



Spring for Apache Kafka

1.0.0.M1

Gary Russell

Copyright © 2016 Pivotal Software Inc.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

1. Preface	1
2. Introduction	2
2.1. Quick Tour for the impatient	2
Introduction	2
Compatibility	2
Very, Very Quick	2
With Java Configuration	4
3. Reference	6
3.1. Using Spring for Apache Kafka	6
Sending Messages with the <code>KafkaTemplate</code>	6
Receiving Messages	7
Message Listener Containers	7
<code>@KafkaListener</code> Annotation	9
3.2. Testing Applications	10
Introduction	10
JUnit	10
Hamcrest Matchers	10
AssertJ Conditions	11
Example	11
4. Spring Integration	13
4.1. Spring Integration Kafka	13
5. Other Resources	14
A. Change History	15

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:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>1.0.0.M1</version>
</dependency>
```

And for Gradle:

```
compile 'org.springframework.kafka:spring-kafka:1.0.0.M1'
```

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:

```

@Test
public void testAutoCommit() throws Exception {
    logger.info("Start auto");
    KafkaMessageListenerContainer<Integer, String> container = createContainer();
    final CountDownLatch latch = new CountDownLatch(4);
    container.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.convertAndSend(0, "foo");
    template.convertAndSend(2, "bar");
    template.convertAndSend(0, "baz");
    template.convertAndSend(2, "qux");
    template.flush();
    assertTrue(latch.await(60, TimeUnit.SECONDS));
    container.stop();
    logger.info("Stop auto");
}

private KafkaMessageListenerContainer<Integer, String> createContainer() {
    Map<String, Object> props = consumerProps();
    DefaultKafkaConsumerFactory<Integer, String> cf =
        new DefaultKafkaConsumerFactory<Integer, String>(props);
    KafkaMessageListenerContainer<Integer, String> container =
        new KafkaMessageListenerContainer<>(cf, topic1);
    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,
        "org.apache.kafka.common.serialization.IntegerDeserializer");
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");
    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,
        "org.apache.kafka.common.serialization.IntegerSerializer");
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringSerializer");
    return props;
}

```

With Java Configuration

A similar example but with Spring configuration in Java:

```

@Autowired
private Listener listener;

@Autowired
private KafkaTemplate<Integer, String> template;

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

@Configuration
@EnableKafka
public class Config {

    @Bean
    SimpleKafkaListenerContainerFactory<Integer, String>
        kafkaListenerContainerFactory() {
        SimpleKafkaListenerContainerFactory<Integer, String> factory =
            new SimpleKafkaListenerContainerFactory<>();
        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 with the `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`.

```
// Async methods

Future<RecordMetadata> convertAndSend(V data);

Future<RecordMetadata> convertAndSend(K key, V data);

Future<RecordMetadata> convertAndSend(int partition, K key, V data);

Future<RecordMetadata> convertAndSend(String topic, V data);

Future<RecordMetadata> convertAndSend(String topic, K key, V data);

Future<RecordMetadata> convertAndSend(String topic, int partition, K key, V data);

// Sync methods

RecordMetadata syncConvertAndSend(V data)
    throws InterruptedException, ExecutionException;

RecordMetadata syncConvertAndSend(K key, V data)
    throws InterruptedException, ExecutionException;

RecordMetadata syncConvertAndSend(int partition, K key, V data)
    throws InterruptedException, ExecutionException;

RecordMetadata syncConvertAndSend(String topic, V data)
    throws InterruptedException, ExecutionException;

RecordMetadata syncConvertAndSend(String topic, K key, V data)
    throws InterruptedException, ExecutionException;

RecordMetadata syncConvertAndSend(String topic, int partition, K key, V data)
    throws InterruptedException, ExecutionException;

// Flush the producer.

void flush();
```

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.

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.

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,
                                     TopicPartition... topicPartitions)

public KafkaMessageListenerContainer(ConsumerFactory<K, V> consumerFactory, String... topics)

public KafkaMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,
                                     Pattern topicPattern)

```

Each takes a `ConsumerFactory` and information about topics and partitions.

The first takes a list of `TopicPartition` arguments to explicitly instruct the container which partitions to use (using the consumer `assign()` method). The second takes a list of topics and Kafka allocates the partitions based on the `group.id` property - distributing partitions across the group. The third is similar to the second, but uses a regex `Pattern` to select the topics.

ConcurrentMessageListenerContainer

The constructors are similar to the `KafkaListenerContainer`:

```
public ConcurrentMessageListenerContainer(ConsumerFactory<K, V> consumerFactory, TopicPartition...
    topicPartitions)

public ConcurrentMessageListenerContainer(ConsumerFactory<K, V> consumerFactory, String... topics)

public ConcurrentMessageListenerContainer(ConsumerFactory<K, V> consumerFactory, Pattern topicPattern)
```

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

For the second and third container, kafka will distribute the partitions across the consumers. For the first 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** - call `commitAsync()` when the listener returns after processing the record.
- **BATCH** - call `commitAsync()` when all the records returned by the `poll()` have been processed.
- **TIME** - call `commitAsync()` when all the records returned by the `poll()` have been processed as long as the `ackTime` since the last commit has been exceeded.
- **COUNT** - call `commitAsync()` 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 **COUNT_TIME** are applied.
- **MANUAL_IMMEDIATE** - call `commitAsync()` immediately when the `Acknowledgment.acknowledge()` method is called by the listener - must be executed on the container's thread.

Note

`MANUAL` and `MANUAL_IMMEDIATE` require the listener to be an `AcknowledgingMessageListener`.

```
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:

```
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.

```
@Bean
KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>>
kafkaListenerContainerFactory() {
    SimpleKafkaListenerContainerFactory<Integer, String> factory =
        new SimpleKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setConcurrency(3);
    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;
}
```

You can also configure POJO listeners with explicit topics and partitions:

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

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

```

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

```

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.

```

/**
 * 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) { ... }

```

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:

```

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.syncConvertAndSend("foo");
        assertThat(records.poll(10, TimeUnit.SECONDS), hasValue("foo"));
        template.syncConvertAndSend(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.syncConvertAndSend(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...

```

...
assertThat(records.poll(10, TimeUnit.SECONDS)).has(value("foo"));
template.syncConvertAndSend(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.syncConvertAndSend(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

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