Spring for Apache Kafka

Gary Russell, Artem Bilan, Biju Kunjummen, Jay Bryant

Version 2.7.3

Table of Contents

This documentation is also available as HTML.

Chapter 1. Preface

The Spring for Apache Kafka project applies core Spring concepts to the development of Kafka-based messaging solutions. We provide a “template” as a high-level abstraction for sending messages. We also provide support for Message-driven POJOs.

Chapter 2. What's new?

2.1. What's New in 2.7 Since 2.6

This section covers the changes made from version 2.6 to version 2.7. For changes in earlier version, see Change History.

2.1.1. Kafka Client Version

This version requires the 2.7.0 `kafka-clients`. It is also compatible with the 2.8.0 clients, since version 2.7.1; see Override Spring Boot Dependencies.

2.1.2. Non-Blocking Delayed Retries Using Topics

This significant new feature is added in this release. When strict ordering is not important, failed deliveries can be sent to another topic to be consumed later. A series of such retry topics can be configured, with increasing delays. See Non-Blocking Retries for more information.

2.1.3. Listener Container Changes

The `onlyLogRecordMetadata` container property is now `true` by default.

A new container property `stopImmediate` is now available.

See Listener Container Properties for more information.

Error handlers that use a `BackOff` between delivery attempts (e.g. `SeekToCurrentErrorHandler` and `DefaultAfterRollbackProcessor`) will now exit the back off interval soon after the container is stopped, rather than delaying the stop. See After-rollback Processor and Seek To Current Container Error Handlers for more information.

Error handlers and after rollback processors that extend `FailedRecordProcessor` can now be configured with one or more `RetryListener` s to receive information about retry and recovery progress.

See See After-rollback Processor, Seek To Current Container Error Handlers, and Recovering Batch Error Handler for more information.

The `RecordInterceptor` now has additional methods called after the listener returns (normally, or by throwing an exception). It also has a sub-interface `ConsumerAwareRecordInterceptor`. In addition, there is now a `BatchInterceptor` for batch listeners. See Message Listener Containers for more information.

2.1.4. `@KafkaListener` Changes

You can now validate the payload parameter of `@KafkaHandler` methods (class-level listeners). See `@KafkaListener` `@Payload` Validation for more information.

You can now set the `rawRecordHeader` property on the `MessagingMessageConverter` and

`BatchMessagingMessageConverter` which causes the raw `ConsumerRecord` to be added to the converted `Message<?>`. This is useful, for example, if you wish to use a `DeadLetterPublishingRecoverer` in a listener error handler. See Listener Error Handlers for more information.

You can now modify `@KafkaListener` annotations during application initialization. See `@KafkaListener` Attribute Modification for more information.

2.1.5. `DeadLetterPublishingRecover` Changes

Now, if both the key and value fail deserialization, the original values are published to the DLT. Previously, the value was populated but the key `DeserializationException` remained in the headers. There is a breaking API change, if you subclassed the recoverer and overrode the `createProducerRecord` method.

In addition, the recoverer verifies that the partition selected by the destination resolver actually exists before publishing to it.

See Publishing Dead-letter Records for more information.

2.1.6. `ChainedKafkaTransactionManager` is Deprecated

See Transactions for more information.

2.1.7. `ReplyingKafkaTemplate` Changes

There is now a mechanism to examine a reply and fail the future exceptionally if some condition exists.

Support for sending and receiving `spring-messaging` `Message<?>` s has been added.

See Using `ReplyingKafkaTemplate` for more information.

2.1.8. Kafka Streams Changes

By default, the `StreamsBuilderFactoryBean` is now configured to not clean up local state. See Configuration for more information.

2.1.9. `KafkaAdmin` Changes

New methods `createOrModifyTopics` and `describeTopics` have been added. `KafkaAdmin.NewTopics` has been added to facilitate configuring multiple topics in a single bean. See Configuring Topics for more information.

2.1.10. `MessageConverter` Changes

It is now possible to add a `spring-messaging` `SmartMessageConverter` to the `MessagingMessageConverter`, allowing content negotiation based on the `contentType` header. See Spring Messaging Message Conversion for more information.

2.1.11. Sequencing `@KafkaListener` s

See [container-sequencing] for more information.

2.1.12. `ExponentialBackOffWithMaxRetries`

A new `BackOff` implementation is provided, making it more convenient to configure the max retries.
See `ExponentialBackOffWithMaxRetries` Implementation for more information.

Chapter 3. Introduction

This first part of the reference documentation is a high-level overview of Spring for Apache Kafka and the underlying concepts and some code snippets that can help you get up and running as quickly as possible.

3.1. Quick Tour

Prerequisites: You must install and run Apache Kafka. Then you must put the Spring for Apache Kafka (`spring-kafka`) JAR and all of its dependencies on your class path. The easiest way to do that is to declare a dependency in your build tool.

If you are not using Spring Boot, declare the `spring-kafka` jar as a dependency in your project.

Maven

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>2.7.3</version>
</dependency>
```

Gradle

```
compile 'org.springframework.kafka:spring-kafka:2.7.3'
```

When using Spring Boot, (and you haven't used start.spring.io to create your project), omit the version and Boot will automatically bring in the correct version that is compatible with your Boot version:

Maven

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

Gradle

```
compile 'org.springframework.kafka:spring-kafka'
```

However, the quickest way to get started is to use start.spring.io (or the wizards in Spring Tool Suits and Intellij IDEA) and create a project, selecting 'Spring for Apache Kafka' as a dependency.

3.1.1. Compatibility

This quick tour works with the following versions:

- Apache Kafka Clients 2.7.0 or 2.8.0 (see Override Spring Boot Dependencies).
- Spring Framework 5.3.x
- Minimum Java version: 8

3.1.2. Getting Started

The simplest way to get started is to use start.spring.io (or the wizards in Spring Tool Suits and Intellij IDEA) and create a project, selecting 'Spring for Apache Kafka' as a dependency. Refer to the Spring Boot documentation for more information about its opinionated auto configuration of the infrastructure beans.

Here is a minimal consumer application.

Spring Boot Consumer App

Example 1. Application

Java

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public NewTopic topic() {
        return TopicBuilder.name("topic1")
                .partitions(10)
                .replicas(1)
                .build();
    }

    @KafkaListener(id = "myId", topics = "topic1")
    public void listen(String in) {
        System.out.println(in);
    }

}
```

Kotlin

```
@SpringBootApplication
class Application {

    @Bean
    fun topic() = NewTopic("topic1", 10, 1)

    @KafkaListener(id = "myId", topics = ["topic1"])
    fun listen(value: String?) {
        println(value)
    }

}

fun main(args: Array<String>) = runApplication<Application>(*args)
```

Example 2. application.properties

```
spring.kafka.consumer.auto-offset-reset=earliest
```

The `NewTopic` bean causes the topic to be created on the broker; it is not needed if the topic already exists.

Spring Boot Producer App

Example 3. Application

Java

```java
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public NewTopic topic() {
        return TopicBuilder.name("topic1")
                .partitions(10)
                .replicas(1)
                .build();
    }

    @Bean
    public ApplicationRunner runner(KafkaTemplate<String, String> template) {
        return args -> {
            template.send("topic1", "test");
        };
    }

}
```

Kotlin

```kotlin
@SpringBootApplication
class Application {

    @Bean
    fun topic() = NewTopic("topic1", 10, 1)

    @Bean
    fun runner(template: KafkaTemplate<String?, String?>) =
        ApplicationRunner { template.send("topic1", "test") }

    companion object {
        @JvmStatic
        fun main(args: Array<String>) = runApplication<Application>(*args)
    }

}
```

With Java Configuration (No Spring Boot)

Spring for Apache Kafka is designed to be used in a Spring Application Context. For example, if you create the listener container yourself outside of a Spring context, not all functions will work unless you satisfy all of the `···Aware` interfaces that the container implements.

Here is an example of an application that does not use Spring Boot; it has both a `Consumer` and `Producer`.

Example 4. Without Boot

Java

```
public class Sender {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(Config.class);
        context.getBean(Sender.class).send("test", 42);
    }

    private final KafkaTemplate<Integer, String> template;

    public Sender(KafkaTemplate<Integer, String> template) {
        this.template = template;
    }

    public void send(String toSend, int key) {
        this.template.send("topic1", key, toSend);
    }

}

public class Listener {

    @KafkaListener(id = "listen1", topics = "topic1")
    public void listen1(String in) {
        System.out.println(in);
    }

}

@Configuration
@EnableKafka
public class Config {

    @Bean
    ConcurrentKafkaListenerContainerFactory<Integer, String>
                        kafkaListenerContainerFactory(ConsumerFactory<Integer,
String> consumerFactory) {
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
                                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);
        return factory;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerProps());
    }
```

```
    private Map<String, Object> consumerProps() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
IntegerDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        // ...
        return props;
    }

    @Bean
    public Sender sender(KafkaTemplate<Integer, String> template) {
        return new Sender(template);
    }

    @Bean
    public Listener listener() {
        return new Listener();
    }

    @Bean
    public ProducerFactory<Integer, String> producerFactory() {
        return new DefaultKafkaProducerFactory<>(senderProps());
    }

    private Map<String, Object> senderProps() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer
.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer
.class);
        //...
        return props;
    }

    @Bean
    public KafkaTemplate<Integer, String> kafkaTemplate(ProducerFactory<Integer,
String> producerFactory) {
        return new KafkaTemplate<Integer, String>(producerFactory);
    }

}
```

Kotlin

```
class Sender(private val template: KafkaTemplate<Int, String>) {

    fun send(toSend: String, key: Int) {
        template.send("topic1", key, toSend)
    }

}

class Listener {

    @KafkaListener(id = "listen1", topics = ["topic1"])
    fun listen1(`in`: String) {
        println(`in`)
    }

}

@Configuration
@EnableKafka
class Config {

    @Bean
    fun kafkaListenerContainerFactory(consumerFactory: ConsumerFactory<Int,
String>) =
        ConcurrentKafkaListenerContainerFactory<Int, String>().also {
it.consumerFactory = consumerFactory }


    @Bean
    fun consumerFactory() = DefaultKafkaConsumerFactory<Int,
String>(consumerProps)

    val consumerProps = mapOf(
        ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG to "localhost:9092",
        ConsumerConfig.GROUP_ID_CONFIG to "group",
        ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG to
IntegerDeserializer::class.java,
        ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG to
StringDeserializer::class.java,
        ConsumerConfig.AUTO_OFFSET_RESET_CONFIG to "earliest"
    )

    @Bean
    fun sender(template: KafkaTemplate<Int, String>) = Sender(template)

    @Bean
    fun listener() = Listener()

    @Bean
    fun producerFactory() = DefaultKafkaProducerFactory<Int, String>(senderProps)
```

```
    val senderProps = mapOf(
        ProducerConfig.BOOTSTRAP_SERVERS_CONFIG to "localhost:9092",
        ProducerConfig.LINGER_MS_CONFIG to 10,
        ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG to
IntegerSerializer::class.java,
        ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG to
StringSerializer::class.java
    )

    @Bean
    fun kafkaTemplate(producerFactory: ProducerFactory<Int, String>) =
KafkaTemplate(producerFactory)
}
```

As you can see, you have to define several infrastructure beans when not using Spring Boot.

Chapter 4. Reference

This part of the reference documentation details the various components that comprise Spring for Apache Kafka. The main chapter covers the core classes to develop a Kafka application with Spring.

4.1. Using Spring for Apache Kafka

This section offers detailed explanations of the various concerns that impact using Spring for Apache Kafka. For a quick but less detailed introduction, see Quick Tour.

4.1.1. Connecting to Kafka

- `KafkaAdmin` - see Configuring Topics
- `ProducerFactory` - see Sending Messages
- `ConsumerFactory` - see Receiving Messages

Starting with version 2.5, each of these extends `KafkaResourceFactory`. This allows changing the bootstrap servers at runtime by adding a `Supplier<String>` to their configuration: `setBootstrapServersSupplier(() → ...)`. This will be called for all new connections to get the list of servers. Consumers and Producers are generally long-lived. To close existing Producers, call `reset()` on the `DefaultKafkaProducerFactory`. To close existing Consumers, call `stop()` (and then `start()`) on the `KafkaListenerEndpointRegistry` and/or `stop()` and `start()` on any other listener container beans.

For convenience, the framework also provides an `ABSwitchCluster` which supports two sets of bootstrap servers; one of which is active at any time. Configure the `ABSwitchCluster` and add it to the producer and consumer factories, and the `KafkaAdmin`, by calling `setBootstrapServersSupplier()`. When you want to switch, call `primary()` or `secondary()` and call `reset()` on the producer factory to establish new connection(s); for consumers, `stop()` and `start()` all listener containers. When using `@KafkaListener` s, `stop()` and `start()` the `KafkaListenerEndpointRegistry` bean.

See the Javadocs for more information.

Factory Listeners

Starting with version 2.5, the `DefaultKafkaProducerFactory` and `DefaultKafkaConsumerFactory` can be configured with a `Listener` to receive notifications whenever a producer or consumer is created or closed.

Producer Factory Listener

```
interface Listener<K, V> {

    default void producerAdded(String id, Producer<K, V> producer) {
    }

    default void producerRemoved(String id, Producer<K, V> producer) {
    }

}
```

Consumer Factory Listener

```
interface Listener<K, V> {

    default void consumerAdded(String id, Consumer<K, V> consumer) {
    }

    default void consumerRemoved(String id, Consumer<K, V> consumer) {
    }

}
```

In each case, the `id` is created by appending the `client-id` property (obtained from the `metrics()` after creation) to the factory `beanName` property, separated by `.`.

These listeners can be used, for example, to create and bind a Micrometer `KafkaClientMetrics` instance when a new client is created (and close it when the client is closed).

The framework provides listeners that do exactly that; see Micrometer Native Metrics.

4.1.2. Configuring Topics

If you define a `KafkaAdmin` bean in your application context, it can automatically add topics to the broker. To do so, you can add a `NewTopic` `@Bean` for each topic to the application context. Version 2.3 introduced a new class `TopicBuilder` to make creation of such beans more convenient. The following example shows how to do so:

Java

```java
@Bean
public KafkaAdmin admin() {
    Map<String, Object> configs = new HashMap<>();
    configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    return new KafkaAdmin(configs);
}

@Bean
public NewTopic topic1() {
    return TopicBuilder.name("thing1")
            .partitions(10)
            .replicas(3)
            .compact()
            .build();
}

@Bean
public NewTopic topic2() {
    return TopicBuilder.name("thing2")
            .partitions(10)
            .replicas(3)
            .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
            .build();
}

@Bean
public NewTopic topic3() {
    return TopicBuilder.name("thing3")
            .assignReplicas(0, Arrays.asList(0, 1))
            .assignReplicas(1, Arrays.asList(1, 2))
            .assignReplicas(2, Arrays.asList(2, 0))
            .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
            .build();
}
```

Kotlin

```
@Bean
fun admin() = KafkaAdmin(mapOf(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG to
"localhost:9092"))

@Bean
fun topic1() =
    TopicBuilder.name("thing1")
        .partitions(10)
        .replicas(3)
        .compact()
        .build()

@Bean
fun topic2() =
    TopicBuilder.name("thing2")
        .partitions(10)
        .replicas(3)
        .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
        .build()

@Bean
fun topic3() =
    TopicBuilder.name("thing3")
        .assignReplicas(0, Arrays.asList(0, 1))
        .assignReplicas(1, Arrays.asList(1, 2))
        .assignReplicas(2, Arrays.asList(2, 0))
        .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
        .build()
```

Starting with version 2.6, you can omit `.partitions()` and/or `replicas()` and the broker defaults will be applied to those properties. The broker version must be at least 2.4.0 to support this feature - see KIP-464.

Java

```
@Bean
public NewTopic topic4() {
    return TopicBuilder.name("defaultBoth")
            .build();
}

@Bean
public NewTopic topic5() {
    return TopicBuilder.name("defaultPart")
            .replicas(1)
            .build();
}

@Bean
public NewTopic topic6() {
    return TopicBuilder.name("defaultRepl")
            .partitions(3)
            .build();
}
```

Kotlin

```
@Bean
fun topic4() = TopicBuilder.name("defaultBoth").build()

@Bean
fun topic5() = TopicBuilder.name("defaultPart").replicas(1).build()

@Bean
fun topic6() = TopicBuilder.name("defaultRepl").partitions(3).build()
```

Starting with version 2.7, you can declare multiple `NewTopic` s in a single `KafkaAdmin.NewTopics` bean definition:

Java

```
@Bean
public KafkaAdmin.NewTopics topics456() {
    return new NewTopics(
            TopicBuilder.name("defaultBoth")
                .build(),
            TopicBuilder.name("defaultPart")
                .replicas(1)
                .build(),
            TopicBuilder.name("defaultRepl")
                .partitions(3)
                .build());
}
```

Kotlin

```
@Bean
fun topics456() = KafkaAdmin.NewTopics(
    TopicBuilder.name("defaultBoth")
        .build(),
    TopicBuilder.name("defaultPart")
        .replicas(1)
        .build(),
    TopicBuilder.name("defaultRepl")
        .partitions(3)
        .build()
)
```

When using Spring Boot, a `KafkaAdmin` bean is automatically registered so you only need the `NewTopic` (and/or `NewTopics`) `@Bean` s.

By default, if the broker is not available, a message is logged, but the context continues to load. You can programmatically invoke the admin's `initialize()` method to try again later. If you wish this condition to be considered fatal, set the admin's `fatalIfBrokerNotAvailable` property to `true`. The context then fails to initialize.

If the broker supports it (1.0.0 or higher), the admin increases the number of partitions if it is found that an existing topic has fewer partitions than the `NewTopic.numPartitions`.

Starting with version 2.7, the `KafkaAdmin` provides methods to create and examine topics at runtime.

- `createOrModifyTopics`
- `describeTopics`

For more advanced features, you can use the `AdminClient` directly. The following example shows

how to do so:

```
@Autowired
private KafkaAdmin admin;

...

    AdminClient client = AdminClient.create(admin.getConfigurationProperties());
    ...
    client.close();
```

4.1.3. Sending Messages

This section covers how to send messages.

Using `KafkaTemplate`

This section covers how to use `KafkaTemplate` to send messages.

Overview

The `KafkaTemplate` wraps a producer and provides convenience methods to send data to Kafka topics. The following listing shows the relevant methods from `KafkaTemplate`:

```
ListenableFuture<SendResult<K, V>> sendDefault(V data);

ListenableFuture<SendResult<K, V>> sendDefault(K key, V data);

ListenableFuture<SendResult<K, V>> sendDefault(Integer partition, K key, V data);

ListenableFuture<SendResult<K, V>> sendDefault(Integer partition, Long timestamp,
K key, V data);

ListenableFuture<SendResult<K, V>> send(String topic, V data);

ListenableFuture<SendResult<K, V>> send(String topic, K key, V data);

ListenableFuture<SendResult<K, V>> send(String topic, Integer partition, K key, V
data);

ListenableFuture<SendResult<K, V>> send(String topic, Integer partition, Long
timestamp, K key, V data);

ListenableFuture<SendResult<K, V>> send(ProducerRecord<K, V> record);

ListenableFuture<SendResult<K, V>> send(Message<?> message);

Map<MetricName, ? extends Metric> metrics();

List<PartitionInfo> partitionsFor(String topic);

<T> T execute(ProducerCallback<K, V, T> callback);

// Flush the producer.

void flush();

interface ProducerCallback<K, V, T> {

    T doInKafka(Producer<K, V> producer);

}
```

See the Javadoc for more detail.

The `sendDefault` API requires that a default topic has been provided to the template.

The API takes in a `timestamp` as a parameter and stores this timestamp in the record. How the user-provided timestamp is stored depends on the timestamp type configured on the Kafka topic. If the topic is configured to use `CREATE_TIME`, the user specified timestamp is recorded (or generated if not specified). If the topic is configured to use `LOG_APPEND_TIME`, the user-specified timestamp is ignored and the broker adds in the local broker time.

The `metrics` and `partitionsFor` methods delegate to the same methods on the underlying `Producer`. The `execute` method provides direct access to the underlying `Producer`.

To use the template, you can configure a producer factory and provide it in the template's constructor. The following example shows how to do so:

```
@Bean
public ProducerFactory<Integer, String> producerFactory() {
    return new DefaultKafkaProducerFactory<>(producerConfigs());
}

@Bean
public Map<String, Object> producerConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.
class);
    // See https://kafka.apache.org/documentation/#producerconfigs for more
properties
    return props;
}

@Bean
public KafkaTemplate<Integer, String> kafkaTemplate() {
    return new KafkaTemplate<Integer, String>(producerFactory());
}
```

Starting with version 2.5, you can now override the factory's `ProducerConfig` properties to create templates with different producer configurations from the same factory.

```
@Bean
public KafkaTemplate<String, String> stringTemplate(ProducerFactory<String,
String> pf) {
    return new KafkaTemplate<>(pf);
}

@Bean
public KafkaTemplate<String, byte[]> bytesTemplate(ProducerFactory<String, byte[]>
pf) {
    return new KafkaTemplate<>(pf,
            Collections.singletonMap(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
ByteArraySerializer.class));
}
```

Note that a bean of type `ProducerFactory<?, ?>` (such as the one auto-configured by Spring Boot) can be referenced with different narrowed generic types.

You can also configure the template by using standard `<bean/>` definitions.

Then, to use the template, you can invoke one of its methods.

When you use the methods with a `Message<?>` parameter, the topic, partition, and key information is provided in a message header that includes the following items:

- `KafkaHeaders.TOPIC`
- `KafkaHeaders.PARTITION_ID`
- `KafkaHeaders.MESSAGE_KEY`
- `KafkaHeaders.TIMESTAMP`

The message payload is the data.

Optionally, you can configure the `KafkaTemplate` with a `ProducerListener` to get an asynchronous callback with the results of the send (success or failure) instead of waiting for the `Future` to complete. The following listing shows the definition of the `ProducerListener` interface:

```
public interface ProducerListener<K, V> {

    void onSuccess(ProducerRecord<K, V> producerRecord, RecordMetadata
recordMetadata);

    void onError(ProducerRecord<K, V> producerRecord, RecordMetadata
recordMetadata,
            Exception exception);

}
```

By default, the template is configured with a `LoggingProducerListener`, which logs errors and does nothing when the send is successful.

For convenience, default method implementations are provided in case you want to implement only one of the methods.

Notice that the send methods return a `ListenableFuture<SendResult>`. You can register a callback with the listener to receive the result of the send asynchronously. The following example shows how to do so:

```
ListenableFuture<SendResult<Integer, String>> future = template.send("myTopic",
"something");
future.addCallback(new ListenableFutureCallback<SendResult<Integer, String>>() {

    @Override
    public void onSuccess(SendResult<Integer, String> result) {
        ...
    }

    @Override
    public void onFailure(Throwable ex) {
        ...
    }

});
```

`SendResult` has two properties, a `ProducerRecord` and `RecordMetadata`. See the Kafka API documentation for information about those objects.

The `Throwable` in `onFailure` can be cast to a `KafkaProducerException`; its `failedProducerRecord` property contains the failed record.

Starting with version 2.5, you can use a `KafkaSendCallback` instead of a `ListenableFutureCallback`, making it easier to extract the failed `ProducerRecord`, avoiding the need to cast the `Throwable`:

```
ListenableFuture<SendResult<Integer, String>> future = template.send("topic", 1,
"thing");
future.addCallback(new KafkaSendCallback<Integer, String>() {

    @Override
    public void onSuccess(SendResult<Integer, String> result) {
        ...
    }

    @Override
    public void onFailure(KafkaProducerException ex) {
        ProducerRecord<Integer, String> failed = ex.getFailedProducerRecord();
        ...
    }

});
```

You can also use a pair of lambdas:

```
ListenableFuture<SendResult<Integer, String>> future = template.send("topic", 1,
"thing");
future.addCallback(result -> {
        ...
    }, (KafkaFailureCallback<Integer, String>) ex -> {
            ProducerRecord<Integer, String> failed = ex.getFailedProducerRecord();
            ...
    });
```

If you wish to block the sending thread to await the result, you can invoke the future's `get()` method; using the method with a timeout is recommended. You may wish to invoke `flush()` before waiting or, for convenience, the template has a constructor with an `autoFlush` parameter that causes the template to `flush()` on each send. Flushing is only needed if you have set the `linger.ms` producer property and want to immediately send a partial batch.

Examples

This section shows examples of sending messages to Kafka:

Example 5. Non Blocking (Async)

```
public void sendToKafka(final MyOutputData data) {
    final ProducerRecord<String, String> record = createRecord(data);

    ListenableFuture<SendResult<Integer, String>> future = template.send(record);
    future.addCallback(new KafkaSendCallback<SendResult<Integer, String>>() {

        @Override
        public void onSuccess(SendResult<Integer, String> result) {
            handleSuccess(data);
        }

        @Override
        public void onFailure(KafkaProducerException ex) {
            handleFailure(data, record, ex);
        }

    });
}
```

Blocking (Sync)

```
public void sendToKafka(final MyOutputData data) {
    final ProducerRecord<String, String> record = createRecord(data);

    try {
        template.send(record).get(10, TimeUnit.SECONDS);
        handleSuccess(data);
    }
    catch (ExecutionException e) {
        handleFailure(data, record, e.getCause());
    }
    catch (TimeoutException | InterruptedException e) {
        handleFailure(data, record, e);
    }
}
```

Note that the cause of the `ExecutionException` is `KafkaProducerException` with the `failedProducerRecord` property.

Using `RoutingKafkaTemplate`

Starting with version 2.5, you can use a `RoutingKafkaTemplate` to select the producer at runtime, based on the destination `topic` name.

The routing template does **not** support transactions, `execute`, `flush`, or `metrics` operations because the topic is not known for those operations.

The template requires a map of `java.util.regex.Pattern` to `ProducerFactory<Object, Object>` instances. This map should be ordered (e.g. a `LinkedHashMap`) because it is traversed in order; you should add more specific patterns at the beginning.

The following simple Spring Boot application provides an example of how to use the same template to send to different topics, each using a different value serializer.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public RoutingKafkaTemplate routingTemplate(GenericApplicationContext context,
            ProducerFactory<Object, Object> pf) {

        // Clone the PF with a different Serializer, register with Spring for
shutdown
        Map<String, Object> configs = new HashMap<>(pf.getConfigurationProperties
());
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
ByteArraySerializer.class);
        DefaultKafkaProducerFactory<Object, Object> bytesPF = new
DefaultKafkaProducerFactory<>(configs);
        context.registerBean(DefaultKafkaProducerFactory.class, "bytesPF",
bytesPF);

        Map<Pattern, ProducerFactory<Object, Object>> map = new LinkedHashMap<>();
        map.put(Pattern.compile("two"), bytesPF);
        map.put(Pattern.compile(".+"), pf); // Default PF with StringSerializer
        return new RoutingKafkaTemplate(map);
    }

    @Bean
    public ApplicationRunner runner(RoutingKafkaTemplate routingTemplate) {
        return args -> {
            routingTemplate.send("one", "thing1");
            routingTemplate.send("two", "thing2".getBytes());
        };
    }

}
```

The corresponding `@KafkaListener` s for this example are shown in Annotation Properties.

For another technique to achieve similar results, but with the additional capability of sending different types to the same topic, see Delegating Serializer and Deserializer.

Using `DefaultKafkaProducerFactory`

As seen in Using `KafkaTemplate`, a `ProducerFactory` is used to create the producer.

When not using Transactions, by default, the `DefaultKafkaProducerFactory` creates a singleton producer used by all clients, as recommended in the `KafkaProducer` javadocs. However, if you call `flush()` on the template, this can cause delays for other threads using the same producer. Starting with version 2.3, the `DefaultKafkaProducerFactory` has a new property `producerPerThread`. When set to `true`, the factory will create (and cache) a separate producer for each thread, to avoid this issue.

When `producerPerThread` is `true`, user code **must** call `closeThreadBoundProducer()` on the factory when the producer is no longer needed. This will physically close the producer and remove it from the `ThreadLocal`. Calling `reset()` or `destroy()` will not clean up these producers.

Also see `KafkaTemplate` Transactional and non-Transactional Publishing.

When creating a `DefaultKafkaProducerFactory`, key and/or value `Serializer` classes can be picked up from configuration by calling the constructor that only takes in a Map of properties (see example in Using `KafkaTemplate`), or `Serializer` instances may be passed to the `DefaultKafkaProducerFactory` constructor (in which case all `Producer` s share the same instances). Alternatively you can provide `Supplier<Serializer>` s (starting with version 2.3) that will be used to obtain separate `Serializer` instances for each `Producer`:

```
@Bean
public ProducerFactory<Integer, CustomValue> producerFactory() {
    return new DefaultKafkaProducerFactory<>(producerConfigs(), null, () -> new
CustomValueSerializer());
}

@Bean
public KafkaTemplate<Integer, CustomValue> kafkaTemplate() {
    return new KafkaTemplate<Integer, CustomValue>(producerFactory());
}
```

Starting with version 2.5.10, you can now update the producer properties after the factory is created. This might be useful, for example, if you have to update SSL key/trust store locations after a credentials change. The changes will not affect existing producer instances; call `reset()` to close any existing producers so that new producers will be created using the new properties. NOTE: You cannot change a transactional producer factory to non-transactional, and vice-versa.

Two new methods are now provided:

```
void updateConfigs(Map<String, Object> updates);

void removeConfig(String configKey);
```

Using `ReplyingKafkaTemplate`

Version 2.1.3 introduced a subclass of `KafkaTemplate` to provide request/reply semantics. The class is named `ReplyingKafkaTemplate` and has two additional methods; the following shows the method signatures:

```
RequestReplyFuture<K, V, R> sendAndReceive(ProducerRecord<K, V> record);

RequestReplyFuture<K, V, R> sendAndReceive(ProducerRecord<K, V> record,
    Duration replyTimeout);
```

(Also see Request/Reply with `Message<?>` s).

The result is a `ListenableFuture` that is asynchronously populated with the result (or an exception, for a timeout). The result also has a `sendFuture` property, which is the result of calling `KafkaTemplate.send()`. You can use this future to determine the result of the send operation.

If the first method is used, or the `replyTimeout` argument is `null`, the template's `defaultReplyTimeout` property is used (5 seconds by default).

The following Spring Boot application shows an example of how to use the feature:

```
@SpringBootApplication
public class KRequestingApplication {

    public static void main(String[] args) {
        SpringApplication.run(KRequestingApplication.class, args).close();
    }

    @Bean
    public ApplicationRunner runner(ReplyingKafkaTemplate<String, String, String>
template) {
        return args -> {
            ProducerRecord<String, String> record = new ProducerRecord<>(
"kRequests", "foo");
            RequestReplyFuture<String, String, String> replyFuture = template
.sendAndReceive(record);
            SendResult<String, String> sendResult = replyFuture.getSendFuture()
.get(10, TimeUnit.SECONDS);
            System.out.println("Sent ok: " + sendResult.getRecordMetadata());
            ConsumerRecord<String, String> consumerRecord = replyFuture.get(10,
TimeUnit.SECONDS);
            System.out.println("Return value: " + consumerRecord.value());
        };
    }

    @Bean
    public ReplyingKafkaTemplate<String, String, String> replyingTemplate(
            ProducerFactory<String, String> pf,
            ConcurrentMessageListenerContainer<String, String> repliesContainer) {

        return new ReplyingKafkaTemplate<>(pf, repliesContainer);
    }

    @Bean
    public ConcurrentMessageListenerContainer<String, String> repliesContainer(
            ConcurrentKafkaListenerContainerFactory<String, String>
containerFactory) {

        ConcurrentMessageListenerContainer<String, String> repliesContainer =
                containerFactory.createContainer("replies");
        repliesContainer.getContainerProperties().setGroupId("repliesGroup");
        repliesContainer.setAutoStartup(false);
        return repliesContainer;
    }

    @Bean
    public NewTopic kRequests() {
        return TopicBuilder.name("kRequests")
            .partitions(10)
            .replicas(2)
```

```
                .build();
    }

    @Bean
    public NewTopic kReplies() {
        return TopicBuilder.name("kReplies")
                .partitions(10)
                .replicas(2)
                .build();
    }

}
```

Note that we can use Boot's auto-configured container factory to create the reply container.

If a non-trivial deserializer is being used for replies, consider using an `ErrorHandlingDeserializer` that delegates to your configured deserializer. When so configured, the `RequestReplyFuture` will be completed exceptionally and you can catch the `ExecutionException`, with the `DeserializationException` in its `cause` property.

Starting with version 2.6.7, in addition to detecting `DeserializationException` s, the template will call the `replyErrorChecker` function, if provided. If it returns an exception, the future will be completed exceptionally.

Here is an example:

```
template.setReplyErrorChecker(record -> {
    Header error = record.headers().lastHeader("serverSentAnError");
    if (error != null) {
        return new MyException(new String(error.value()));
    }
    else {
        return null;
    }
});

...

RequestReplyFuture<Integer, String, String> future = template.sendAndReceive
(record);
try {
    future.getSendFuture().get(10, TimeUnit.SECONDS); // send ok
    ConsumerRecord<Integer, String> consumerRecord = future.get(10, TimeUnit
.SECONDS);
    ...
}
catch (InterruptedException e) {
    ...
}
catch (ExecutionException e) {
    if (e.getCause instanceof MyException) {
        ...
    }
}
catch (TimeoutException e) {
    ...
}
```

The template sets a header (named `KafkaHeaders.CORRELATION_ID` by default), which must be echoed back by the server side.

In this case, the following `@KafkaListener` application responds:

```
@SpringBootApplication
public class KReplyingApplication {

    public static void main(String[] args) {
        SpringApplication.run(KReplyingApplication.class, args);
    }

    @KafkaListener(id="server", topics = "kRequests")
    @SendTo // use default replyTo expression
    public String listen(String in) {
        System.out.println("Server received: " + in);
        return in.toUpperCase();
    }

    @Bean
    public NewTopic kRequests() {
        return TopicBuilder.name("kRequests")
            .partitions(10)
            .replicas(2)
            .build();
    }

    @Bean // not required if Jackson is on the classpath
    public MessagingMessageConverter simpleMapperConverter() {
        MessagingMessageConverter messagingMessageConverter = new
MessagingMessageConverter();
        messagingMessageConverter.setHeaderMapper(new SimpleKafkaHeaderMapper());
        return messagingMessageConverter;
    }

}
```

The `@KafkaListener` infrastructure echoes the correlation ID and determines the reply topic.

See Forwarding Listener Results using `@SendTo` for more information about sending replies. The template uses the default header `KafKaHeaders.REPLY_TOPIC` to indicate the topic to which the reply goes.

Starting with version 2.2, the template tries to detect the reply topic or partition from the configured reply container. If the container is configured to listen to a single topic or a single `TopicPartitionOffset`, it is used to set the reply headers. If the container is configured otherwise, the user must set up the reply headers. In this case, an `INFO` log message is written during initialization. The following example uses `KafkaHeaders.REPLY_TOPIC`:

```
record.headers().add(new RecordHeader(KafkaHeaders.REPLY_TOPIC, "kReplies"
.getBytes()));
```

When you configure with a single reply `TopicPartitionOffset`, you can use the same reply topic for multiple templates, as long as each instance listens on a different partition. When configuring with a single reply topic, each instance must use a different `group.id`. In this case, all instances receive each reply, but only the instance that sent the request finds the correlation ID. This may be useful for auto-scaling, but with the overhead of additional network traffic and the small cost of discarding each unwanted reply. When you use this setting, we recommend that you set the template's `sharedReplyTopic` to `true`, which reduces the logging level of unexpected replies to DEBUG instead of the default ERROR.

The following is an example of configuring the reply container to use the same shared reply topic:

```
@Bean
public ConcurrentMessageListenerContainer<String, String> replyContainer(
        ConcurrentKafkaListenerContainerFactory<String, String> containerFactory)
{

    ConcurrentMessageListenerContainer<String, String> container =
containerFactory.createContainer("topic2");
    container.getContainerProperties().setGroupId(UUID.randomUUID().toString());
// unique
    Properties props = new Properties();
    props.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest"); // so
the new group doesn't get old replies
    container.getContainerProperties().setKafkaConsumerProperties(props);
    return container;
}
```

If you have multiple client instances and you do not configure them as discussed in the preceding paragraph, each instance needs a dedicated reply topic. An alternative is to set the `KafkaHeaders.REPLY_PARTITION` and use a dedicated partition for each instance. The `Header` contains a four-byte int (big-endian). The server must use this header to route the reply to the correct partition (`@KafkaListener` does this). In this case, though, the reply container must not use Kafka's group management feature and must be configured to listen on a fixed partition (by using a `TopicPartitionOffset` in its `ContainerProperties` constructor).

The `DefaultKafkaHeaderMapper` requires Jackson to be on the classpath (for the `@KafkaListener`). If it is not available, the message converter has no header mapper, so you must configure a `MessagingMessageConverter` with a `SimpleKafkaHeaderMapper`, as shown earlier.

By default, 3 headers are used:

- `KafkaHeaders.CORRELATION_ID` - used to correlate the reply to a request
- `KafkaHeaders.REPLY_TOPIC` - used to tell the server where to reply
- `KafkaHeaders.REPLY_PARTITION` - (optional) used to tell the server which partition to reply to

These header names are used by the `@KafkaListener` infrastructure to route the reply.

Starting with version 2.3, you can customize the header names - the template has 3 properties `correlationHeaderName`, `replyTopicHeaderName`, and `replyPartitionHeaderName`. This is useful if your server is not a Spring application (or does not use the `@KafkaListener`).

Request/Reply with `Message<?>` s

Version 2.7 added methods to the `ReplyingKafkaTemplate` to send and receive `spring-messaging` 's `Message<?>` abstraction:

```
RequestReplyMessageFuture<K, V> sendAndReceive(Message<?> message);

<P> RequestReplyTypedMessageFuture<K, V, P> sendAndReceive(Message<?> message,
        ParameterizedTypeReference<P> returnType);
```

These will use the template's default `replyTimeout`, there are also overloaded versions that can take a timeout in the method call.

Use the first method if the consumer's `Deserializer` or the template's `MessageConverter` can convert the payload without any additional information, either via configuration or type metadata in the reply message.

Use the second method if you need to provide type information for the return type, to assist the message converter. This also allows the same template to receive different types, even if there is no type metadata in the replies, such as when the server side is not a Spring application. The following is an example of the latter:

Example 6. Template Bean

Java

```
@Bean
ReplyingKafkaTemplate<String, String, String> template(
        ProducerFactory<String, String> pf,
        ConcurrentKafkaListenerContainerFactory<String, String> factory) {

    ConcurrentMessageListenerContainer<String, String> replyContainer =
            factory.createContainer("replies");
    replyContainer.getContainerProperties().setGroupId("request.replies");
    ReplyingKafkaTemplate<String, String, String> template =
            new ReplyingKafkaTemplate<>(pf, replyContainer);
    template.setMessageConverter(new ByteArrayJsonMessageConverter());
    template.setDefaultTopic("requests");
    return template;
}
```

Kotlin

```
@Bean
fun template(
    pf: ProducerFactory<String?, String>?,
    factory: ConcurrentKafkaListenerContainerFactory<String?, String?>
): ReplyingKafkaTemplate<String?, String, String?> {
    val replyContainer = factory.createContainer("replies")
    replyContainer.containerProperties.groupId = "request.replies"
    val template = ReplyingKafkaTemplate(pf, replyContainer)
    template.messageConverter = ByteArrayJsonMessageConverter()
    template.defaultTopic = "requests"
    return template
}
```

Example 7. Using the template

Java

```
RequestReplyTypedMessageFuture<String, String, Thing> future1 =
        template.sendAndReceive(MessageBuilder.withPayload("getAThing").build(),
                new ParameterizedTypeReference<Thing>() { });
log.info(future1.getSendFuture().get(10, TimeUnit.SECONDS).getRecordMetadata()
.toString());
Thing thing = future1.get(10, TimeUnit.SECONDS).getPayload();
log.info(thing.toString());

RequestReplyTypedMessageFuture<String, String, List<Thing>> future2 =
        template.sendAndReceive(MessageBuilder.withPayload("getThings").build(),
                new ParameterizedTypeReference<List<Thing>>() { });
log.info(future2.getSendFuture().get(10, TimeUnit.SECONDS).getRecordMetadata()
.toString());
List<Thing> things = future2.get(10, TimeUnit.SECONDS).getPayload();
things.forEach(thing1 -> log.info(thing1.toString()));
```

Kotlin

```
val future1: RequestReplyTypedMessageFuture<String?, String?, Thing?>? =
    template.sendAndReceive(MessageBuilder.withPayload("getAThing").build(),
        object : ParameterizedTypeReference<Thing?>() {})
log.info(future1?.sendFuture?.get(10,
TimeUnit.SECONDS)?.recordMetadata?.toString())
val thing = future1?.get(10, TimeUnit.SECONDS)?.payload
log.info(thing.toString())

val future2: RequestReplyTypedMessageFuture<String?, String?, List<Thing?>?>? =
    template.sendAndReceive(MessageBuilder.withPayload("getThings").build(),
        object : ParameterizedTypeReference<List<Thing?>?>() {})
log.info(future2?.sendFuture?.get(10,
TimeUnit.SECONDS)?.recordMetadata.toString())
val things = future2?.get(10, TimeUnit.SECONDS)?.payload
things?.forEach(Consumer { thing1: Thing? -> log.info(thing1.toString()) })
```

Reply Type Message<?>

When the `@KafkaListener` returns a `Message<?>`, with versions before 2.5, it was necessary to populate the reply topic and correlation id headers. In this example, we use the reply topic header from the request:

```
@KafkaListener(id = "requestor", topics = "request")
@SendTo
public Message<?> messageReturn(String in) {
    return MessageBuilder.withPayload(in.toUpperCase())
            .setHeader(KafkaHeaders.TOPIC, replyTo)
            .setHeader(KafkaHeaders.MESSAGE_KEY, 42)
            .setHeader(KafkaHeaders.CORRELATION_ID, correlation)
            .build();
}
```

This also shows how to set a key on the reply record.

Starting with version 2.5, the framework will detect if these headers are missing and populate them with the topic - either the topic determined from the `@SendTo` value or the incoming `KafkaHeaders.REPLY_TOPIC` header (if present). It will also echo the incoming `KafkaHeaders.CORRELATION_ID` and `KafkaHeaders.REPLY_PARTITION`, if present.

```
@KafkaListener(id = "requestor", topics = "request")
@SendTo  // default REPLY_TOPIC header
public Message<?> messageReturn(String in) {
    return MessageBuilder.withPayload(in.toUpperCase())
            .setHeader(KafkaHeaders.MESSAGE_KEY, 42)
            .build();
}
```

Aggregating Multiple Replies

The template in Using `ReplyingKafkaTemplate` is strictly for a single request/reply scenario. For cases where multiple receivers of a single message return a reply, you can use the `AggregatingReplyingKafkaTemplate`. This is an implementation of the client-side of the Scatter-Gather Enterprise Integration Pattern.

Like the `ReplyingKafkaTemplate`, the `AggregatingReplyingKafkaTemplate` constructor takes a producer factory and a listener container to receive the replies; it has a third parameter `BiPredicate<List<ConsumerRecord<K, R>>, Boolean> releaseStrategy` which is consulted each time a reply is received; when the predicate returns `true`, the collection of `ConsumerRecord` s is used to complete the `Future` returned by the `sendAndReceive` method.

There is an additional property `returnPartialOnTimeout` (default false). When this is set to `true`, instead of completing the future with a `KafkaReplyTimeoutException`, a partial result completes the future normally (as long as at least one reply record has been received).

Starting with version 2.3.5, the predicate is also called after a timeout (if `returnPartialOnTimeout` is `true`). The first argument is the current list of records; the second is `true` if this call is due to a timeout. The predicate can modify the list of records.

```
AggregatingReplyingKafkaTemplate<Integer, String, String> template =
        new AggregatingReplyingKafkaTemplate<>(producerFactory, container,
                        coll -> coll.size() == releaseSize);
...
RequestReplyFuture<Integer, String, Collection<ConsumerRecord<Integer, String>>>
future =
        template.sendAndReceive(record);
future.getSendFuture().get(10, TimeUnit.SECONDS); // send ok
ConsumerRecord<Integer, Collection<ConsumerRecord<Integer, String>>>
consumerRecord =
        future.get(30, TimeUnit.SECONDS);
```

Notice that the return type is a `ConsumerRecord` with a value that is a collection of `ConsumerRecord` s. The "outer" `ConsumerRecord` is not a "real" record, it is synthesized by the template, as a holder for the actual reply records received for the request. When a normal release occurs (release strategy returns true), the topic is set to `aggregatedResults`; if `returnPartialOnTimeout` is true, and timeout occurs (and at least one reply record has been received), the topic is set to `partialResultsAfterTimeout`. The template provides constant static variables for these "topic" names:

```
/**
 * Pseudo topic name for the "outer" {@link ConsumerRecords} that has the
aggregated
 * results in its value after a normal release by the release strategy.
 */
public static final String AGGREGATED_RESULTS_TOPIC = "aggregatedResults";

/**
 * Pseudo topic name for the "outer" {@link ConsumerRecords} that has the
aggregated
 * results in its value after a timeout.
 */
public static final String PARTIAL_RESULTS_AFTER_TIMEOUT_TOPIC =
"partialResultsAfterTimeout";
```

The real `ConsumerRecord` s in the `Collection` contain the actual topic(s) from which the replies are received.

The listener container for the replies MUST be configured with `AckMode.MANUAL` or `AckMode.MANUAL_IMMEDIATE`; the consumer property `enable.auto.commit` must be `false` (the default since version 2.3). To avoid any possibility of losing messages, the template only commits offsets when there are zero requests outstanding, i.e. when the last outstanding request is released by the release strategy. After a rebalance, it is possible for duplicate reply deliveries; these will be ignored for any in-flight requests; you may see error log messages when duplicate replies are received for already released replies.

If you use an `ErrorHandlingDeserializer` with this aggregating template, the framework will not automatically detect `DeserializationException` s. Instead, the record (with a `null` value) will be returned intact, with the deserialization exception(s) in headers. It is recommended that applications call the utility method `ReplyingKafkaTemplate.checkDeserialization()` method to determine if a deserialization exception occurred. See its javadocs for more information. The `replyErrorChecker` is also not called for this aggregating template; you should perform the checks on each element of the reply.

4.1.4. Receiving Messages

You can receive messages by configuring a `MessageListenerContainer` and providing a message listener or by using the `@KafkaListener` annotation.

Message Listeners

When you use a message listener container, you must provide a listener to receive data. There are currently eight supported interfaces for message listeners. The following listing shows these interfaces:

```
public interface MessageListener<K, V> { ①

    void onMessage(ConsumerRecord<K, V> data);

}

public interface AcknowledgingMessageListener<K, V> { ②

    void onMessage(ConsumerRecord<K, V> data, Acknowledgment acknowledgment);

}

public interface ConsumerAwareMessageListener<K, V> extends MessageListener<K, V>
{ ③

    void onMessage(ConsumerRecord<K, V> data, Consumer<?, ?> consumer);

}

public interface AcknowledgingConsumerAwareMessageListener<K, V> extends
MessageListener<K, V> { ④

    void onMessage(ConsumerRecord<K, V> data, Acknowledgment acknowledgment,
Consumer<?, ?> consumer);

}

public interface BatchMessageListener<K, V> { ⑤

    void onMessage(List<ConsumerRecord<K, V>> data);

}

public interface BatchAcknowledgingMessageListener<K, V> { ⑥

    void onMessage(List<ConsumerRecord<K, V>> data, Acknowledgment acknowledgment
);

}

public interface BatchConsumerAwareMessageListener<K, V> extends
BatchMessageListener<K, V> { ⑦

    void onMessage(List<ConsumerRecord<K, V>> data, Consumer<?, ?> consumer);

}

public interface BatchAcknowledgingConsumerAwareMessageListener<K, V> extends
BatchMessageListener<K, V> { ⑧
```

```
    void onMessage(List<ConsumerRecord<K, V>> data, Acknowledgment acknowledgment,
Consumer<?, ?> consumer);

}
```

① Use this interface for processing individual `ConsumerRecord` instances received from the Kafka consumer `poll()` operation when using auto-commit or one of the container-managed commit methods.

② Use this interface for processing individual `ConsumerRecord` instances received from the Kafka consumer `poll()` operation when using one of the manual commit methods.

③ Use this interface for processing individual `ConsumerRecord` instances received from the Kafka consumer `poll()` operation when using auto-commit or one of the container-managed commit methods. Access to the `Consumer` object is provided.

④ Use this interface for processing individual `ConsumerRecord` instances received from the Kafka consumer `poll()` operation when using one of the manual commit methods. Access to the `Consumer` object is provided.

⑤ Use this interface for processing all `ConsumerRecord` instances received from the Kafka consumer `poll()` operation when using auto-commit or one of the container-managed commit methods. `AckMode.RECORD` is not supported when you use this interface, since the listener is given the complete batch.

⑥ Use this interface for processing all `ConsumerRecord` instances received from the Kafka consumer `poll()` operation when using one of the manual commit methods.

⑦ Use this interface for processing all `ConsumerRecord` instances received from the Kafka consumer `poll()` operation when using auto-commit or one of the container-managed commit methods. `AckMode.RECORD` is not supported when you use this interface, since the listener is given the complete batch. Access to the `Consumer` object is provided.

⑧ Use this interface for processing all `ConsumerRecord` instances received from the Kafka consumer `poll()` operation when using one of the manual commit methods. Access to the `Consumer` object is provided.

The `Consumer` object is not thread-safe. You must only invoke its methods on the thread that calls the listener.

You should not execute any `Consumer<?, ?>` methods that affect the consumer's positions and or committed offsets in your listener; the container needs to manage such information.

Message Listener Containers

Two `MessageListenerContainer` implementations are provided:

- `KafkaMessageListenerContainer`
- `ConcurrentMessageListenerContainer`

The `KafkaMessageListenerContainer` receives all message from all topics or partitions on a single thread. The `ConcurrentMessageListenerContainer` delegates to one or more `KafkaMessageListenerContainer` instances to provide multi-threaded consumption.

Starting with version 2.2.7, you can add a `RecordInterceptor` to the listener container; it will be invoked before calling the listener allowing inspection or modification of the record. If the interceptor returns null, the listener is not called. Starting with version 2.7, it has additional methods which are called after the listener exits (normally, or by throwing an exception). Also, starting with version 2.7, there is now a `BatchInterceptor`, providing similar functionality for Batch Listeners. In addition, the `ConsumerAwareRecordInterceptor` (and `BatchInterceptor`) provide access to the `Consumer<?, ?>`. This might be used, for example, to access the consumer metrics in the interceptor.

You should not execute any methods that affect the consumer's positions and or committed offsets in these interceptors; the container needs to manage such information.

The `CompositeRecordInterceptor` and `CompositeBatchInterceptor` can be used to invoke multiple interceptors.

By default, when using transactions, the interceptor is invoked after the transaction has started. Starting with version 2.3.4, you can set the listener container's `interceptBeforeTx` property to invoke the interceptor before the transaction has started instead.

Starting with versions 2.3.8, 2.4.6, the `ConcurrentMessageListenerContainer` now supports Static Membership when the concurrency is greater than one. The `group.instance.id` is suffixed with `-n` with `n` starting at `1`. This, together with an increased `session.timeout.ms`, can be used to reduce rebalance events, for example, when application instances are restarted.

Using `KafkaMessageListenerContainer`

The following constructor is available:

```
public KafkaMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,
                    ContainerProperties containerProperties)
```

It receives a `ConsumerFactory` and information about topics and partitions, as well as other configuration, in a `ContainerProperties` object. `ContainerProperties` has the following constructors:

```
public ContainerProperties(TopicPartitionOffset... topicPartitions)

public ContainerProperties(String... topics)

public ContainerProperties(Pattern topicPattern)
```

The first constructor takes an array of `TopicPartitionOffset` arguments to explicitly instruct the container about which partitions to use (using the consumer `assign()` method) and with an optional initial offset. A positive value is an absolute offset by default. A negative value is relative to the current last offset within a partition by default. A constructor for `TopicPartitionOffset` that takes an additional `boolean` argument is provided. If this is `true`, the initial offsets (positive or negative) are relative to the current position for this consumer. The offsets are applied when the container is started. The second takes an array of topics, and Kafka allocates the partitions based on the `group.id` property — distributing partitions across the group. The third uses a regex `Pattern` to select the topics.

To assign a `MessageListener` to a container, you can use the `ContainerProps.setMessageListener` method when creating the Container. The following example shows how to do so:

```
ContainerProperties containerProps = new ContainerProperties("topic1", "topic2");
containerProps.setMessageListener(new MessageListener<Integer, String>() {
    ...
});
DefaultKafkaConsumerFactory<Integer, String> cf =
                        new DefaultKafkaConsumerFactory<>(consumerProps());
KafkaMessageListenerContainer<Integer, String> container =
                        new KafkaMessageListenerContainer<>(cf, containerProps);
return container;
```

Note that when creating a `DefaultKafkaConsumerFactory`, using the constructor that just takes in the properties as above means that key and value `Deserializer` classes are picked up from configuration. Alternatively, `Deserializer` instances may be passed to the `DefaultKafkaConsumerFactory` constructor for key and/or value, in which case all Consumers share the same instances. Another option is to provide `Supplier<Deserializer>` s (starting with version 2.3) that will be used to obtain separate `Deserializer` instances for each `Consumer`:

```
DefaultKafkaConsumerFactory<Integer, CustomValue> cf =
                        new DefaultKafkaConsumerFactory<>(consumerProps(), null,
() -> new CustomValueDeserializer());
KafkaMessageListenerContainer<Integer, String> container =
                        new KafkaMessageListenerContainer<>(cf, containerProps);
return container;
```

Refer to the Javadoc for `ContainerProperties` for more information about the various properties that you can set.

Since version 2.1.1, a new property called `logContainerConfig` is available. When `true` and `INFO` logging is enabled each listener container writes a log message summarizing its configuration properties.

By default, logging of topic offset commits is performed at the `DEBUG` logging level. Starting with version 2.1.2, a property in `ContainerProperties` called `commitLogLevel` lets you specify the log level for these messages. For example, to change the log level to `INFO`, you can use `containerProperties.setCommitLogLevel(LogIfLevelEnabled.Level.INFO);`.

Starting with version 2.2, a new container property called `missingTopicsFatal` has been added (default: `false` since 2.3.4). This prevents the container from starting if any of the configured topics are not present on the broker. It does not apply if the container is configured to listen to a topic pattern (regex). Previously, the container threads looped within the `consumer.poll()` method waiting for the topic to appear while logging many messages. Aside from the logs, there was no indication that there was a problem.

As of version 2.3.5, a new container property called `authorizationExceptionRetryInterval` has been introduced. This causes the container to retry fetching messages after getting any `AuthorizationException` from `KafkaConsumer`. This can happen when, for example, the configured user is denied access to read certain topic. Defining `authorizationExceptionRetryInterval` should help the application to recover as soon as proper permissions are granted.

By default, no interval is configured - authorization errors are considered fatal, which causes the container to stop.

Using `ConcurrentMessageListenerContainer`

The single constructor is similar to the `KafkaListenerContainer` constructor. The following listing shows the constructor's signature:

```
public ConcurrentMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,
                                          ContainerProperties containerProperties)
```

It also has a `concurrency` property. For example, `container.setConcurrency(3)` creates three `KafkaMessageListenerContainer` instances.

For the first constructor, Kafka distributes the partitions across the consumers using its group management capabilities.

When listening to multiple topics, the default partition distribution may not be what you expect. For example, if you have three topics with five partitions each and you want to use `concurrency=15`, you see only five active consumers, each assigned one partition from each topic, with the other 10 consumers being idle. This is because the default Kafka `PartitionAssignor` is the `RangeAssignor` (see its Javadoc). For this scenario, you may want to consider using the `RoundRobinAssignor` instead, which distributes the partitions across all of the consumers. Then, each consumer is assigned one topic or partition. To change the `PartitionAssignor`, you can set the `partition.assignment.strategy` consumer property (`ConsumerConfigs.PARTITION_ASSIGNMENT_STRATEGY_CONFIG`) in the properties provided to the `DefaultKafkaConsumerFactory`.

When using Spring Boot, you can assign set the strategy as follows:

```
spring.kafka.consumer.properties.partition.assignment.strategy=\
org.apache.kafka.clients.consumer.RoundRobinAssignor
```

When the container properties are configured with `TopicPartitionOffset` s, the `ConcurrentMessageListenerContainer` distributes the `TopicPartitionOffset` instances across the delegate `KafkaMessageListenerContainer` instances.

If, say, six `TopicPartitionOffset` instances are provided and the `concurrency` is `3`; each container gets two partitions. For five `TopicPartitionOffset` instances, two containers get two partitions, and the third gets one. If the `concurrency` is greater than the number of `TopicPartitions`, the `concurrency` is adjusted down such that each container gets one partition.

The `client.id` property (if set) is appended with `-n` where `n` is the consumer instance that corresponds to the concurrency. This is required to provide unique names for MBeans when JMX is enabled.

Starting with version 1.3, the `MessageListenerContainer` provides access to the metrics of the underlying `KafkaConsumer`. In the case of `ConcurrentMessageListenerContainer`, the `metrics()` method returns the metrics for all the target `KafkaMessageListenerContainer` instances. The metrics are grouped into the `Map<MetricName, ? extends Metric>` by the `client-id` provided for the underlying `KafkaConsumer`.

Starting with version 2.3, the `ContainerProperties` provides an `idleBetweenPolls` option to let the main loop in the listener container to sleep between `KafkaConsumer.poll()` calls. An actual sleep interval is selected as the minimum from the provided option and difference between the `max.poll.interval.ms` consumer config and the current records batch processing time.

Committing Offsets

Several options are provided for committing offsets. If the `enable.auto.commit` consumer property is `true`, Kafka auto-commits the offsets according to its configuration. If it is `false`, the containers support several `AckMode` settings (described in the next list). The default `AckMode` is `BATCH`. Starting

with version 2.3, the framework sets `enable.auto.commit` to `false` unless explicitly set in the configuration. Previously, the Kafka default (`true`) was used if the property was not set.

The consumer `poll()` method returns one or more `ConsumerRecords`. The `MessageListener` is called for each record. The following lists describes the action taken by the container for each `AckMode` (when transactions are not being used):

- `RECORD`: Commit the offset when the listener returns after processing the record.
- `BATCH`: Commit the offset when all the records returned by the `poll()` have been processed.
- `TIME`: Commit the offset when all the records returned by the `poll()` have been processed, as long as the `ackTime` since the last commit has been exceeded.
- `COUNT`: Commit the offset when all the records returned by the `poll()` have been processed, as long as `ackCount` records have been received since the last commit.
- `COUNT_TIME`: Similar to `TIME` and `COUNT`, but the commit is performed if either condition is `true`.
- `MANUAL`: The message listener is responsible to `acknowledge()` the `Acknowledgment`. After that, the same semantics as `BATCH` are applied.
- `MANUAL_IMMEDIATE`: Commit the offset immediately when the `Acknowledgment.acknowledge()` method is called by the listener.

When using transactions, the offset(s) are sent to the transaction and the semantics are equivalent to `RECORD` or `BATCH`, depending on the listener type (record or batch).

`MANUAL`, and `MANUAL_IMMEDIATE` require the listener to be an `AcknowledgingMessageListener` or a `BatchAcknowledgingMessageListener`. See Message Listeners.

Depending on the `syncCommits` container property, the `commitSync()` or `commitAsync()` method on the consumer is used. `syncCommits` is `true` by default; also see `setSyncCommitTimeout`. See `setCommitCallback` to get the results of asynchronous commits; the default callback is the `LoggingCommitCallback` which logs errors (and successes at debug level).

Because the listener container has it's own mechanism for committing offsets, it prefers the Kafka `ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG` to be `false`. Starting with version 2.3, it unconditionally sets it to false unless specifically set in the consumer factory or the container's consumer property overrides.

The `Acknowledgment` has the following method:

```
public interface Acknowledgment {

    void acknowledge();

}
```

This method gives the listener control over when offsets are committed.

Starting with version 2.3, the `Acknowledgment` interface has two additional methods `nack(long sleep)` and `nack(int index, long sleep)`. The first one is used with a record listener, the second with a batch listener. Calling the wrong method for your listener type will throw an `IllegalStateException`.

If you want to commit a partial batch, using `nack()`, When using transactions, set the `AckMode` to `MANUAL`; invoking `nack()` will send the offsets of the successfully processed records to the transaction.

`nack()` can only be called on the consumer thread that invokes your listener.

With a record listener, when `nack()` is called, any pending offsets are committed, the remaing records from the last poll are discarded, and seeks are performed on their partitions so that the failed record and unprocessed records are redelivered on the next `poll()`. The consumer thread can be paused before redelivery, by setting the `sleep` argument. This is similar functionality to throwing an exception when the container is configured with a `SeekToCurrentErrorHandler`.

When using a batch listener, you can specify the index within the batch where the failure occurred. When `nack()` is called, offsets will be committed for records before the index and seeks are performed on the partitions for the failed and discarded records so that they will be redelivered on the next `poll()`. This is an improvement over the `SeekToCurrentBatchErrorHandler`, which can only seek the entire batch for redelivery.

See Seek To Current Container Error Handlers for more information. Also see Retrying Batch Error Handler.

When using partition assignment via group management, it is important to ensure the `sleep` argument (plus the time spent processing records from the previous poll) is less than the consumer `max.poll.interval.ms` property.

Listener Container Auto Startup

The listener containers implement `SmartLifecycle`, and `autoStartup` is `true` by default. The containers are started in a late phase (`Integer.MAX-VALUE - 100`). Other components that implement `SmartLifecycle`, to handle data from listeners, should be started in an earlier phase. The `- 100` leaves room for later phases to enable components to be auto-started after the containers.

`@KafkaListener` Annotation

The `@KafkaListener` annotation is used to designate a bean method as a listener for a listener container. The bean is wrapped in a `MessagingMessageListenerAdapter` configured with various features, such as converters to convert the data, if necessary, to match the method parameters.

You can configure most attributes on the annotation with SpEL by using `#{…}` or property placeholders (`${…}`). See the Javadoc for more information.

Record Listeners

The `@KafkaListener` annotation provides a mechanism for simple POJO listeners. The following example shows how to use it:

```
public class Listener {

    @KafkaListener(id = "foo", topics = "myTopic", clientIdPrefix = "myClientId")
    public void listen(String data) {
        ...
    }

}
```

This mechanism requires an `@EnableKafka` annotation on one of your `@Configuration` classes and a listener container factory, which is used to configure the underlying `ConcurrentMessageListenerContainer`. By default, a bean with name `kafkaListenerContainerFactory` is expected. The following example shows how to use `ConcurrentMessageListenerContainer`:

```
@Configuration
@EnableKafka
public class KafkaConfig {

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer,
String>>
                        kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
                                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        return factory;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, embeddedKafka
.getBrokersAsString());
        ...
        return props;
    }
}
```

Notice that, to set container properties, you must use the `getContainerProperties()` method on the factory. It is used as a template for the actual properties injected into the container.

Starting with version 2.1.1, you can now set the `client.id` property for consumers created by the annotation. The `clientIdPrefix` is suffixed with `-n`, where `n` is an integer representing the container number when using concurrency.

Starting with version 2.2, you can now override the container factory's `concurrency` and `autoStartup` properties by using properties on the annotation itself. The properties can be simple values, property placeholders, or SpEL expressions. The following example shows how to do so:

```
@KafkaListener(id = "myListener", topics = "myTopic",
        autoStartup = "${listen.auto.start:true}", concurrency =
"${listen.concurrency:3}")
public void listen(String data) {
    ...
}
```

Explicit Partition Assignment

You can also configure POJO listeners with explicit topics and partitions (and, optionally, their initial offsets). The following example shows how to do so:

```
@KafkaListener(id = "thing2", topicPartitions =
        { @TopicPartition(topic = "topic1", partitions = { "0", "1" }),
          @TopicPartition(topic = "topic2", partitions = "0",
             partitionOffsets = @PartitionOffset(partition = "1", initialOffset =
"100"))
        })
public void listen(ConsumerRecord<?, ?> record) {
    ...
}
```

You can specify each partition in the `partitions` or `partitionOffsets` attribute but not both.

As with most annotation properties, you can use SpEL expressions; for an example of how to generate a large list of partitions, see Manually Assigning All Partitions.

Starting with version 2.5.5, you can apply an initial offset to all assigned partitions:

```
@KafkaListener(id = "thing3", topicPartitions =
        { @TopicPartition(topic = "topic1", partitions = { "0", "1" },
             partitionOffsets = @PartitionOffset(partition = "*", initialOffset =
"0"))
        })
public void listen(ConsumerRecord<?, ?> record) {
    ...
}
```

The `*` wildcard represents all partitions in the `partitions` attribute. There must only be one `@PartitionOffset` with the wildcard in each `@TopicPartition`.

In addition, when the listener implements `ConsumerSeekAware`, `onPartitionsAssigned` is now called,

even when using manual assignment. This allows, for example, any arbitrary seek operations at that time.

Starting with version 2.6.4, you can specify a comma-delimited list of partitions, or partition ranges:

```
@KafkaListener(id = "pp", autoStartup = "false",
        topicPartitions = @TopicPartition(topic = "topic1",
                partitions = "0-5, 7, 10-15"))
public void process(String in) {
    ...
}
```

The range is inclusive; the example above will assign partitions `0, 1, 2, 3, 4, 5, 7, 10, 11, 12, 13, 14, 15`.

The same technique can be used when specifying initial offsets:

```
@KafkaListener(id = "thing3", topicPartitions =
        { @TopicPartition(topic = "topic1",
             partitionOffsets = @PartitionOffset(partition = "0-5", initialOffset
= "0"))
        })
public void listen(ConsumerRecord<?, ?> record) {
    ...
}
```

The initial offset will be applied to all 6 partitions.

Manual Acknowledgment

When using manual `AckMode`, you can also provide the listener with the `Acknowledgment`. The following example also shows how to use a different container factory.

```
@KafkaListener(id = "cat", topics = "myTopic",
          containerFactory = "kafkaManualAckListenerContainerFactory")
public void listen(String data, Acknowledgment ack) {
    ...
    ack.acknowledge();
}
```

Consumer Record Metadata

Finally, metadata about the record is available from message headers. You can use the following

header names to retrieve the headers of the message:

- `KafkaHeaders.OFFSET`
- `KafkaHeaders.RECEIVED_MESSAGE_KEY`
- `KafkaHeaders.RECEIVED_TOPIC`
- `KafkaHeaders.RECEIVED_PARTITION_ID`
- `KafkaHeaders.RECEIVED_TIMESTAMP`
- `KafkaHeaders.TIMESTAMP_TYPE`

Starting with version 2.5 the `RECEIVED_MESSAGE_KEY` is not present if the incoming record has a `null` key; previously the header was populated with a `null` value. This change is to make the framework consistent with `spring-messaging` conventions where `null` valued headers are not present.

The following example shows how to use the headers:

```
@KafkaListener(id = "qux", topicPattern = "myTopic1")
public void listen(@Payload String foo,
        @Header(name = KafkaHeaders.RECEIVED_MESSAGE_KEY, required = false)
Integer key,
        @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
        @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
        @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long ts
        ) {
    ...
}
```

Starting with version 2.5, instead of using discrete headers, you can receive record metadata in a `ConsumerRecordMetadata` parameter.

```
@KafkaListener(...)
public void listen(String str, ConsumerRecordMetadata meta) {
    ...
}
```

This contains all the data from the `ConsumerRecord` except the key and value.

Batch Listeners

Starting with version 1.1, you can configure `@KafkaListener` methods to receive the entire batch of consumer records received from the consumer poll. To configure the listener container factory to create batch listeners, you can set the `batchListener` property. The following example shows how to do so:

```
@Bean
public KafkaListenerContainerFactory<?, ?> batchFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setBatchListener(true);  // <<<<<<<<<<<<<<<<<<<<<<<<<
    return factory;
}
```

The following example shows how to receive a list of payloads:

```
@KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<String> list) {
    ...
}
```

The topic, partition, offset, and so on are available in headers that parallel the payloads. The following example shows how to use the headers:

```
@KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<String> list,
        @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) List<Integer> keys,
        @Header(KafkaHeaders.RECEIVED_PARTITION_ID) List<Integer> partitions,
        @Header(KafkaHeaders.RECEIVED_TOPIC) List<String> topics,
        @Header(KafkaHeaders.OFFSET) List<Long> offsets) {
    ...
}
```

Alternatively, you can receive a `List` of `Message<?>` objects with each offset and other details in each message, but it must be the only parameter (aside from optional `Acknowledgment`, when using manual commits, and/or `Consumer<?, ?>` parameters) defined on the method. The following example shows how to do so:

```
@KafkaListener(id = "listMsg", topics = "myTopic", containerFactory =
"batchFactory")
public void listen14(List<Message<?>> list) {
    ...
}

@KafkaListener(id = "listMsgAck", topics = "myTopic", containerFactory =
"batchFactory")
public void listen15(List<Message<?>> list, Acknowledgment ack) {
    ...
}

@KafkaListener(id = "listMsgAckConsumer", topics = "myTopic", containerFactory =
"batchFactory")
public void listen16(List<Message<?>> list, Acknowledgment ack, Consumer<?, ?>
consumer) {
    ...
}
```

No conversion is performed on the payloads in this case.

If the `BatchMessagingMessageConverter` is configured with a `RecordMessageConverter`, you can also add a generic type to the `Message` parameter and the payloads are converted. See Payload Conversion with Batch Listeners for more information.

You can also receive a list of `ConsumerRecord<?, ?>` objects, but it must be the only parameter (aside from optional `Acknowledgment`, when using manual commits and `Consumer<?, ?>` parameters) defined on the method. The following example shows how to do so:

```
@KafkaListener(id = "listCRs", topics = "myTopic", containerFactory =
"batchFactory")
public void listen(List<ConsumerRecord<Integer, String>> list) {
    ...
}

@KafkaListener(id = "listCRsAck", topics = "myTopic", containerFactory =
"batchFactory")
public void listen(List<ConsumerRecord<Integer, String>> list, Acknowledgment ack)
{
    ...
}
```

Starting with version 2.2, the listener can receive the complete `ConsumerRecords<?, ?>` object returned by the `poll()` method, letting the listener access additional methods, such as `partitions()` (which returns the `TopicPartition` instances in the list) and `records(TopicPartition)` (which gets

selective records). Again, this must be the only parameter (aside from optional `Acknowledgment`, when using manual commits or `Consumer<?, ?>` parameters) on the method. The following example shows how to do so:

```
@KafkaListener(id = "pollResults", topics = "myTopic", containerFactory =
"batchFactory")
public void pollResults(ConsumerRecords<?, ?> records) {
    ...
}
```

If the container factory has a `RecordFilterStrategy` configured, it is ignored for `ConsumerRecords<?, ?>` listeners, with a `WARN` log message emitted. Records can only be filtered with a batch listener if the `<List<?>>` form of listener is used.

Annotation Properties

Starting with version 2.0, the `id` property (if present) is used as the Kafka consumer `group.id` property, overriding the configured property in the consumer factory, if present. You can also set `groupId` explicitly or set `idIsGroup` to false to restore the previous behavior of using the consumer factory `group.id`.

You can use property placeholders or SpEL expressions within most annotation properties, as the following example shows:

```
@KafkaListener(topics = "${some.property}")

@KafkaListener(topics = "#{someBean.someProperty}",
    groupId = "#{someBean.someProperty}.group")
```

Starting with version 2.1.2, the SpEL expressions support a special token: `__listener`. It is a pseudo bean name that represents the current bean instance within which this annotation exists.

Consider the following example:

```
@Bean
public Listener listener1() {
    return new Listener("topic1");
}

@Bean
public Listener listener2() {
    return new Listener("topic2");
}
```

Given the beans in the previous example, we can then use the following:

```
public class Listener {

    private final String topic;

    public Listener(String topic) {
        this.topic = topic;
    }

    @KafkaListener(topics = "#{__listener.topic}",
        groupId = "#{__listener.topic}.group")
    public void listen(...) {
        ...
    }

    public String getTopic() {
        return this.topic;
    }

}
```

If, in the unlikely event that you have an actual bean called `__listener`, you can change the expression token byusing the `beanRef` attribute. The following example shows how to do so:

```
@KafkaListener(beanRef = "__x", topics = "#{__x.topic}",
    groupId = "#{__x.topic}.group")
```

Starting with version 2.2.4, you can specify Kafka consumer properties directly on the annotation, these will override any properties with the same name configured in the consumer factory. You **cannot** specify the `group.id` and `client.id` properties this way; they will be ignored; use the `groupId` and `clientIdPrefix` annotation properties for those.

The properties are specified as individual strings with the normal Java `Properties` file format: `foo:bar`, `foo=bar`, or `foo bar`.

```
@KafkaListener(topics = "myTopic", groupId = "group", properties = {
    "max.poll.interval.ms:60000",
    ConsumerConfig.MAX_POLL_RECORDS_CONFIG + "=100"
})
```

The following is an example of the corresponding listeners for the example in Using `RoutingKafkaTemplate`.

```
@KafkaListener(id = "one", topics = "one")
public void listen1(String in) {
    System.out.println("1: " + in);
}

@KafkaListener(id = "two", topics = "two",
        properties =
"value.deserializer:org.apache.kafka.common.serialization.ByteArrayDeserializer")
public void listen2(byte[] in) {
    System.out.println("2: " + new String(in));
}
```

Obtaining the Consumer `group.id`

When running the same listener code in multiple containers, it may be useful to be able to determine which container (identified by its `group.id` consumer property) that a record came from.

You can call `KafkaUtils.getConsumerGroupId()` on the listener thread to do this. Alternatively, you can access the group id in a method parameter.

```
@KafkaListener(id = "bar", topicPattern = "${topicTwo:annotated2}", exposeGroupId
= "${always:true}")
public void listener(@Payload String foo,
        @Header(KafkaHeaders.GROUP_ID) String groupId) {
...
}
```

This is available in record listeners and batch listeners that receive a `List<?>` of records. It is **not** available in a batch listener that receives a `ConsumerRecords<?, ?>` argument. Use the `KafkaUtils` mechanism in that case.

Container Thread Naming

Listener containers currently use two task executors, one to invoke the consumer and another that is used to invoke the listener when the kafka consumer property `enable.auto.commit` is `false`. You can provide custom executors by setting the `consumerExecutor` and `listenerExecutor` properties of the container's `ContainerProperties`. When using pooled executors, be sure that enough threads are available to handle the concurrency across all the containers in which they are used. When using the `ConcurrentMessageListenerContainer`, a thread from each is used for each consumer (`concurrency`).

If you do not provide a consumer executor, a `SimpleAsyncTaskExecutor` is used. This executor creates threads with names similar to `<beanName>-C-1` (consumer thread). For the `ConcurrentMessageListenerContainer`, the `<beanName>` part of the thread name becomes `<beanName>-m`, where `m` represents the consumer instance. `n` increments each time the container is started. So, with a bean name of `container`, threads in this container will be named `container-0-C-1`, `container-1-C-1` etc., after the container is started the first time; `container-0-C-2`, `container-1-C-2` etc., after a stop and subsequent start.

`@KafkaListener` as a Meta Annotation

Starting with version 2.2, you can now use `@KafkaListener` as a meta annotation. The following example shows how to do so:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@KafkaListener
public @interface MyThreeConsumersListener {

    @AliasFor(annotation = KafkaListener.class, attribute = "id")
    String id();

    @AliasFor(annotation = KafkaListener.class, attribute = "topics")
    String[] topics();

    @AliasFor(annotation = KafkaListener.class, attribute = "concurrency")
    String concurrency() default "3";

}
```

You must alias at least one of `topics`, `topicPattern`, or `topicPartitions` (and, usually, `id` or `groupId` unless you have specified a `group.id` in the consumer factory configuration). The following example shows how to do so:

```
@MyThreeConsumersListener(id = "my.group", topics = "my.topic")
public void listen1(String in) {
    ...
}
```

`@KafkaListener` on a Class

When you use `@KafkaListener` at the class-level, you must specify `@KafkaHandler` at the method level. When messages are delivered, the converted message payload type is used to determine which method to call. The following example shows how to do so:

```
@KafkaListener(id = "multi", topics = "myTopic")
static class MultiListenerBean {

    @KafkaHandler
    public void listen(String foo) {
        ...
    }

    @KafkaHandler
    public void listen(Integer bar) {
        ...
    }

    @KafkaHandler(isDefault = true)
    public void listenDefault(Object object) {
        ...
    }

}
```

Starting with version 2.1.3, you can designate a `@KafkaHandler` method as the default method that is invoked if there is no match on other methods. At most, one method can be so designated. When using `@KafkaHandler` methods, the payload must have already been converted to the domain object (so the match can be performed). Use a custom deserializer, the `JsonDeserializer`, or the `JsonMessageConverter` with its `TypePrecedence` set to `TYPE_ID`. See Serialization, Deserialization, and Message Conversion for more information.

Due to some limitations in the way Spring resolves method arguments, a default `@KafkaHandler` cannot receive discrete headers; it must use the `ConsumerRecordMetadata` as discussed in Consumer Record Metadata.

For example:

```
@KafkaHandler(isDefault = true)
public void listenDefault(Object object, @Header(KafkaHeaders.RECEIVED_TOPIC)
String topic) {
    ...
}
```

This won't work if the object is a `String`; the `topic` parameter will also get a reference to `object`.

If you need metadata about the record in a default method, use this:

```
@KafkaHandler(isDefault = true)
void listen(Object in, @Header(KafkaHeaders.RECORD_METADATA)
ConsumerRecordMetadata meta) {
    String topic = meta.topic();
    ...
}
```

`@KafkaListener` Attribute Modification

Starting with version 2.7.2, you can now programmatically modify annotation attributes before the container is created. To do so, add one or more `KafkaListenerAnnotationBeanPostProcessor.AnnotationEnhancer` to the application context. `AnnotationEnhancer` is a `BiFunction<Map<String, Object>, AnnotatedElement, Map<String, Object>` and must return a map of attributes. The attribute values can contain SpEL and/or property placeholders; the enhancer is called before any resolution is performed. If more than one enhancer is present, and they implement `Ordered`, they will be invoked in order.

`AnnotationEnhancer` bean definitions must be declared `static` because they are required very early in the application context's lifecycle.

An example follows:

```
@Bean
public static AnnotationEnhancer groupIdEnhancer() {
    return (attrs, element) -> {
        attrs.put("groupId", attrs.get("id") + "." + (element instanceof Class
                ? ((Class<?>) element).getSimpleName()
                : ((Method) element).getDeclaringClass().getSimpleName()
                        +  "." + ((Method) element).getName()));
        return attrs;
    };
}
```

`@KafkaListener` Lifecycle Management

The listener containers created for `@KafkaListener` annotations are not beans in the application context. Instead, they are registered with an infrastructure bean of type `KafkaListenerEndpointRegistry`. This bean is automatically declared by the framework and manages the containers' lifecycles; it will auto-start any containers that have `autoStartup` set to `true`. All containers created by all container factories must be in the same `phase`. See Listener Container Auto Startup for more information. You can manage the lifecycle programmatically by using the registry. Starting or stopping the registry will start or stop all the registered containers. Alternatively, you can get a reference to an individual container by using its `id` attribute. You can set `autoStartup` on the annotation, which overrides the default setting configured into the container factory. You can get a reference to the bean from the application context, such as auto-wiring, to manage its registered containers. The following examples show how to do so:

```
@KafkaListener(id = "myContainer", topics = "myTopic", autoStartup = "false")
public void listen(...) { ... }
```

```
@Autowired
private KafkaListenerEndpointRegistry registry;

...

    this.registry.getListenerContainer("myContainer").start();

...
```

The registry only maintains the life cycle of containers it manages; containers declared as beans are not managed by the registry and can be obtained from the application context. A collection of managed containers can be obtained by calling the registry's `getListenerContainers()` method. Version 2.2.5 added a convenience method `getAllListenerContainers()`, which returns a collection of all containers, including those managed by the registry and those declared as beans. The collection returned will include any prototype beans that have been initialized, but it will not

initialize any lazy bean declarations.

`@KafkaListener` `@Payload` Validation

Starting with version 2.2, it is now easier to add a `Validator` to validate `@KafkaListener` `@Payload` arguments. Previously, you had to configure a custom `DefaultMessageHandlerMethodFactory` and add it to the registrar. Now, you can add the validator to the registrar itself. The following code shows how to do so:

```
@Configuration
@EnableKafka
public class Config implements KafkaListenerConfigurer {

    ...

    @Override
    public void configureKafkaListeners(KafkaListenerEndpointRegistrar registrar)
{
      registrar.setValidator(new MyValidator());
    }

}
```

When you use Spring Boot with the validation starter, a `LocalValidatorFactoryBean` is auto-configured, as the following example shows:

```
@Configuration
@EnableKafka
public class Config implements KafkaListenerConfigurer {

    @Autowired
    private LocalValidatorFactoryBean validator;
    ...

    @Override
    public void configureKafkaListeners(KafkaListenerEndpointRegistrar registrar)
{
      registrar.setValidator(this.validator);
    }
}
```

The following examples show how to validate:

```
public static class ValidatedClass {

  @Max(10)
  private int bar;

  public int getBar() {
    return this.bar;
  }

  public void setBar(int bar) {
    this.bar = bar;
  }

}
```

```
@KafkaListener(id="validated", topics = "annotated35", errorHandler =
"validationErrorHandler",
      containerFactory = "kafkaJsonListenerContainerFactory")
public void validatedListener(@Payload @Valid ValidatedClass val) {
    ...
}

@Bean
public KafkaListenerErrorHandler validationErrorHandler() {
    return (m, e) -> {
        ...
    };
}
```

Starting with version 2.5.11, validation now works on payloads for `@KafkaHandler` methods in a class-level listener. See `@KafkaListener` on a Class.

Rebalancing Listeners

`ContainerProperties` has a property called `consumerRebalanceListener`, which takes an implementation of the Kafka client's `ConsumerRebalanceListener` interface. If this property is not provided, the container configures a logging listener that logs rebalance events at the `INFO` level. The framework also adds a sub-interface `ConsumerAwareRebalanceListener`. The following listing shows the `ConsumerAwareRebalanceListener` interface definition:

```
public interface ConsumerAwareRebalanceListener extends ConsumerRebalanceListener
{

    void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer, Collection
<TopicPartition> partitions);

    void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collection
<TopicPartition> partitions);

    void onPartitionsAssigned(Consumer<?, ?> consumer, Collection<TopicPartition>
partitions);

    void onPartitionsLost(Consumer<?, ?> consumer, Collection<TopicPartition>
partitions);

}
```

Notice that there are two callbacks when partitions are revoked. The first is called immediately. The second is called after any pending offsets are committed. This is useful if you wish to maintain offsets in some external repository, as the following example shows:

```
containerProperties.setConsumerRebalanceListener(new
ConsumerAwareRebalanceListener() {

    @Override
    public void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer,
Collection<TopicPartition> partitions) {
        // acknowledge any pending Acknowledgments (if using manual acks)
    }

    @Override
    public void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer,
Collection<TopicPartition> partitions) {
        // ...
            store(consumer.position(partition));
        // ...
    }

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        // ...
            consumer.seek(partition, offsetTracker.getOffset() + 1);
        // ...
    }
});
```

Starting with version 2.4, a new method `onPartitionsLost()` has been added (similar to a method with the same name in `ConsumerRebalanceLister`). The default implementation on `ConsumerRebalanceLister` simply calls `onPartionsRevoked`. The default implementation on `ConsumerAwareRebalanceListener` does nothing. When supplying the listener container with a custom listener (of either type), it is important that your implementation not call `onPartitionsRevoked` from `onPartitionsLost`. If you implement `ConsumerRebalanceListener` you should override the default method. This is because the listener container will call its own `onPartitionsRevoked` from its implementation of `onPartitionsLost` after calling the method on your implementation. If you implementation delegates to the default behavior, `onPartitionsRevoked` will be called twice each time the `Consumer` calls that method on the container's listener.

Forwarding Listener Results using `@SendTo`

Starting with version 2.0, if you also annotate a `@KafkaListener` with a `@SendTo` annotation and the method invocation returns a result, the result is forwarded to the topic specified by the `@SendTo`.

The `@SendTo` value can have several forms:

- `@SendTo("someTopic")` routes to the literal topic
- `@SendTo("#{someExpression}")` routes to the topic determined by evaluating the expression once during application context initialization.
- `@SendTo("!{someExpression}")` routes to the topic determined by evaluating the expression at runtime. The `#root` object for the evaluation has three properties:
  - `request`: The inbound `ConsumerRecord` (or `ConsumerRecords` object for a batch listener))
  - `source`: The `org.springframework.messaging.Message<?>` converted from the `request`.
  - `result`: The method return result.
- `@SendTo` (no properties): This is treated as `!{source.headers['kafka_replyTopic']}` (since version 2.1.3).

Starting with versions 2.1.11 and 2.2.1, property placeholders are resolved within `@SendTo` values.

The result of the expression evaluation must be a `String` that represents the topic name. The following examples show the various ways to use `@SendTo`:

```
@KafkaListener(topics = "annotated21")
@SendTo("!{request.value()}") // runtime SpEL
public String replyingListener(String in) {
    ...
}

@KafkaListener(topics = "${some.property:annotated22}")
@SendTo("#{myBean.replyTopic}") // config time SpEL
public Collection<String> replyingBatchListener(List<String> in) {
    ...
}

@KafkaListener(topics = "annotated23", errorHandler = "replyErrorHandler")
@SendTo("annotated23reply") // static reply topic definition
public String replyingListenerWithErrorHandler(String in) {
    ...
}
...
@KafkaListener(topics = "annotated25")
@SendTo("annotated25reply1")
public class MultiListenerSendTo {

    @KafkaHandler
    public String foo(String in) {
        ...
    }

    @KafkaHandler
    @SendTo("!{'annotated25reply2'}")
    public String bar(@Payload(required = false) KafkaNull nul,
            @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) int key) {
        ...
    }

}
```

In order to support `@SendTo`, the listener container factory must be provided with a `KafkaTemplate` (in its `replyTemplate` property), which is used to send the reply. This should be a `KafkaTemplate` and not a `ReplyingKafkaTemplate` which is used on the client-side for request/reply processing. When using Spring Boot, boot will auto-configure the template into the factory; when configuring your own factory, it must be set as shown in the examples below.

Starting with version 2.2, you can add a `ReplyHeadersConfigurer` to the listener container factory. This is consulted to determine which headers you want to set in the reply message. The following example shows how to add a `ReplyHeadersConfigurer`:

```
@Bean
public ConcurrentKafkaListenerContainerFactory<Integer, String>
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(cf());
    factory.setReplyTemplate(template());
    factory.setReplyHeadersConfigurer((k, v) -> k.equals("cat"));
    return factory;
}
```

You can also add more headers if you wish. The following example shows how to do so:

```
@Bean
public ConcurrentKafkaListenerContainerFactory<Integer, String>
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(cf());
    factory.setReplyTemplate(template());
    factory.setReplyHeadersConfigurer(new ReplyHeadersConfigurer() {

      @Override
      public boolean shouldCopy(String headerName, Object headerValue) {
        return false;
      }

      @Override
      public Map<String, Object> additionalHeaders() {
        return Collections.singletonMap("qux", "fiz");
      }

    });
    return factory;
}
```

When you use `@SendTo`, you must configure the `ConcurrentKafkaListenerContainerFactory` with a `KafkaTemplate` in its `replyTemplate` property to perform the send.

Unless you use request/reply semantics only the simple `send(topic, value)` method is used, so you may wish to create a subclass to generate the partition or key. The following example shows how to do so:

```
@Bean
public KafkaTemplate<String, String> myReplyingTemplate() {
    return new KafkaTemplate<Integer, String>(producerFactory()) {

        @Override
        public ListenableFuture<SendResult<String, String>> send(String topic,
String data) {
            return super.send(topic, partitionForData(data), keyForData(data),
data);
        }

        ...

    };
}
```

If the listener method returns `Message<?>` or `Collection<Message<?>>`, the listener method is responsible for setting up the message headers for the reply. For example, when handling a request from a `ReplyingKafkaTemplate`, you might do the following:

```
@KafkaListener(id = "messageReturned", topics = "someTopic")
public Message<?> listen(String in, @Header(KafkaHeaders
.REPLY_TOPIC) byte[] replyTo,
        @Header(KafkaHeaders.CORRELATION_ID) byte[] correlation) {
    return MessageBuilder.withPayload(in.toUpperCase())
            .setHeader(KafkaHeaders.TOPIC, replyTo)
            .setHeader(KafkaHeaders.MESSAGE_KEY, 42)
            .setHeader(KafkaHeaders.CORRELATION_ID, correlation)
            .setHeader("someOtherHeader", "someValue")
            .build();
}
```

When using request/reply semantics, the target partition can be requested by the sender.

You can annotate a `@KafkaListener` method with `@SendTo` even if no result is returned. This is to allow the configuration of an `errorHandler` that can forward information about a failed message delivery to some topic. The following example shows how to do so:

```
@KafkaListener(id = "voidListenerWithReplyingErrorHandler", topics
= "someTopic",
        errorHandler = "voidSendToErrorHandler")
@SendTo("failures")
public void voidListenerWithReplyingErrorHandler(String in) {
    throw new RuntimeException("fail");
}

@Bean
public KafkaListenerErrorHandler voidSendToErrorHandler() {
    return (m, e) -> {
        return ... // some information about the failure and input
data
    };
}
```

See Handling Exceptions for more information.

If a listener method returns an `Iterable`, by default a record for each element as the value is sent. Starting with version 2.3.5, set the `splitIterables` property on `@KafkaListener` to `false` and the entire result will be sent as the value of a single `ProducerRecord`. This requires a suitable serializer in the reply template's producer configuration. However, if the reply is `Iterable<Message<?>>` the property is ignored and each message is sent separately.

Filtering Messages

In certain scenarios, such as rebalancing, a message that has already been processed may be redelivered. The framework cannot know whether such a message has been processed or not. That is an application-level function. This is known as the Idempotent Receiver pattern and Spring Integration provides an implementation of it.

The Spring for Apache Kafka project also provides some assistance by means of the `FilteringMessageListenerAdapter` class, which can wrap your `MessageListener`. This class takes an implementation of `RecordFilterStrategy` in which you implement the `filter` method to signal that a message is a duplicate and should be discarded. This has an additional property called `ackDiscarded`, which indicates whether the adapter should acknowledge the discarded record. It is `false` by default.

When you use `@KafkaListener`, set the `RecordFilterStrategy` (and optionally `ackDiscarded`) on the container factory so that the listener is wrapped in the appropriate filtering adapter.

In addition, a `FilteringBatchMessageListenerAdapter` is provided, for when you use a batch message listener.

The `FilteringBatchMessageListenerAdapter` is ignored if your `@KafkaListener` receives a `ConsumerRecords<?, ?>` instead of `List<ConsumerRecord<?, ?>>`, because `ConsumerRecords` is immutable.

Retrying Deliveries

If your listener throws an exception, the default behavior is to invoke the Container Error Handlers, if configured, or logged otherwise.

NOTE: To retry deliveries, a convenient listener adapter `RetryingMessageListenerAdapter` is provided.

You can configure it with a `RetryTemplate` and `RecoveryCallback<Void>` - see the spring-retry project for information about these components. If a recovery callback is not provided, the exception is thrown to the container after retries are exhausted. In that case, the `ErrorHandler` is invoked, if configured, or logged otherwise.

When you use `@KafkaListener`, you can set the `RetryTemplate` (and optionally `recoveryCallback`) on the container factory. When you do so, the listener is wrapped in the appropriate retrying adapter.

The contents of the `RetryContext` passed into the `RecoveryCallback` depend on the type of listener. The context always has a `record` attribute, which is the record for which the failure occurred. If your listener is acknowledging or consumer aware, additional `acknowledgment` or `consumer` attributes are available. For convenience, the `RetryingMessageListenerAdapter` provides static constants for these keys. See its Javadoc for more information.

A retry adapter is not provided for any of the batch message listeners, because the framework has no knowledge of where in a batch the failure occurred. If you need retry capabilities when you use a batch listener, we recommend that you use a `RetryTemplate` within the listener itself.

Stateful Retry

Now that the `SeekToCurrentErrorHandler` can be configured with a `BackOff` and has the ability to retry only certain exceptions (since version 2.3), the use of stateful retry, via the listener adapter retry configuration, is no longer necessary. You can provide the same functionality with appropriate configuration of the error handler and remove all retry configuration from the listener adapter. See Seek To Current Container Error Handlers for more information.

You should understand that the retry discussed in the preceding section suspends the consumer thread (if a `BackOffPolicy` is used). There are no calls to `Consumer.poll()` during the retries. Kafka has two properties to determine consumer health. The `session.timeout.ms` is used to determine if the consumer is active. Since `kafka-clients` version `0.10.1.0`, heartbeats are sent on a background thread, so a slow consumer no longer affects that. `max.poll.interval.ms` (default: five minutes) is used to determine if a consumer appears to be hung (taking too long to process records from the last poll). If the time between `poll()` calls exceeds this, the broker revokes the assigned partitions

and performs a rebalance. For lengthy retry sequences, with back off, this can easily happen.

Since version 2.1.3, you can avoid this problem by using stateful retry in conjunction with a `SeekToCurrentErrorHandler`. In this case, each delivery attempt throws the exception back to the container, the error handler re-seeks the unprocessed offsets, and the same message is redelivered by the next `poll()`. This avoids the problem of exceeding the `max.poll.interval.ms` property (as long as an individual delay between attempts does not exceed it). So, when you use an `ExponentialBackOffPolicy`, you must ensure that the `maxInterval` is less than the `max.poll.interval.ms` property. To enable stateful retry, you can use the `RetryingMessageListenerAdapter` constructor that takes a `stateful boolean` argument (set it to `true`). When you configure the listener container factory (for `@KafkaListener`), set the factory's `statefulRetry` property to `true`.

Version 2.2 added recovery to the `SeekToCurrentErrorHandler`, such as sending a failed record to a dead-letter topic. When using stateful retry, you must perform the recovery in the retry `RecoveryCallback` and NOT in the error handler. Otherwise, if the recovery is done in the error handler, the retry template's state will never be cleared. Also, you must ensure that the `maxFailures` in the `SeekToCurrentErrorHandler` must be at least as many as configured in the retry policy, again to ensure that the retries are exhausted and the state cleared. Here is an example for retry configuration when used with a `SeekToCurrentErrorHandler` where `factory` is the `ConcurrentKafkaListenerContainerFactory`.

```
@Autowired
DeadLetterPublishingRecoverer recoverer;

...
    factory.setRetryTemplate(new RetryTemplate()); // 3 retries by default
    factory.setStatefulRetry(true);
    factory.setRecoveryCallback(context -> {
        recoverer.accept((ConsumerRecord<?, ?>) context.getAttribute("record"),
                (Exception) context.getLastThrowable());
        return null;
    });
...

@Bean
public SeekToCurrentErrorHandler eh() {
    return new SeekToCurrentErrorHandler(new FixedBackOff(0L, 3L)); // at least 3
}
```

However, see the note at the beginning of this section; you can avoid using the `RetryTemplate` altogether.

If the recoverer fails (throws an exception), the failed record will be included in the seeks. Starting with version 2.5.5, if the recoverer fails, the `BackOff` will be reset by default and redeliveries will again go through the back offs before recovery is attempted again. With earlier versions, the `BackOff` was not reset and recovery was re-attempted on the next failure. To revert to the previous behavior, set the error handler's `resetStateOnRecoveryFailure` to `false`.

Starting with version 2.6, you can now provide the error handler with a `BiFunction<ConsumerRecord<?, ?>, Exception, BackOff>` to determine the `BackOff` to use, based on the failed record and/or the exception:

```
handler.setBackOffFunction((record, ex) -> { ... });
```

If the function returns `null`, the handler's default `BackOff` will be used.

Starting with version 2.6.3, set `resetStateOnExceptionChange` to `true` and the retry sequence will be restarted (including the selection of a new `BackOff`, if so configured) if the exception type changes between failures. By default, the exception type is not considered.

Starting `@KafkaListener` s in Sequence

A common use case is to start a listener after another listener has consumed all the records in a topic. For example, you may want to load the contents of one or more compacted topics into memory before processing records from other topics. Starting with version 2.7.3, a new component `ContainerGroupSequencer` has been introduced. It uses the `@KafkaListener containerGroup` property to group containers together and start the containers in the next group, when all the containers in the current group have gone idle.

It is best illustrated with an example.

```
@KafkaListener(id = "listen1", topics = "topic1", containerGroup = "g1",
concurrency = "2")
public void listen1(String in) {
}

@KafkaListener(id = "listen2", topics = "topic2", containerGroup = "g1",
concurrency = "2")
public void listen2(String in) {
}

@KafkaListener(id = "listen3", topics = "topic3", containerGroup = "g2",
concurrency = "2")
public void listen3(String in) {
}

@KafkaListener(id = "listen4", topics = "topic4", containerGroup = "g2",
concurrency = "2")
public void listen4(String in) {
}

@Bean
ContainerGroupSequencer sequencer(KafkaListenerEndpointRegistry registry) {
    return new ContainerGroupSequencer(registry, 5000, "g1", "g2");
}
```

Here, we have 4 listeners in two groups, `g1` and `g2`.

During application context initialization, the sequencer, sets the `autoStartup` property of all the containers in the provided groups to `false`. It also sets the `idleEventInterval` for any containers (that do not already have one set) to the supplied value (5000ms in this case). Then, when the sequencer is started by the application context, the containers in the first group are started. As `ListenerContainerIdleEvent` s are received, each individual child container in each container is stopped. When all child containers in a `ConcurrentMessageListenerContainer` are stopped, the parent container is stopped. When all containers in a group have been stopped, the containers in the next group are started. There is no limit to the number of groups or containers in a group.

By default, the containers in the final group (`g2` above) are not stopped when they go idle. To modify that behavior, set `stopLastGroupWhenIdle` to `true` on the sequencer.

As an aside; previously, containers in each group were added to a bean of type `Collection<MessageListenerContainer>` with the bean name being the `containerGroup`. These collections are now deprecated in favor of beans of type `ContainerGroup` with a bean name that is the group name, suffixed with `.group`; in the example above, there would be 2 beans `g1.group` and `g2.group`. The `Collection` beans will be removed in a future release.

4.1.5. Listener Container Properties

Table 1. `ContainerProperties` Properties

Property	Default	Description
ackCount	1	The number of records before committing pending offsets when the `ackMode` is `COUNT` or `COUNT_TIME`.
adviceChain	`null`	A chain of `Advice` objects (e.g. `MethodInterceptor` around advice) wrapping the message listener, invoked in order.
ackMode	BATCH	Controls how often offsets are committed - see Committing Offsets.
ackOnError	`false`	[DEPRECATED in favor of `ErrorHandler.isAckAfterHandle()`]
ackTime	5000	The time in milliseconds after which pending offsets are committed when the `ackMode` is `TIME` or `COUNT_TIME`.
assignmentCommitOption	LATEST_ONLY_NO_TX	Whether or not to commit the initial position on assignment; by default, the initial offset will only be committed if the `ConsumerConfig.AUTO_OFFSET_RESET_CONFIG` is `latest` and it won't run in a transaction even if there is a transaction manager present. See the javadocs for `ContainerProperties.AssignmentCommitOption` for more information about the available options.
authorizationExceptionRetryInterval	`null`	When not null, a `Duration` to sleep between polls when an `AuthorizationException` is thrown by the Kafka client. When null, such exceptions are considered fatal and the container will stop.
clientId	(empty string)	A prefix for the `client.id` consumer property. Overrides the consumer factory `client.id` property; in a concurrent container, `-n` is added as a suffix for each consumer instance.
commitCallback	`null`	When present and `syncCommits` is `false` a callback invoked after the commit completes.
commitLogLevel	DEBUG	The logging level for logs pertaining to committing offsets.
consumerRebalanceListener	`null`	A rebalance listener; see Rebalancing Listeners.
consumerStartTimout	30s	The time to wait for the consumer to start before logging an error; this might happen if, say, you use a task executor with insufficient threads.
consumerTaskExecutor	`SimpleAsyncTaskExecutor`	A task executor to run the consumer threads. The default executor creates threads named `<name>-C-n`; with the `KafkaMessageListenerContainer`, the name is the bean name; with the `ConcurrentMessageListenerContainer` the name is the bean name suffixed with `-n` where n is incremented for each child container.

Property	Default	Description
deliveryAttemptHeader	false	See Delivery Attempts Header.
eosMode	BETA	Exactly Once Semantics mode; see Exactly Once Semantics.
fixTxOffsets	false	When consuming records produced by a transactional producer, and the consumer is positioned at the end of a partition, the lag can incorrectly be reported as greater than zero, due to the pseudo record used to indicate transaction commit/rollback and, possibly, the presence of rolled-back records. This does not functionally affect the consumer but some users have expressed concern that the "lag" is non-zero. Set this property to true and the container will correct such mis-reported offsets. The check is performed before the next poll to avoid adding significant complexity to the commit processing. At the time of writing, the lag will only be corrected if the consumer is configured with isolation.level=read_committed and max.poll.records is greater than 1. See KAFKA-10683 for more information.
groupId	null	Overrides the consumer group.id property; automatically set by the @KafkaListener id or groupId property.
idleBetweenPolls	0	Used to slow down deliveries by sleeping the thread between polls. The time to process a batch of records plus this value must be less than the max.poll.interval.ms consumer property.
idleEventInterval	null	When set, enables publication of ListenerContainerIdleEvent s, see Application Events and Detecting Idle and Non-Responsive Consumers.
idlePartitionEventInterval	null	When set, enables publication of ListenerContainerIdlePartitionEvent s, see Application Events and Detecting Idle and Non-Responsive Consumers.
kafkaConsumerProperties	None	Used to override any arbitrary consumer properties configured on the consumer factory.
logContainerConfig	false	Set to true to log at INFO level all container properties.
messageListener	null	The message listener.
micrometerEnabled	true	Whether or not to maintain Micrometer timers for the consumer threads.
missingTopicsFatal	false	When true prevents the container from starting if the confifgured topic(s) are not present on the broker.
monitorInterval	30s	How often to check the state of the consumer threads for NonResponsiveConsumerEvent s. See noPollThreshold and pollTimeout.
noPollThreshold	3.0	Multiplied by pollTimeOut to determine whether to publish a NonResponsiveConsumerEvent. See monitorInterval.

Property	Default	Description
onlyLogRecordMetadata	false	Set to false to log the complete consumer record (in error, debug logs etc) instead of just topic-partition@offset.
pollTimeout	5000	The timeout passed into Consumer.poll().
scheduler	ThreadPoolTaskScheduler	A scheduler on which to run the consumer monitor task.
shutdownTimeout	10000	The maximum time in ms to block the stop() method until all consumers stop and before publishing the container stopped event.
stopContainerWhenFenced	false	Stop the listener container if a ProducerFencedException is thrown. See After-rollback Processor for more information.
stopImmediate	false	When the container is stopped, stop processing after the current record instead of after processing all the records from the previous poll.
subBatchPerPartition	See desc.	When using a batch listener, if this is true, the listener is called with the results of the poll split into sub batches, one per partition. Default false except when using transactions with EOSMode.ALPHA - see Exactly Once Semantics.
syncCommitTimeout	null	The timeout to use when syncCommits is true. When not set, the container will attempt to determine the default.api.timeout.ms consumer property and use that; otherwise it will use 60 seconds.
syncCommits	true	Whether to use sync or async commits for offsets; see commitCallback.
topics topicPattern topicPartitions	n/a	The configured topics, topic pattern or explicitly assigned topics/partitions. Mutually exclusive; at least one must be provided; enforced by ContainerProperties constructors.
transactionManager	null	See Transactions.

Table 2. AbstractListenerContainer Properties

Property	Default	Description
afterRollbackProcessor	DefaultAfterRollbackProcessor	An AfterRollbackProcessor to invoke after a transaction is rolled back.
applicationEventPublisher	application context	The event publisher.
batchErrorHandler	See desc.	An error handler for a batch listener; defaults to a RecoveringBatchErrorHandler or null if transactions are being used (errors are handled by the AfterRollbackProcessor).

Property	Default	Description
batch Interceptor	null	Set a BatchInterceptor to call before invoking the batch listener; does not apply to record listeners. Also see interceptBeforeTx.
beanName	bean name	The bean name of the container; suffixed with -n for child containers.
containerProperties	Container Properties	The container properties instance.
errorHandler	See desc.	An error handler for a record listener; defaults to a SeekToCurrentErrorHandler or null if transactions are being used (errors are handled by the AfterRollbackProcessor).
genericErrorHandler	See desc.	Either a batch or record error handler - see batchErrorHandler and errorHandler.
groupId	See desc.	The containerProperties.groupId, if present, otherwise the group.id property from the consumer factory.
intercept BeforeTx	false	Determines whether the recordInterceptor is called before or after a transaction starts.
listenerId	See desc.	The bean name for user-configured containers or the id attribute of @KafkaListener s.
pause Requested	(read only)	True if a consumer pause has been requested.
record Interceptor	null	Set a RecordInterceptor to call before invoking the record listener; does not apply to batch listeners. Also see interceptBeforeTx.
topicCheck Timeout	30s	When the missingTopicsFatal container property is true, how long to wait, in seconds, for the describeTopics operation to complete.

Table 3. KafkaMessageListenerContainer *Properties*

Property	Default	Description
assigned Partitions	(read only)	The partitions currently assigned to this container (explicitly or not).
assigned Partitions ByClientId	(read only)	The partitions currently assigned to this container (explicitly or not).
clientId Suffix	null	Used by the concurrent container to give each child container's consumer a unique client.id.
containerPaused	n/a	True if pause has been requested and the consumer has actually paused.

Table 4. ConcurrentMessageListenerContainer *Properties*

Property	Default	Description
alwaysClientIdSuffix	true	Set to false to suppress adding a suffix to the client.id consumer property, when the concurrency is only 1.
assignedPartitions	(read only)	The aggregate of partitions currently assigned to this container's child KafkaMessageListenerContainer s (explicitly or not).
assignedPartitionsByClientId	(read only)	The partitions currently assigned to this container's child KafkaMessageListenerContainer s (explicitly or not), keyed by the child container's consumer's client.id property.
concurrency	1	The number of child KafkaMessageListenerContainer s to manage.
containerPaused	n/a	True if pause has been requested and all child containers' consumer has actually paused.
containers	n/a	A reference to all child KafkaMessageListenerContainer s.

4.1.6. Application Events

The following Spring application events are published by listener containers and their consumers:

- ConsumerStartingEvent - published when a consumer thread is first started, before it starts polling.
- ConsumerStartedEvent - published when a consumer is about to start polling.
- ConsumerFailedToStartEvent - published if no ConsumerStartingEvent is published within the consumerStartTimeout container property. This event might signal that the configured task executor has insufficient threads to support the containers it is used in and their concurrency. An error message is also logged when this condition occurs.
- ListenerContainerIdleEvent: published when no messages have been received in idleInterval (if configured).
- ListenerContainerNoLongerIdleEvent: published when a record is consumed after previously publishing a ListenerContainerIdleEvent.
- ListenerContainerPartitionIdleEvent: published when no messages have been received from that partition in idlePartitionEventInterval (if configured).
- ListenerContainerPartitionNoLongerIdleEvent: published when a record is consumed from a partition that has previously published a ListenerContainerPartitionIdleEvent.
- NonResponsiveConsumerEvent: published when the consumer appears to be blocked in the poll method.
- ConsumerPartitionPausedEvent: published by each consumer when a partition is paused.
- ConsumerPartitionResumedEvent: published by each consumer when a partition is resumed.
- ConsumerPausedEvent: published by each consumer when the container is paused.
- ConsumerResumedEvent: published by each consumer when the container is resumed.
- ConsumerStoppingEvent: published by each consumer just before stopping.

- `ConsumerStoppedEvent`: published after the consumer is closed. See Thread Safety.
- `ContainerStoppedEvent`: published when all consumers have stopped.

By default, the application context's event multicaster invokes event listeners on the calling thread. If you change the multicaster to use an async executor, you must not invoke any `Consumer` methods when the event contains a reference to the consumer.

The `ListenerContainerIdleEvent` has the following properties:

- `source`: The listener container instance that published the event.
- `container`: The listener container or the parent listener container, if the source container is a child.
- `id`: The listener ID (or container bean name).
- `idleTime`: The time the container had been idle when the event was published.
- `topicPartitions`: The topics and partitions that the container was assigned at the time the event was generated.
- `consumer`: A reference to the Kafka `Consumer` object. For example, if the consumer's `pause()` method was previously called, it can `resume()` when the event is received.
- `paused`: Whether the container is currently paused. See Pausing and Resuming Listener Containers for more information.

The `ListenerContainerNoLongerIdleEvent` has the same properties, except `idleTime` and `paused`.

The `ListenerContainerPartitionIdleEvent` has the following properties:

- `source`: The listener container instance that published the event.
- `container`: The listener container or the parent listener container, if the source container is a child.
- `id`: The listener ID (or container bean name).
- `idleTime`: The time partition consumption had been idle when the event was published.
- `topicPartition`: The topic and partition that triggered the event.
- `consumer`: A reference to the Kafka `Consumer` object. For example, if the consumer's `pause()` method was previously called, it can `resume()` when the event is received.
- `paused`: Whether that partition consumption is currently paused for that consumer. See Pausing and Resuming Listener Containers for more information.

The `ListenerContainerPartitionNoLongerIdleEvent` has the same properties, except `idleTime` and `paused`.

The `NonResponsiveConsumerEvent` has the following properties:

- `source`: The listener container instance that published the event.
- `container`: The listener container or the parent listener container, if the source container is a

child.

- `id`: The listener ID (or container bean name).
- `timeSinceLastPoll`: The time just before the container last called `poll()`.
- `topicPartitions`: The topics and partitions that the container was assigned at the time the event was generated.
- `consumer`: A reference to the Kafka `Consumer` object. For example, if the consumer's `pause()` method was previously called, it can `resume()` when the event is received.
- `paused`: Whether the container is currently paused. See Pausing and Resuming Listener Containers for more information.

The `ConsumerPausedEvent`, `ConsumerResumedEvent`, and `ConsumerStopping` events have the following properties:

- `source`: The listener container instance that published the event.
- `container`: The listener container or the parent listener container, if the source container is a child.
- `partitions`: The `TopicPartition` instances involved.

The `ConsumerPartitionPausedEvent`, `ConsumerPartitionResumedEvent` events have the following properties:

- `source`: The listener container instance that published the event.
- `container`: The listener container or the parent listener container, if the source container is a child.
- `partition`: The `TopicPartition` instance involved.

The `ConsumerStartingEvent`, `ConsumerStartingEvent`, `ConsumerFailedToStartEvent`, `ConsumerStoppedEvent` and `ContainerStoppedEvent` events have the following properties:

- `source`: The listener container instance that published the event.
- `container`: The listener container or the parent listener container, if the source container is a child.

All containers (whether a child or a parent) publish `ContainerStoppedEvent`. For a parent container, the source and container properties are identical.

In addition, the `ConsumerStoppedEvent` has the following additional property:

- `reason`
  - `NORMAL` - the consumer stopped normally (container was stopped).
  - `ERROR` - a `java.lang.Error` was thrown.
  - `FENCED` - the transactional producer was fenced and the `stopContainerWhenFenced` container property is `true`.
  - `AUTH` - an `AuthorizationException` was thrown and the `authorizationExceptionRetryInterval` is

not configured.

- `NO_OFFSET` - there is no offset for a partition and the `auto.offset.reset` policy is `none`.

You can use this event to restart the container after such a condition:

```
if (event.getReason.equals(Reason.FENCED)) {
    event.getSource(MessageListenerContainer.class).start();
}
```

Detecting Idle and Non-Responsive Consumers

While efficient, one problem with asynchronous consumers is detecting when they are idle. You might want to take some action if no messages arrive for some period of time.

You can configure the listener container to publish a `ListenerContainerIdleEvent` when some time passes with no message delivery. While the container is idle, an event is published every `idleEventInterval` milliseconds.

To configure this feature, set the `idleEventInterval` on the container. The following example shows how to do so:

```
@Bean
public KafkaMessageListenerContainer(ConsumerFactory<String, String>
consumerFactory) {
    ContainerProperties containerProps = new ContainerProperties("topic1", "
topic2");
    ...
    containerProps.setIdleEventInterval(60000L);
    ...
    KafkaMessageListenerContainer<String, String> container = new
KafKaMessageListenerContainer<>(...);
    return container;
}
```

The following example shows how to set the `idleEventInterval` for a `@KafkaListener`:

```
@Bean
public ConcurrentKafkaListenerContainerFactory kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
    ...
    factory.getContainerProperties().setIdleEventInterval(60000L);
    ...
    return factory;
}
```

In each of these cases, an event is published once per minute while the container is idle.

In addition, if the broker is unreachable, the consumer `poll()` method does not exit, so no messages are received and idle events cannot be generated. To solve this issue, the container publishes a `NonResponsiveConsumerEvent` if a poll does not return within `3x` the `pollTimeout` property. By default, this check is performed once every 30 seconds in each container. You can modify this behavior by setting the `monitorInterval` (default 30 seconds) and `noPollThreshold` (default 3.0) properties in the `ContainerProperties` when configuring the listener container. The `noPollThreshold` should be greater than `1.0` to avoid getting spurious events due to a race condition. Receiving such an event lets you stop the containers, thus waking the consumer so that it can stop.

Starting with version 2.6.2, if a container has published a `ListenerContainerIdleEvent`, it will publish a `ListenerContainerNoLongerIdleEvent` when a record is subsequently received.

Event Consumption

You can capture these events by implementing `ApplicationListener` — either a general listener or one narrowed to only receive this specific event. You can also use `@EventListener`, introduced in Spring Framework 4.2.

The next example combines `@KafkaListener` and `@EventListener` into a single class. You should understand that the application listener gets events for all containers, so you may need to check the listener ID if you want to take specific action based on which container is idle. You can also use the `@EventListener condition` for this purpose.

See Application Events for information about event properties.

The event is normally published on the consumer thread, so it is safe to interact with the `Consumer` object.

The following example uses both `@KafkaListener` and `@EventListener`:

```
public class Listener {

    @KafkaListener(id = "qux", topics = "annotated")
    public void listen4(@Payload String foo, Acknowledgment ack) {
        ...
    }

    @EventListener(condition = "event.listenerId.startsWith('qux-')")
    public void eventHandler(ListenerContainerIdleEvent event) {
        ...
    }

}
```

Event listeners see events for all containers. Consequently, in the preceding example, we narrow the events received based on the listener ID. Since containers created for the `@KafkaListener` support concurrency, the actual containers are named `id-n` where the `n` is a unique value for each instance to support the concurrency. That is why we use `startsWith` in the condition.

If you wish to use the idle event to stop the lister container, you should not call `container.stop()` on the thread that calls the listener. Doing so causes delays and unnecessary log messages. Instead, you should hand off the event to a different thread that can then stop the container. Also, you should not `stop()` the container instance if it is a child container. You should stop the concurrent container instead.

Current Positions when Idle

Note that you can obtain the current positions when idle is detected by implementing `ConsumerSeekAware` in your listener. See `onIdleContainer()` in Seeking to a Specific Offset.

4.1.7. Topic/Partition Initial Offset

There are several ways to set the initial offset for a partition.

When manually assigning partitions, you can set the initial offset (if desired) in the configured `TopicPartitionOffset` arguments (see Message Listener Containers). You can also seek to a specific offset at any time.

When you use group management where the broker assigns partitions:

- For a new `group.id`, the initial offset is determined by the `auto.offset.reset` consumer property (`earliest` or `latest`).
- For an existing group ID, the initial offset is the current offset for that group ID. You can, however, seek to a specific offset during initialization (or at any time thereafter).

4.1.8. Seeking to a Specific Offset

In order to seek, your listener must implement `ConsumerSeekAware`, which has the following methods:

```
void registerSeekCallback(ConsumerSeekCallback callback);

void onPartitionsAssigned(Map<TopicPartition, Long> assignments,
ConsumerSeekCallback callback);

void onPartitionsRevoked(Collection<TopicPartition> partitions)

void onIdleContainer(Map<TopicPartition, Long> assignments, ConsumerSeekCallback
callback);
```

The `registerSeekCallback` is called when the container is started and whenever partitions are assigned. You should use this callback when seeking at some arbitrary time after initialization. You should save a reference to the callback. If you use the same listener in multiple containers (or in a `ConcurrentMessageListenerContainer`), you should store the callback in a `ThreadLocal` or some other structure keyed by the listener `Thread`.

When using group management, `onPartitionsAssigned` is called when partitions are assigned. You can use this method, for example, for setting initial offsets for the partitions, by calling the callback. You can also use this method to associate this thread's callback with the assigned partitions (see the example below). You must use the callback argument, not the one passed into `registerSeekCallback`. Starting with version 2.5.5, this method is called, even when using manual partition assignment.

`onPartitionsRevoked` is called when the container is stopped or Kafka revokes assignments. You should discard this thread's callback and remove any associations to the revoked partitions.

The callback has the following methods:

```
void seek(String topic, int partition, long offset);

void seekToBeginning(String topic, int partition);

void seekToBeginning(Collection=<TopicPartitions> partitions);

void seekToEnd(String topic, int partition);

void seekToEnd(Collection=<TopicPartitions> partitions);

void seekRelative(String topic, int partition, long offset, boolean toCurrent);

void seekToTimestamp(String topic, int partition, long timestamp);

void seekToTimestamp(Collection<TopicPartition> topicPartitions, long timestamp);
```

`seekRelative` was added in version 2.3, to perform relative seeks.

- `offset` negative and `toCurrent` `false` - seek relative to the end of the partition.
- `offset` positive and `toCurrent` `false` - seek relative to the beginning of the partition.
- `offset` negative and `toCurrent` `true` - seek relative to the current position (rewind).
- `offset` positive and `toCurrent` `true` - seek relative to the current position (fast forward).

The `seekToTimestamp` methods were also added in version 2.3.

When seeking to the same timestamp for multiple partitions in the `onIdleContainer` or `onPartitionsAssigned` methods, the second method is preferred because it is more efficient to find the offsets for the timestamps in a single call to the consumer's `offsetsForTimes` method. When called from other locations, the container will gather all timestamp seek requests and make one call to `offsetsForTimes`.

You can also perform seek operations from `onIdleContainer()` when an idle container is detected. See Detecting Idle and Non-Responsive Consumers for how to enable idle container detection.

The `seekToBeginning` method that accepts a collection is useful, for example, when processing a compacted topic and you wish to seek to the beginning every time the application is started:

```
public class MyListener implements ConsumerSeekAware {

...

    @Override
    public void onPartitionsAssigned(Map<TopicPartition, Long> assignments,
ConsumerSeekCallback callback) {
        callback.seekToBeginning(assignments.keySet());
    }

}
```

To arbitrarily seek at runtime, use the callback reference from the `registerSeekCallback` for the appropriate thread.

Here is a trivial Spring Boot application that demonstrates how to use the callback; it sends 10 records to the topic; hitting `<Enter>` in the console causes all partitions to seek to the beginning.

```
@SpringBootApplication
public class SeekExampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(SeekExampleApplication.class, args);
    }

    @Bean
    public ApplicationRunner runner(Listener listener, KafkaTemplate<String,
String> template) {
        return args -> {
            IntStream.range(0, 10).forEach(i -> template.send(
                new ProducerRecord<>("seekExample", i % 3, "foo", "bar")));
            while (true) {
                System.in.read();
                listener.seekToStart();
            }
        };
    }

    @Bean
    public NewTopic topic() {
        return new NewTopic("seekExample", 3, (short) 1);
    }

}

@Component
class Listener implements ConsumerSeekAware {

    private static final Logger logger = LoggerFactory.getLogger(Listener.class);

    private final ThreadLocal<ConsumerSeekCallback> callbackForThread = new
ThreadLocal<>();

    private final Map<TopicPartition, ConsumerSeekCallback> callbacks = new
ConcurrentHashMap<>();

    @Override
    public void registerSeekCallback(ConsumerSeekCallback callback) {
        this.callbackForThread.set(callback);
    }

    @Override
    public void onPartitionsAssigned(Map<TopicPartition, Long> assignments,
ConsumerSeekCallback callback) {
        assignments.keySet().forEach(tp -> this.callbacks.put(tp, this
.callbackForThread.get()));
    }
```

```
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        partitions.forEach(tp -> this.callbacks.remove(tp));
        this.callbackForThread.remove();
    }

    @Override
    public void onIdleContainer(Map<TopicPartition, Long> assignments,
ConsumerSeekCallback callback) {
    }

    @KafkaListener(id = "seekExample", topics = "seekExample", concurrency = "3")
    public void listen(ConsumerRecord<String, String> in) {
        logger.info(in.toString());
    }

    public void seekToStart() {
        this.callbacks.forEach((tp, callback) -> callback.seekToBeginning(tp.
topic(), tp.partition()));
    }

}
```

To make things simpler, version 2.3 added the `AbstractConsumerSeekAware` class, which keeps track of which callback is to be used for a topic/partition. The following example shows how to seek to the last record processed, in each partition, each time the container goes idle. It also has methods that allow arbitrary external calls to rewind partitions by one record.

```
public class SeekToLastOnIdleListener extends AbstractConsumerSeekAware {

    @KafkaListener(id = "seekOnIdle", topics = "seekOnIdle")
    public void listen(String in) {
        ...
    }

    @Override
    public void onIdleContainer(Map<org.apache.kafka.common.TopicPartition, Long>
assignments,
            ConsumerSeekCallback callback) {

            assignments.keySet().forEach(tp -> callback.seekRelative(tp.topic(),
tp.partition(), -1, true));
    }

    /**
    * Rewind all partitions one record.
    */
    public void rewindAllOneRecord() {
        getSeekCallbacks()
            .forEach((tp, callback) ->
                callback.seekRelative(tp.topic(), tp.partition(), -1, true));
    }

    /**
    * Rewind one partition one record.
    */
    public void rewindOnePartitionOneRecord(String topic, int partition) {
        getSeekCallbackFor(new org.apache.kafka.common.TopicPartition(topic,
partition))
            .seekRelative(topic, partition, -1, true);
    }

}
```

Version 2.6 added convenience methods to the abstract class:

- `seekToBeginning()` - seeks all assigned partitions to the beginning
- `seekToEnd()` - seeks all assigned partitions to the end
- `seekToTimestamp(long time)` - seeks all assigned partitions to the offset represented by that timestamp.

Example:

```
public class MyListener extends AbstractConsumerSeekAware {

    @KafkaListener(...)
    void listn(...) {
        ...
    }
}

public class SomeOtherBean {

    MyListener listener;

    ...

    void someMethod() {
        this.listener.seekToTimestamp(System.currentTimeMillis - 60_000);
    }

}
```

4.1.9. Container factory

As discussed in `@KafkaListener` Annotation, a `ConcurrentKafkaListenerContainerFactory` is used to create containers for annotated methods.

Starting with version 2.2, you can use the same factory to create any `ConcurrentMessageListenerContainer`. This might be useful if you want to create several containers with similar properties or you wish to use some externally configured factory, such as the one provided by Spring Boot auto-configuration. Once the container is created, you can further modify its properties, many of which are set by using `container.getContainerProperties()`. The following example configures a `ConcurrentMessageListenerContainer`:

```
@Bean
public ConcurrentMessageListenerContainer<String, String>(
        ConcurrentKafkaListenerContainerFactory<String, String> factory) {

    ConcurrentMessageListenerContainer<String, String> container =
        factory.createContainer("topic1", "topic2");
    container.setMessageListener(m -> { ... } );
    return container;
}
```

Containers created this way are not added to the endpoint registry. They should be created as `@Bean` definitions so that they are registered with the application context.

Starting with version 2.3.4, you can add a `ContainerCustomizer` to the factory to further configure each container after it has been created and configured.

```
@Bean
public KafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    ...
    factory.setContainerCustomizer(container -> { /* customize the container */ }
);
    return factory;
}
```

4.1.10. Thread Safety

When using a concurrent message listener container, a single listener instance is invoked on all consumer threads. Listeners, therefore, need to be thread-safe, and it is preferable to use stateless listeners. If it is not possible to make your listener thread-safe or adding synchronization would significantly reduce the benefit of adding concurrency, you can use one of a few techniques:

- Use `n` containers with `concurrency=1` with a prototype scoped `MessageListener` bean so that each container gets its own instance (this is not possible when using `@KafkaListener`).
- Keep the state in `ThreadLocal<?>` instances.
- Have the singleton listener delegate to a bean that is declared in `SimpleThreadScope` (or a similar scope).

To facilitate cleaning up thread state (for the second and third items in the preceding list), starting with version 2.2, the listener container publishes a `ConsumerStoppedEvent` when each thread exits. You can consume these events with an `ApplicationListener` or `@EventListener` method to remove `ThreadLocal<?>` instances or `remove()` thread-scoped beans from the scope. Note that `SimpleThreadScope` does not destroy beans that have a destruction interface (such as `DisposableBean`), so you should `destroy()` the instance yourself.

By default, the application context's event multicaster invokes event listeners on the calling thread. If you change the multicaster to use an async executor, thread cleanup is not effective.

4.1.11. Monitoring

Monitoring Listener Performance

Starting with version 2.3, the listener container will automatically create and update Micrometer `Timer` s for the listener, if `Micrometer` is detected on the class path, and a single `MeterRegistry` is present in the application context. The timers can be disabled by setting the `ContainerProperty` `micrometerEnabled` to `false`.

Two timers are maintained - one for successful calls to the listener and one for failures.

The timers are named `spring.kafka.listener` and have the following tags:

- `name` : (container bean name)
- `result` : `success` or `failure`
- `exception` : `none` or `ListenerExecutionFailedException`

You can add additional tags using the `ContainerProperties` `micrometerTags` property.

With the concurrent container, timers are created for each thread and the `name` tag is suffixed with `-n` where n is `0` to `concurrency-1`.

Monitoring KafkaTemplate Performance

Starting with version 2.5, the template will automatically create and update Micrometer `Timer` s for send operations, if `Micrometer` is detected on the class path, and a single `MeterRegistry` is present in the application context. The timers can be disabled by setting the template's `micrometerEnabled` property to `false`.

Two timers are maintained - one for successful calls to the listener and one for failures.

The timers are named `spring.kafka.template` and have the following tags:

- `name` : (template bean name)
- `result` : `success` or `failure`
- `exception` : `none` or the exception class name for failures

You can add additional tags using the template's `micrometerTags` property.

Micrometer Native Metrics

Starting with version 2.5, the framework provides Factory Listeners to manage a Micrometer `KafkaClientMetrics` instance whenever producers and consumers are created and closed.

To enable this feature, simply add the listeners to your producer and consumer factories:

```
@Bean
public ConsumerFactory<String, String> myConsumerFactory() {
    Map<String, Object> configs = consumerConfigs();
    ...
    DefaultKafkaConsumerFactory<String, String> cf = new
DefaultKafkaConsumerFactory<>(configs);
    ...
    cf.addListener(new MicrometerConsumerListener<String, String>(meterRegistry(),
            Collections.singletonList(new ImmutableTag("customTag",
"customTagValue"))));
    ...
    return cf;
}

@Bean
public ProducerFactory<String, String> myProducerFactory() {
    Map<String, Object> configs = producerConfigs();
    configs.put(ProducerConfig.CLIENT_ID_CONFIG, "myClientId");
    ...
    DefaultKafkaProducerFactory<String, String> pf = new
DefaultKafkaProducerFactory<>(configs);
    ...
    pf.addListener(new MicrometerProducerListener<String, String>(meterRegistry(),
            Collections.singletonList(new ImmutableTag("customTag",
"customTagValue"))));
    ...
    return pf;
}
```

The consumer/producer `id` passed to the listener is added to the meter's tags with tag name `spring.id`.

An example of obtaining one of the Kafka metrics

```
double count = this.meterRegistry.get("kafka.producer.node.incoming.byte.total")
                .tag("customTag", "customTagValue")
                .tag("spring.id", "myProducerFactory.myClientId-1")
                .functionCounter()
                .count()
```

A similar listener is provided for the `StreamsBuilderFactoryBean` - see KafkaStreams Micrometer Support.

4.1.12. Transactions

This section describes how Spring for Apache Kafka supports transactions.

Overview

The 0.11.0.0 client library added support for transactions. Spring for Apache Kafka adds support in the following ways:

- `KafkaTransactionManager`: Used with normal Spring transaction support (`@Transactional`, `TransactionTemplate` etc).
- Transactional `KafkaMessageListenerContainer`
- Local transactions with `KafkaTemplate`
- Transaction synchronization with other transaction managers

Transactions are enabled by providing the `DefaultKafkaProducerFactory` with a `transactionIdPrefix`. In that case, instead of managing a single shared `Producer`, the factory maintains a cache of transactional producers. When the user calls `close()` on a producer, it is returned to the cache for reuse instead of actually being closed. The `transactional.id` property of each producer is `transactionIdPrefix` + `n`, where `n` starts with `0` and is incremented for each new producer, unless the transaction is started by a listener container with a record-based listener. In that case, the `transactional.id` is `<transactionIdPrefix>.<group.id>.<topic>.<partition>`. This is to properly support fencing zombies, as described here. This new behavior was added in versions 1.3.7, 2.0.6, 2.1.10, and 2.2.0. If you wish to revert to the previous behavior, you can set the `producerPerConsumerPartition` property on the `DefaultKafkaProducerFactory` to `false`.

While transactions are supported with batch listeners, by default, zombie fencing is not supported because a batch may contain records from multiple topics or partitions. However, starting with version 2.3.2, zombie fencing is supported if you set the container property `subBatchPerPartition` to true. In that case, the batch listener is invoked once per partition received from the last poll, as if each poll only returned records for a single partition. This is `true` by default since version 2.5 when transactions are enabled with `EOSMode.ALPHA`; set it to `false` if you are using transactions but are not concerned about zombie fencing. Also see Exactly Once Semantics.

Also see `transactionIdPrefix`.

With Spring Boot, it is only necessary to set the `spring.kafka.producer.transaction-id-prefix` property - Boot will automatically configure a `KafkaTransactionManager` bean and wire it into the listener container.

Starting with version 2.5.8, you can now configure the `maxAge` property on the producer factory. This is useful when using transactional producers that might lay idle for the broker's `transactional.id.expiration.ms`. With current `kafka-clients`, this can cause a `ProducerFencedException` without a rebalance. By setting the `maxAge` to less than `transactional.id.expiration.ms`, the factory will refresh the producer if it is past it's max age.

Using `KafkaTransactionManager`

The `KafkaTransactionManager` is an implementation of Spring Framework's `PlatformTransactionManager`. It is provided with a reference to the producer factory in its constructor. If you provide a custom producer factory, it must support transactions. See `ProducerFactory.transactionCapable()`.

You can use the `KafkaTransactionManager` with normal Spring transaction support (`@Transactional`, `TransactionTemplate`, and others). If a transaction is active, any `KafkaTemplate` operations performed within the scope of the transaction use the transaction's `Producer`. The manager commits or rolls back the transaction, depending on success or failure. You must configure the `KafkaTemplate` to use the same `ProducerFactory` as the transaction manager.

Transaction Synchronization

This section refers to producer-only transactions (transactions not started by a listener container); see Using Consumer-Initiated Transactions for information about chaining transactions when the container starts the transaction.

If you want to send records to kafka and perform some database updates, you can use normal Spring transaction management with, say, a `DataSourceTransactionManager`.

```
@Transactional
public void process(List<Thing> things) {
    things.forEach(thing -> this.kafkaTemplate.send("topic", thing));
    updateDb(things);
}
```

The interceptor for the `@Transactional` annotation starts the transaction and the `KafkaTemplate` will synchronize a transaction with that transaction manager; each send will participate in that transaction. When the method exits, the database transaction will commit followed by the Kafka transaction. If you wish the commits to be performed in the reverse order (Kafka first), use nested `@Transactional` methods, with the outer method configured to use the `DataSourceTransactionManager`, and the inner method configured to use the `KafkaTransactionManager`.

See Examples of Kafka Transactions with Other Transaction Managers for examples of an application that synchronizes JDBC and Kafka transactions in Kafka-first or DB-first configurations.

Using Consumer-Initiated Transactions

The `ChainedKafkaTransactionManager` is now deprecated, since version 2.7; see the javadocs for its super class `ChainedTransactionManager` for more information. Instead, use a `KafkaTransactionManager` in the container to start the Kafka transaction and annotate the listener method with `@Transactional` to start the other transaction.

See Examples of Kafka Transactions with Other Transaction Managers for an example application that chains JDBC and Kafka transactions.

`KafkaTemplate` Local Transactions

You can use the `KafkaTemplate` to execute a series of operations within a local transaction. The following example shows how to do so:

```
boolean result = template.executeInTransaction(t -> {
    t.sendDefault("thing1", "thing2");
    t.sendDefault("cat", "hat");
    return true;
});
```

The argument in the callback is the template itself (`this`). If the callback exits normally, the transaction is committed. If an exception is thrown, the transaction is rolled back.

If there is a `KafkaTransactionManager` (or synchronized) transaction in process, it is not used. Instead, a new "nested" transaction is used.

`transactionIdPrefix`

As mentioned in the overview, the producer factory is configured with this property to build the producer `transactional.id` property. There is a dichotomy when specifying this property in that, when running multiple instances of the application with `EOSMode.ALPHA`, it must be the same on all instances to satisfy fencing zombies (also mentioned in the overview) when producing records on a listener container thread. However, when producing records using transactions that are **not** started by a listener container, the prefix has to be different on each instance. Version 2.3, makes this simpler to configure, especially in a Spring Boot application. In previous versions, you had to create two producer factories and `KafkaTemplate` s - one for producing records on a listener container thread and one for stand-alone transactions started by `kafkaTemplate.executeInTransaction()` or by a transaction interceptor on a `@Transactional` method.

Now, you can override the factory's `transactionalIdPrefix` on the `KafkaTemplate` and the `KafkaTransactionManager`.

When using a transaction manager and template for a listener container, you would normally leave this to default to the producer factory's property. This value should be the same for all application instances when using `EOSMode.ALPHA`. With `EOSMode.BETA` it is no longer necessary to use the same `transactional.id`, even for consumer-initiated transactions; in fact, it must be unique on each

instance the same as producer-initiated transactions. For transactions started by the template (or the transaction manager for `@Transaction`) you should set the property on the template and transaction manager respectively. This property must have a different value on each application instance.

This problem (different rules for `transactional.id`) has been eliminated when `EOSMode.BETA` is being used (with broker versions >= 2.5); see Exactly Once Semantics.

`KafkaTemplate` Transactional and non-Transactional Publishing

Normally, when a `KafkaTemplate` is transactional (configured with a transaction-capable producer factory), transactions are required. The transaction can be started by a `TransactionTemplate`, a `@Transactional` method, calling `executeInTransaction`, or by a listener container, when configured with a `KafkaTransactionManager`. Any attempt to use the template outside the scope of a transaction results in the template throwing an `IllegalStateException`. Starting with version 2.4.3, you can set the template's `allowNonTransactional` property to `true`. In that case, the template will allow the operation to run without a transaction, by calling the `ProducerFactory` 's `createNonTransactionalProducer()` method; the producer will be cached, or thread-bound, as normal for reuse. See Using `DefaultKafkaProducerFactory`.

Transactions with Batch Listeners

When a listener fails while transactions are being used, the `AfterRollbackProcessor` is invoked to take some action after the rollback occurs. When using the default `AfterRollbackProcessor` with a record listener, seeks are performed so that the failed record will be redelivered. With a batch listener, however, the whole batch will be redelivered because the framework doesn't know which record in the batch failed. See After-rollback Processor for more information.

When using a batch listener, version 2.4.2 introduced an alternative mechanism to deal with failures while processing a batch; the `BatchToRecordAdapter`. When a container factory with `batchListener` set to true is configured with a `BatchToRecordAdapter`, the listener is invoked with one record at a time. This enables error handling within the batch, while still making it possible to stop processing the entire batch, depending on the exception type. A default `BatchToRecordAdapter` is provided, that can be configured with a standard `ConsumerRecordRecoverer` such as the `DeadLetterPublishingRecoverer`. The following test case configuration snippet illustrates how to use this feature:

```
public static class TestListener {

    final List<String> values = new ArrayList<>();

    @KafkaListener(id = "batchRecordAdapter", topics = "test")
    public void listen(String data) {
        values.add(data);
        if ("bar".equals(data)) {
            throw new RuntimeException("reject partial");
        }
    }

}

@Configuration
@EnableKafka
public static class Config {

    ConsumerRecord<?, ?> failed;

    @Bean
    public TestListener test() {
        return new TestListener();
    }

    @Bean
    public ConsumerFactory<?, ?> consumerFactory() {
        return mock(ConsumerFactory.class);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory factory = new
ConcurrentKafkaListenerContainerFactory();
        factory.setConsumerFactory(consumerFactory());
        factory.setBatchListener(true);
        factory.setBatchToRecordAdapter(new DefaultBatchToRecordAdapter<>((record,
ex) ->  {
            this.failed = record;
        }));
        return factory;
    }

}
```

4.1.13. Exactly Once Semantics

You can provide a listener container with a `KafkaAwareTransactionManager` instance. When so configured, the container starts a transaction before invoking the listener. Any `KafkaTemplate` operations performed by the listener participate in the transaction. If the listener successfully processes the record (or multiple records, when using a `BatchMessageListener`), the container sends the offset(s) to the transaction by using `producer.sendOffsetsToTransaction()`), before the transaction manager commits the transaction. If the listener throws an exception, the transaction is rolled back and the consumer is repositioned so that the rolled-back record(s) can be retrieved on the next poll. See After-rollback Processor for more information and for handling records that repeatedly fail.

Using transactions enables Exactly Once Semantics (EOS).

This means that, for a `read→process-write` sequence, it is guaranteed that the **sequence** is completed exactly once. (The read and process are have at least once semantics).

Spring for Apache Kafka version 2.5 and later supports two EOS modes:

- `ALPHA` - aka `transactional.id` fencing (since version 0.11.0.0)
- `BETA` - aka fetch-offset-request fencing (since version 2.5)

With mode `ALPHA`, the producer is "fenced" if another instance with the same `transactional.id` is started. Spring manages this by using a `Producer` for each `group.id/topic/partition`; when a rebalance occurs a new instance will use the same `transactional.id` and the old producer is fenced.

With mode `BETA`, it is not necessary to have a producer for each `group.id/topic/partition` because consumer metadata is sent along with the offsets to the transaction and the broker can determine if the producer is fenced using that information instead.

Starting with version 2.6, the default `EOSMode` is `BETA`.

To configure the container to use mode `ALPHA`, set the container property `EOSMode` to `ALPHA`, to revert to the previous behavior.

With `BETA`, your brokers must be version 2.5 or later, however with `kafka-clients` version 2.6, the producer will automatically fall back to `ALPHA` if the broker does not support `BETA`. The `DefaultKafkaProducerFactory` is configured to enable that behavior. If your brokers are earlier than 2.5, be sure to leave the `DefaultKafkaProducerFactory producerPerConsumerPartition` set to `true` and, if you are using a batch listener, you should set `subBatchPerPartition` to `true`.

When your brokers are upgraded to 2.5 or later, the producer will automatically switch to using mode `BETA`, but the number of producers will remain as before. You can then do a rolling upgrade of your application with `producerPerConsumerPartition` set to `false` to reduce the number of producers; you should also no longer set the `subBatchPerPartition` container property.

If your brokers are already 2.5 or newer, you should set the `DefaultKafkaProducerFactory producerPerConsumerPartition` property to `false`, to reduce the number of producers needed.

When using `EOSMode.BETA` with `producerPerConsumerPartition=false` the `transactional.id` must be unique across all application instances.

When using `BETA` mode, it is no longer necessary to set the `subBatchPerPartition` to `true`; it will default to `false` when the `EOSMode` is `BETA`.

Refer to KIP-447 for more information.

4.1.14. Wiring Spring Beans into Producer/Consumer Interceptors

Apache Kafka provides a mechanism to add interceptors to producers and consumers. These objects are managed by Kafka, not Spring, and so normal Spring dependency injection won't work for wiring in dependent Spring Beans. However, you can manually wire in those dependencies using the interceptor `config()` method. The following Spring Boot application shows how to do this by overriding boot's default factories to add some dependent bean into the configuration properties.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public ConsumerFactory<?, ?> kafkaConsumerFactory(KafkaProperties properties,
SomeBean someBean) {
        Map<String, Object> consumerProperties = properties
.buildConsumerProperties();
        consumerProperties.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
MyConsumerInterceptor.class.getName());
        consumerProperties.put("some.bean", someBean);
        return new DefaultKafkaConsumerFactory<>(consumerProperties);
    }

    @Bean
    public ProducerFactory<?, ?> kafkaProducerFactory(KafkaProperties properties,
SomeBean someBean) {
        Map<String, Object> producerProperties = properties
.buildProducerProperties();
        producerProperties.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
MyProducerInterceptor.class.getName());
        producerProperties.put("some.bean", someBean);
        DefaultKafkaProducerFactory<?, ?> factory = new
DefaultKafkaProducerFactory<>(producerProperties);
        String transactionIdPrefix = properties.getProducer()
                .getTransactionIdPrefix();
        if (transactionIdPrefix != null) {
            factory.setTransactionIdPrefix(transactionIdPrefix);
        }
        return factory;
    }

    @Bean
    public SomeBean someBean() {
        return new SomeBean();
    }

    @KafkaListener(id = "kgk897", topics = "kgh897")
    public void listen(String in) {
        System.out.println("Received " + in);
    }

    @Bean
    public ApplicationRunner runner(KafkaTemplate<String, String> template) {
        return args -> template.send("kgh897", "test");
```

```
    }

    @Bean
    public NewTopic kRequests() {
        return TopicBuilder.name("kgh897")
            .partitions(1)
            .replicas(1)
            .build();
    }

}
```

```
public class SomeBean {

    public void someMethod(String what) {
        System.out.println(what + " in my foo bean");
    }

}
```

```
public class MyProducerInterceptor implements ProducerInterceptor<String, String>
{

    private SomeBean bean;

    @Override
    public void configure(Map<String, ?> configs) {
        this.bean = (SomeBean) configs.get("some.bean");
    }

    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String, String>
record) {
        this.bean.someMethod("producer interceptor");
        return record;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
    }

    @Override
    public void close() {
    }

}
```

```
public class MyConsumerInterceptor implements ConsumerInterceptor<String, String>
{

    private SomeBean bean;

    @Override
    public void configure(Map<String, ?> configs) {
        this.bean = (SomeBean) configs.get("some.bean");
    }

    @Override
    public ConsumerRecords<String, String> onConsume(ConsumerRecords<String,
String> records) {
        this.bean.someMethod("consumer interceptor");
        return records;
    }

    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets) {
    }

    @Override
    public void close() {
    }

}
```

Result:

```
producer interceptor in my foo bean
consumer interceptor in my foo bean
Received test
```

4.1.15. Pausing and Resuming Listener Containers

Version 2.1.3 added `pause()` and `resume()` methods to listener containers. Previously, you could pause a consumer within a `ConsumerAwareMessageListener` and resume it by listening for a `ListenerContainerIdleEvent`, which provides access to the `Consumer` object. While you could pause a consumer in an idle container by using an event listener, in some cases, this was not thread-safe, since there is no guarantee that the event listener is invoked on the consumer thread. To safely pause and resume consumers, you should use the `pause` and `resume` methods on the listener containers. A `pause()` takes effect just before the next `poll()`; a `resume()` takes effect just after the current `poll()` returns. When a container is paused, it continues to `poll()` the consumer, avoiding a rebalance if group management is being used, but it does not retrieve any records. See the Kafka

documentation for more information.

Starting with version 2.1.5, you can call `isPauseRequested()` to see if `pause()` has been called. However, the consumers might not have actually paused yet. `isConsumerPaused()` returns true if all `Consumer` instances have actually paused.

In addition (also since 2.1.5), `ConsumerPausedEvent` and `ConsumerResumedEvent` instances are published with the container as the `source` property and the `TopicPartition` instances involved in the `partitions` property.

The following simple Spring Boot application demonstrates by using the container registry to get a reference to a `@KafkaListener` method's container and pausing or resuming its consumers as well as receiving the corresponding events:

```
@SpringBootApplication
public class Application implements ApplicationListener<KafkaEvent> {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args).close();
    }

    @Override
    public void onApplicationEvent(KafkaEvent event) {
        System.out.println(event);
    }

    @Bean
    public ApplicationRunner runner(KafkaListenerEndpointRegistry registry,
            KafkaTemplate<String, String> template) {
        return args -> {
            template.send("pause.resume.topic", "thing1");
            Thread.sleep(10_000);
            System.out.println("pausing");
            registry.getListenerContainer("pause.resume").pause();
            Thread.sleep(10_000);
            template.send("pause.resume.topic", "thing2");
            Thread.sleep(10_000);
            System.out.println("resuming");
            registry.getListenerContainer("pause.resume").resume();
            Thread.sleep(10_000);
        };
    }

    @KafkaListener(id = "pause.resume", topics = "pause.resume.topic")
    public void listen(String in) {
        System.out.println(in);
    }

    @Bean
    public NewTopic topic() {
        return TopicBuilder.name("pause.resume.topic")
            .partitions(2)
            .replicas(1)
            .build();
    }

}
```

The following listing shows the results of the preceding example:

```
partitions assigned: [pause.resume.topic-1, pause.resume.topic-0]
thing1
pausing
ConsumerPausedEvent [partitions=[pause.resume.topic-1, pause.resume.topic-0]]
resuming
ConsumerResumedEvent [partitions=[pause.resume.topic-1, pause.resume.topic-0]]
thing2
```

4.1.16. Pausing and Resuming Partitions on Listener Containers

Since version 2.7 you can pause and resume the consumption of specific partitions assigned to that consumer by using the `pausePartition(TopicPartition topicPartition)` and `resumePartition(TopicPartition topicPartition)` methods in the listener containers. The pausing and resuming takes place respectively before and after the `poll()` similar to the `pause()` and `resume()` methods. The `isPartitionPauseRequested()` method returns true if pause for that partition has been requested. The `isPartitionPaused()` method returns true if that partition has effectively been paused.

Also since version 2.7 `ConsumerPartitionPausedEvent` and `ConsumerPartitionResumedEvent` instances are published with the container as the `source` property and the `TopicPartition` instance.

4.1.17. Serialization, Deserialization, and Message Conversion

Overview

Apache Kafka provides a high-level API for serializing and deserializing record values as well as their keys. It is present with the `org.apache.kafka.common.serialization.Serializer<T>` and `org.apache.kafka.common.serialization.Deserializer<T>` abstractions with some built-in implementations. Meanwhile, we can specify serializer and deserializer classes by using `Producer` or `Consumer` configuration properties. The following example shows how to do so:

```
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeserializer.class
);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.
class);
...
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
```

For more complex or particular cases, the `KafkaConsumer` (and, therefore, `KafkaProducer`) provides overloaded constructors to accept `Serializer` and `Deserializer` instances for `keys` and `values`, respectively.

When you use this API, the `DefaultKafkaProducerFactory` and `DefaultKafkaConsumerFactory` also

provide properties (through constructors or setter methods) to inject custom `Serializer` and `Deserializer` instances into the target `Producer` or `Consumer`. Also, you can pass in `Supplier<Serializer>` or `Supplier<Deserializer>` instances through constructors - these `Supplier` s are called on creation of each `Producer` or `Consumer`.

String serialization

Since version 2.5, Spring for Apache Kafka provides `ToStringSerializer` and `ParseStringDeserializer` classes that use String representation of entities. They rely on methods `toString` and some `Function<String>` or `BiFunction<String, Headers>` to parse the String and populate properties of an instance. Usually, this would invoke some static method on the class, such as `parse`:

```
ToStringSerializer<Thing> thingSerializer = new ToStringSerializer<>();
//...
ParseStringDeserializer<Thing> deserializer = new ParseStringDeserializer<>(Thing
::parse);
```

By default, the `ToStringSerializer` is configured to convey type information about the serialized entity in the record `Headers`. You can disable this by setting the `addTypeInfo` property to false. This information can be used by `ParseStringDeserializer` on the receiving side.

- `ToStringSerializer.ADD_TYPE_INFO_HEADERS` (default `true`): You can set it to `false` to disable this feature on the `ToStringSerializer` (sets the `addTypeInfo` property).

```
ParseStringDeserializer<Object> deserializer = new ParseStringDeserializer<>((str,
headers) -> {
    byte[] header = headers.lastHeader(ToStringSerializer.VALUE_TYPE).value();
    String entityType = new String(header);

    if (entityType.contains("Thing")) {
        return Thing.parse(str);
    }
    else {
        // ...parsing logic
    }
});
```

You can configure the `Charset` used to convert `String` to/from `byte[]` with the default being `UTF-8`.

You can configure the deserializer with the name of the parser method using `ConsumerConfig` properties:

- `ParseStringDeserializer.KEY_PARSER`

- `ParseStringDeserializer.VALUE_PARSER`

The properties must contain the fully qualified name of the class followed by the method name, separated by a period `.`. The method must be static and have a signature of either `(String, Headers)` or `(String)`.

A `ToFromStringSerde` is also provided, for use with Kafka Streams.

JSON

Spring for Apache Kafka also provides `JsonSerializer` and `JsonDeserializer` implementations that are based on the Jackson JSON object mapper. The `JsonSerializer` allows writing any Java object as a JSON `byte[]`. The `JsonDeserializer` requires an additional `Class<?> targetType` argument to allow the deserialization of a consumed `byte[]` to the proper target object. The following example shows how to create a `JsonDeserializer`:

```
JsonDeserializer<Thing> thingDeserializer = new JsonDeserializer<>(Thing.class);
```

You can customize both `JsonSerializer` and `JsonDeserializer` with an `ObjectMapper`. You can also extend them to implement some particular configuration logic in the `configure(Map<String, ?> configs, boolean isKey)` method.

Starting with version 2.3, all the JSON-aware components are configured by default with a `JacksonUtils.enhancedObjectMapper()` instance, which comes with the `MapperFeature.DEFAULT_VIEW_INCLUSION` and `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` features disabled. Also such an instance is supplied with well-known modules for custom data types, such a Java time and Kotlin support. See `JacksonUtils.enhancedObjectMapper()` JavaDocs for more information. This method also registers a `org.springframework.kafka.support.JacksonMimeTypeModule` for `org.springframework.util.MimeType` objects serialization into the plain string for inter-platform compatibility over the network. A `JacksonMimeTypeModule` can be registered as a bean in the application context and it will be auto-configured into Spring Boot `ObjectMapper` instance.

Also starting with version 2.3, the `JsonDeserializer` provides `TypeReference`-based constructors for better handling of target generic container types.

Starting with version 2.1, you can convey type information in record `Headers`, allowing the handling of multiple types. In addition, you can configure the serializer and deserializer by using the following Kafka properties. They have no effect if you have provided `Serializer` and `Deserializer` instances for `KafkaConsumer` and `KafkaProducer`, respectively.

Configuration Properties

- `JsonSerializer.ADD_TYPE_INFO_HEADERS` (default `true`): You can set it to `false` to disable this feature on the `JsonSerializer` (sets the `addTypeInfo` property).
- `JsonSerializer.TYPE_MAPPINGS` (default `empty`): See Mapping Types.
- `JsonDeserializer.USE_TYPE_INFO_HEADERS` (default `true`): You can set it to `false` to ignore headers

set by the serializer.

- `JsonDeserializer.REMOVE_TYPE_INFO_HEADERS` (default `true`): You can set it to `false` to retain headers set by the serializer.
- `JsonDeserializer.KEY_DEFAULT_TYPE`: Fallback type for deserialization of keys if no header information is present.
- `JsonDeserializer.VALUE_DEFAULT_TYPE`: Fallback type for deserialization of values if no header information is present.
- `JsonDeserializer.TRUSTED_PACKAGES` (default `java.util`, `java.lang`): Comma-delimited list of package patterns allowed for deserialization. `*` means deserialize all.
- `JsonDeserializer.TYPE_MAPPINGS` (default `empty`): See Mapping Types.
- `JsonDeserializer.KEY_TYPE_METHOD` (default `empty`): See Using Methods to Determine Types.
- `JsonDeserializer.VALUE_TYPE_METHOD` (default `empty`): See Using Methods to Determine Types.

Starting with version 2.2, the type information headers (if added by the serializer) are removed by the deserializer. You can revert to the previous behavior by setting the `removeTypeHeaders` property to `false`, either directly on the deserializer or with the configuration property described earlier.

Mapping Types

Starting with version 2.2, when using JSON, you can now provide type mappings by using the properties in the preceding list. Previously, you had to customize the type mapper within the serializer and deserializer. Mappings consist of a comma-delimited list of `token:className` pairs. On outbound, the payload's class name is mapped to the corresponding token. On inbound, the token in the type header is mapped to the corresponding class name.

The following example creates a set of mappings:

```
senderProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.
class);
senderProps.put(JsonSerializer.TYPE_MAPPINGS, "cat:com.mycat.Cat,
hat:com.myhat.hat");
...
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);
consumerProps.put(JsonDeSerializer.TYPE_MAPPINGS, "cat:com.yourcat.Cat,
hat:com.yourhat.hat");
```

The corresponding objects must be compatible.

If you use Spring Boot, you can provide these properties in the `application.properties` (or yaml) file. The following example shows how to do so:

```
spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties.spring.json.type.mapping=cat:com.mycat.Cat,hat:co
m.myhat.Hat
```

You can perform only simple configuration with properties. For more advanced configuration (such as using a custom `ObjectMapper` in the serializer and deserializer), you should use the producer and consumer factory constructors that accept a pre-built serializer and deserializer. The following Spring Boot example overrides the default factories:

```
@Bean
public ConsumerFactory<Foo, Bar> kafkaConsumerFactory
(KafkaProperties properties,
    JsonDeserializer customDeserializer) {

    return new DefaultKafkaConsumerFactory<>(properties
.buildConsumerProperties(),
        customDeserializer, customDeserializer);
}

@Bean
public ProducerFactory<Foo, Bar> kafkaProducerFactory
(KafkaProperties properties,
    JsonSerializer customSerializer) {

    return new DefaultKafkaProducerFactory<>(properties
.buildProducerProperties(),
        customSerializer, customSerializer);
}
```

Setters are also provided, as an alternative to using these constructors.

Starting with version 2.2, you can explicitly configure the deserializer to use the supplied target type and ignore type information in headers by using one of the overloaded constructors that have a boolean `useHeadersIfPresent` (which is `true` by default). The following example shows how to do so:

```
DefaultKafkaConsumerFactory<Integer, Cat1> cf = new DefaultKafkaConsumerFactory<>
(props,
        new IntegerDeserializer(), new JsonDeserializer<>(Cat1.class, false));
```

Using Methods to Determine Types

Starting with version 2.5, you can now configure the deserializer, via properties, to invoke a method to determine the target type. If present, this will override any of the other techniques discussed above. This can be useful if the data is published by an application that does not use the Spring serializer and you need to deserialize to different types depending on the data, or other headers. Set these properties to the method name - a fully qualified class name followed by the method name, separated by a period `.`. The method must be declared as `public static`, have one of three signatures `(String topic, byte[] data, Headers headers)`, `(byte[] data, Headers headers)` or `(byte[] data)` and return a Jackson `JavaType`.

- `JsonDeserializer.KEY_TYPE_METHOD` : `spring.json.key.type.method`
- `JsonDeserializer.VALUE_TYPE_METHOD` : `spring.json.value.type.method`

You can use arbitrary headers or inspect the data to determine the type.

Example

```
JavaType thing1Type = TypeFactory.defaultInstance().constructType(Thing1.class);

JavaType thing2Type = TypeFactory.defaultInstance().constructType(Thing2.class);

public static JavaType thingOneOrThingTwo(byte[] data, Headers headers) {
    // {"thisIsAFieldInThing1":"value", ...
    if (data[21] == '1') {
        return thing1Type;
    }
    else {
        return thing2Type;
    }
}
```

For more sophisticated data inspection consider using `JsonPath` or similar but, the simpler the test to determine the type, the more efficient the process will be.

The following is an example of creating the deserializer programmatically (when providing the consumer factory with the deserializer in the constructor):

```
JsonDeserializer<Object> deser = new JsonDeserializer<>()
        .trustedPackages("*")
        .typeResolver(SomeClass::thing1Thing2JavaTypeForTopic);

...

public static JavaType thing1Thing2JavaTypeForTopic(String topic, byte[] data,
Headers headers) {
    ...
}
```

Programmatic Construction

When constructing the serializer/deserializer programmatically for use in the producer/consumer factory, since version 2.3, you can use the fluent API, which simplifies configuration.

The following example assumes you are using Spring Boot:

```
@Bean
public DefaultKafkaProducerFactory pf(KafkaProperties properties) {
    Map<String, Object> props = properties.buildProducerProperties();
    DefaultKafkaProducerFactory pf = new DefaultKafkaProducerFactory(props,
        new JsonSerializer<>(MyKeyType.class)
            .forKeys()
            .noTypeInfo(),
        new JsonSerializer<>(MyValueType.class)
            .noTypeInfo());
}

@Bean
public DefaultKafkaConsumerFactory pf(KafkaProperties properties) {
    Map<String, Object> props = properties.buildConsumerProperties();
    DefaultKafkaConsumerFactory pf = new DefaultKafkaConsumerFactory(props,
        new JsonDeserializer<>(MyKeyType.class)
            .forKeys()
            .ignoreTypeHeaders(),
        new JsonDeserializer<>(MyValueType.class)
            .ignoreTypeHeaders());
}
```

To provide type mapping programmatically, similar to Using Methods to Determine Types, use the `typeFunction` property.

Example

```
JsonDeserializer<Object> deser = new JsonDeserializer<>()
        .trustedPackages("*")
        .typeFunction(MyUtils::thingOneOrThingTwo);
```

Delegating Serializer and Deserializer

Version 2.3 introduced the `DelegatingSerializer` and `DelegatingDeserializer`, which allow producing and consuming records with different key and/or value types. Producers must set a header `DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR` to a selector value that is used to select which serializer to use for the value and `DelegatingSerializer.KEY_SERIALIZATION_SELECTOR` for the key; if a match is not found, an `IllegalStateException` is thrown.

For incoming records, the deserializer uses the same headers to select the deserializer to use; if a match is not found or the header is not present, the raw `byte[]` is returned.

You can configure the map of selector to `Serializer` / `Deserializer` via a constructor, or you can configure it via Kafka producer/consumer properties with the keys `DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR_CONFIG` and `DelegatingSerializer.KEY_SERIALIZATION_SELECTOR_CONFIG`. For the serializer, the producer property can be a `Map<String, Object>` where the key is the selector and the value is a `Serializer` instance, a serializer `Class` or the class name. The property can also be a String of comma-delimited map entries, as shown below.

For the deserializer, the consumer property can be a `Map<String, Object>` where the key is the selector and the value is a `Deserializer` instance, a deserializer `Class` or the class name. The property can also be a String of comma-delimited map entries, as shown below.

To configure using properties, use the following syntax:

```
producerProps.put(DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR_CONFIG,
    "thing1:com.example.MyThing1Serializer, thing2:com.example.MyThing2Serializer
")

consumerProps.put(DelegatingDeserializer.VALUE_SERIALIZATION_SELECTOR_CONFIG,
    "thing1:com.example.MyThing1Deserializer,
thing2:com.example.MyThing2Deserializer")
```

Producers would then set the `DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR` header to `thing1` or `thing2`.

This technique supports sending different types to the same topic (or different topics).

Starting with version 2.5.1, it is not necessary to set the selector header, if the type (key or value) is one of the standard types supported by `Serdes` (`Long`, `Integer`, etc). Instead, the serializer will set the header to the class name of the type. It is not necessary to configure serializers or deserializers for these types, they will be created (once) dynamically.

For another technique to send different types to different topics, see Using `RoutingKafkaTemplate`.

Retrying Deserializer

The `RetryingDeserializer` uses a delegate `Deserializer` and `RetryTemplate` to retry deserialization when the delegate might have transient errors, such a network issues, during deserialization.

```
ConsumerFactory cf = new DefaultKafkaConsumerFactory(myConsumerConfigs,
    new RetryingDeserializer(myUnreliableKeyDeserializer, retryTemplate),
    new RetryingDeserializer(myUnreliableValueDeserializer, retryTemplate));
```

Refer to the spring-retry project for configuration of the `RetryTemplate` with a retry policy, back off policy, etc.

Spring Messaging Message Conversion

Although the `Serializer` and `Deserializer` API is quite simple and flexible from the low-level Kafka `Consumer` and `Producer` perspective, you might need more flexibility at the Spring Messaging level, when using either `@KafkaListener` or Spring Integration's Apache Kafka Support. To let you easily convert to and from `org.springframework.messaging.Message`, Spring for Apache Kafka provides a `MessageConverter` abstraction with the `MessagingMessageConverter` implementation and its `JsonMessageConverter` (and subclasses) customization. You can inject the `MessageConverter` into a `KafkaTemplate` instance directly and by using `AbstractKafkaListenerContainerFactory` bean definition for the `@KafkaListener.containerFactory()` property. The following example shows how to do so:

```
@Bean
public KafkaListenerContainerFactory<?, ?> kafkaJsonListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setMessageConverter(new JsonMessageConverter());
    return factory;
}
...
@KafkaListener(topics = "jsonData",
                containerFactory = "kafkaJsonListenerContainerFactory")
public void jsonListener(Cat cat) {
...
}
```

When using Spring Boot, simply define the converter as a `@Bean` and Spring Boot auto configuration will wire it into the auto-configured template and container factory.

When you use a `@KafkaListener`, the parameter type is provided to the message converter to assist with the conversion.

This type inference can be achieved only when the `@KafkaListener` annotation is declared at the method level. With a class-level `@KafkaListener`, the payload type is used to select which `@KafkaHandler` method to invoke, so it must already have been converted before the method can be chosen.

On the consumer side, you can configure a `JsonMessageConverter`; it can handle `ConsumerRecord` values of type `byte[]`, `Bytes` and `String` so should be used in conjunction with a `ByteArrayDeserializer`, `BytesDeserializer` or `StringDeserializer`. (`byte[]` and `Bytes` are more efficient because they avoid an unnecessary `byte[]` to `String` conversion). You can also configure the specific subclass of `JsonMessageConverter` corresponding to the deserializer, if you so wish.

On the producer side, when you use Spring Integration or the `KafkaTemplate.send(Message<?> message)` method (see Using `KafkaTemplate`), you must configure a message converter that is compatible with the configured Kafka `Serializer`.

- `StringJsonMessageConverter` with `StringSerializer`
- `BytesJsonMessageConverter` with `BytesSerializer`
- `ByteArrayJsonMessageConverter` with `ByteArraySerializer`

Again, using `byte[]` or `Bytes` is more efficient because they avoid a `String` to `byte[]` conversion.

For convenience, starting with version 2.3, the framework also provides a `StringOrBytesSerializer` which can serialize all three value types so it can be used with any of the message converters.

Starting with version 2.7.1, message payload conversion can be delegated to a `spring-messaging` `SmartMessageConverter`; this enables conversion, for example, to be based on the `MessageHeaders.CONTENT_TYPE` header.

The `KafkaMessageConverter.fromMessage()` method is called for outbound conversion to a `ProducerRecord` with the message payload in the `ProducerRecord.value()` property. The `KafkaMessageConverter.toMessage()` method is called for inbound conversion from `ConsumerRecord` with the payload being the `ConsumerRecord.value()` property. The `SmartMessageConverter.toMessage()` method is called to create a new outbound `Message<?>` from the `Message` passed to `fromMessage()` (usually by `KafkaTemplate.send(Message<?> msg)`). Similarly, in the `KafkaMessageConverter.toMessage()` method, after the converter has created a new `Message<?>` from the `ConsumerRecord`, the `SmartMessageConverter.fromMessage()` method is called and then the final inbound message is created with the newly converted payload. In either case, if the `SmartMessageConverter` returns `null`, the original message is used.

When the default converter is used in the `KafkaTemplate` and listener container factory, you configure the `SmartMessageConverter` by calling `setMessagingConverter()` on the template and via the `contentMessageConverter` property on `@KafkaListener` methods.

Examples:

```
template.setMessagingConverter(mySmartConverter);
```

```
@KafkaListener(id = "withSmartConverter", topics = "someTopic",
    contentTypeConverter = "mySmartConverter")
public void smart(Thing thing) {
    ...
}
```

Using Spring Data Projection Interfaces

Starting with version 2.1.1, you can convert JSON to a Spring Data Projection interface instead of a concrete type. This allows very selective, and low-coupled bindings to data, including the lookup of values from multiple places inside the JSON document. For example the following interface can be defined as message payload type:

```
interface SomeSample {

  @JsonPath({ "$.username", "$.user.name" })
  String getUsername();

}
```

```
@KafkaListener(id="projection.listener", topics = "projection")
public void projection(SomeSample in) {
    String username = in.getUsername();
    ...
}
```

Accessor methods will be used to lookup the property name as field in the received JSON document by default. The `@JsonPath` expression allows customization of the value lookup, and even to define multiple JSON Path expressions, to lookup values from multiple places until an expression returns an actual value.

To enable this feature, use a `ProjectingMessageConverter` configured with an appropriate delegate converter (used for outbound conversion and converting non-projection interfaces). You must also add `spring-data:spring-data-commons` and `com.jayway.jsonpath:json-path` to the class path.

When used as the parameter to a `@KafkaListener` method, the interface type is automatically passed to the converter as normal.

Using `ErrorHandlingDeserializer`

When a deserializer fails to deserialize a message, Spring has no way to handle the problem, because it occurs before the `poll()` returns. To solve this problem, the `ErrorHandlingDeserializer` has been introduced. This deserializer delegates to a real deserializer (key or value). If the delegate fails to deserialize the record content, the `ErrorHandlingDeserializer` returns a `null` value and a `DeserializationException` in a header that contains the cause and the raw bytes. When you use a record-level `MessageListener`, if the `ConsumerRecord` contains a `DeserializationException` header for either the key or value, the container's `ErrorHandler` is called with the failed `ConsumerRecord`. The record is not passed to the listener.

Alternatively, you can configure the `ErrorHandlingDeserializer` to create a custom value by providing a `failedDeserializationFunction`, which is a `Function<FailedDeserializationInfo, T>`. This function is invoked to create an instance of `T`, which is passed to the listener in the usual fashion. An object of type `FailedDeserializationInfo`, which contains all the contextual information is provided to the function. You can find the `DeserializationException` (as a serialized Java object) in headers. See the Javadoc for the `ErrorHandlingDeserializer` for more information.

When you use a `BatchMessageListener`, you must provide a `failedDeserializationFunction`. Otherwise, the batch of records are not type safe.

You can use the `DefaultKafkaConsumerFactory` constructor that takes key and value `Deserializer` objects and wire in appropriate `ErrorHandlingDeserializer` instances that you have configured with the proper delegates. Alternatively, you can use consumer configuration properties (which are used by the `ErrorHandlingDeserializer`) to instantiate the delegates. The property names are `ErrorHandlingDeserializer.KEY_DESERIALIZER_CLASS` and `ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS`. The property value can be a class or class name. The following example shows how to set these properties:

```
... // other props
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, ErrorHandlingDeserializer
.class);
props.put(ErrorHandlingDeserializer.KEY_DESERIALIZER_CLASS, JsonDeserializer.
class);
props.put(JsonDeserializer.KEY_DEFAULT_TYPE, "com.example.MyKey")
props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS, JsonDeserializer
.class.getName());
props.put(JsonDeserializer.VALUE_DEFAULT_TYPE, "com.example.MyValue")
props.put(JsonDeserializer.TRUSTED_PACKAGES, "com.example")
return new DefaultKafkaConsumerFactory<>(props);
```

The following example uses a `failedDeserializationFunction`.

```
public class BadFoo extends Foo {

  private final FailedDeserializationInfo failedDeserializationInfo;

  public BadFoo(FailedDeserializationInfo failedDeserializationInfo) {
    this.failedDeserializationInfo = failedDeserializationInfo;
  }

  public FailedDeserializationInfo getFailedDeserializationInfo() {
    return this.failedDeserializationInfo;
  }

}

public class FailedFooProvider implements Function<FailedDeserializationInfo, Foo>
{

  @Override
  public Foo apply(FailedDeserializationInfo info) {
    return new BadFoo(info);
  }

}
```

The preceding example uses the following configuration:

```
...
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
consumerProps.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
JsonDeserializer.class);
consumerProps.put(ErrorHandlingDeserializer.VALUE_FUNCTION, FailedFooProvider
.class);
...
```

Payload Conversion with Batch Listeners

You can also use a `JsonMessageConverter` within a `BatchMessagingMessageConverter` to convert batch messages when you use a batch listener container factory. See Serialization, Deserialization, and Message Conversion and Spring Messaging Message Conversion for more information.

By default, the type for the conversion is inferred from the listener argument. If you configure the `JsonMessageConverter` with a `DefaultJackson2TypeMapper` that has its `TypePrecedence` set to `TYPE_ID` (instead of the default `INFERRED`), the converter uses the type information in headers (if present) instead. This allows, for example, listener methods to be declared with interfaces instead of

concrete classes. Also, the type converter supports mapping, so the deserialization can be to a different type than the source (as long as the data is compatible). This is also useful when you use class-level `@KafkaListener` instances where the payload must have already been converted to determine which method to invoke. The following example creates beans that use this method:

```
@Bean
public KafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setBatchListener(true);
    factory.setMessageConverter(new BatchMessagingMessageConverter(converter()));
    return factory;
}

@Bean
public JsonMessageConverter converter() {
    return new JsonMessageConverter();
}
```

Note that, for this to work, the method signature for the conversion target must be a container object with a single generic parameter type, such as the following:

```
@KafkaListener(topics = "blc1")
public void listen(List<Foo> foos, @Header(KafkaHeaders.OFFSET) List<Long>
offsets) {
    ...
}
```

Note that you can still access the batch headers.

If the batch converter has a record converter that supports it, you can also receive a list of messages where the payloads are converted according to the generic type. The following example shows how to do so:

```
@KafkaListener(topics = "blc3", groupId = "blc3")
public void listen1(List<Message<Foo>> fooMessages) {
    ...
}
```

`ConversionService` Customization

Starting with version 2.1.1, the `org.springframework.core.convert.ConversionService` used by the default `o.s.messaging.handler.annotation.support.MessageHandlerMethodFactory` to resolve parameters for the invocation of a listener method is supplied with all beans that implement any of the following interfaces:

- `org.springframework.core.convert.converter.Converter`
- `org.springframework.core.convert.converter.GenericConverter`
- `org.springframework.format.Formatter`

This lets you further customize listener deserialization without changing the default configuration for `ConsumerFactory` and `KafkaListenerContainerFactory`.

Setting a custom `MessageHandlerMethodFactory` on the `KafkaListenerEndpointRegistrar` through a `KafkaListenerConfigurer` bean disables this feature.

Adding custom `HandlerMethodArgumentResolver` to `@KafkaListener`

Starting with version 2.4.2 you are able to add your own `HandlerMethodArgumentResolver` and resolve custom method parameters. All you need is to implement `KafkaListenerConfigurer` and use method `setCustomMethodArgumentResolvers()` from class `KafkaListenerEndpointRegistrar`.

```
@Configuration
class CustomKafkaConfig implements KafkaListenerConfigurer {

    @Override
    public void configureKafkaListeners(KafkaListenerEndpointRegistrar registrar)
{
        registrar.setCustomMethodArgumentResolvers(
            new HandlerMethodArgumentResolver() {

                @Override
                public boolean supportsParameter(MethodParameter parameter) {
                    return CustomMethodArgument.class.isAssignableFrom(parameter
.getParameterType());
                }

                @Override
                public Object resolveArgument(MethodParameter parameter, Message<
?> message) {
                    return new CustomMethodArgument(
                        message.getHeaders().get(KafkaHeaders.RECEIVED_TOPIC,
String.class)
                    );
                }
            }
        );
    }

}
```

4.1.18. Message Headers

The 0.11.0.0 client introduced support for headers in messages. As of version 2.0, Spring for Apache Kafka now supports mapping these headers to and from `spring-messaging` `MessageHeaders`.

Previous versions mapped `ConsumerRecord` and `ProducerRecord` to spring-messaging `Message<?>`, where the value property is mapped to and from the `payload` and other properties (`topic`, `partition`, and so on) were mapped to headers. This is still the case, but additional (arbitrary) headers can now be mapped.

Apache Kafka headers have a simple API, shown in the following interface definition:

```
public interface Header {

    String key();

    byte[] value();

}
```

The `KafkaHeaderMapper` strategy is provided to map header entries between Kafka `Headers` and `MessageHeaders`. Its interface definition is as follows:

```
public interface KafkaHeaderMapper {

    void fromHeaders(MessageHeaders headers, Headers target);

    void toHeaders(Headers source, Map<String, Object> target);

}
```

The `DefaultKafkaHeaderMapper` maps the key to the `MessageHeaders` header name and, in order to support rich header types for outbound messages, JSON conversion is performed. A "special" header (with a key of `spring_json_header_types`) contains a JSON map of `<key>:<type>`. This header is used on the inbound side to provide appropriate conversion of each header value to the original type.

On the inbound side, all Kafka `Header` instances are mapped to `MessageHeaders`. On the outbound side, by default, all `MessageHeaders` are mapped, except `id`, `timestamp`, and the headers that map to `ConsumerRecord` properties.

You can specify which headers are to be mapped for outbound messages, by providing patterns to the mapper. The following listing shows a number of example mappings:

```
public DefaultKafkaHeaderMapper() { ①
    ...
}

public DefaultKafkaHeaderMapper(ObjectMapper objectMapper) { ②
    ...
}

public DefaultKafkaHeaderMapper(String... patterns) { ③
    ...
}

public DefaultKafkaHeaderMapper(ObjectMapper objectMapper, String... patterns) {
④
    ...
}
```

① Uses a default Jackson `ObjectMapper` and maps most headers, as discussed before the example.

② Uses the provided Jackson `ObjectMapper` and maps most headers, as discussed before the example.

③ Uses a default Jackson `ObjectMapper` and maps headers according to the provided patterns.

④ Uses the provided Jackson `ObjectMapper` and maps headers according to the provided patterns.

Patterns are rather simple and can contain a leading wildcard (`), a trailing wildcard, or both (for example, .cat.*`). You can negate patterns with a leading `!`. The first pattern that matches a header name (whether positive or negative) wins.

When you provide your own patterns, we recommend including `!id` and `!timestamp`, since these headers are read-only on the inbound side.

By default, the mapper deserializes only classes in `java.lang` and `java.util`. You can trust other (or all) packages by adding trusted packages with the `addTrustedPackages` method. If you receive messages from untrusted sources, you may wish to add only those packages you trust. To trust all packages, you can use `mapper.addTrustedPackages("*")`.

Mapping `String` header values in a raw form is useful when communicating with systems that are not aware of the mapper's JSON format.

Starting with version 2.2.5, you can specify that certain string-valued headers should not be mapped using JSON, but to/from a raw `byte[]`. The `AbstractKafkaHeaderMapper` has new properties; `mapAllStringsOut` when set to true, all string-valued headers will be converted to `byte[]` using the `charset` property (default `UTF-8`). In addition, there is a property `rawMappedHeaders`, which is a map of

`header name : boolean`; if the map contains a header name, and the header contains a `String` value, it will be mapped as a raw `byte[]` using the charset. This map is also used to map raw incoming `byte[]` headers to `String` using the charset if, and only if, the boolean in the map value is `true`. If the boolean is `false`, or the header name is not in the map with a `true` value, the incoming header is simply mapped as the raw unmapped header.

The following test case illustrates this mechanism.

```
@Test
public void testSpecificStringConvert() {
    DefaultKafkaHeaderMapper mapper = new DefaultKafkaHeaderMapper();
    Map<String, Boolean> rawMappedHeaders = new HashMap<>();
    rawMappedHeaders.put("thisOnesAString", true);
    rawMappedHeaders.put("thisOnesBytes", false);
    mapper.setRawMappedHeaders(rawMappedHeaders);
    Map<String, Object> headersMap = new HashMap<>();
    headersMap.put("thisOnesAString", "thing1");
    headersMap.put("thisOnesBytes", "thing2");
    headersMap.put("alwaysRaw", "thing3".getBytes());
    MessageHeaders headers = new MessageHeaders(headersMap);
    Headers target = new RecordHeaders();
    mapper.fromHeaders(headers, target);
    assertThat(target).containsExactlyInAnyOrder(
            new RecordHeader("thisOnesAString", "thing1".getBytes()),
            new RecordHeader("thisOnesBytes", "thing2".getBytes()),
            new RecordHeader("alwaysRaw", "thing3".getBytes()));
    headersMap.clear();
    mapper.toHeaders(target, headersMap);
    assertThat(headersMap).contains(
            entry("thisOnesAString", "thing1"),
            entry("thisOnesBytes", "thing2".getBytes()),
            entry("alwaysRaw", "thing3".getBytes()));
}
```

By default, the `DefaultKafkaHeaderMapper` is used in the `MessagingMessageConverter` and `BatchMessagingMessageConverter`, as long as Jackson is on the class path.

With the batch converter, the converted headers are available in the `KafkaHeaders.BATCH_CONVERTED_HEADERS` as a `List<Map<String, Object>>` where the map in a position of the list corresponds to the data position in the payload.

If there is no converter (either because Jackson is not present or it is explicitly set to `null`), the headers from the consumer record are provided unconverted in the `KafkaHeaders.NATIVE_HEADERS` header. This header is a `Headers` object (or a `List<Headers>` in the case of the batch converter), where the position in the list corresponds to the data position in the payload).

Certain types are not suitable for JSON serialization, and a simple `toString()` serialization might be preferred for these types. The `DefaultKafkaHeaderMapper` has a method called `addToStringClasses()` that lets you supply the names of classes that should be treated this way for outbound mapping. During inbound mapping, they are mapped as `String`. By default, only `org.springframework.util.MimeType` and `org.springframework.http.MediaType` are mapped this way.

Starting with version 2.3, handling of String-valued headers is simplified. Such headers are no longer JSON encoded, by default (i.e. they do not have enclosing `"…"` added). The type is still added to the JSON_TYPES header so the receiving system can convert back to a String (from `byte[]`). The mapper can handle (decode) headers produced by older versions (it checks for a leading `"`); in this way an application using 2.3 can consume records from older versions.

To be compatible with earlier versions, set `encodeStrings` to `true`, if records produced by a version using 2.3 might be consumed by applications using earlier versions. When all applications are using 2.3 or higher, you can leave the property at its default value of `false`.

```
@Bean
MessagingMessageConverter converter() {
    MessagingMessageConverter converter = new MessagingMessageConverter();
    DefaultKafkaHeaderMapper mapper = new DefaultKafkaHeaderMapper();
    mapper.setEncodeStrings(true);
    converter.setHeaderMapper(mapper);
    return converter;
}
```

If using Spring Boot, it will auto configure this converter bean into the auto-configured `KafkaTemplate`; otherwise you should add this converter to the template.

4.1.19. Null Payloads and Log Compaction of 'Tombstone' Records

When you use Log Compaction, you can send and receive messages with `null` payloads to identify the deletion of a key.

You can also receive `null` values for other reasons, such as a `Deserializer` that might return `null` when it cannot deserialize a value.

To send a `null` payload by using the `KafkaTemplate`, you can pass null into the value argument of the `send()` methods. One exception to this is the `send(Message<?> message)` variant. Since `spring-messaging Message<?>` cannot have a `null` payload, you can use a special payload type called `KafkaNull`, and the framework sends `null`. For convenience, the static `KafkaNull.INSTANCE` is provided.

When you use a message listener container, the received `ConsumerRecord` has a `null value()`.

To configure the `@KafkaListener` to handle `null` payloads, you must use the `@Payload` annotation with `required = false`. If it is a tombstone message for a compacted log, you usually also need the key so that your application can determine which key was "deleted". The following example shows such a configuration:

```
@KafkaListener(id = "deletableListener", topics = "myTopic")
public void listen(@Payload(required = false) String value, @Header(KafkaHeaders
.RECEIVED_MESSAGE_KEY) String key) {
    // value == null represents key deletion
}
```

When you use a class-level `@KafkaListener` with multiple `@KafkaHandler` methods, some additional configuration is needed. Specifically, you need a `@KafkaHandler` method with a `KafkaNull` payload. The following example shows how to configure one:

```
@KafkaListener(id = "multi", topics = "myTopic")
static class MultiListenerBean {

    @KafkaHandler
    public void listen(String cat) {
        ...
    }

    @KafkaHandler
    public void listen(Integer hat) {
        ...
    }

    @KafkaHandler
    public void delete(@Payload(required = false) KafkaNull nul, @Header
(KafkaHeaders.RECEIVED_MESSAGE_KEY) int key) {
        ...
    }

}
```

Note that the argument is `null`, not `KafkaNull`.

See Manually Assigning All Partitions.

4.1.20. Handling Exceptions

This section describes how to handle various exceptions that may arise when you use Spring for Apache Kafka.

Listener Error Handlers

Starting with version 2.0, the `@KafkaListener` annotation has a new attribute: `errorHandler`.

You can use the `errorHandler` to provide the bean name of a `KafkaListenerErrorHandler` implementation. This functional interface has one method, as the following listing shows:

```
@FunctionalInterface
public interface KafkaListenerErrorHandler {

    Object handleError(Message<?> message, ListenerExecutionFailedException
exception) throws Exception;

}
```

You have access to the spring-messaging `Message<?>` object produced by the message converter and the exception that was thrown by the listener, which is wrapped in a `ListenerExecutionFailedException`. The error handler can throw the original or a new exception, which is thrown to the container. Anything returned by the error handler is ignored.

Starting with version 2.7, you can set the `rawRecordHeader` property on the `MessagingMessageConverter` and `BatchMessagingMessageConverter` which causes the raw `ConsumerRecord` to be added to the converted `Message<?>` in the `KafkaHeaders.RAW_DATA` header. This is useful, for example, if you wish to use a `DeadLetterPublishingRecoverer` in a listener error handler. It might be used in a request/reply scenario where you wish to send a failure result to the sender, after some number of retries, after capturing the failed record in a dead letter topic.

```
@Bean
KafkaListenerErrorHandler eh(DeadLetterPublishingRecoverer recoverer) {
    return (msg, ex) -> {
        if (msg.getHeaders().get(KafkaHeaders.DELIVERY_ATTEMPT, Integer.class) >
9) {
            recoverer.accept(msg.getHeaders().get(KafkaHeaders.RAW_DATA,
ConsumerRecord.class), ex);
            return "FAILED";
        }
        throw ex;
    };
}
```

It has a sub-interface (`ConsumerAwareListenerErrorHandler`) that has access to the consumer object, through the following method:

```
Object handleError(Message<?> message, ListenerExecutionFailedException exception,
Consumer<?, ?> consumer);
```

If your error handler implements this interface, you can, for example, adjust the offsets accordingly. For example, to reset the offset to replay the failed message, you could do something like the following:

```
@Bean
public ConsumerAwareListenerErrorHandler listen3ErrorHandler() {
    return (m, e, c) -> {
        this.listen3Exception = e;
        MessageHeaders headers = m.getHeaders();
        c.seek(new org.apache.kafka.common.TopicPartition(
                headers.get(KafkaHeaders.RECEIVED_TOPIC, String.class),
                headers.get(KafkaHeaders.RECEIVED_PARTITION_ID, Integer.class)),
                headers.get(KafkaHeaders.OFFSET, Long.class));
        return null;
    };
}
```

Similarly, you could do something like the following for a batch listener:

```
@Bean
public ConsumerAwareListenerErrorHandler listen10ErrorHandler() {
    return (m, e, c) -> {
        this.listen10Exception = e;
        MessageHeaders headers = m.getHeaders();
        List<String> topics = headers.get(KafkaHeaders.RECEIVED_TOPIC, List.class
);
        List<Integer> partitions = headers.get(KafkaHeaders.RECEIVED_PARTITION_ID,
List.class);
        List<Long> offsets = headers.get(KafkaHeaders.OFFSET, List.class);
        Map<TopicPartition, Long> offsetsToReset = new HashMap<>();
        for (int i = 0; i < topics.size(); i++) {
            int index = i;
            offsetsToReset.compute(new TopicPartition(topics.get(i), partitions
.get(i)),
                    (k, v) -> v == null ? offsets.get(index) : Math.min(v,
offsets.get(index)));
        }
        offsetsToReset.forEach((k, v) -> c.seek(k, v));
        return null;
    };
}
```

This resets each topic/partition in the batch to the lowest offset in the batch.

The preceding two examples are simplistic implementations, and you would probably want more checking in the error handler.

Container Error Handlers

Two error handler interfaces (`ErrorHandler` and `BatchErrorHandler`) are provided. You must configure the appropriate type to match the message listener.

Starting with version 2.5, the default error handlers, when transactions are not being used, are the `SeekToCurrentErrorHandler` and `RecoveringBatchErrorHandler` with default configuration. See Seek To Current Container Error Handlers and Recovering Batch Error Handler. To restore the previous behavior, use the `LoggingErrorHandler` and `BatchLoggingErrorHandler` instead.

When transactions are being used, no error handlers are configured, by default, so that the exception will roll back the transaction. Error handling for transactional containers are handled by the `AfterRollbackProcessor`. If you provide a custom error handler when using transactions, it must throw an exception if you want the transaction rolled back.

Starting with version 2.3.2, these interfaces have a default method `isAckAfterHandle()` which is called by the container to determine whether the offset(s) should be committed if the error handler returns without throwing an exception. Starting with version 2.4, this returns true by default.

Typically, the error handlers provided by the framework will throw an exception when the error is not "handled" (e.g. after performing a seek operation). By default, such exceptions are logged by the container at `ERROR` level. Starting with version 2.5, all the framework error handlers extend `KafkaExceptionLogLevelAware` which allows you to control the level at which these exceptions are logged.

```
/**
 * Set the level at which the exception thrown by this handler is logged.
 * @param logLevel the level (default ERROR).
 */
public void setLogLevel(KafkaException.Level logLevel) {
    ...
}
```

You can specify a global error handler to be used for all listeners in the container factory. The following example shows how to do so:

```
@Bean
public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer,
String>>
        kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    ...
    factory.setErrorHandler(myErrorHandler);
    ...
    return factory;
}
```

Similarly, you can set a global batch error handler:

```
@Bean
public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer,
String>>
        kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    ...
    factory.setBatchErrorHandler(myBatchErrorHandler);
    ...
    return factory;
}
```

By default, if an annotated listener method throws an exception, it is thrown to the container, and the message is handled according to the container configuration.

If you are using Spring Boot, you simply need to add the error handler as a `@Bean` and boot will add it to the auto-configured factory.

Consumer-Aware Container Error Handlers

The container-level error handlers (`ErrorHandler` and `BatchErrorHandler`) have sub-interfaces called `ConsumerAwareErrorHandler` and `ConsumerAwareBatchErrorHandler`. The `handle` method of the `ConsumerAwareErrorHandler` has the following signature:

```
void handle(Exception thrownException, ConsumerRecord<?, ?> data, Consumer<?, ?>
consumer);
```

The `handle` method of the `ConsumerAwareBatchErrorHandler` has the following signature:

```
void handle(Exception thrownException, ConsumerRecords<?, ?> data, Consumer<?, ?>
consumer);
```

Similar to the `@KafkaListener` error handlers, you can reset the offsets as needed, based on the data that failed.

Unlike the listener-level error handlers, however, you should set the `ackOnError` container property to `false` (default) when making adjustments. Otherwise, any pending acks are applied after your repositioning.

Seek To Current Container Error Handlers

If an `ErrorHandler` implements `RemainingRecordsErrorHandler`, the error handler is provided with the failed record and any unprocessed records retrieved by the previous `poll()`. Those records are not passed to the listener after the handler exits. The following listing shows the `RemainingRecordsErrorHandler` interface definition:

```
@FunctionalInterface
public interface RemainingRecordsErrorHandler extends ConsumerAwareErrorHandler {

    void handle(Exception thrownException, List<ConsumerRecord<?, ?>> records,
Consumer<?, ?> consumer);

}
```

This interface lets implementations seek all unprocessed topics and partitions so that the current record (and the others remaining) are retrieved by the next poll. The `SeekToCurrentErrorHandler` does exactly this.

`ackOnError` must be `false` (which is the default). Otherwise, if the container is stopped after the seek, but before the record is reprocessed, the record will be skipped when the container is restarted.

This is now the default error handler for record listeners.

The container commits any pending offset commits before calling the error handler.

To configure the listener container with this handler, add it to the container factory.

For example, with the `@KafkaListener` container factory, you can add `SeekToCurrentErrorHandler` as follows:

```
@Bean
public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory = new
ConcurrentKafkaListenerContainerFactory();
    factory.setConsumerFactory(consumerFactory());
    factory.getContainerProperties().setAckOnError(false);
    factory.getContainerProperties().setAckMode(AckMode.RECORD);
    factory.setErrorHandler(new SeekToCurrentErrorHandler(new FixedBackOff(1000L,
2L)));
    return factory;
}
```

This will retry a delivery up to 2 times (3 delivery attempts) with a back off of 1 second, instead of the default configuration (`FixedBackOff(0L, 9)`). Failures are simply logged after retries are exhausted.

As an example; if the `poll` returns six records (two from each partition 0, 1, 2) and the listener throws an exception on the fourth record, the container acknowledges the first three messages by committing their offsets. The `SeekToCurrentErrorHandler` seeks to offset 1 for partition 1 and offset 0 for partition 2. The next `poll()` returns the three unprocessed records.

If the `AckMode` was `BATCH`, the container commits the offsets for the first two partitions before calling the error handler.

Starting with version 2.2, the `SeekToCurrentErrorHandler` can now recover (skip) a record that keeps failing. By default, after ten failures, the failed record is logged (at the `ERROR` level). You can configure the handler with a custom recoverer (`BiConsumer`) and maximum failures. Using a `FixedBackOff` with `FixedBackOff.UNLIMITED_ATTEMPTS` causes (effectively) infinite retries. The following example configures recovery after three tries:

```
SeekToCurrentErrorHandler errorHandler =
    new SeekToCurrentErrorHandler((record, exception) -> {
        // recover after 3 failures, woth no back off - e.g. send to a dead-letter
topic
    }, new FixedBackOff(0L, 2L));
```

Starting with version 2.2.4, when the container is configured with `AckMode.MANUAL_IMMEDIATE`, the error handler can be configured to commit the offset of recovered records; set the `commitRecovered` property to `true`.

See also Publishing Dead-letter Records.

When using transactions, similar functionality is provided by the `DefaultAfterRollbackProcessor`. See After-rollback Processor.

Starting with version 2.3, the `SeekToCurrentErrorHandler` considers certain exceptions to be fatal, and retries are skipped for such exceptions; the recoverer is invoked on the first failure. The exceptions that are considered fatal, by default, are:

- `DeserializationException`
- `MessageConversionException`
- `ConversionException`
- `MethodArgumentResolutionException`
- `NoSuchMethodException`
- `ClassCastException`

since these exceptions are unlikely to be resolved on a retried delivery.

You can add more exception types to the not-retryable category, or completely replace the map of classified exceptions. See the Javadocs for `SeekToCurrentErrorHandler.setClassifications()` for more information, as well as those for the `spring-retry` `BinaryExceptionClassifier`.

Here is an example that adds `IllegalArgumentException` to the not-retryable exceptions:

```
@Bean
public SeekToCurrentErrorHandler errorHandler(ConsumerRecordRecoverer recoverer) {
    SeekToCurrentErrorHandler handler = new SeekToCurrentErrorHandler(recoverer);
    handler.addNotRetryableException(IllegalArgumentException.class);
    return handler;
}
```

Starting with version 2.7, the error handler can be configured with one or more `RetryListener` s, receiving notifications of retry and recovery progress.

```
@FunctionalInterface
public interface RetryListener {

    void failedDelivery(ConsumerRecord<?, ?> record, Exception ex, int
deliveryAttempt);

    default void recovered(ConsumerRecord<?, ?> record, Exception ex) {
    }

    default void recoveryFailed(ConsumerRecord<?, ?> record, Exception original,
Exception failure) {
    }

}
```

See the javadocs for more information.

The `SeekToCurrentBatchErrorHandler` seeks each partition to the first record in each partition in the batch, so the whole batch is replayed. Also see Committing Offsets for an alternative. Also see Retrying Batch Error Handler. This error handler does not support recovery, because the framework cannot know which message in the batch is failing.

After seeking, an exception that wraps the `ListenerExecutionFailedException` is thrown. This is to cause the transaction to roll back (if transactions are enabled).

Starting with version 2.3, a `BackOff` can be provided to the `SeekToCurrentErrorHandler` and `DefaultAfterRollbackProcessor` so that the consumer thread can sleep for some configurable time between delivery attempts. Spring Framework provides two out of the box `BackOff` s, `FixedBackOff` and `ExponentialBackOff`. The maximum back off time must not exceed the `max.poll.interval.ms` consumer property, to avoid a rebalance.

Previously, the configuration was "maxFailures" (which included the first delivery attempt). When using a `FixedBackOff`, its `maxAttempts` property represents the number of delivery retries (one less than the old `maxFailures` property). Also, `maxFailures=-1` meant retry indefinitely with the old configuration, with a `BackOff` you would set the `maxAttempts` to `Long.MAX_VALUE` for a `FixedBackOff` and leave the `maxElapsedTime` to its default in an `ExponentialBackOff`.

The `SeekToCurrentBatchErrorHandler` can also be configured with a `BackOff` to add a delay between delivery attempts. Generally, you should configure the `BackOff` to never return `STOP`. However, since this error handler has no mechanism to "recover" after retries are exhausted, if the `BackOffExecution` returns `STOP`, the previous interval will be used for all subsequent delays. Again, the maximum delay must be less than the `max.poll.interval.ms` consumer property. Also see Retrying Batch Error Handler.

If the recoverer fails (throws an exception), the failed record will be included in the seeks. Starting with version 2.5.5, if the recoverer fails, the `BackOff` will be reset by default and redeliveries will again go through the back offs before recovery is attempted again. With earlier versions, the `BackOff` was not reset and recovery was re-attempted on the next failure. To revert to the previous behavior, set the error handler's `resetStateOnRecoveryFailure` to `false`.

Starting with version 2.3.2, after a record has been recovered, its offset will be committed (if one of the container `AckMode` s is configured). To revert to the previous behavior, set the error handler's `ackAfterHandle` property to false.

Starting with version 2.6, you can now provide the error handler with a `BiFunction<ConsumerRecord<?, ?>, Exception, BackOff>` to determine the `BackOff` to use, based on the failed record and/or the exception:

```
handler.setBackOffFunction((record, ex) -> { ... });
```

If the function returns `null`, the handler's default `BackOff` will be used.

Starting with version 2.6.3, set `resetStateOnExceptionChange` to `true` and the retry sequence will be restarted (including the selection of a new `BackOff`, if so configured) if the exception type changes between failures. By default, the exception type is not considered.

Also see Delivery Attempts Header.

Starting with version 2.7, while waiting for a `BackOff` interval, the error handler will loop with a short sleep until the desired delay, while checking to see if the container has been stopped, allowing the sleep to exit soon after the `stop()` rather than causing a delay.

Retrying Batch Error Handler

As discussed above, the `SeekToCurrentBatchErrorHandler` has no mechanism to recover after a certain number of failures. One reason for this is there is no guarantee that, when a batch is redelivered, the batch has the same number of records and/or the redelivered records are in the same order. It is impossible, therefore, to maintain retry state for a batch. The `RetryingBatchErrorHandler` takes a different approach. If a batch listener throws an exception, and this error handler is configured, the retries are performed from the in-memory batch of records. In order to avoid a rebalance during an extended retry sequence, the error handler pauses the consumer, polls it before sleeping for the back off, for each retry, and calls the listener again. If/when retries are exhausted, the `ConsumerRecordRecoverer` is called for each record in the batch. If the recoverer throws an exception, or the thread is interrupted during its sleep, a `SeekToCurrentErrorHandler` is invoked so that the batch of records will be redelivered on the next poll. Before exiting, regardless of the outcome, the consumer is resumed.

This error handler cannot be used with transactions.

Starting with version 2.7, while waiting for a `BackOff` interval, the error handler will loop with a short sleep until the desired delay is reached, while checking to see if the container has been stopped, allowing the sleep to exit soon after the `stop()` rather than causing a delay.

Also see Recovering Batch Error Handler.

Recovering Batch Error Handler

As an alternative to the Retrying Batch Error Handler, version 2.5 introduced the `RecoveringBatchErrorHandler`.

This is now the default error handler for batch listeners. The default configuration retries 9 times (10 delivery attempts) with no back off between deliveries.

This error handler works in conjunction with the listener throwing a `BatchListenerFailedException` providing the index in the batch where the failure occurred (or the failed record itself). If the listener throws a different exception, or the index is out of range, the error handler falls back to invoking a `SeekToCurrentBatchErrorHandler` and the whole batch is retried, with no recovery available. The sequence of events is:

- Commit the offsets of the records before the index.
- If retries are not exhausted, perform seeks so that all the remaining records (including the failed record) will be redelivered.
- If retries are exhausted, attempt recovery of the failed record (default log only) and perform seeks so that the remaining records (excluding the failed record) will be redelivered. The recovered record's offset is committed
- If retries are exhausted and recovery fails, seeks are performed as if retries are not exhausted.

The default recoverer logs the failed record after retries are exhausted. You can use a custom recoverer, or one provided by the framework such as the `DeadLetterPublishingRecoverer`.

In all cases, a `BackOff` can be configured to enable a delay between delivery attempts.

Example:

```
@Bean
public RecoveringBatchErrorHandler batchErrorHandler(KafkaTemplate<String, String>
template) {
    DeadLetterPublishingRecoverer recoverer =
            new DeadLetterPublishingRecoverer(template);
    RecoveringBatchErrorHandler errorHandler =
            new RecoveringBatchErrorHandler(recoverer, new FixedBackOff(2L, 5000)
);
}
```

```
@KafkaListener(id = "recovering", topics = "someTopic")
public void listen(List<ConsumerRecord<String, String>> records) {
    records.forEach(record -> {
        try {
            process(record);
        }
        catch (Exception e) {
            throw new BatchListenerFailedException("Failed to process", record);
        }
    });
}
```

For example; say 10 records are in the original batch and no more records are added to the topic during the retries, and the failed record is at index `4` in the list. After the first delivery fails, the offsets for the first 4 records will be committed; the remaing 6 will be redelivered after 5 seconds. Most likely (but not necessarily) the failed record will be at index `0` in the redelivery. If it fails again, it will be retried one more time and, if it again fails, it will be sent to a dead letter topic.

When using a POJO batch listener (e.g. `List<Thing>`), and you don't have the full consumer record to add to the exception, you can just add the index of the record that failed:

```
@KafkaListener(id = "recovering", topics = "someTopic")
public void listen(List<Thing> things) {
    for (int i = 0; i < records.size(); i++) {
        try {
            process(things.get(i));
        }
        catch (Exception e) {
            throw new BatchListenerFailedException("Failed to process", i);
        }
    }
}
```

This error handler cannot be used with transactions

If the recoverer fails (throws an exception), the failed record will be included in the seeks. Starting with version 2.5.5, if the recoverer fails, the `BackOff` will be reset by default and redeliveries will again go through the back offs before recovery is attempted again. With earlier versions, the `BackOff` was not reset and recovery was re-attempted on the next failure. To revert to the previous behavior, set the error handler's `resetStateOnRecoveryFailure` to `false`.

Starting with version 2.6, you can now provide the error handler with a `BiFunction<ConsumerRecord<?, ?>, Exception, BackOff>` to determine the `BackOff` to use, based on

the failed record and/or the exception:

```
handler.setBackOffFunction((record, ex) -> { ... });
```

If the function returns `null`, the handler's default `BackOff` will be used.

Starting with version 2.6.3, set `resetStateOnExceptionChange` to `true` and the retry sequence will be restarted (including the selection of a new `BackOff`, if so configured) if the exception type changes between failures. By default, the exception type is not considered.

Starting with version 2.7, while waiting for a `BackOff` interval, the error handler will loop with a short sleep until the desired delay is reached, while checking to see if the container has been stopped, allowing the sleep to exit soon after the `stop()` rather than causing a delay.

Starting with version 2.7, the error handler can be configured with one or more `RetryListener` s, receiving notifications of retry and recovery progress.

```
@FunctionalInterface
public interface RetryListener {

    void failedDelivery(ConsumerRecord<?, ?> record, Exception ex, int
deliveryAttempt);

    default void recovered(ConsumerRecord<?, ?> record, Exception ex) {
    }

    default void recoveryFailed(ConsumerRecord<?, ?> record, Exception original,
Exception failure) {
    }

}
```

See the javadocs for more information.

Container Stopping Error Handlers

The `ContainerStoppingErrorHandler` (used with record listeners) stops the container if the listener throws an exception. When the `AckMode` is `RECORD`, offsets for already processed records are committed. When the `AckMode` is any manual value, offsets for already acknowledged records are committed. When the `AckMode` is `BATCH`, the entire batch is replayed when the container is restarted (unless transactions are enabled — in which case, only the unprocessed records are re-fetched).

The `ContainerStoppingBatchErrorHandler` (used with batch listeners) stops the container, and the entire batch is replayed when the container is restarted.

After the container stops, an exception that wraps the `ListenerExecutionFailedException` is thrown. This is to cause the transaction to roll back (if transactions are enabled).

After-rollback Processor

When using transactions, if the listener throws an exception (and an error handler, if present, throws an exception), the transaction is rolled back. By default, any unprocessed records (including the failed record) are re-fetched on the next poll. This is achieved by performing `seek` operations in the `DefaultAfterRollbackProcessor`. With a batch listener, the entire batch of records is reprocessed (the container has no knowledge of which record in the batch failed). To modify this behavior, you can configure the listener container with a custom `AfterRollbackProcessor`. For example, with a record-based listener, you might want to keep track of the failed record and give up after some number of attempts, perhaps by publishing it to a dead-letter topic.

Starting with version 2.2, the `DefaultAfterRollbackProcessor` can now recover (skip) a record that keeps failing. By default, after ten failures, the failed record is logged (at the `ERROR` level). You can configure the processor with a custom recoverer (`BiConsumer`) and maximum failures. Setting the `maxFailures` property to a negative number causes infinite retries. The following example configures recovery after three tries:

```
AfterRollbackProcessor<String, String> processor =
    new DefaultAfterRollbackProcessor((record, exception) -> {
        // recover after 3 failures, with no back off - e.g. send to a dead-letter
topic
    }, new FixedBackOff(0L, 2L));
```

When you do not use transactions, you can achieve similar functionality by configuring a `SeekToCurrentErrorHandler`. See Seek To Current Container Error Handlers.

Recovery is not possible with a batch listener, since the framework has no knowledge about which record in the batch keeps failing. In such cases, the application listener must handle a record that keeps failing.

See also Publishing Dead-letter Records.

Starting with version 2.2.5, the `DefaultAfterRollbackProcessor` can be invoked in a new transaction (started after the failed transaction rolls back). Then, if you are using the `DeadLetterPublishingRecoverer` to publish a failed record, the processor will send the recovered record's offset in the original topic/partition to the transaction. To enable this feature, set the `commitRecovered` and `kafkaTemplate` properties on the `DefaultAfterRollbackProcessor`.

If the recoverer fails (throws an exception), the failed record will be included in the seeks. Starting with version 2.5.5, if the recoverer fails, the `BackOff` will be reset by default and redeliveries will again go through the back offs before recovery is attempted again. With earlier versions, the `BackOff` was not reset and recovery was re-attempted on the next failure. To revert to the previous behavior, set the processor's `resetStateOnRecoveryFailure` property to `false`.

Starting with version 2.6, you can now provide the processor with a `BiFunction<ConsumerRecord<?, ?>, Exception, BackOff>` to determine the `BackOff` to use, based on the failed record and/or the exception:

```
handler.setBackOffFunction((record, ex) -> { ... });
```

If the function returns `null`, the processor's default `BackOff` will be used.

Starting with version 2.6.3, set `resetStateOnExceptionChange` to `true` and the retry sequence will be restarted (including the selection of a new `BackOff`, if so configured) if the exception type changes between failures. By default, the exception type is not considered.

Starting with version 2.3.1, similar to the `SeekToCurrentErrorHandler`, the `DefaultAfterRollbackProcessor` considers certain exceptions to be fatal, and retries are skipped for such exceptions; the recoverer is invoked on the first failure. The exceptions that are considered fatal, by default, are:

- `DeserializationException`
- `MessageConversionException`
- `ConversionException`
- `MethodArgumentResolutionException`
- `NoSuchMethodException`
- `ClassCastException`

since these exceptions are unlikely to be resolved on a retried delivery.

You can add more exception types to the not-retryable category, or completely replace the map of classified exceptions. See the Javadocs for `DefaultAfterRollbackProcessor.setClassifications()` for more information, as well as those for the `spring-retry` `BinaryExceptionClassifier`.

Here is an example that adds `IllegalArgumentException` to the not-retryable exceptions:

```
@Bean
public DefaultAfterRollbackProcessor errorHandler(BiConsumer<ConsumerRecord<?, ?>,
Exception> recoverer) {
    DefaultAfterRollbackProcessor processor = new DefaultAfterRollbackProcessor
(recoverer);
    processor.addNotRetryableException(IllegalArgumentException.class);
    return processor;
}
```

Also see Delivery Attempts Header.

With current `kafka-clients`, the container cannot detect whether a `ProducerFencedException` is caused by a rebalance or if the producer's `transactional.id` has been revoked due to a timeout or expiry. Because, in most cases, it is caused by a rebalance, the container does not call the `AfterRollbackProcessor` (because it's not appropriate to seek the partitions because we no longer are assigned them). If you ensure the timeout is large enough to process each transaction and periodically perform an "empty" transaction (e.g. via a `ListenerContainerIdleEvent`) you can avoid fencing due to timeout and expiry. Or, you can set the `stopContainerWhenFenced` container property to `true` and the container will stop, avoiding the loss of records. You can consume a `ConsumerStoppedEvent` and check the `Reason` property for `FENCED` to detect this condition. Since the event also has a reference to the container, you can restart the container using this event.

Starting with version 2.7, while waiting for a `BackOff` interval, the error handler will loop with a short sleep until the desired delay is reached, while checking to see if the container has been stopped, allowing the sleep to exit soon after the `stop()` rather than causing a delay.

Starting with version 2.7, the processor can be configured with one or more `RetryListener` s, receiving notifications of retry and recovery progress.

```
@FunctionalInterface
public interface RetryListener {

    void failedDelivery(ConsumerRecord<?, ?> record, Exception ex, int
deliveryAttempt);

    default void recovered(ConsumerRecord<?, ?> record, Exception ex) {
    }

    default void recoveryFailed(ConsumerRecord<?, ?> record, Exception original,
Exception failure) {
    }

}
```

See the javadocs for more information.

Delivery Attempts Header

The following applies to record listeners only, not batch listeners.

Starting with version 2.5, when using an `ErrorHandler` or `AfterRollbackProcessor` that implements `DeliveryAttemptAware`, it is possible to enable the addition of the `KafkaHeaders.DELIVERY_ATTEMPT` header (`kafka_deliveryAttempt`) to the record. The value of this header is an incrementing integer starting at 1. When receiving a raw `ConsumerRecord<?, ?>` the integer is in a `byte[4]`.

```
int delivery = ByteBuffer.wrap(record.headers()
    .lastHeader(KafkaHeaders.DELIVERY_ATTEMPT).value())
    .getInt()
```

When using `@KafkaListener` with the `DefaultKafkaHeaderMapper` or `SimpleKafkaHeaderMapper`, it can be obtained by adding `@Header(KafkaHeaders.DELIVERY_ATTEMPT) int delivery` as a parameter to the listener method.

To enable population of this header, set the container property `deliveryAttemptHeader` to `true`. It is disabled by default to avoid the (small) overhead of looking up the state for each record and adding the header.

The `SeekToCurrentErrorHandler` and `DefaultAfterRollbackProcessor` support this feature.

Publishing Dead-letter Records

As discussed earlier, you can configure the `SeekToCurrentErrorHandler` and `DefaultAfterRollbackProcessor` (as well as the `RecoveringBatchErrorHandler`) with a record recoverer when the maximum number of failures is reached for a record. The framework provides the

DeadLetterPublishingRecoverer, which publishes the failed message to another topic. The recoverer requires a KafkaTemplate<Object, Object>, which is used to send the record. You can also, optionally, configure it with a BiFunction<ConsumerRecord<?, ?>, Exception, TopicPartition>, which is called to resolve the destination topic and partition.

By default, the dead-letter record is sent to a topic named <originalTopic>.DLT (the original topic name suffixed with .DLT) and to the same partition as the original record. Therefore, when you use the default resolver, the dead-letter topic **must have at least as many partitions as the original topic.**

If the returned TopicPartition has a negative partition, the partition is not set in the ProducerRecord, so the partition is selected by Kafka. Starting with version 2.2.4, any ListenerExecutionFailedException (thrown, for example, when an exception is detected in a @KafkaListener method) is enhanced with the groupId property. This allows the destination resolver to use this, in addition to the information in the ConsumerRecord to select the dead letter topic.

The following example shows how to wire a custom destination resolver:

```
DeadLetterPublishingRecoverer recoverer = new DeadLetterPublishingRecoverer
(template,
        (r, e) -> {
            if (e instanceof FooException) {
                return new TopicPartition(r.topic() + ".Foo.failures", r.
partition());
            }
            else {
                return new TopicPartition(r.topic() + ".other.failures", r
.partition());
            }
        });
ErrorHandler errorHandler = new SeekToCurrentErrorHandler(recoverer, new
FixedBackOff(0L, 2L));
```

The record sent to the dead-letter topic is enhanced with the following headers:

- KafkaHeaders.DLT_EXCEPTION_FQCN: The Exception class name.
- KafkaHeaders.DLT_EXCEPTION_STACKTRACE: The Exception stack trace.
- KafkaHeaders.DLT_EXCEPTION_MESSAGE: The Exception message.
- KafkaHeaders.DLT_KEY_EXCEPTION_FQCN: The Exception class name (key deserialization errors only).
- KafkaHeaders.DLT_KEY_EXCEPTION_STACKTRACE: The Exception stack trace (key deserialization errors only).
- KafkaHeaders.DLT_KEY_EXCEPTION_MESSAGE: The Exception message (key deserialization errors only).

- KafkaHeaders.DLT_ORIGINAL_TOPIC: The original topic.
- KafkaHeaders.DLT_ORIGINAL_PARTITION: The original partition.
- KafkaHeaders.DLT_ORIGINAL_OFFSET: The original offset.
- KafkaHeaders.DLT_ORIGINAL_TIMESTAMP: The original timestamp.
- KafkaHeaders.DLT_ORIGINAL_TIMESTAMP_TYPE: The original timestamp type.

There are two mechanisms to add more headers.

1. Subclass the recoverer and override createProducerRecord() - call super.createProducerRecord() and add more headers.
2. Provide a BiFunction to receive the consumer record and exception, returning a Headers object; headers from there will be copied to the final producer record. Use setHeadersFunction() to set the BiFunction.

The second is simpler to implement but the first has more information available, including the already assembled standard headers.

Starting with version 2.3, when used in conjunction with an ErrorHandlingDeserializer, the publisher will restore the record value(), in the dead-letter producer record, to the original value that failed to be deserialized. Previously, the value() was null and user code had to decode the DeserializationException from the message headers. In addition, you can provide multiple KafkaTemplate s to the publisher; this might be needed, for example, if you want to publish the byte[] from a DeserializationException, as well as values using a different serializer from records that were deserialized successfully. Here is an example of configuring the publisher with KafkaTemplate s that use a String and byte[] serializer:

```
@Bean
public DeadLetterPublishingRecoverer publisher(KafkaTemplate<?, ?> stringTemplate,
        KafkaTemplate<?, ?> bytesTemplate) {

    Map<Class<?>, KafkaTemplate<?, ?>> templates = new LinkedHashMap<>();
    templates.put(String.class, stringTemplate);
    templates.put(byte[].class, bytesTemplate);
    return new DeadLetterPublishingRecoverer(templates);
}
```

The publisher uses the map keys to locate a template that is suitable for the value() about to be published. A LinkedHashMap is recommended so that the keys are examined in order.

When publishing null values, when there are multiple templates, the recoverer will look for a template for the Void class; if none is present, the first template from the values().iterator() will be used.

Since 2.7 you can use the setFailIfSendResultIsError method so that an exception is thrown when message publishing fails. You can also set a timeout for the verification of the sender success with

`setWaitForSendResultTimeout`.

If the recoverer fails (throws an exception), the failed record will be included in the seeks. Starting with version 2.5.5, if the recoverer fails, the `BackOff` will be reset by default and redeliveries will again go through the back offs before recovery is attempted again. With earlier versions, the `BackOff` was not reset and recovery was re-attempted on the next failure. To revert to the previous behavior, set the error handler's `resetStateOnRecoveryFailure` property to `false`.

Starting with version 2.6.3, set `resetStateOnExceptionChange` to `true` and the retry sequence will be restarted (including the selection of a new `BackOff`, if so configured) if the exception type changes between failures. By default, the exception type is not considered.

Starting with version 2.3, the recoverer can also be used with Kafka Streams - see Recovery from Deserialization Exceptions for more information.

The `ErrorHandlingDeserializer` adds the deserialization exception(s) in headers `ErrorHandlingDeserializer.VALUE_DESERIALIZER_EXCEPTION_HEADER` and `ErrorHandlingDeserializer.KEY_DESERIALIZER_EXCEPTION_HEADER` (using java serialization). By default, these headers are not retained in the message published to the dead letter topic. Starting with version 2.7, if both the key and value fail deserialization, the original values of both are populated in the record sent to the DLT.

If incoming records are dependent on each other, but may arrive out of order, it may be useful to republish a failed record to the tail of the original topic (for some number of times), instead of sending it directly to the dead letter topic. See this Stack Overflow Question for an example.

The following error handler configuration will do exactly that:

```
@Bean
public ErrorHandler eh(KafkaOperations<String, String> template) {
    return new SeekToCurrentErrorHandler(new DeadLetterPublishingRecoverer
(template,
            (rec, ex) -> {
                org.apache.kafka.common.header.Header retries = rec.headers()
.lastHeader("retries");
                if (retries == null) {
                    retries = new RecordHeader("retries", new byte[] { 1 });
                    rec.headers().add(retries);
                }
                else {
                    retries.value()[0]++;
                }
                return retries.value()[0] > 5
                        ? new TopicPartition("topic.DLT", rec.partition())
                        : new TopicPartition("topic", rec.partition());
            }), new FixedBackOff(0L, 0L));
}
```

Starting with version 2.7, the recoverer checks that the partition selected by the destination resolver actually exists. If the partition is not present, the partition in the `ProducerRecord` is set to `null`, allowing the `KafkaProducer` to select the partition. You can disable this check by setting the `verifyPartition` property to `false`.

`ExponentialBackOffWithMaxRetries` Implementation

Spring Framework provides a number of `BackOff` implementations. By default, the `ExponentialBackOff` will retry indefinitely; to give up after some number of retry attempts requires calculating the `maxElapsedTime`. Since version 2.7.3, Spring for Apache Kafka provides the `ExponentialBackOffWithMaxRetries` which is a subclass that receives the `maxRetries` property and automatically calculates the `maxElapsedTime`, which is a little more convenient.

```
@Bean
SeekToCurrentErrorHandler handler() {
    ExponentialBackOffWithMaxRetries bo = new ExponentialBackOffWithMaxRetries(6);
    bo.setInitialInterval(1_000L);
    bo.setMultiplier(2.0);
    bo.setMaxInterval(10_000L);
    return new SeekToCurrentErrorHandler(myRecoverer, bo);
}
```

This will retry after `1, 2, 4, 8, 10, 10` seconds, before calling the recoverer.

4.1.21. JAAS and Kerberos

Starting with version 2.0, a `KafkaJaasLoginModuleInitializer` class has been added to assist with Kerberos configuration. You can add this bean, with the desired configuration, to your application context. The following example configures such a bean:

```
@Bean
public KafkaJaasLoginModuleInitializer jaasConfig() throws IOException {
    KafkaJaasLoginModuleInitializer jaasConfig = new
KafkaJaasLoginModuleInitializer();
    jaasConfig.setControlFlag("REQUIRED");
    Map<String, String> options = new HashMap<>();
    options.put("useKeyTab", "true");
    options.put("storeKey", "true");
    options.put("keyTab", "/etc/security/keytabs/kafka_client.keytab");
    options.put("principal", "kafka-client-1@EXAMPLE.COM");
    jaasConfig.setOptions(options);
    return jaasConfig;
}
```

4.2. Apache Kafka Streams Support

Starting with version 1.1.4, Spring for Apache Kafka provides first-class support for Kafka Streams. To use it from a Spring application, the `kafka-streams` jar must be present on classpath. It is an optional dependency of the Spring for Apache Kafka project and is not downloaded transitively.

4.2.1. Basics

The reference Apache Kafka Streams documentation suggests the following way of using the API:

```
// Use the builders to define the actual processing topology, e.g. to specify
// from which input topics to read, which stream operations (filter, map, etc.)
// should be called, and so on.

StreamsBuilder builder = ...;  // when using the Kafka Streams DSL

// Use the configuration to tell your application where the Kafka cluster is,
// which serializers/deserializers to use by default, to specify security
settings,
// and so on.
StreamsConfig config = ...;

KafkaStreams streams = new KafkaStreams(builder, config);

// Start the Kafka Streams instance
streams.start();

// Stop the Kafka Streams instance
streams.close();
```

So, we have two main components:

- `StreamsBuilder`: With an API to build `KStream` (or `KTable`) instances.
- `KafkaStreams`: To manage the lifecycle of those instances.

All `KStream` instances exposed to a `KafkaStreams` instance by a single `StreamsBuilder` are started and stopped at the same time, even if they have different logic. In other words, all streams defined by a `StreamsBuilder` are tied with a single lifecycle control. Once a `KafkaStreams` instance has been closed by `streams.close()`, it cannot be restarted. Instead, a new `KafkaStreams` instance to restart stream processing must be created.

4.2.2. Spring Management

To simplify using Kafka Streams from the Spring application context perspective and use the lifecycle management through a container, the Spring for Apache Kafka introduces `StreamsBuilderFactoryBean`. This is an `AbstractFactoryBean` implementation to expose a `StreamsBuilder` singleton instance as a bean. The following example creates such a bean:

```
@Bean
public FactoryBean<StreamsBuilder> myKStreamBuilder(KafkaStreamsConfiguration
streamsConfig) {
    return new StreamsBuilderFactoryBean(streamsConfig);
}
```

Starting with version 2.2, the stream configuration is now provided as a `KafkaStreamsConfiguration` object rather than a `StreamsConfig`.

The `StreamsBuilderFactoryBean` also implements `SmartLifecycle` to manage the lifecycle of an internal `KafkaStreams` instance. Similar to the Kafka Streams API, you must define the `KStream` instances before you start the `KafkaStreams`. That also applies for the Spring API for Kafka Streams. Therefore, when you use default `autoStartup = true` on the `StreamsBuilderFactoryBean`, you must declare `KStream` instances on the `StreamsBuilder` before the application context is refreshed. For example, `KStream` can be a regular bean definition, while the Kafka Streams API is used without any impacts. The following example shows how to do so:

```
@Bean
public KStream<?, ?> kStream(StreamsBuilder kStreamBuilder) {
    KStream<Integer, String> stream = kStreamBuilder.stream(STREAMING_TOPIC1);
    // Fluent KStream API
    return stream;
}
```

If you would like to control the lifecycle manually (for example, stopping and starting by some condition), you can reference the `StreamsBuilderFactoryBean` bean directly by using the factory bean (`&`) prefix. Since `StreamsBuilderFactoryBean` use its internal `KafkaStreams` instance, it is safe to stop and restart it again. A new `KafkaStreams` is created on each `start()`. You might also consider using different `StreamsBuilderFactoryBean` instances, if you would like to control the lifecycles for `KStream` instances separately.

You also can specify `KafkaStreams.StateListener`, `Thread.UncaughtExceptionHandler`, and `StateRestoreListener` options on the `StreamsBuilderFactoryBean`, which are delegated to the internal `KafkaStreams` instance. Also, apart from setting those options indirectly on `StreamsBuilderFactoryBean`, starting with *version 2.1.5*, you can use a `KafkaStreamsCustomizer` callback interface to configure an inner `KafkaStreams` instance. Note that `KafkaStreamsCustomizer` overrides the options provided by `StreamsBuilderFactoryBean`. If you need to perform some `KafkaStreams` operations directly, you can access that internal `KafkaStreams` instance by using `StreamsBuilderFactoryBean.getKafkaStreams()`. You can autowire `StreamsBuilderFactoryBean` bean by type, but you should be sure to use the full type in the bean definition, as the following example shows:

```
@Bean
public StreamsBuilderFactoryBean myKStreamBuilder(KafkaStreamsConfiguration
streamsConfig) {
    return new StreamsBuilderFactoryBean(streamsConfig);
}
...
@Autowired
private StreamsBuilderFactoryBean myKStreamBuilderFactoryBean;
```

Alternatively, you can add `@Qualifier` for injection by name if you use interface bean definition. The following example shows how to do so:

```
@Bean
public FactoryBean<StreamsBuilder> myKStreamBuilder(KafkaStreamsConfiguration
streamsConfig) {
    return new StreamsBuilderFactoryBean(streamsConfig);
}
...
@Autowired
@Qualifier("&myKStreamBuilder")
private StreamsBuilderFactoryBean myKStreamBuilderFactoryBean;
```

Starting with version 2.4.1, the factory bean has a new property `infrastructureCustomizer` with type `KafkaStreamsInfrastructureCustomizer`; this allows customization of the `StreamsBuilder` (e.g. to add a state store) and/or the `Topology` before the stream is created.

```
public interface KafkaStreamsInfrastructureCustomizer {

    void configureBuilder(StreamsBuilder builder);

    void configureTopology(Topology topology);

}
```

Default no-op implementations are provided to avoid having to implement both methods if one is not required.

A `CompositeKafkaStreamsInfrastructureCustomizer` is provided, for when you need to apply multiple customizers.

4.2.3. KafkaStreams Micrometer Support

Introduced in version 2.5.3, you can configure a `KafkaStreamsMicrometerListener` to automatically register micrometer meters for the `KafkaStreams` object managed by the factory bean:

```
streamsBuilderFactoryBean.setListener(new KafkaStreamsMicrometerListener
(meterRegistry,
        Collections.singletonList(new ImmutableTag("customTag", "customTagValue")
)));
```

4.2.4. Streams JSON Serialization and Deserialization

For serializing and deserializing data when reading or writing to topics or state stores in JSON format, Spring for Apache Kafka provides a `JsonSerde` implementation that uses JSON, delegating to the `JsonSerializer` and `JsonDeserializer` described in Serialization, Deserialization, and Message Conversion. The `JsonSerde` implementation provides the same configuration options through its constructor (target type or `ObjectMapper`). In the following example, we use the `JsonSerde` to serialize and deserialize the `Cat` payload of a Kafka stream (the `JsonSerde` can be used in a similar fashion wherever an instance is required):

```
stream.through(Serdes.Integer(), new JsonSerde<>(Cat.class), "cats");
```

When constructing the serializer/deserializer programmatically for use in the producer/consumer factory, since version 2.3, you can use the fluent API, which simplifies configuration.

```
stream.through(new JsonSerde<>(MyKeyType.class)
        .forKeys()
        .noTypeInfo(),
    new JsonSerde<>(MyValueType.class)
        .noTypeInfo(),
    "myTypes");
```

4.2.5. Using `KafkaStreamBrancher`

The `KafkaStreamBrancher` class introduces a more convenient way to build conditional branches on top of `KStream`.

Consider the following example that does not use `KafkaStreamBrancher`:

```
KStream<String, String>[] branches = builder.stream("source").branch(
      (key, value) -> value.contains("A"),
      (key, value) -> value.contains("B"),
      (key, value) -> true
     );
branches[0].to("A");
branches[1].to("B");
branches[2].to("C");
```

The following example uses `KafkaStreamBrancher`:

```
new KafkaStreamBrancher<String, String>()
   .branch((key, value) -> value.contains("A"), ks -> ks.to("A"))
   .branch((key, value) -> value.contains("B"), ks -> ks.to("B"))
   //default branch should not necessarily be defined in the end of the chain!
   .defaultBranch(ks -> ks.to("C"))
   .onTopOf(builder.stream("source"));
   //onTopOf method returns the provided stream so we can continue with method
chaining
```

4.2.6. Configuration

To configure the Kafka Streams environment, the `StreamsBuilderFactoryBean` requires a `KafkaStreamsConfiguration` instance. See the Apache Kafka documentation for all possible options.

Starting with version 2.2, the stream configuration is now provided as a `KafkaStreamsConfiguration` object, rather than as a `StreamsConfig`.

To avoid boilerplate code for most cases, especially when you develop microservices, Spring for Apache Kafka provides the `@EnableKafkaStreams` annotation, which you should place on a `@Configuration` class. All you need is to declare a `KafkaStreamsConfiguration` bean named `defaultKafkaStreamsConfig`. A `StreamsBuilderFactoryBean` bean, named `defaultKafkaStreamsBuilder`, is automatically declared in the application context. You can declare and use any additional `StreamsBuilderFactoryBean` beans as well. You can perform additional customization of that bean, by providing a bean that implements `StreamsBuilderFactoryBeanConfigurer`. If there are multiple such beans, they will be applied according to their `Ordered.order` property.

By default, when the factory bean is stopped, the `KafkaStreams.cleanUp()` method is called. Starting with version 2.1.2, the factory bean has additional constructors, taking a `CleanupConfig` object that has properties to let you control whether the `cleanUp()` method is called during `start()` or `stop()` or neither. Starting with version 2.7, the default is to never clean up local state.

4.2.7. Header Enricher

Version 2.3 added the `HeaderEnricher` implementation of `Transformer`. This can be used to add headers within the stream processing; the header values are SpEL expressions; the root object of the expression evaluation has 3 properties:

- `context` - the `ProcessorContext`, allowing access to the current record metadata
- `key` - the key of the current record
- `value` - the value of the current record

The expressions must return a `byte[]` or a `String` (which will be converted to `byte[]` using `UTF-8`).

To use the enricher within a stream:

```
.transform(() -> enricher)
```

The transformer does not change the `key` or `value`; it simply adds headers.

If your stream is multi-threaded, you need a new instance for each record.

```
.transform(() -> new HeaderEnricher<..., ...>(expressionMap))
```

Here is a simple example, adding one literal header and one variable:

```
Map<String, Expression> headers = new HashMap<>();
headers.put("header1", new LiteralExpression("value1"));
SpelExpressionParser parser = new SpelExpressionParser();
headers.put("header2", parser.parseExpression("context.timestamp() + ' @' +
context.offset()"));
HeaderEnricher<String, String> enricher = new HeaderEnricher<>(headers);
KStream<String, String> stream = builder.stream(INPUT);
stream
        .transform(() -> enricher)
        .to(OUTPUT);
```

4.2.8. `MessagingTransformer`

Version 2.3 added the `MessagingTransformer` this allows a Kafka Streams topology to interact with a Spring Messaging component, such as a Spring Integration flow. The transformer requires an implementation of `MessagingFunction`.

```
@FunctionalInterface
public interface MessagingFunction {

    Message<?> exchange(Message<?> message);

}
```

Spring Integration automatically provides an implementation using its `GatewayProxyFactoryBean`. It also requires a `MessagingMessageConverter` to convert the key, value and metadata (including headers) to/from a Spring Messaging `Message<?>`. See [Calling a Spring Integration Flow from a `KStream`] for more information.

4.2.9. Recovery from Deserialization Exceptions

Version 2.3 introduced the `RecoveringDeserializationExceptionHandler` which can take some action when a deserialization exception occurs. Refer to the Kafka documentation about `DeserializationExceptionHandler`, of which the `RecoveringDeserializationExceptionHandler` is an implementation. The `RecoveringDeserializationExceptionHandler` is configured with a `ConsumerRecordRecoverer` implementation. The framework provides the `DeadLetterPublishingRecoverer` which sends the failed record to a dead-letter topic. See Publishing Dead-letter Records for more information about this recoverer.

To configure the recoverer, add the following properties to your streams configuration:

```
@Bean(name = KafkaStreamsDefaultConfiguration.DEFAULT_STREAMS_CONFIG_BEAN_NAME)
public KafkaStreamsConfiguration kStreamsConfigs() {
    Map<String, Object> props = new HashMap<>();
    ...
    props.put(StreamsConfig.
DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,
            RecoveringDeserializationExceptionHandler.class);
    props.put(RecoveringDeserializationExceptionHandler
.KSTREAM_DESERIALIZATION_RECOVERER, recoverer());
    ...
    return new KafkaStreamsConfiguration(props);
}

@Bean
public DeadLetterPublishingRecoverer recoverer() {
    return new DeadLetterPublishingRecoverer(kafkaTemplate(),
            (record, ex) -> new TopicPartition("recovererDLQ", -1));
}
```

Of course, the `recoverer()` bean can be your own implementation of `ConsumerRecordRecoverer`.

4.2.10. Kafka Streams Example

The following example combines all the topics we have covered in this chapter:

```
@Configuration
@EnableKafka
@EnableKafkaStreams
public static class KafkaStreamsConfig {

    @Bean(name = KafkaStreamsDefaultConfiguration.
DEFAULT_STREAMS_CONFIG_BEAN_NAME)
    public KafkaStreamsConfiguration kStreamsConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "testStreams");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.Integer()
.getClass().getName());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String()
.getClass().getName());
        props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
WallclockTimestampExtractor.class.getName());
        return new KafkaStreamsConfiguration(props);
    }

    @Bean
    public StreamsBuilderFactoryBeanConfigurer configurer() {
        return fb -> fb.setStateListener((newState, oldState) -> {
            System.out.println("State transition from " + oldState + " to " +
newState);
        });
    }

    @Bean
    public KStream<Integer, String> kStream(StreamsBuilder kStreamBuilder) {
        KStream<Integer, String> stream = kStreamBuilder.stream("streamingTopic1"
);
        stream
                .mapValues((ValueMapper<String, String>) String::toUpperCase)
                .groupByKey()
                .windowedBy(TimeWindows.of(Duration.ofMillis(1000)))
                .reduce((String value1, String value2) -> value1 + value2,
                        Named.as("windowStore"))
                .toStream()
                .map((windowedId, value) -> new KeyValue<>(windowedId.key(),
value))
                .filter((i, s) -> s.length() > 40)
                .to("streamingTopic2");

        stream.print(Printed.toSysOut());

        return stream;
    }

}
```

4.3. Testing Applications

The `spring-kafka-test` jar contains some useful utilities to assist with testing your applications.

4.3.1. KafkaTestUtils

`o.s.kafka.test.utils.KafkaTestUtils` provides a number of static helper methods to consume records, retrieve various record offsets, and others. Refer to its Javadocs for complete details.

4.3.2. JUnit

`o.s.kafka.test.utils.KafkaTestUtils` also provides some static methods to set up producer and consumer properties. The following listing shows those method signatures:

```
/**
 * Set up test properties for an {@code <Integer, String>} consumer.
 * @param group the group id.
 * @param autoCommit the auto commit.
 * @param embeddedKafka a {@link EmbeddedKafkaBroker} instance.
 * @return the properties.
 */
public static Map<String, Object> consumerProps(String group, String autoCommit,
                                        EmbeddedKafkaBroker embeddedKafka) { ... }

/**
 * Set up test properties for an {@code <Integer, String>} producer.
 * @param embeddedKafka a {@link EmbeddedKafkaBroker} instance.
 * @return the properties.
 */
public static Map<String, Object> producerProps(EmbeddedKafkaBroker embeddedKafka)
{ ... }
```

Starting with version 2.5, the `consumerProps` method sets the `ConsumerConfig.AUTO_OFFSET_RESET_CONFIG` to `earliest`. This is because, in most cases, you want the consumer to consume any messages sent in a test case. The `ConsumerConfig` default is `latest` which means that messages already sent by a test, before the consumer starts, will not receive those records. To revert to the previous behavior, set the property to `latest` after calling the method.

A JUnit 4 `@Rule` wrapper for the `EmbeddedKafkaBroker` is provided to create an embedded Kafka and an embedded Zookeeper server. (See @EmbeddedKafka Annotation for information about using `@EmbeddedKafka` with JUnit 5). The following listing shows the signatures of those methods:

```
/**
 * Create embedded Kafka brokers.
 * @param count the number of brokers.
 * @param controlledShutdown passed into TestUtils.createBrokerConfig.
 * @param topics the topics to create (2 partitions per).
 */
public EmbeddedKafkaRule(int count, boolean controlledShutdown, String... topics)
{ ... }

/**
 *
 * Create embedded Kafka brokers.
 * @param count the number of brokers.
 * @param controlledShutdown passed into TestUtils.createBrokerConfig.
 * @param partitions partitions per topic.
 * @param topics the topics to create.
 */
public EmbeddedKafkaRule(int count, boolean controlledShutdown, int partitions,
String... topics) { ... }
```

The `EmbeddedKafkaBroker` class has a utility method that lets you consume for all the topics it created. The following example shows how to use it:

```
Map<String, Object> consumerProps = KafkaTestUtils.consumerProps("testT", "false",
embeddedKafka);
DefaultKafkaConsumerFactory<Integer, String> cf = new DefaultKafkaConsumerFactory
<Integer, String>(
        consumerProps);
Consumer<Integer, String> consumer = cf.createConsumer();
embeddedKafka.consumeFromAllEmbeddedTopics(consumer);
```

The `KafkaTestUtils` has some utility methods to fetch results from the consumer. The following listing shows those method signatures:

```
/**
 * Poll the consumer, expecting a single record for the specified topic.
 * @param consumer the consumer.
 * @param topic the topic.
 * @return the record.
 * @throws org.junit.ComparisonFailure if exactly one record is not received.
 */
public static <K, V> ConsumerRecord<K, V> getSingleRecord(Consumer<K, V> consumer,
String topic) { ... }

/**
 * Poll the consumer for records.
 * @param consumer the consumer.
 * @return the records.
 */
public static <K, V> ConsumerRecords<K, V> getRecords(Consumer<K, V> consumer) {
... }
```

The following example shows how to use `KafkaTestUtils`:

```
...
template.sendDefault(0, 2, "bar");
ConsumerRecord<Integer, String> received = KafkaTestUtils.getSingleRecord(
consumer, "topic");
...
```

When the embedded Kafka and embedded Zookeeper server are started by the `EmbeddedKafkaBroker`, a system property named `spring.embedded.kafka.brokers` is set to the address of the Kafka brokers and a system property named `spring.embedded.zookeeper.connect` is set to the address of Zookeeper. Convenient constants (`EmbeddedKafkaBroker.SPRING_EMBEDDED_KAFKA_BROKERS` and `EmbeddedKafkaBroker.SPRING_EMBEDDED_ZOOKEEPER_CONNECT`) are provided for this property.

With the `EmbeddedKafkaBroker.brokerProperties(Map<String, String>)`, you can provide additional properties for the Kafka servers. See Kafka Config for more information about possible broker properties.

4.3.3. Configuring Topics

The following example configuration creates topics called `cat` and `hat` with five partitions, a topic called `thing1` with 10 partitions, and a topic called `thing2` with 15 partitions:

```
public class MyTests {

    @ClassRule
    private static EmbeddedKafkaRule embeddedKafka = new EmbeddedKafkaRule(1,
false, 5, "cat", "hat");

    @Test
    public void test() {
        embeddedKafkaRule.getEmbeddedKafka()
              .addTopics(new NewTopic("thing1", 10, (short) 1), new NewTopic(
"thing2", 15, (short) 1));
        ...
      }

}
```

By default, `addTopics` will throw an exception when problems arise (such as adding a topic that already exists). Version 2.6 added a new version of that method that returns a `Map<String, Exception>`; the key is the topic name and the value is `null` for success, or an `Exception` for a failure.

4.3.4. Using the Same Brokers for Multiple Test Classes

There is no built-in support for doing so, but you can use the same broker for multiple test classes with something similar to the following:

```
public final class EmbeddedKafkaHolder {

    private static EmbeddedKafkaRule embeddedKafka = new EmbeddedKafkaRule(1,
false);

    private static boolean started;

    public static EmbeddedKafkaRule getEmbeddedKafka() {
        if (!started) {
            try {
                embeddedKafka.before();
            }
            catch (Exception e) {
                throw new KafkaException(e);
            }
            started = true;
        }
        return embeddedKafka;
    }

    private EmbeddedKafkaHolder() {
        super();
    }

}
```

Then, in each test class, you can use something similar to the following:

```
static {
    EmbeddedKafkaHolder.getEmbeddedKafka().addTopics(topic1, topic2);
}

private static EmbeddedKafkaRule embeddedKafka = EmbeddedKafkaHolder
.getEmbeddedKafka();
```

The preceding example provides no mechanism for shutting down the brokers when all tests are complete. This could be a problem if, say, you run your tests in a Gradle daemon. You should not use this technique in such a situation, or you should use something to call `destroy()` on the `EmbeddedKafkaBroker` when your tests are complete.

4.3.5. @EmbeddedKafka Annotation

We generally recommend that you use the rule as a `@ClassRule` to avoid starting and stopping the

broker between tests (and use a different topic for each test). Starting with version 2.0, if you use Spring's test application context caching, you can also declare a `EmbeddedKafkaBroker` bean, so a single broker can be used across multiple test classes. For convenience, we provide a test class-level annotation called `@EmbeddedKafka` to register the `EmbeddedKafkaBroker` bean. The following example shows how to use it:

```
@RunWith(SpringRunner.class)
@DirtiesContext
@EmbeddedKafka(partitions = 1,
         topics = {
                 KafkaStreamsTests.STREAMING_TOPIC1,
                 KafkaStreamsTests.STREAMING_TOPIC2 })
public class KafkaStreamsTests {

    @Autowired
    private EmbeddedKafkaBroker embeddedKafka;

    @Test
    public void someTest() {
        Map<String, Object> consumerProps = KafkaTestUtils.consumerProps(
"testGroup", "true", this.embeddedKafka);
        consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        ConsumerFactory<Integer, String> cf = new DefaultKafkaConsumerFactory<>
(consumerProps);
        Consumer<Integer, String> consumer = cf.createConsumer();
        this.embeddedKafka.consumeFromAnEmbeddedTopic(consumer, KafkaStreamsTests
.STREAMING_TOPIC2);
        ConsumerRecords<Integer, String> replies = KafkaTestUtils.getRecords
(consumer);
        assertThat(replies.count()).isGreaterThanOrEqualTo(1);
    }

    @Configuration
    @EnableKafkaStreams
    public static class KafkaStreamsConfiguration {

        @Value("${" + EmbeddedKafkaBroker.SPRING_EMBEDDED_KAFKA_BROKERS + "}")
        private String brokerAddresses;

        @Bean(name = KafkaStreamsDefaultConfiguration
.DEFAULT_STREAMS_CONFIG_BEAN_NAME)
        public KafkaStreamsConfiguration kStreamsConfigs() {
            Map<String, Object> props = new HashMap<>();
            props.put(StreamsConfig.APPLICATION_ID_CONFIG, "testStreams");
            props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, this.
brokerAddresses);
            return new KafkaStreamsConfiguration(props);
        }

    }

}
```

Starting with version 2.2.4, you can also use the `@EmbeddedKafka` annotation to specify the Kafka

ports property.

The following example sets the `topics`, `brokerProperties`, and `brokerPropertiesLocation` attributes of `@EmbeddedKafka` support property placeholder resolutions:

```
@TestPropertySource(locations = "classpath:/test.properties")
@EmbeddedKafka(topics = { "any-topic", "${kafka.topics.another-topic}" },
        brokerProperties = { "log.dir=${kafka.broker.logs-dir}",
                             "listeners=PLAINTEXT://localhost:${kafka.broker.port}
",
                             "auto.create.topics.enable=${kafka.broker.topics-
enable:true}" },
        brokerPropertiesLocation = "classpath:/broker.properties")
```

In the preceding example, the property placeholders `${kafka.topics.another-topic}`, `${kafka.broker.logs-dir}`, and `${kafka.broker.port}` are resolved from the Spring `Environment`. In addition, the broker properties are loaded from the `broker.properties` classpath resource specified by the `brokerPropertiesLocation`. Property placeholders are resolved for the `brokerPropertiesLocation` URL and for any property placeholders found in the resource. Properties defined by `brokerProperties` override properties found in `brokerPropertiesLocation`.

You can use the `@EmbeddedKafka` annotation with JUnit 4 or JUnit 5.

4.3.6. @EmbeddedKafka Annotation with JUnit5

Starting with version 2.3, there are two ways to use the `@EmbeddedKafka` annotation with JUnit5. When used with the `@SpringJunitConfig` annotation, the embedded broker is added to the test application context. You can auto wire the broker into your test, at the class or method level, to get the broker address list.

When **not** using the spring test context, the `EmbdeddedKafkaCondition` creates a broker; the condition includes a parameter resolver so you can access the broker in your test method...

```
@EmbeddedKafka
public class EmbeddedKafkaConditionTests {

    @Test
    public void test(EmbeddedKafkaBroker broker) {
        String brokerList = broker.getBrokersAsString();
        ...
    }

}
```

A stand-alone (not Spring test context) broker will be created if the class annotated with

`@EmbeddedBroker` is not also annotated (or meta annotated) with `ExtendedWith(SpringExtension.class)`. `@SpringJunitConfig` and `@SpringBootTest` are so meta annotated and the context-based broker will be used when either of those annotations are also present.

When there is a Spring test application context available, the topics and broker properties can contain property placeholders, which will be resolved as long as the property is defined somewhere. If there is no Spring context available, these placeholders won't be resolved.

4.3.7. Embedded Broker in `@SpringBootTest` Annotations

Spring Initializr now automatically adds the `spring-kafka-test` dependency in test scope to the project configuration.

If your application uses the Kafka binder in `spring-cloud-stream` and if you want to use an embedded broker for tests, you must remove the `spring-cloud-stream-test-support` dependency, because it replaces the real binder with a test binder for test cases. If you wish some tests to use the test binder and some to use the embedded broker, tests that use the real binder need to disable the test binder by excluding the binder auto configuration in the test class. The following example shows how to do so:

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.autoconfigure.exclude="
    +
"org.springframework.cloud.stream.test.binder.TestSupportBinderAuto
Configuration")
public class MyApplicationTests {
    ...
}
```

There are several ways to use an embedded broker in a Spring Boot application test.

They include:

- JUnit4 Class Rule
- `@EmbeddedKafka` Annotation or `EmbeddedKafkaBroker` Bean

JUnit4 Class Rule

The following example shows how to use a JUnit4 class rule to create an embedded broker:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyApplicationTests {

    @ClassRule
    public static EmbeddedKafkaRule broker = new EmbeddedKafkaRule(1,
        false, "someTopic")
            .brokerListProperty("spring.kafka.bootstrap-servers");
    }

    @Autowired
    private KafkaTemplate<String, String> template;

    @Test
    public void test() {
        ...
    }

}
```

Notice that, since this is a Spring Boot application, we override the broker list property to set Boot's property.

`@EmbeddedKafka` Annotation or `EmbeddedKafkaBroker` Bean

The following example shows how to use an `@EmbeddedKafka` Annotation to create an embedded broker:

```
@RunWith(SpringRunner.class)
@EmbeddedKafka(topics = "someTopic",
        bootstrapServersProperty = "spring.kafka.bootstrap-servers")
public class MyApplicationTests {

    @Autowired
    private KafkaTemplate<String, String> template;

    @Test
    public void test() {
        ...
    }

}
```

4.3.8. Hamcrest Matchers

The `o.s.kafka.test.hamcrest.KafkaMatchers` provides the following matchers:

```
/**
 * @param key the key
 * @param <K> the type.
 * @return a Matcher that matches the key in a consumer record.
 */
public static <K> Matcher<ConsumerRecord<K, ?>> hasKey(K key) { ... }

/**
 * @param value the value.
 * @param <V> the type.
 * @return a Matcher that matches the value in a consumer record.
 */
public static <V> Matcher<ConsumerRecord<?, V>> hasValue(V value) { ... }

/**
 * @param partition the partition.
 * @return a Matcher that matches the partition in a consumer record.
 */
public static Matcher<ConsumerRecord<?, ?>> hasPartition(int partition) { ... }

/**
 * Matcher testing the timestamp of a {@link ConsumerRecord} assuming the topic
has been set with
 * {@link org.apache.kafka.common.record.TimestampType#CREATE_TIME CreateTime}.
 *
 * @param ts timestamp of the consumer record.
 * @return a Matcher that matches the timestamp in a consumer record.
 */
public static Matcher<ConsumerRecord<?, ?>> hasTimestamp(long ts) {
  return hasTimestamp(TimestampType.CREATE_TIME, ts);
}

/**
 * Matcher testing the timestamp of a {@link ConsumerRecord}
 * @param type timestamp type of the record
 * @param ts timestamp of the consumer record.
 * @return a Matcher that matches the timestamp in a consumer record.
 */
public static Matcher<ConsumerRecord<?, ?>> hasTimestamp(TimestampType type, long
ts) {
  return new ConsumerRecordTimestampMatcher(type, ts);
}
```

4.3.9. AssertJ Conditions

You can use the following AssertJ conditions:

```
/**
 * @param key the key
 * @param <K> the type.
 * @return a Condition that matches the key in a consumer record.
 */
public static <K> Condition<ConsumerRecord<K, ?>> key(K key) { ... }

/**
 * @param value the value.
 * @param <V> the type.
 * @return a Condition that matches the value in a consumer record.
 */
public static <V> Condition<ConsumerRecord<?, V>> value(V value) { ... }

/**
 * @param key the key.
 * @param value the value.
 * @param <K> the key type.
 * @param <V> the value type.
 * @return a Condition that matches the key in a consumer record.
 * @since 2.2.12
 */
public static <K, V> Condition<ConsumerRecord<K, V>> keyValue(K key, V value) { .
.. }

/**
 * @param partition the partition.
 * @return a Condition that matches the partition in a consumer record.
 */
public static Condition<ConsumerRecord<?, ?>> partition(int partition) { ... }

/**
 * @param value the timestamp.
 * @return a Condition that matches the timestamp value in a consumer record.
 */
public static Condition<ConsumerRecord<?, ?>> timestamp(long value) {
  return new ConsumerRecordTimestampCondition(TimestampType.CREATE_TIME, value);
}

/**
 * @param type the type of timestamp
 * @param value the timestamp.
 * @return a Condition that matches the timestamp value in a consumer record.
 */
public static Condition<ConsumerRecord<?, ?>> timestamp(TimestampType type, long
value) {
  return new ConsumerRecordTimestampCondition(type, value);
}
```

4.3.10. Example

The following example brings together most of the topics covered in this chapter:

```
public class KafkaTemplateTests {

    private static final String TEMPLATE_TOPIC = "templateTopic";

    @ClassRule
    public static EmbeddedKafkaRule embeddedKafka = new EmbeddedKafkaRule(1, true,
TEMPLATE_TOPIC);

    @Test
    public void testTemplate() throws Exception {
        Map<String, Object> consumerProps = KafkaTestUtils.consumerProps("testT",
"false",
            embeddedKafka.getEmbeddedKafka());
        DefaultKafkaConsumerFactory<Integer, String> cf =
                            new DefaultKafkaConsumerFactory<Integer, String>
(consumerProps);
        ContainerProperties containerProperties = new ContainerProperties
(TEMPLATE_TOPIC);
        KafkaMessageListenerContainer<Integer, String> container =
                            new KafkaMessageListenerContainer<>(cf,
containerProperties);
        final BlockingQueue<ConsumerRecord<Integer, String>> records = new
LinkedBlockingQueue<>();
        container.setupMessageListener(new MessageListener<Integer, String>() {

            @Override
            public void onMessage(ConsumerRecord<Integer, String> record) {
                System.out.println(record);
                records.add(record);
            }

        });
        container.setBeanName("templateTests");
        container.start();
        ContainerTestUtils.waitForAssignment(container,
                            embeddedKafka.getEmbeddedKafka().
getPartitionsPerTopic());
        Map<String, Object> producerProps =
                            KafkaTestUtils.producerProps(embeddedKafka
.getEmbeddedKafka());
        ProducerFactory<Integer, String> pf =
                            new DefaultKafkaProducerFactory<Integer, String>
(producerProps);
        KafkaTemplate<Integer, String> template = new KafkaTemplate<>(pf);
        template.setDefaultTopic(TEMPLATE_TOPIC);
        template.sendDefault("foo");
        assertThat(records.poll(10, TimeUnit.SECONDS), hasValue("foo"));
        template.sendDefault(0, 2, "bar");
        ConsumerRecord<Integer, String> received = records.poll(10, TimeUnit
```

```
.SECONDS);
        assertThat(received, hasKey(2));
        assertThat(received, hasPartition(0));
        assertThat(received, hasValue("bar"));
        template.send(TEMPLATE_TOPIC, 0, 2, "baz");
        received = records.poll(10, TimeUnit.SECONDS);
        assertThat(received, hasKey(2));
        assertThat(received, hasPartition(0));
        assertThat(received, hasValue("baz"));
    }

}
```

The preceding example uses the Hamcrest matchers. With `AssertJ`, the final part looks like the following code:

```
assertThat(records.poll(10, TimeUnit.SECONDS)).has(value("foo"));
template.sendDefault(0, 2, "bar");
ConsumerRecord<Integer, String> received = records.poll(10, TimeUnit.SECONDS);
// using individual assertions
assertThat(received).has(key(2));
assertThat(received).has(value("bar"));
assertThat(received).has(partition(0));
template.send(TEMPLATE_TOPIC, 0, 2, "baz");
received = records.poll(10, TimeUnit.SECONDS);
// using allOf()
assertThat(received).has(allOf(keyValue(2, "baz"), partition(0)));
```

4.4. Non-Blocking Retries

This is an experimental feature and the usual rule of no breaking API changes does not apply to this feature until the experimental designation is removed. Users are encouraged to try out the feature and provide feedback via GitHub Issues or GitHub discussions. This is regarding the API only; the feature is considered to be complete, and robust.

Achieving non-blocking retry / dlt functionality with Kafka usually requires setting up extra topics and creating and configuring the corresponding listeners. Since 2.7 Spring for Apache Kafka offers support for that via the `@RetryableTopic` annotation and `RetryTopicConfiguration` class to simplify that bootstrapping.

4.4.1. How The Pattern Works

If message processing fails, the message is forwarded to a retry topic with a back off timestamp. The retry topic consumer then checks the timestamp and if it's not due it pauses the consumption

for that topic's partition. When it is due the partition consumption is resumed, and the message is consumed again. If the message processing fails again the message will be forwarded to the next retry topic, and the pattern is repeated until a successful processing occurs, or the attempts are exhausted, and the message is sent to the Dead Letter Topic (if configured).

To illustrate, if you have a "main-topic" topic, and want to setup non-blocking retry with an exponential backoff of 1000ms with a multiplier of 2 and 4 max attempts, it will create the main-topic-retry-1000, main-topic-retry-2000, main-topic-retry-4000 and main-topic-dlt topics and configure the respective consumers. The framework also takes care of creating the topics and setting up and configuring the listeners.

By using this strategy you lose Kafka's ordering guarantees for that topic.

You can set the `AckMode` mode you prefer, but `RECORD` is suggested.

At this time this functionality doesn't support class level `@KafkaListener` annotations

4.4.2. Back Off Delay Precision

Overview and Guarantees

All message processing and backing off is handled by the consumer thread, and, as such, delay precision is guaranteed on a best-effort basis. If one message's processing takes longer than the next message's back off period for that consumer, the next message's delay will be higher than expected. Also, for short delays (about 1s or less), the maintenance work the thread has to do, such as committing offsets, may delay the message processing execution. The precision can also be affected if the retry topic's consumer is handling more than one partition, because we rely on waking up the consumer from polling and having full pollTimeouts to make timing adjustments.

That being said, for consumers handling a single partition the message's processing should happen under 100ms after it's exact due time for most situations.

It is guaranteed that a message will never be processed before its due time.

Tuning the Delay Precision

The message's processing delay precision relies on two `ContainerProperties`: `ContainerProperties.pollTimeout` and `ContainerProperties.idlePartitionEventInterval`. Both properties will be automatically set in the retry topic and dlt's `ListenerContainerFactory` to one quarter of the smallest delay value for that topic, with a minimum value of 250ms and a maximum value of 5000ms. These values will only be set if the property has its default values - if you change either value yourself your change will not be overridden. This way you can tune the precision and performance for the retry topics if you need to.

You can have separate `ListenerContainerFactory` instances for the main and retry topics - this way you can have different settings to better suit your needs, such as having a higher polling timeout setting for the main topics and a lower one for the retry topics.

4.4.3. Configuration

Using the `@RetryableTopic` annotation

To configure the retry topic and dlt for a `@KafkaListener` annotated method, you just have to add the `@RetryableTopic` annotation to it and Spring for Apache Kafka will bootstrap all the necessary topics and consumers with the default configurations.

```
@RetryableTopic(kafkaTemplate = "myRetryableTopicKafkaTemplate")
@KafkaListener(topics = "my-annotated-topic", groupId = "myGroupId")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

You can specify a method in the same class to process the dlt messages by annotating it with the `@DltHandler` annotation. If no DltHandler method is provided a default consumer is created which only logs the consumption.

```
@DltHandler
public void processMessage(MyPojo message) {
// ... message processing, persistence, etc
}
```

If you don't specify a kafkaTemplate name a bean with name `retryTopicDefaultKafkaTemplate` will be looked up. If no bean is found an exception is thrown.

Using `RetryTopicConfiguration` beans

You can also configure the non-blocking retry support by creating `RetryTopicConfiguration` beans in a `@Configuration` annotated class.

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, Object>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .create(template);
}
```

This will create retry topics and a dlt, as well as the corresponding consumers, for all topics in methods annotated with '@KafkaListener' using the default configurations. The `KafkaTemplate` instance is required for message forwarding.

To achieve more fine-grained control over how to handle non-blocking retrials for each topic, more than one `RetryTopicConfiguration` bean can be provided.

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .fixedBackoff(3000)
            .maxAttempts(5)
            .includeTopics("my-topic", "my-other-topic")
            .create(template);
}

@Bean
public RetryTopicConfiguration myOtherRetryTopic(KafkaTemplate<String,
MyOtherPojo> template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .exponentialBackoff(1000, 2, 5000)
            .maxAttempts(4)
            .excludeTopics("my-topic", "my-other-topic")
            .retryOn(MyException.class)
            .create(template);
}
```

The retry topics' and dlt's consumers will be assigned to a consumer group with a group id that is the combination of the one with you provide in the `groupId` parameter of the `@KafkaListener` annotation with the topic's suffix. If you don't provide any they'll all belong to the same group, and rebalance on a retry topic will cause an unnecessary rebalance on the main topic.

4.4.4. Features

Most of the features are available both for the `@RetryableTopic` annotation and the `RetryTopicConfiguration` beans.

BackOff Configuration

The BackOff configuration relies on the `BackOffPolicy` interface from the `Spring Retry` project.

It includes:

- Fixed Back Off
- Exponential Back Off
- Random Exponential Back Off
- Uniform Random Back Off
- No Back Off
- Custom Back Off

```
@RetryableTopic(attempts = 5,
    backoff = @Backoff(delay = 1000, multiplier = 2, maxDelay = 5000))
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .fixedBackoff(3000)
            .maxAttempts(4)
            .build();
}
```

You can also provide a custom implementation of Spring Retry's `SleepingBackOffPolicy`:

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .customBackOff(new MyCustomBackOffPolicy())
            .maxAttempts(5)
            .build();
}
```

The default backoff policy is FixedBackOffPolicy with a maximum of 3 attempts and 1000ms intervals.

The first attempt counts against the maxAttempts, so if you provide a maxAttempts value of 4 there'll be the original attempt plus 3 retries.

Single Topic Fixed Delay Retries

If you're using fixed delay policies such as `FixedBackOffPolicy` or `NoBackOffPolicy` you can use a single topic to accomplish the non-blocking retries. This topic will be suffixed with the provided or default suffix, and will not have either the index or the delay values appended.

```
@RetryableTopic(backoff = @Backoff(2000), fixedDelayTopicStrategy =
FixedDelayStrategy.SINGLE_TOPIC)
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .fixedBackoff(3000)
            .maxAttempts(5)
            .useSingleTopicForFixedDelays()
            .build();
}
```

The default behavior is creating separate retry topics for each attempt, appended with their index value: retry-0, retry-1, ...

Global timeout

You can set the global timeout for the retrying process. If that time is reached, the next time the consumer throws an exception the message goes straight to the DLT, or just ends the processing if no DLT is available.

```
@RetryableTopic(backoff = @Backoff(2000), timeout = 5000)
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .fixedBackoff(2000)
            .timeoutAfter(5000)
            .build();
}
```

The default is having no timeout set, which can also be achieved by providing -1 as the timout value.

Exception Classifier

You can specify which exceptions you want to retry on and which not to. You can also set it to traverse the causes to lookup nested exceptions.

```
@RetryableTopic(include = {MyRetryException.class, MyOtherRetryException.class},
traversingCauses = true)
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        throw new RuntimeException(new MyRetryException()); // Will retry
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, MyOtherPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .notRetryOn(MyDontRetryException.class)
            .create(template);
}
```

The default behavior is retrying on all exceptions and not traversing causes.

Include and Exclude Topics

You can decide which topics will and will not be handled by a `RetryTopicConfiguration` bean via the .includeTopic(String topic), .includeTopics(Collection<String> topics) .excludeTopic(String topic) and .excludeTopics(Collection<String> topics) methods.

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<Integer, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .includeTopics(List.of("my-included-topic", "my-other-included-topic"
))
            .create(template);
}

@Bean
public RetryTopicConfiguration myOtherRetryTopic(KafkaTemplate<Integer, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .excludeTopic("my-excluded-topic")
            .create(template);
}
```

The default behavior is to include all topics.

Topics AutoCreation

Unless otherwise specified the framework will auto create the required topics using `NewTopic` beans that are consumed by the `KafkaAdmin` bean. You can specify the number of partitions and the replication factor with which the topics will be created, and you can turn this feature off.

Note that if you're not using Spring Boot you'll have to provide a KafkaAdmin bean in order to use this feature.

```
@RetryableTopic(numPartitions = 2, replicationFactor = 3)
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}

@RetryableTopic(autoCreateTopics = false)
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<Integer, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .autoCreateTopicsWith(2, 3)
            .create(template);
}

@Bean
public RetryTopicConfiguration myOtherRetryTopic(KafkaTemplate<Integer, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .doNotAutoCreateRetryTopics()
            .create(template);
}
```

By default the topics are autocreated with one partition and a replication factor of one.

4.4.5. Topic Naming

Retry topics and DLT are named by suffixing the main topic with a provided or default value, appended by either the delay or index for that topic.

Examples:

"my-topic" → "my-topic-retry-0", "my-topic-retry-1", ..., "my-topic-dlt"

"my-other-topic" → "my-topic-myRetrySuffix-1000", "my-topic-myRetrySuffix-2000", ..., "my-topic-

myDltSuffix".

Retry Topics and Dlt Suffixes

You can specify the suffixes that will be used by the retry and dlt topics.

```
@RetryableTopic(retryTopicSuffix = "-my-retry-suffix", dltTopicSuffix = "-my-dlt
-suffix")
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, MyOtherPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .retryTopicSuffix("-my-retry-suffix")
            .dltTopicSuffix("-my-dlt-suffix")
            .create(template);
}
```

The default suffixes are "-retry" and "-dlt", for retry topics and dlt respectively.

Appending the Topic's Index or Delay

You can either append the topic's index or delay values after the suffix.

```
@RetryableTopic(topicSuffixingStrategy = TopicSuffixingStrategy
.SUFFIX_WITH_INDEX_VALUE)
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<String, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .suffixTopicsWithIndexValues()
            .create(template);
    }
```

The default behavior is to suffix with the delay values, except for fixed delay configurations with multiple topics, in which case the topics are suffixed with the topic's index.

Custom naming strategies

More complex naming strategies can be accomplished by registering a bean that implements `RetryTopicNamesProviderFactory`. The default implementation is `SuffixingRetryTopicNamesProviderFactory` and a different implementation can be registered in the following way:

```
@Bean
public RetryTopicNamesProviderFactory myRetryNamingProviderFactory() {
    return new CustomRetryTopicNamesProviderFactory();
}
```

As an example the following implementation, in addition to the standard suffix, adds a prefix to retry/dl topics names:

```
public class CustomRetryTopicNamesProviderFactory implements
RetryTopicNamesProviderFactory {

    @Override
    public RetryTopicNamesProvider createRetryTopicNamesProvider(
                DestinationTopic.Properties properties) {

        if(properties.isMainEndpoint()) {
            return new SuffixingRetryTopicNamesProvider(properties);
        }
        else {
            return new SuffixingRetryTopicNamesProvider(properties) {

                @Override
                public String getTopicName(String topic) {
                    return "my-prefix-" + super.getTopicName(topic);
                }

            };
        }
    }

}
```

4.4.6. Dlt Strategies

The framework provides a few strategies for working with DLTs. You can provide a method for DLT processing, use the default logging method, or have no DLT at all. Also you can choose what happens if DLT processing fails.

Dlt Processing Method

You can specify the method used to process the Dlt for the topic, as well as the behavior if that processing fails.

To do that you can use the `@DltHandler` annotation in a method of the class with the `@RetryableTopic` annotation(s). Note that the same method will be used for all the `@RetryableTopic` annotated methods within that class.

```
@RetryableTopic
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}

@DltHandler
public void processMessage(MyPojo message) {
// ... message processing, persistence, etc
}
```

The DLT handler method can also be provided through the RetryTopicConfigurationBuilder.dltHandlerMethod(Class, String) method, passing as arguments the class and method name that should process the DLT's messages. If a bean instance of the provided class is found in the application context that bean is used for Dlt processing, otherwise an instance is created with full dependency injection support.

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<Integer, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .dltProcessor(MyCustomDltProcessor.class, "processDltMessage")
            .create(template);
}

@Component
public class MyCustomDltProcessor {

    private final MyDependency myDependency;

    public MyCustomDltProcessor(MyDependency myDependency) {
        this.myDependency = myDependency;
    }

    public void processDltMessage(MyPojo message) {
       // ... message processing, persistence, etc
    }
}
```

If no DLT handler is provided, the default RetryTopicConfigurer.LoggingDltListenerHandlerMethod is used.

Dlt Failure Behavior

Should the Dlt processing fail, there are two possible behaviors available: `ALWAYS_RETRY_ON_ERROR` and `FAIL_ON_ERROR`.

In the former the message is forwarded back to the dlt topic so it doesn't block other dlt messages' processing. In the latter the consumer ends the execution without forwarding the message.

```
@RetryableTopic(dltProcessingFailureStrategy =
            DltStrategy.FAIL_ON_ERROR)
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<Integer, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .dltProcessor(MyCustomDltProcessor.class, "processDltMessage")
            .doNotRetryOnDltFailure()
            .create(template);
}
```

The default behavior is to `ALWAYS_RETRY_ON_ERROR`.

Configuring No Dlt

The framework also provides the possibility of not configuring a DLT for the topic. In this case after retrials are exhausted the processing simply ends.

```
@RetryableTopic(dltProcessingFailureStrategy =
            DltStrategy.NO_DLT)
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<Integer, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .doNotConfigureDlt()
            .create(template);
}
```

4.4.7. Specifying a ListenerContainerFactory

By default the RetryTopic configuration will use the provided factory from the `@KafkaListener` annotation, but you can specify a different one to be used to create the retry topic and dlt listener containers.

The provided factory will be configured for the retry topic functionality, so you should not use the same factory for both retrying and non-retrying topics. You can however share the same factory between many retry topic configurations.

For the `@RetryableTopic` annotation you can provide the factory's bean name, and using the `RetryTopicConfiguration` bean you can either provide the bean name or the instance itself.

```
@RetryableTopic(listenerContainerFactory = "my-retry-topic-factory")
@KafkaListener(topics = "my-annotated-topic")
public void processMessage(MyPojo message) {
        // ... message processing
}
```

```
@Bean
public RetryTopicConfiguration myRetryTopic(KafkaTemplate<Integer, MyPojo>
template,
        ConcurrentKafkaListenerContainerFactory<Integer, MyPojo> factory) {

    return RetryTopicConfigurationBuilder
            .newInstance()
            .listenerFactory(factory)
            .create(template);
}

@Bean
public RetryTopicConfiguration myOtherRetryTopic(KafkaTemplate<Integer, MyPojo>
template) {
    return RetryTopicConfigurationBuilder
            .newInstance()
            .listenerFactory("my-retry-topic-factory")
            .create(template);
}
```

Chapter 5. Tips, Tricks and Examples

5.1. Manually Assigning All Partitions

Let's say you want to always read all records from all partitions (such as when using a compacted topic to load a distributed cache), it can be useful to manually assign the partitions and not use Kafka's group management. Doing so can be unwieldy when there are many partitions, because you have to list the partitions. It's also an issue if the number of partitions changes over time, because you would have to recompile your application each time the partition count changes.

The following is an example of how to use the power of a SpEL expression to create the partition list dynamically when the application starts:

```
@KafkaListener(topicPartitions = @TopicPartition(topic = "compacted",
            partitions = "#{@finder.partitions('compacted')}"),
            partitionOffsets = @PartitionOffset(partition = "*", initialOffset =
"0")))
public void listen(@Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) String key, String
payload) {
    ...
}

@Bean
public PartitionFinder finder(ConsumerFactory<String, String> consumerFactory) {
    return new PartitionFinder(consumerFactory);
}

public static class PartitionFinder {

    private final ConsumerFactory<String, String> consumerFactory;

    public PartitionFinder(ConsumerFactory<String, String> consumerFactory) {
        this.consumerFactory = consumerFactory;
    }

    public String[] partitions(String topic) {
        try (Consumer<String, String> consumer = consumerFactory.createConsumer())
{
            return consumer.partitionsFor(topic).stream()
                .map(pi -> "" + pi.partition())
                .toArray(String[]::new);
        }
    }

}
```

Using this in conjunction with `ConsumerConfig.AUTO_OFFSET_RESET_CONFIG=earliest` will load all records each time the application is started. You should also set the container's `AckMode` to `MANUAL` to prevent the container from committing offsets for a `null` consumer group. Howewever, starting with version 2.5.5, as shown above, you can apply an initial offset to all partitions; see Explicit Partition Assignment for more information.

5.2. Examples of Kafka Transactions with Other Transaction Managers

The following Spring Boot application is an example of chaining database and Kafka transactions. The listener container starts the Kafka transaction and the `@Transactional` annotation starts the DB transaction. The DB transaction is committed first; if the Kafka transaction fails to commit, the record will be redelivered so the DB update should be idempotent.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public ApplicationRunner runner(KafkaTemplate<String, String> template) {
        return args -> template.executeInTransaction(t -> t.send("topic1", "test"
));
    }

    @Bean
    public DataSourceTransactionManager dstm(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Component
    public static class Listener {

        private final JdbcTemplate jdbcTemplate;

        private final KafkaTemplate<String, String> kafkaTemplate;

        public Listener(JdbcTemplate jdbcTemplate, KafkaTemplate<String, String>
kafkaTemplate) {
            this.jdbcTemplate = jdbcTemplate;
            this.kafkaTemplate = kafkaTemplate;
        }

        @KafkaListener(id = "group1", topics = "topic1")
        @Transactional("dstm")
        public void listen1(String in) {
            this.kafkaTemplate.send("topic2", in.toUpperCase());
            this.jdbcTemplate.execute("insert into mytable (data) values ('" + in
+ "')");
        }

        @KafkaListener(id = "group2", topics = "topic2")
        public void listen2(String in) {
            System.out.println(in);
        }

    }

    @Bean
    public NewTopic topic1() {
        return TopicBuilder.name("topic1").build();
```

```
    }

    @Bean
    public NewTopic topic2() {
        return TopicBuilder.name("topic2").build();
    }

}
```

```
spring.datasource.url=jdbc:mysql://localhost/integration?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.enable-auto-commit=false
spring.kafka.consumer.properties.isolation.level=read_committed

spring.kafka.producer.transaction-id-prefix=tx-

#logging.level.org.springframework.transaction=trace
#logging.level.org.springframework.kafka.transaction=debug
#logging.level.org.springframework.jdbc=debug
```

```
create table mytable (data varchar(20));
```

For producer-only transactions, transaction synchronization works:

```
@Transactional("dstm")
public void someMethod(String in) {
    this.kafkaTemplate.send("topic2", in.toUpperCase());
    this.jdbcTemplate.execute("insert into mytable (data) values ('" + in + "')");
}
```

The `KafkaTemplate` will synchronize its transaction with the DB transaction and the commit/rollback occurs after the database.

If you wish to commit the Kafka transaction first, and only commit the DB transaction if the Kafka transaction is successful, use nested `@Transactional` methods:

```
@Transactional("dstm")
public void someMethod(String in) {
    this.jdbcTemplate.execute("insert into mytable (data) values ('" + in + "')");
    sendToKafka(in);
}

@Transactional("kafkaTransactionManager")
public void sendToKafka(String in) {
    this.kafkaTemplate.send("topic2", in.toUpperCase());
}
```

Chapter 6. Other Resources

In addition to this reference documentation, we recommend a number of other resources that may help you learn about Spring and Apache Kafka.

- Apache Kafka Project Home Page
- Spring for Apache Kafka Home Page
- Spring for Apache Kafka GitHub Repository
- Spring Integration Kafka Extension GitHub Repository

Appendix A: Override Spring Boot Dependencies

When using Spring for Apache Kafka in a Spring Boot application, the Apache Kafka dependency versions are determined by Spring Boot's dependency management. If you wish to use a different version of `kafka-clients` or `kafka-streams`, such as 2.8.0, you need to override all of the associated dependencies. This is especially true when using the embedded Kafka broker in `spring-kafka-test`.

Backwards compatibility is not supported for all Boot versions; Spring for Apache Kafka 2.7 has been tested with Spring Boot 2.4 and 2.5.

Maven

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>2.7.3</version>
</dependency>

<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka-test</artifactId>
    <version>2.7.3</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>{kafka-version}</version>
</dependency>

<!-- optional - only needed when using kafka-streams -->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>{kafka-version}</version>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>{kafka-version}</version>
    <classifier>test</classifier>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.13</artifactId>
    <version>{kafka-version}</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.13</artifactId>
    <version>{kafka-version}</version>
    <classifier>test</classifier>
    <scope>test</scope>
</dependency>
```

Gradle

```
dependencies {

    implementation 'org.springframework.kafka:spring-kafka:2.7.3'

    implementation "org.apache.kafka:kafka-clients:$kafkaVersion"
    implementation "org.apache.kafka:kafka-streams:$kafkaVersion" // optional -
only needed when using kafka-streams
    testImplementation 'org.springframework.kafka:spring-kafka-test:2.7.3'
    testImplementation "org.apache.kafka:kafka-clients:$kafkaVersion:test"
    testImplementation "org.apache.kafka:kafka_2.13:$kafkaVersion"
    testImplementation "org.apache.kafka:kafka_2.13:$kafkaVersion:test"

}
```

The test scope dependencies are only needed if you are using the embedded Kafka broker in tests.

Appendix B: Change History

B.1. Changes between 2.5 and 2.6

B.1.1. Kafka Client Version

This version requires the 2.6.0 `kafka-clients`.

B.1.2. Listener Container Changes

The default `EOSMode` is now `BETA`. See Exactly Once Semantics for more information.

Various error handlers (that extend `FailedRecordProcessor`) and the `DefaultAfterRollbackProcessor` now reset the `BackOff` if recovery fails. In addition, you can now select the `BackOff` to use based on the failed record and/or exception. See Seek To Current Container Error Handlers, Recovering Batch Error Handler, Publishing Dead-letter Records and After-rollback Processor for more information.

You can now configure an `adviceChain` in the container properties. See Listener Container Properties for more information.

When the container is configured to publish `ListenerContainerIdleEvent` s, it now publishes a `ListenerContainerNoLongerIdleEvent` when a record is received after publishing an idle event. See Application Events and Detecting Idle and Non-Responsive Consumers for more information.

B.1.3. @KafkaListener Changes

When using manual partition assignment, you can now specify a wildcard for determining which partitions should be reset to the initial offset. In addition, if the listener implements `ConsumerSeekAware`, `onPartitionsAssigned()` is called after the manual assignment. (Also added in version 2.5.5). See Explicit Partition Assignment for more information.

Convenience methods have been added to `AbstractConsumerSeekAware` to make seeking easier. See Seeking to a Specific Offset for more information.

B.1.4. ErrorHandler Changes

Subclasses of `FailedRecordProcessor` (e.g. `SeekToCurrentErrorHandler`, `DefaultAfterRollbackProcessor`, `RecoveringBatchErrorHandler`) can now be configured to reset the retry state if the exception is a different type to that which occurred previously with this record. See Seek To Current Container Error Handlers, After-rollback Processor, Recovering Batch Error Handler for more information.

B.1.5. Producer Factory Changes

You can now set a maximum age for producers after which they will be closed and recreated. See Transactions for more information.

You can now update the configuration map after the `DefaultKafkaProducerFactory` has been created. This might be useful, for example, if you have to update SSL key/trust store locations after a

credentials change. See Using `DefaultKafkaProducerFactory` for more information.

B.2. Changes between 2.4 and 2.5

This section covers the changes made from version 2.4 to version 2.5. For changes in earlier version, see Change History.

B.2.1. Consumer/Producer Factory Changes

The default consumer and producer factories can now invoke a callback whenever a consumer or producer is created or closed. Implementations for native Micrometer metrics are provided. See Factory Listeners for more information.

You can now change bootstrap server properties at runtime, enabling failover to another Kafka cluster. See Connecting to Kafka for more information.

B.2.2. `StreamsBuilderFactoryBean` Changes

The factory bean can now invoke a callback whenever a `KafkaStreams` created or destroyed. An Implementation for native Micrometer metrics is provided. See KafkaStreams Micrometer Support for more information.

B.2.3. Kafka Client Version

This version requires the 2.5.0 `kafka-clients`.

B.2.4. Class/Package Changes

`SeekUtils` has been moved from the `o.s.k.support` package to `o.s.k.listener`.

B.2.5. Delivery Attempts Header

There is now an option to to add a header which tracks delivery attempts when using certain error handlers and after rollback processors. See Delivery Attempts Header for more information.

B.2.6. @KafkaListener Changes

Default reply headers will now be populated automatically if needed when a `@KafkaListener` return type is `Message<?>`. See Reply Type Message<?> for more information.

The `KafkaHeaders.RECEIVED_MESSAGE_KEY` is no longer populated with a `null` value when the incoming record has a `null` key; the header is omitted altogether.

`@KafkaListener` methods can now specify a `ConsumerRecordMetadata` parameter instead of using discrete headers for metadata such as topic, partition, etc. See Consumer Record Metadata for more information.

B.2.7. Listener Container Changes

The `assignmentCommitOption` container property is now `LATEST_ONLY_NO_TX` by default. See Listener Container Properties for more information.

The `subBatchPerPartition` container property is now `true` by default when using transactions. See Transactions for more information.

A new `RecoveringBatchErrorHandler` is now provided. See Recovering Batch Error Handler for more information.

Static group membership is now supported. See Message Listener Containers for more information.

When incremental/cooperative rebalancing is configured, if offsets fail to commit with a non-fatal `RebalanceInProgressException`, the container will attempt to re-commit the offsets for the partitions that remain assigned to this instance after the rebalance is completed.

The default error handler is now the `SeekToCurrentErrorHandler` for record listeners and `RecoveringBatchErrorHandler` for batch listeners. See Container Error Handlers for more information.

You can now control the level at which exceptions intentionally thrown by standard error handlers are logged. See Container Error Handlers for more information.

The `getAssignmentsByClientId()` method has been added, making it easier to determine which consumers in a concurrent container are assigned which partition(s). See Listener Container Properties for more information.

You can now suppress logging entire `ConsumerRecord` s in error, debug logs etc. See `onlyLogRecordMetadata` in Listener Container Properties.

B.2.8. KafkaTemplate Changes

The `KafkaTemplate` can now maintain micrometer timers. See Monitoring for more information.

The `KafkaTemplate` can now be configured with `ProducerConfig` properties to override those in the producer factory. See Using `KafkaTemplate` for more information.

A `RoutingKafkaTemplate` has now been provided. See Using `RoutingKafkaTemplate` for more information.

You can now use `KafkaSendCallback` instead of `ListenerFutureCallback` to get a narrower exception, making it easier to extract the failed `ProducerRecord`. See Using `KafkaTemplate` for more information.

B.2.9. Kafka String Serializer/Deserializer

New `ToStringSerializer`/`StringDeserializer` s as well as an associated `SerDe` are now provided. See String serialization for more information.

B.2.10. JsonDeserializer

The `JsonDeserializer` now has more flexibility to determine the deserialization type. See Using Methods to Determine Types for more information.

B.2.11. Delegating Serializer/Deserializer

The `DelegatingSerializer` can now handle "standard" types, when the outbound record has no header. See Delegating Serializer and Deserializer for more information.

B.2.12. Testing Changes

The `KafkaTestUtils.consumerProps()` helper record now sets `ConsumerConfig.AUTO_OFFSET_RESET_CONFIG` to `earliest` by default. See JUnit for more information.

B.3. Changes between 2.3 and 2.4

B.3.1. Kafka Client Version

This version requires the 2.4.0 `kafka-clients` or higher and supports the new incremental rebalancing feature.

B.3.2. ConsumerAwareRebalanceListener

Like `ConsumerRebalanceListener`, this interface now has an additional method `onPartitionsLost`. Refer to the Apache Kafka documentation for more information.

Unlike the `ConsumerRebalanceListener`, The default implementation does **not** call `onPartitionsRevoked`. Instead, the listener container will call that method after it has called `onPartitionsLost`; you should not, therefore, do the same when implementing `ConsumerAwareRebalanceListener`.

See the IMPORTANT note at the end of Rebalancing Listeners for more information.

B.3.3. GenericErrorHandler

The `isAckAfterHandle()` default implementation now returns true by default.

B.3.4. KafkaTemplate

The `KafkaTemplate` now supports non-transactional publishing alongside transactional. See `KafkaTemplate` Transactional and non-Transactional Publishing for more information.

B.3.5. AggregatingReplyingKafkaTemplate

The `releaseStrategy` is now a `BiConsumer`. It is now called after a timeout (as well as when records arrive); the second parameter is `true` in the case of a call after a timeout.

See Aggregating Multiple Replies for more information.

B.3.6. Listener Container

The `ContainerProperties` provides an `authorizationExceptionRetryInterval` option to let the listener container to retry after any `AuthorizationException` is thrown by the `KafkaConsumer`. See its JavaDocs and Using `KafkaMessageListenerContainer` for more information.

B.3.7. @KafkaListener

The `@KafkaListener` annotation has a new property `splitIterables`; default true. When a replying listener returns an `Iterable` this property controls whether the return result is sent as a single record or a record for each element is sent. See Forwarding Listener Results using `@SendTo` for more information

Batch listeners can now be configured with a `BatchToRecordAdapter`; this allows, for example, the batch to be processed in a transaction while the listener gets one record at a time. With the default implementation, a `ConsumerRecordRecoverer` can be used to handle errors within the batch, without stopping the processing of the entire batch - this might be useful when using transactions. See Transactions with Batch Listeners for more information.

B.3.8. Kafka Streams

The `StreamsBuilderFactoryBean` accepts a new property `KafkaStreamsInfrastructureCustomizer`. This allows configuration of the builder and/or topology before the stream is created. See Spring Management for more information.

B.4. Changes Between 2.2 and 2.3

This section covers the changes made from version 2.2 to version 2.3.

B.4.1. Tips, Tricks and Examples

A new chapter Tips, Tricks and Examples has been added. Please submit GitHub issues and/or pull requests for additional entries in that chapter.

B.4.2. Kafka Client Version

This version requires the 2.3.0 `kafka-clients` or higher.

B.4.3. Class/Package Changes

`TopicPartitionInitialOffset` is deprecated in favor of `TopicPartitionOffset`.

B.4.4. Configuration Changes

Starting with version 2.3.4, the `missingTopicsFatal` container property is false by default. When this is true, the application fails to start if the broker is down; many users were affected by this change; given that Kafka is a high-availability platform, we did not anticipate that starting an application with no active brokers would be a common use case.

B.4.5. Producer and Consumer Factory Changes

The `DefaultKafkaProducerFactory` can now be configured to create a producer per thread. You can also provide `Supplier<Serializer>` instances in the constructor as an alternative to either configured classes (which require no-arg constructors), or constructing with `Serializer` instances, which are then shared between all Producers. See Using `DefaultKafkaProducerFactory` for more information.

The same option is available with `Supplier<Deserializer>` instances in `DefaultKafkaConsumerFactory`. See Using `KafkaMessageListenerContainer` for more information.

B.4.6. Listener Container Changes

Previously, error handlers received `ListenerExecutionFailedException` (with the actual listener exception as the `cause`) when the listener was invoked using a listener adapter (such as `@KafkaListener` s). Exceptions thrown by native `GenericMessageListener` s were passed to the error handler unchanged. Now a `ListenerExecutionFailedException` is always the argument (with the actual listener exception as the `cause`), which provides access to the container's `group.id` property.

Because the listener container has it's own mechanism for committing offsets, it prefers the Kafka `ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG` to be `false`. It now sets it to false automatically unless specifically set in the consumer factory or the container's consumer property overrides.

The `ackOnError` property is now `false` by default. See Seek To Current Container Error Handlers for more information.

It is now possible to obtain the consumer's `group.id` property in the listener method. See Obtaining the Consumer `group.id` for more information.

The container has a new property `recordInterceptor` allowing records to be inspected or modified before invoking the listener. A `CompositeRecordInterceptor` is also provided in case you need to invoke multiple interceptors. See Message Listener Containers for more information.

The `ConsumerSeekAware` has new methods allowing you to perform seeks relative to the beginning, end, or current position and to seek to the first offset greater than or equal to a time stamp. See Seeking to a Specific Offset for more information.

A convenience class `AbstractConsumerSeekAware` is now provided to simplify seeking. See Seeking to a Specific Offset for more information.

The `ContainerProperties` provides an `idleBetweenPolls` option to let the main loop in the listener container to sleep between `KafkaConsumer.poll()` calls. See its JavaDocs and Using `KafkaMessageListenerContainer` for more information.

When using `AckMode.MANUAL` (or `MANUAL_IMMEDIATE`) you can now cause a redelivery by calling `nack` on the `Acknowledgment`. See Committing Offsets for more information.

Listener performance can now be monitored using Micrometer `Timer` s. See Monitoring for more information.

The containers now publish additional consumer lifecycle events relating to startup. See

Application Events for more information.

Transactional batch listeners can now support zombie fencing. See Transactions for more information.

The listener container factory can now be configured with a `ContainerCustomizer` to further configure each container after it has been created and configured. See Container factory for more information.

B.4.7. ErrorHandler Changes

The `SeekToCurrentErrorHandler` now treats certain exceptions as fatal and disables retry for those, invoking the recoverer on first failure.

The `SeekToCurrentErrorHandler` and `SeekToCurrentBatchErrorHandler` can now be configured to apply a `BackOff` (thread sleep) between delivery attempts.

Starting with version 2.3.2, recovered records' offsets will be committed when the error handler returns after recovering a failed record.

See Seek To Current Container Error Handlers for more information.

The `DeadLetterPublishingRecoverer`, when used in conjunction with an `ErrorHandlingDeserializer`, now sets the payload of the message sent to the dead-letter topic, to the original value that could not be deserialized. Previously, it was `null` and user code needed to extract the `DeserializationException` from the message headers. See Publishing Dead-letter Records for more information.

B.4.8. TopicBuilder

A new class `TopicBuilder` is provided for more convenient creation of `NewTopic` `@Bean` s for automatic topic provisioning. See Configuring Topics for more information.

B.4.9. Kafka Streams Changes

You can now perform additional configuration of the `StreamsBuilderFactoryBean` created by `@EnableKafkaStreams`. See Streams Configuration for more information.

A `RecoveringDeserializationExceptionHandler` is now provided which allows records with deserialization errors to be recovered. It can be used in conjunction with a `DeadLetterPublishingRecoverer` to send these records to a dead-letter topic. See Recovery from Deserialization Exceptions for more information.

The `HeaderEnricher` transformer has been provided, using SpEL to generate the header values. See Header Enricher for more information.

The `MessagingTransformer` has been provided. This allows a Kafka streams topology to interact with a spring-messaging component, such as a Spring Integration flow. See `MessagingTransformer` and See [Calling a Spring Integration Flow from a `KStream`] for more information.

B.4.10. JSON Component Changes

Now all the JSON-aware components are configured by default with a Jackson `ObjectMapper` produced by the `JacksonUtils.enhancedObjectMapper()`. The `JsonDeserializer` now provides `TypeReference`-based constructors for better handling of target generic container types. Also a `JacksonMimeTypeModule` has been introduced for serialization of `org.springframework.util.MimeType` to plain string. See its JavaDocs and Serialization, Deserialization, and Message Conversion for more information.

A `ByteArrayJsonMessageConverter` has been provided as well as a new super class for all Json converters, `JsonMessageConverter`. Also, a `StringOrBytesSerializer` is now available; it can serialize `byte[]`, `Bytes` and `String` values in `ProducerRecord` s. See Spring Messaging Message Conversion for more information.

The `JsonSerializer`, `JsonDeserializer` and `JsonSerde` now have fluent APIs to make programmatic configuration simpler. See the javadocs, Serialization, Deserialization, and Message Conversion, and Streams JSON Serialization and Deserialization for more informaion.

B.4.11. ReplyingKafkaTemplate

When a reply times out, the future is completed exceptionally with a `KafkaReplyTimeoutException` instead of a `KafkaException`.

Also, an overloaded `sendAndReceive` method is now provided that allows specifying the reply timeout on a per message basis.

B.4.12. AggregatingReplyingKafkaTemplate

Extends the `ReplyingKafkaTemplate` by aggregating replies from multiple receivers. See Aggregating Multiple Replies for more information.

B.4.13. Transaction Changes

You can now override the producer factory's `transactionIdPrefix` on the `KafkaTemplate` and `KafkaTransactionManager`. See `transactionIdPrefix` for more information.

B.4.14. New Delegating Serializer/Deserializer

The framework now provides a delegating `Serializer` and `Deserializer`, utilizing a header to enable producing and consuming records with multiple key/value types. See Delegating Serializer and Deserializer for more information.

B.4.15. New Retrying Deserializer

The framework now provides a delegating `RetryingDeserializer`, to retry serialization when transient errors such as network problems might occur. See Retrying Deserializer for more information.

B.5. Changes Between 2.1 and 2.2

B.5.1. Kafka Client Version

This version requires the 2.0.0 `kafka-clients` or higher.

B.5.2. Class and Package Changes

The `ContainerProperties` class has been moved from `org.springframework.kafka.listener.config` to `org.springframework.kafka.listener`.

The `AckMode` enum has been moved from `AbstractMessageListenerContainer` to `ContainerProperties`.

The `setBatchErrorHandler()` and `setErrorHandler()` methods have been moved from `ContainerProperties` to both `AbstractMessageListenerContainer` and `AbstractKafkaListenerContainerFactory`.

B.5.3. After Rollback Processing

A new `AfterRollbackProcessor` strategy is provided. See After-rollback Processor for more information.

B.5.4. `ConcurrentKafkaListenerContainerFactory` Changes

You can now use the `ConcurrentKafkaListenerContainerFactory` to create and configure any `ConcurrentMessageListenerContainer`, not only those for `@KafkaListener` annotations. See Container factory for more information.

B.5.5. Listener Container Changes

A new container property (`missingTopicsFatal`) has been added. See Using `KafkaMessageListenerContainer` for more information.

A `ConsumerStoppedEvent` is now emitted when a consumer stops. See Thread Safety for more information.

Batch listeners can optionally receive the complete `ConsumerRecords<?, ?>` object instead of a `List<ConsumerRecord<?, ?>`. See Batch Listeners for more information.

The `DefaultAfterRollbackProcessor` and `SeekToCurrentErrorHandler` can now recover (skip) records that keep failing, and, by default, does so after 10 failures. They can be configured to publish failed records to a dead-letter topic.

Starting with version 2.2.4, the consumer's group ID can be used while selecting the dead letter topic name.

See After-rollback Processor, Seek To Current Container Error Handlers, and Publishing Dead-letter Records for more information.

The `ConsumerStoppingEvent` has been added. See Application Events for more information.

The `SeekToCurrentErrorHandler` can now be configured to commit the offset of a recovered record when the container is configured with `AckMode.MANUAL_IMMEDIATE` (since 2.2.4). See Seek To Current Container Error Handlers for more information.

B.5.6. @KafkaListener Changes

You can now override the `concurrency` and `autoStartup` properties of the listener container factory by setting properties on the annotation. You can now add configuration to determine which headers (if any) are copied to a reply message. See `@KafkaListener` Annotation for more information.

You can now use `@KafkaListener` as a meta-annotation on your own annotations. See `@KafkaListener` as a Meta Annotation for more information.

It is now easier to configure a `Validator` for `@Payload` validation. See `@KafkaListener` `@Payload` Validation for more information.

You can now specify kafka consumer properties directly on the annotation; these will override any properties with the same name defined in the consumer factory (since version 2.2.4). See Annotation Properties for more information.

B.5.7. Header Mapping Changes

Headers of type `MimeType` and `MediaType` are now mapped as simple strings in the `RecordHeader` value. Previously, they were mapped as JSON and only `MimeType` was decoded. `MediaType` could not be decoded. They are now simple strings for interoperability.

Also, the `DefaultKafkaHeaderMapper` has a new `addToStringClasses` method, allowing the specification of types that should be mapped by using `toString()` instead of JSON. See Message Headers for more information.

B.5.8. Embedded Kafka Changes

The `KafkaEmbedded` class and its `KafkaRule` interface have been deprecated in favor of the `EmbeddedKafkaBroker` and its JUnit 4 `EmbeddedKafkaRule` wrapper. The `@EmbeddedKafka` annotation now populates an `EmbeddedKafkaBroker` bean instead of the deprecated `KafkaEmbedded`. This change allows the use of `@EmbeddedKafka` in JUnit 5 tests. The `@EmbeddedKafka` annotation now has the attribute `ports` to specify the port that populates the `EmbeddedKafkaBroker`. See Testing Applications for more information.

B.5.9. JsonSerializer/Deserializer Enhancements

You can now provide type mapping information by using producer and consumer properties.

New constructors are available on the deserializer to allow overriding the type header information with the supplied target type.

The `JsonDeserializer` now removes any type information headers by default.

You can now configure the `JsonDeserializer` to ignore type information headers by using a Kafka property (since 2.2.3).

See Serialization, Deserialization, and Message Conversion for more information.

B.5.10. Kafka Streams Changes

The streams configuration bean must now be a `KafkaStreamsConfiguration` object instead of a `StreamsConfig` object.

The `StreamsBuilderFactoryBean` has been moved from package `…core` to `…config`.

The `KafkaStreamBrancher` has been introduced for better end-user experience when conditional branches are built on top of `KStream` instance.

See Apache Kafka Streams Support and Configuration for more information.

B.5.11. Transactional ID

When a transaction is started by the listener container, the `transactional.id` is now the `transactionIdPrefix` appended with `<group.id>.<topic>.<partition>`. This change allows proper fencing of zombies, as described here.

B.6. Changes Between 2.0 and 2.1

B.6.1. Kafka Client Version

This version requires the 1.0.0 `kafka-clients` or higher.

The 1.1.x client is supported natively in version 2.2.

B.6.2. JSON Improvements

The `StringJsonMessageConverter` and `JsonSerializer` now add type information in `Headers`, letting the converter and `JsonDeserializer` create specific types on reception, based on the message itself rather than a fixed configured type. See Serialization, Deserialization, and Message Conversion for more information.

B.6.3. Container Stopping Error Handlers

Container error handlers are now provided for both record and batch listeners that treat any exceptions thrown by the listener as fatal/ They stop the container. See Handling Exceptions for more information.

B.6.4. Pausing and Resuming Containers

The listener containers now have `pause()` and `resume()` methods (since version 2.1.3). See Pausing and Resuming Listener Containers for more information.

B.6.5. Stateful Retry

Starting with version 2.1.3, you can configure stateful retry. See Stateful Retry for more

information.

B.6.6. Client ID

Starting with version 2.1.1, you can now set the `client.id` prefix on `@KafkaListener`. Previously, to customize the client ID, you needed a separate consumer factory (and container factory) per listener. The prefix is suffixed with `-n` to provide unique client IDs when you use concurrency.

B.6.7. Logging Offset Commits

By default, logging of topic offset commits is performed with the `DEBUG` logging level. Starting with version 2.1.2, a new property in `ContainerProperties` called `commitLogLevel` lets you specify the log level for these messages. See Using `KafkaMessageListenerContainer` for more information.

B.6.8. Default @KafkaHandler

Starting with version 2.1.3, you can designate one of the `@KafkaHandler` annotations on a class-level `@KafkaListener` as the default. See `@KafkaListener` on a Class for more information.

B.6.9. ReplyingKafkaTemplate

Starting with version 2.1.3, a subclass of `KafkaTemplate` is provided to support request/reply semantics. See Using `ReplyingKafkaTemplate` for more information.

B.6.10. ChainedKafkaTransactionManager

Version 2.1.3 introduced the `ChainedKafkaTransactionManager`. (It is now deprecated).

B.6.11. Migration Guide from 2.0

See the 2.0 to 2.1 Migration guide.

B.7. Changes Between 1.3 and 2.0

B.7.1. Spring Framework and Java Versions

The Spring for Apache Kafka project now requires Spring Framework 5.0 and Java 8.

B.7.2. `@KafkaListener` Changes

You can now annotate `@KafkaListener` methods (and classes and `@KafkaHandler` methods) with `@SendTo`. If the method returns a result, it is forwarded to the specified topic. See Forwarding Listener Results using `@SendTo` for more information.

B.7.3. Message Listeners

Message listeners can now be aware of the `Consumer` object. See Message Listeners for more information.

B.7.4. Using `ConsumerAwareRebalanceListener`

Rebalance listeners can now access the `Consumer` object during rebalance notifications. See Rebalancing Listeners for more information.

B.8. Changes Between 1.2 and 1.3

B.8.1. Support for Transactions

The 0.11.0.0 client library added support for transactions. The `KafkaTransactionManager` and other support for transactions have been added. See Transactions for more information.

B.8.2. Support for Headers

The 0.11.0.0 client library added support for message headers. These can now be mapped to and from `spring-messaging MessageHeaders`. See Message Headers for more information.

B.8.3. Creating Topics

The 0.11.0.0 client library provides an `AdminClient`, which you can use to create topics. The `KafkaAdmin` uses this client to automatically add topics defined as `@Bean` instances.

B.8.4. Support for Kafka Timestamps

`KafkaTemplate` now supports an API to add records with timestamps. New `KafkaHeaders` have been introduced regarding `timestamp` support. Also, new `KafkaConditions.timestamp()` and `KafkaMatchers.hasTimestamp()` testing utilities have been added. See Using `KafkaTemplate`, `@KafkaListener` Annotation, and Testing Applications for more details.

B.8.5. `@KafkaListener` Changes

You can now configure a `KafkaListenerErrorHandler` to handle exceptions. See Handling Exceptions for more information.

By default, the `@KafkaListener id` property is now used as the `group.id` property, overriding the property configured in the consumer factory (if present). Further, you can explicitly configure the `groupId` on the annotation. Previously, you would have needed a separate container factory (and consumer factory) to use different `group.id` values for listeners. To restore the previous behavior of using the factory configured `group.id`, set the `idIsGroup` property on the annotation to `false`.

B.8.6. `@EmbeddedKafka` Annotation

For convenience, a test class-level `@EmbeddedKafka` annotation is provided, to register `KafkaEmbedded` as a bean. See Testing Applications for more information.

B.8.7. Kerberos Configuration

Support for configuring Kerberos is now provided. See JAAS and Kerberos for more information.

B.9. Changes Between 1.1 and 1.2

This version uses the 0.10.2.x client.

B.10. Changes Between 1.0 and 1.1

B.10.1. Kafka Client

This version uses the Apache Kafka 0.10.x.x client.

B.10.2. Batch Listeners

Listeners can be configured to receive the entire batch of messages returned by the `consumer.poll()` operation, rather than one at a time.

B.10.3. Null Payloads

Null payloads are used to “delete” keys when you use log compaction.

B.10.4. Initial Offset

When explicitly assigning partitions, you can now configure the initial offset relative to the current position for the consumer group, rather than absolute or relative to the current end.

B.10.5. Seek

You can now seek the position of each topic or partition. You can use this to set the initial position during initialization when group management is in use and Kafka assigns the partitions. You can also seek when an idle container is detected or at any arbitrary point in your application’s execution. See Seeking to a Specific Offset for more information.