#poll()`. The records in each batch are automatically acknowledged when the Flux corresponding to the batch terminates.

```
KafkaReceiver.create(receiverOptions)
    .receiveAutoAck()
    .concatMap(r -> r)
    .subscribe(r -> System.out.println("Received: " + r));
```

- ① Concatenate in order
- ② Print out each consumer record received, no explicit ack required

The maximum number of records in each batch can be controlled using the `KafkaConsumer` property `MAX_POLL_RECORDS`. This is used together with the fetch size and wait times configured on the `KafkaConsumer` to control the amount of data fetched from Kafka brokers in each poll. Each batch is returned as a Flux that is acknowledged after the Flux terminates. Acknowledged records are committed periodically based on the configured commit interval and batch size. This mode is simple to use since applications do not need to perform any acknowledge or commit actions. It is efficient as well but can not be used for at-least-once delivery of messages.

5.3.7. Manual acknowledgement of batches of records

`KafkaReceiver#receiveBatch` returns a Flux of batches of records returned by each `KafkaConsumer#poll()`. The records in each batch should be manually acknowledged or committed.

```
KafkaReceiver.create(receiverOptions)
    .receiveBatch()
    .concatMap(b -> b)
    .subscribe(r -> {
        System.out.println("Received message: " + r);
        r.receiverOffset().acknowledge();
    });
```

- ① Concatenate in order
- ② Print out each consumer record received
- ③ Explicit ack for each message

Same as the `KafkaReceiver#receiveAutoAck` method, the maximum number of records in each batch can be controlled using the `KafkaConsumer` property `MAX_POLL_RECORDS`. This is used together with the fetch size and wait times configured on the `KafkaConsumer` to control the amount of data fetched from Kafka brokers in each poll. But unlike the `KafkaReceiver#receiveAutoAck`, each batch is returned as a Flux that should be acknowledged or committed using `ReceiverOffset`.

As the `KafkaReceiver#receive` method messages, each message in the batch is represented as a `ReceiverRecord` which has a committable `ReceiverOffset` instance.

`KafkaReceiver#receiveBatch` combines the batch consumption mode of `KafkaReceiver#receiveAutoAck` with the manual acknowledgement/commit mode of `KafkaReceiver#receive`. This batching mode is efficient and is easy to use for at-least-once delivery of messages.

5.3.8. Disabling automatic commits

Applications which don't require offset commits to Kafka may disable automatic commits by not acknowledging any records consumed using `KafkaReceiver#receive()`.

```
receiverOptions = ReceiverOptions.<Integer, String>create()
    .commitInterval(Duration.ZERO)           ①
    .commitBatchSize(0);                     ②
KafkaReceiver.create(receiverOptions)
    .receive()
    .subscribe(r -> process(r));           ③
```

- ① Disable periodic commits
- ② Disable commits based on batch size
- ③ Process records, but don't acknowledge

5.3.9. At-most-once delivery

Applications may disable automatic commits to avoid re-delivery of records. `ConsumerConfig#AUTO_OFFSET_RESET_CONFIG` can be configured to "latest" to consume only new records. But this could mean that an unpredictable number of records are not consumed if an application fails and restarts.

`KafkaReceiver#receiveAtmostOnce` can be used to consume records with at-most-once semantics with a configurable number of records-per-partition that may be lost if the application fails or crashes. Offsets are committed synchronously before the corresponding record is dispatched. Records are guaranteed not to be re-delivered even if the consuming application fails, but some records may not be processed if an application fails after the commit before the records could be processed.

This mode is expensive since each record is committed individually and records are not delivered until the commit operation succeeds. `ReceiverOptions#atmostOnceCommitCommitAheadSize` may be configured to reduce the cost of commits and avoid blocking before dispatch if the offset of the record has already been committed. By default, commit-ahead is disabled and at-most one record is lost per-partition if an application crashes. If commit-ahead is configured, the maximum number of records that may be lost per-partition is `ReceiverOptions#atmostOnceCommitCommitAheadSize + 1`.

```
KafkaReceiver.create(receiverOptions)
    .receiveAtmostOnce()
    .subscribe(r -> System.out.println("Received: " + r)); ①
```

- ① Process each consumer record, this record is not re-delivered if the processing fails

5.3.10. Partition assignment and revocation listeners

Applications can enable assignment and revocation listeners to perform any actions when partitions are assigned or revoked from a consumer.

When group management is used, assignment listeners are invoked whenever partitions are assigned to the consumer after a rebalance operation. When manual assignment is used, assignment listeners are invoked when the consumer is started. Assignment listeners can be used to seek to particular offsets in the assigned partitions so that messages are consumed from the specified offset. When a user pauses topics/partitions before rebalancing, the behavior depends on the value of `pauseAllAfterRebalance`. If it is set to `false`, the paused topics/partitions will remain paused after the rebalance. However, if it is set to `true`, all assigned topics/partitions will be paused after the rebalance.

When group management is used, revocation listeners are invoked whenever partitions are revoked from a consumer after a rebalance operation. When manual assignment is used, revocation listeners are invoked before the consumer is closed. Revocation listeners can be used to commit processed offsets when manual commits are used. Acknowledged offsets are automatically committed on revocation if automatic commits are enabled.

5.3.11. Controlling start offsets for consuming records

By default, receivers start consuming records from the last committed offset of each assigned partition. If a committed offset is not available, the offset reset strategy `ConsumerConfig#AUTO_OFFSET_RESET_CONFIG` configured for the `KafkaConsumer` is used to set the start offset to the earliest or latest offset on the partition. Applications can override offsets by seeking to new offsets in an assignment listener. Methods are provided on `ReceiverPartition` to seek to the earliest, latest, a specific offset in the partition, or to a record with a timestamp later than a point in time.

```
void seekToBeginning();
void seekToEnd();
void seek(long offset);
void seekToTimestamp(long timestamp);
```

For example, the following code block starts consuming messages from the latest offset.

```
receiverOptions = receiverOptions
    .addAssignListener(partitions -> partitions.forEach(p -> p.seekToEnd()))
    .subscription(Collections.singleton(topic));
KafkaReceiver.create(receiverOptions).receive().subscribe();
```

① Seek to the last offset in each assigned partition

Other methods are available on `ReceiverPartition` to determine the current position, the beginning offset, and ending offset, at the time the partition is assigned.

```
long position();
Long beginningOffset();
Long endOffset();
```

5.3.12. Consumer lifecycle

Each `KafkaReceiver` instance is associated with a `KafkaConsumer` that is created when the inbound Flux returned by one of the receive methods in `KafkaReceiver` is subscribed to. The consumer is kept alive until the Flux completes. When the Flux completes, all acknowledged offsets are committed and the underlying consumer is closed.

Only one receive operation may be active in a `KafkaReceiver` at any one time. Any of the receive methods can be invoked after the receive Flux corresponding to the last receive is terminated.

5.4. Micrometer Metrics

To enable micrometer metrics for the underlying Kafka Consumers and Producers, add a `MicrometerConsumerListener` to the `ReceiverOptions` or a `MicrometerProducerListener` to the `SenderOptions` respectively.

5.5. Micrometer Observation

To enable Micrometer observation for produced and consumed records, add an `ObservationRegistry` to the `SenderOptions` and `ReceiverOptions` using the `withObservation()` API. A custom `KafkaSenderObservationConvention` (and `KafkaReceiverObservationConvention`) can also be set. See their default implementations in the `KafkaSenderObservation` and `KafkaReceiverObservation`, respectively. The `DefaultKafkaSenderObservationConvention` exposes two low-cardinality tags: `reactor.kafka.type = sender` and `reactor.kafka.client.id` with the `ProducerConfig.CLIENT_ID_CONFIG` option or identity hash code of the `DefaultKafkaSender` instance prefixed with the `reactor-kafka-sender-`. The `DefaultKafkaReceiverObservationConvention` exposes two low-cardinality tags: `reactor.kafka.type = receiver` and `reactor.kafka.client.id` with the `ConsumerConfig.CLIENT_ID_CONFIG` option or identity hash code of the `DefaultKafkaReceiver` instance prefixed with the `reactor-kafka-receiver-`.

If a `PropagatingSenderTracingObservationHandler` is configured on the `ObservationRegistry`, the tracing information from the context around a producer record is stored into its headers before publishing this record to the Kafka topic. If a `PropagatingReceiverTracingObservationHandler` is configured on the `ObservationRegistry`, the tracing information from the mentioned Kafka record headers, is restored into the context on the receiver side with a child span.

Because the reverse order nature of the Reactor context, the observation functionality on the `KafkaReceiver` is limited just to a single `trace` logging message for each received record. Restored tracing information will be correlated into logs if so configured for the logging system. If there are requirements to continue an observation on the consumer side, the `KafkaReceiverObservation.RECEIVER_OBSERVATION` API must be used manually in the record processing operator:

```

KafkaReceiver.create(receiverOptions.subscription(List.of(topic)))
    .receive()
    .flatMap(record -> {
        Observation receiverObservation =
            KafkaReceiverObservation.RECEIVER_OBSERVATION.start(null,
                KafkaReceiverObservation
                    .DefaultKafkaReceiverObservationConvention.INSTANCE,
                () ->
                    new KafkaRecordReceiverContext(
                        record, "user.receiver", receiverOptions
                            .bootstrapServers()),
                observationRegistry);

        return Mono.just(record)
            .flatMap(TARGET_RECORD_HANDLER)
            .doOnTerminate(receiverObservation::stop)
            .doOnError(receiverObservation::error)
            .contextWrite(context -> context.put
                (ObservationThreadLocalAccessor.KEY, receiverObservation));
    })
    .subscribe();

```

Chapter 6. Sample Scenarios

This section shows sample code segments for typical scenarios where Reactor Kafka API may be used. Full code listing for these scenarios are included in the [samples sub-project](#).

6.1. Sending records to Kafka

See [KafkaSender API](#) for details on the KafkaSender API for sending outbound records to Kafka. The following code segment creates a simple pipeline that sends records to Kafka and processes the responses. The outbound flow is triggered when the returned Flux is subscribed to.

```
KafkaSender.create(SenderOptions.<Integer, String>create(producerProps).maxInFlight  
(512)) ①  
    .send(outbound.map(r -> senderRecord(r)))  
②  
    .doOnNext(result -> processResponse(result))  
③  
    .doOnError(e -> processError(e));
```

① Create a sender with maximum 512 messages in-flight

② Send a sequence of sender records

③ Process send result when onNext is triggered

6.2. Replaying records from Kafka topics

See [KafkaReceiver API](#) for details on the KafkaReceiver API for consuming records from Kafka topics. The following code segment creates a Flux that replays all records on a topic and commits offsets after processing the messages. Manual acknowledgement provides at-least-once delivery semantics.

```
ReceiverOptions<Integer, String> options =  
    ReceiverOptions.<Integer, String>create(consumerProps)  
        .consumerProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,  
"earliest") ①  
        .commitBatchSize(10)  
②  
        .subscription(Collections.singleton("demo-topic"));  
③  
KafkaReceiver.create(options)  
    .receive()  
    .doOnNext(r -> {  
        processRecord(r); ④  
        r.receiverOffset().acknowledge(); ⑤  
    })  
    .subscribe();
```

- ① Start consuming from first available offset on each partition if committed offsets are not available
- ② Commit every 10 acknowledged messages
- ③ Topics to consume from
- ④ Process consumer record from Kafka
- ⑤ Acknowledge that record has been consumed

6.3. Reactive pipeline with Kafka sink

The code segment below consumes messages from an external source, performs some transformation and stores the output records in Kafka. Large number of retry attempts are configured on the Kafka producer so that transient failures don't impact the pipeline. Source commits are performed only after records are successfully written to Kafka.

```
senderOptions = senderOptions
    .producerProperty(ProducerConfig.ACKS_CONFIG, "all")           ①
    .producerProperty(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE) ②
    .maxInFlight(128);                                           ③
KafkaSender.create(senderOptions)
    .send(source.flux().map(r -> transform(r)))                  ④
    .doOnError(e -> log.error("Send failed, terminating.", e))    ⑤
    .doOnNext(r -> source.commit(r.correlationMetadata()))        ⑥
    .retryWhen(Retry.backoff(3, Duration.of(10L, ChronoUnit.SECONDS)));
```

- ① Send is acknowledged by Kafka for acks=all after message is delivered to all in-sync replicas
- ② Large number of retries in the producer to cope with transient failures in brokers
- ③ Low in-flight count to avoid filling up producer buffer and blocking the pipeline, default stopOnError=true
- ④ Receive from external source, transform and send to Kafka
- ⑤ If a send fails, it indicates catastrophic error, fail the whole pipeline
- ⑥ Use correlation metadata in the sender record to commit source record

6.4. Reactive pipeline with Kafka source

The code segment below consumes records from Kafka topics, transforms the record and sends the output to an external sink. Kafka consumer offsets are committed after records are successfully output to sink.

```

receiverOptions = receiverOptions
    .commitInterval(Duration.ZERO)           ①
    .commitBatchSize(0)                     ②
    .subscription(Pattern.compile(topics));  ③
KafkaReceiver.create(receiverOptions)
    .receive()
    .publishOn(aBoundedElasticScheduler) ④
    .concatMap(m -> sink.store(transform(m))
        .doOnSuccess(r -> m.receiverOffset().commit().block()))
    .retryWhen(Retry.backoff(3, Duration.of(10L, ChronoUnit.SECONDS)));
    ⑤
    ⑥

```

- ① Disable periodic commits
- ② Disable commits by batch size
- ③ Wildcard subscription
- ④ Cannot block the receiver thread
- ⑤ Transform Kafka record and store in external sink
- ⑥ Synchronous commit after record is successfully delivered to sink

6.5. Reactive pipeline with Kafka source and sink

The code segment below consumes messages from Kafka topic, performs some transformation on the incoming messages and stores the result in some Kafka topics. Manual acknowledgement mode provides at-least-once semantics with messages acknowledged after the output records are delivered to Kafka. Acknowledged offsets are committed periodically based on the configured commit interval.

```

receiverOptions = receiverOptions
    .commitInterval(Duration.ofSeconds(10)) ①
    .subscription(Pattern.compile(topics));
sender.send(KafkaReceiver.create(receiverOptions)
    .receive()
    .map(m -> SenderRecord.create(transform(m.value()), m
    .receiverOffset())) ②
    .doOnNext(m -> m.correlationMetadata().acknowledge()); ③

```

- ① Configure interval for automatic commits
- ② Transform incoming record and create outbound record with transformed data in the payload and inbound offset as correlation metadata
- ③ Acknowledge the inbound offset using the offset instance in correlation metadata after outbound record is delivered to Kafka

6.6. At-most-once delivery

The code segment below demonstrates a flow with at-most once delivery. Producer does not wait for acks and does not perform any retries. Messages that cannot be delivered to Kafka on the first attempt are dropped. `KafkaReceiver` commits offsets before delivery to the application to ensure that if the consumer restarts, messages are not redelivered. With replication factor 1 for topic partitions, this code can be used for at-most-once delivery.

```
senderOptions = senderOptions
    .producerProperty(ProducerConfig.ACKS_CONFIG, "0")           ①
    .producerProperty(ProducerConfig.RETRIES_CONFIG, "0")        ②
    .stopOnError(false);                                         ③
receiverOptions = receiverOptions
    .subscription(Collections.singleton(sourceTopic));
KafkaSender.create(senderOptions)
    .send(KafkaReceiver.create(receiverOptions)
        .receiveAtmostOnce()                                     ④
        .map(cr -> SenderRecord.create(transform(cr.value()),
cr.offset())));
```

- ① Send with acks=0 completes when message is buffered locally, before it is delivered to Kafka broker
- ② No retries in producer
- ③ Ignore any error and continue to send remaining records
- ④ At-most-once receive

6.7. Fan-out with Multiple Streams

The code segment below demonstrates fan-out with the same records processed in multiple independent streams. Each stream is processed on a different thread and which transforms the input record and stores the output in a Kafka topic.

Reactor's `EmitterProcessor` is used to broadcast the input records from Kafka to multiple subscribers.

```

EmitterProcessor<Person> processor = EmitterProcessor.create(); ①
BlockingSink<Person> incoming = processor.connectSink(); ②
inputRecords = KafkaReceiver.create(receiverOptions)
    .receive()
    .doOnNext(m -> incoming.emit(m.value())); ③

outputRecords1 = processor.publishOn(scheduler1).map(p -> process1(p)); ④
outputRecords2 = processor.publishOn(scheduler2).map(p -> process2(p)); ⑤

Flux.merge(sender.send(outputRecords1), sender.send(outputRecords2))
    .doOnSubscribe(s -> inputRecords.subscribe())
    .subscribe(); ⑥

```

- ① Create publish/subscribe EmitterProcessor for fan-out of Kafka inbound records
- ② Create BlockingSink to which records are emitted
- ③ Receive from Kafka and emit to BlockingSink
- ④ Consume records on a scheduler, process and generate output records to send to Kafka
- ⑤ Add another processor for the same input data on a different scheduler
- ⑥ Merge the streams and subscribe to start the flow

6.8. Concurrent Processing with Partition-Based Ordering

The code segment below demonstrates a flow where messages are consumed from a Kafka topic, processed by multiple threads and the results stored in another Kafka topic. Messages are grouped by partition to guarantee ordering in message processing and commit operations. Messages from each partition are processed on a single thread.

```

Scheduler scheduler = Schedulers.newElastic("sample", 60, true);
KafkaReceiver.create(receiverOptions)
    .receive()
    .groupBy(m -> m.receiverOffset().topicPartition()) ①
    .flatMap(partitionFlux ->
        partitionFlux.publishOn(scheduler)
            .map(r -> processRecord(partitionFlux.key(), r))
            .sample(Duration.ofMillis(5000)) ②
            .concatMap(offset -> offset.commit())); ③

```

- ① Group by partition to guarantee ordering
- ② Commit periodically
- ③ Commit in sequence using concatMap

6.9. Transactional send

The code segment below consumes messages from an external source, performs some transformation and stores multiple transformed records in different Kafka topics within a transaction.

```
senderOptions = senderOptions
    .producerProperty(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "SampleTxn"); ①
KafkaSender.create(senderOptions)
    .sendTransactionally(source.map(r -> Flux.fromIterable(transform(r)))) ②
    .concatMap(r -> r)
    .doOnError(e -> log.error("Send failed, terminating.", e))
    .doOnNext(r -> log.debug("Send completed {}", r.correlationMetadata()));
```

① Configure transactional id for producer

② Send multiple records generated from each source record within a transaction

6.10. Exactly-once delivery

The code segment below demonstrates a flow with exactly once delivery. Source records received from a Kafka topic are transformed and sent to Kafka. Each batch of records is delivered to the application in a new transaction. Offsets of the source records of each batch are automatically committed within its transaction. Each transaction is committed by the application after the transformed records of the batch are successfully delivered to the destination topic. Next batch of records is delivered to the application in a new transaction after the current transaction is committed.

```
senderOptions = senderOptions
    .producerProperty(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "SampleTxn"); ①
receiverOptions = receiverOptions
    .consumerProperty(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed") ②
    .subscription(Collections.singleton(sourceTopic));
sender = KafkaSender.create(senderOptions);
transactionManager = sender.transactionManager();
receiver.receiveExactlyOnce(transactionManager) ③
    .concatMap(f -> sender.send(f.map(r -> transform(r))) ④
        .concatWith(transactionManager.commit())) ⑤
    .onErrorResume(e -> transactionManager.abort().then(Mono.error(e))) ⑥
```

① Configure transactional id for producer

② Consume only committed messages

③ Receive exactly once within transactions, offsets are auto-committed when transaction is committed

④ Send transformed records within the same transaction as source record offsets

⑤ Commit transaction after sends complete successfully

⑥ Abort transaction if send fails and propagate error