Spring for Apache Kafka

1.0.1.RELEASE

Gary Russell , Artem Bilan

Table of Contents

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.

2. 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 will get you up and running as quickly as possible.

2.1 Quick Tour for the impatient

Introduction

This is the 5 minute tour to get started with Spring Kafka.

Prerequisites: install and run Apache Kafka Then 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, e.g. for Maven:

```xml
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>1.0.1.RELEASE</version>
</dependency>
```

And for Gradle:

```
compile 'org.springframework.kafka:spring-kafka:1.0.1.RELEASE'
```

Compatibility

- Apache Kafka 0.9.0.1

- Tested with Spring Framework version dependency is 4.2.5 but it is expected that the framework will work with earlier versions of Spring.

- Annotation-based listeners require Spring Framework 4.1 or higher, however.

- Minimum Java version: 7.

Very, Very Quick

Using plain Java to send and receive a message:

```java
@Test
public void testAutoCommit() throws Exception {
    logger.info("Start auto");
    ContainerProperties containerProps = new ContainerProperties("topic1", "topic2");
    KafkaMessageListenerContainer<Integer, String> container = createContainer(containerProps);
    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();
        }

    });
    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");

}
```

```java
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;
}
```

With Java Configuration

A similar example but with Spring configuration in Java:

```java
@Autowired
private Listener listener;

@Autowired
private KafkaTemplate<Integer, String> template;

@Test
public void testSimple() throws Exception {
    waitListening("foo");
    template.send("annotated1", 0, "foo");
    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. 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.

3.1 Using Spring for Apache Kafka

Sending Messages

KafkaTemplate

The `KafkaTemplate` wraps a producer and provides convenience methods to send data to kafka topics. Both asynchronous and synchronous methods are provided, with the async methods returning a `Future`.

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

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

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

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

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

// Flush the producer.

void flush();
```

The first 3 methods require that a default topic has been provided to the template.

To use the template, configure a producer factory and provide it in the template's constructor:

```
@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");
    ...
    return props;
}

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

The template can also be configured using standard `<bean/>` definitions.

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

When using the methods with a `Message<?>` parameter, topic, partition and key information is provided in a message header:

- `KafkaHeaders.TOPIC`

- `KafkaHeaders.PARTITION_ID`

- `KafkaHeaders.MESSAGE_KEY`

with the message payload being the data.

Optionally, you can configure the `KafkaTemplate` with a `ProducerListener` to get an async callback with the results of the send (success or failure) instead of waiting for the `Future` to complete.

```java
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 only called if `isInterestedInSuccess` returns `true`.

For convenience, the abstract `ProducerListenerAdapter` is provided in case you only want to implement 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.

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

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

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

});
```

The `SendResult` has two properties, a `ProducerRecord` and `RecordMetadata`; refer to 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 which will cause the template to `flush()` on each send. Note, however that flushing will likely significantly reduce performance.

Receiving Messages

Messages can be received by configuring a `MessageListenerContainer` and providing a `MessageListener`, or by using the `@KafkaListener` annotation.

Message Listener Containers

Two `MessageListenerContainer` implementations are provided:

- `KafkaMessageListenerContainer`

- `ConcurrentMessageListenerContainer`

The `KafkaMessageListenerContainer` receives all message from all topics/partitions on a single thread. The `ConcurrentMessageListenerContainer` delegates to 1 or more `KafkaMessageListenerContainer` s to provide multi-threaded consumption.

KafkaMessageListenerContainer

The following constructors are available.

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

public KafkaMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,
                    ContainerProperties containerProperties,
                    TopicPartitionInitialOffset... 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` (see below) to distribute `TopicPartitionInitialOffset` across the consumer instances. `ContainerProperties` has the following constructors:

```
public ContainerProperties(TopicPartitionInitialOffset... topicPartitions)

public ContainerProperties(String... topics)

public ContainerProperties(Pattern topicPattern)
```

The first takes an array of `TopicPartitionInitialOffset` arguments to explicitly instruct the container which partitions to use (using the consumer `assign()` method), and with an optional initial offset: a positive value is an absolute offset; a negative value is relative to the current last offset within a partition. 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.

Refer to the JavaDocs for `ContainerProperties` for more information about the various properties that can be set.

ConcurrentMessageListenerContainer

The single constructor is similar to the first `KafkaListenerContainer` constructor:

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

It also has a property `concurrency`, e.g. `container.setConcurrency(3)` will create 3 `KafkaMessageListenerContainer` s.

For the first constructor, kafka will distribute the partitions across the consumers. For the second constructor, the `ConcurrentMessageListenerContainer` distributes the `TopicPartition` s across the delegate `KafkaMessageListenerContainer` s.

If, say, 6 `TopicPartition` s are provided and the `concurrency` is 3; each container will get 2 partitions. For 5 `TopicPartition` s, 2 containers will get 2 partitions and the third will get 1. If the `concurrency` is greater than the number of `TopicPartitions`, the `concurrency` will be adjusted down such that each container will get one partition.

Committing Offsets

Several options are provided for committing offsets. If the `enable.auto.commit` consumer property is true, kafka will auto-commit the offsets according to its configuration. If it is false, the containers support the following `AckMode` s.

The consumer `poll()` method will return one or more `ConsumerRecords`; the `MessageListener` is called for each record; the following 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 (`AcknowledgingMessageListener`) is responsible to `acknowledge()` the `Acknowledgment`; after which, the same semantics as `BATCH` are applied.

- MANUAL_IMMEDIATE - commit the offset immediately when the `Acknowledgment.acknowledge()` method is called by the listener.

> **Note**
>
> `MANUAL`, and `MANUAL_IMMEDIATE` require the listener to be an `AcknowledgingMessageListener`.

The `commitSync()` or `commitAsync()` method on the consumer is used, depending on the `syncCommits` container property.

```java
public interface AcknowledgingMessageListener<K, V> {

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

}

public interface Acknowledgment {

    void acknowledge();

}
```

This gives the listener control over when offsets are committed.

@KafkaListener Annotation

The `@KafkaListener` annotation provides a mechanism for simple POJO listeners:

```java
public class Listener {

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

}
```

This mechanism requires a listener container factory, which is used to configure the underlying ConcurrentMessageListenerContainer: by default, a bean with name kafkaListenerContainerFactory is expected.

```java
@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.

You can also configure POJO listeners with explicit topics and partitions (and, optionally, their initial offsets):

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

Each partition can be specified in the partitions or partitionOffsets attribute, but not both.

When using manual AckMode, the listener can also be provided with the Acknowledgment; this example also shows how to use a different container factory.

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

Finally, metadata about the message is available from message 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) {
    ...
}
```

Filtering Messages

In certain scenarios, such as rebalancing, a message may be redelivered that has already been processed. 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 thereof](#).

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` where you implement the `filter` method to signal that a message is a duplicate and should be discarded.

A `FilteringAcknowledgingMessageListenerAdapter` is also provided for wrapping an `AcknowledgingMessageListener`. This has an additional property `ackDiscarded` which indicates whether the adapter should acknowledge the discarded record; it is `true` by default.

When using `@KafkaListener`, set the `RecordFilterStrategy` (and optionally `ackDiscarded`) on the container factory and the listener will be wrapped in the appropriate filtering adapter.

Retrying Deliveries

If your listener throws an exception, the default behavior is to invoke the `ErrorHandler`, if configured, or logged otherwise.

To retry deliveries, convenient listener adapters - `RetryingMessageListenerAdapter` and `RetryingAcknowledgingMessageListenerAdapter` are provided, depending on whether you are using a `MessageListener` or an `AcknowledgingMessageListener`.

These can be configured 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` will be invoked, if configured, or logged otherwise.

When using `@KafkaListener`, set the `RetryTemplate` (and optionally `recoveryCallback`) on the container factory and the listener will be wrapped in the appropriate retrying adapter.

Serialization/Deserialization and Message Conversion

Apache Kafka provides a high-level API for serializing/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 simple (de)serializer classes using Producer and/or Consumer configuration properties, e.g.:

```
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 `(De)Serializer` instances for `keys` and/or `values`, respectively.

To meet this API, the `DefaultKafkaProducerFactory` and `DefaultKafkaConsumerFactory` also provide properties to allow to inject a custom `(De)Serializer` to target `Producer/Consumer`.

For this purpose Spring for Apache Kafka also provides `JsonSerializer/JsonDeserializer` implementations based on the Jackson JSON processor. When `JsonSerializer` is pretty simple and just lets to write any Java object as a JSON `byte[]`, the `JsonDeserializer` requires an additional `Class<?> targetType` argument to allow to deserializer consumed `byte[]` to the proper target object. The `JsonDeserializer` can be extended to the particular generic type, when the last one is resolved at runtime, instead of compile-time additional `type` argument:

```
JsonDeserializer<Bar> barDeserializer = new JsonDeserializer<>(Bar.class);
...
JsonDeserializer<Foo> fooDeserializer = new JsonDeserializer<Foo>() { };
```

Both `JsonSerializer` and `JsonDeserializer` can be customized with provided `ObjectMapper`. Plus you can extend them to implement some particular configuration logic in the `configure(Map<String, ?> configs, boolean isKey)` method.

Although `Serializer/Deserializer` API is pretty simple and flexible from the low-level Kafka `Consumer` and `Producer` perspective, it is not enough on the Messaging level, where `KafkaTemplate` and `@KafkaListener` are present. To easy convert to/from `org.springframework.messaging.Message`, Spring for Apache Kafka provides `MessageConverter` abstraction with the `MessagingMessageConverter` implementation and its `StringJsonMessageConverter` customization. The `MessageConverter` can be injected into `KafkaTemplate` instance directly and via `AbstractKafkaListenerContainerFactory` bean definition for the `@KafkaListener.containerFactory()` property:

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

Detecting Idle Asynchronous Consumers

While efficient, one problem with asynchronous consumers is detecting when they are idle - users 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 will be published every `idleEventInterval` milliseconds.

To configure this feature, set the `idleEventInterval` on the container:

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

Or, 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 will be published once per minute while the container is idle.

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 following example combines the `@KafkaListener` and `@EventListener` into a single class. It's important to understand that the application listener will get 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.

The events have 4 properties:

- `source` - the listener container instance

- `id` - the listener id (or container bean name)

- `idleTime` - the time the container had been idle when the event was published

- `topicPartitions` - the topics/partitions that the container was assigned at the time the event was generated

```
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) {
        this.event = event;
        eventLatch.countDown();
    }

}
```

Important

Event listeners will see events for all containers; so, in the example above, 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. Hence we use `startsWith` in the condition.

Caution

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 - it will cause 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 in the event if it is a child container, you should stop the concurrent container instead.

3.2 Testing Applications

Introduction

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

JUnit

`o.s.kafka.test.utils.KafkaUtils` provides some static methods to set up producer and consumer properties:

```
/**
 * Set up test properties for an {@code <Integer, String>} consumer.
 * @param group the group id.
 * @param autoCommit the auto commit.
 * @param embeddedKafka a {@link KafkaEmbedded} instance.
 * @return the properties.
 */
public static Map<String, Object> consumerProps(String group, String autoCommit,
                                    KafkaEmbedded embeddedKafka) { ... }

/**
 * Set up test properties for an {@code <Integer, String>} producer.
 * @param embeddedKafka a {@link KafkaEmbedded} instance.
 * @return the properties.
 */
public static Map<String, Object> senderProps(KafkaEmbedded embeddedKafka) { ... }
```

A JUnit `@Rule` is provided that creates an embedded kafka server.

```java
/**
 * 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 KafkaEmbedded(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 KafkaEmbedded(int count, boolean controlledShutdown, int partitions, String... topics) { ... }
```

The embedded kafka class has a utility method allowing you to consume for all the topics it created:

```java
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:

```java
/**
 * 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) { ... }
```

Usage:

```java
...
template.sendDefault(0, 2, "bar");
ConsumerRecord<Integer, String> received = KafkaTestUtils.getSingleRecord(consumer, "topic");
...
```

When the embedded server is started by JUnit, it sets a system property `spring.embedded.kafka.brokers` to the address of the broker(s). A convenient constant `KafkaEmbedded.SPRING_EMBEDDED_KAFKA_BROKERS` is provided for this property.

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) { ... }
```

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) { ... }
```

Example

Putting it all together:

```java
public class KafkaTemplateTests {

    private static final String TEMPLATE_TOPIC = "templateTopic";

    @ClassRule
    public static KafkaEmbedded embeddedKafka = new KafkaEmbedded(1, true, TEMPLATE_TOPIC);

    @Test
    public void testTemplate() throws Exception {
        Map<String, Object> consumerProps = KafkaTestUtils.consumerProps("testT", "false",
 embeddedKafka);
        DefaultKafkaConsumerFactory<Integer, String> cf =
                            new DefaultKafkaConsumerFactory<Integer, String>(consumerProps);
        KafkaMessageListenerContainer<Integer, String> container =
                            new KafkaMessageListenerContainer<>(cf, TEMPLATE_TOPIC);
        final BlockingQueue<ConsumerRecord<Integer, String>> records = new LinkedBlockingQueue<>();
        container.setMessageListener(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.getPartitionsPerTopic());
        Map<String, Object> senderProps = KafkaTestUtils.senderProps(embeddedKafka);
        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 above uses the hamcrest matchers; with AssertJ, the final part looks like this…

```java
...
        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"));
    }
}
```

4. Spring Integration

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

4.1 Spring Integration Kafka

Introduction

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

- Outbound Channel Adapter

- Message-Driven Channel Adapter

These are discussed in the following sections.

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 via Spring Integration messages, which are internally converted to Kafka messages by the outbound channel adapter, as follows: the payload of the Spring Integration message will be used to populate the payload of the Kafka message, and (by default) the `kafka_messageKey` header of the Spring Integration message will be used to populate the key of the Kafka message.

The target topic and partition for publishing the message can be customized 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 the mutually exclusive pairs of attributes `topic`/`topic-expression`, `message-key`/`message-key-expression`, and `partition-id`/`partition-id-expression`, to allow the specification of `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.

> **Important**
>
> 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>`, or simply change the headers upstream to the new headers from `KafkaHeaders` using a `<header-enricher>` or `MessageBuilder`. Or, of course, configure them on the adapter using `topic` and `message-key` if you are using constant values.

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:

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

The adapter requires a `KafkaTemplate`.

Here is an example of how the Kafka outbound channel adapter is configured with XML:

```xml
<int-kafka:outbound-channel-adapter id="kafkaOutboundChannelAdapter"
                                    kafka-template="template"
                                    auto-startup="false"
                                    channel="inputToKafka"
                                    topic="foo"
                                    message-key-expression="'bar'"
                                    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>
```

As you can see, the adapter requires a `KafkaTemplate` which, in turn, requires a suitably configured `KafkaProducerFactory`.

When using Java Configuration:

```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"));
    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);
}
```

Message Driven Channel Adapter:

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

An example of xml configuration variant is shown here:

```xml
<int-kafka:message-driven-channel-adapter
        id="kafkaListener"
        listener-container="container1"
        auto-startup="false"
        phase="100"
        send-timeout="5000"
        channel="nullChannel"
        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 name="topics" value="foo" />
</bean>
```

When using Java Configuration:

```java
@Bean
public KafkaMessageDrivenChannelAdapter<String, String>
            adapter(KafkaMessageListenerContainer<String, String> container) {
    KafkaMessageDrivenChannelAdapter<String, String> kafkaMessageDrivenChannelAdapter =
            new KafkaMessageDrivenChannelAdapter<>(container);
    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);
}
```

5. Other Resources

In addition to this reference documentation, there exist a number of other resources that may help you learn about Spring and Apache Kafka.

Appendix A. Change History