Spring for Apache Kafka

Gary Russell, Artem Bilan, Biju Kunjummen, Jay Bryant

Version 2.3.0.RELEASE

Table of Contents

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 2.3 Since 2.2

This section covers the changes made from version 2.2 to version 2.3.

Also see What's new in Spring Integration for Apache Kafka (version 3.2).

2.1.1. 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.

2.1.2. Kafka Client Version

This version requires the 2.3.0 `kafka-clients` or higher.

2.1.3. Class/Package Changes

`TopicPartitionInitialOffset` is deprecated in favor of `TopicPartitionOffset`.

2.1.4. 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.

2.1.5. 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 Listener Performance for more information.

The containers now publish additional consumer lifecycle events relating to startup. See Listener Consumer Lifecycle Events for more information.

2.1.6. 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.

See Seek To Current Container Error Handlers for more information.

The `DeadLetterPublishingRecoverer`, when used in conjunction with an `ErrorHandlingDeserializer2`, 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.

2.1.7. 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.

2.1.8. 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 Calling a Spring Integration flow from a `KStream` for more information.

2.1.9. 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.

2.1.10. 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.

2.1.11. AggregatingReplyingKafkaTemplate

Extends the `ReplyingKafkaTemplate` by aggregating replies from multiple receivers. See Aggregating Multiple Replies for more information.

2.1.12. Transaction Changes

You can now override the producer factory’s `transactionIdPrefix` on the `KafkaTemplate` and `KafkaTransactionManager`. See `transactionIdPrefix` for more information.

2.1.13. 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.

2.1.14. 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.

2.1.15. New function for recovering from deserializing errors

`ErrorHandlingDeserializer2` now uses a POJO (`FailedDeserializationInfo`) for passing all the contextual information around a deserialization error. This enables the code to access to extra information that was missing in the old `BiFunction<byte[], Headers, T> failedDeserializationFunction`.

2.1.16. EmbeddedKafkaBroker Changes

You can now override the default broker list property name in the annotation. See `@EmbeddedKafka` Annotation or `EmbeddedKafkaBroker` Bean for more information.

2.1.17. ReplyingKafkaTemplate Changes

You can now customize the header names for correlation, reply topic and reply partition. See Using `ReplyingKafkaTemplate` for more information.

2.1.18. Header Mapper Changes

The `DefaultKafkaHeaderMapper` no longer encodes simple String-valued headers as JSON. See [header-mapping] 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 for the Impatient

This is the five-minute tour to get started with Spring Kafka.

Prerequisites: You must install and run Apache Kafka. Then you must grab the spring-kafka JAR and all of its dependencies. The easiest way to do that is to declare a dependency in your build tool. The following example shows how to do so with Maven:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>2.3.0.RELEASE</version>
</dependency>
```

The following example shows how to do so with Gradle:

```
compile 'org.springframework.kafka:spring-kafka:2.3.0.RELEASE'
```

3.1.1. Compatibility

This quick tour works with the following versions:

- Apache Kafka Clients 2.2.0
- Spring Framework 5.2.x
- Minimum Java version: 8

3.1.2. A Very, Very Quick Example

As the following example shows, you can use plain Java to send and receive a message:

```
@Test
public void testAutoCommit() throws Exception {
    logger.info("Start auto");
    ContainerProperties containerProps = new ContainerProperties("topic1",
"topic2");
    final CountDownLatch latch = new CountDownLatch(4);
    containerProps.setMessageListener(new MessageListener<Integer, String>() {

        @Override
        public void onMessage(ConsumerRecord<Integer, String> message) {
            logger.info("received: " + message);
            latch.countDown();
        }

    });
    KafkaMessageListenerContainer<Integer, String> container =
createContainer(containerProps);
    container.setBeanName("testAuto");
    container.start();
    Thread.sleep(1000); // wait a bit for the container to start
    KafkaTemplate<Integer, String> template = createTemplate();
    template.setDefaultTopic(topic1);
    template.sendDefault(0, "foo");
    template.sendDefault(2, "bar");
    template.sendDefault(0, "baz");
    template.sendDefault(2, "qux");
    template.flush();
    assertTrue(latch.await(60, TimeUnit.SECONDS));
    container.stop();
    logger.info("Stop auto");

}
```

```
private KafkaMessageListenerContainer<Integer, String> createContainer(
                        ContainerProperties containerProps) {
    Map<String, Object> props = consumerProps();
    DefaultKafkaConsumerFactory<Integer, String> cf =
                            new DefaultKafkaConsumerFactory<Integer,
String>(props);
    KafkaMessageListenerContainer<Integer, String> container =
                            new KafkaMessageListenerContainer<>(cf,
containerProps);
    return container;
}

private KafkaTemplate<Integer, String> createTemplate() {
    Map<String, Object> senderProps = senderProps();
    ProducerFactory<Integer, String> pf =
              new DefaultKafkaProducerFactory<Integer, String>(senderProps);
    KafkaTemplate<Integer, String> template = new KafkaTemplate<>(pf);
    return template;
}

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.ENABLE_AUTO_COMMIT_CONFIG, true);
    props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
    props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "15000");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
IntegerDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
    return props;
}

private Map<String, Object> senderProps() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ProducerConfig.RETRIES_CONFIG, 0);
    props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
    props.put(ProducerConfig.LINGER_MS_CONFIG, 1);
    props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
    return props;
}
```

3.1.3. With Java Configuration

You can do the same work as appears in the previous example with Spring configuration in Java. The following example shows how to do so:

```
@Autowired
private Listener listener;

@Autowired
private KafkaTemplate<Integer, String> template;

@Test
public void testSimple() throws Exception {
    template.send("annotated1", 0, "foo");
    template.flush();
    assertTrue(this.listener.latch1.await(10, TimeUnit.SECONDS));
}

@Configuration
@EnableKafka
public class Config {

    @Bean
    ConcurrentKafkaListenerContainerFactory<Integer, String>
                        kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
                                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        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(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
embeddedKafka.getBrokersAsString());
        ...
        return props;
    }

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

    @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,
embeddedKafka.getBrokersAsString());
        ...
        return props;
    }

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

}
```

```
public class Listener {

    private final CountDownLatch latch1 = new CountDownLatch(1);

    @KafkaListener(id = "foo", topics = "annotated1")
    public void listen1(String foo) {
        this.latch1.countDown();
    }

}
```

3.1.4. Even Quicker, with Spring Boot

Spring Boot can make things even simpler. The following Spring Boot application sends three messages to a topic, receives them, and stops:

```
@SpringBootApplication
public class Application implements CommandLineRunner {

    public static Logger logger = LoggerFactory.getLogger(Application.class);

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

    @Autowired
    private KafkaTemplate<String, String> template;

    private final CountDownLatch latch = new CountDownLatch(3);

    @Override
    public void run(String... args) throws Exception {
        this.template.send("myTopic", "foo1");
        this.template.send("myTopic", "foo2");
        this.template.send("myTopic", "foo3");
        latch.await(60, TimeUnit.SECONDS);
        logger.info("All received");
    }

    @KafkaListener(topics = "myTopic")
    public void listen(ConsumerRecord<?, ?> cr) throws Exception {
        logger.info(cr.toString());
        latch.countDown();
    }

}
```

Boot takes care of most of the configuration. When we use a local broker, the only properties we need are the following:

Example 1. application.properties

```
spring.kafka.consumer.group-id=foo
spring.kafka.consumer.auto-offset-reset=earliest
```

We need the first property because we are using group management to assign topic partitions to consumers, so we need a group. The second property ensures the new consumer group gets the messages we sent, because the container might start after the sends have completed.

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 for the Impatient.

4.1.1. 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:

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

When using Spring Boot, a `KafkaAdmin` bean is automatically registered so you only need the `NewTopic @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`.

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.getConfig());
    ...
    client.close();
```

4.1.2. 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());
}
```

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(String topic, Integer partition, K key, V value, RecordMetadata
recordMetadata);

    void onError(String topic, Integer partition, K key, V value, Exception
exception);

    boolean isInterestedInSuccess();

}
```

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

`onSuccess` is called only if `isInterestedInSuccess` returns `true`.

For convenience, the abstract `ProducerListenerAdapter` is provided in case you want to implement only one of the methods. It returns `false` for `isInterestedInSuccess`.

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("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.

If you wish to block the sending thread to await the result, you can invoke the future's `get()` method. 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. Note, however, that flushing likely significantly reduces performance.

Examples

This section shows examples of sending messages to Kafka:

Example 2. 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 ListenableFutureCallback<SendResult<Integer, String>>()
{

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

        @Override
        public void onFailure(Throwable 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);
    }
}
```

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.

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());
}
```

Using `ReplyingKafkaTemplate`

Version 2.1.3 introduced a subclass of `KafkaTemplate` to provide request/reply semantics. The class is named `ReplyingKafkaTemplate` and has one method (in addition to those in the superclass). The following listing shows the method signatures:

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

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

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<Long, 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.

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.

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 topic (`@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`).

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 `Predicate<Collection<ConsumerRecord<K, R>>> 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).

```
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.

4.1.3. 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.

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. The interceptor is not invoked when the listener is a batch listener.

Starting with version 2.3, the `CompositeRecordInterceptor` can be used to invoke multiple interceptors.

Using `KafkaMessageListenerContainer`

The following constructors are available:

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

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

Each takes a `ConsumerFactory` and information about topics and partitions, as well as other configuration in a `ContainerProperties` object. The second constructor is used by the `ConcurrentMessageListenerContainer` (described later) to distribute `TopicPartitionOffset` across the consumer instances. `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: `true`). 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. To restore the previous behavior, you can set the property to `false`.

Using `ConcurrentMessageListenerContainer`

The single constructor is similar to the first `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
> ```

For the second constructor, the `ConcurrentMessageListenerContainer` distributes the `TopicPartition` instances across the delegate `KafkaMessageListenerContainer` instances.

If, say, six `TopicPartition` instances are provided and the `concurrency` is `3`; each container gets two partitions. For five `TopicPartition` 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`:

- `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.

> `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`.

> `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.

> 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) {
    ...
}
```

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.

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();
}
```

Finally, metadata about the message 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`

The following example shows how to use the headers:

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

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"
})
```

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.

@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:

```java
public static class ValidatedClass {

  @Max(10)
  private int bar;

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

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

}
```

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

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

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);

}
```

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);
        // ...
    }
});
```

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) {
        ...
    }

}
```

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`:

```java
@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:

```java
@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.

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 `ErrorHandler`, if configured, or logged otherwise.

> Two error handler interfaces (`ErrorHandler` and `BatchErrorHandler`) are provided. You must configure the appropriate type to match the message listener.

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 adatper. 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.

Listener Consumer Lifecycle Events

The following events are published when containers are started and stopped:

- 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.
* `IdleContainerEvent` - discussed in Detecting Idle and Non-Responsive Consumers.
* `NonResponsiveConsumerEvent` - discussed in Detecting Idle and Non-Responsive Consumers.
* `ConsumerPausedEvent` - discussed in Pausing and Resuming Listener Containers.
* `ConsumerResumedEvent` - discussed in Pausing and Resuming Listener Containers.
* `ConsumerStoppingEvent` - published when a consumer begins to stop.
* `ConsumerStartedEvent` - published when a consumer is stopped.

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 `pollInterval` property. By default, this check is performed once every 30 seconds in each container. You can modify this behavior by setting the `monitorInterval` and `noPollThreshold` properties in the `ContainerProperties` when configuring the listener container. Receiving such an event lets you stop the containers, thus waking the consumer so that it can terminate.

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 Listener Consumer Lifecycle 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.

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).

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`. This method is never called if you explicitly assign partitions yourself. Use the `TopicPartitionOffset` in that case.

`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 seekToEnd(String topic, int partition);

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.

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.

```java
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);
    }

}
```

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.

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.

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 `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`.

4.1.4. 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`

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, zombie fencing cannot be supported because a batch may contain records from multiple topics or partitions.

Also see `transactionIdPrefix`.

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.

Transactional Listener Container and Exactly Once Processing

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 offsets 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.

Transaction Synchronization

If you need to synchronize a Kafka transaction with some other transaction, configure the listener container with the appropriate transaction manager (one that supports synchronization, such as the `DataSourceTransactionManager`). Any operations performed on a transactional `KafkaTemplate` from the listener participate in a single transaction. The Kafka transaction is committed (or rolled back) immediately after the controlling transaction. Before exiting the listener, you should invoke one of the template's `sendOffsetsToTransaction` methods (unless you use a `ChainedKafkaTransactionManager`). For convenience, the listener container binds its consumer group ID to the thread, so, generally, you can use the first method. The following listing shows the two method signatures:

```
void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets);

void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets,
String consumerGroupId);
```

The following example shows how to use the first signature of the `sendOffsetsToTransaction` method:

```
@Bean
KafkaMessageListenerContainer container(ConsumerFactory<String, String> cf,
            final KafkaTemplate template) {
    ContainerProperties props = new ContainerProperties("foo");
    props.setGroupId("group");
    props.setTransactionManager(new SomeOtherTransactionManager());
    ...
    props.setMessageListener((MessageListener<String, String>) m -> {
        template.send("foo", "bar");
        template.send("baz", "qux");
        template.sendOffsetsToTransaction(
            Collections.singletonMap(new TopicPartition(m.topic(), m.partition()),
                new OffsetAndMetadata(m.offset() + 1)));
    });
    return new KafkaMessageListenerContainer<>(cf, props);
}
```

The offset to be committed is one greater than the offset of the records processed by the listener.

You should call this only when you use transaction synchronization. When a listener container is configured to use a `KafkaTransactionManager` or `ChainedKafkaTransactionManager`, it takes care of sending the offsets to the transaction.

See Example of Transaction Synchronization for an example application that synchronizes JDBC and Kafka transactions.

Using `ChainedKafkaTransactionManager`

The `ChainedKafkaTransactionManager` was introduced in version 2.1.3. This is a subclass of `ChainedTransactionManager` that can have exactly one `KafkaTransactionManager`. Since it is a `KafkaAwareTransactionManager`, the container can send the offsets to the transaction in the same way as when the container is configured with a simple `KafkaTransactionManager`. This provides another mechanism for synchronizing transactions without having to send the offsets to the transaction in the listener code. You should chain your transaction managers in the desired order and provide the `ChainedTransactionManager` in the `ContainerProperties`.

See Example of Transaction Synchronization for an example application that synchronizes 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 rather a dichotomy when specifying this property in that, when running multiple instances of the application, 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. 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.

4.1.5. 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() {
    }

}
```

```java
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
```

4.1.6. 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 byi 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:

```java
@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
```

4.1.7. Events

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

- `ListenerContainerIdleEvent`: Issued when no messages have been received in `idleInterval` (if configured).
- `NonResponsiveConsumerEvent`: Issued when the consumer appears to be blocked in the `poll` method.
- `ConsumerPausedEvent`: Issued by each consumer when the container is paused.
- `ConsumerResumedEvent`: Issued by each consumer when the container is resumed.
- `ConsumerStoppingEvent`: Issued by each consumer just before stopping.
- `ConsumerStoppedEvent`: Issued after the consumer is closed. See Thread Safety.
- `ContainerStoppedEvent`: Issued when all consumers have terminated.

> 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 `ContainerIdleEvent` 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 `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 `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.

4.1.8. 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`.

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:

* `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.

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.

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 JsonSerializer<>(MyValueType.class)
            .ignoreTypeHeaders());
}
```

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));
```

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.SERIALIZATION_SELECTOR` to a selector value that is used to select which serializer to use; if a match is not found, an `IllegalStateException` is thrown.

For incoming records, the deserializer uses the same header 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 key `DelegatingSerializer.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.SERIALIZATION_SELECTOR_CONFIG,
    "thing1:com.example.MyThing1Serializer,
thing2:com.example.MyThing2Serializer")

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

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

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. 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 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.

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, version 2.2 introduced the `ErrorHandlingDeserializer2`. This deserializer delegates to a real deserializer (key or value). If the delegate fails to deserialize the record content, the `ErrorHandlingDeserializer2` 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 `ErrorHandlingDeserializer2` 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 `ErrorHandlingDeserializer2` 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 `ErrorHandlingDeserializer2` 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 `ErrorHandlingDeserializer2.KEY_DESERIALIZER_CLASS` and `ErrorHandlingDeserializer2.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,
ErrorHandlingDeserializer2.class);
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer2.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,
ErrorHandlingDeserializer2.class);
consumerProps.put(ErrorHandlingDeserializer2.VALUE_DESERIALIZER_CLASS,
JsonDeserializer.class);
consumerProps.put(ErrorHandlingDeserializer2.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 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.

4.1.9. 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`.

4.1.10. 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.

4.1.11. 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`.

By default, this attribute is not configured.

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.

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

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.

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.

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());
    return factory;
}
```

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. Setting the `maxFailures` property to a negative number causes 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`
- `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 `BinaryExceptionClassifier` with your own configured classifier. See the Javadocs for `SeekToCurrentErrorHandler` 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(BiConsumer<ConsumerRecord<?, ?>,
Exception> recoverer) {
    SeekToCurrentErrorHandler handler = new SeekToCurrentErrorHandler(recoverer);
    handler.addNotRetryableException(IllegalArgumentException.class);
    return handler;
}
```

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. 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` properties 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.

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`.

Publishing Dead-letter Records

As discussed earlier, you can configure the `SeekToCurrentErrorHandler` and `DefaultAfterRollbackProcessor` 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_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.

Starting with version 2.3, when used in conjunction with an ErrorHandlingDeserializer2, 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.

Starting with version 2.3, the recoverer can also be used with Kafka Streams - see Recovery from Deserialization Exceptions for more information.

4.1.12. 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. 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-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<StreamsBuilderFactoryBean>
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;
```

4.2.3. Streams JSON Serialization and Deserialization

For serializing and deserializing data when reading or writing to topics or state stores in JSON format, Spring 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");
```

4.2.4. Using `KafkaStreamsBrancher`

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 KafkaStreamsBrancher<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
```

4.2.5. 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. Starting with version 2.3, you can perform additional customization of that bean, by providing a bean that implements `StreamsBuilderFactoryBeanCustomizer`. There must only be one such bean, or one must be marked `@Primary`.

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.

4.2.6. 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.

4.2.7. `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.

4.2.8. 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_RECOVE
RER, 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`.

4.2.9. 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.KEY_SERDE_CLASS_CONFIG,
Serdes.Integer().getClass().getName());
        props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
        props.put(StreamsConfig.TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
WallclockTimestampExtractor.class.getName());
        return new KafkaStreamsConfiguration(props);
    }

    @Bean
    public StreamsBuilderFactoryBeanCustomizer customizer() {
        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(String::toUpperCase)
                .groupByKey()
                .reduce((String value1, String value2) -> value1 + value2,
                        TimeWindows.of(1000),
                        "windowStore")
                .toStream()
                .map((windowedId, value) -> new KeyValue<>(windowedId.key(),
value))
                .filter((i, s) -> s.length() > 40)
                .to("streamingTopic2");

        stream.print();

        return stream;
    }

}
```

4.3. Testing Applications

The `spring-kafka-test` jar contains some useful utilities to assist with testing your applications.

4.3.1. JUnit

`o.s.kafka.test.utils.KafkaTestUtils` 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> senderProps(EmbeddedKafkaBroker embeddedKafka) {
... }
```

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.

4.3.2. 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));
        ...
      }

}
```

4.3.3. 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.

4.3.4. @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.

4.3.5. @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.

4.3.6. 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:

```java
@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:

```java
@RunWith(SpringRunner.class)
@EmbeddedKafka(topics = "someTopic",
        bootstrapServersProperty = "spring.kafka.bootstrap-servers")
public class MyApplicationTests {

    @Autowired
    private KafkaTemplate<String, String> template;

    @Test
    public void test() {
        ...
    }

}
```

4.3.7. 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);
}
```

4.3.8. 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 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);
}
```

4.3.9. 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> senderProps =

KafkaTestUtils.senderProps(embeddedKafka.getEmbeddedKafka().getBrokersAsString());
        ProducerFactory<Integer, String> pf =
                            new DefaultKafkaProducerFactory<Integer,
String>(senderProps);
        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);
assertThat(received).has(key(2));
assertThat(received).has(partition(0));
assertThat(received).has(value("bar"));
template.send(TEMPLATE_TOPIC, 0, 2, "baz");
received = records.poll(10, TimeUnit.SECONDS);
assertThat(received).has(key(2));
assertThat(received).has(partition(0));
assertThat(received).has(value("baz"));
```

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')}"))
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.

5.2. Example of Transaction Synchronization

The following Spring Boot application is an example of synchronizing database and Kafka transactions.

```
@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 ChainedKafkaTransactionManager<Object, Object> chainedTm(
            KafkaTransactionManager<String, String> ktm,
            DataSourceTransactionManager dstm) {

        return new ChainedKafkaTransactionManager<>(ktm, dstm);
    }

    @Bean
    public DataSourceTransactionManager dstm(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<?, ?>
kafkaListenerContainerFactory(
            ConcurrentKafkaListenerContainerFactoryConfigurer configurer,
            ConsumerFactory<Object, Object> kafkaConsumerFactory,
            ChainedKafkaTransactionManager<Object, Object> chainedTM) {

        ConcurrentKafkaListenerContainerFactory<Object, Object> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        configurer.configure(factory, kafkaConsumerFactory);
        factory.getContainerProperties().setTransactionManager(chainedTM);
        return factory;
    }

    @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")
        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));
```

Chapter 6. Spring Integration

This part of the reference guide shows how to use the `spring-integration-kafka` module of Spring Integration.

6.1. Spring Integration for Apache Kafka

6.1.1. Overview

This documentation pertains to versions 2.0.0 and above. For documentation for earlier releases, see the 1.3.x README.

Spring Integration Kafka is an extension module to the Spring Integration Project.

Spring Integration Kafka is now based on the Spring for Apache Kafka project. It provides the following components:

- Outbound Channel Adapter
- Message-driven Channel Adapter
- Inbound Channel Adapter
- Outbound Gateway
- Inbound Gateway

6.1.2. What's new in Spring Integration for Apache Kafka (version 3.2)

- The pollable `KafkaMessageSource` now implements `Pausable` so the consumer can be `paused` and `resumed`. You must continue to poll the adapter while paused, to avoid a topic/partition rebalance. See the discussion about `max.poll.records` for more information.
- XML configuration is now supported for the gateways and polled inbound channel adapter (in addition to the existing XML support for the other adapters).
- The pollable message source can now be configured to fetch multiple records at-a-time.

6.1.3. Outbound Channel Adapter

The Outbound channel adapter is used to publish messages from a Spring Integration channel to Kafka topics. The channel is defined in the application context and then wired into the application that sends messages to Kafka. Sender applications can publish to Kafka by using Spring Integration messages, which are internally converted to Kafka messages by the outbound channel adapter, as follows:

- The payload of the Spring Integration message is used to populate the payload of the Kafka message.
- By default, the `kafka_messageKey` header of the Spring Integration message is used to populate the key of the Kafka message.

You can customize the target topic and partition for publishing the message through the `kafka_topic` and `kafka_partitionId` headers, respectively.

In addition, the `<int-kafka:outbound-channel-adapter>` provides the ability to extract the key, target topic, and target partition by applying SpEL expressions on the outbound message. To that end, it supports three mutually exclusive pairs of attributes:

- `topic` and `topic-expression`
- `message-key` and `message-key-expression`
- `partition-id` and `partition-id-expression`

These let you specify `topic`, `message-key`, and `partition-id`, respectively, as static values on the adapter or to dynamically evaluate their values at runtime against the request message.

> The `KafkaHeaders` interface (provided by `spring-kafka`) contains constants used for interacting with headers. The `messageKey` and `topic` default headers now require a `kafka_` prefix. When migrating from an earlier version that used the old headers, you need to specify `message-key-expression="headers['messageKey']"` and `topic-expression="headers['topic']"` on the `<int-kafka:outbound-channel-adapter>`. Alternatively, you can change the headers upstream to the new headers from `KafkaHeaders` by using a `<header-enricher>` or a `MessageBuilder`. If you use constant values, you can also configure them on the adapter by using `topic` and `message-key`.

NOTE : If the adapter is configured with a topic or message key (either with a constant or expression), those are used and the corresponding header is ignored. If you wish the header to override the configuration, you need to configure it in an expression, such as the following:

```
topic-expression="headers['topic'] != null ? headers['topic'] : 'myTopic'"
```

The adapter requires a `KafkaTemplate`, which, in turn, requires a suitably configured `KafkaProducerFactory`.

If a `send-failure-channel` (`sendFailureChannel`) is provided and a send failure (sync or async) is received, an `ErrorMessage` is sent to the channel. The payload is a `KafkaSendFailureException` with `failedMessage`, `record` (the `ProducerRecord`) and `cause` properties. You can override the `DefaultErrorMessageStrategy` by setting the `error-message-strategy` property.

If a `send-success-channel` (`sendSuccessChannel`) is provided, a message with a payload of type `org.apache.kafka.clients.producer.RecordMetadata` is sent after a successful send.

If your application uses transactions and the same channel adapter is used to publish messages where the transaction is started by a listener container, as well as publishing where there is no existing transaction, you must configure a `transactionIdPrefix` on the `KafkaTemplate` to override the prefix used by the container or transaction manager. The prefix used by container-initiated transactions (the producer factory or transaction manager property) must be the same on all application instances. The prefix used for producer-only transactions must be unique on all application instances.

Java Configuration

The following example shows how to configure the Kafka outbound channel adapter with Java:

```
@Bean
@ServiceActivator(inputChannel = "toKafka")
public MessageHandler handler() throws Exception {
    KafkaProducerMessageHandler<String, String> handler =
            new KafkaProducerMessageHandler<>(kafkaTemplate());
    handler.setTopicExpression(new LiteralExpression("someTopic"));
    handler.setMessageKeyExpression(new LiteralExpression("someKey"));
    handler.setSuccessChannel(successes());
    handler.setFailureChannel(failures());
    return handler;
}

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

@Bean
public ProducerFactory<String, String> producerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, this.brokerAddress);
    // set more properties
    return new DefaultKafkaProducerFactory<>(props);
}
```

Java DSL Configuration

The following example shows how to configure the Kafka outbound channel adapter Spring Integration Java DSL:

```
@Bean
public ProducerFactory<Integer, String> producerFactory() {
    return new
DefaultKafkaProducerFactory<>(KafkaTestUtils.producerProps(embeddedKafka));
}

@Bean
public IntegrationFlow sendToKafkaFlow() {
    return f -> f
            .<String>split(p -> Stream.generate(() -> p).limit(101).iterator(),
null)
            .publishSubscribeChannel(c -> c
                    .subscribe(sf -> sf.handle(
                            kafkaMessageHandler(producerFactory(), TEST_TOPIC1)

.timestampExpression("T(Long).valueOf('1487694048633')"),
                            e -> e.id("kafkaProducer1")))
                    .subscribe(sf -> sf.handle(
                            kafkaMessageHandler(producerFactory(), TEST_TOPIC2)
                                    .timestamp(m -> 1487694048644L),
                            e -> e.id("kafkaProducer2")))
            );
}

@Bean
public DefaultKafkaHeaderMapper mapper() {
    return new DefaultKafkaHeaderMapper();
}

private KafkaProducerMessageHandlerSpec<Integer, String, ?> kafkaMessageHandler(
        ProducerFactory<Integer, String> producerFactory, String topic) {
    return Kafka
            .outboundChannelAdapter(producerFactory)
            .messageKey(m -> m
                    .getHeaders()
                    .get(IntegrationMessageHeaderAccessor.SEQUENCE_NUMBER))
            .headerMapper(mapper())
            .partitionId(m -> 10)
            .topicExpression("headers[kafka_topic] ?: '" + topic + "'")
            .configureKafkaTemplate(t -> t.id("kafkaTemplate:" + topic));
}
```

XML Configuration

The following example shows how to configure the Kafka outbound channel adapter with XML:

```
<int-kafka:outbound-channel-adapter id="kafkaOutboundChannelAdapter"
                                    kafka-template="template"
                                    auto-startup="false"
                                    channel="inputToKafka"
                                    topic="foo"
                                    sync="false"
                                    message-key-expression="'bar'"
                                    send-failure-channel="failures"
                                    send-success-channel="successes"
                                    error-message-strategy="ems"
                                    partition-id-expression="2">
</int-kafka:outbound-channel-adapter>

<bean id="template" class="org.springframework.kafka.core.KafkaTemplate">
    <constructor-arg>
        <bean class="org.springframework.kafka.core.DefaultKafkaProducerFactory">
            <constructor-arg>
                <map>
                    <entry key="bootstrap.servers" value="localhost:9092" />
                    ... <!-- more producer properties -->
                </map>
            </constructor-arg>
        </bean>
    </constructor-arg>
</bean>
```

6.1.4. Message-driven Channel Adapter

The `KafkaMessageDrivenChannelAdapter` (`<int-kafka:message-driven-channel-adapter>`) uses a `spring-kafka` `KafkaMessageListenerContainer` or `ConcurrentListenerContainer`.

Starting with spring-integration-kafka version 2.1, the `mode` attribute is available. It can accept values of `record` or `batch` (default: `record`). For `record` mode, each message payload is converted from a single `ConsumerRecord`. For `batch` mode, the payload is a list of objects that are converted from all the `ConsumerRecord` instances returned by the consumer poll. As with the batched `@KafkaListener`, the `KafkaHeaders.RECEIVED_MESSAGE_KEY`, `KafkaHeaders.RECEIVED_PARTITION_ID`, `KafkaHeaders.RECEIVED_TOPIC`, and `KafkaHeaders.OFFSET` headers are also lists, with positions corresponding to the position in the payload.

Received messages have certain headers populated. See the `KafkaHeaders` class for more information.

> **!** The `Consumer` object (in the `kafka_consumer` header) is not thread-safe. You must invoke its methods only on the thread that calls the listener within the adapter. If you hand off the message to another thread, you must not call its methods.

When a `retry-template` is provided, delivery failures are retried according to its retry policy. An error-channel is not allowed in this case. You can use the recovery-callback to handle the error when retries are exhausted. In most cases, this is an ErrorMessageSendingRecoverer that sends the ErrorMessage to a channel.

When building an ErrorMessage (for use in the error-channel or recovery-callback), you can customize the error message by setting the error-message-strategy property. By default, a RawRecordHeaderErrorMessageStrategy is used, to provide access to the converted message as well as the raw ConsumerRecord.

Java Configuration

The following example shows how to configure a message-driven channel adapter with Java:

```
@Bean
public KafkaMessageDrivenChannelAdapter<String, String>
            adapter(KafkaMessageListenerContainer<String, String> container) {
    KafkaMessageDrivenChannelAdapter<String, String>
kafkaMessageDrivenChannelAdapter =
            new KafkaMessageDrivenChannelAdapter<>(container,
ListenerMode.record);
    kafkaMessageDrivenChannelAdapter.setOutputChannel(received());
    return kafkaMessageDrivenChannelAdapter;
}

@Bean
public KafkaMessageListenerContainer<String, String> container() throws Exception
{
    ContainerProperties properties = new ContainerProperties(this.topic);
    // set more properties
    return new KafkaMessageListenerContainer<>(consumerFactory(), properties);
}

@Bean
public ConsumerFactory<String, String> consumerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, this.brokerAddress);
    // set more properties
    return new DefaultKafkaConsumerFactory<>(props);
}
```

Java DSL Configuration

The following example shows how to configure a message-driven channel adapter with the Spring Integration Java DSL:

```
@Bean
public IntegrationFlow topic1ListenerFromKafkaFlow() {
    return IntegrationFlows
            .from(Kafka.messageDrivenChannelAdapter(consumerFactory(),
                    KafkaMessageDrivenChannelAdapter.ListenerMode.record,
TEST_TOPIC1)
                    .configureListenerContainer(c ->

c.ackMode(AbstractMessageListenerContainer.AckMode.MANUAL)
                                    .id("topic1ListenerContainer"))
                    .recoveryCallback(new
ErrorMessageSendingRecoverer(errorChannel(),
                            new RawRecordHeaderErrorMessageStrategy()))
                    .retryTemplate(new RetryTemplate())
                    .filterInRetry(true))
            .filter(Message.class, m ->
                            m.getHeaders().get(KafkaHeaders.RECEIVED_MESSAGE_KEY,
Integer.class) < 101,
                    f -> f.throwExceptionOnRejection(true))
            .<String, String>transform(String::toUpperCase)
            .channel(c -> c.queue("listeningFromKafkaResults1"))
            .get();
}
```

Starting with Spring for Apache Kafka version 2.2 (Spring Integration Kafka 3.1), you can also use the container factory that is used for `@KafkaListener` annotations to create `ConcurrentMessageListenerContainer` instances for other purposes. See Container factory for an example.

With the Java DSL, the container does not have to be configured as a `@Bean`, because the DSL registers the container as a bean. The following example shows how to do so:

```
@Bean
public IntegrationFlow topic2ListenerFromKafkaFlow() {
    return IntegrationFlows

.from(Kafka.messageDrivenChannelAdapter(kafkaListenerContainerFactory().createCont
ainer(TEST_TOPIC2),
            KafkaMessageDrivenChannelAdapter.ListenerMode.record)
                .id("topic2Adapter"))
            ...
            get();
}
```

Notice that, in this case, the adapter is given an `id` (`topic2Adapter`). The container is registered in the application context with a name of `topic2Adapter.container`. If the adapter does not have an `id` property, the container's bean name is the container's fully qualified class name plus `#n`, where `n` is incremented for each container.

XML Configuration

The following example shows how to configure a message-driven channel adapter with XML:

```
<int-kafka:message-driven-channel-adapter
        id="kafkaListener"
        listener-container="container1"
        auto-startup="false"
        phase="100"
        send-timeout="5000"
        mode="record"
        retry-template="template"
        recovery-callback="callback"
        error-message-strategy="ems"
        channel="someChannel"
        error-channel="errorChannel" />

<bean id="container1"
class="org.springframework.kafka.listener.KafkaMessageListenerContainer">
    <constructor-arg>
        <bean class="org.springframework.kafka.core.DefaultKafkaConsumerFactory">
            <constructor-arg>
                <map>
                <entry key="bootstrap.servers" value="localhost:9092" />
                ...
                </map>
            </constructor-arg>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean
class="org.springframework.kafka.listener.config.ContainerProperties">
            <constructor-arg name="topics" value="foo" />
        </bean>
    </constructor-arg>

</bean>
```

6.1.5. Inbound Channel Adapter

Introduced in version 3.0.1, the `KafkaMessageSource` provides a pollable channel adapter implementation.

Java Configuration

```
@InboundChannelAdapter(channel = "fromKafka", poller = @Poller(fixedDelay =
"5000"))
@Bean
public KafkaMessageSource<String, String> source(ConsumerFactory<String, String>
cf)  {
    KafkaMessageSource<String, String> source = new KafkaMessageSource<>(cf,
"myTopic");
    source.setGroupId("myGroupId");
    source.setClientId("myClientId");
    return source;
}
```

Refer to the javadocs for available properties.

By default, `max.poll.records` must be either explicitly set in the consumer factory, or it will be forced to 1 if the consumer factory is a `DefaultKafkaConsumerFactory`. Starting with version 3.2, you can set the property `allowMultiFetch` to `true` to override this behavior.

> You must poll the consumer within `max.poll.interval.ms` to avoid a rebalance. If you set `allowMultiFetch` to `true` you must process all the retrieved records, and poll again, within `max.poll.interval.ms`.

Messages emitted by this adapter contain a header `kafka_remainingRecords` with a count of records remaining from the previous poll.

Java DSL Configuration

```
@Bean
public IntegrationFlow flow(ConsumerFactory<String, String> cf)  {
    return IntegrationFlows.from(Kafka.inboundChannelAdapter(cf, "myTopic")
                .groupId("myDslGroupId"), e -> e.poller(Pollers.fixedDelay(5000)))
            .handle(System.out::println)
            .get();
}
```

XML Configuration

```
<int-kafka:inbound-channel-adapter
        id="adapter1"
        consumer-factory="consumerFactory"
        ack-factory="ackFactory"
        topics="topic1"
        channel="inbound"
        client-id="client"
        group-id="group"
        message-converter="converter"
        payload-type="java.lang.String"
        raw-header="true"
        auto-startup="false"
        rebalance-listener="rebal">
    <int:poller fixed-delay="5000"/>
</int-kafka:inbound-channel-adapter>
```

6.1.6. Outbound Gateway

The outbound gateway is for request/reply operations. It differs from most Spring Integration gateways in that the sending thread does not block in the gateway and the reply is processed on the reply listener container thread. If your code invokes the gateway behind a synchronous Messaging Gateway, the user thread blocks there until the reply is received (or a timeout occurs).

> The gateway does not accept requests until the reply container has been assigned its topics and partitions. It is suggested that you add a `ConsumerRebalanceListener` to the template's reply container properties and wait for the `onPartitionsAssigned` call before sending messages to the gateway.

Java Configuration

The following example shows how to configure a gateway with Java:

```
@Bean
@ServiceActivator(inputChannel = "kafkaRequests", outputChannel = "kafkaReplies")
public KafkaProducerMessageHandler<String, String> outGateway(
        ReplyingKafkaTemplate<String, String, String> kafkaTemplate) {
    return new KafkaProducerMessageHandler<>(kafkaTemplate);
}
```

Refer to the javadocs for available properties.

Notice that the same class as the outbound channel adapter is used, the only difference being that the Kafka template passed into the constructor is a `ReplyingKafkaTemplate`. See Using `ReplyingKafkaTemplate` for more information.

The outbound topic, partition, key, and so on are determined in the same way as the outbound adapter. The reply topic is determined as follows:

1. A message header named `KafkaHeaders.REPLY_TOPIC` (if present, it must have a `String` or `byte[]` value) is validated against the template's reply container's subscribed topics.
2. If the template's `replyContainer` is subscribed to only one topic, it is used.

You can also specify a `KafkaHeaders.REPLY_PARTITION` header to determine a specific partition to be used for replies. Again, this is validated against the template's reply container's subscriptions.

Java DSL Configuration

The following example shows how to configure an outbound gateway with the Java DSL:

```
@Bean
public IntegrationFlow outboundGateFlow(
        ReplyingKafkaTemplate<String, String, String> kafkaTemplate) {

    return IntegrationFlows.from("kafkaRequests")
            .handle(Kafka.outboundGateway(kafkaTemplate))
            .channel("kafkaReplies")
            .get();
}
```

Alternatively, you can also use a configuration similar to the following bean:

```
@Bean
public IntegrationFlow outboundGateFlow() {
    return IntegrationFlows.from("kafkaRequests")
            .handle(Kafka.outboundGateway(producerFactory(), replyContainer())
                .configureKafkaTemplate(t -> t.replyTimeout(30_000)))
            .channel("kafkaReplies")
            .get();
}
```

XML Configuration

```
<int-kafka:outbound-gateway
    id="allProps"
    error-message-strategy="ems"
    kafka-template="template"
    message-key-expression="'key'"
    order="23"
    partition-id-expression="2"
    reply-channel="replies"
    reply-timeout="43"
    request-channel="requests"
    requires-reply="false"
    send-success-channel="successes"
    send-failure-channel="failures"
    send-timeout-expression="44"
    sync="true"
    timestamp-expression="T(System).currentTimeMillis()"
    topic-expression="'topic'"/>
```

6.1.7. Inbound Gateway

The inbound gateway is for request/reply operations.

The following example shows how to configure an inbound gateway with Java:

```
@Bean
public KafkaInboundGateway<Integer, String, String> inboundGateway(
        AbstractMessageListenerContainer<Integer, String>container,
        KafkaTemplate<Integer, String> replyTemplate) {

    KafkaInboundGateway<Integer, String, String> gateway =
        new KafkaInboundGateway<>(container, replyTemplate);
    gateway.setRequestChannel(requests);
    gateway.setReplyChannel(replies);
    gateway.setReplyTimeout(30_000);
    return gateway;
}
```

Refer to the javadocs for available properties.

The following example shows how to configure a simple upper case converter with the Java DSL:

```
@Bean
public IntegrationFlow serverGateway(
        ConcurrentMessageListenerContainer<Integer, String> container,
        KafkaTemplate<Integer, String> replyTemplate) {
    return IntegrationFlows
            .from(Kafka.inboundGateway(container, template)
                .replyTimeout(30_000))
            .<String, String>transform(String::toUpperCase)
            .get();
}
```

Alternatively, you could configure an upper-case converter by using code similar to the following:

```
@Bean
public IntegrationFlow serverGateway() {
    return IntegrationFlows
            .from(Kafka.inboundGateway(consumerFactory(), containerProperties(),
                    producerFactory())
                .replyTimeout(30_000))
            .<String, String>transform(String::toUpperCase)
            .get();
}
```

Starting with Spring for Apache Kafka version 2.2 (Spring Integration Kafka 3.1), you can also use the container factory that is used for `@KafkaListener` annotations to create `ConcurrentMessageListenerContainer` instances for other purposes. See Container factory and Message-driven Channel Adapter for examples.

XML Configuration

```
<int-kafka:inbound-gateway
        id="gateway1"
        listener-container="container1"
        kafka-template="template"
        auto-startup="false"
        phase="100"
        request-timeout="5000"
        request-channel="nullChannel"
        reply-channel="errorChannel"
        reply-timeout="43"
        message-converter="messageConverter"
        payload-type="java.lang.String"
        error-message-strategy="ems"
        retry-template="retryTemplate"
        recovery-callback="recoveryCallback"/>
```

See the XML schema for a description of each property.

6.1.8. Message Conversion

A `StringJsonMessageConverter` is provided. See Serialization, Deserialization, and Message Conversion for more information.

When using this converter with a message-driven channel adapter, you can specify the type to which you want the incoming payload to be converted. This is achieved by setting the `payload-type` attribute (`payloadType` property) on the adapter. The following example shows how to do so in XML configuration:

```
<int-kafka:message-driven-channel-adapter
        id="kafkaListener"
        listener-container="container1"
        auto-startup="false"
        phase="100"
        send-timeout="5000"
        channel="nullChannel"
        message-converter="messageConverter"
        payload-type="com.example.Foo"
        error-channel="errorChannel" />

<bean id="messageConverter"

class="org.springframework.kafka.support.converter.MessagingMessageConverter"/>
```

The following example shows how to set the `payload-type` attribute (`payloadType` property) on the adapter in Java configuration:

```
@Bean
public KafkaMessageDrivenChannelAdapter<String, String>
            adapter(KafkaMessageListenerContainer<String, String> container) {
    KafkaMessageDrivenChannelAdapter<String, String>
kafkaMessageDrivenChannelAdapter =
            new KafkaMessageDrivenChannelAdapter<>(container,
ListenerMode.record);
    kafkaMessageDrivenChannelAdapter.setOutputChannel(received());
    kafkaMessageDrivenChannelAdapter.setMessageConverter(converter());
    kafkaMessageDrivenChannelAdapter.setPayloadType(Foo.class);
    return kafkaMessageDrivenChannelAdapter;
}
```

6.1.9. Null Payloads and Log Compaction 'Tombstone' Records

Spring Messaging `Message<?>` objects cannot have `null` payloads. When you use the Kafka endpoints, `null` payloads (also known as tombstone records) are represented by a payload of type `KafkaNull`. See Null Payloads and Log Compaction of 'Tombstone' Records for more information.

Starting with version 3.1 of Spring Integration Kafka, such records can now be received by Spring Integration POJO methods with a true `null` value instead. To do so, mark the parameter with `@Payload(required = false)`. The following example shows how to do so:

```
@ServiceActivator(inputChannel = "fromSomeKafkaInboundEndpoint")
public void in(@Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) String key,
               @Payload(required = false) Customer customer) {
    // customer is null if a tombstone record
    ...
}
```

6.1.10. Calling a Spring Integration flow from a `KStream`

You can use a `MessagingTransformer` to invoke an integration flow from a `KStream`:

```java
@Bean
public KStream<byte[], byte[]> kStream(StreamsBuilder kStreamBuilder,
        MessagingTransformer<byte[], byte[], byte[]> transformer)  transformer) {
    KStream<byte[], byte[]> stream = kStreamBuilder.stream(STREAMING_TOPIC1);
    stream.mapValues((ValueMapper<byte[], byte[]>) String::toUpperCase)
            ...
            .transform(() -> transformer)
            .to(streamingTopic2);

    stream.print(Printed.toSysOut());

    return stream;
}

@Bean
@DependsOn("flow")
public MessagingTransformer<byte[], byte[], String> transformer(
        MessagingFunction function) {

    MessagingMessageConverter converter = new MessagingMessageConverter();
    converter.setHeaderMapper(new SimpleKafkaHeaderMapper("*"));
    return new MessagingTransformer<>(function, converter);
}

@Bean
public IntegrationFlow flow() {
    return IntegrationFlows.from(MessagingFunction.class)
        ...
        .get();
}
```

When an integration flow starts with an interface, the proxy that is created has the name of the flow bean, appended with ".gateway" so this bean name can be used a a `@Qualifier` if needed.

6.1.11. What’s New in Spring Integration for Apache Kafka

See the Spring for Apache Kafka Project Page for a matrix of compatible `spring-kafka` and `kafka-clients` versions.

3.2.x

- The `KafkaMessageSource` 's `Consumer` can now be paused and resumed.
- XML configuration for gateways and the pollable source.
- The `KafkaMessageSource` can now be configured to fetch multiple records on each `poll()`.
- The `MessagingTransformer` allows you to invoke a Spring Integration flow from a Kafka streams topology.

3.1.x

* Update to `spring-kafka` 2.2.x and `kafka-clients` 2.0.0
* Support tombstones in EIP POJO Methods

3.0.x

* Update to `spring-kafka` 2.1.x and `kafka-clients` 1.0.0
* Support `ConsumerAwareMessageListener` (`Consumer` is available in a message header)
* Update to Spring Integration 5.0 and Java 8
* Moved Java DSL to the main project
* Added inbound and outbound gateways (3.0.2)

2.3.x

The 2.3.x branch introduced the following changes:

* Update to `spring-kafka` 1.3.x, including support for transactions and header mapping provided by `kafka-clients` 0.11.0.0
* Support for record timestamps

2.2.x

The 2.2.x branch introduced the following changes:

* Update to `spring-kafka` 1.2.x

2.1.x

The 2.1.x branch introduced the following changes:

* Update to `spring-kafka` 1.1.x, including support of batch payloads
* Support `sync` outbound requests in XML configuration
* Support `payload-type` for inbound channel adapters
* Support for enhanced error handling for the inbound channel adapter (2.1.1)
* Support for send success and failure messages (2.1.2)

2.0.x

The 2.0.x version was the first version to be based on Spring for Apache Kafka and the Java clients. Earlier versions used the scala clients directly.

Chapter 7. 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: Change History

A.1. Changes between 2.1 and 2.2

A.1.1. Kafka Client Version

This version requires the 2.0.0 `kafka-clients` or higher.

A.1.2. 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`.

A.1.3. After Rollback Processing

A new `AfterRollbackProcessor` strategy is provided. See After-rollback Processor for more information.

A.1.4. `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.

A.1.5. 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 terminates. 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 Listener Consumer Lifecycle 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.

A.1.6. @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.

A.1.7. 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.

A.1.8. 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.

A.1.9. 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.

A.1.10. 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 Kafka Streams Support and Configuration for more information.

A.1.11. 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.

A.2. Changes between 2.0 and 2.1

A.2.1. Kafka Client Version

This version requires the 1.0.0 kafka-clients or higher.

> The 1.1.x client is supported with version 2.1.5, but you need to override dependencies as described in [deps-for-11x].

The 1.1.x client is supported natively in version 2.2.

A.2.2. 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.

A.2.3. 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.

A.2.4. 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.

A.2.5. Stateful Retry

Starting with version 2.1.3, you can configure stateful retry. See Stateful Retry for more information.

A.2.6. 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.

A.2.7. 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.

A.2.8. 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.

A.2.9. ReplyingKafkaTemplate

Starting with version 2.1.3, a subclass of `KafkaTemplate` is provided to support request/reply semantics. See Using `ReplyingKafkaTemplate` for more information.

A.2.10. ChainedKafkaTransactionManager

Version 2.1.3 introduced the `ChainedKafkaTransactionManager`. See Using `ChainedKafkaTransactionManager` for more information.

A.2.11. Migration Guide from 2.0

See the 2.0 to 2.1 Migration guide.

A.3. Changes Between 1.3 and 2.0

A.3.1. Spring Framework and Java Versions

The Spring for Apache Kafka project now requires Spring Framework 5.0 and Java 8.

A.3.2. `@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.

A.3.3. Message Listeners

Message listeners can now be aware of the `Consumer` object. See Message Listeners for more information.

A.3.4. Using `ConsumerAwareRebalanceListener`

Rebalance listeners can now access the `Consumer` object during rebalance notifications. See Rebalancing Listeners for more information.

A.4. Changes Between 1.2 and 1.3

A.4.1. 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.

A.4.2. 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.

A.4.3. 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.

A.4.4. 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.

A.4.5. `@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`.

A.4.6. `@EmbeddedKafka` Annotation

For convenience, a test class-level `@EmbeddedKafka` annotation is provided, to register `KafkaEmbedded` as a bean. See Testing Applications for more information.

A.4.7. Kerberos Configuration

Support for configuring Kerberos is now provided. See Kerberos for more information.

A.5. Changes between 1.1 and 1.2

This version uses the 0.10.2.x client.

A.6. Changes between 1.0 and 1.1

A.6.1. Kafka Client

This version uses the Apache Kafka 0.10.x.x client.

A.6.2. 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.

A.6.3. Null Payloads

Null payloads are used to “delete” keys when you use log compaction.

A.6.4. 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.

A.6.5. 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.