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

4.1.1 Configuring Topics

If you define a `KafkaAdmin` bean in your application context, it can automatically add topics to the broker. Simply add a `NewTopic` `@Bean` for each topic to the application context.

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

@Bean
public NewTopic topic1() {
    return new NewTopic("foo", 10, (short) 2);
}

@Bean
public NewTopic topic2() {
    return new NewTopic("bar", 10, (short) 2);
}
```

By default, if the broker is not available, a message will be logged, but the context will continue 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` and the context will fail to initialize.

The admin does not alter existing topics; it will log (INFO) if the number of partitions don't match.

4.1.2 Sending Messages

KafkaTemplate

Overview

The `KafkaTemplate` wraps a producer and provides convenience methods to send data to kafka topics.

```
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);

}
```

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

The API which take in a `timestamp` as a parameter will store this timestamp in the record. The behavior of the user provided timestamp is stored is dependent on the timestamp type configured on the Kafka topic. If the topic is configured to use `CREATE_TIME` then the user specified timestamp will be recorded or generated if not specified. If the topic is configured to use `LOG_APPEND_TIME` then the user specified timestamp will be ignored and broker will add in the local broker time.

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

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");
    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());
}
```

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`
- `KafkaHeaders.TIMESTAMP`

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.

```
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.

```
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.

Examples

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);
    }
}
```

Transactions

The 0.11.0.0 client library added support for transactions. Spring for Apache Kafka adds support in several 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 `close()` s 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.

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` etc). If a transaction is active, any `KafkaTemplate` operations performed within the scope of the transaction will use the transaction's `Producer`. The manager will commit or rollback the transaction depending on success or failure. The `KafkaTemplate` must be configured to use the same `ProducerFactory` as the transaction manager.

Transactional Listener Container

You can provide a listener container with a `KafkaTransactionManager` instance; when so configured, the container will start a transaction before invoking the listener. If the listener successfully processes the record (or records when using a `BatchMessageListener`), the container will send the offset(s) to the transaction 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 records will be retrieved on the next poll.

Transaction Synchronization

If you need to synchronize a Kafka transaction with some other transaction; simply 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 will participate in a single transaction. The Kafka transaction will be committed (or rolled back) immediately after the controlling transaction. Before exiting the listener, you should invoke one of the template's `sendOffsetsToTransaction` methods. For convenience, the listener container binds its consumer group id to the thread so, generally, you can use the first method:

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

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

For example:

```
@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 record(s) processed by the listener.

Important

This should only be called when using transaction synchronization. When a listener container is configured to use a `KafkaTransactionManager`, it will take care of sending the offsets to the transaction.

KafkaTemplate Local Transactions

You can use the `KafkaTemplate` to execute a series of operations within a local transaction.

```
boolean result = template.executeInTransaction(t -> {
    t.sendDefault("foo", "bar");
    t.sendDefault("baz", "qux");
    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 will not be used; a new "nested" transaction is used.

4.1.3 Receiving Messages

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

Message Listeners

When using a Message Listener Container you must provide a listener to receive data. There are currently eight supported interfaces for message listeners:

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

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

}

public interface AcknowledgingMessageListener<K, V> { ❷

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

}

public interface ConsumerAwareMessageListener<K, V> extends MessageListener<K, V> { ❸
```

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

}

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

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

}

public interface BatchMessageListener<K, V> { ❺

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

}

public interface BatchAcknowledgingMessageListener<K, V> { ❻

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

}

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

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

}

public interface BatchAcknowledgingConsumerAwareMessageListener<K, V> extends BatchMessageListener<K, V> { ❽

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

}
```

❶ Use this for processing individual `ConsumerRecord` s received from the kafka consumer `poll()` operation when using auto-commit, or one of the container-managed commit methods.

❷ Use this for processing individual `ConsumerRecord` s received from the kafka consumer `poll()` operation when using one of the manual commit methods.

❸ Use this for processing individual `ConsumerRecord` s 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.

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

❺ Use this for processing all `ConsumerRecord` s 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 using this interface since the listener is given the complete batch.

❻ Use this for processing all `ConsumerRecord` s received from the kafka consumer `poll()` operation when using one of the manual commit methods.

❼ Use this for processing all `ConsumerRecord` s 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 using this interface since the listener is given the complete batch. Access to the `Consumer` object is provided.

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

Important

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/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 by default; a negative value is relative to the current last offset within a partition by default. A constructor for `TopicPartitionInitialOffset` is provided that takes an additional `boolean` argument. 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, use the `ContainerProps.setMessageListener` method when creating the Container:

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

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.

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

Starting with *version 1.3*, the `MessageListenerContainer` provides an access to the metrics of the underlying `KafkaConsumer`. In 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`.

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 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.

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

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

The `Acknowledgment` has this method:

```
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 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.

```
@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.

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

```
@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.

```
@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, the following header names can be used for retrieving the headers of the message:

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

```
@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
        ) {
    ...
}
```

Starting with *version 1.1*, `@KafkaListener` methods can be configured to receive the entire batch of consumer records received from the consumer poll. To configure the listener container factory to create batch listeners, set the `batchListener` property:

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

To receive a simple list of payloads:

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

The topic, partition, offset etc are available in headers which parallel the payloads:

```
@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, etc in each message, but it must be the only parameter (aside from an optional `Acknowledgment` when using manual commits) defined on the method:

```
@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) {
    ...
}
```

You can also receive a list of `ConsumerRecord<?, ?>` objects but it must be the only parameter (aside from an optional `Acknowledgment` when using manual commits) defined on the method:

```
@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 2.0*, the `id` attribute (if present) is used as the Kafka `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`.

Container Thread Naming

Listener containers currently use two task executors, one to invoke the consumer and another which will be 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 don't provide a consumer executor, a `SimpleAsyncTaskExecutor` is used; this executor creates threads with names `<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/start.

@KafkaListener on a class

When using `@KafkaListener` at the class-level, you specify `@KafkaHandler` at the method level. When messages are delivered, the converted message payload type is used to determine which method to call.

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

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

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

}
```

Rebalance Listeners

`ContainerProperties` has a property `consumerRebalanceListener` which takes an implementation of the Kafka client's `ConsumerRebalanceListener` interface. If this property is not provided, the container will configure a simple logging listener that logs rebalance events under the `INFO` level. The framework also adds a sub-interface `ConsumerAwareRebalanceListener`:

```
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; for example:

```
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 will be 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 3 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.

The result of the expression evaluation must be a `String` representing the topic name.

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

@KafkaListener(topics = "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) {
        ...
    }

}
```

When using `@SendTo`, the `ConcurrentKafkaListenerContainerFactory` must be configured with a `KafkaTemplate` in its `replyTemplate` property, to perform the send. Note: only the simple `send(topic, value)` method is used, so you may wish to create a subclass to generate the partition and/or key…

```
@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);
        }

        ...

    };
}
```

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.

```
@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 Section 4.1.7, “Handling Exceptions” for more information.

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.

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

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 are provided `ErrorHandler` and `BatchErrorHandler`; the appropriate type must be configured to match the Message Listener.

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.

The contents of the `RetryContext` passed into the `RecoveryCallback` will depend on the type of listener. The context will always have an attribute `record` which is the record for which the failure occurred. If your listener is acknowledging and/or consumer aware, additional attributes `acknowledgment` and/or `consumer` will be available. For convenience, the `RetryingAcknowledgingMessageListenerAdapter` provides static constants for these keys. See its javadocs 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. Users wishing retry capabilities, when using a batch listener, are advised to use a `RetryTemplate` within the listener itself.

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 5 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
- `consumer` - a reference to the kafka `Consumer` object; for example, if the consumer was previously `pause()` d, it can be `resume()` d when the event is received.

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

```
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) {
        ...
    }

}
```

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.

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 `the section called “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, simply set the initial offset (if desired) in the configured `TopicPartitionInitialOffset` arguments (see the section called “Message Listener Containers”). You can also seek to a specific offset at any time.

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

The first is called when the container is started; this callback should be used when seeking at some arbitrary time after initialization. You should save a reference to the callback; if you are using 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, the second method is called when assignments change. You can use this method, for example, for setting initial offsets for the partitions, by calling the callback; you must use the callback argument, not the one passed into `registerSeekCallback`. This method will never be called if you explicitly assign partitions yourself; use the `TopicPartitionInitialOffset` in that case.

The callback has these methods:

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

void seekToBeginning(String topic, int partition);

void seekToEnd(String topic, int partition);
```

You can also perform seek operations from `onIdleContainer()` when an idle container is detected; see the section called “Detecting Idle Asynchronous Consumers” for how to enable idle container detection.

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

4.1.4 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 object mapper. The `JsonSerializer` is quite simple and just 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.

```
JsonDeserializer<Bar> barDeserializer = new JsonDeserializer<>(Bar.class);
```

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

Although the `Serializer`/`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, either when using `@KafkaListener` or Spring Integration. To easily convert to/from `org.springframework.messaging.Message`, Spring for Apache Kafka provides a `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) {
...
}
```

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

> This type inference can only be achieved 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.

> When using the `StringJsonMessageConverter`, you should use a `StringDeserializer` in the kafka consumer configuration and `StringSerializer` in the kafka producer configuration, when using Spring Integration or the `KafkaTemplate.send(Message<?> message)` method.

4.1.5 Message Headers

The 0.11.0.0 client introduced support for headers in messages. Spring for Apache Kafka *version 2.0* now supports mapping these headers to/from `spring-messaging` `MessageHeaders`.

> Previous versions mapped `ConsumerRecord` and `ProducerRecord` to spring-messaging `Message<?>` where the value property is mapped to/from the `payload` and other properties (`topic`, `partition`, etc) were mapped to headers. This is still the case but additional, arbitrary, headers can now be mapped.

Apache Kafka headers have a simple API:

```
public interface Header {

    String key();

    byte[] value();

}
```

The `KafkaHeaderMapper` strategy is provided to map header entries between Kafka `Headers` and `MessageHeaders`:

```
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 key, `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` s 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.

```
public DefaultKafkaHeaderMapper() {
    ...
}

public DefaultKafkaHeaderMapper(ObjectMapper objectMapper) {
    ...
}

public DefaultKafkaHeaderMapper(String... patterns) {
    ...
}

public DefaultKafkaHeaderMapper(ObjectMapper objectMapper, String... patterns) {
    ...
}
```

The first constructor will use a default Jackson `ObjectMapper` and map most headers, as discussed above. The second constructor will use the provided Jackson `ObjectMapper` and map most headers, as discussed above. The third constructor will use a default Jackson `ObjectMapper` and map headers according to the provided patterns. The third constructor will use the provided Jackson `ObjectMapper` and map headers according to the provided patterns.

Patterns are rather simple and can contain either a leading or trailing wildcard `*`, or both, e.g. `*.foo.*`. Patterns can be negated with a leading `!`. The first pattern that matches a header name wins (positive or negative).

When providing your own patterns, it is recommended to include `!id` and `!timestamp` since these headers are read-only on the inbound side.

Important

By default, the mapper will only deserialize classes in `java.lang` and `java.util`. You can trust other (or all) packages by adding trusted packages using the `addTrustedPackages` method. If you are receiving messages from untrusted sources, you may wish to add just those packages that you trust. To trust all packages use `mapper.addTrustedPackages("*")`.

The `DefaultKafkaHeaderMapper` is used in the `MessagingMessageConverter` and `BatchMessagingMessageConverter` by default, 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 the converter has 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 (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).

Important

The `Jackson` `ObjectMapper` (even if provided) will be enhanced to support deserializing `org.springframewor.utils.MimeType` objects, often used in the `spring-messaging` `ContentType` header. If you don't wish your mapper to be enhanced in this way, for some reason, you should subclass the `DefaultKafkaHeaderMapper` and override `getObjectMapper()` to return your mapper.

4.1.6 Log Compaction

When using Log Compaction, it is possible to send and receive messages with `null` payloads which identifies the deletion of a key.

Starting with *version 1.0.3*, this is now fully supported.

To send a `null` payload using the `KafkaTemplate` simply 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, a special payload type `KafkaNull` is used and the framework will send `null`. For convenience, the static `KafkaNull.INSTANCE` is provided.

When using a message listener container, the received `ConsumerRecord` will have a `null` `value()`.

To configure the `@KafkaListener` to handle `null` payloads, you must use the `@Payload` annotation with `required = false`; you will usually also need the key so your application knows which key was "deleted":

```
@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 using a class-level `@KafkaListener`, some additional configuration is needed - a `@KafkaHandler` method with a `KafkaNull` payload:

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

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

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

    @KafkaHandler
    public void delete(@Payload(required = false) KafkaNull nul, @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) int key) {
        ...
    }

}
```

4.1.7 Handling Exceptions

You can specify a global error handler used for all listeners in the container factory.

```
@Bean
public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>>
        kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    ...
    factory.getContainerProperties().setErrorHandler(myErrorHandler);
    ...
    return factory;
}
```

By default, if an annotated listener method throws an exception, it is thrown to the container, and the message will be handled according to the container configuration. Nothing is returned to the sender.

Starting with *version 2.0*, the `@KafkaListener` annotation has a new attribute: `errorHandler`.

This attribute is not configured by default.

Use the `errorHandler` to provide the bean name of a `KafkaListenerErrorHandler` implementation. This functional interface has one method:

```
@FunctionalInterface
public interface KafkaListenerErrorHandler {

    Object handleError(Message<?> message, ListenerExecutionFailedException exception) throws Exception;

}
```

As you can see, you have access to the spring-messaging `Message<?>` object produced by the message converter and the exception that was thrown by the listener, wrapped in a `ListenerExecutionFailedException`. The error handler can throw the original or a new exception which will be 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, via the 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; note however, these are simplistic implementations and you would probably want more checking in the error handler.

```
@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;
    };
}
```

And 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.

Similarly, the container-level error handler (`ErrorHandler` and `BatchErrorHandler`) have sub-interfaces `ConsumerAwareErrorHandler` and `ConsumerAwareBatchErrorHandler` with method signatures:

```
void handle(Exception thrownException, ConsumerRecord<?, ?> data, Consumer<?, ?> consumer);

void handle(Exception thrownException, ConsumerRecords<?, ?> data, Consumer<?, ?> consumer);
```

respectively.

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 container property `ackOnError` to false when making adjustments; otherwise any pending acks will be applied after your repositioning.

4.1.8 Kerberos

Starting with version 2.0 a `KafkaJaasLoginModuleInitializer` class has been added to assist with Kerberos configuration. Simply add this bean, with the desired configuration, to your application context.

```
@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

4.2.1 Introduction

Starting with *version 1.1.4*, Spring for Apache Kafka provides first class support for Kafka Streams. For using 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 isn't downloaded transitively.

4.2.2 Basics

The reference Apache Kafka Streams documentation suggests this 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.

KStreamBuilder builder = ...;  // when using the Kafka Streams DSL
//
// OR
//
```

```
TopologyBuilder builder = ...; // when using the Processor API

// 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: `KStreamBuilder` (which extends `TopologyBuilder` as well) with an API to build `KStream` (or `KTable`) instances and `KafkaStreams` to manage their lifecycle. Note: all `KStream` instances exposed to a `KafkaStreams` instance by a single `KStreamBuilder` will be started and stopped at the same time, even if they have a fully different logic. In other words all our streams defined by a `KStreamBuilder` are tied with a single lifecycle control. Once a `KafkaStreams` instance has been closed via `streams.close()` it cannot be restarted, and a new `KafkaStreams` instance to restart stream processing must be created instead.

4.2.3 Spring Management

To simplify the usage of Kafka Streams from the Spring application context perspective and utilize the lifecycle management via container, the Spring for Apache Kafka introduces `KStreamBuilderFactoryBean`. This is an `AbstractFactoryBean` implementation to expose a `KStreamBuilder` singleton instance as a bean:

```
@Bean
public FactoryBean<KStreamBuilder> myKStreamBuilder(StreamsConfig streamsConfig) {
    return new KStreamBuilderFactoryBean(streamsConfig);
}
```

The `KStreamBuilderFactoryBean` also implements `SmartLifecycle` to manage lifecycle of an internal `KafkaStreams` instance. Similar to the Kafka Streams API, the `KStream` instances must be defined before starting the `KafkaStreams`, and that also applies for the Spring API for Kafka Streams. Therefore we have to declare `KStream` s on the `KStreamBuilder` before the application context is refreshed, when we use default `autoStartup = true` on the `KStreamBuilderFactoryBean`. For example, `KStream` can be just as a regular bean definition, meanwhile the Kafka Streams API is used without any impacts:

```
@Bean
public KStream<?, ?> kStream(KStreamBuilder kStreamBuilder) {
    KStream<Integer, String> stream = kStreamBuilder.stream(STREAMING_TOPIC1);
    // Fluent KStream API
    return stream;
}
```

If you would like to control lifecycle manually (e.g. stop and start by some condition), you can reference the `KStreamBuilderFactoryBean` bean directly using factory bean (`&`) prefix. Since `KStreamBuilderFactoryBean` utilize its internal `KafkaStreams` instance, it is safe to stop and restart it again - a new `KafkaStreams` is created on each `start()`. Also consider using different `KStreamBuilderFactoryBean` s, if you would like to control lifecycles for `KStream` instances separately.

You can specify `KafkaStreams.StateListener` and `Thread.UncaughtExceptionHandler` options on the `KStreamBuilderFactoryBean` which are delegated to the internal `KafkaStreams` instance. That internal `KafkaStreams` instance can be accessed via `KStreamBuilderFactoryBean.getKafkaStreams()` if you need to perform some `KafkaStreams` operations directly. You can autowire `KStreamBuilderFactoryBean` bean by type, but you should be sure that you use full type in the bean definition, for example:

```
@Bean
public KStreamBuilderFactoryBean myKStreamBuilder(StreamsConfig streamsConfig) {
    return new KStreamBuilderFactoryBean(streamsConfig);
}
...
@Autowired
private KStreamBuilderFactoryBean myKStreamBuilderFactoryBean;
```

Or add `@Qualifier` for injection by name if you use interface bean definition:

```
@Bean
public FactoryBean<KStreamBuilder> myKStreamBuilder(StreamsConfig streamsConfig) {
    return new KStreamBuilderFactoryBean(streamsConfig);
}
...
@Autowired
@Qualifier("&myKStreamBuilder")
private KStreamBuilderFactoryBean myKStreamBuilderFactoryBean;
```

4.2.4 JSON Serdes

For serializing and deserializing data when reading or writing to topics or state stores in JSON format, Spring Kafka provides a `JsonSerde` implementation using JSON, delegating to the `JsonSerializer` and `JsonDeserializer` described in the serialization/deserialization section. The `JsonSerde` provides the same configuration options via its constructor (target type and/or `ObjectMapper`). In the following example we use the `JsonSerde` to serialize and deserialize the `Foo` 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<>(Foo.class), "foos");
```

4.2.5 Configuration

To configure the Kafka Streams environment, the `KStreamBuilderFactoryBean` requires a `Map` of particular properties or a `StreamsConfig` instance. See Apache Kafka documentation for all possible options.

To avoid boilerplate code for most cases, especially when you develop micro services, Spring for Apache Kafka provides the `@EnableKafkaStreams` annotation, which should be placed alongside with `@Configuration`. Only you need is to declare `StreamsConfig` bean with the `defaultKafkaStreamsConfig` name. A `KStreamBuilder` bean with the `defaultKStreamBuilder` name will be declare in the application context automatically. Any additional `KStreamBuilderFactoryBean` beans can be declared and used as well.

4.2.6 Kafka Streams Example

Putting it all together:

```
@Configuration
@EnableKafka
@EnableKafkaStreams
public static class KafkaStreamsConfiguration {

    @Bean(name = KafkaStreamsDefaultConfiguration.DEFAULT_STREAMS_CONFIG_BEAN_NAME)
    public StreamsConfig 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 StreamsConfig(props);
    }

    @Bean
    public KStream<Integer, String> kStream(KStreamBuilder 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

4.3.1 Introduction

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

4.3.2 JUnit

`o.s.kafka.test.utils.KafkaTestUtils` 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) { ... }
```

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

```
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:

```
/**
 * 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:

```
...
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.

With the `KafkaEmbedded.brokerProperties(Map<String, String>)` you can provide additional properties for the Kafka server(s). See Kafka Config for more information about possible broker properties.

4.3.3 @EmbeddedKafka Annotation

It is generally recommended to use the rule as a `@ClassRule` to avoid starting/stopping the broker between tests (and use a different topic for each test). Starting with *version 2.0*, if you are using Spring's test application context caching, you can also declare a `KafkaEmbedded` bean, so a single broker can be used across multiple test classes. The JUnit `ExternalResource` `before()/after()` lifecycle is wrapped to the `afterPropertiesSet()` and `destroy()` Spring infrastructure hooks. For convenience a test class level `@EmbeddedKafka` annotation is provided with the purpose to register `KafkaEmbedded` bean:

```
@RunWith(SpringRunner.class)
@DirtiesContext
@EmbeddedKafka(partitions = 1,
         topics = {
                 KafkaStreamsTests.STREAMING_TOPIC1,
                 KafkaStreamsTests.STREAMING_TOPIC2 })
public class KafkaStreamsTests {

    @Autowired
    private KafkaEmbedded kafkaEmbedded;

        @Test
        public void someTest() {
            Map<String, Object> consumerProps = KafkaTestUtils.consumerProps("testGroup", "true", this.kafkaEmbedded);
        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("${" + KafkaEmbedded.SPRING_EMBEDDED_KAFKA_BROKERS + "}")
        private String brokerAddresses;

        @Bean(name = KafkaStreamsDefaultConfiguration.DEFAULT_STREAMS_CONFIG_BEAN_NAME)
        public StreamsConfig kStreamsConfigs() {
            Map<String, Object> props = new HashMap<>();
            props.put(StreamsConfig.APPLICATION_ID_CONFIG, "testStreams");
            props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, this.brokerAddresses);
            return new StreamsConfig(props);
        }
```

```
    }

}
```

4.3.4 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} asssuming 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);
}
```

4.3.5 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);
}
```

4.3.6 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);
        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.getPartitionsPerTopic());
        Map<String, Object> senderProps =
                            KafkaTestUtils.senderProps(embeddedKafka.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 above uses the hamcrest matchers; with `AssertJ`, the final part looks like this…

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

Prev Next

3. Introduction Home 5. Spring Integration