Using Apache Kafka for Integration and Data Processing Pipelines with Spring

 ENGINEERING

https://gravatar.com/avatar/fb22593caf24e4bb4c98d467cdd247e6?s=20&d=mm [JOSH LONG](https://spring.io/team/jlong)

 APRIL 15, 2015

[30 COMMENTS](https://spring.io/blog/2015/04/15/using-apache-kafka-for-integration-and-data-processing-pipelines-with-spring#disqus_thread)

Applications generated more and more data than ever before and a huge part of the challenge - before it can even be analyzed - is accommodating the load in the first place. [Apache’s Kafka](http://kafka.apache.org/)meets this challenge. It was originally designed by LinkedIn and subsequently open-sourced in 2011. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. The design is heavily influenced by transaction logs. It is a messaging system, similar to traditional messaging systems like RabbitMQ, ActiveMQ, MQSeries, but it’s ideal for log aggregation, persistent messaging, fast (\_hundreds\_ of megabytes per second!) reads and writes, and can accommodate numerous clients. Naturally, this makes it *perfect* for cloud-scale architectures!

Kafka [powers many large production systems](https://cwiki.apache.org/confluence/display/KAFKA/Powered+By). LinkedIn uses it for activity data and operational metrics to power the LinkedIn news feed, and LinkedIn Today, as well as offline analytics going into Hadoop. Twitter uses it as part of their stream-processing infrastructure. Kafka powers online-to-online and online-to-offline messaging at Foursquare. It is used to integrate Foursquare monitoring and production systems with Hadoop-based offline infrastructures. Square uses Kafka as a bus to move all system events through Square’s various data centers. This includes metrics, logs, custom events, and so on. On the consumer side, it outputs into Splunk, Graphite, or Esper-like real-time alerting. Netflix uses it for 300-600BN messages per day. It’s also used by Airbnb, Mozilla, Goldman Sachs, Tumblr, Yahoo, PayPal, Coursera, Urban Airship, Hotels.com, and a seemingly endless list of other big-web stars. Clearly, it’s earning its keep in some powerful systems!

**Installing Apache Kafka**

There are many different ways to get Apache Kafka installed. If you’re on OSX, and you’re using Homebrew, it can be as simple as brew install kafka. You can also [download the latest distribution from Apache](http://kafka.apache.org/downloads.html). I downloaded kafka\_2.10-0.8.2.1.tgz, unzipped it, and then within you’ll find there’s a distribution of [Apache Zookeeper](https://zookeeper.apache.org/) as well as Kafka, so nothing else is required. I installed Apache Kafka in my $HOME directory, under another directory, bin, then I created an environment variable, KAFKA\_HOME, that points to $HOME/bin/kafka.

Start Apache Zookeeper first, specifying where the configuration properties file it requires is:

$KAFKA\_HOME/bin/zookeeper-server-start.sh $KAFKA\_HOME/config/zookeeper.properties

The Apache Kafka distribution comes with default configuration files for both Zookeeper and Kafka, which makes getting started easy. You will in more advanced use cases need to customize these files.

Then start Apache Kafka. It too requires a configuration file, like this:

$KAFKA\_HOME/bin/kafka-server-start.sh $KAFKA\_HOME/config/server.properties

The server.properties file contains, among other things, default values for where to connect to Apache Zookeeper (zookeeper.connect), how much data should be sent across sockets, how many partitions there are by default, and the broker ID (broker.id - which must be unique across a cluster).

There are other scripts in the same directory that can be used to send and receive dummy data, very handy in establishing that everything’s up and running!

Now that Apache Kafka is up and running, let’s look at working with Apache Kafka from our application.

**Some High Level Concepts..**

A Kafka *broker* cluster consists of one or more servers where each may have one or more broker processes running. Apache Kafka is designed to be highly available; there are no *master*nodes. All nodes are interchangeable. Data is replicated from one node to another to ensure that it is still available in the event of a failure.

In Kafka, a *topic* is a category, similar to a JMS destination or both an AMQP exchange and queue. Topics are partitioned, and the choice of which of a topic’s partition a message should be sent to is made by the message producer. Each message in the partition is assigned a unique sequenced ID, its *offset*. More partitions allow greater parallelism for consumption, but this will also result in more files across the brokers.

*Producers* send messages to Apache Kafka broker topics and specify the partition to use for every message they produce. Message production may be synchronous or asynchronous. Producers also specify what sort of replication guarantees they want.

*Consumers* listen for messages on topics and process the feed of published messages. As you’d expect if you’ve used other messaging systems, this is usually (and usefully!) asynchronous.

Like [Spring XD](http://spring.io/projects/spring-xd) and numerous other distributed system, Apache Kafka uses Apache Zookeeper to coordinate cluster information. Apache Zookeeper provides a shared hierarchical namespace (called *znodes*) that nodes can share to understand cluster topology and availability (yet another reason that [Spring Cloud](https://github.com/spring-cloud/spring-cloud-zookeeper) has forthcoming support for it..).

Zookeeper is very present in your interactions with Apache Kafka. Apache Kafka has, for example, two different APIs for acting as a consumer. The higher level API is simpler to get started with and it handles all the nuances of handling partitioning and so on. It will need a reference to a Zookeeper instance to keep the coordination state.

Let’s turn now turn to using Apache Kafka with Spring.

**Using Apache Kafka with Spring Integration**

The recently released [Spring Integration for Apache Kafka 1.1](https://spring.io/blog/2015/03/26/spring-integration-kafka-support-1-1-ga-is-available) is very powerful, and provides inbound adapters for working with both the lower level Apache Kafka API as well as the higher level API.

The adapter, currently, is XML-configuration first, though work is already underway on a Spring Integration Java configuration DSL for the adapter and milestones are available. We’ll look at both here, now.

To make all these examples work, I added the [libs-milestone-local Maven repository](http://repo.spring.io/simple/libs-milestone-local) and used the following dependencies:

* org.apache.kafka:kafka\_2.10:0.8.1.1
* org.springframework.boot:spring-boot-starter-integration:1.2.3.RELEASE
* org.springframework.boot:spring-boot-starter:1.2.3.RELEASE
* org.springframework.integration:spring-integration-kafka:1.1.1.RELEASE
* org.springframework.integration:spring-integration-java-dsl:1.1.0.M1

**Using the Spring Integration Apache Kafka with the Spring Integration XML DSL**

First, let’s look at how to use the Spring Integration outbound adapter to send Message<T>instances from a Spring Integration flow to an external Apache Kafka instance. The example is fairly straightforward: a Spring Integration channel named inputToKafka acts as a conduit that forwards Message<T> messages to the outbound adapter, kafkaOutboundChannelAdapter. The adapter itself can take its configuration from the defaults specified in thekafka:producer-context element or it from the adapter-local configuration overrides. There may be one or many configurations in a given kafka:producer-context element.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:int="http://www.springframework.org/schema/integration"

xmlns:int-kafka="http://www.springframework.org/schema/integration/kafka"

xmlns:task="http://www.springframework.org/schema/task"

xsi:schemaLocation="http://www.springframework.org/schema/integration/kafka http://www.springframework.org/schema/integration/kafka/spring-integration-kafka.xsd

http://www.springframework.org/schema/integration http://www.springframework.org/schema/integration/spring-integration.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task.xsd">

<int:channel id="inputToKafka">

<int:queue/>

</int:channel>

<int-kafka:outbound-channel-adapter

id="kafkaOutboundChannelAdapter"

kafka-producer-context-ref="kafkaProducerContext"

channel="inputToKafka">

<int:poller fixed-delay="1000" time-unit="MILLISECONDS" receive-timeout="0" task-executor="taskExecutor"/>

</int-kafka:outbound-channel-adapter>

<task:executor id="taskExecutor" pool-size="5" keep-alive="120" queue-capacity="500"/>

<int-kafka:producer-context id="kafkaProducerContext">

<int-kafka:producer-configurations>

<int-kafka:producer-configuration broker-list="localhost:9092"

topic="event-stream"

compression-codec="default"/>

</int-kafka:producer-configurations>

</int-kafka:producer-context>

</beans>

Here’s the Java code from a Spring Boot application to trigger message sends using the outbound adapter by sending messages into the incoming inputToKafka MessageChannel.

package xml;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.DependsOn;

import org.springframework.context.annotation.ImportResource;

import org.springframework.integration.config.EnableIntegration;

import org.springframework.messaging.MessageChannel;

import org.springframework.messaging.support.GenericMessage;

@SpringBootApplication

@EnableIntegration

@ImportResource("/xml/outbound-kafka-integration.xml")

public class DemoApplication {

private Log log = LogFactory.getLog(getClass());

@Bean

@DependsOn("kafkaOutboundChannelAdapter")

CommandLineRunner kickOff(@Qualifier("inputToKafka") MessageChannel in) {

return args -> {

for (int i = 0; i < 1000; i++) {

in.send(new GenericMessage<>("#" + i));

log.info("sending message #" + i);

}

};

}

public static void main(String args[]) {

SpringApplication.run(DemoApplication.class, args);

}

}

**Using the New Apache Kafka Spring Integration Java Configuration DSL**

Shortly after the Spring Integration 1.1 release, Spring Integration rockstar [Artem Bilan](https://spring.io/team/artembilan) got to work [on adding a Spring Integration Java Configuration DSL analog](http://repo.spring.io/simple/libs-milestone-local/org/springframework/integration/spring-integration-java-dsl/1.1.0.M1/) and the result is a thing of beauty! It’s not yet GA (you need to add the libs-milestone repository for now), but I encourage you to try it out and kick the tires. It’s working well for me and the Spring Integration team are always keen on getting early feedback whenever possible! Here’s an example that demonstrates both sending messages and consuming them from two differentIntegrationFlows. The producer is similar to the example XML above.

New in this example is the polling consumer. It is batch-centric, and will pull down all the messages it sees at a fixed interval. In our code, the message received will be a map that contains as its keys the topic and as its value another map with the partition ID and the batch (in this case, of 10 records), of records read. There is a MessageListenerContainer-based alternative that processes messages as they come.

package jc;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.DependsOn;

import org.springframework.integration.IntegrationMessageHeaderAccessor;

import org.springframework.integration.config.EnableIntegration;

import org.springframework.integration.dsl.IntegrationFlow;

import org.springframework.integration.dsl.IntegrationFlows;

import org.springframework.integration.dsl.SourcePollingChannelAdapterSpec;

import org.springframework.integration.dsl.kafka.Kafka;

import org.springframework.integration.dsl.kafka.KafkaHighLevelConsumerMessageSourceSpec;

import org.springframework.integration.dsl.kafka.KafkaProducerMessageHandlerSpec;

import org.springframework.integration.dsl.support.Consumer;

import org.springframework.integration.kafka.support.ZookeeperConnect;

import org.springframework.messaging.MessageChannel;

import org.springframework.messaging.support.GenericMessage;

import org.springframework.stereotype.Component;

import java.util.List;

import java.util.Map;

/\*\*

\* Demonstrates using the Spring Integration Apache Kafka Java Configuration DSL.

\* Thanks to Spring Integration ninja <a href="http://spring.io/team/artembilan">Artem Bilan</a>

\* for getting the Java Configuration DSL working so quickly!

\*

\* @author Josh Long

\*/

@EnableIntegration

@SpringBootApplication

public class DemoApplication {

public static final String TEST\_TOPIC\_ID = "event-stream";

@Component

public static class KafkaConfig {

@Value("${kafka.topic:" + TEST\_TOPIC\_ID + "}")

private String topic;

@Value("${kafka.address:localhost:9092}")

private String brokerAddress;

@Value("${zookeeper.address:localhost:2181}")

private String zookeeperAddress;

KafkaConfig() {

}

public KafkaConfig(String t, String b, String zk) {

this.topic = t;

this.brokerAddress = b;

this.zookeeperAddress = zk;

}

public String getTopic() {

return topic;

}

public String getBrokerAddress() {

return brokerAddress;

}

public String getZookeeperAddress() {

return zookeeperAddress;

}

}

@Configuration

public static class ProducerConfiguration {

@Autowired

private KafkaConfig kafkaConfig;

private static final String OUTBOUND\_ID = "outbound";

private Log log = LogFactory.getLog(getClass());

@Bean

@DependsOn(OUTBOUND\_ID)

CommandLineRunner kickOff(

@Qualifier(OUTBOUND\_ID + ".input") MessageChannel in) {

return args -> {

for (int i = 0; i < 1000; i++) {

in.send(new GenericMessage<>("#" + i));

log.info("sending message #" + i);

}

};

}

@Bean(name = OUTBOUND\_ID)

IntegrationFlow producer() {

log.info("starting producer flow..");

return flowDefinition -> {

Consumer<KafkaProducerMessageHandlerSpec.ProducerMetadataSpec> spec =

(KafkaProducerMessageHandlerSpec.ProducerMetadataSpec metadata)->

metadata.async(true)

.batchNumMessages(10)

.valueClassType(String.class)

.<String>valueEncoder(String::getBytes);

KafkaProducerMessageHandlerSpec messageHandlerSpec =

Kafka.outboundChannelAdapter(

props -> props.put("queue.buffering.max.ms", "15000"))

.messageKey(m -> m.getHeaders().get(IntegrationMessageHeaderAccessor.SEQUENCE\_NUMBER))

.addProducer(this.kafkaConfig.getTopic(),

this.kafkaConfig.getBrokerAddress(), spec);

flowDefinition

.handle(messageHandlerSpec);

};

}

}

@Configuration

public static class ConsumerConfiguration {

@Autowired

private KafkaConfig kafkaConfig;

private Log log = LogFactory.getLog(getClass());

@Bean

IntegrationFlow consumer() {

log.info("starting consumer..");

KafkaHighLevelConsumerMessageSourceSpec messageSourceSpec = Kafka.inboundChannelAdapter(

new ZookeeperConnect(this.kafkaConfig.getZookeeperAddress()))

.consumerProperties(props ->

props.put("auto.offset.reset", "smallest")

.put("auto.commit.interval.ms", "100"))

.addConsumer("myGroup", metadata -> metadata.consumerTimeout(100)

.topicStreamMap(m -> m.put(this.kafkaConfig.getTopic(), 1))

.maxMessages(10)

.valueDecoder(String::new));

Consumer<SourcePollingChannelAdapterSpec> endpointConfigurer = e -> e.poller(p -> p.fixedDelay(100));

return IntegrationFlows

.from(messageSourceSpec, endpointConfigurer)

.<Map<String, List<String>>>handle((payload, headers) -> {

payload.entrySet().forEach(e -> log.info(e.getKey() + '=' + e.getValue()));

return null;

})

.get();

}

}

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

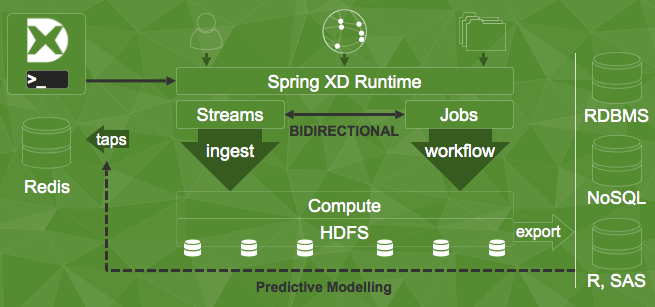
The example makes heavy use of Java 8 lambdas.

The producer spends a bit of time establishing how many messages will be sent in a single send operation, how keys and values are encoded (Kafka only knows about byte[] arrays, after all) and whether messages should be sent synchronously or asynchronously. In the next line, we configure the outbound adapter itself and then define an IntegrationFlow such that all messages get sent out via the Kafka outbound adapter.

The consumer spends a bit of time establishing which Zookeeper instance to connect to, how many messages to receive (10) in a batch, etc. Once the message batches are recieved, they’re handed to the handle method where I’ve passed in a lambda that’ll enumerate the payload’s body and print it out. Nothing fancy.

**Using Apache Kafka with Spring XD**

Apache Kafka is a message bus and it can be very powerful when used as an integration bus. However, it really comes into its own because it’s fast enough and scalable enough that it can be used to route big-data through processing pipelines. And if you’re doing data processing, you really want [Spring XD](http://projects.spring.io/spring-xd/)! Spring XD makes it dead simple to use Apache Kafka (as the support is built on the Apache Kafka Spring Integration adapter!) in complex stream-processing pipelines. Apache Kafka is exposed as a Spring XD *source* - where data comes from - and a sink - where data goes to.



Spring XD exposes a super convenient DSL for creating bash-like pipes-and-filter flows. Spring XD is a centralized runtime that manages, scales, and monitors data processing jobs. It builds on top of Spring Integration, Spring Batch, Spring Data and Spring for Hadoop to be a one-stop data-processing shop. Spring XD Jobs read data from *sources*, run them through processing components that may count, filter, enrich or transform the data, and then write them to sinks.

Spring Integration and Spring XD ninja [Marius Bogoevici](https://twitter.com/mariusbogoevici), who did a lot of the recent work in the Spring Integration and Spring XD implementation of Apache Kafka, put together a really nice example demonstrating [how to get a full working Spring XD and Kafka flow working](https://github.com/spring-projects/spring-xd-samples/tree/master/kafka-source). TheREADME walks you through getting Apache Kafka, Spring XD and the requisite topics all setup. The essence, however, is when you use the Spring XD shell and the shell DSL to compose a stream. Spring XD components are named components that are pre-configured but have lots of parameters that you can override with --.. arguments via the XD shell and DSL. (That DSL, by the way, is written by the amazing [Andy Clement](https://spring.io/team/aclement) of Spring Expression language fame!) Here’s an example that configures a stream to read data from an Apache Kafka source and then write the message a component called log, which is a sink. log, in this case, could be syslogd, Splunk, HDFS, etc.

xd> stream create kafka-source-test --definition "kafka --zkconnect=localhost:2181 --topic=event-stream | log" --deploy

And that’s it! Naturally, this is just a tase of Spring XD, but hopefully you’ll agree the possibilities are tantalizing.

**Deploying a Kafka Server with Lattice and Docker**

It’s easy to get an example Kafka installation all setup using [Lattice](http://lattice.cf/), a distributed runtime that supports, among other container formats, the very popular Docker image format. [There’s a Docker image provided by Spotify that sets up a collocated Zookeeper and Kafka image](https://github.com/spotify/docker-kafka). You can easily deploy this to a Lattice cluster, as follows:

ltc create --run-as-root m-kafka spotify/kafka

From there, you can easily scale the Apache Kafka instances and even more easily still consume Apache Kafka from your cloud-based services.

**Next Steps**

You can find the code [for this blog on my GitHub account](https://github.com/joshlong/spring-and-kafka).

We’ve only scratched the surface!

If you want to learn more (and why wouldn’t you?), then be sure to check out Marius Bogoevici and Dr. Mark Pollack’s upcoming [webinar on Reactive data-pipelines using Spring XD and Apache Kafka](https://spring.io/blog/2015/03/17/webinar-reactive-data-pipelines-with-spring-xd-and-kafka) where they’ll demonstrate how easy it can be to use RxJava, Spring XD and Apache Kafka!