Spring for Apache Kafka Deep Dive – Part 3: Apache Kafka and Spring Cloud Data Flow

Ilayaperumal Gopinathan
May 30, 2019

Following part 1 and part 2 of the Spring for Apache Kafka Deep Dive blog series, here in part 3 we will discuss another project from the Spring team: Spring Cloud Data Flow, which focuses on enabling developers to easily develop, deploy, and orchestrate event streaming pipelines based on Apache Kafka®. As a continuation from the previous blog series, this blog post explains how Spring Cloud Data Flow helps you gain developer productivity and manage Apache-Kafka-based event streaming application development.

We will cover the following in this post:

- Overview of the Spring Cloud Data Flow ecosystem
- How to develop, deploy, and orchestrate event streaming pipelines and applications using Spring Cloud Data Flow from the foundations laid by the previous two Spring for Apache Kafka Deep Dive blog series

Spring Cloud Data Flow ecosystem

Spring Cloud Data Flow is a toolkit for designing, developing, and continuously delivering data pipelines. It provides support for centrally managing event streaming application development right from design to deployment in production. In Spring Cloud Data Flow, the data pipelines can be a composition of either *event streaming* (real-time and long-running) or task/batch (short-lived) data-intensive applications. There are a variety of ways to interact with Spring Cloud Data Flow:

- Dashboard GUI
- Command Line Shell
- Stream Java DSL (domain-specific language)
- RESTful APIs via curl, etc.

To orchestrate the deployment of event streaming pipelines to platforms such as Cloud Foundry (CF) and Kubernetes (K8s), Spring Cloud Data Flow delegates the application lifecycle operations (deploy, update, rollback) to another server component named Spring Cloud Skipper. While the event stream pipeline deployment is handled by Spring Cloud Skipper, the deployment of short-lived (task/batch) data pipelines to target platforms is managed by Spring Cloud Data Flow itself.

Both the Spring Cloud Data Flow and Spring Cloud Skipper runtimes are configured to provide authentication and authorization via OAuth 2.0 and OpenID Connect. Spring Cloud Data Flow helps monitor the event streaming applications using Micrometer-based integration and provides Grafana dashboards that you can install and customize.

Developing event streaming applications

In Spring Cloud Data Flow, the event streaming pipelines are typically comprised of Spring Cloud Stream applications, though any custom-built applications can fit in the pipeline. A developer can directly use or extend any of the out-of-the-box utility event streaming applications to cover common use cases or write a custom application using Spring Cloud Stream.

All the out-of-the-box event streaming applications are:

- Available as Apache Maven artifacts or Docker images
- Built with RabbitMQ or the Apache Kafka Spring Cloud Stream binder
- Built with Prometheus and InfluxDB monitoring systems

The out-of-the-box applications are similar to Kafka Connect applications except they use the Spring Cloud Stream framework for integration and plumbing.

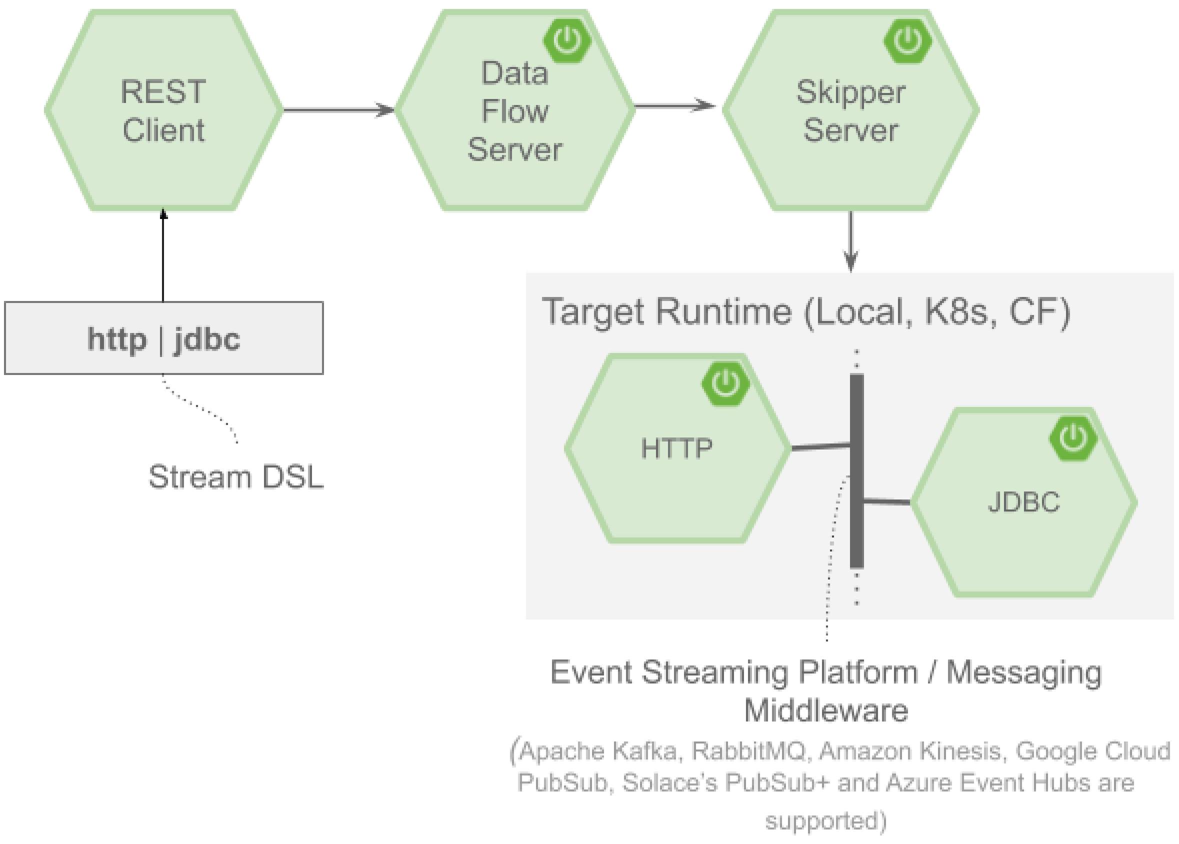
To build an event streaming pipeline, Spring Cloud Data Flow provides a set of application types:

- A `source` represents the first step in the data pipeline, a producer that extracts data from the external systems like databases, filesystem, FTP servers, IoT devices, etc.
- A `processor` represents an application that can consume from an upstream producer (a source or a processor), perform the business operation on the consumed data and emit the processed data for downstream consumption
- A `sink` represents the final stage in the data pipeline, which can write the consumed data to external systems like Cassandra, PostgreSQL, Amazon S3, etc.

It is important to note that in Spring Cloud Data Flow, the event streaming data pipeline is *linear* by default. This means that each application in the pipeline communicates with another using a single destination (for instance, a Kafka topic), with data flowing from producer to consumer linearly. However, there are use cases where a streaming pipeline is *non-linear* and can have *multiple* inputs and outputs—a typical setting for Kafka Streams applications.

It is also possible to have non-Spring-Cloud-Stream applications (Kafka Connect applications, Polygot applications, etc.) in the event streaming data pipeline.

Spring Cloud Data Flow supports these cases using the Stream Application DSL and the application type `app` is used to highlight such applications.


REST Client
Data Flow Server
Skipper Server
http | jdbc
Stream DSL
Target Runtime (Local, K8s, CF)
HTTP
JDBC
Event Streaming Platform / Messaging Middleware
(Apache Kafka, RabbitMQ, Amazon Kinesis, Google Cloud PubSub, Solace's PubSub+ and Azure Event Hubs are supported)

The above visual illustrates an event streaming pipeline composed of two applications, in which `http` and `jdbc` can be deployed using Spring Cloud Data Flow. Both the applications are built using the Spring Cloud Stream framework, which we covered in part 2, and are available in a public Maven repository/Docker Hub. The pipe symbol `|` (i.e., `http | jdbc`) in the Stream DSL represents an event streaming platform such as Apache Kafka, configured for the communication of event streaming applications.

The event streaming platform or the messaging middleware provides loose coupling between the producer `http` source and the consumer `jdbc` sink applications of the stream. This loose coupling is critical for the cloud-native deployment model since the applications inside the pipeline can independently evolve, scale, or perform a rolling upgrade without impacting the upstream producers or the downstream consumers. With the wide range of offerings on the streaming platform, Apache Kafka resonates well when Spring Cloud Data Flow uses it for the event streaming applications.

Spring Cloud Data Flow environment setup

The Spring Cloud Data Flow website has getting started guides for Local, Kubernetes, and Cloud Foundry. For this blog, let’s use Docker to run this setup locally. To get started, you need to download the Docker Compose file from the Spring Cloud Data Flow GitHub repo.

This Docker Compose configuration has:

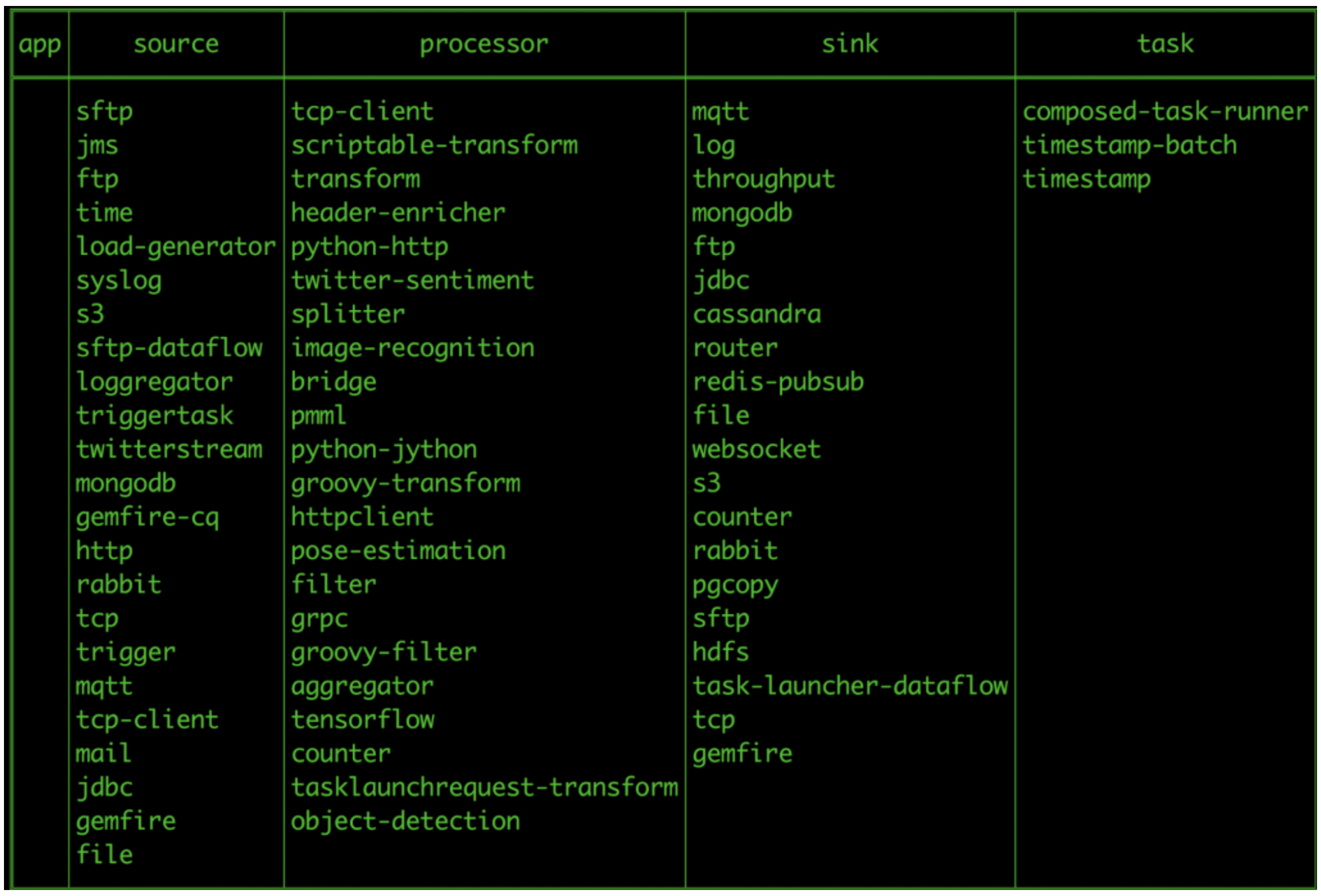
- Apache Kafka
- Spring Cloud Data Flow server
- Spring Cloud Skipper server
- Prometheus (application metrics and monitoring)
- Grafana (data visualization)
- Automatic registration of out-of-the-box event streaming applications

Since all the above components will be running along with the event streaming applications in our Docker environment, please make sure to allocate a minimum of 6GB for your Docker setup.

Next, install docker-compose and run the following:

```
export DATAFLOW_VERSION=2.1.0.RELEASE
export SKIPPER_VERSION=2.0.2.RELEASE
docker-compose up
```

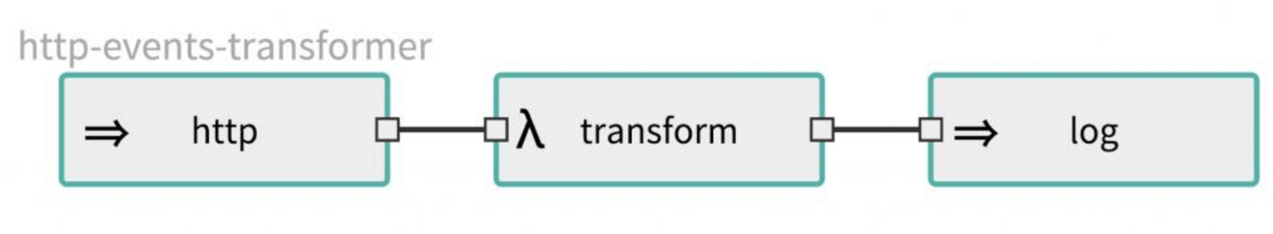
When all the components are started, the Spring Cloud Data Flow dashboard can be accessed at `http://localhost:9393/dashboard` with the following out-of-the-box event streaming applications registered:



app	source	processor	sink	task
	sftp	tcp-client	mqtt	composed-task-runner
	jms	scriptable-transform	log	timestamp-batch
	ftp	transform	throughput	timestamp
	time	header-enricher	mongodb	
	load-generator	python-http	ftp	
	syslog	twitter-sentiment	jdbc	
	s3	splitter	cassandra	
	sftp-dataflow	image-recognition	router	
	loggregator	bridge	redis-pubsub	
	triggertask	pmml	file	
	twitterstream	python-jython	websocket	
	mongodb	groovy-transform	s3	
	gemfire-cq	httpclient	counter	
	http	pose-estimation	rabbit	
	rabbit	filter	pgcopy	
	tcp	grpc	sftp	
	trigger	groovy-filter	hdfs	
	mqtt	aggregator	task-launcher-dataflow	
	tcp-client	tensorflow	tcp	
	mail	counter	gemfire	
	jdbc	tasklaunchrequest-transform		
	gemfire	object-detection		
	file			

Creating an event streaming pipeline

Let's create an event pipeline in Spring Cloud Data Flow using the same `uppercase-processor` and `log-sink` applications introduced from the previous blog post. With these applications, let's create a simple stream `http-events-transformer` as follows:



1. An `http` source listens to an HTTP web endpoint for incoming data and publishes them to a Kafka topic.
2. A `transform` processor consumes the events from the Kafka topic where the `http` source publishes data in step 1. It then applies the transformation logic—converting the incoming payload to uppercase, and publishes the processed data to another Kafka topic.
3. A `log` sink consumes the events from the `transform` processor's output Kafka topic in step 2 and its responsibility is simply to display the result in the logs.

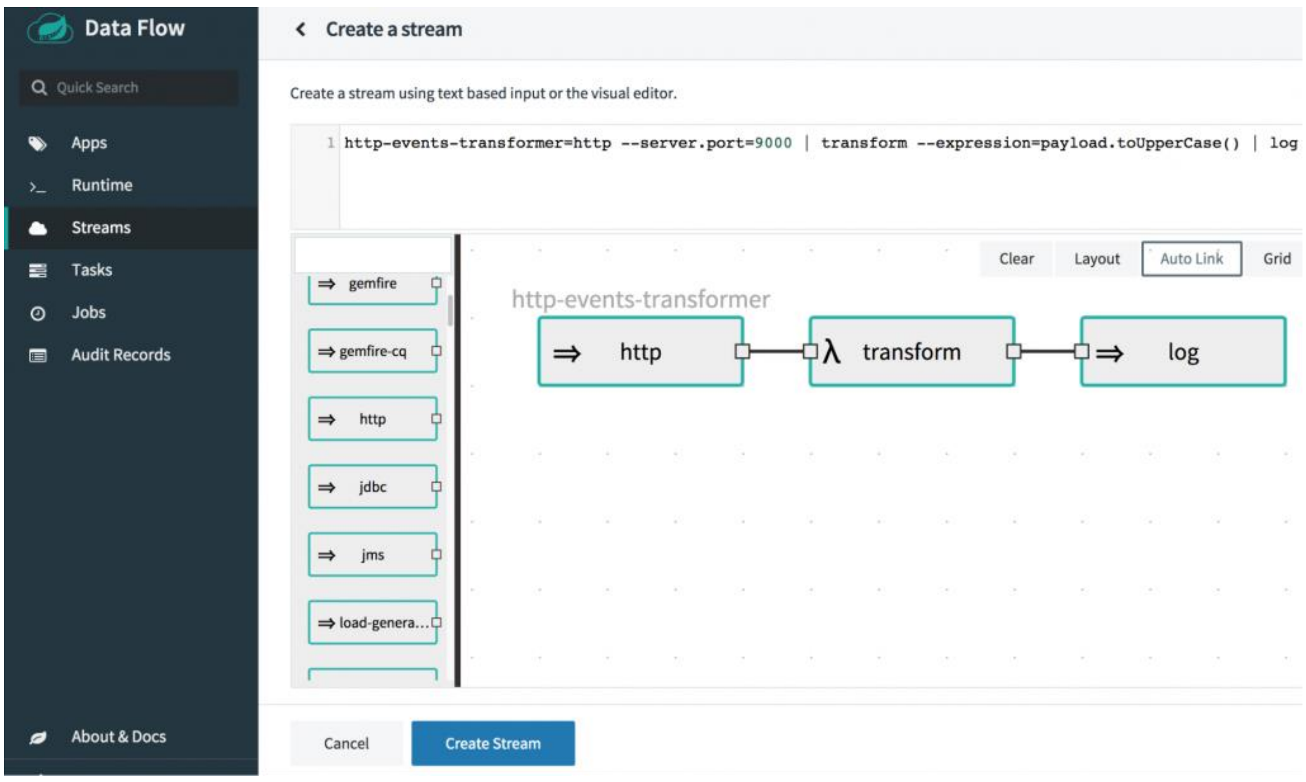
The Stream DSL syntax in Spring Cloud Data Flow would look like this:

```
http | transform | log
```

From the "Streams" page of the Spring Cloud Data Flow dashboard, you can create a new stream as follows.

Type in the following Stream DSL text:

```
http-events-transformer=http --server.port=9000 | transform --expression=payload.toUpperCase() | log
```



Create a stream using text based input or the visual editor.

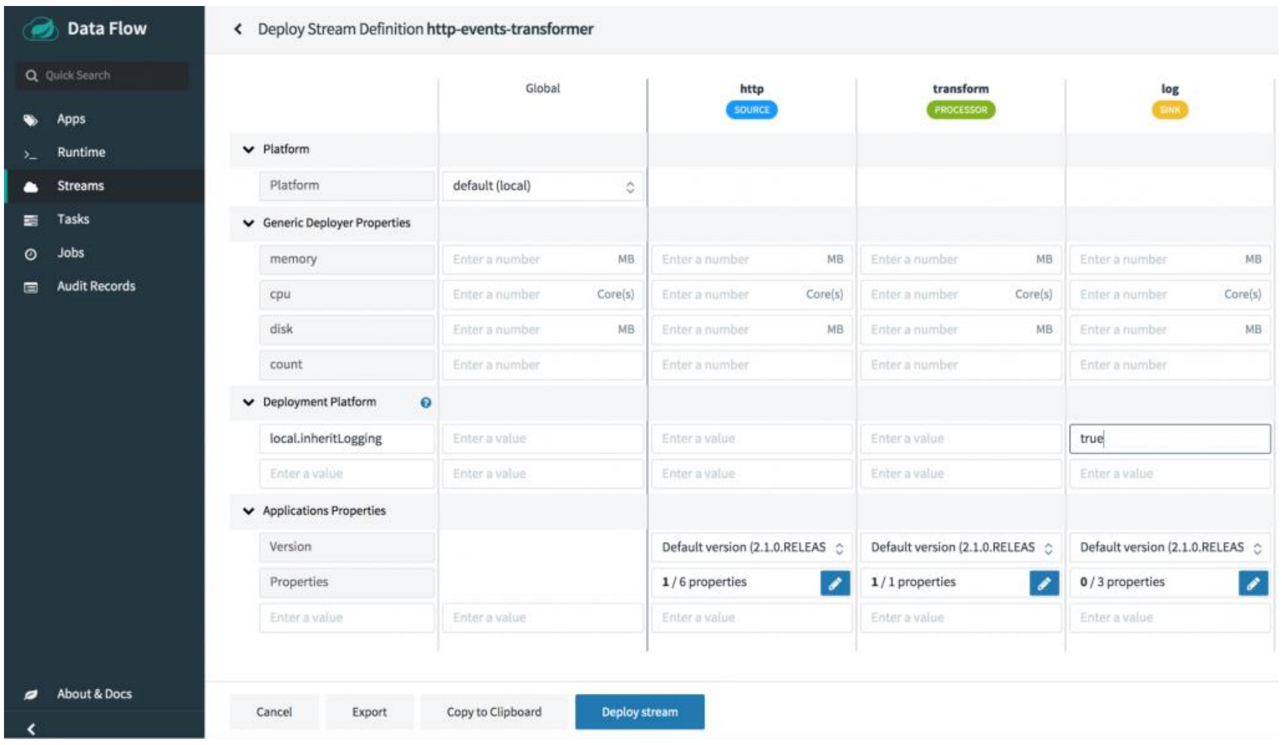
```
http-events-transformer=http --server.port=9000 | transform --expression=payload.toUpperCase() | log
```

When deploying the stream, there are two types of properties that can be overridden:

1. Application level properties, which are the configuration properties for the Spring Cloud stream application
2. Deployer properties for the target platform, such as Local, Kubernetes, or Cloud Foundry

From Spring Cloud Data Flow dashboard's "Streams" page, choose the stream `http-events-transformer` and click "deploy."

When deploying the stream, make sure to select the platform as `local` to deploy the stream in the `local` environment. Set the local platform deployer property `inheritLogging` to `true` for the `log` application (as indicated in the screenshot below), which lets you copy the log files of the `log` application into the Spring Cloud Skipper server log. Having the application logs under the Skipper server log makes the demo easier.



Deploy Stream Definition **http-events-transformer**

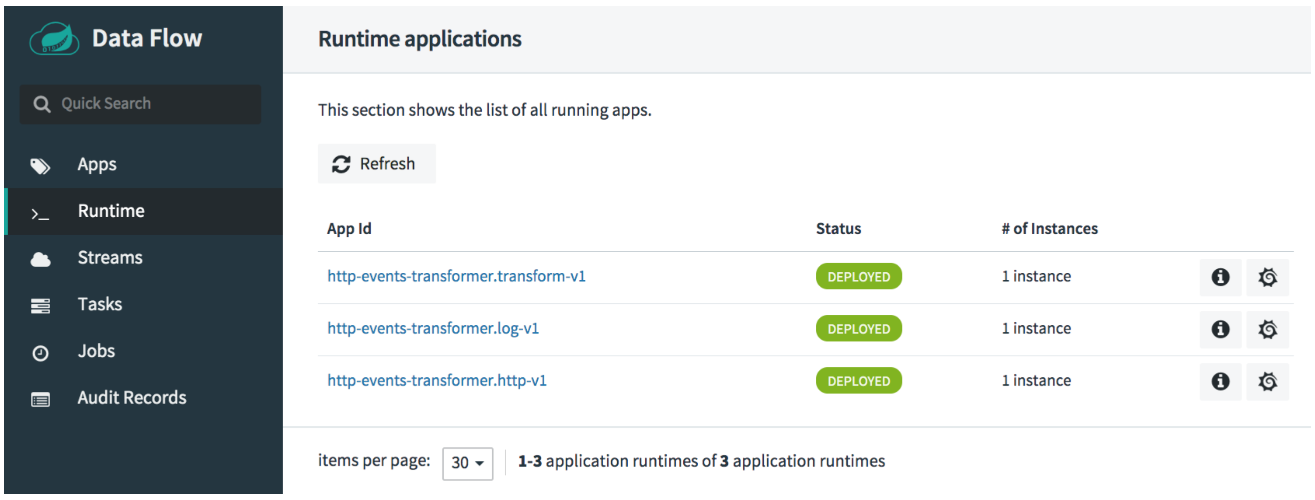
	Global	http SOURCE	transform PROCESSOR	log SINK
Platform				
Platform	default (local)			
Generic Deployer Properties				
memory	Enter a number MB	Enter a number MB	Enter a number MB	Enter a number MB
cpu	Enter a number Core(s)	Enter a number Core(s)	Enter a number Core(s)	Enter a number Core(s)
disk	Enter a number MB	Enter a number MB	Enter a number MB	Enter a number MB
count	Enter a number	Enter a number	Enter a number	Enter a number
Deployment Platform				
local.inheritLogging	Enter a value	Enter a value	Enter a value	true
Enter a value	Enter a value	Enter a value	Enter a value	Enter a value
Applications Properties				
Version		Default version (2.1.0.RELEAS	Default version (2.1.0.RELEAS	Default version (2.1.0.RELEAS
Properties		1 / 6 properties	1 / 1 properties	0 / 3 properties
Enter a value	Enter a value	Enter a value	Enter a value	Enter a value

When the stream is deployed, the individual applications `http`, `transform` and `log` are retrieved, and the deployment requests for each application are sent to the targeted platforms (i.e., Local, Kubernetes, and CloudFoundry) by Spring Cloud Data Flow. Likewise, when the applications bootstrap, the following Kafka topics are created automatically by the Spring Cloud Stream framework, which is how these applications come together at runtime as a coherent event streaming pipeline.

1. `http-events-transformer.http` (a topic that connects the `http` source's output to the `transform` processor's input)
2. `http-events-transformer.transform` (a topic that connects the `transform` processor's output to the `log` sink's input)

The Kafka topic names are derived by Spring Cloud Data Flow based on the `stream` and `application` naming conventions. You can override the names by using the appropriate Spring Cloud Stream binding properties.

To view all the runtime stream applications, see the "Runtime" page:



Runtime applications

This section shows the list of all running apps.

App Id	Status	# of Instances
http-events-transformer.transform-v1	DEPLOYED	1 instance
http-events-transformer.log-v1	DEPLOYED	1 instance
http-events-transformer.http-v1	DEPLOYED	1 instance

items per page: 30 | 1-3 application runtimes of 3 application runtimes

Once the stream is successfully deployed, the HTTP application is ready to accept the data at `http://localhost:9000`. Let’s post some test data to the `http` web endpoint:

```
curl -X POST http://localhost:9000 -d "spring" -H "Content-Type: text/plain"
```

Since we inherited the logs of the `log` application, the `log` application’s output in the Spring Cloud Skipper server log can be seen as:

```
log-sink                                  : SPRING
```

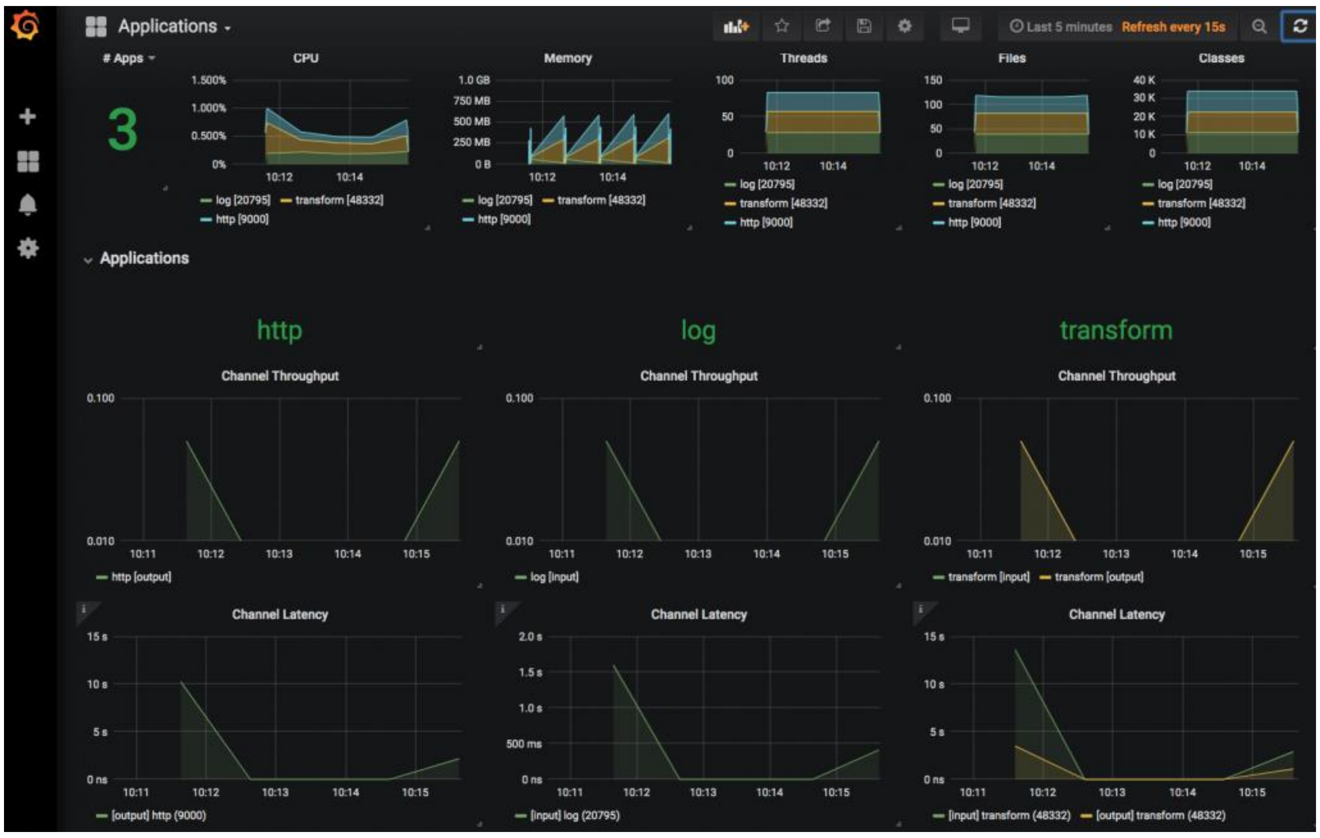
Debugging the streaming applications

You can debug the deployed applications at runtime. The debugging configuration varies based on the target platform. Refer to the documentation for debugging the deployed applications on Local, Kubernetes, and Cloud Foundry target environments. For debugging the application on a local development environment, it’s as simple as passing the `local` deployer property `debugPort`.

Monitoring the event streaming applications

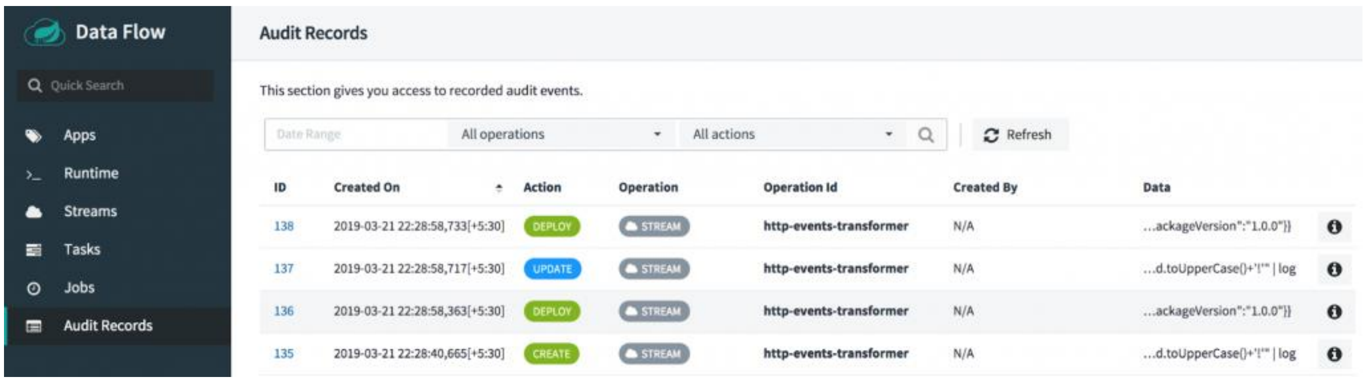
For the current setup, we used Prometheus-based application monitoring and set up a Grafana dashboard with the credentials `admin/admin` by default.

You can monitor the event stream deployment from the Grafana dashboard by clicking the “Grafana Dashboard” icon for the event stream `http-events-transformer` from the “Streams” page of the Spring Cloud Data Flow dashboard.



Auditing user operations

All the operations involved with the Spring Cloud Data Flow server are audited, and the audit records can be accessed from the "Audit Records" page in the Spring Cloud Data Flow dashboard.



Audit Records

This section gives you access to recorded audit events.

Date Range | All operations | All actions | Refresh

ID	Created On	Action	Operation	Operation Id	Created By	Data
138	2019-03-21 22:28:58,733[+5:30]	DEPLOY	STREAM	http-events-transformer	N/A	...ackageVersion":"1.0.0"}}
137	2019-03-21 22:28:58,717[+5:30]	UPDATE	STREAM	http-events-transformer	N/A	...d.toUpperCase()+'!'" | log
136	2019-03-21 22:28:58,363[+5:30]	DEPLOY	STREAM	http-events-transformer	N/A	...ackageVersion":"1.0.0"}}
135	2019-03-21 22:28:40,665[+5:30]	CREATE	STREAM	http-events-transformer	N/A	...d.toUpperCase()+'!'" | log

You can delete the stream by clicking the `Destroy Stream` option for `http-events-transformer` from "Streams" page.

For more detail on the event streaming application development and deployment, you can refer to the stream developer guide.

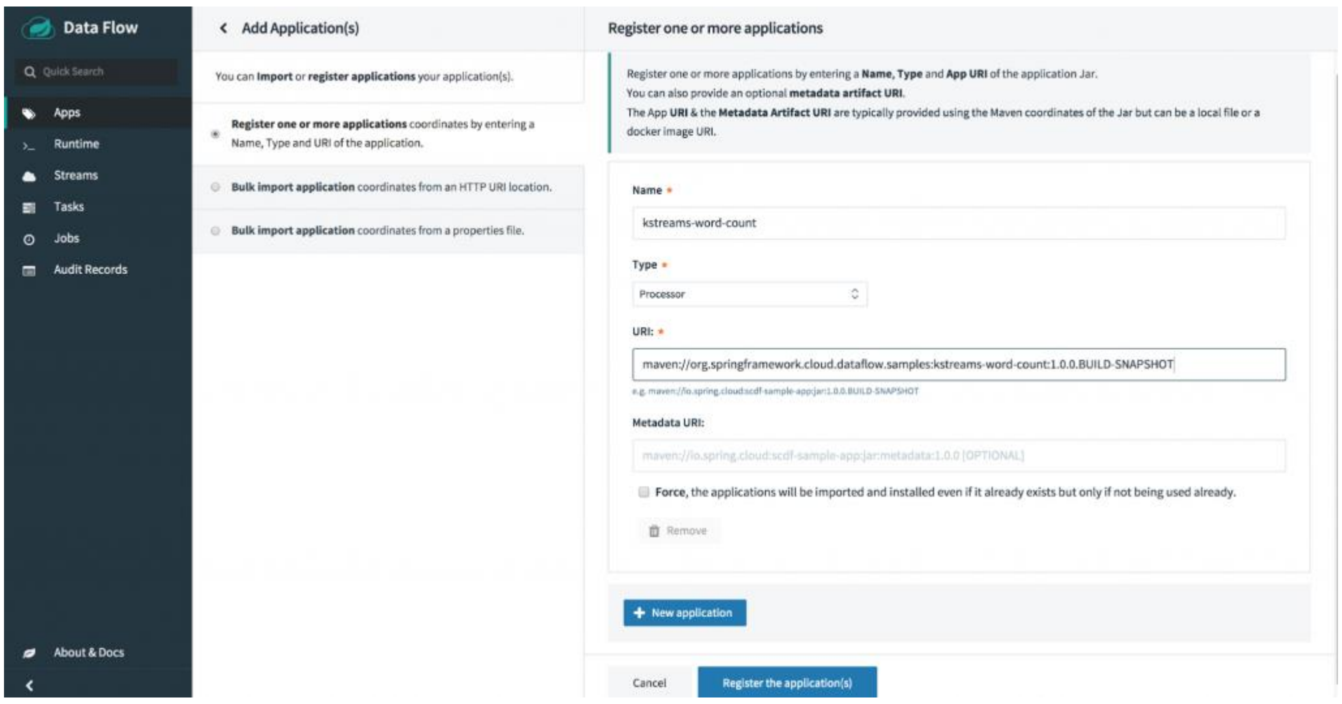
Developing an event streaming pipeline with a Kafka Streams application

When you have an event streaming pipeline that uses Kafka Streams applications, they can be used as `Processor` applications in the Spring Cloud Data Flow event streaming pipeline. In the following example, you will see how a Kafka Streams application can be registered as a Spring Cloud Data Flow `processor` application and subsequently used in an event streaming pipeline.

All the sample applications used in this blog are available on GitHub. The application kstreams-word-count is a Kafka Streams application built using the Spring Cloud Stream framework to count the incoming words within a given time window. This application is built and published into the Spring Maven repo.

From Spring Cloud Data Flow “Apps” page’s “Add Application(s),” you can register the `kstreams-word-count` application by choosing its application type as `Processor`, as well as its Maven URI:

```
maven://org.springframework.cloud.dataflow.samples:kstreams-word-
count:1.0.0.BUILD-SNAPSHOT
```


Data Flow
Quick Search
Apps
Runtime
Streams
Tasks
Jobs
Audit Records
About & Docs

Add Application(s)

You can **Import** or **register applications** your application(s).

Register one or more applications coordinates by entering a Name, Type and URI of the application.

Bulk import application coordinates from an HTTP URI location.

Bulk import application coordinates from a properties file.

Register one or more applications

Register one or more applications by entering a **Name, Type** and **App URI** of the application Jar.
You can also provide an optional **metadata artifact URI**.
The App **URI** & the **Metadata Artifact URI** are typically provided using the Maven coordinates of the Jar but can be a local file or a docker image URI.

Name *: kstreams-word-count

Type *: Processor

URI: *: maven://org.springframework.cloud.dataflow.samples:kstreams-word-count:1.0.0.BUILD-SNAPSHOT

e.g. maven://io.spring.cloud:scdf-sample-app:jar:1.0.0.BUILD-SNAPSHOT

Metadata URI: maven://io.spring.cloud:scdf-sample-app:jar:metadata:1.0.0 [OPTIONAL]

Force, the applications will be imported and installed even if it already exists but only if not being used already.

Remove

New application

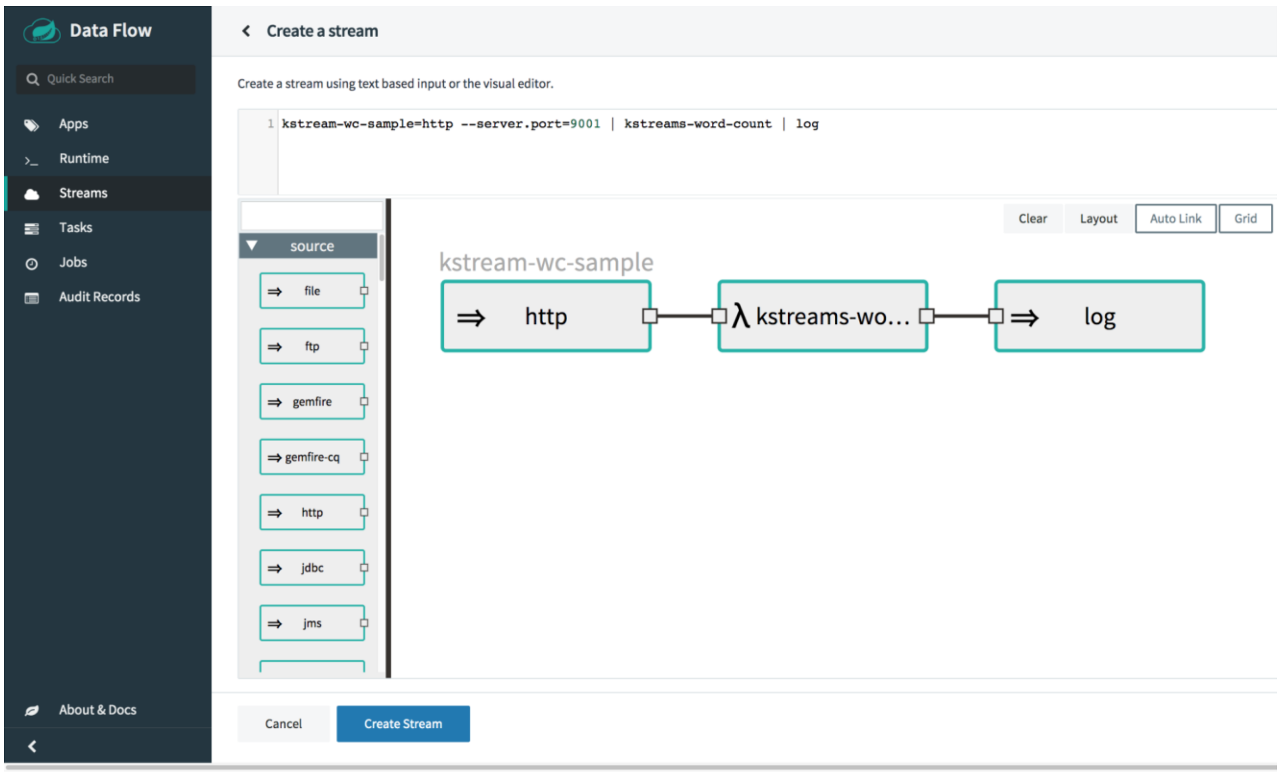
Cancel

Register the application(s)

Let’s use the out-of-the-box `http` source application, which listens at HTTP web endpoint `http://localhost:9001` for incoming data, and publishes the consumed data to the `kstream-word-count` processor registered in the step above. The Kafka Streams processor computes the word count by the time window, and its output is then propagated over to the out-of-the-box `log` application, which logs the result from the word count Kafka Streams processor.

From “Streams” page in the Spring Cloud Data Flow dashboard, create a stream with the Stream DSL:

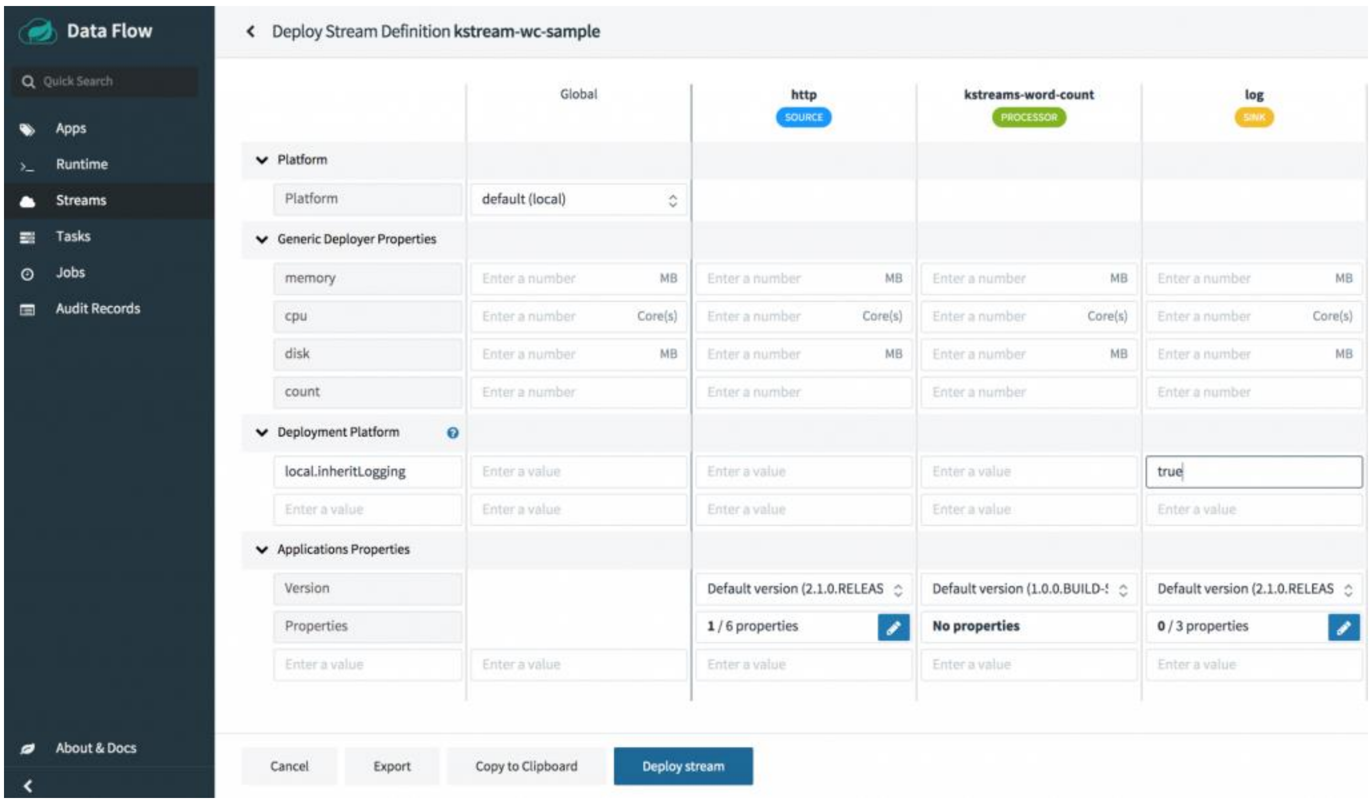
```
kstream-wc-sample=http --server.port=9001 | kstreams-word-count | log
```



Create a stream using text based input or the visual editor.

```
kstream-wc-sample=http --server.port=9001 | kstreams-word-count | log
```

Deploy the `kstream-wc-sample` stream from “Streams” page by specifying the platform as `local`. Also, specify the deployer property `local.inheritLogging` for the `log` application to `true`.



Deploy Stream Definition **kstream-wc-sample**

	Global	http SOURCE	kstreams-word-count PROCESSOR	log SINK
Platform				
Platform	default (local)			
Generic Deployer Properties				
memory	Enter a number MB	Enter a number MB	Enter a number MB	Enter a number MB
cpu	Enter a number Core(s)	Enter a number Core(s)	Enter a number Core(s)	Enter a number Core(s)
disk	Enter a number MB	Enter a number MB	Enter a number MB	Enter a number MB
count	Enter a number	Enter a number	Enter a number	Enter a number
Deployment Platform				
local.inheritLogging	Enter a value	Enter a value	Enter a value	true
Enter a value	Enter a value	Enter a value	Enter a value	Enter a value
Applications Properties				
Version		Default version (2.1.0.RELEAS	Default version (1.0.0.BUILD-!	Default version (2.1.0.RELEAS
Properties		1 / 6 properties	No properties	0 / 3 properties
Enter a value	Enter a value	Enter a value	Enter a value	Enter a value

When the stream is successfully deployed, all the `http`, `kstream-word-count`, and `log` are running as distributed applications, connected via specific Kafka topics configured in the event streaming pipeline.

Now, you can post some words for the Kafka Streams application to process:

```
curl -X POST http://localhost:9001 -H "Content-Type: text/plain" -d "Baby shark,
doo doo doo doo doo doo"
```

You can see the `log` application now has the following:

```
skipper              | 2019-03-25 09:53:37.228  INFO 66 --- [container-0-C-1]
log-sink                             : {"word":"baby","count":1,"start":"2019-
03-25T09:53:30.000+0000","end":"2019-03-25T09:54:00.000+0000"}
skipper              | 2019-03-25 09:53:37.229  INFO 66 --- [container-0-C-1]
log-sink                            : {"word":"shark","count":1,"start":"2019-
03-25T09:53:30.000+0000","end":"2019-03-25T09:54:00.000+0000"}
skipper              | 2019-03-25 09:53:37.234  INFO 66 --- [container-0-C-1]
log-sink                              : {"word":"doo","count":6,"start":"2019-
03-25T09:53:30.000+0000","end":"2019-03-25T09:54:00.000+0000"}
```

From the above sample, you can see how a Kafka Streams application can fit into an event streaming data pipeline. You have also seen how such an event streaming pipeline can be managed in Spring Cloud Data Flow. At this point, feel free to undeploy and delete the stream from the `kstream-wc-sample` stream page.

Conclusion

For event streaming application developers and data enthusiasts who use Apache Kafka, this blog provides a glimpse of how Spring Cloud Data Flow helps develop and deploy event streaming applications with all the essential features, such as ease of development and management, monitoring, and security.

Spring Cloud Data Flow offers a range of tools and automation to deploy and manage event streaming pipelines across cloud-native platforms. Part 4 of this series will present common event streaming topologies and continuous deployment patterns as a native set of primitives for event streaming applications in Spring Cloud Data Flow. Stay tuned!

Interested in more?

If you'd like to know more, you can refer to the Spring Cloud Data Flow website and reach out on GitHub.

You can also download the Confluent Platform to get started with the leading distribution of Apache Kafka.

Related articles

- Spring for Apache Kafka Deep Dive – Part 1: Error Handling, Message Conversion and Transaction Support
- Spring for Apache Kafka Deep Dive – Part 2: Apache Kafka and Spring Cloud Stream
- Spring for Apache Kafka Deep Dive – Part 4: Continuous Delivery of Event Streaming Pipelines
- How to Work with Apache Kafka in Your Spring Boot Application