Stream Processing with Apache Kafka

In this guide, we develop three Spring Boot applications that use Spring Cloud Stream's support for Apache Kafka and deploy them to Cloud Foundry, Kubernetes, and your local machine. In another guide, we deploy these applications by using Spring Cloud Data Flow. By deploying the applications manually, you get a better understanding of the steps that Data Flow can automate for you.

The following sections describe how to build these applications from scratch. If you prefer, you can download a zip file that contains the sources for these applications, unzip it, and proceed to the deployment section.

You can download a zip file containing the completed application that contains all three applications from your browser. You can also download the zip file from the command line by using the following command:

```
wget https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/
```

Development

We create three Spring Cloud Stream applications that communicate using Kafka.

The scenario is a cell phone company creating bills for its customers. Each call made by a user has a `duration` and an amount of `data` used during the call. As part of the process to generate a bill, the raw call data needs to be converted to a cost for the duration of the call and a cost for the amount of data used.

The call is modeled by using the `UsageDetail` class, which contains the `duration` of the call and the amount of `data` used during the call. The bill is modeled by using the `UsageCostDetail` class, which contains the cost of the call (`costCall`) and the cost of the data (`costData`). Each class contains an ID (`userId`) to identify the person making the call.

The three streaming applications are as follows:

- The `Source` application (named `UsageDetailSender`) generates the user's call `duration` and amount of `data` used per `userId` and sends a message containing the `UsageDetail` object as JSON.
- The `Processor` application (named `UsageCostProcessor`) consumes the `UsageDetail` and computes the cost of the call and the cost of the data per `userId`. It sends the `UsageCostDetail` object as JSON.
- The `Sink` application (named `UsageCostLogger`) consumes the `UsageCostDetail` object and logs the cost of the call and the cost of the data.

UsageDetailSender source

Either download the initialzr generated project directly or visit the Spring Initialzr site and follow these instructions:

1. Create a new Maven project with a Group name of `io.spring.dataflow.sample` and an Artifact name of `usage-detail-sender-kafka`.
2. In the **Dependencies** text box, type `Kafka` to select the Kafka binder dependency.
3. In the **Dependencies** text box, type `Cloud Stream` to select the Spring Cloud Stream dependency.

4. In the **Dependencies** text box, type `Actuator` to select the Spring Boot actuator dependency.
5. If your target platform is `Cloud Foundry`, type `Cloud Connectors` to select the Spring Cloud Connector dependency.
6. Click the **Generate Project** button.

Now you should `unzip` the `usage-detail-sender-kafka.zip` file and import the project into your favorite IDE.

Business Logic

Now we can create the code required for this application. To do so:

1. Create a `UsageDetail` class in the `io.spring.dataflow.sample.usagedetailsender` package with content that resembles UsageDetail.java. This `UsageDetail` model contains `userId`, `data`, and `duration` properties.
2. Create the `UsageDetailSender` class in the `io.spring.dataflow.sample.usagedetailsender` package with content that resembles the following:

```
package io.spring.dataflow.sample.usagedetailsender;

import java.util.Random;
import java.util.function.Supplier;

import io.spring.dataflow.sample.UsageDetail;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class UsageDetailSender {

    private String[] users = {"user1", "user2", "user3", "user4", "user5"};

    @Bean
    public Supplier<UsageDetail> sendEvents() {
        return () -> {
            UsageDetail usageDetail = new UsageDetail();
```

```
        usageDetail.setUserId(this.users[new Random().nextInt(5)]);
        usageDetail.setDuration(new Random().nextInt(300));
        usageDetail.setData(new Random().nextInt(700));
```

This is a simple `Configuration` class with a single bean that returns a `java.util.function.Supplier`. Spring Cloud Stream, behind the scenes will turn this `Supplier` into a producer. By default, the supplier will be invoked every second. On each invocation, the supplier method `sendEvents` constructs a `UsageDetail` object.

Configuring the UsageDetailSender application

When configuring the `producer` application, we need to set the producer binding destination (Kafka topic) where the producer publishes the data. The default producer output binding for the above method is going to be `sendEvents-out-0` (method name followed by the literal `-out-0` where `0` is the index). If the application does not set a destination, Spring Cloud Stream will use this same binding name as the output destination (Kafka topic). However, in our case, we neither want this default binding name used by Spring Cloud Stream nor the destination name. We want to use the binding name as `output` and provide a custom destination.

In `src/main/resources/application.properties`, you can add the following properties to override:

```
spring.cloud.stream.function.bindings.sendEvents-out-0=output
spring.cloud.stream.bindings.output.destination=usage-detail
```

The first property will override the default binding name to `output` and the second one will set destination on that binding.

Building

Now we can build the Usage Detail Sender application. In the `usage-detail-sender` directory, use the following command to build the project using maven:

```
./mvnw clean package
```

Testing

Spring Cloud Stream provides a test binder to test an application. Following are the maven coordinates for this artifact.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
    <type>test-jar</type>
    <classifier>test-binder</classifier>
    <scope>test</scope>
</dependency>
```

Instead of the `Kafka` binder, the tests use the `Test` binder to trace and test your application's outbound and inbound messages. The `Test` binder provides abstractions for output and input destinations as `OutputDestination` and `InputDestination`. Using them, you can simulate the behavior of actual middleware based binders.

To unit test this `UsageDetailSender` application, add the following code in the `UsageDetailSenderApplicationTests` class:

```
package io.spring.dataflow.sample.usagedetailsender;

import io.spring.dataflow.sample.UsageDetail;
import org.junit.jupiter.api.Test;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.cloud.stream.binder.test.OutputDestination;
import org.springframework.cloud.stream.binder.test.TestChannelBinderConfigurat
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.messaging.Message;
import org.springframework.messaging.converter.CompositeMessageConverter;
import org.springframework.messaging.converter.MessageConverter;

import static org.assertj.core.api.Assertions.assertThat;

public class UsageDetailSenderApplicationTests {
```

```
    @Test
    public void contextLoads() {
    }
```

- The `contextLoads` test case verifies the application starts successfully.
- The `testUsageDetailSender` test case uses the test binder to receive messages from the output destination where the supplier publishes messages to.

`UsageCostProcessor` Processor

Either download the initialzr generated project directly or visit the Spring Initialzr site and follow these instructions:

1. Create a new Maven project with a Group name of `io.spring.dataflow.sample` and an Artifact name of `usage-cost-processor-kafka`.
2. In the **Dependencies** text box, type `kafka` to select the Kafka binder dependency.
3. In the **Dependencies** text box, type `cloud stream` to select the Spring Cloud Stream dependency.
4. In the **Dependencies** text box, type `Actuator` to select the Spring Boot actuator dependency.
5. If your target platform is `Cloud Foundry`, type `Cloud Connectors` to select the Spring Cloud Connector dependency.
6. Click the **Generate Project** button.

Now you should `unzip` the `usage-cost-processor-kafka.zip` file and import the project into your favorite IDE.

Business Logic

Now we can create the code required for this application.

1. Create the `UsageDetail` class in the `io.spring.dataflow.sample.usagecostprocessor` with content that resembles UsageDetail.java. The `UsageDetail` class contains `userId`, `data` and, `duration` properties.

2. Create the `UsageCostDetail` class in the `io.spring.dataflow.sample.usagecostprocessor` package with content that resembles UsageCostDetail.java. This `UsageCostDetail` class contains `userId`, `callCost`, and `dataCost` properties.
3. Create the `UsageCostProcessor` class in the `io.spring.dataflow.sample.usagecostprocessor` package that receives the `UsageDetail` message, computes the call and data cost and sends a `UsageCostDetail` message. The following listing shows the source code:

```
package io.spring.dataflow.sample.usagecostprocessor;

import java.util.function.Function;

import io.spring.dataflow.sample.UsageCostDetail;
import io.spring.dataflow.sample.UsageDetail;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class UsageCostProcessor {

    private double ratePerSecond = 0.1;

    private double ratePerMB = 0.05;

    @Bean
    public Function<UsageDetail, UsageCostDetail> processUsageCost() {
        return usageDetail -> {
            UsageCostDetail usageCostDetail = new UsageCostDetail();
            usageCostDetail.setUserId(usageDetail.getUserId());
```

In the preceding application, we are providing a bean that returns a `java.util.function.Function` that consumes a `UsageDetail` as input and publishes a `UsageCostDetail` as ouptut.

Configuring the `UsageCostProcessor` Application

When configuring this `processor` application, we need to set both the input and output destinations (Kafka topics). By default, Sprig Cloud Stream uses binding names as `processUsageCost-in-0` and `processUsageCost-out-0` which becomes the topic names unless the application overrides them. However, in our case, as in the producer above, we don't want these defaults but rather we would want to make them more descriptive. We want to use the binding name as `input` and `output` and provide custom destinations on them.

In `src/main/resources/application.properties`, you can add the following properties:

```
spring.cloud.stream.function.bindings.processUsageCost-in-0=input
spring.cloud.stream.function.bindings.processUsageCost-out-0=output
spring.cloud.stream.bindings.input.destination=usage-detail
spring.cloud.stream.bindings.output.destination=usage-cost
```

1. The `spring.cloud.stream.function.bindings.processUsageCost-in-0` overrides the binding name to `input`.
2. The `spring.cloud.stream.function.bindings.processUsageCost-out-0` overrides the binding name to `output`.
3. The `spring.cloud.stream.bindings.processUsageCost-in-0.destination` sets the destination to the `usage-detail` Kafka topic.
4. The `spring.cloud.stream.bindings.processUsageCost-out-0.destination` sets the destination to the `usage-cost` Kafka topic.

Building

Now we can build the Usage Cost Processor application. In the `usage-cost-processor` directory, use the following command to build the project with Maven:

```
./mvnw clean package
```

Testing

We can use the same test binder that we used above for testing the supplier.

To unit test the `UsageCostProcessor`, add the following code in the `UsageCostProcessorApplicationTests` class:

```
package io.spring.dataflow.sample.usagecostprocessor;

import java.util.HashMap;
import java.util.Map;

import io.spring.dataflow.sample.UsageCostDetail;
import io.spring.dataflow.sample.UsageDetail;
import org.junit.jupiter.api.Test;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.cloud.stream.binder.test.InputDestination;
import org.springframework.cloud.stream.binder.test.OutputDestination;
import org.springframework.cloud.stream.binder.test.TestChannelBinderConfigurat
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.converter.CompositeMessageConverter;
import org.springframework.messaging.converter.MessageConverter;

import static org.assertj.core.api.Assertions.assertThat;
```

- The `contextLoads` test case verifies the application starts successfully.
- The `testUsageCostProcessor` test case uses the test binder's `InputDestination` to publish a message which is consumed by the function in the processor. Then we use the `OutputDestination` to verify that the `UsageDetail` is property transformed into a `UsageCostDetail`.

`UsageCostLogger` Sink

Either download the initialzr generated project directly or visit the Spring Initialzr site and follow these instructions:

1. Create a new Maven project with a Group name of `io.spring.dataflow.sample` and an Artifact name of `usage-cost-logger-kafka`.

2. In the **Dependencies** text box, type `kafka` to select the Kafka binder dependency.
3. In the **Dependencies** text box, type `cloud stream` to select the Spring Cloud Stream dependency.
4. In the **Dependencies** text box, type `Actuator` to select the Spring Boot actuator dependency.
5. If your target platform is `Cloud Foundry`, type `Cloud Connectors` to select the Spring Cloud Connector dependency.
6. Click the **Generate Project** button.

Now you should `unzip` the `usage-cost-logger-kafka.zip` file and import the project into your favorite IDE.

Business Logic

Now we can create the business logic for the sink application. To do so:

1. Create a `UsageCostDetail` class in the `io.spring.dataflow.sample.usagecostlogger` package with content that resembles UsageCostDetail.java. The `UsageCostDetail` class contains `userId`, `callCost`, and `dataCost` properties.
2. Create the `UsageCostLogger` class in the `io.spring.dataflow.sample.usagecostlogger` package to receive the `UsageCostDetail` message and log it. The following listing shows the source code:

```
package io.spring.dataflow.sample.usagecostlogger;

import java.util.function.Consumer;

import io.spring.dataflow.sample.UsageCostDetail;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class UsageCostLogger {
```

```
    private static final Logger logger = LoggerFactory.getLogger(UsageCostLogge

    @Bean
    public Consumer<UsageCostDetail> process() {
        return usageCostDetail -> {
            logger.info(usageCostDetail.toString());
        };
    }
```

Here we have a `java.util.function.Consumer` bean that consumes a `UsageCostDetail` and then logs that information.

Configuring the `UsageCostLogger` Application

When configuring the `consumer` application, we need to set the `input` binding destination (a Kafka topic). By default, the input binding used by Spring Cloud Stream will be `process-in-0` (so does the destination name if the application does not override it). We want to override these to make the sink application work with the above two applications (source and processor).

In `src/main/resources/application.properties`, you can add them:

```
spring.cloud.stream.function.bindings.process-in-0=input
spring.cloud.stream.bindings.input.destination=usage-cost
```

The `spring.cloud.stream.function.bindings.process-in-0` property overrides the binding name to `input` and spring.cloud.stream.bindings.input.destination `property sets the destination to the` usage-cost` Kafka topic.

There are many configuration options that you can choose to extend/override to achieve the desired runtime behavior when using Apache Kafka as the message broker. The Apache Kafka-specific binder configuration properties are listed in Apache Kafka-binder documentation

Building

Now we can build the Usage Cost Logger application. In the `usage-cost-logger` directory, run the following command to build the project with Maven:

```
./mvnw clean package
```

Testing

To unit test the `UsageCostLogger`, add the following code in the `UsageCostLoggerApplicationTests` class:

```
package io.spring.dataflow.sample.usagecostlogger;

import java.util.HashMap;
import java.util.Map;

import io.spring.dataflow.sample.UsageCostDetail;
import org.awaitility.Awaitility;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.test.system.CapturedOutput;
import org.springframework.boot.test.system.OutputCaptureExtension;
import org.springframework.cloud.stream.binder.test.InputDestination;
import org.springframework.cloud.stream.binder.test.TestChannelBinderConfigurat
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.converter.CompositeMessageConverter;
import org.springframework.messaging.converter.MessageConverter;
```

- The `contextLoads` test case verifies the application starts successfully.
- The `testUsageCostLogger` test case verifies that the `process` method of `UsageCostLogger` is invoked. We use the `OutputCaptureExtension` facility provided by Spring Boot testing infrastructure to verify that the message is logged to the console.

Deployment

In this section, we deploy the applications we created earlier to the local machine, to Cloud Foundry, and to Kubernetes.

When you deploy these three applications (`UsageDetailSender`, `UsageCostProcessor` and `UsageCostLogger`), the flow of message is as follows:

```
UsageDetailSender -> UsageCostProcessor -> UsageCostLogger
```

The `UsageDetailSender` source application's output is connected to the `UsageCostProcessor` processor application's input. The `UsageCostProcessor` application's output is connected to the `UsageCostLogger` sink application's input.

When these applications run, the `Kafka` binder binds the applications' output and input boundaries to the corresponding topics in Kafka.

Local

This section shows how to run the three applications as standalone applications in your `local` environment.

If you have not already done so, you must download and set up `Kafka` in your local environment.

After unpacking the downloaded archive, you can start the `ZooKeeper` and `Kafka` servers by running the following commands:

```
./bin/zookeeper-server-start.sh config/zookeeper.properties &
```

```
./bin/kafka-server-start.sh config/server.properties &
```

Running the Source

By using the pre-defined configuration properties (along with a unique server port) for `UsageDetailSender`, you can run the application, as follows:

```
java -jar target/usage-detail-sender-kafka-0.0.1-SNAPSHOT.jar --server.port=900
```

Now you can see the messages being sent to the `usage-detail` Kafka topic by using the Kafka console consumer, as follows:

```
./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic usage
```

To list the topics, run the following command:

```
./bin/kafka-topics.sh --zookeeper localhost:2181 --list
```

Running the Processor

By using the pre-defined configuration properties(along with a unique server port) for `UsageCostProcessor`, you can run the application, as follows:

```
java -jar target/usage-cost-processor-kafka-0.0.1-SNAPSHOT.jar --server.port=90
```

With the `UsageDetail` data in the `usage-detail` Kafka topic from the `UsageDetailSender` source application, you can see the `UsageCostDetail` from the `usage-cost` Kafka topic, as follows:

```
 ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic usag
```

Running the Sink

By using the pre-defined configuration properties (along with a unique server port) for `UsageCostLogger`, you can run the application, as follows:

```
java -jar target/usage-cost-logger-kafka-0.0.1-SNAPSHOT.jar --server.port=9003
```

Now you can see that this application logs the usage cost detail.

Cloud Foundry

This section walks you through how to deploy the `UsageDetailSender`, `UsageCostProcessor`, and `UsageCostLogger` applications on CloudFoundry.

Create a CF Manifest for the `UsageDetail` Sender

You need to create a CF manifest YAML file called `usage-detail-sender.yml` for the `UsageDetailSender` to define its configuration properties, as follows

```
applications:
- name: usage-detail-sender
  timeout: 120
  path: ./target/usage-detail-sender-kafka-0.0.1-SNAPSHOT.jar
  memory: 1G
  buildpack: java_buildpack
  env:
    SPRING_CLOUD_STREAM_KAFKA_BINDER_BROKERS: [Kafka_Service_IP_Address:Kafka_S
    SPRING_CLOUD_STREAM_KAFKA_BINDER_ZKNODES: [ZooKeeper_Service_IP_Address:Zoo
```

Then you need to push the `UsageDetailSender` application by using its manifest YAML file, as follows:

```
cf push -f usage-detail-sender.yml
```

You need to create a CF manifest YAML file called `usage-cost-processor.yml` for the `UsageCostProcessor` to define its configuration properties, as follows

```
applications:
- name: usage-cost-processor
  timeout: 120
  path: ./target/usage-cost-processor-kafka-0.0.1-SNAPSHOT.jar
```

```
  memory: 1G
  buildpack: java_buildpack
  env:
    SPRING_CLOUD_STREAM_KAFKA_BINDER_BROKERS: [Kafka_Service_IP_Address:Kafka_S
    SPRING_CLOUD_STREAM_KAFKA_BINDER_ZKNODES: [ZooKeeper_Service_IP_Address:Zoo
```

Then you need to push the `UsageCostProcessor` application by using its manifest YAML file, as follows:

```
cf push -f usage-cost-processor.yml
```

You need to create a CF manifest YAML file called `usage-cost-logger.yml` for the `UsageCostLogger` to define its configuration properties, as follows:

```
applications:
- name: usage-cost-logger
  timeout: 120
  path: ./target/usage-cost-logger-kafka-0.0.1-SNAPSHOT.jar
  memory: 1G
  buildpack: java_buildpack
  env:
    SPRING_CLOUD_STREAM_KAFKA_BINDER_BROKERS: [Kafka_Service_IP_Address:Kafka_S
    SPRING_CLOUD_STREAM_KAFKA_BINDER_ZKNODES: [ZooKeeper_Service_IP_Address:Zoo
```

Then you need to push the `UsageCostLogger` application by using its manifest YAML file, as follows:

```
cf push -f usage-cost-logger.yml
```

You can see the applications by running the `cf apps` command, as the folowing example (with output) shows:

```
cf apps
```

```
name                    requested state   instances   memory   disk   urls
usage-cost-logger       started           1/1         1G       1G     usage-cost
usage-cost-processor    started           1/1         1G       1G     usage-cost
usage-detail-sender     started           1/1         1G       1G     usage-deta
```

```
   2019-05-13T23:23:33.36+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:33.362  IN
   2019-05-13T23:23:33.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:33.467  IN
   2019-05-13T23:23:34.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:34.466  IN
   2019-05-13T23:23:35.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:35.469  IN
```

Kubernetes

This section walks you through how to deploy the three Spring Cloud Stream applications on Kubernetes.

Setting up the Kubernetes Cluster

For this we need a running Kubernetes cluster. For this example we will deploy to `minikube`.

Verifying Minikube is Running

To verify that Minikube is running, run the following command (shown with typical output if Minikube is running):

```
$minikube status

host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.99.100
```

Installing Apache Kafka

Now we can install the Kafka message broker by using the default configuration from Spring Cloud Data Flow. To do so, run the following command:

```
kubectl apply -f https://raw.githubusercontent.com/spring-cloud/spring-cloud-da
-f https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v2.7.1/
-f https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v2.7.1/
-f https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v2.7.1/
```

Building Docker Images

To build Docker images, we use the jib Maven plugin. If you downloaded the source distribution, the jib plugin is already configured. If you built the apps from scratch, add the following under `plugins` in each `pom.xml` file:

```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>0.10.1</version>
    <configuration>
        <from>
            <image>springcloud/openjdk</image>
        </from>
        <to>
            <image>${docker.org}/${project.artifactId}:${docker.version}</image
        </to>
        <container>
            <useCurrentTimestamp>true</useCurrentTimestamp>
        </container>
    </configuration>
</plugin>
```

Then add the following properties under the `properties` section of each `pom.xml` file. For this example, we use the following properties:

```
<docker.org>springcloudstream</docker.org>
<docker.version>${project.version}</docker.version>
```

Now you can run the Maven build to create the Docker images in the `minikube` Docker registry. To do so, run the following commands:

```
$ eval $(minikube docker-env)
$./mvnw package jib:dockerBuild
```

> If you downloaded the project source, the project includes a parent pom to let you build all the modules with a single command.

Otherwise, run the build for the source, processor, and sink individually. You need only run `eval $(minikube docker-env)` once for each terminal session.

Deploying the Stream

To deploy the stream, you must first copy and paste the following YAML and save it to `usage-cost-stream.yaml`:

```
kind: Pod
apiVersion: v1
metadata:
  name: usage-detail-sender
  labels:
    app: usage-cost-stream
spec:
  containers:
    - name: usage-detail-sender
      image: springcloudstream/usage-detail-sender-kafka:0.0.1-SNAPSHOT
      ports:
        - containerPort: 80
```

```
      protocol: TCP
    env:
      - name: SPRING_CLOUD_STREAM_KAFKA_BINDER_BROKERS
        value: kafka
      - name: SPRING_CLOUD_STREAM_BINDINGS_OUTPUT_DESTINATION
        value: user-details
      - name: SERVER_PORT
        value: '80'
  restartPolicy: Always
```

Then you need to deploy the apps, by running the following command:

```
kubectl apply -f usage-cost-stream.yaml
```

If all is well, you should see the following output:

```
pod/usage-detail-sender created
pod/usage-cost-processor created
pod/usage-cost-logger created
```

The preceding YAML specifies three pod resources, for the source, processor, and sink applications. Each pod has a single container that references the corresponding docker image.

We set the Kafka binding parameters as environment variables. The input and output destination names have to be correct to wire the stream. Specifically, the output of the source must be the same as the input of the processor, and the output of the processor must be the same as the input of the sink. We also set the logical hostname for the Kafka broker so that each application can connect to it. Here we use the Kafka service name — `kafka`, in this case. We set the `app: user-cost-stream` label to logically group our apps.

We set the Spring Cloud Stream binding parameters by using environment variables. The input and output destination names have to be correct to wire the stream. Specifically, the output of the source must be the same as the input of the processor, and the output of the processor must be the same as the input of the sink. We set the inputs and outputs as follows:

- Usage Detail Sender:
  `SPRING_CLOUD_STREAM_BINDINGS_OUTPUT_DESTINATION=user-details`

- Usage Cost Processor:
  `SPRING_CLOUD_STREAM_BINDINGS_INPUT_DESTINATION=user-details` and
  `SPRING_CLOUD_STREAM_BINDINGS_OUTPUT_DESTINATION=user-cost`
- Usage Cost Logger: `SPRING_CLOUD_STREAM_BINDINGS_INPUT_DESTINATION=user-cost`

Verifying the Deployment

You can use the following command to tail the log for the `usage-cost-logger` sink:

```
kubectl logs -f usage-cost-logger
```

You should see messages similar to the following messages:

```
2019-05-02 15:48:18.550  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:19.553  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:20.549  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:21.553  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:22.551  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:23.556  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:24.557  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:25.555  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:26.557  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:27.556  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:28.559  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:29.562  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:30.561  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:31.562  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:32.564  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:33.567  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
2019-05-02 15:48:34.567  INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLogger
```

Cleaning up

To delete the stream, we can use the label we created earlier. The following command shows how to do so:

```
kubectl delete pod -l app=usage-cost-stream
```

To uninstall Kafka, run the following command:

```
kubectl delete all -l app=kafka
```

What's Next

The RabiitMQ shows you how to create the same three applications but with RabbitMQ instead. Alternatively, you can use Spring Cloud Data Flow to deploy the three applications, as described in Create and Deploy a Stream Processing Pipeline using Spring Cloud Data Flow.

Edit this page on GitHub

Documentation
Stream Application Development on RabbitMQ

Documentation
Stream Processing using Spring Cloud Data Flow