# Getting Started with Stream Processing

Spring Cloud Data Flow provides over 70 prebuilt streaming applications that you can use right away to implement common streaming use cases. In this guide, we use two of these applications to construct a simple data pipeline that produces data sent from an external HTTP request and consumes that data by logging the payload to the terminal.

Instructions for registering these prebuilt applications with Data Flow are provided in the [Installation guide](https://dataflow.spring.io/docs/installation/).

## Stream DSL overview

You can create streams by using a Domain Specific Language (DSL) through the shell or the dashboard as well as programmatically in Java. The dashboard also lets you drag and drop applications onto a palate and connect them visually. The dashboard is bi-directional, so visual actions update the DSL and edits to the DSL update the view of the stream.

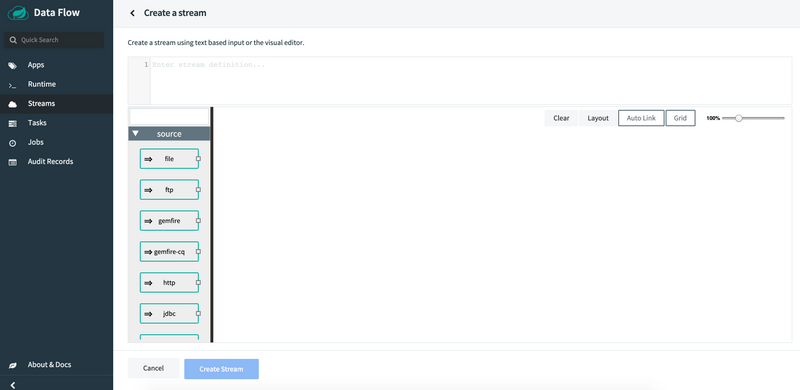
The DSL is modeled after the Unix pipes and filter syntax. As an example, a stream DSL defined as http | log represents an http application sending the data it received from a HTTP post to the messaging middleware.  
The log application receives the message with that data from the messaging middleware and logs it to the terminal. Each name in the DSL is associated with an application through the application registration process. The applications are connected through a | symbol that represents the messaging middleware that acts as the 'pipe' between the applications.

## Creating the Stream

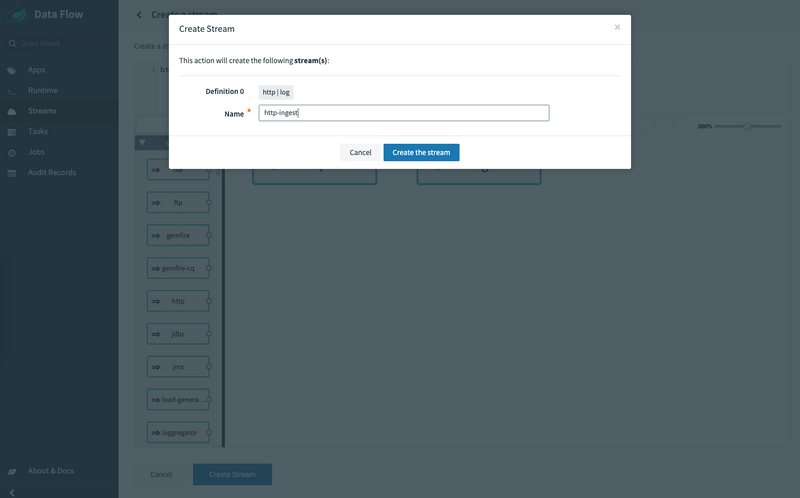
To create a stream:

1. In the menu, click **Streams**.
2. Click the **Create Stream(s)** button.

The screen changes to the following image:

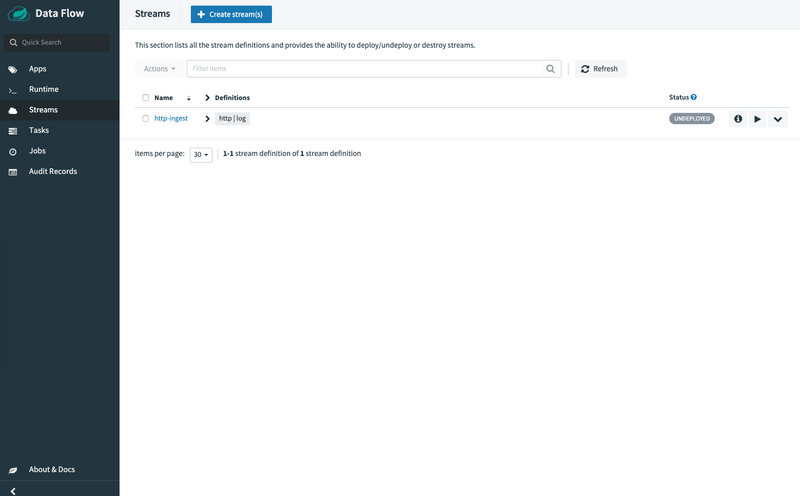


1. In the text area, type http | log.
2. Click **Create Stream**.
3. Enter http-ingest for the stream name, as shown in the following image:



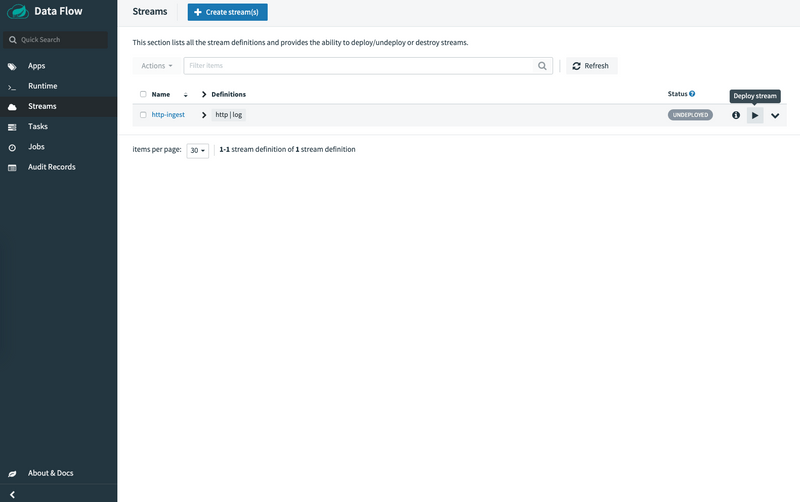
1. Click the **Create the stream** button.

The Definitions page appears.

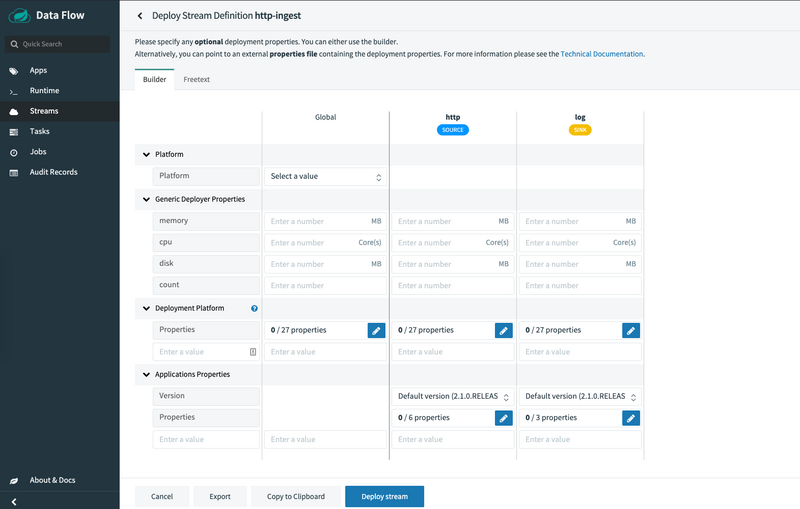


## Deploying a Stream

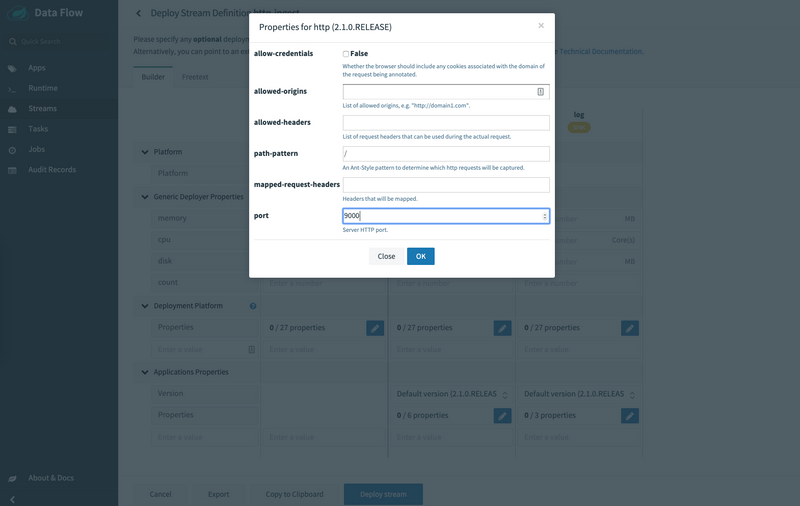
Now that you have defined a stream, you can deploy it. To do so:

1. Click the play (deploy) button next to the "http-ingest" definition that you created in the previous section.

The UI shows the available properties that can be applied to the apps in the "http-ingest" stream. This example shown in the following image uses the defaults:



If you use the local Data Flow Server, add the following deployment property to set the port to avoid a port collision.



1. Click the **Deploy Stream** button.

The UI returns to the Definitions page.

The stream is now in "deploying" status, and its status becomes "deployed" when it has finished deploying. You may need to refresh your browser to see the updated status.

## Verifying Output

Once your application is deployed, you can verify its output. How to do so depends on where you run your application:

* [Local](https://dataflow.spring.io/docs/stream-developer-guides/getting-started/stream/#local)
* [Cloud Foundry](https://dataflow.spring.io/docs/stream-developer-guides/getting-started/stream/#cloud-foundry)
* [Kubernetes](https://dataflow.spring.io/docs/stream-developer-guides/getting-started/stream/#kubernetes)

### Local

This section details how to verify output when your application runs on a local server.

#### Test Data

Once the stream is deployed and running, you can now post some data. You can use the following curl command to do so:

**curl** http://localhost:9000 -H "Content-type: text/plain" -d "Happy streaming"

#### Results

Once a stream is deployed, you can view its logs. To do so:

1. Click **Runtime** in the menu.
2. Click "http-ingest.log".
3. the path in the "stdout" text box on the dashboard
4. In another console window, type the following, replacing /path/from/stdout/textbox/in/dashboard with the value you copied in the previous step:

$ docker exec -it skipper tail -f /path/from/stdout/textbox/in/dashboard

The output of the log sink appears in the new window. You will see output as shown below. When you have seen enough output from sending http requests, press Ctrl+C to end the tail command.

log-sink : Happy streaming

### Cloud Foundry

This section details how to verify output when your application runs on Cloud Foundry.

#### Test Data

Once the stream is deployed and running in Cloud Foundry, you can now post some data. You can use the following curl command to do so:

**curl** http://http-ingest-314-log-v1.cfapps.io -H "Content-type: text/plain" -d "Happy streaming"

#### Results

Now you can list the running applications again and see your applications in the list, as the following example shows:

$ cf apps [1h] ✭

Getting apps in org ORG / space SPACE as email@pivotal.io...

name requested state instances memory disk urls

http-ingest-314-log-v1 started 1/1 1G 1G http-ingest-314-log-v1.cfapps.io

http-ingest-314-http-v1 started 1/1 1G 1G http-ingest-314-http-v1.cfapps.io

skipper-server started 1/1 1G 1G skipper-server.cfapps.io

dataflow-server started 1/1 1G 1G dataflow-server.cfapps.io

Now you can verify the logs, as the following example shows:

cf logs http-ingest-314-log-v1

...

...

2017-11-20T15:39:43.76-0800 [APP/PROC/WEB/0] OUT 2017-11-20 23:39:43.761 INFO 12 --- [ http-ingest-314.ingest-314-1] log-sink : Happy streaming

### Kubernetes

This section details how to verify output when your application runs on Kubernetes.

#### Test Data

Once the stream is deployed and running in Kubernetes, you can now post some data. You can use the following curl command to do so:

**curl** http://<EXTERNAL\_IP\_OF\_http-ingest-log-v1-0-2k4r8\_SERVICE> -H "Content-type: text/plain" -d "Happy streaming"

#### Results

kubectl get pods

NAME READY STATUS RESTARTS AGE

http-ingest-log-v1-0-2k4r8 1/1 Running 0 2m

http-ingest-http-v1-qhdqq 1/1 Running 0 2m

mysql-777890292-z0dsw 1/1 Running 0 49m

rabbitmq-317767540-2qzrr 1/1 Running 0 49m

scdf-server-2734071167-bjd3g 1/1 Running 0 12m

skipper-2408247821-50z31 1/1 Running 0 15m

Now you can verify the logs.

kubectl logs -f http-ingest-log-v1-0-2k4r8

...

...

2017-10-30 22:59:04.966 INFO 1 --- [ http-ingest.http.http-ingest-1] log-sink : Happy streaming

## Deleting a Stream

Now you can delete the stream you created. To do so:

1. Click **Streams** in the menu.
2. Click the down chevron on the "http-ingest" row.
3. Click **Destroy Stream**.
4. When prompted for confirmation, click **Destroy Stream Definition(s)**.

## Updating and Rolling back a Stream

You can find this information in the [Continuous Delivery](https://dataflow.spring.io/docs/stream-developer-guides/continuous-delivery/) guide.

## Monitoring

You can find this information in the [Stream Monitoring](https://dataflow.spring.io/docs/feature-guides/streams/monitoring/) guide.

# 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](https://dataflow.spring.io/docs/stream-developer-guides/streams/data-flow-stream/). 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](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-kafka/#deployment) section.

You can [download a zip file containing the completed application](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-kafka/dist/usage-cost-stream-kafka.zip?raw=true) 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/dataflow-website/stream-developer-guides/streams/standalone-stream-kafka/dist/usage-cost-stream-kafka.zip?raw=true -O usage-cost-stream-kafka.zip

## 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](https://start.spring.io/starter.zip?type=maven-project&language=java&bootVersion=2.1.4.RELEASE&baseDir=usage-detail-sender-kafka&groupId=io.spring.dataflow.sample&artifactId=usage-detail-sender-kafka&name=usage-detail-sender-kafka&description=Demo+project+for+Spring+Boot&packageName=io.spring.dataflow.sample.usagedetailsender&packaging=jar&javaVersion=1.8&dependencies=kafka&dependencies=cloud-stream&dependencies=actuator&dependencies=web&dependencies=cloud-connectors) or visit the [Spring Initialzr site](https://start.spring.io/) 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](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-kafka/usage-detail-sender/src/main/java/io/spring/dataflow/sample/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 io.spring.dataflow.sample.UsageDetail;

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

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.messaging.Source;

import org.springframework.messaging.support.MessageBuilder;

import org.springframework.scheduling.annotation.EnableScheduling;

import org.springframework.scheduling.annotation.Scheduled;

@EnableScheduling

@EnableBinding(Source.class)

public class UsageDetailSender {

@Autowired

private Source source;

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

@Scheduled(fixedDelay = 1000)

public void **sendEvents**() {

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.source.**output**().**send**(MessageBuilder.**withPayload**(usageDetail).**build**());

}

}

The @EnableBinding annotation indicates that you want to bind your application to the messaging middleware. The annotation takes one or more interfaces as a parameter — in this case, the [Source](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream/src/main/java/org/springframework/cloud/stream/messaging/Source.java) interface that defines an output channel named output. In the case of Kafka, messages sent to the output channel are, in turn, sent the Kafka topic.

The @EnableScheduling annotation indicates that you want to enable Spring's scheduling capabilities, which invoke methods annotated with @Scheduled with the specified fixedDelay of 1 second.

The sendEvents method constructs a UsageDetail object and then sends it to the the output channel by accessing the Source object's output().send() method.

#### Configuring the UsageDetailSender application

When configuring the producer application, we need to set the output binding destination (Kafka topic) where the producer publishes the data.

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

spring.cloud.stream.bindings.output.destination=usage-detail

The spring.cloud.stream.bindings.output.destination property binds the UsageDetailSender object's output to the usage-detail Kafka topic.

#### 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 the spring-cloud-stream-test-support dependency to test the Spring Cloud Stream application. 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 uses a utility class called MessageCollector, which stores the messages in-memory.

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

package io.spring.dataflow.sample.usagedetailsender;

import java.util.concurrent.TimeUnit;

import com.fasterxml.jackson.databind.ObjectMapper;

import io.spring.dataflow.sample.UsageDetail;

import org.json.JSONObject;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.cloud.stream.messaging.Source;

import org.springframework.cloud.stream.test.binder.MessageCollector;

import org.springframework.messaging.Message;

import org.springframework.test.context.junit4.SpringRunner;

import org.springframework.util.Assert;

import static org.junit.Assert.assertTrue;

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class UsageDetailSenderApplicationTests {

@Autowired

private MessageCollector messageCollector;

@Autowired

private Source source;

@Test

public void **contextLoads**() {

}

@Test

public void **testUsageDetailSender**() throws Exception {

Message message = this.messageCollector.**forChannel**(this.source.**output**()).**poll**(1, TimeUnit.SECONDS);

String usageDetailJSON = message.**getPayload**().**toString**();

**assertTrue**(usageDetailJSON.**contains**("userId"));

**assertTrue**(usageDetailJSON.**contains**("duration"));

**assertTrue**(usageDetailJSON.**contains**("data"));

}

}

When using the spring-cloud-stream-test-support dependency, your application's output and input are bound to the Test binder.

* The contextLoads test case verifies the application starts successfully.
* The testUsageDetailSender test case uses the Test binder's MessageCollector to collect the messages sent by the UsageDetailSender.

### UsageCostProcessor Processor

Either [download the initialzr generated project directly](https://start.spring.io/starter.zip?&type=maven-project&language=java&bootVersion=2.1.4.RELEASE&baseDir=usage-cost-processor-kafka&groupId=io.spring.dataflow.sample&artifactId=usage-cost-processor-kafka&name=usage-cost-processor-kafka&description=Demo+project+for+Spring+Boot&packageName=io.spring.dataflow.sample.usagecostprocessor&packaging=jar&javaVersion=1.8&dependencies=kafka&dependencies=cloud-stream&dependencies=actuator&dependencies=web&dependencies=cloud-connectors) or visit the [Spring Initialzr site](https://start.spring.io/) 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](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-kafka/usage-cost-processor/src/main/java/io/spring/dataflow/sample/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](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-kafka/usage-cost-processor/src/main/java/io/spring/dataflow/sample/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 io.spring.dataflow.sample.UsageCostDetail;

import io.spring.dataflow.sample.UsageDetail;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.annotation.StreamListener;

import org.springframework.cloud.stream.messaging.Processor;

import org.springframework.messaging.handler.annotation.SendTo;

@EnableBinding(Processor.class)

public class UsageCostProcessor {

private double ratePerSecond = 0.1;

private double ratePerMB = 0.05;

@StreamListener(Processor.INPUT)

@SendTo(Processor.OUTPUT)

public UsageCostDetail **processUsageCost**(UsageDetail usageDetail) {

UsageCostDetail usageCostDetail = new UsageCostDetail();

usageCostDetail.**setUserId**(usageDetail.**getUserId**());

usageCostDetail.**setCallCost**(usageDetail.**getDuration**() \* this.ratePerSecond);

usageCostDetail.**setDataCost**(usageDetail.**getData**() \* this.ratePerMB);

return usageCostDetail;

}

}

In the preceding application, the @EnableBinding annotation indicates that you want to bind your application to the messaging middleware. The annotation takes one or more interfaces as a parameter — in this case, the [Processor](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream/src/main/java/org/springframework/cloud/stream/messaging/Processor.java) that defines and input and output channels.

The @StreamListener annotation binds the application's input channel to the processUsageCost method by converting the incoming JSON into UsageDetail object. We configure the Kafka topic that is bound to the input channel later.

The @SendTo annotation sends the processUsageCost method's output to the application's output channel, which is, in turn, sent to the a Kafka topic that we configure later.

#### Configuring the UsageCostProcessor Application

When configuring the consumer application, we need to set the input binding destination (a Kafka topic).

Since the UsageCostProcessor application is also a producer application, we need to set the output binding destination (a Kafka topic) where the producer publishes the data.

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

spring.cloud.stream.bindings.input.destination=usage-detail

spring.cloud.stream.bindings.output.destination=usage-cost

1. The spring.cloud.stream.bindings.input.destination property binds the UsageCostProcessor object's input to the usage-detail Kafka topic.
2. The spring.cloud.stream.bindings.output.destination property binds the UsageCostProcessor object's output 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

Spring Cloud Stream provides the spring-cloud-stream-test-support dependency to test the Spring Cloud Stream application. Instead of the Kafka binder, it uses the Test binder to trace and test your application's outbound and inbound messages. The Test binder uses a utility class called MessageCollector, which stores the messages in-memory.

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

package io.spring.dataflow.sample.usagecostprocessor;

import java.util.concurrent.TimeUnit;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.cloud.stream.messaging.Processor;

import org.springframework.cloud.stream.test.binder.MessageCollector;

import org.springframework.messaging.Message;

import org.springframework.messaging.support.MessageBuilder;

import org.springframework.test.context.junit4.SpringRunner;

import static org.junit.Assert.assertTrue;

@RunWith(SpringRunner.class)

@SpringBootTest

public class UsageCostProcessorApplicationTests {

@Autowired

private Processor processor;

@Autowired

private MessageCollector messageCollector;

@Test

public void **contextLoads**() {

}

@Test

public void **testUsageCostProcessor**() throws Exception {

this.processor.**input**().**send**(MessageBuilder.**withPayload**("{\"userId\":\"user3\",\"duration\":101,\"data\":502}").**build**());

Message message = this.messageCollector.**forChannel**(this.processor.**output**()).**poll**(1, TimeUnit.SECONDS);

**assertTrue**(message.**getPayload**().**toString**().**equals**("{\"userId\":\"user3\",\"callCost\":10.100000000000001,\"dataCost\":25.1}"));

}

}

* The contextLoads test case verifies the application starts successfully.
* The testUsageCostProcessor test case uses the Test binder's MessageCollector to collect the messages from the UsageCostProcessor object's output.

### UsageCostLogger Sink

Either [download the initialzr generated project directly](https://start.spring.io/starter.zip?type=maven-project&language=java&bootVersion=2.1.4.RELEASE&baseDir=usage-cost-logger-kafka&groupId=io.spring.dataflow.sample&artifactId=usage-cost-logger-kafka&name=usage-cost-logger-kafka&description=Demo+project+for+Spring+Boot&packageName=io.spring.dataflow.sample.usagecostlogger&packaging=jar&javaVersion=1.8&dependencies=kafka&dependencies=cloud-stream&dependencies=actuator&dependencies=web&dependencies=cloud-connectors) or visit the [Spring Initialzr site](https://start.spring.io/) 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](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-kafka/usage-cost-logger/src/main/java/io/spring/dataflow/sample/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 io.spring.dataflow.sample.UsageCostDetail;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.annotation.StreamListener;

import org.springframework.cloud.stream.messaging.Sink;

@EnableBinding(Sink.class)

public class UsageCostLogger {

private static final Logger logger = LoggerFactory.**getLogger**(UsageCostLoggerApplication.class);

@StreamListener(Sink.INPUT)

public void **process**(UsageCostDetail usageCostDetail) {

logger.**info**(usageCostDetail.**toString**());

}

}

In the preceding application, the @EnableBinding annotation indicates that you want to bind your application to the messaging middleware. The annotation takes one or more interfaces as a parameter — in this case, the [Sink](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream/src/main/java/org/springframework/cloud/stream/messaging/Sink.java) interface that defines the input channel.

The @StreamListener annotation binds the application's input channel to the process method by converting the incoming JSON to a UsageCostDetail object.

We configure the Kafka topic that is bound to the input channel later.

#### Configuring the UsageCostLogger Application

When configuring the consumer application, we need to set the input binding destination (a Kafka topic).

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

spring.cloud.stream.bindings.input.destination=usage-cost

The spring.cloud.stream.bindings.input.destination property binds the UsageCostLogger object's input to the usage-cost Kafka topic.

#### 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 io.spring.dataflow.sample.UsageCostDetail;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.ArgumentCaptor;

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

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.messaging.Sink;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Primary;

import org.springframework.messaging.support.MessageBuilder;

import org.springframework.test.context.junit4.SpringRunner;

import static org.mockito.Mockito.spy;

import static org.mockito.Mockito.verify;

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class UsageCostLoggerApplicationTests {

@Autowired

protected Sink sink;

@Autowired

protected UsageCostLogger usageCostLogger;

@Test

public void **contextLoads**() {

}

@Test

public void **testUsageCostLogger**() throws Exception {

ArgumentCaptor<UsageCostDetail> captor = ArgumentCaptor.**forClass**(UsageCostDetail.class);

this.sink.**input**().**send**(MessageBuilder.**withPayload**("{\"userId\":\"user3\",\"callCost\":10.100000000000001,\"dataCost\":25.1}").**build**());

**verify**(this.usageCostLogger).**process**(captor.**capture**());

}

@EnableAutoConfiguration

@EnableBinding(Sink.class)

static class TestConfig {

*// Override `UsageCostLogger` bean for spying.*

@Bean

@Primary

public UsageCostLogger **usageCostLogger**() {

return **spy**(new UsageCostLogger());

}

}

}

* The contextLoads test case verifies the application starts successfully.
* The testUsageCostLogger test case verifies that the process method of UsageCostLogger is invoked by using Mockito. To do this, the TestConfig static class overrides the existing UsageCostLogger bean to create a Mock bean of UsageCostLogger. Since we are mocking the UsageCostLogger bean, the TestConfig also explicitly annotates @EnableBinding and @EnableAutoConfiguration.

## 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](https://kafka.apache.org/downloads) 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](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-kafka/#configuring-the-usagedetailsender-application) 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=9001 &

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-detail

To list the topics, run the following command:

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

#### Running the Processor

By using the [pre-defined](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-kafka/#configuring-the-usagecostprocessor-application) 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=9002 &

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 usage-cost

#### Running the Sink

By using the [pre-defined](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-kafka/#configuring-the-usagecostlogger-application) 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\_Service\_Port]

SPRING\_CLOUD\_STREAM\_KAFKA\_BINDER\_ZKNODES: [ZooKeeper\_Service\_IP\_Address:ZooKeeper\_Service\_Port]

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\_Service\_Port]

SPRING\_CLOUD\_STREAM\_KAFKA\_BINDER\_ZKNODES: [ZooKeeper\_Service\_IP\_Address:ZooKeeper\_Service\_Port]

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\_Service\_Port]

SPRING\_CLOUD\_STREAM\_KAFKA\_BINDER\_ZKNODES: [ZooKeeper\_Service\_IP\_Address:ZooKeeper\_Service\_Port]

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-logger.cfapps.io

usage-cost-processor started 1/1 1G 1G usage-cost-processor.cfapps.io

usage-detail-sender started 1/1 1G 1G usage-detail-sender.cfapps.io

2019-05-13T23:23:33.36+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:33.362 INFO 15 --- [e-cost.logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user5", "callCost": "1.0", "dataCost": "12.350000000000001" }

2019-05-13T23:23:33.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:33.467 INFO 15 --- [e-cost.logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user1", "callCost": "19.0", "dataCost": "10.0" }

2019-05-13T23:23:34.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:34.466 INFO 15 --- [e-cost.logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user4", "callCost": "2.2", "dataCost": "5.15" }

2019-05-13T23:23:35.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:35.469 INFO 15 --- [e-cost.logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user3", "callCost": "21.0", "dataCost": "17.3" }

### 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](https://dataflow.spring.io/docs/installation/kubernetes/#creating-a-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-dataflow/master/src/kubernetes/kafka/kafka-deployment.yaml \

-f https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/master/src/kubernetes/kafka/kafka-svc.yaml \

-f https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/master/src/kubernetes/kafka/kafka-zk-deployment.yaml \

-f https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/master/src/kubernetes/kafka/kafka-zk-svc.yaml

#### Building Docker Images

To build Docker images, we use the [jib Maven plugin](https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin#build-your-image). If you downloaded the [source distribution](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-kafka/#development), 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 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

---

kind: Pod

apiVersion: v1

metadata:

name: usage-cost-processor

labels:

app: usage-cost-stream

spec:

containers:

- name: usage-cost-processor

image: springcloudstream/usage-cost-processor-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\_INPUT\_GROUP

value: usage-cost-stream

- name: SPRING\_CLOUD\_STREAM\_BINDINGS\_INPUT\_DESTINATION

value: user-details

- name: SPRING\_CLOUD\_STREAM\_BINDINGS\_OUTPUT\_DESTINATION

value: user-cost

- name: SERVER\_PORT

value: '80'

restartPolicy: Always

---

kind: Pod

apiVersion: v1

metadata:

name: usage-cost-logger

labels:

app: usage-cost-stream

spec:

containers:

- name: usage-cost-logger

image: springcloudstream/usage-cost-logger-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\_INPUT\_GROUP

value: usage-cost-stream

- name: SPRING\_CLOUD\_STREAM\_BINDINGS\_INPUT\_DESTINATION

value: user-cost

- 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.UsageCostLoggerApplication : {"userId": "Mark", "callCost": "21.1", "dataCost": "26.05" }

2019-05-02 15:48:19.553 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "4.2", "dataCost": "15.75" }

2019-05-02 15:48:20.549 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Mark", "callCost": "28.400000000000002", "dataCost": "15.0" }

2019-05-02 15:48:21.553 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "16.8", "dataCost": "28.5" }

2019-05-02 15:48:22.551 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Mark", "callCost": "22.700000000000003", "dataCost": "20.3" }

2019-05-02 15:48:23.556 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "16.6", "dataCost": "2.6" }

2019-05-02 15:48:24.557 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "6.7", "dataCost": "1.0" }

2019-05-02 15:48:25.555 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Glenn", "callCost": "3.7", "dataCost": "2.6500000000000004" }

2019-05-02 15:48:26.557 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "24.200000000000003", "dataCost": "32.9" }

2019-05-02 15:48:27.556 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Glenn", "callCost": "19.200000000000003", "dataCost": "7.4" }

2019-05-02 15:48:28.559 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Sabby", "callCost": "17.7", "dataCost": "27.35" }

2019-05-02 15:48:29.562 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "26.8", "dataCost": "32.45" }

2019-05-02 15:48:30.561 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "26.5", "dataCost": "33.300000000000004" }

2019-05-02 15:48:31.562 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Sabby", "callCost": "16.1", "dataCost": "5.0" }

2019-05-02 15:48:32.564 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "16.3", "dataCost": "23.6" }

2019-05-02 15:48:33.567 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "29.400000000000002", "dataCost": "2.1" }

2019-05-02 15:48:34.567 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "5.2", "dataCost": "20.200000000000003" }

#### 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

# Stream Processing with RabbitMQ

In this guide, we develop three Spring Boot applications that use Spring Cloud Stream's support for RabbitMQ and deploy them to Cloud Foundry, to Kubernetes, and on your local machine. In another guide, we [deploy these applications by using Data Flow](https://dataflow.spring.io/docs/stream-developer-guides/streams/data-flow-stream/). By deploying the applications manually, you get a better understanding of the steps that Data Flow automates for you.

The following sections describe how to build these applications from scratch. If you prefer, you can download a zip file containing the sources for these applications, unzip it, and proceed to the [deployment](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-rabbitmq/#deployment) section.

You can [download the project](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-rabbitmq/dist/usage-cost-stream-rabbit.zip?raw=true) that contains all three applications from your browser. You can also use the command line, as the following example shows:

**wget** https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-rabbitmq/dist/usage-cost-stream-rabbit.zip?raw=true -O usage-cost-stream-rabbit.zip

## Development

We create three Spring Cloud Stream applications that communicate by using RabbitMQ.

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 that contains the the duration of the call and the amount of data used during the call. The bill is modeled by using the UsageCostDetail class that 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 users' call duration and the amount of data used for each userId and sends a message that contains 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 data.

### Source

In this step, we create the UsageDetailSender source.

You can either [download the initialzr generated project directly](https://start.spring.io/starter.zip?type=maven-project&language=java&bootVersion=2.1.4.RELEASE&baseDir=usage-detail-sender-rabbit&groupId=io.spring.dataflow.sample&artifactId=usage-detail-sender-rabbit&name=usage-detail-sender-rabbit&description=Demo+project+for+Spring+Boot&packageName=io.spring.dataflow.sample.usagedetailsender&packaging=jar&javaVersion=1.8&dependencies=amqp&dependencies=cloud-stream&dependencies=actuator&dependencies=web&dependencies=cloud-connectors) orvisit the [Spring Initialzr site](https://start.spring.io/) 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-rabbit.
2. In the **Dependencies** text box, type RabbitMQ to select the RabbitMQ 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-rabbit.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 that looks like the contents in [UsageDetail.java](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-rabbitmq/usage-detail-sender/src/main/java/io/spring/dataflow/sample/UsageDetail.java). The UsageDetail class contains userId, data, and duration properties.
2. Create the UsageDetailSender class in the io.spring.dataflow.sample.usagedetailsender package, which resembles the following listing:

package io.spring.dataflow.sample.usagedetailsender;

import java.util.Random;

import io.spring.dataflow.sample.domain.UsageDetail;

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

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.messaging.Source;

import org.springframework.messaging.support.MessageBuilder;

import org.springframework.scheduling.annotation.EnableScheduling;

import org.springframework.scheduling.annotation.Scheduled;

@EnableScheduling

@EnableBinding(Source.class)

public class UsageDetailSender {

@Autowired

private Source source;

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

@Scheduled(fixedDelay = 1000)

public void **sendEvents**() {

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.source.**output**().**send**(MessageBuilder.**withPayload**(usageDetail).**build**());

}

}

The @EnableBinding annotation indicates that you want to bind your application to messaging middleware. The annotation takes one or more interfaces as a parameter — in this case, the [Source](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream/src/main/java/org/springframework/cloud/stream/messaging/Source.java) interface that defines an output channel named output. In the case of RabbitMQ, messages sent to the output channel are in turn sent to the RabbitMQ message broker by using a TopicExchange.

The @EnableScheduling annotation indicates that you want to enable Spring's scheduling capabilities, which invokes methods annotated with @Scheduled with the specified fixedDelay of 1 second.

The sendEvents method constructs a UsageDetail object and then sends it to the the output channel by accessing the Source object's output().send() method.

#### Configuration

When configuring the source application, we need to set:

* The output binding destination (RabbitMQ exchange) where the producer publishes the data.
* The requiredGroups to specify the consumer groups to ensure the message delivery to consumer applications.

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

spring.cloud.stream.bindings.output.destination=usage-detail

spring.cloud.stream.bindings.output.producer.requiredGroups=usage-cost-consumer

* The spring.cloud.stream.bindings.output.destination property binds the UsageDetailSender object's output to the usage-detail RabbitMQ exchange.
* The spring.cloud.stream.bindings.output.producer.requiredGroups property makes sure to create a durable queue named usage-detail.usage-cost-consumer, which consumes from the usage-detail RabbitMQ exchange.

##### Durable Queues

By default, the Spring Cloud Stream consumer application creates an anonymous auto-delete queue. This can result in a message not being stored and forwarded by the producer if the producer application started before the consumer application. Even though the exchange is durable, we need a durable queue to be bound to the exchange for the message to be stored for later consumption. Hence, for guaranteed message delivery, you need a durable queue.

To pre-create durable queues and bind them to the exchange, the producer application should set the following property:

spring.cloud.stream.bindings.<channelName>.producer.requiredGroups

The requiredGroups property accepts a comma-separated list of groups to which the producer must ensure message delivery. When this property is set, a durable queue is created by using the <exchange>.<requiredGroup> format.

#### 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 the spring-cloud-stream-test-support dependency to test the Spring Cloud Stream application. Instead of the RabbitMQ binder, the tests use the Test binder to trace and test your application's outbound and inbound messages. The Test binder uses a utility class called MessageCollector, which stores the messages in-memory.

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

package io.spring.dataflow.sample.usagedetailsender;

import java.util.concurrent.TimeUnit;

import com.fasterxml.jackson.databind.ObjectMapper;

import io.spring.dataflow.sample.UsageDetail;

import org.json.JSONObject;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.cloud.stream.messaging.Source;

import org.springframework.cloud.stream.test.binder.MessageCollector;

import org.springframework.messaging.Message;

import org.springframework.test.context.junit4.SpringRunner;

import org.springframework.util.Assert;

import static org.junit.Assert.assertTrue;

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class UsageDetailSenderApplicationTests {

@Autowired

private MessageCollector messageCollector;

@Autowired

private Source source;

@Test

public void **contextLoads**() {

}

@Test

public void **testUsageDetailSender**() throws Exception {

Message message = this.messageCollector.**forChannel**(this.source.**output**()).**poll**(1, TimeUnit.SECONDS);

String usageDetailJSON = message.**getPayload**().**toString**();

**assertTrue**(usageDetailJSON.**contains**("userId"));

**assertTrue**(usageDetailJSON.**contains**("duration"));

**assertTrue**(usageDetailJSON.**contains**("data"));

}

}

* The contextLoads test case verifies that the application starts successfully.
* The testUsageDetailSender test case uses the Test binder's MessageCollector to collect the messages sent by the UsageDetailSender.

### Processor

In this step, we create the UsageCostProcessor processor.

Either [download the initialzr generated project directly](https://start.spring.io/starter.zip?type=maven-project&language=java&bootVersion=2.1.4.RELEASE&baseDir=usage-cost-processor-rabbit&groupId=io.spring.dataflow.sample&artifactId=usage-cost-processor-rabbit&name=usage-cost-processor-rabbit&description=Demo+project+for+Spring+Boot&packageName=io.spring.dataflow.sample.usagecostprocessor&packaging=jar&javaVersion=1.8&dependencies=amqp&dependencies=cloud-stream&dependencies=actuator&dependencies=web&dependencies=cloud-connectors) or visit the [Spring Initialzr site](https://start.spring.io/) 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-rabbit.
2. In the **Dependencies** text box, type Rabbitmq to select the RabbitMQ 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-rabbit.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 the UsageDetail class in the io.spring.dataflow.sample.usagecostprocessor. Its contents resemble the contents of [UsageDetail.java](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-rabbitmq/usage-cost-processor/src/main/java/io/spring/dataflow/sample/UsageDetail.java). The UsageDetail class contains userId, data, and duration properties
2. Create the UsageCostDetail class in the io.spring.dataflow.sample.usagecostprocessor package. Its contents resemble the contents of [UsageCostDetail.java](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-rabbitmq/usage-cost-processor/src/main/java/io/spring/dataflow/sample/UsageCostDetail.java). The UsageCostDetail class contains userId, callCost, and dataCost properties.
3. Create the UsageCostProcessor class in the io.spring.dataflow.sample.usagecostprocessor package, which 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 io.spring.dataflow.sample.UsageCostDetail;

import io.spring.dataflow.sample.UsageDetail;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.annotation.StreamListener;

import org.springframework.cloud.stream.messaging.Processor;

import org.springframework.messaging.handler.annotation.SendTo;

@EnableBinding(Processor.class)

public class UsageCostProcessor {

private double ratePerSecond = 0.1;

private double ratePerMB = 0.05;

@StreamListener(Processor.INPUT)

@SendTo(Processor.OUTPUT)

public UsageCostDetail **processUsageCost**(UsageDetail usageDetail) {

UsageCostDetail usageCostDetail = new UsageCostDetail();

usageCostDetail.**setUserId**(usageDetail.**getUserId**());

usageCostDetail.**setCallCost**(usageDetail.**getDuration**() \* this.ratePerSecond);

usageCostDetail.**setDataCost**(usageDetail.**getData**() \* this.ratePerMB);

return usageCostDetail;

}

}

In the preceding application, the @EnableBinding annotation indicates that you want to bind your application to the messaging middleware. The annotation takes one or more interfaces as a parameter — in this case, the [Processor](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream/src/main/java/org/springframework/cloud/stream/messaging/Processor.java) that defines and input and output channel.

The @StreamListener annotation binds the application's input channel to the processUsageCost method by converting the incoming JSON into UsageDetail object.

The @SendTo annotation sends the processUsageCost method's output to the application's output channel, which is, in turn, sent to the a RabbitMQ message broker by using a TopicExchange.

#### Configuration

When configuring the processor application, we need to set the following properties:

* The input binding destination (RabbitMQ exchange) where this application is subscribed through an anonymous auto-delete or durable queue.
* The group to specify the consumer group to which this consumer application belongs.
* The output binding destination (RabbitMQ exchange) where the producer publishes the data.
* The requiredGroups to specify the consumer groups to ensure the message delivery guarantee.

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

spring.cloud.stream.bindings.input.destination=usage-detail

spring.cloud.stream.bindings.input.group=usage-cost-consumer

spring.cloud.stream.bindings.output.destination=usage-cost

spring.cloud.stream.bindings.output.producer.requiredGroups=logger

* The spring.cloud.stream.bindings.input.destination and spring.cloud.stream.bindings.input.group properties bind the UsageCostProcessor object's input to the usage-detail RabbitMQ exchange through the usage-detail.usage-cost-consumer durable queue.
* The spring.cloud.stream.bindings.output.destination property binds the UsageCostProcessor object's output to the usage-cost RabbitMQ exchange.
* The spring.cloud.stream.bindings.output.producer.requiredGroups property makes sure to create a durable queue named usage-cost.logger, which consumes from the usage-cost RabbitMQ exchange.

#### Building

Now we can build the Usage Cost Processor application. In the usage-cost-processor directory, use the following command to build the project using maven.

./mvnw clean package

#### Testing

Spring Cloud Stream provides the spring-cloud-stream-test-support dependency to test the Spring Cloud Stream application. Instead of the RabbitMQ binder, it uses the Test binder to trace and test your application's outbound and inbound messages. The Test binder uses a utility class MessageCollector, which stores the messages in-memory.

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

package io.spring.dataflow.sample.usagecostprocessor;

import java.util.concurrent.TimeUnit;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.cloud.stream.messaging.Processor;

import org.springframework.cloud.stream.test.binder.MessageCollector;

import org.springframework.messaging.Message;

import org.springframework.messaging.support.MessageBuilder;

import org.springframework.test.context.junit4.SpringRunner;

import static org.junit.Assert.assertTrue;

@RunWith(SpringRunner.class)

@SpringBootTest

public class UsageCostProcessorApplicationTests {

@Autowired

private Processor processor;

@Autowired

private MessageCollector messageCollector;

@Test

public void **contextLoads**() {

}

@Test

public void **testUsageCostProcessor**() throws Exception {

this.processor.**input**().**send**(MessageBuilder.**withPayload**("{\"userId\":\"user3\",\"duration\":101,\"data\":502}").**build**());

Message message = this.messageCollector.**forChannel**(this.processor.**output**()).**poll**(1, TimeUnit.SECONDS);

**assertTrue**(message.**getPayload**().**toString**().**equals**("{\"userId\":\"user3\",\"callCost\":10.100000000000001,\"dataCost\":25.1}"));

}

}

* The test case contextLoads verifies the application starts successfully.
* The test case testUsageCostProcessor uses the Test binder's MessageCollector to collect the messages from the UsageCostProcessor object's output.

### Sink

In this step, we create the UsageCostLogger sink.

Either [download the initialzr generated project directly](https://start.spring.io/starter.zip?type=maven-project&language=java&bootVersion=2.1.4.RELEASE&baseDir=usage-cost-logger-rabbit&groupId=io.spring.dataflow.sample&artifactId=usage-cost-logger-rabbit&name=usage-cost-logger-rabbit&description=Demo+project+for+Spring+Boot&packageName=io.spring.dataflow.sample.usagecostlogger&packaging=jar&javaVersion=1.8&dependencies=cloud-stream&dependencies=amqp&dependencies=actuator&dependencies=web&dependencies=cloud-connectors) or visit the [Spring Initialzr site](https://start.spring.io/) and follow these instructions:

1. Create a new Maven project with a Group name of io.spring.dataflow and an Artifact name of usage-cost-logger-rabbit.
2. In the **Dependencies** text box, type rabbitmq to select the RabbitMQ 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-rabbit.zip file and import the project into your favorite IDE.

#### Business Logic

To create the business logic:

1. Create a UsageCostDetail class in the io.spring.dataflow.sample.sagecostlogger package. Its contents should resemble the contents of [UsageCostDetail.java](https://github.com/spring-cloud/spring-cloud-dataflow-samples/blob/master/dataflow-website/stream-developer-guides/streams/standalone-stream-rabbitmq/usage-cost-logger/src/main/java/io/spring/dataflow/sample/UsageCostDetail.java). The UsageCostDetail class contains userId, callCost, and dataCost properties.
2. Create the UsageCostLogger class in the io.spring.dataflow.sample.usagecostlogger package, which receives the UsageCostDetail message and logs it. The following listing shows the source code:

package io.spring.dataflow.sample.usagecostlogger;

import io.spring.dataflow.sample.UsageCostDetail;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.annotation.StreamListener;

import org.springframework.cloud.stream.messaging.Sink;

@EnableBinding(Sink.class)

public class UsageCostLogger {

private static final Logger logger = LoggerFactory.**getLogger**(UsageCostLoggerApplication.class);

@StreamListener(Sink.INPUT)

public void **process**(UsageCostDetail usageCostDetail) {

logger.**info**(usageCostDetail.**toString**());

}

}

In the preceding application, the @EnableBinding annotation indicates that you want to bind your application to the messaging middleware. The annotation takes one or more interfaces as a parameter — in this case, the [Sink](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream/src/main/java/org/springframework/cloud/stream/messaging/Sink.java) interface that defines an input channel.

The @StreamListener annotation binds the application's input channel to the process method by converting the incoming JSON to a UsageCostDetail object.

#### Configuration

When configuring the sink application, we need to set:

* The input binding destination (RabbitMQ exchange) to which this application is subscribed through an anonymous auto-delete or durable queue.
* The group to specify the consumer group to which this consumer application belongs.

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

spring.cloud.stream.bindings.input.destination=usage-cost

spring.cloud.stream.bindings.input.group=logger

The spring.cloud.stream.bindings.input.destination and spring.cloud.stream.bindings.input.group properties bind the UsageCostLogger object's input to the usage-cost RabbitMQ exchange through the usage-cost.logger durable queue.

#### Building

Now we can build the Usage Cost Logger application. In the usage-cost-logger directory, use 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 io.spring.dataflow.sample.UsageCostDetail;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.ArgumentCaptor;

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

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.messaging.Sink;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Primary;

import org.springframework.messaging.support.MessageBuilder;

import org.springframework.test.context.junit4.SpringRunner;

import static org.mockito.Mockito.spy;

import static org.mockito.Mockito.verify;

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class UsageCostLoggerApplicationTests {

@Autowired

protected Sink sink;

@Autowired

protected UsageCostLogger usageCostLogger;

@Test

public void **contextLoads**() {

}

@Test

public void **testUsageCostLogger**() throws Exception {

ArgumentCaptor<UsageCostDetail> captor = ArgumentCaptor.**forClass**(UsageCostDetail.class);

this.sink.**input**().**send**(MessageBuilder.**withPayload**("{\"userId\":\"user3\",\"callCost\":10.100000000000001,\"dataCost\":25.1}").**build**());

**verify**(this.usageCostLogger).**process**(captor.**capture**());

}

@EnableAutoConfiguration

@EnableBinding(Sink.class)

static class TestConfig {

*// Override `UsageCostLogger` bean for spying.*

@Bean

@Primary

public UsageCostLogger **usageCostLogger**() {

return **spy**(new UsageCostLogger());

}

}

}

* The contextLoads test case verifies the application starts successfully.
* The testUsageCostLogger test case verifies that the process method of UsageCostLogger is invoked by using Mockito. To do this, the static TestConfig class overrides the existing UsageCostLogger bean to create a mock bean of UsageCostLogger. Since we are mocking the UsageCostLogger bean, the TestConfig also explicitly annotates @EnableBinding and @EnableAutoConfiguration.

## Deployment

In this section, we deploy the applications created earlier to the local machine, Cloud Foundry, and 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 RabbitMQ binder binds the applications' output and input boundaries into the corresponding exchanges and queues at RabbitMQ message broker.

### Local

You can run the applications as standalone applications on your local environment.

To install and run the RabbitMQ docker image, run the following command:

docker run -d --hostname rabbitmq --name rabbitmq -p 15672:15672 -p 5672:5672 rabbitmq:3.7.14-management

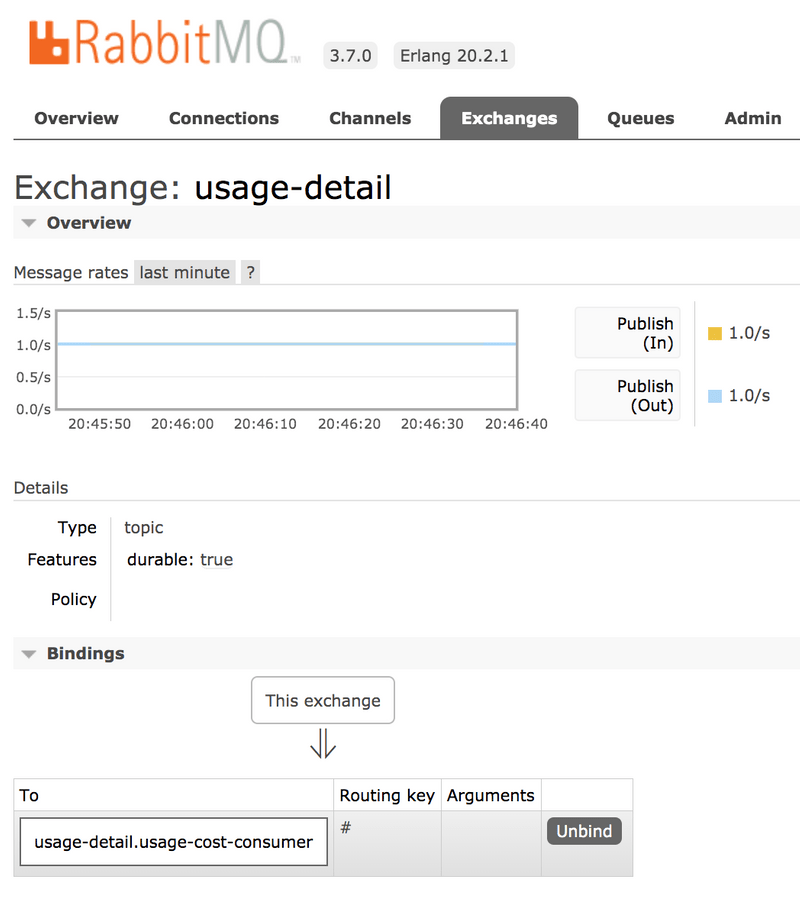
Once installed, you can log in to the RabbitMQ management console on your local machine on [http://localhost:15672](http://localhost:15672/). You can use the default account username and password: guest and guest.

#### Running the UsageDetailSender Source

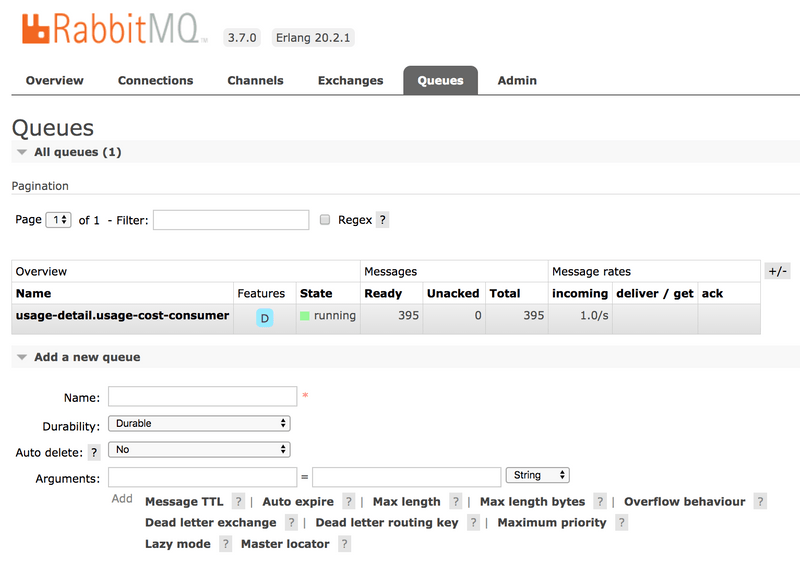
By using the [pre-defined](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-rabbitmq/#configuring-the-usagedetailsender-application) configuration properties(along with a unique server port) for UsageDetailSender, you can run the application, as follows:

java -jar target/usage-detail-sender-rabbit-0.0.1-SNAPSHOT.jar --server.port=9001 &

When this application is running, you can see that the usage-detail RabbitMQ exchange is created and the durable queue named usage-detail.usage-cost-consumer is bound to this exchange, as the following example shows:



Also, if you click on the Queues and check the queue usage-detail.usage-cost-consumer, you can see the messages being consumed and stored in this durable queue, as the following example shows:



When configuring the consumer applications for this Source application, you can set the group binding property to connect to the corresponding durable queue.

If you do not set the requiredGroups property, you can see that there is no queue for consuming the messages from the usage-detail exchange and, therefore, the messages are lost if the consumer is not up before this application is started.

#### Running the Processor

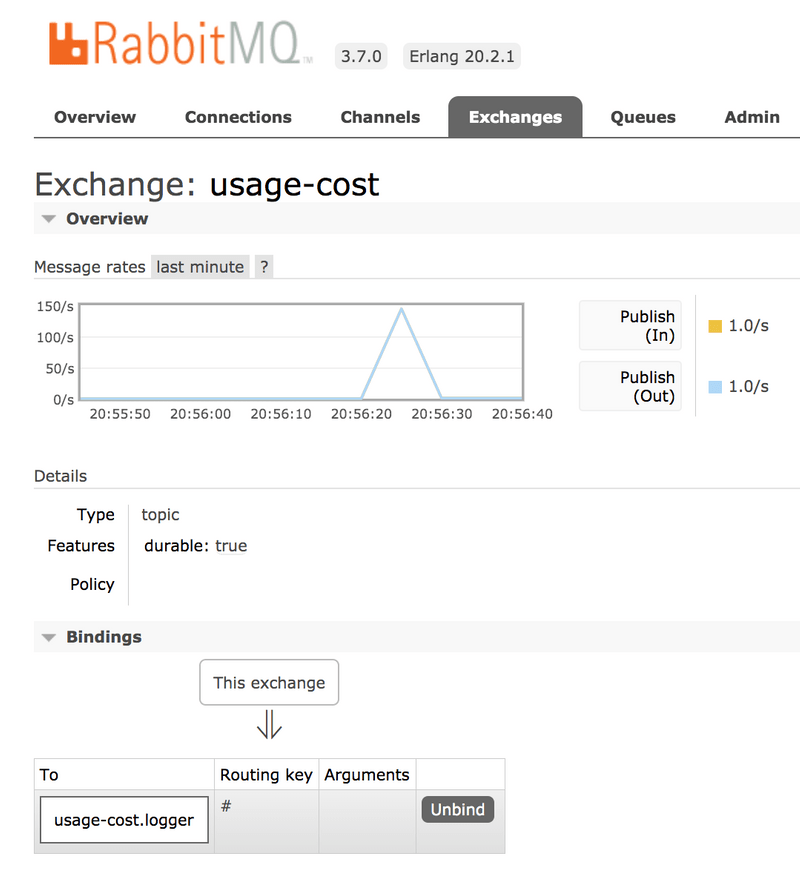
By using the [pre-defined](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-rabbitmq/#configuring-the-usagecostprocessor-application) configuration properties (along with a unique server port) for UsageCostProcessor, you can run the application, as follows:

java -jar target/usage-cost-processor-rabbit-0.0.1-SNAPSHOT.jar --server.port=9002 &

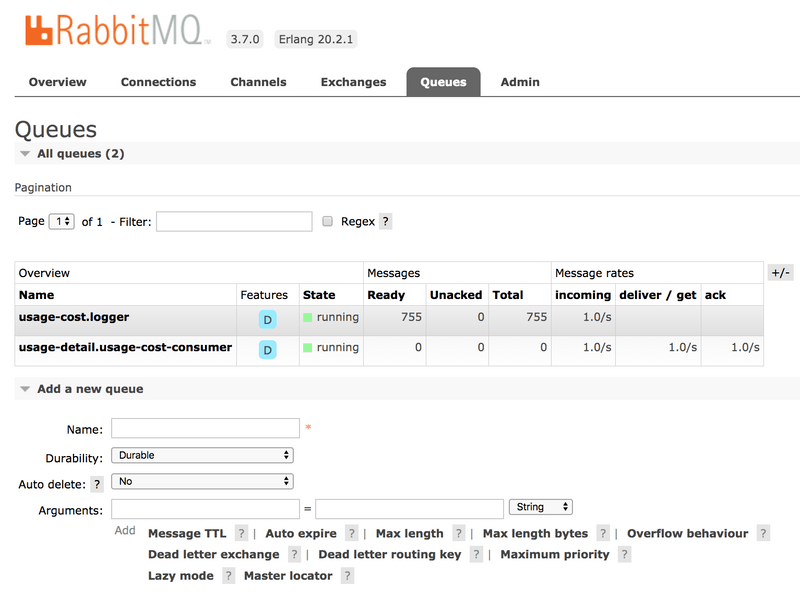
From the RabbitMQ console, you can see:

* The UsageCostProcessor application consumes from the usage-detail.usage-cost-consumer durable queue, based on the spring.cloud.stream.bindings.input.group=usage-cost-consumer property.
* The UsageCostProcessor application produces the UsageCostDetail and sends it to the exchange usage-cost, based on the spring.cloud.stream.bindings.output.destination=usage-cost property.
* The usage-cost.logger durable queue is created. It consumes the messages from the usage-cost exchange, based on the spring.cloud.stream.bindings.output.producer.requiredGroups=logger property.

When this application is running, you can see that the usage-cost RabbitMQ exchange is created and the durable queue named usage-cost.logger is bound to this exchange, as the following image shows:



Also, if you click on the Queues and check the usage-cost.logger queue, you can see the messages being consumed and stored in this durable queue, as the following image shows:



#### Running the Sink

By using the [pre-defined](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-rabbitmq/#configuring-the-usagecostlogger-application) configuration properties (along with a unique server port) for UsageCostLogger, you can run the application, as follows:

java -jar target/usage-cost-logger-rabbit-0.0.1-SNAPSHOT.jar --server.port=9003 &

Now you can see that this application logs the usage cost detail it receives from the usage-cost RabbitMQ exchange through the usage-cost.logger durable queue, as the following example shows:

2019-05-08 08:16:46.442 INFO 10769 --- [o6VmGALOP\_onw-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user2", "callCost": "28.3", "dataCost": "29.8" }

2019-05-08 08:16:47.446 INFO 10769 --- [o6VmGALOP\_onw-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user2", "callCost": "12.0", "dataCost": "23.75" }

2019-05-08 08:16:48.451 INFO 10769 --- [o6VmGALOP\_onw-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user4", "callCost": "16.0", "dataCost": "30.05" }

2019-05-08 08:16:49.454 INFO 10769 --- [o6VmGALOP\_onw-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user1", "callCost": "17.7", "dataCost": "18.0" }

### Cloud Foundry

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

#### Creating a RabbitMQ service

To create a RabbitMQ service:

1. Log in to the PWS with your credentials.
2. From the CF market place, create a RabbitMQ service instance that uses the cloudamqp service on the lemur plan, as follows:

cf create-service cloudamqp lemur rabbitmq

#### Cloud Foundry Deployment

To deploy on Cloud Foundry:

1. Create a CF manifest YAML file named usage-detail-sender.yml for the UsageDetailSender by using its [configuration properties](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-rabbitmq/#configuring-the-usagedetailsender-application), as follows:

applications:

- name: usage-detail-sender

timeout: 120

path: ./target/usage-detail-sender-rabbit-0.0.1-SNAPSHOT.jar

memory: 1G

buildpack: java\_buildpack

services:

- rabbitmq

Push the UsageDetailSender application by using its manifest YAML file, as follows:

cf push -f usage-detail-sender.yml

Create a CF manifest YAML file named usage-cost-processor.yml for the UsageCostProcessor by using its [configuration properties](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-rabbitmq/#configuring-the-usagecostprocessor-application), as follows:

applications:

- name: usage-cost-processor

timeout: 120

path: ./target/usage-cost-processor-rabbit-0.0.1-SNAPSHOT.jar

memory: 1G

buildpack: java\_buildpack

services:

- rabbitmq

Push the UsageCostProcessor application by using its manifest YAML file, as follows:

cf push -f usage-cost-processor.yml

Create a CF manifest YAML file named usage-cost-logger.yml for the UsageCostLogger by using its [configuration properties](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-rabbitmq/#configuring-the-usagecostlogger-application), as follows:

applications:

- name: usage-cost-logger

timeout: 120

path: ./target/usage-cost-logger-rabbit-0.0.1-SNAPSHOT.jar

memory: 1G

buildpack: java\_buildpack

services:

- rabbitmq

Push the UsageCostLogger application by using its manifest YAML file, as follows:

cf push -f usage-cost-logger.yml

You can see the applications running by using the cf apps command, as follows:

cf apps

The following listings shows typical output:

name requested state instances memory disk urls

usage-cost-logger started 1/1 1G 1G usage-cost-logger.cfapps.io

usage-cost-processor started 1/1 1G 1G usage-cost-processor.cfapps.io

usage-detail-sender started 1/1 1G 1G usage-detail-sender.cfapps.io

2019-05-13T23:23:33.36+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:33.362 INFO 15 --- [e-cost.logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user5", "callCost": "1.0", "dataCost": "12.350000000000001" }

2019-05-13T23:23:33.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:33.467 INFO 15 --- [e-cost.logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user1", "callCost": "19.0", "dataCost": "10.0" }

2019-05-13T23:23:34.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:34.466 INFO 15 --- [e-cost.logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user4", "callCost": "2.2", "dataCost": "5.15" }

2019-05-13T23:23:35.46+0530 [APP/PROC/WEB/0] OUT 2019-05-13 17:53:35.469 INFO 15 --- [e-cost.logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user3", "callCost": "21.0", "dataCost": "17.3" }

### Running on Kubernetes

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

#### Setting up the Kubernetes cluster

For this example, we need a running [Kubernetes cluster](https://dataflow.spring.io/docs/installation/kubernetes/#creating-a-kubernetes-cluster). For this example, we deploy to minikube.

##### Verifying Minikube is running

To verify that you have a running Minikube instance, run the following command (show with sample output):

$minikube status

host: Running

kubelet: Running

apiserver: Running

kubectl: Correctly Configured: pointing to minikube-vm at 192.168.99.100

#### Installing RabbitMQ

You can install the RabbitMQ 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-dataflow/master/src/kubernetes/rabbitmq/rabbitmq-deployment.yaml \

-f https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/master/src/kubernetes/rabbitmq/rabbitmq-svc.yaml

#### Building the Docker Images

To build the Docker images, we use the [jib maven plugin](https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin#build-your-image). If you downloaded the [source distribution](https://dataflow.spring.io/docs/stream-developer-guides/streams/standalone-stream-rabbitmq/#development), 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 properties. For this example, we use the following properties:

<docker.org>springcloudstream</docker.org>

<docker.version>${project.version}</docker.version>

Now run the maven build to create the Docker images in the minikube docker registry, as follows:

$ **eval** $(minikube docker-env)

$./mvnw package jib:dockerBuild

If you downloaded the project source, the project includes a parent pom file to build all the modules with a single command. Otherwise, run the builds for the source, processor, and sink individually. You need to run eval $(minikube docker-env) only once for each terminal session.

#### Deploying the Stream

To deploy the stream, you must first and paste the following YAML content 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-rabbit:0.0.1-SNAPSHOT

ports:

- containerPort: 80

protocol: TCP

env:

- name: SPRING\_RABBITMQ\_ADDRESSES

value: rabbitmq

- name: SERVER\_PORT

value: '80'

restartPolicy: Always

---

kind: Pod

apiVersion: v1

metadata:

name: usage-cost-processor

labels:

app: usage-cost-stream

spec:

containers:

- name: usage-cost-processor

image: springcloudstream/usage-cost-processor-rabbit:0.0.1-SNAPSHOT

ports:

- containerPort: 80

protocol: TCP

env:

- name: SPRING\_RABBITMQ\_ADDRESSES

value: rabbitmq

- name: SERVER\_PORT

value: '80'

restartPolicy: Always

---

kind: Pod

apiVersion: v1

metadata:

name: usage-cost-logger

labels:

app: usage-cost-stream

spec:

containers:

- name: usage-cost-logger

image: springcloudstream/usage-cost-logger-rabbit:0.0.1-SNAPSHOT

ports:

- containerPort: 80

protocol: TCP

env:

- name: SPRING\_RABBITMQ\_ADDRESSES

value: rabbitmq

- name: SERVER\_PORT

value: '80'

restartPolicy: Always

Then you can deploy the apps, as follows:

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 respective docker image.

We set the logical hostname for the RabbitMQ broker for each app to connect to it. Here we use the RabbitMQ service name, rabbitmq in this case. We also set the label app: user-cost-stream to logically group our apps.

#### Verifying the Deployment

To verify the deployment, 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 streaming:

2019-05-02 15:48:18.550 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Mark", "callCost": "21.1", "dataCost": "26.05" }

2019-05-02 15:48:19.553 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "4.2", "dataCost": "15.75" }

2019-05-02 15:48:20.549 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Mark", "callCost": "28.400000000000002", "dataCost": "15.0" }

2019-05-02 15:48:21.553 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "16.8", "dataCost": "28.5" }

2019-05-02 15:48:22.551 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Mark", "callCost": "22.700000000000003", "dataCost": "20.3" }

2019-05-02 15:48:23.556 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "16.6", "dataCost": "2.6" }

2019-05-02 15:48:24.557 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "6.7", "dataCost": "1.0" }

2019-05-02 15:48:25.555 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Glenn", "callCost": "3.7", "dataCost": "2.6500000000000004" }

2019-05-02 15:48:26.557 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "24.200000000000003", "dataCost": "32.9" }

2019-05-02 15:48:27.556 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Glenn", "callCost": "19.200000000000003", "dataCost": "7.4" }

2019-05-02 15:48:28.559 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Sabby", "callCost": "17.7", "dataCost": "27.35" }

2019-05-02 15:48:29.562 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "26.8", "dataCost": "32.45" }

2019-05-02 15:48:30.561 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "26.5", "dataCost": "33.300000000000004" }

2019-05-02 15:48:31.562 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Sabby", "callCost": "16.1", "dataCost": "5.0" }

2019-05-02 15:48:32.564 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "16.3", "dataCost": "23.6" }

2019-05-02 15:48:33.567 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "29.400000000000002", "dataCost": "2.1" }

2019-05-02 15:48:34.567 INFO 1 --- [container-0-C-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "5.2", "dataCost": "20.200000000000003" }

#### Cleaning Up

To delete the stream, we can use the label we created earlier, as follows:

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

To uninstall RabbitMQ, run the following command:

kubectl delete all -l app=rabbitmq

# Stream Processing with Data Flow and RabbitMQ

This section shows how to register stream applications with Data Flow, create a Stream DSL, and deploy the stream to Cloud Foundry, Kubernetes, and your local machine.

In the previous guides, we created Source, Processor and Sink streaming applications and deployed them as standalone applications on multiple platforms. In this guide, we register these applications with Data Flow, create a Stream DSL, and deploy the stream to Cloud Foundry, Kubernetes, and your local machine.

## Development

All the sample applications from the previous guide are available as maven and docker artifacts at the https://repo.spring.io Maven repository.

For the UsageDetailSender source, use one of the following:

maven://io.spring.dataflow.sample:usage-detail-sender-rabbit:0.0.1-SNAPSHOT

docker://springcloudstream/usage-detail-sender-rabbit:0.0.1-SNAPSHOT

For the UsageCostProcessor processor, use one of the following:

maven://io.spring.dataflow.sample:usage-cost-processor-rabbit:0.0.1-SNAPSHOT

docker://springcloudstream/usage-cost-processor-rabbit:0.0.1-SNAPSHOT

For the UsageCostLogger sink, use one of the following:

maven://io.spring.dataflow.sample:usage-cost-logger-rabbit:0.0.1-SNAPSHOT

docker://springcloudstream/usage-cost-logger-rabbit:0.0.1-SNAPSHOT

### The Data Flow Dashboard

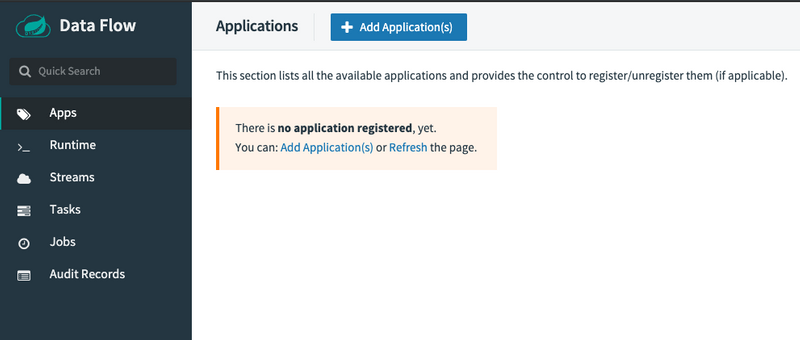
Assuming Data Flow is [installed](https://dataflow.spring.io/docs/installation/) and running on one of the supported platforms, open your browser at <data-flow-url>/dashboard. Here, <data-flow-url> depends on the platform. Consult the [installation guide](https://dataflow.spring.io/docs/installation/) to determining the base URL for your installation. If Data Flow is running on your local machine, go to <http://localhost:9393/dashboard>.

### Application Registration

Applications in Spring Cloud Data Flow are registered as named resources so that they may be referenced when you use the Data Flow DSL to configure and compose streaming pipelines. Registration associates a logical application name and type with a physical resource, which is given by a URI.

The URI conforms to a [schema](https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#spring-cloud-dataflow-register-stream-apps) and may represent a Maven artifact, a Docker image, or an actual http(s) or file URL. Data Flow defines some logical application types to indicate its role as a streaming component, a task, or a standalone application. For streaming applications, as you might expect, we use Source,Processor, and Sink types.

The Data Flow Dashboard lands on the Application Registration view, where we can register the source, processor, and sink applications, as the following image shows:



In this step, we register the applications we previously created. When you register an application, you provide its:

* Location URI (Maven, HTTP, Docker, file, and so on)
* Application version
* Application type (source, processor, or sink)
* Application name

The following table shows the applications we created in the previous guides:

| **App Name** | **App Type** | **App URI** |
| --- | --- | --- |
| usage-detail-sender | Source | maven://io.spring.dataflow.sample:usage-detail-sender-rabbit:0.0.1-SNAPSHOT |
| usage-cost-processor | Processor | maven://io.spring.dataflow.sample:usage-cost-processor-rabbit:0.0.1-SNAPSHOT |
| usage-cost-logger | Sink | maven://io.spring.dataflow.sample:usage-cost-logger-rabbit:0.0.1-SNAPSHOT |

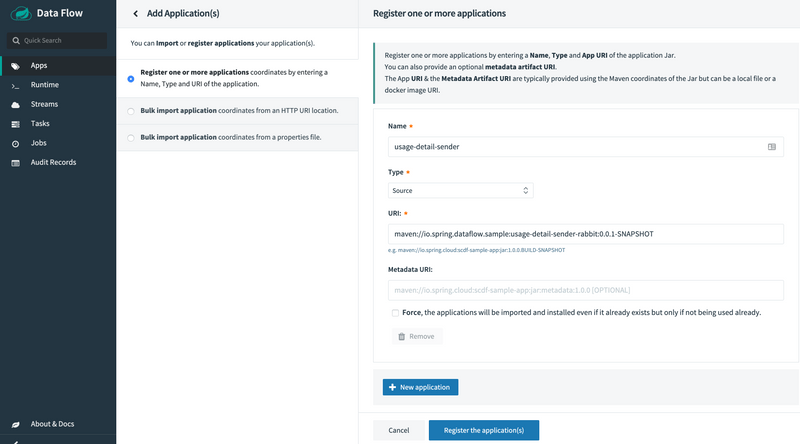
If you run the Spring Cloud Data Flow server on the Docker environment, make sure that your application artifact URIs are accessible. For instance, you may not be able to access file:/ from SCDF or Skipper Docker containers unless you have made the application locations be accessible. We recommend using http://, maven:// or docker:// for application URIs.

For this example, assume you run Spring Cloud Data Flow and Skipper servers on your local development environment.

You can register the UsageDetailSender source application. To do so:

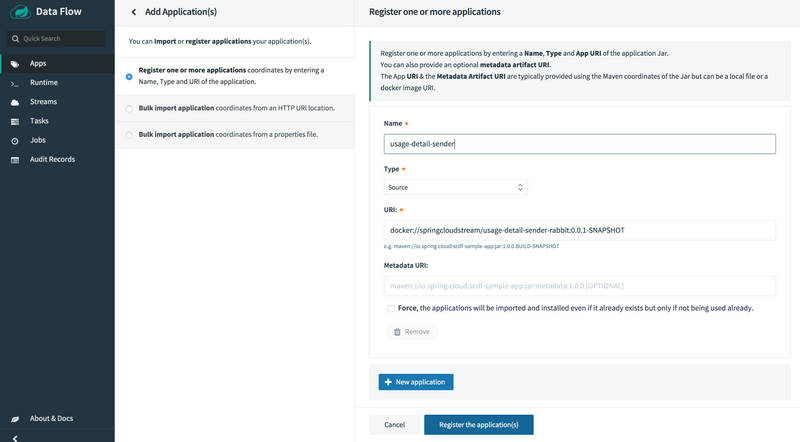
1. From the Applications view, select Add Application(s). This shows a view that lets you register applications.
2. Register the maven artifact of the UsageDetailSender application named usage-detail-sender, as the following image shows:

*(uri = maven://io.spring.dataflow.sample:usage-detail-sender-rabbit:0.0.1-SNAPSHOT)*



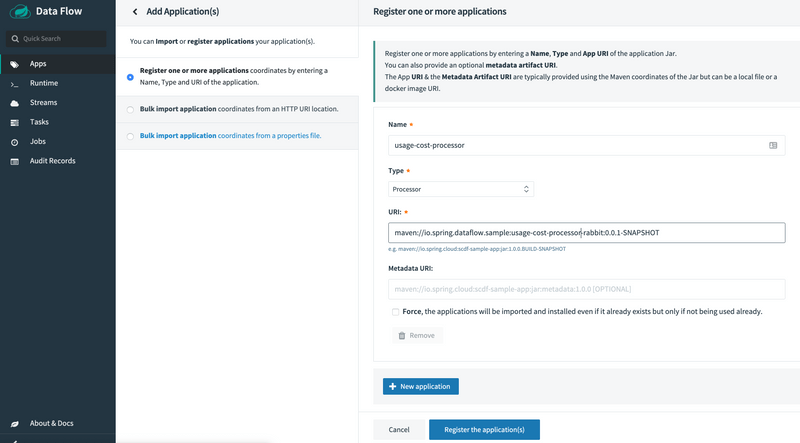
If you use a docker artifact, then register it as the following image shows:

*(uri = docker://springcloudstream/usage-detail-sender-rabbit:0.0.1-SNAPSHOT)*



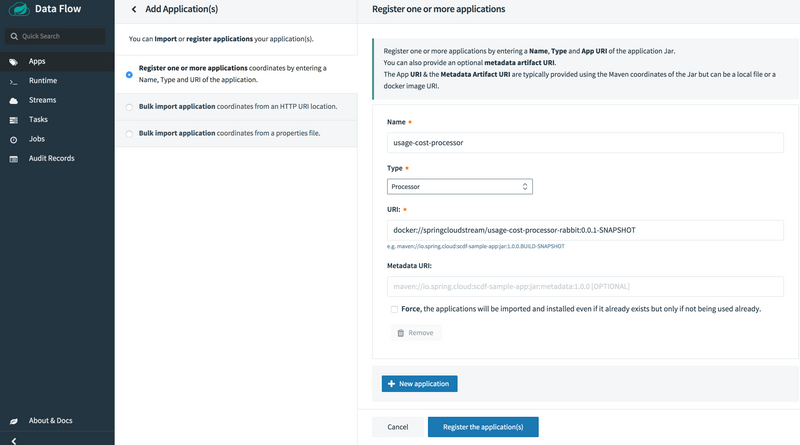
1. Select Register one or more applications and enter the name, type, and URI for the source application.
2. Click on New application to display another instance of the form to enter the values for the processor.
3. Register the maven artifact of the UsageCostProcessor processor application named usage-cost-processor, as the following image shows:

*(uri = maven://io.spring.dataflow.sample:usage-cost-processor-rabbit:0.0.1-SNAPSHOT)*



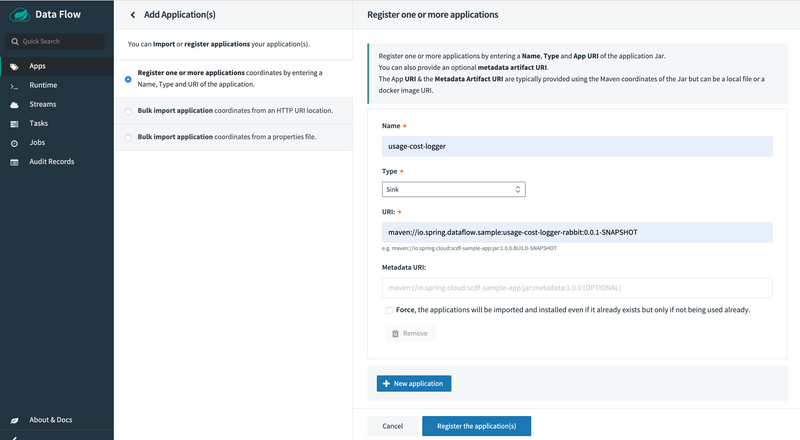
If you use a docker artifact, then register it, as the following image shows:

*(uri = docker://springcloudstream/usage-cost-processor-rabbit:0.0.1-SNAPSHOT)*



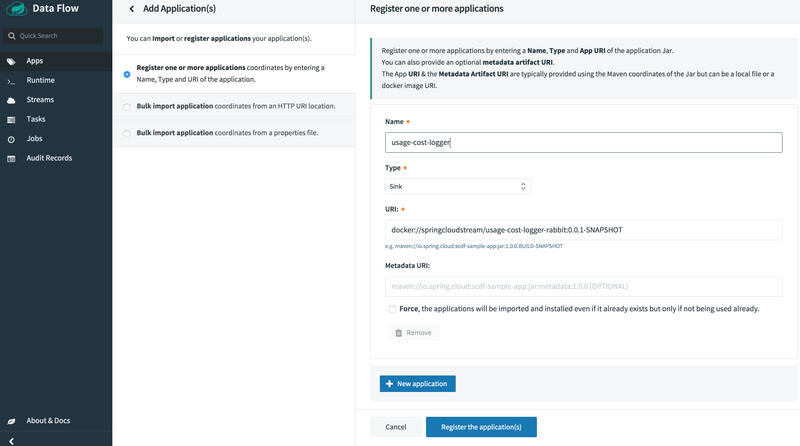
1. Register the maven artifact of the UsageCostLogger sink application named usage-cost-logger, as the following image shows

*(uri = maven://io.spring.dataflow.sample:usage-cost-logger-rabbit:0.0.1-SNAPSHOT)*

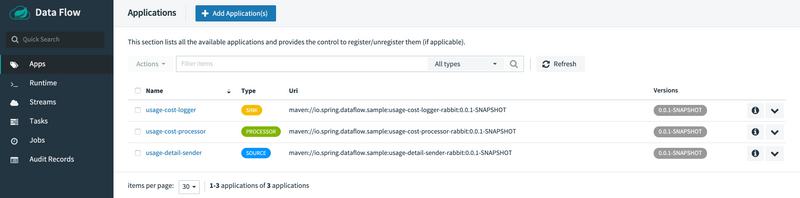


If you use a docker artifact, then register it, as the following image shows:

*(uri = docker://springcloudstream/usage-cost-logger-rabbit:0.0.1-SNAPSHOT)*



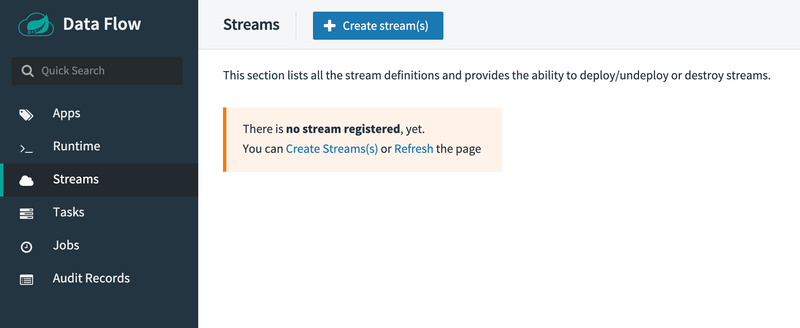
1. Click on Register the application(s) to complete the registration. Doing so takes you back to the Applications view, which lists your applications. The following image shows an example:



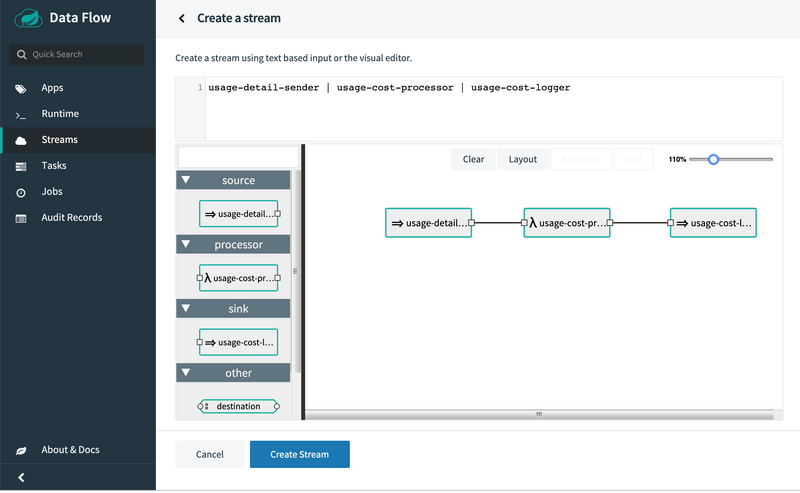
### Creating the Stream Definition

To create the stream definition:

1. Select Streams from the left navigation bar. This shows the main Streams view, as the following image shows:



1. Select Create stream(s) to display a graphical editor to create the stream definition, as the following image shows:



You can see the Source, Processor and Sink applications, as registered above, in the left panel.

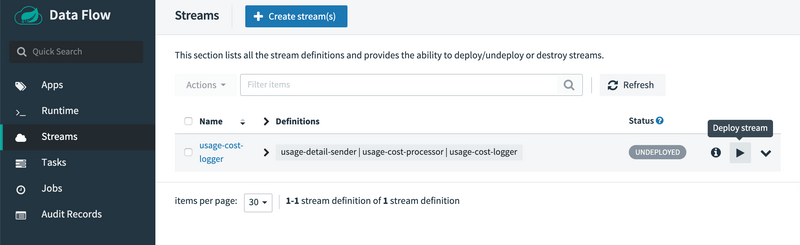
1. Drag and drop each application to the canvas and then use the handles to connect them together. Notice the equivalent Data Flow DSL definition in the top text panel.
2. Click Create Stream.

You can type the name of the stream usage-cost-logger when creating the stream.

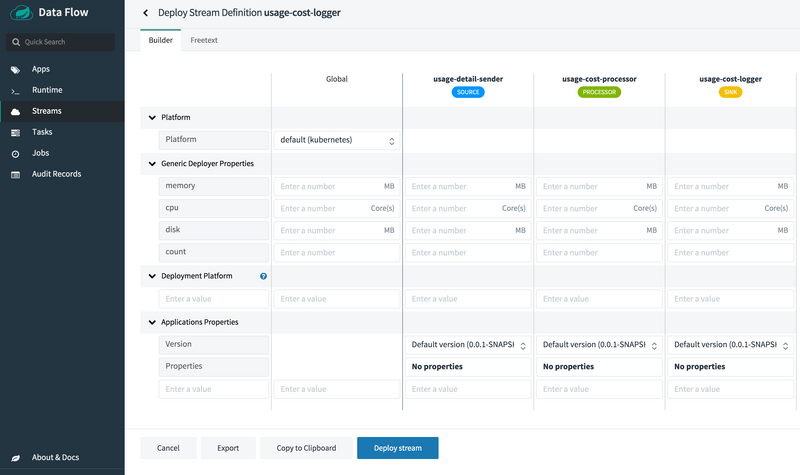
## Deployment

To deploy your stream,

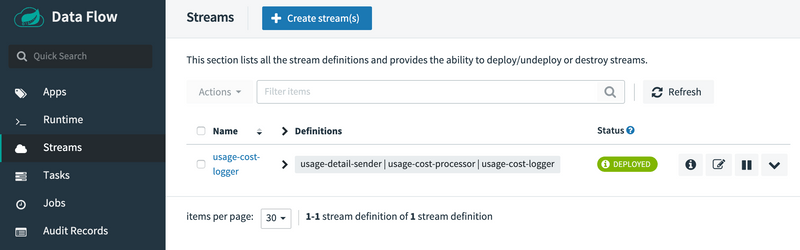
1. Click on the arrow head icon to deploy the stream. Doing so takes you to the Deploy Stream page, where you may enter additional deployment properties.
2. Select Deploy Stream, as the following image shows:



1. When deploying the stream, choose the target platform accounts from local, Kubernetes, or Cloud Foundry. This is based on the Spring Cloud Skipper server deployer platform account setup.



When all the applications are running, the stream is successfully deployed.



The preceding process is basically the same for all platforms. The following sections addresses platform-specific details for deploying on Data Flow on Local, Cloud Foundry, and Kubernetes.

### Local

If you deploy the stream on the local environment, you need to set a unique value for the server.port application property for each application so that they can use different ports on local.

Once the stream is deployed on the local development environment, you can look at the runtime applications by using the dashboard's runtime page or by using the SCDF shell command, runtime apps. The runtime applications show information about where each application runs in the local environment and their log files locations.

If you run SCDF on Docker, to access the log files of the streaming applications, you can run the following command (shown with its output):

docker exec <stream-application-docker-container-id> tail -f <stream-application-log-file>

2019-04-19 22:16:04.864 INFO 95238 --- [container-0-C-1] c.e.demo.UsageCostLoggerApplication : {"userId": "Mark", "callCost": "0.17", "dataCost": "0.32800000000000007" }

2019-04-19 22:16:04.872 INFO 95238 --- [container-0-C-1] c.e.demo.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "0.20800000000000002", "dataCost": "0.298" }

2019-04-19 22:16:04.872 INFO 95238 --- [container-0-C-1] c.e.demo.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "0.175", "dataCost": "0.16150000000000003" }

2019-04-19 22:16:04.872 INFO 95238 --- [container-0-C-1] c.e.demo.UsageCostLoggerApplication : {"userId": "Glenn", "callCost": "0.145", "dataCost": "0.269" }

2019-04-19 22:16:05.256 INFO 95238 --- [container-0-C-1] c.e.demo.UsageCostLoggerApplication : {"userId": "Ilaya", "callCost": "0.083", "dataCost": "0.23800000000000002" }

2019-04-19 22:16:06.257 INFO 95238 --- [container-0-C-1] c.e.demo.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "0.251", "dataCost": "0.026500000000000003" }

2019-04-19 22:16:07.264 INFO 95238 --- [container-0-C-1] c.e.demo.UsageCostLoggerApplication : {"userId": "Janne", "callCost": "0.15100000000000002", "dataCost": "0.08700000000000001" }

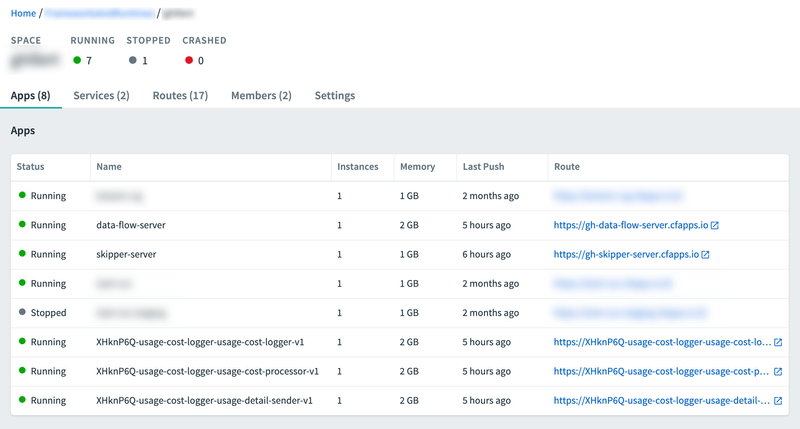
2019-04-19 22:16:08.263 INFO 95238 --- [container-0-C-1] c.e.demo.UsageCostLoggerApplication : {"userId": "Sabby", "callCost": "0.10100000000000002", "dataCost": "0.33" }

2019-04

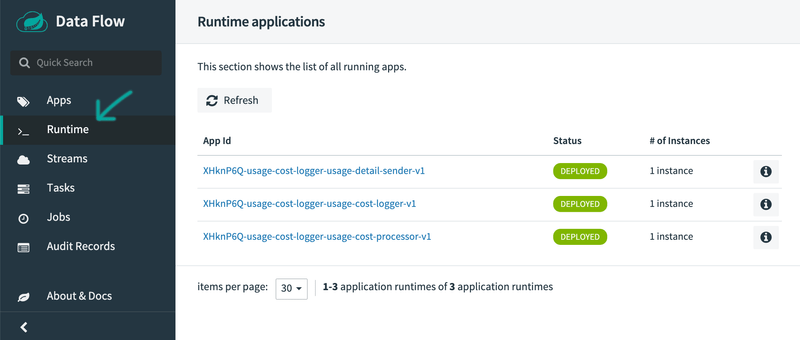
### Cloud Foundry

Before registering and deploying stream applications to Cloud Foundry by using the instructions shown earlier, you should ensure that you have an instance of Spring Cloud Data Flow running on Cloud Foundry. Follow the Cloud Foundry [installation guide](https://dataflow.spring.io/docs/installation/cloudfoundry/cf-cli/) for reference.

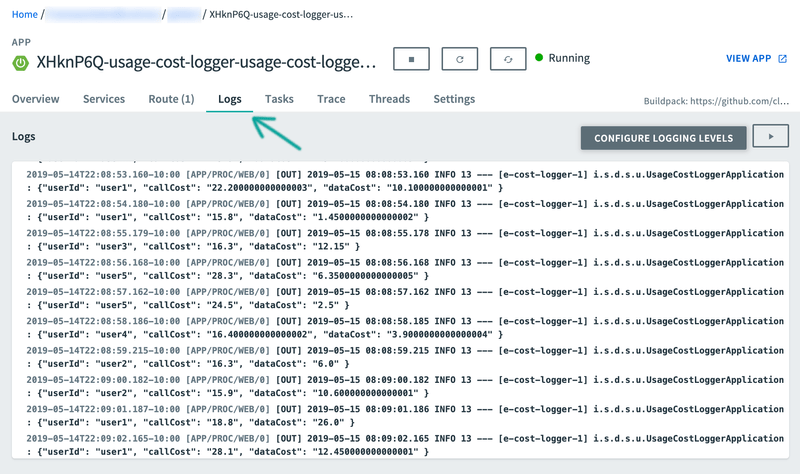
Once you have followed the steps shown earlier in this chapter and have registered the applications as well as deployed the stream, you can see the successfully deployed applications in your in your Organd Space in Cloud Foundry, as the following image shows:



You can access the runtime information of your stream applications in the Spring Cloud Data Flow dashboard as well. To do so, click the Runtime button in the left navigation:



Besides verifying the runtime status of your stream, you should also verify the logging output produced by the usage-cost-logger sink. In Cloud Foundry Apps Manager, click the Logs tab of the usage-cost-logger sink application. The logging statements should look like the following:



### Kubernetes

Once you have the Spring Cloud Data Flow server running in Kubernetes (by using the instructions from the [installation guide](https://dataflow.spring.io/docs/installation/kubernetes/)), you can:

* Register the stream applications
* Create, deploy, and manage streams

#### Registering Applications with Spring Cloud Data Flow server

The Kubernetes environment requires the application artifacts to be docker images.

For the UsageDetailSender source, use the following:

docker://springcloudstream/usage-detail-sender-rabbit:0.0.1-SNAPSHOT

For the UsageCostProcessor processor, use the following:

docker://springcloudstream/usage-cost-processor-rabbit:0.0.1-SNAPSHOT

For the UsageCostLogger sink, use the following:

docker://springcloudstream/usage-cost-logger-rabbit:0.0.1-SNAPSHOT

You can register these applications as described in the application registration step [described earlier](https://dataflow.spring.io/docs/stream-developer-guides/streams/data-flow-stream/#application-registration).

#### Stream Deployment

Once you have registered the applications, you can deploy the stream per the instructions from the stream deployment section [above](https://dataflow.spring.io/docs/stream-developer-guides/streams/data-flow-stream/#deployment).

##### Listing the Pods

To lists the pods (including the server components and the streaming applications), run the following command (shown with its output):

kubectl get pods

NAME READY STATUS RESTARTS AGE

scdf-release-data-flow-server-795c77b85c-tqdtx 1/1 Running 0 36m

scdf-release-data-flow-skipper-85b6568d6b-2jgcv 1/1 Running 0 36m

scdf-release-mysql-744757b689-tsnnz 1/1 Running 0 36m

scdf-release-rabbitmq-5fb7f7f644-878pz 1/1 Running 0 36m

usage-cost-logger-usage-cost-logger-v1-568599d459-hk9b6 1/1 Running 0 2m41s

usage-cost-logger-usage-cost-processor-v1-79745cf97d-dwjpw 1/1 Running 0 2m42s

usage-cost-logger-usage-detail-sender-v1-6cd7d9d9b8-m2qf6 1/1 Running 0 2m41s

##### Verifying the Logs

To be sure the steps in the previous sections have worked correctly, you should verify the logs. The following example (shown with its output) shows how to make sure that the values you expect appear in the logs:

kubectl logs -f usage-cost-logger-usage-cost-logger-v1-568599d459-hk9b6

2019-05-17 17:53:44.189 INFO 1 --- [e-cost-logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user2", "callCost": "0.7000000000000001", "dataCost": "23.950000000000003" }

2019-05-17 17:53:45.190 INFO 1 --- [e-cost-logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user4", "callCost": "2.9000000000000004", "dataCost": "10.65" }

2019-05-17 17:53:46.190 INFO 1 --- [e-cost-logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user3", "callCost": "5.2", "dataCost": "28.85" }

2019-05-17 17:53:47.192 INFO 1 --- [e-cost-logger-1] i.s.d.s.u.UsageCostLoggerApplication : {"userId": "user4", "callCost": "1.7000000000000002", "dataCost": "30.35" }

## Comparison with Standalone Deployment

In this section, we deployed the stream byusing Spring Cloud Data Flow with the stream DSL:

usage-detail-sender | usage-cost-processor | usage-cost-logger

When these three applications are deployed as standalone applications, you need to set the binding properties that connect the applications to make them into a stream.

Instead, Spring Cloud Data Flow lets you deploy all three streaming applications as a single stream by taking care of the plumbing of one application to the other to form the data flow.