Spring Cloud Data Flow Server for Cloud Foundry

1.0.0.M4

Sabby Anandan, Eric Bottard, Mark Fisher, Ilayaperumal Gopinathan,
Gunnar Hillert, Mark Pollack, Thomas Risberg, Marius Bogoevici, Josh Long

Table of Contents

Part I. Spring Cloud Data Flow for Cloud Foundry

This project provides support for orchestrating the deployment of Spring Cloud Stream applications to Cloud Foundry.

1. Spring Cloud Data Flow

Spring Cloud Data Flow is a cloud-native programming and operating model for composable data microservices on a structured platform. With Spring Cloud Data Flow, developers can create and orchestrate data pipelines for common use cases such as data ingest, real-time analytics, and data import/export.

The Spring Cloud Data Flow architecture consists of a server that deploys Streams. A future release will also support deploying Tasks. Streams are defined using a DSL or visually through the browser based designer UI. Streams are based on the Spring Cloud Stream programming model. The sections below describe more information about creating your own custom Streams.

For more details about the core architecture components and the supported features, please review Spring Cloud Data Flow's core reference guide. There're several samples available for reference.

2. Spring Cloud Stream

Spring Cloud Stream is a framework for building message-driven microservice applications. Spring Cloud Stream builds upon Spring Boot to create standalone, production-grade Spring applications, and uses Spring Integration to provide connectivity to message brokers. It provides opinionated configuration of middleware from several vendors, introducing the concepts of persistent publish-subscribe semantics, consumer groups, and partitions.

For more details about the core framework components and the supported features, please review Spring Cloud Stream's reference guide.

There's a rich ecosystem of Spring Cloud Stream Application-Starters that can be used either as standalone data microservice applications or in Spring Cloud Data Flow. For convenience, we have generated RabbitMQ and Apache Kafka variants of these application-starters that are available for use from Maven Repo and Docker Hub as maven artifacts and docker images, respectively.

Do you have a requirement to develop custom applications? No problem. Refer to this guide to create custom stream applications. There're several samples available for reference.

Part II. Architecture

3. Introduction

Spring Cloud Data Flow simplifies the development and deployment of applications focused on data processing use-cases. The major concepts of the architecture are Applications, the Data Flow Server, and the target runtime.

Applications are Long lived Stream applications where an unbounded amount of data is consumed or produced via messaging middleware.

Depending on the runtime, applications can be packaged in two ways

- Spring Boot uber-jar that is hosted in a maven repository, file, http or any other Spring resource implementation.
- Docker

The runtime is the place where applications execute. The target runtimes for applications are platforms that you may already be using for other application deployments.

The supported runtimes are

- Cloud Foundry
- Apache YARN
- Kubernetes
- Apache Mesos
- Local Server for development

There is a deployer Service Provider Interface (SPI) that enables you to extend Data Flow to deploy onto other runtimes, for example to support Hashicorp's Nomad or Docker Swarm. Contributions are welcome!

The component that is responsible for deploying applications to a runtime is the Data Flow Server. There is a Data Flow Server executable jar provided for each of the target runtimes. The Data Flow server is responsible for interpreting

- A stream DSL that describes the logical flow of data through multiple applications.
- A deployment manifest that describes the mapping of applications onto the runtime. For example, to set the initial number of instances, memory requirements, and data partitioning.

As an example, the DSL to describe the flow of data from an http source to an Apache Cassandra sink would be written as "http | cassandra". These names in the DSL are registered with the Data Flow Server and map onto application artifacts that can be hosted in Maven or Docker repositories. Many source, processor, and sink applications for common use-cases (e.g. jdbc, hdfs, http, router) are provided by the Spring Cloud Data Flow team. The pipe symbol represents the communication between the two applications via messaging middleware. The two messaging middleware brokers that are supported are

- Apache Kafka
- RabbitMQ

In the case of Kafka, when deploying the stream, the Data Flow server is responsible to create the topics that correspond to each pipe symbol and configure each application to produce or consume from the topics so the desired flow of data is achieved.

The interaction of the main components is shown below

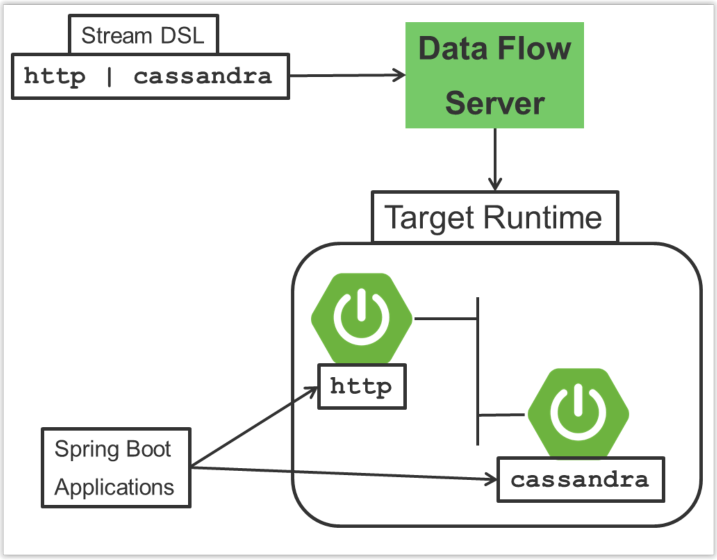

Figure 3.1. The Spring Cloud Data High Level Architecure

In this diagram a DSL description of a stream is POSTed to the Data Flow Server. Based on the mapping of DSL application names to Maven and Docker artifacts, the http source and cassandra sink application are deployed on the target runtime.

4. Microservice Architectural Style

The Data Flow Server deploys applications onto the target runtime that conform to the microservice architectural style. For example, a stream represents a high level application that consists of multiple small microservice applications each running in their own process. Each microservice application can be scaled up or down independent of the other and each has their own versioning lifecycle.

Both Streaming based microservice applications build upon Spring Boot as the foundational library. This gives all microservice applications functionality such as health checks, security, configurable logging, monitoring and management functionality, as well as executable JAR packaging.

It is important to emphasise that these microservice applications are ‘just apps’ that you can run by yourself using ‘java -jar’ and passing in appropriate configuration properties. We provide many common microservice applications for common operations so you don’t have to start from scratch when addressing common use-cases which build upon the rich ecosystem of Spring Projects, e.g Spring Integration, Spring Data, Spring Hadoop and Spring Batch. Creating your own microservice application is similar to creating other Spring Boot applications, you can start using the Spring Initialzr web site or the UI to create the basic scaffolding of either a Stream or Task based microservice.

In addition to passing in the appropriate configuration to the applications, the Data Flow server is responsible for preparing the target platform’s infrastructure so that the application can be deployed. For example, in Cloud Foundry it would be binding specified services to the applications and executing the ‘cf push’ command for each application. For Kubernetes it would be creating the replication controller, service, and load balancer.

The Data Flow Server helps simplify the deployment of multiple applications onto a target runtime, but one could also opt to deploy each of the microservice applications manually and not use Data Flow at all. This approach might be more appropriate to start out with for small scale deployments, gradually adopting the convenience and consistency of Data Flow as you develop more applications. Manual deployment of Stream based microservices is also a useful educational exercise that will help you better understand some of the automatic applications configuration and platform targeting steps that the Data Flow Server provides.

4.1 Comparison to other Platform architectures

Spring Cloud Data Flow’s architectural style is different than other Stream and Batch processing platforms. For example in Apache Spark, Apache Flink, and Google Cloud Dataflow applications run on a dedicated compute engine cluster. The nature of the compute engine gives these platforms a richer environment for performing complex calculations on the data as compared to Spring Cloud Data Flow, but it introduces complexity of another execution environment that is often not needed when creating data centric applications. That doesn’t mean you cannot do real time data computations when using Spring Cloud Data Flow. Refer to the analytics section which describes the integration of Redis to handle common counting based use-cases as well as the RxJava integration for functional API driven analytics use-cases, such as time-sliding-window and moving-average among others.

Similarly, Apache Storm, Hortonworks DataFlow and Spring Cloud Data Flow’s predecessor, Spring XD, use a dedicated application execution cluster, unique to each product, that determines where your code should execute on the cluster and perform health checks to ensure that long lived applications are restarted if they fail. Often, framework specific interfaces are required to be used in order to correctly “plug in” to the cluster’s execution framework.

As we discovered during the evolution of Spring XD, the rise of multiple container frameworks in 2015 made creating our own runtime a duplication of efforts. There is no reason to build your own resource management mechanics, when there’s multiple runtime platforms that offer this functionality already. Taking these considerations into account is what made us shift to the current architecture where we delegate the execution to popular runtimes, runtimes that you may already be using for other purposes. This is an advantage in that it reduces the cognitive distance for creating and managing data centric applications as many of the same skills used for deploying other end-user/web applications are applicable.

5. Streaming Applications

While Spring Boot provides the foundation for creating DevOps friendly microservice applications, other libraries in the Spring ecosystem help create Stream based microservice applications. The most important of these is Spring Cloud Stream.

The essence of the Spring Cloud Stream programming model is to provide an easy way to describe multiple inputs and outputs of an application that communicate over messaging middleware. These input and outputs map onto Kafka topics or Rabbit exchanges and queues. Common application configuration for a Source that generates data, a Process that consumes and produces data and a Sink that consumes data is provided as part of the library.

5.1 Imperative Programming Model

Spring Cloud Stream is most closely integrated with Spring Integration's imperative "event at a time" programming model. This means you write code that handles a single event callback. For example,

```
@EnableBinding(Sink.class)
public class LoggingSink {

    @StreamListener(Sink.INPUT)
    public void log(String message) {
        System.out.println(message);
    }
}
```

In this case the String payload of a message coming on the input channel, is handed to the log method. The `@EnableBinding` annotation is what is used to tie together the input channel to the external middleware.

5.2 Functional Programming Model

However, Spring Cloud Stream can support other programming styles. There is initial support for functional style programming via RxJava Observable APIs and upcoming versions will support callback methods with Project Reactor's Flux API and Apache Kafka's KStream API.

6. Streams

6.1 Topologies

The Stream DSL describes linear sequences of data flowing through the system. For example, in the stream definition `http | transformer | cassandra`, each pipe symbol connects the application on the left to the one on the right. Named channels can be used for routing and to fan out data to multiple messaging destinations.

Taps can be used to 'listen in' to the data that if flowing across any of the pipe symbols. Taps can be used as sources for new streams with an in independent life cycle.

6.2 Concurrency

For an application that will consume events, Spring Cloud stream exposes a concurrency setting that controls the size of a thread pool used for dispatching incoming messages. See the Consumer properties documentation for more information.

6.3 Partitioning

A common pattern in stream processing is to partition the data as it moves from one application to the next. Partitioning is a critical concept in stateful processing, for either performance or consistency reasons, to ensure that all related data is processed together. For example, in a time-windowed average calculation example, it is important that all measurements from any given sensor are processed by the same application instance. Alternatively, you may want to cache some data related to the incoming events so that it can be enriched without making a remote procedure call to retrieve the related data.

Spring Cloud Data Flow supports partitioning by configuring Spring Cloud Stream's output and input bindings. Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion across different types of middleware. Partitioning can thus be used whether the broker itself is naturally partitioned (e.g., Kafka topics) or not (e.g., RabbitMQ). The following image shows how data could be partitioned into two buckets, such that each instance of the average processor application consumes a unique set of data.

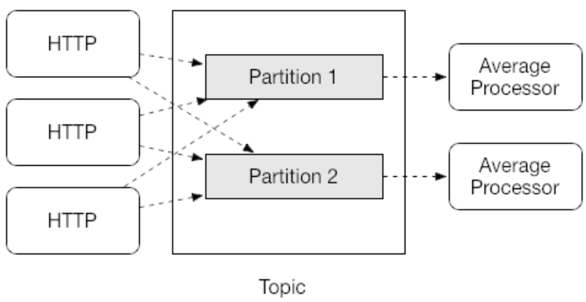

Figure 6.1. Spring Cloud Stream Partitioning

To use a simple partitioning strategy in Spring Cloud Data Flow, you only need set the instance count for each application in the stream and a `partitionKeyExpression` producer property when deploying the stream. The `partitionKeyExpression` identifies what part of the message will be used as the key to partition data in the underlying middleware. An `ingest` stream can be defined as `http | averageprocessor | cassandra` (Note that the Cassandra sink isn't shown in the diagram above).

Suppose the payload being sent to the http source was in JSON format and had a field called `sensorId`. Deploying the stream with the shell command `stream deploy ingest --propertiesFile ingestStream.properties` where the contents of the file `ingestStream.properties` are

```
app.http.count=3
app.averageprocessor.count=2
app.http.producer.partitionKeyExpression=payload.sensorId
```

will deploy the stream such that all the input and output destinations are configured for data to flow through the applications but also ensure that a unique set of data is always delivered to each averageprocessor instance. In this case the default algorithm is to evaluate `payload.sensorId % partitionCount` where the `partitionCount` is the application count in the case of RabbitMQ and the partition count of the topic in the case of Kafka.

Please refer to the section called “Passing stream partition properties during stream deployment” for additional strategies to partition streams during deployment and how they map onto the underlying Spring Cloud Stream Partitioning properties.

Also note, that you can’t currently scale partitioned streams. Read the section Section 9.3, “Scaling at runtime” for more information.

6.4 Message Delivery Guarantees

For consumer applications, there is a retry policy for exceptions generated during message handling. The default is to retry the callback method invocation 3 times and wait one second for the first retry. A backoff multiplier of 2 is used for the second and third attempts. All of these retry properties are configurable.

If there is still an exception on the last retry attempt, and dead letter queues are enabled, the message and exception message are published to the dead letter queue. The dead letter queue is a destination and its nature depends on the messaging middleware (e.g in the case of Kafka it is a dedicated topic). If dead letter functionality is not enabled, the message and exception is sent to the error channel, which by default logs the message and exception.

Additional messaging delivery guarantees are those provided by the underlying messaging middleware that is chosen for the application for both producing and consuming applications. Refer to the Kafka Consumer and Producer and Rabbit Consumer and Producer documentation for more details. You will find there to be extensive declarative support for all the native QOS options.

7. Analytics

Spring Cloud Data Flow is aware of certain Sink applications that will write counter data to Redis and provides an REST endpoint to read counter data. The types of counters supported are

- Counter - Counts the number of messages it receives, optionally storing counts in a separate store such as redis.
- Field Value Counter - Counts occurrences of unique values for a named field in a message payload
- Aggregate Counter - Stores total counts but also retains the total count values for each minute, hour day and month.

It is important to note that the timestamp that is used in the aggregate counter can come from a field in the message itself so that out of order messages are properly accounted.

8. Data Flow Server

8.1 Endpoints

The Data Flow Server uses an embedded servlet container and exposes REST endpoints for creating, deploying, undeploying, and destroying streams and tasks, querying runtime state, analytics, and the like. The Data Flow Server is implemented using Spring's MVC framework and the Spring HATEOAS library to create REST representations that follow the HATEOAS principle.

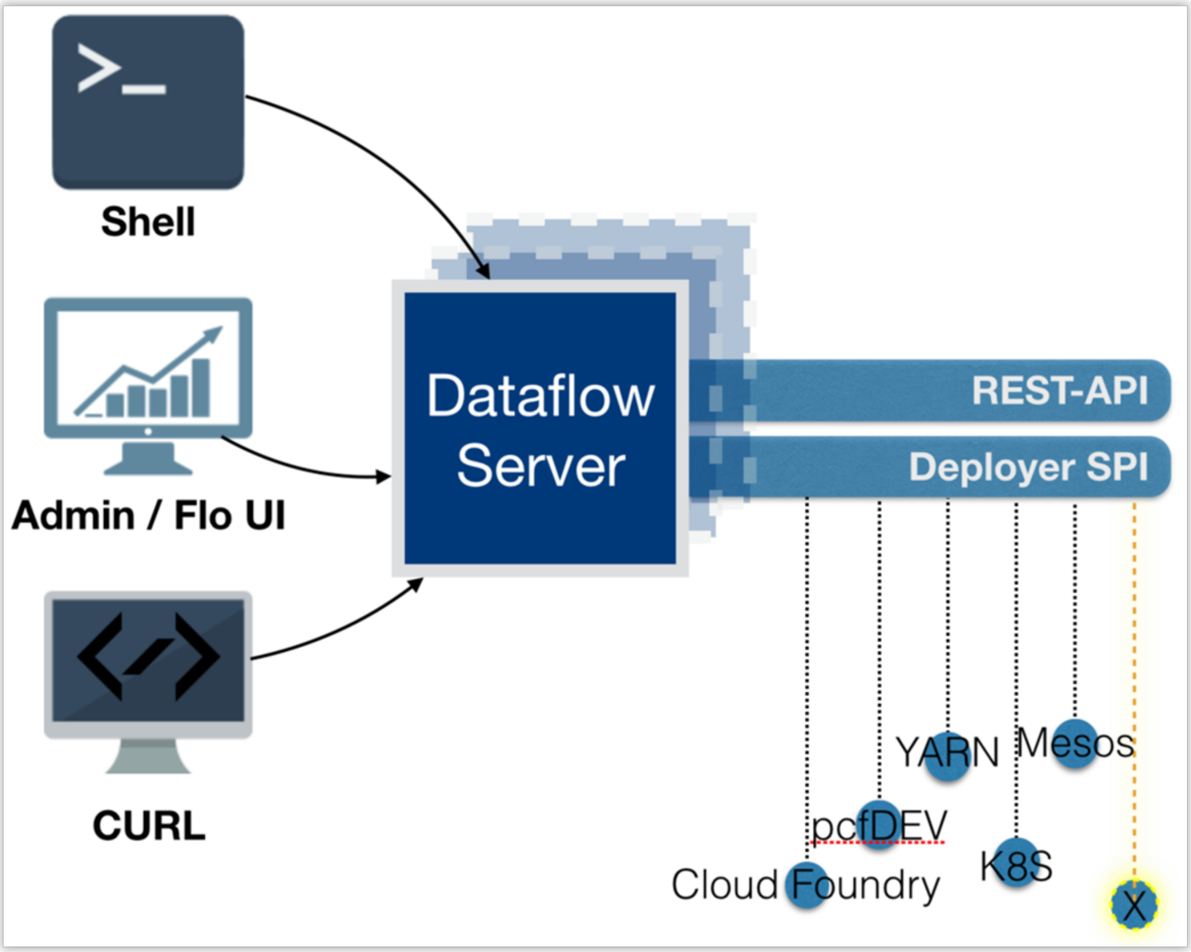

Figure 8.1. The Spring Cloud Data Flow Server

8.2 Customization

Each Data Flow Server executable jar targets a single runtime by delegating to the implementation of the deployer Service Provider Interface found on the classpath.

We provide a Data Flow Server executable jar that targets a single runtime. The Data Flow server delegates to the implementation of the deployer Service Provider Interface found on the classpath. In the current version, there are no endpoints specific to a target runtime, but may be available in future releases as a convenience to access runtime specific features

While we provide a server executable for each of the target runtimes you can also create your own customized server application using Spring Initialzr. This let's you add or remove functionality relative to the executable jar we provide. For example, adding additional security implementations, custom endpoints, or removing Task or Analytics REST endpoints. You can also enable or disable some features through the use of feature toggles.

8.3 Security

The Data Flow Server executable jars support basic http and OAuth 2.0 authentication to access it endpoints. Refer to the security section for more information.

Authorization via groups is planned for a future release.

9. Runtime

9.1 Fault Tolerance

The target runtimes supported by Data Flow all have the ability to restart a long lived application should it fail. Spring Cloud Data Flow sets up whatever health probe is required by the runtime environment when deploying the application.

The collective state of all applications that comprise the stream is used to determine the state of the stream. If an application fails, the state of the stream will change from ‘deployed’ to ‘partial’.

9.2 Resource Management

Each target runtime lets you control the amount of memory, disk and CPU that is allocated to each application. These are passed as properties in the deployment manifest using key names that are unique to each runtime. Refer to the each platforms server documentation for more information.

9.3 Scaling at runtime

When deploying a stream, you can set the instance count for each individual application that comprises the stream. Once the stream is deployed, each target runtime lets you control the target number of instances for each individual application. Using the APIs, UIs, or command line tools for each runtime, you can scale up or down the number of instances as required. Future work will provide a portable command in the Data Flow Server to perform this operation.

Currently, this is not supported with the Kafka binder (based on the 0.8 simple consumer at the time of the release), as well as partitioned streams, for which the suggested workaround is redeploying the stream with an updated number of instances. Both cases require a static consumer set up based on information about the total instance count and current instance index, a limitation intended to be addressed in future releases. For example, Kafka 0.9 and higher provides good infrastructure for scaling applications dynamically and will be available as an alternative to the current Kafka 0.8 based binder in the near future. One specific concern regarding scaling partitioned streams is the handling of local state, which is typically reshuffled as the number of instances is changed. This is also intended to be addressed in the future versions, by providing first class support for local state management.

9.4 Application Versioning

Application versioning, that is upgrading or downgrading an application from one version to another, is not directly supported by Spring Cloud Data Flow. You must rely on specific target runtime features to perform these operational tasks.

The roadmap for Spring Cloud Data Flow will deploy applications that are compatible with Spinnaker to manage the complete application lifecycle. This also includes automated canary analysis backed by application metrics. Portable commands in the Data Flow server to trigger pipelines in Spinnaker are also planned.

Part III. Getting started

10. Deploying on Cloud Foundry

Spring Cloud Data Flow can be used to deploy modules in a Cloud Foundry environment. When doing so, the server application can either run itself on Cloud Foundry, or on another installation (e.g. a simple laptop).

The required configuration amounts to the same in either case, and is merely related to providing credentials to the Cloud Foundry instance so that the server can spawn applications itself. Any Spring Boot compatible configuration mechanism can be used (passing program arguments, editing configuration files before building the application, using Spring Cloud Config, using environment variables, etc.), although some may prove more practicable than others when running *on* Cloud Foundry.

Note

By default, the application registry in Spring Cloud Data Flow's Cloud Foundry server is empty. It is intentionally designed to allow users to have the flexibility of choosing and registering applications, as they find appropriate for the given use-case requirement. Depending on the message-binder of choice, users can register between RabbitMQ or Apache Kafka based maven artifacts.

10.1 Provision a Redis service instance on Cloud Foundry.

Use `cf marketplace` to discover which plans are available to you, depending on the details of your Cloud Foundry setup. For example when using Pivotal Web Services:

```
cf create-service rediscloud 30mb redis
```

10.2 Provision a Rabbit service instance on Cloud Foundry.

Use `cf marketplace` to discover which plans are available to you, depending on the details of your Cloud Foundry setup. For example when using Pivotal Web Services:

```
cf create-service cloudamqp lemur rabbit
```

10.3 Download the Spring Cloud Data Flow Server and Shell apps:

```
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-server-
cloudfoundry/1.0.0.M4/spring-cloud-dataflow-server-cloudfoundry-1.0.0.M4.jar
wget http://repo.spring.io/release/org/springframework/cloud/spring-cloud-dataflow-shell/1.0.0.RELEASE/
spring-cloud-dataflow-shell-1.0.0.RELEASE.jar
```

You can either deploy the server application on Cloud Foundry itself or on your local machine. The following two sections explain each way of running the server.

10.4 Deploying the Server app on Cloud Foundry

Push the server application on Cloud Foundry, configure it (see below) and start it.

Note

You must use a unique name for your app; an app with the same name in the same organization will cause your deployment to fail

```
cf push dataflow-server --no-start -p spring-cloud-dataflow-server-cloudfoundry-1.0.0.M4.jar
cf bind-service dataflow-server redis
cf bind-service dataflow-server rabbit
```

Note

If you are pushing to a space with multiple users, for example on PWS, there may already be a route taken for the applicaiton name you have chosen. You can use the options `--random-route` to avoid this when pushing the app.

Now we can configure the app. The following configuration is for Pivotal Web Services. You need to fill in {org}, {space}, {email} and {password} before running these commands.

Note

Only set 'Skip SSL Validation' to true if you're running on a Cloud Foundry instance using self-signed certs (e.g. in development). Do not use for production.

Note

If you are deploying in an environment that requires you to sign on using the Pivotal Single Sign-On Service, refer to the section Chapter 14, *Authentication and Cloud Foundry* for information on how to configure the server.

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL https://api.run.pivotal.io
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG {org}
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE {space}
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN cfapps.io
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SERVICES redis,rabbit
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME {email}
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD {password}
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION false
```

Spring Cloud Data Flow server implementations (cf, mesos, yarn, or kubernetes) do not have 'any' default remote maven repository configured. This is intentionally designed to provide the flexibility for the users, so they can override and point to a remote repository of their choice. The out-of-the-box applications that are supported by Spring Cloud Data Flow are available in Spring's repository, so if you want to use them, you 'must' set it as the remote repository as listed below.

```
cf set-env dataflow-server MAVEN_REMOTE_REPOSITORIES_REPO1_URL https://repo.spring.io/libs-snapshot
```

where `repo1` is the alias name for the remote repository.

You can also set other optional properties for deployment to Cloud Foundry.

- You can set the buildpack that will be used to deploy the application. For example, to use the Java offline buildback, set the following environment variable

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_BUILDPACK java_buildpack_offline
```

- If you'd like to use `config-server` to manage centralized configurations for all the applications orchestrated by Spring Cloud Data Flow, you can set it up like the following.

```
cf set-env dataflow-server SPRING_APPLICATION_JSON
 '{"spring.cloud.dataflow.applicationProperties.stream.spring.cloud.config.uri": "http://
<CONFIG_SERVER_URI>"}'
```

- The default memory and disk sizes for a deployed application can also be configured. By default they are 1024 MB memory and 1024 MB disk.

Thse are controlled by setting an integer value, representing the number of MB, to the following properties, `spring.cloud.deployer.cloudfoundry.memory` and `spring.cloud.deployer.cloudfoundry.disk`. The default number of instances to deploy is set to 1, but can be overridden using with the `spring.cloud.deployer.cloudfoundry.instances` property. All these properties are `@ConfigurationProperties` of the Cloud Foundry deployer. See CloudFoundryDeployerProperties.java for more information.

We are now ready to start the app.

```
cf start dataflow-server
```

Alternatively, you can run the Admin application locally on your machine which is described in the next section.

10.5 Running the Server app locally

To run the server application locally, targeting your Cloud Foundry installation, you you need to configure the application either by passing in command line arguments (see below) or setting a number of environment variables.

To use environment variables set the following:

```
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL=https://api.run.pivotal.io
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG={org}
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE={space}
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN=cfapps.io
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SERVICES=redis,rabbit
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME={email}
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD={password}
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION=false
```

You need to fill in {org}, {space}, {email} and {password} before running these commands.

Note

Only set 'Skip SSL Validation' to true if you're running on a Cloud Foundry instance using self-signed certs (e.g. in development). Do not use for production.

Now we are ready to start the server application:

```
java -jar spring-cloud-dataflow-server-cloudfoundry-1.0.0.M4.jar [--option1=value1] [--option2=value2]
 [etc.]
```

10.6 Running Spring Cloud Data Flow Shell locally

Run the shell and optionally target the Admin application if not running on the same host (will typically be the case if deployed on Cloud Foundry as explained here)

```
$ java -jar spring-cloud-dataflow-shell-1.0.0.RELEASE.jar
```

```
server-unknown:>dataflow config server http://dataflow-server.cfapps.io
Successfully targeted http://dataflow-server.cfapps.io
dataflow:>
```

By default, the application registry will be empty. If you would like to register all out-of-the-box stream applications built with the RabbitMQ binder in bulk, you can with the following command. For more details, review how to register applications.

```
dataflow:>app import --uri http://bit.ly/stream-applications-rabbit-maven
```

You can now use the shell commands to list available applications (source/processors/sink) and create streams. For example:

```
dataflow:> stream create --name httptest --definition "http | log" --deploy
```

Note

You will need to wait a little while until the apps are actually deployed successfully before posting data. Tail the log file for each application to verify the application has started.

Now post some data. The URL will be unique to your deployment, the following is just an example

```
dataflow:> http post --target http://dataflow-nonconcentrative-knar-httptest-http.cfapps.io --
data "hello world"
```

Look to see if `hello world` ended up in log files for the `log` application.

11. Security

By default, the Data Flow server is unsecured and runs on an unencrypted HTTP connection. You can secure your REST endpoints, as well as the Data Flow Dashboard by enabling HTTPS and requiring clients to authenticate. More details about securing the REST endpoints and configuring to authenticate against an OAUTH backend (*i.e: UAA/SSO running on Cloud Foundry*), please review the security section from the core reference guide. The security configurations can be configured in `dataflow-server.yml` or passed as environment variables through `cf set-env` commands.

12. Security

By default, the Data Flow server is unsecured and runs on an unencrypted HTTP connection. You can secure your REST endpoints, as well as the Data Flow Dashboard by enabling HTTPS and requiring clients to authenticate. More details about securing the REST endpoints and configuring to authenticate against an OAUTH backend (*i.e: UAA/SSO running on Cloud Foundry*), please review the security section from the core reference guide. The security configurations can be configured in `dataflow-server.yml` or passed as environment variables through `cf set-env` commands.

13. Application Names and Prefixes

To help avoid clashes with routes across spaces in Cloud Foundry, a naming strategy to provide a random prefix to a deployed application is available and is enabled by default. The default configurations are overridable and the respective properties can be set via `cf set-env` commands.

For instance, if you'd like to disable the randmoization, you can override it through:

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ENABLE_RANDOM_APP_NAME_PREFIX false
```

14. Authentication and Cloud Foundry

When deploying Spring Cloud Data Flow to Cloud Foundry, you can take advantage of the *Spring Cloud Single Sign-On Connector*, which provides Cloud Foundry specific auto-configuration support for OAuth 2.0, when used in conjunction with the *Pivotal Single Sign-On Service*.

Simply set `security.basic.enabled` to `true` and in Cloud Foundry bind the SSO service to your Data Flow Server app and SSO will be enabled.

15. Configuration Reference

The following pieces of configuration must be provided. These are Spring Boot @ConfigurationProperties so you can set them as environment variables or by any other means that Spring Boot supports. Here is a listing in environment variable format as that is an easy way to get started configuring Boot applications in Cloud Foundry.

```
# Default values cited after the equal sign.
# Example values, typical for Pivotal Web Services, cited as a comment

# url of the CF API (used when using cf login -a for example), e.g. https://api.run.pivotal.io
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL)
spring.cloud.deployer.cloudfoundry.url=

# name of the organization that owns the space above, e.g. youruser-org
# (For Setting Env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG)
spring.cloud.deployer.cloudfoundry.org=

# name of the space into which modules will be deployed
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE)
spring.cloud.deployer.cloudfoundry.space=<same space as server when running on CF, or 'development'>

# the root domain to use when mapping routes, e.g. cfapps.io
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN)
spring.cloud.deployer.cloudfoundry.domain=

# Comma separated set of service instance names to bind to the module.
# Amongst other things, this should include a service that will be used
# for Spring Cloud Stream binding
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SERVICES)
spring.cloud.deployer.cloudfoundry.services=redis,rabbit

# username and password of the user to use to create apps (modules)
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME and
 SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD)
spring.cloud.deployer.cloudfoundry.username=
spring.cloud.deployer.cloudfoundry.password=

# Whether to allow self-signed certificates during SSL validation
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION)
spring.cloud.deployer.cloudfoundry.skipSslValidation=false
```

Note that you can set the following properties `SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SERVICES`, `spring.cloud.deployer.cloudfoundry.memory`, and `spring.cloud.deployer.cloudfoundry.disk` as part of an individual deployment request prefixed by the `app.<name of application>`. For example

```
>stream create --name ticktock --definition "time | log"
>stream deploy --name ticktock --properties "app.time.spring.cloud.deployer.cloudfoundry.memory=2048"
```

will deploy the time source with 2048MB of memory, while the log sink will use the default 1024MB.

Part IV. Streams

In this section you will learn all about Streams and how to use them with Spring Cloud Data Flow.

16. Introduction

In Spring Cloud Data Flow, a basic stream defines the ingestion of event driven data from a *source* to a *sink* that passes through any number of *processors*. Streams are composed of spring-cloud-stream applications and the deployment of stream definitions is done via the Data Flow Server (REST API). The Getting Started section shows you how to start these servers and how to start and use the Spring Cloud Data Flow shell.

A high level DSL is used to create stream definitions. The DSL to define a stream that has an http source and a file sink (with no processors) is shown below

```
http | file
```

The DSL mimics a UNIX pipes and filters syntax. Default values for ports and filenames are used in this example but can be overridden using `--` options, such as

```
http --server.port=8091 | file --directory=/tmp/httpdata/
```

To create these stream definitions you use the shell or make an HTTP POST request to the Spring Cloud Data Flow Server. More details can be found in the sections below.

17. Stream DSL

In the examples above, we connected a source to a sink using the pipe symbol `|`. You can also pass properties to the source and sink configurations. The property names will depend on the individual app implementations, but as an example, the `http` source app exposes a `server.port` setting which allows you to change the data ingestion port from the default value. To create the stream using port 8000, we would use

```
dataflow:> stream create --definition "http --server.port=8000 | log" --name myhttpstream
```

The shell provides tab completion for application properties and also the shell command `app info` provides some additional documentation.

18. Register a Stream App

Register a Stream App with the App Registry using the Spring Cloud Data Flow Shell `app register` command. You must provide a unique name, application type, and a URI that can be resolved to the app artifact. For the type, specify "source", "processor", or "sink". Here are a few examples:

```
dataflow:>app register --name mysource --type source --uri maven://com.example:mysource:0.0.1-SNAPSHOT

dataflow:>app register --name myprocessor --type processor --uri file:///Users/example/
myprocessor-1.2.3.jar

dataflow:>app register --name mysink --type sink --uri http://example.com/mysink-2.0.1.jar
```

When providing a URI with the `maven` scheme, the format should conform to the following:

```
maven://<groupId>:<artifactId>[:<extension>[:<classifier>]]:<version>
```

For example, if you would like to register the snapshot versions of the `http` and `log` applications built with the RabbitMQ binder, you could do the following:

```
dataflow:>app register --name http --type source --uri maven://
org.springframework.cloud.stream.app:http-source-rabbit:1.0.0.BUILD-SNAPSHOT
dataflow:>app register --name log --type sink --uri maven://org.springframework.cloud.stream.app:log-
sink-rabbit:1.0.0.BUILD-SNAPSHOT
```

If you would like to register multiple apps at one time, you can store them in a properties file where the keys are formatted as `<type>.<name>` and the values are the URIs.

For example, if you would like to register the snapshot versions of the `http` and `log` applications built with the RabbitMQ binder, you could have the following in a properties file [*eg: stream-apps.properties*]:

```
source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.0.0.BUILD-SNAPSHOT
sink.log=maven://org.springframework.cloud.stream.app:log-sink-rabbit:1.0.0.BUILD-SNAPSHOT
```

Then to import the apps in bulk, use the `app import` command and provide the location of the properties file via `--uri`:

```
dataflow:>app import --uri file:///<YOUR_FILE_LOCATION>/stream-apps.properties
```

For convenience, we have the static files with application-URIs (for both maven and docker) available for all the out-of-the-box Stream app-starters. You can point to this file and import all the application-URIs in bulk. Otherwise, as explained in previous paragraphs, you can register them individually or have your own custom property file with only the required application-URIs in it. It is recommended, however, to have a "focused" list of desired application-URIs in a custom property file.

List of available static property files:

- Maven based Stream Applications with RabbitMQ Binder: bit.ly/stream-applications-rabbit-maven
- Maven based Stream Applications with Kafka Binder: bit.ly/stream-applications-kafka-maven
- Docker based Stream Applications with RabbitMQ Binder: bit.ly/stream-applications-rabbit-docker
- Docker based Stream Applications with Kafka Binder: bit.ly/stream-applications-kafka-docker

For example, if you would like to register all out-of-the-box stream applications built with the RabbitMQ binder in bulk, you can with the following command.

```
dataflow:>app import --uri http://bit.ly/stream-applications-rabbit-maven
```

You can also pass the `--local` option (which is TRUE by default) to indicate whether the properties file location should be resolved within the shell process itself. If the location should be resolved from the Data Flow Server process, specify `--local false`.

When using either `app register` or `app import`, if a stream app is already registered with the provided name and type, it will not be overridden by default. If you would like to override the pre-existing stream app, then include the `--force` option.

Note

In some cases the Resource is resolved on the server side, whereas in others the URI will be passed to a runtime container instance where it is resolved. Consult the specific documentation of each Data Flow Server for more detail.

18.1 Whitelisting application properties

Stream applications are Spring Boot applications which are aware of many common application properties, e.g. `server.port` but also families of properties such as those with the prefix `spring.jmx` and `logging`. When creating your own application it is desirable to whitelist properties so that the shell and the UI can display them first as primary properties when presenting options via TAB completion or in drop-down boxes.

To whitelist application properties create a file named `spring-configuration-metadata-whitelist.properties` in the `META-INF` resource directory. There are two property keys that can be used inside this file. The first key is named `configuration-properties.classes`. The value is a comma separated list of fully qualified `@ConfigurationProperty` class names. The second key is `configuration-properties.names` whose value is a comma separated list of property names. This can contain the full name of property, such as `server.port` or a partial name to whitelist a category of property names, e.g. `spring.jmx`.

The Spring Cloud Stream application starters are a good place to look for examples of usage. Here is a simple example of the file source's `spring-configuration-metadata-whitelist.properties` file

```
configuration.classes=org.springframework.cloud.stream.app.file.sink.FileSinkProperties
```

If for some reason we also wanted to add `file.prefix` to this file, it would look like

```
configuration.classes=org.springframework.cloud.stream.app.file.sink.FileSinkProperties
configuration-properties.names=server.port
```

Important

As of Spring Cloud Data Flow `1.0.0.RELEASE` the whitelisting of application properties is only explicitly supported for Spring Boot `1.3.x` based application. Milestone releases of the upcoming Spring Boot `1.4.0` release are not explicitly supported, yet.

The `spring-boot-maven-plugin` used in `1.4.x` has a different approach in handling the nested archives inside the `jar`. As a result you will notice that the application properties are not listed using `app info` command at all. As a temporary workaround, you can override the managed version of your app's `spring-boot-maven-plugin` explicitly and revert to a version of the latest 1.3.x release:

For example, if your app's `pom.xml` specifies to use Spring Boot `1.4.0.M3`:

```
<parent>
  <artifactId>spring-boot-starter-parent</artifactId>
  <groupId>org.springframework.boot</groupId>
  <version>1.4.0.M3</version>
  <relativePath></relativePath>
</parent>
```

Then you can override the managed version of the `spring-boot-maven-plugin` with:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <version>1.3.5.RELEASE</version>  ❶
</plugin>
```

❶ Overriding the managed version `1.4.0.M3`.

Also, if you have your own `dataflow` server built using `@EnableDataflowServer` and using Spring Boot `1.4.x` in that, you would need to explicitly override the `spring-boot-maven-plugin` with any of `1.3.x` releases.

19. Creating a Stream

The Spring Cloud Data Flow Server exposes a full RESTful API for managing the lifecycle of stream definitions, but the easiest way to use is it is via the Spring Cloud Data Flow shell. Start the shell as described in the Getting Started section.

New streams are created by posting stream definitions. The definitions are built from a simple DSL. For example, let's walk through what happens if we execute the following shell command:

```
dataflow:> stream create --definition "time | log" --name ticktock
```

This defines a stream named `ticktock` based off the DSL expression `time | log`. The DSL uses the "pipe" symbol `|`, to connect a source to a sink.

Then to deploy the stream execute the following shell command (or alternatively add the `--deploy` flag when creating the stream so that this step is not needed):

```
dataflow:> stream deploy --name ticktock
```

The Data Flow Server resolves `time` and `log` to maven coordinates and uses those to launch the `time` and `log` applications of the stream.

```
2016-06-01 09:41:21.728  INFO 79016 --- [nio-9393-exec-6] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app ticktock.log instance 0
   Logs will be in /var/folders/wn/8jxm_tbd1vj28c8vj37n900m0000gn/T/spring-cloud-
dataflow-912434582726479179/ticktock-1464788481708/ticktock.log
2016-06-01 09:41:21.914  INFO 79016 --- [nio-9393-exec-6] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app ticktock.time instance 0
   Logs will be in /var/folders/wn/8jxm_tbd1vj28c8vj37n900m0000gn/T/spring-cloud-
dataflow-912434582726479179/ticktock-1464788481910/ticktock.time
```

In this example, the time source simply sends the current time as a message each second, and the log sink outputs it using the logging framework. You can tail the `stdout` log (which has an "_<instance>" suffix). The log files are located within the directory displayed in the Data Flow Server's log output, as shown above.

```
$ tail -f /var/folders/wn/8jxm_tbd1vj28c8vj37n900m0000gn/T/spring-cloud-dataflow-912434582726479179/
ticktock-1464788481708/ticktock.log/stdout_0.log
2016-06-01 09:45:11.250  INFO 79194 --- [  kafka-binder-] log.sink    : 06/01/16 09:45:11
2016-06-01 09:45:12.250  INFO 79194 --- [  kafka-binder-] log.sink    : 06/01/16 09:45:12
2016-06-01 09:45:13.251  INFO 79194 --- [  kafka-binder-] log.sink    : 06/01/16 09:45:13
```

19.1 Application properties

Application properties are the properties associated with each application in the stream. When the application is deployed, the application properties are applied to the application via command line arguments or environment variables based on the underlying deployment implementation.

Passing application properties when creating a stream

The following stream

```
dataflow:> stream create --definition "time | log" --name ticktock
```

can have application properties defined at the time of stream creation.

The shell command `app info` displays the white-listed application properties for the application. For more info on the property white listing refer to Section 18.1, "Whitelisting application properties"

Below are the white listed properties for the app `time`:

```
dataflow:> app info source:time
###########################################################################################################################
#         Option Name          #         Description          #           Default            #
    Type             #
###########################################################################################################################
#trigger.time-unit             #The TimeUnit to apply to delay#<none>
 #java.util.concurrent.TimeUnit #
#                              #values.                       #                              #
                     #
#trigger.fixed-delay           #Fixed delay for periodic      #1
 #java.lang.Integer             #
#                              #triggers.                     #                              #
                     #
#trigger.cron                  #Cron expression value for the #<none>
 #java.lang.String              #
#                              #Cron Trigger.                 #                              #
                     #
#trigger.initial-delay         #Initial delay for periodic    #0
 #java.lang.Integer             #
#                              #triggers.                     #                              #
                     #
#trigger.max-messages          #Maximum messages per poll, -1 #1
 #java.lang.Long                #
#                              #means infinity.               #                              #
                     #
#trigger.date-format           #Format for the date value.    #<none>
 #java.lang.String              #
###########################################################################################################################
```

Below are the white listed properties for the app `log`:

```
dataflow:> app info sink:log
###########################################################################################################################
#         Option Name          #         Description          #           Default            #
    Type             #
###########################################################################################################################
#log.name                      #The name of the logger to use.#<none>
 #java.lang.String              #
#log.level                     #The level at which to log     #<none>
 #org.springframework.integratio#
#                              #messages.                     #
 #n.handler.LoggingHandler$Level#
#log.expression                #A SpEL expression (against the#payload
 #java.lang.String              #
#                              #incoming message) to evaluate #                              #
                     #
#                              #as the logged message.        #                              #
                     #
###########################################################################################################################
```

The application properties for the `time` and `log` apps can be specified at the time of `stream` creation as follows:

```
dataflow:> stream create --definition "time --fixed-delay=5 | log --level=WARN" --name ticktock
```

Note that the properties `fixed-delay` and `level` defined above for the apps `time` and `log` are the 'short-form' property names provided by the shell completion. These 'short-form' property names are applicable only for the white-listed properties and in all other cases, only *fully qualified* property names should be used.

Passing application properties when deploying a stream

The application properties can also be specified when deploying a stream. When specified during deployment, these application properties can either be specified as 'short-form' property names (applicable for white-listed properties) or *fully qualified* property names. The application properties should have the prefix "app.<appName/label>".

For example, the stream

```
dataflow:> stream create --definition "time | log" --name ticktock
```

can be deployed with application properties using the 'short-form' property names:

```
dataflow:>stream deploy ticktock --properties "app.time.fixed-delay=5,app.log.level=ERROR"
```

When using the app label,

```
stream create ticktock --definition "a: time | b: log"
```

the application properties can be defined as:

```
stream deploy ticktock --properties "app.a.fixed-delay=4,app.b.level=ERROR"
```

Passing stream partition properties during stream deployment

A common pattern in stream processing is to partition the data as it is streamed. This entails deploying multiple instances of a message consuming app and using content-based routing so that messages with a given key (as determined at runtime) are always routed to the same app instance. You can pass the partition properties during stream deployment to declaratively configure a partitioning strategy to route each message to a specific consumer instance.

See below for examples of deploying partitioned streams:

app.[app/label name].producer.partitionKeyExtractorClass
: The class name of a PartitionKeyExtractorStrategy (default `null`)

app.[app/label name].producer.partitionKeyExpression
: A SpEL expression, evaluated against the message, to determine the partition key; only applies if `partitionKeyExtractorClass` is null. If both are null, the app is not partitioned (default `null`)

app.[app/label name].producer.partitionSelectorClass
: The class name of a PartitionSelectorStrategy (default `null`)

app.[app/label name].producer.partitionSelectorExpression
: A SpEL expression, evaluated against the partition key, to determine the partition index to which the message will be routed. The final partition index will be the return value (an integer) modulo `[nextModule].count`. If both the class and expression are null, the underlying binder’s default PartitionSelectorStrategy will be applied to the key (default `null`)

In summary, an app is partitioned if its count is > 1 and the previous app has a `partitionKeyExtractorClass` or `partitionKeyExpression` (class takes precedence). When a partition key is extracted, the partitioned app instance is determined by invoking the `partitionSelectorClass`, if present, or the `partitionSelectorExpression % partitionCount`, where `partitionCount` is application count in the case of RabbitMQ, and the underlying partition count of the topic in the case of Kafka.

If neither a `partitionSelectorClass` nor a `partitionSelectorExpression` is present the result is `key.hashCode() % partitionCount`.

Overriding application properties during stream deployment

Application properties that are defined during deployment override the same properties defined during the stream creation.

For example, the following stream has application properties defined during stream creation:

```
dataflow:> stream create --definition "time --fixed-delay=5 | log --level=WARN" --name ticktock
```

To override these application properties, one can specify the new property values during deployment:

```
dataflow:>stream deploy ticktock --properties "app.time.fixed-delay=4,app.log.level=ERROR"
```

19.2 Deployment properties

When deploying the stream, properties that control the deployment of the apps into the target platform are known as `deployment` properties. For instance, one can specify how many instances need to be deployed for the specific application defined in the stream using the deployment property called `count`.

Passing instance count as deployment property

If you would like to have multiple instances of an application in the stream, you can include a property with the deploy command:

```
dataflow:> stream deploy --name ticktock --properties "app.time.count=3"
```

Note that `count` is the **reserved** property name used by the underlying deployer. Hence, if the application also has a custom property named `count`, it is **not** supported when specified in 'short-form' form during stream *deployment* as it could conflict with the *instance* count deployer property. Instead, the `count` as a custom application property can be specified in its *fully qualified* form (example: `app.foo.bar.count`) during stream *deployment* or it can be specified using 'short-form' or *fully qualified* form during the stream *creation* where it will be considered as an app property.

Important

See Chapter 26, *Using Labels in a Stream*.

Inline vs file reference properties

When using the Spring Cloud Dataflow Shell, there are two ways to provide deployment properties: either **inline** or via a **file reference**. Those two ways are exclusive and documented below:

Inline properties

use the `--properties` shell option and list properties as a comma separated list of key=value pairs, like so:

```
stream deploy foo
    --properties "app.transform.count=2,app.transform.producer.partitionKeyExpression=payload"
```

Using a file reference

use the `--propertiesFile` option and point it to a local Java `.properties` file (i.e. that lives in the filesystem of the machine running the shell). Being read as a `.properties` file, normal rules apply (ISO 8859-1 encoding, =, `<space>` or `:` delimiter, etc.) although we recommend using = as a key-value pair delimiter for consistency:

```
stream deploy foo --propertiesFile myprops.properties
```

where `myprops.properties` contains:

```
app.transform.count=2
app.transform.producer.partitionKeyExpression=payload
```

Both the above properties will be passed as deployment properties for the stream `foo` above.

20. Destroying a Stream

You can delete a stream by issuing the `stream destroy` command from the shell:

```
dataflow:> stream destroy --name ticktock
```

If the stream was deployed, it will be undeployed before the stream definition is deleted.

21. Deploying and Undeploying Streams

Often you will want to stop a stream, but retain the name and definition for future use. In that case you can `undeploy` the stream by name and issue the `deploy` command at a later time to restart it.

```
dataflow:> stream undeploy --name ticktock
dataflow:> stream deploy --name ticktock
```

22. Other Source and Sink Application Types

Let's try something a bit more complicated and swap out the `time` source for something else. Another supported source type is `http`, which accepts data for ingestion over HTTP POSTs. Note that the `http` source accepts data on a different port from the Data Flow Server (default 8080). By default the port is randomly assigned.

To create a stream using an `http` source, but still using the same `log` sink, we would change the original command above to

```
dataflow:> stream create --definition "http | log" --name myhttpstream --deploy
```

which will produce the following output from the server

```
2016-06-01 09:47:58.920  INFO 79016 --- [io-9393-exec-10] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app myhttpstream.log instance 0
   Logs will be in /var/folders/wn/8jxm_tbd1vj28c8vj37n900m0000gn/T/spring-cloud-
dataflow-912434582726479179/myhttpstream-1464788878747/myhttpstream.log
2016-06-01 09:48:06.396  INFO 79016 --- [io-9393-exec-10] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app myhttpstream.http instance 0
   Logs will be in /var/folders/wn/8jxm_tbd1vj28c8vj37n900m0000gn/T/spring-cloud-
dataflow-912434582726479179/myhttpstream-1464788886383/myhttpstream.http
```

Note that we don't see any other output this time until we actually post some data (using a shell command). In order to see the randomly assigned port on which the http source is listening, execute:

```
dataflow:> runtime apps
```

You should see that the corresponding http source has a `url` property containing the host and port information on which it is listening. You are now ready to post to that url, e.g.:

```
dataflow:> http post --target http://localhost:1234 --data "hello"
dataflow:> http post --target http://localhost:1234 --data "goodbye"
```

and the stream will then funnel the data from the http source to the output log implemented by the log sink

```
2016-06-01 09:50:22.121  INFO 79654 --- [  kafka-binder-] log.sink    : hello
2016-06-01 09:50:26.810  INFO 79654 --- [  kafka-binder-] log.sink    : goodbye
```

Of course, we could also change the sink implementation. You could pipe the output to a file (`file`), to hadoop (`hdfs`) or to any of the other sink apps which are available. You can also define your own apps.

23. Simple Stream Processing

As an example of a simple processing step, we can transform the payload of the HTTP posted data to upper case using the stream definitions

```
http | transform --expression=payload.toUpperCase() | log
```

To create this stream enter the following command in the shell

```
dataflow:> stream create --definition "http | transform --expression=payload.toUpperCase() | log" --name
 mystream --deploy
```

Posting some data (using a shell command)

```
dataflow:> http post --target http://localhost:1234 --data "hello"
```

Will result in an uppercased 'HELLO' in the log

```
2016-06-01 09:54:37.749  INFO 80083 --- [  kafka-binder-] log.sink    : HELLO
```

24. Stateful Stream Processing

To demonstrate the data partitioning functionality, let’s deploy the following stream with Kafka as the binder.

```
dataflow:>stream create --name words --definition "http --server.port=9900 | splitter --
expression=payload.split(' ') | log"
Created new stream 'words'

dataflow:>stream deploy words --properties
 "app.splitter.producer.partitionKeyExpression=payload,app.log.count=2"
Deployed stream 'words'

dataflow:>http post --target http://localhost:9900 --data "How much wood would a woodchuck chuck if a
 woodchuck could chuck wood"
> POST (text/plain;Charset=UTF-8) http://localhost:9900 How much wood would a woodchuck chuck if a
 woodchuck could chuck wood
> 202 ACCEPTED
```

You’ll see the following in the server logs.

```
2016-06-05 18:33:24.982  INFO 58039 --- [nio-9393-exec-9] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app words.log instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-
dataflow-694182453710731989/words-1465176804970/words.log
2016-06-05 18:33:24.988  INFO 58039 --- [nio-9393-exec-9] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app words.log instance 1
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-
dataflow-694182453710731989/words-1465176804970/words.log
```

Review the `words.log instance 0` logs:

```
2016-06-05 18:35:47.047  INFO 58638 --- [  kafka-binder-] log.sink                                 : How
2016-06-05 18:35:47.066  INFO 58638 --- [  kafka-binder-] log.sink                                 :
 chuck
2016-06-05 18:35:47.066  INFO 58638 --- [  kafka-binder-] log.sink                                 :
 chuck
```

Review the `words.log instance 1` logs:

```
2016-06-05 18:35:47.047  INFO 58639 --- [  kafka-binder-] log.sink                                 :
 much
2016-06-05 18:35:47.066  INFO 58639 --- [  kafka-binder-] log.sink                                 :
 wood
2016-06-05 18:35:47.066  INFO 58639 --- [  kafka-binder-] log.sink                                 :
 would
2016-06-05 18:35:47.066  INFO 58639 --- [  kafka-binder-] log.sink                                 : a
2016-06-05 18:35:47.066  INFO 58639 --- [  kafka-binder-] log.sink                                 :
 woodchuck
2016-06-05 18:35:47.067  INFO 58639 --- [  kafka-binder-] log.sink                                 : if
2016-06-05 18:35:47.067  INFO 58639 --- [  kafka-binder-] log.sink                                 : a
2016-06-05 18:35:47.067  INFO 58639 --- [  kafka-binder-] log.sink                                 :
 woodchuck
2016-06-05 18:35:47.067  INFO 58639 --- [  kafka-binder-] log.sink                                 :
 could
2016-06-05 18:35:47.067  INFO 58639 --- [  kafka-binder-] log.sink                                 :
 wood
```

This shows that payload splits that contain the same word are routed to the same application instance.

25. Tap a Stream

Taps can be created at various producer endpoints in a stream. For a stream like this:

```
stream create --definition "http | step1: transform --expression=payload.toUpperCase() | step2:
 transform --expression=payload+'!' | log" --name mainstream --deploy
```

taps can be created at the output of `http`, `step1` and `step2`.

To create a stream that acts as a 'tap' on another stream requires to specify the `source destination name` for the tap stream. The syntax for source destination name is:

```
`:<stream-name>.<label/app-name>`
```

To create a tap at the output of `http` in the stream above, the source destination name is `mainstream.http` To create a tap at the output of the first transform app in the stream above, the source destination name is `mainstream.step1`

The tap stream DSL looks like this:

```
stream create --definition ":mainstream.http > counter" --name tap_at_http --deploy

stream create --definition ":mainstream.step1 > jdbc" --name tap_at_step1_transformer --deploy
```

Note the colon (:) prefix before the destination names. The colon allows the parser to recognize this as a destination name instead of an app name.

26. Using Labels in a Stream

When a stream is comprised of multiple apps with the same name, they must be qualified with labels:

```
stream create --definition "http | firstLabel: transform --expression=payload.toUpperCase() |
 secondLabel: transform --expression=payload+'!' | log" --name myStreamWithLabels --deploy
```

27. Explicit Broker Destinations in a Stream

One can connect to a specific destination name located in the broker (Rabbit, Kafka etc.,) either at the `source` or at the `sink` position.

The following stream has the destination name at the `source` position:

```
stream create --definition ":myDestination > log" --name ingest_from_broker --deploy
```

This stream receives messages from the destination `myDestination` located at the broker and connects it to the `log` app.

The following stream has the destination name at the `sink` position:

```
stream create --definition "http > :myDestination" --name ingest_to_broker --deploy
```

This stream sends the messages from the `http` app to the destination `myDestination` located at the broker.

From the above streams, notice that the `http` and `log` apps are interacting with each other via the broker (through the destination `myDestination`) rather than having a pipe directly between `http` and `log` within a single stream.

It is also possible to connect two different destinations (`source` and `sink` positions) at the broker in a stream.

```
stream create --definition ":destination1 > :destination2" --name bridge_destinations --deploy
```

In the above stream, both the destinations (`destination1` and `destination2`) are located in the broker. The messages flow from the source destination to the sink destination via a `bridge` app that connects them.

28. Directed Graphs in a Stream

If directed graphs are needed instead of the simple linear streams described above, two features are relevant.

First, named destinations may be used as a way to combine the output from multiple streams or for multiple consumers to share the output from a single stream. This can be done using the DSL syntax `http > :mydestination` or `:mydestination > log`.

Second, you may need to determine the output channel of a stream based on some information that is only known at runtime. In that case, a router may be used in the sink position of a stream definition. For more information, refer to the Router Sink starter's README.

28.1 Common application properties

In addition to configuration via DSL, Spring Cloud Data Flow provides a mechanism for setting common properties to all the streaming applications that are launched by it. This can be done by adding properties prefixed with `spring.cloud.dataflow.applicationProperties.stream` when starting the server. When doing so, the server will pass all the properties, without the prefix, to the instances it launches.

For example, all the launched applications can be configured to use a specific Kafka broker by launching the configuration server with the following options:

```
--
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.brokers=192.168.1.100:9092
--
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.zkNodes=192.168.1.100:2181
```

This will cause the properties `spring.cloud.stream.kafka.binder.brokers` and `spring.cloud.stream.kafka.binder.zkNodes` to be passed to all the launched applications.

Note

Properties configured using this mechanism have lower precedence than stream deployment properties. They will be overridden if a property with the same key is specified at stream deployment time (e.g. `app.http.spring.cloud.stream.kafka.binder.brokers` will override the common property).

Part V. Dashboard

This section describe how to use the Dashboard of Spring Cloud Data Flow.

29. Introduction

Spring Cloud Data Flow provides a browser-based GUI which currently has 6 sections:

- **Apps** Lists all available applications and provides the control to register/unregister them
- **Runtime** Provides the Data Flow cluster view with the list of all running applications
- **Streams** Deploy/undeploy Stream Definitions
- **Jobs** Perform Batch Job related functions
- **Analytics** Create data visualizations for the various analytics applications

Upon starting Spring Cloud Data Flow, the Dashboard is available at:

`http://<host>:<port>/dashboard`

For example: http://localhost:9393/dashboard

If you have enabled https, then it will be located at `https://localhost:9393/dashboard`. If you have enabled security, a login form is available at `http://localhost:9393/dashboard/#/login`.

Note: The default Dashboard server port is 9393

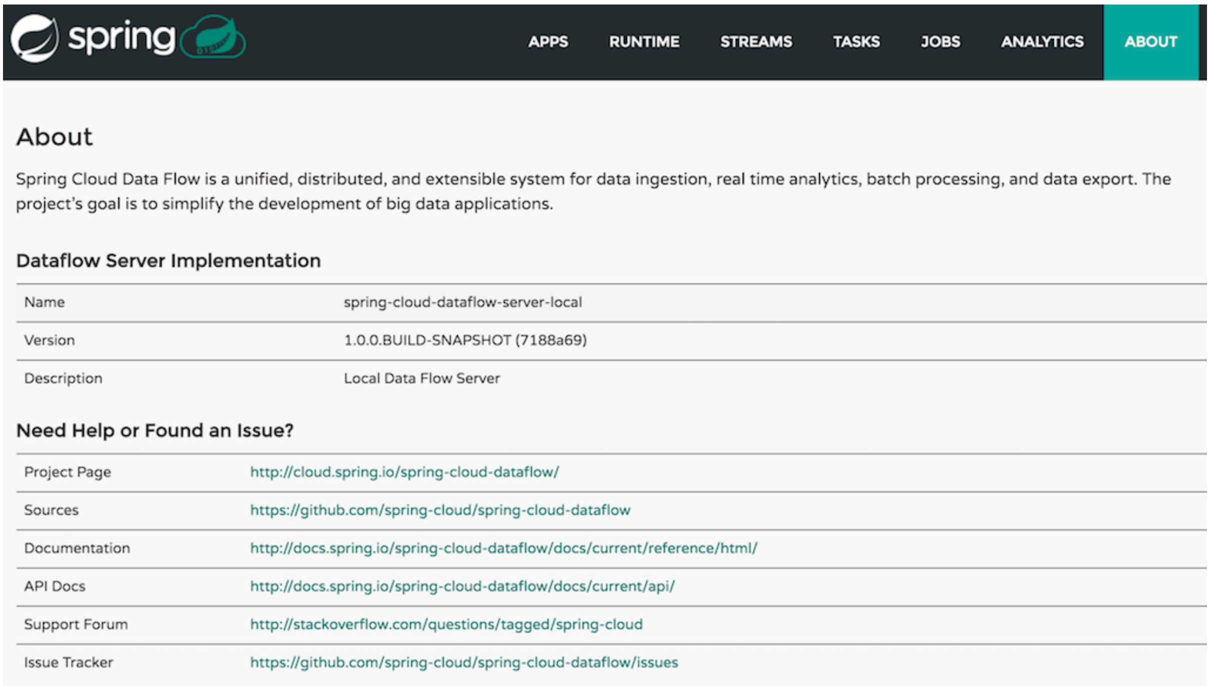

Figure 29.1. The Spring Cloud Data Flow Dashboard

30. Apps

The *Apps* section of the Dashboard lists all the available applications and provides the control to register/unregister them (if applicable). By clicking on the magnifying glass, you will get a listing of available definition properties.

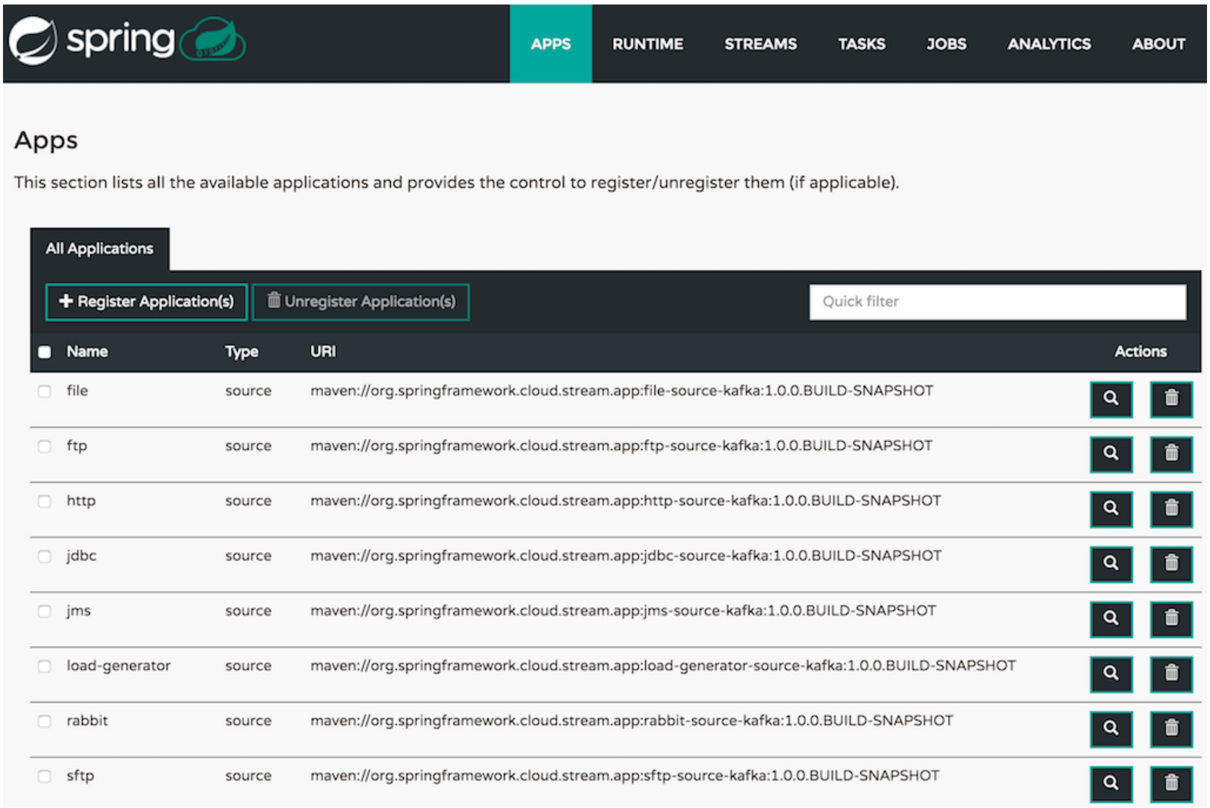

Figure 30.1. List of Available Applications

31. Runtime

The *Runtime* section of the Dashboard application shows the Spring Cloud Data Flow cluster view with the list of all running applications. For each runtime app the state of the deployment and the number of deployed instances is shown. A list of the used deployment properties is available by clicking on the app id.

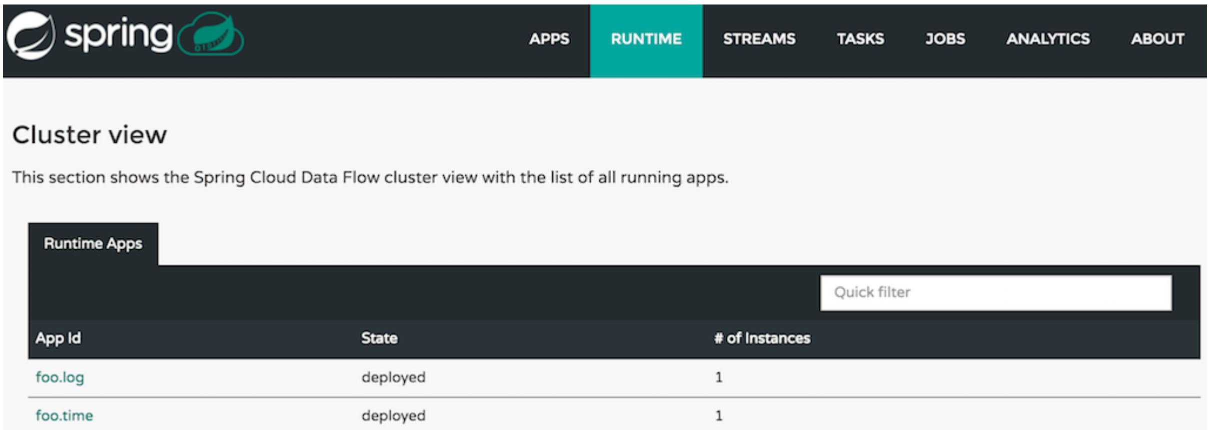

Figure 31.1. List of Running Applications

32. Streams

The *Streams* section of the Dashboard provides the *Definitions* tab that provides a listing of Stream definitions. There you have the option to **deploy** or **undeploy** those stream definitions. Additionally you can remove the definition by clicking on **destroy**.

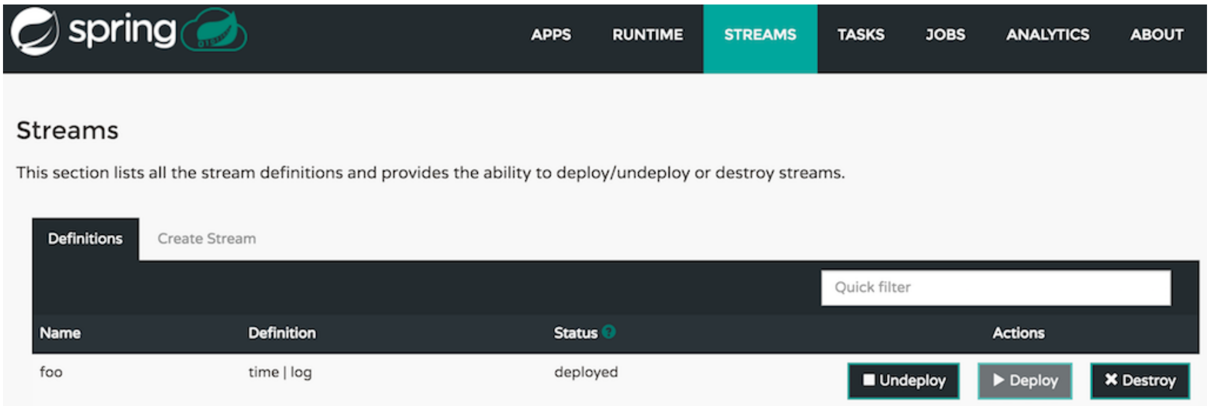

Figure 32.1. List of Stream Definitions

33. Create Stream

The *Create Stream* section of the Dashboard includes the Spring Flo designer tab that provides the canvas application, offering a interactive graphical interface for creating data pipelines.

In this tab, you can:

- Create, manage, and visualize stream pipelines using DSL, a graphical canvas, or both
- Write pipelines via DSL with content-assist and auto-complete
- Use auto-adjustment and grid-layout capabilities in the GUI for simpler and interactive organization of pipelines

Watch this screencast that highlights some of the "Flo for Spring Cloud Data Flow" capabilities. Spring Flo wiki includes more detailed content on core Flo capabilities.

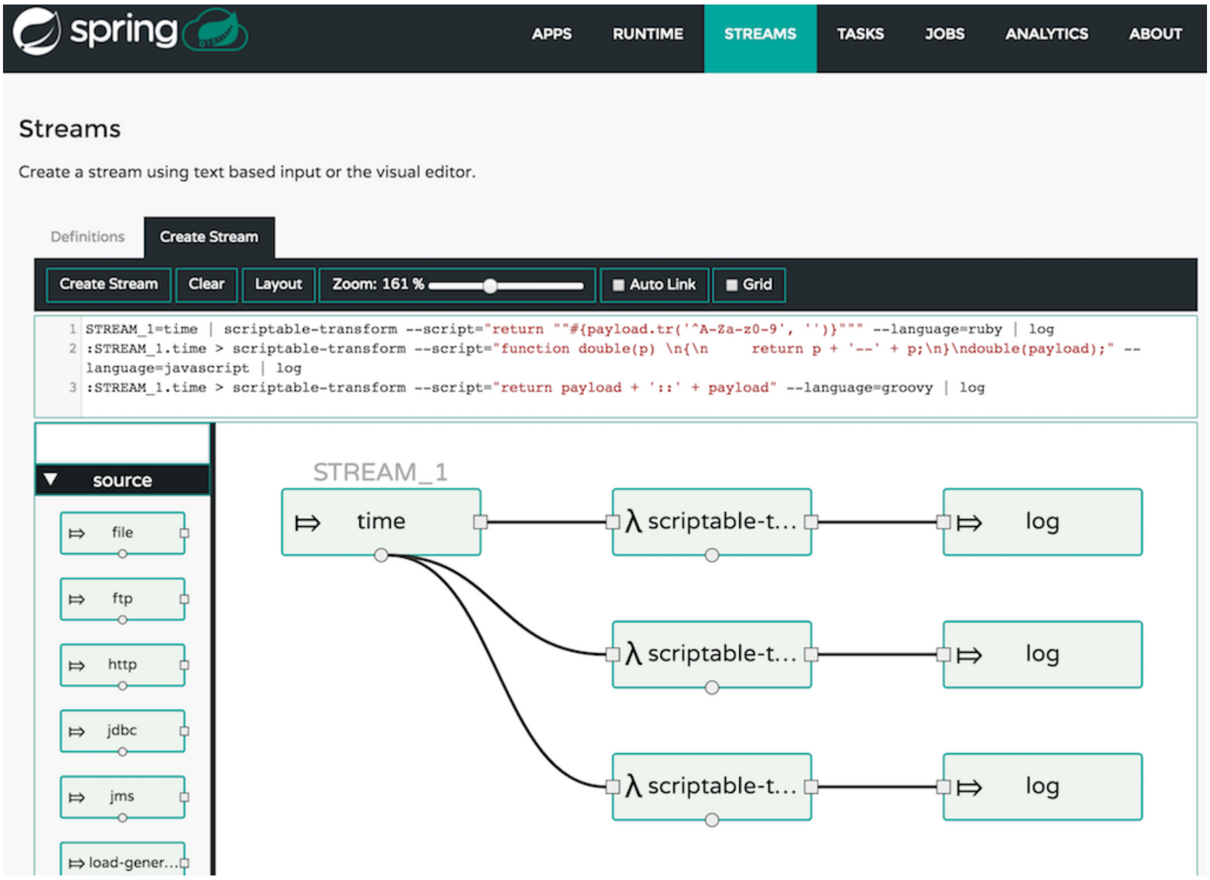

Figure 33.1. Flo for Spring Cloud Data Flow

34. Analytics

The *Analytics* section of the Dashboard provided data visualization capabilities for the various analytics applications available in *Spring Cloud Data Flow*:

- Counters
- Field-Value Counters

For example, if you have created the `springtweets` stream and the corresponding counter in the Counter chapter, you can now easily create the corresponding graph from within the **Dashboard** tab:

1. Under `Metric Type`, select `Counters` from the select box
2. Under `Stream`, select `tweetcount`
3. Under `Visualization`, select the desired chart option, `Bar Chart`

Using the icons to the right, you can add additional charts to the Dashboard, re-arange the order of created dashboards or remove data visualizations.

Part VI. ‘How-to’ guides

This section provides answers to some common ‘how do I do that…’ type of questions that often arise when using Spring Cloud Data Flow.

If you are having a specific problem that we don’t cover here, you might want to check out stackoverflow.com to see if someone has already provided an answer; this is also a great place to ask new questions (please use the `spring-cloud-dataflow` tag).

We’re also more than happy to extend this section; If you want to add a ‘how-to’ you can send us a pull request.

35. Configure Maven Properties

You can set the maven properties such as local maven repository location, remote maven repositories and their authentication credentials including the proxy server properties via commandline properties when starting the Dataflow server or using the `SPRING_APPLICATION_JSON` environment property for the Dataflow server.

The remote maven repositories need to be configured explicitly if the apps are resolved using maven repository as except `local` Data Flow server, other Data Flow server implementations (that use maven resources for app artifacts resolution) have no default value for remote repositories. The `local` server has [repo.spring.io/libs-snapshot]() as the default remote repository.

To pass the properties as commandline options:

```
$ java -jar <dataflow-server>.jar --maven.localRepository=mylocal
--maven.remote-repositories.repo1.url=https://repo1
--maven.remote-repositories.repo1.auth.username=repo1user
--maven.remote-repositories.repo1.auth.password=repo1pass
--maven.remote-repositories.repo2.url=https://repo2 --maven.proxy.host=proxyhost
--maven.proxy.port=9018 --maven.proxy.auth.username=proxyuser
--maven.proxy.auth.password=proxypass
```

or, using the `SPRING_APPLICATION_JSON` environment property:

```
export SPRING_APPLICATION_JSON='{ "maven": { "local-repository": "local","remote-repositories":
 { "repo1": { "url": "https://repo1", "auth": { "username": "repo1user", "password": "repo1pass" } },
"repo2": { "url": "https://repo2" } }, "proxy": { "host": "proxyhost", "port":
 9018, "auth": { "username": "proxyuser", "password": "proxypass" } } } }'
```

Formatted JSON:

```
SPRING_APPLICATION_JSON='{
  "maven": {
    "local-repository": "local",
    "remote-repositories": {
      "repo1": {
        "url": "https://repo1",
        "auth": {
          "username": "repo1user",
          "password": "repo1pass"
        }
      },
      "repo2": {
        "url": "https://repo2"
      }
    },
    "proxy": {
      "host": "proxyhost",
      "port": 9018,
      "auth": {
        "username": "proxyuser",
        "password": "proxypass"
      }
    }
  }
}'
```

Note

Depending on Spring Cloud Data Flow server implementation, you may have to pass the environment properties using the platform specific environment-setting capabilities. For instance, in Cloud Foundry, you'd be passing them as `cf set-env SPRING_APPLICATION_JSON`.

Part VII. Appendices

Appendix A. Migrating from Spring XD to Spring Cloud Data Flow

A.1 Terminology Changes

Old	New
XD-Admin	Server (*implementations*: local, cloud foundry, apache yarn, kubernetes, and apache mesos)
XD-Container	N/A
Modules	Applications
Admin UI	Dashboard
Message Bus	Binders
Batch / Job	Task

A.2 Modules to Applications

If you have custom Spring XD modules, you'd have to refactor them to use Spring Cloud Stream and Spring Cloud Task annotations, with updated dependencies and built as normal Spring Boot "applications".

Custom Applications

- Spring XD's stream and batch modules are refactored into Spring Cloud Stream and Spring Cloud Task application-starters, respectively. These applications can be used as the reference while refactoring Spring XD modules
- There are also some samples for Stream and Task applications for reference
- If you'd like to create a brand new custom application, use the getting started guide for Stream and Task applications and as well as review the development guide
- Alternatively, if you'd like to patch any of the out-of-the-box stream applications, you can follow the procedure here

Application Registration

- Custom Stream/Task application requires being installed to a maven repository for Local, YARN, and CF implementations or as docker images, when deploying to Kubernetes and Mesos. Other than maven and docker resolution, you can also resolve application artifacts from `http`, `file`, or as `hdfs` coordinates
- Unlike Spring XD, you do not have to upload the application bits while registering custom applications anymore; instead, you're expected to register the application coordinates that are hosted in the maven repository or by other means as discussed in the previous bullet

- By default, none of the out-of-the-box applications are preloaded already. It is intentionally designed to provide the flexibility to register app(s), as you find appropriate for the given use-case requirement
- Depending on the binder choice, you can manually add the appropriate binder dependency to build applications specific to that binder-type. Alternatively, you can follow the Spring Initialzr procedure to create an application with binder embedded in it

Application Properties

- counter-sink:
  - The peripheral `redis` is not required in Spring Cloud Data Flow. If you intend to use the `counter-sink`, then `redis` becomes required, and you're expected to have your own running `redis` cluster
- field-value-counter-sink:
  - The peripheral `redis` is not required in Spring Cloud Data Flow. If you intend to use the `field-value-counter-sink`, then `redis` becomes required, and you're expected to have your own running `redis` cluster
- aggregate-counter-sink:
  - The peripheral `redis` is not required in Spring Cloud Data Flow. If you intend to use the `aggregate-counter-sink`, then `redis` becomes required, and you're expected to have your own running `redis` cluster

A.3 Message Bus to Binders

Terminology wise, in Spring Cloud Data Flow, the message bus implementation is commonly referred to as binders.

Message Bus

Similar to Spring XD, there's an abstraction available to extend the binder interface. By default, we take the opinionated view of Apache Kafka and RabbitMQ as the production-ready binders and are available as GA releases. We also have an experimental version of the Gemfire binder.

Binders

Selecting a binder is as simple as providing the right binder dependency in the classpath. If you're to choose Kafka as the binder, you'd register stream applications that are pre-built with Kafka binder in it. If you were to create a custom application with Kafka binder, you'd add the following dependency in the classpath.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-kafka</artifactId>
    <version>1.0.2.RELEASE</version>
</dependency>
```

- Spring Cloud Stream supports Apache Kafka, RabbitMQ and an experimental Gemfire binder implementation. All binder implementations are maintained and managed in their individual repositories

- Every Stream/Task application can be built with a binder implementation of your choice. All the out-of-the-box applications are pre-built for both Kafka and Rabbit and they're readily available for use as maven artifacts [stream / task] or docker images [stream / task] Changing the binder requires selecting the right binder dependency. Alternatively, you can download the pre-built application from this version of Spring Initializr with the desired "binder-starter" dependency

Named Channels

Fundamentally, all the messaging channels are backed by pub/sub semantics. Unlike Spring XD, the messaging channels are backed only by `topics` or `topic-exchange` and there's no representation of `queues` in the new architecture.

- `${xd.module.index}` is not supported anymore; instead, you can directly interact with named destinations
- `stream.index` changes to `:<stream-name>.<label/app-name>`
  - *for instance:* `ticktock.0` changes to `:ticktock.time`
- "topic/queue" prefixes are not required to interact with named-channels
  - *for instance:* `topic:foo` changes to `:foo`
  - *for instance:* `stream create stream1 --definition ":foo > log"`

Directed Graphs

If you're building non-linear streams, you could take advantage of named destinations to build directed graphs.

for instance, in Spring XD:

```
stream create f --definition "queue:foo > transform --expression=payload+'-foo' | log" --deploy
stream create b --definition "queue:bar > transform --expression=payload+'-bar' | log" --deploy
stream create r --definition "http | router --expression=payload.contains('a')?'queue:foo':'queue:bar'"
 --deploy
```

for instance, in Spring Cloud Data Flow:

```
stream create f --definition ":foo > transform --expression=payload+'-foo' | log" --deploy
stream create b --definition ":bar > transform --expression=payload+'-bar' | log" --deploy
stream create r --definition "http | router --expression=payload.contains('a')?':foo':':bar'" --deploy
```

A.4 Batch to Tasks

A Task by definition, is any application that does not run forever, including Spring Batch jobs, and they end/stop at some point. Task applications can be majorly used for on-demand use-cases such as database migration, machine learning, scheduled operations etc. Using Spring Cloud Task, users can build Spring Batch jobs as microservice applications.

- Spring Batch jobs from Spring XD are being refactored to Spring Boot applications a.k.a link: Spring Cloud Task applications
- Unlike Spring XD, these "Tasks" don't require explicit deployment; instead, a task is ready to be launched directly once the definition is declared

A.5 Shell/DSL Commands

Old Command	New Command
module upload	app register / app import
module list	app list
module info	app info
admin config server	dataflow config server
job create	task create
job launch	task launch
job list	task list
job status	task status
job display	task display
job destroy	task destroy
job execution list	task execution list
runtime modules	runtime apps

A.6 REST-API

Old API	New API
/modules	/apps
/runtime/modules	/runtime/apps
/runtime/modules/(moduleId}	/runtime/apps/{appId}
/jobs/definitions	/task/definitions
/jobs/deployments	/task/deployments

A.7 UI / Flo

The Admin-UI is now renamed as Dashboard. The URI for accessing the Dashboard is changed from localhost:9393/admin-ui to localhost:9393/dashboard

- *(New)* Apps: Lists all the registered applications that are available for use. This view includes informational details such as the URI and the properties supported by each application. You can also register/unregister applications from this view
- Runtime: Container changes to Runtime. The notion of `xd-container` is gone, replaced by out-of-the-box applications running as autonomous Spring Boot applications. The Runtime tab displays the applications running in the runtime platforms (*implementations:* cloud foundry, apache yarn, apache mesos, or kubernetes). You can click on each application to review relevant details about the application such as where it is running with, and what resources etc.

- Spring Flo is now an OSS product. Flo for Spring Cloud Data Flow's "Create Stream", the designer-tab comes pre-built in the Dashboard
- *(New)* Tasks:
  - The sub-tab "Modules" is renamed to "Apps"
  - The sub-tab "Definitions" lists all the Task definitions, including Spring Batch jobs that are orchestrated as Tasks
  - The sub-tab "Executions" lists all the Task execution details similar to Spring XD's Job executions

A.8 Architecture Components

Spring Cloud Data Flow comes with a significantly simplified architecture. In fact, when compared with Spring XD, there are less peripherals that are necessary to operationalize Spring Cloud Data Flow.

ZooKeeper

ZooKeeper is not used in the new architecture.

RDBMS

Spring Cloud Data Flow uses an RDBMS instead of Redis for stream/task definitions, application registration, and for job repositories.The default configuration uses an embedded H2 instance, but Oracle, SqlServer, MySQL/MariaDB, PostgreSQL, H2, and HSQLDB databases are supported. To use Oracle and SqlServer you will need to create your own Data Flow Server using Spring Initializr and add the appropriate JDBC driver dependency.

Redis

Running a Redis cluster is only required for analytics functionality. Specifically, when the `counter-sink`, `field-value-counter-sink`, or `aggregate-counter-sink` applications are used, it is expected to also have a running instance of Redis cluster.

Cluster Topology

Spring XD's `xd-admin` and `xd-container` server components are replaced by stream and task applications themselves running as autonomous Spring Boot applications. The applications run natively on various platforms including Cloud Foundry, Apache YARN, Apache Mesos, or Kubernetes. You can develop, test, deploy, scale +/-, and interact with (Spring Boot) applications individually, and they can evolve in isolation.

A.9 Central Configuration

To support centralized and consistent management of an application's configuration properties, Spring Cloud Config client libraries have been included into the Spring Cloud Data Flow server as well as the Spring Cloud Stream applications provided by the Spring Cloud Stream App Starters. You can also pass common application properties to all streams when the Data Flow Server starts.

A.10 Distribution

Spring Cloud Data Flow is a Spring Boot application. Depending on the platform of your choice, you can download the respective release uber-jar and deploy/push it to the runtime platform (cloud foundry, apache yarn, kubernetes, or apache mesos). For example, if you're running Spring Cloud Data Flow on Cloud Foundry, you'd download the Cloud Foundry server implementation and do a `cf push` as explained in the reference guide.

A.11 Hadoop Distribution Compatibility

The `hdfs-sink` application builds upon Spring Hadoop 2.4.0 release, so this application is compatible with following Hadoop distributions.

- Cloudera - cdh5
- Pivotal Hadoop - phd30
- Hortonworks Hadoop - hdp24
- Hortonworks Hadoop - hdp23
- Vanilla Hadoop - hadoop26
- Vanilla Hadoop - 2.7.x (default)

A.12 YARN Deployment

Spring Cloud Data Flow can be deployed and used with Apche YARN in two different ways.

- Deploy the server directly in a YARN cluster
- Leverage Apache Ambari plugin to provision Spring Cloud Data Flow as a service

A.13 Use Case Comparison

Let's review some use-cases to compare and contrast the differences between Spring XD and Spring Cloud Data Flow.

Use Case #1

(*It is assumed both XD and SCDF distributions are already downloaded*)

Description: Simple `ticktock` example using local/singlenode.

Spring XD	Spring Cloud Data Flow
Start `xd-singlenode` server from CLI `# xd-singlenode`	Start a binder of your choice Start `local-server` implementation of SCDF from the CLI `# java -jar spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar`
Start `xd-shell` server from the CLI `# xd-shell`	Start `dataflow-shell` server from the CLI

Spring XD	Spring Cloud Data Flow
	# java -jar spring-cloud-dataflow-shell-1.0.0.BUILD-SNAPSHOT.jar
Create ticktock stream xd:>stream create ticktock --definition “time | log” --deploy	Create ticktock stream dataflow:>stream create ticktock --definition “time | log” --deploy
Review ticktock results in the xd-singlenode server console	Review ticktock results by tailing the ticktock.log/stdout_log application logs

Use Case #2

(*It is assumed both XD and SCDF distributions are already downloaded*)

Description: Stream with custom module/application.

Spring XD	Spring Cloud Data Flow
Start xd-singlenode server from CLI # xd-singlenode	Start a binder of your choice Start local-server implementation of SCDF from the CLI # java -jar spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar
Start xd-shell server from the CLI # xd-shell	Start dataflow-shell server from the CLI # java -jar spring-cloud-dataflow-shell-1.0.0.BUILD-SNAPSHOT.jar
Register custom “processor” module to transform payload to a desired format xd:>module upload --name toupper --type processor --file <CUSTOM_JAR_FILE_LOCATION>	Register custom “processor” application to transform payload to a desired format dataflow:>app register --name toupper --type processor --uri <MAVEN_URI_COORDINATES>
Create a stream with custom module xd:>stream create testupper --definition “http | toupper | log” --deploy	Create a stream with custom application dataflow:>stream create testupper --definition “http | toupper | log” --deploy
Review results in the xd-singlenode server console	Review results by tailing the testupper.log/stdout_log application logs

Use Case #3

(*It is assumed both XD and SCDF distributions are already downloaded*)

Description: Simple batch-job.

Spring XD	Spring Cloud Data Flow
Start `xd-singlenode` server from CLI `# xd-singlenode`	Start `local-server` implementation of SCDF from the CLI `# java -jar spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar`
Start `xd-shell` server from the CLI `# xd-shell`	Start `dataflow-shell` server from the CLI `# java -jar spring-cloud-dataflow-shell-1.0.0.BUILD-SNAPSHOT.jar`
Register custom “batch-job” module `xd:>module upload --name simple-batch --type job --file <CUSTOM_JAR_FILE_LOCATION>`	Register custom “batch-job” as task application `dataflow:>app register --name simple-batch --type task --uri <MAVEN_URI_COORDINATES>`
Create a job with custom batch-job module `xd:>job create batchtest --definition “simple-batch”`	Create a task with custom batch-job application `dataflow:>task create batchtest --definition “simple-batch”`
Deploy job `xd:>job deploy batchtest`	NA
Launch job `xd:>job launch batchtest`	Launch task `dataflow:>task launch batchtest`
Review results in the `xd-singlenode` server console as well as Jobs tab in UI (executions sub-tab should include all step details)	Review results by tailing the `batchtest/stdout_log` application logs as well as Task tab in UI (executions sub-tab should include all step details)

Appendix B. Building

B.1 Basic Compile and Test

To build the source you will need to install JDK 1.8.

The build uses the Maven wrapper so you don't have to install a specific version of Maven. To enable the tests for Redis you should run the server before bulding. See below for more information on how run Redis.

The main build command is

```
$ ./mvnw clean install
```

You can also add '-DskipTests' if you like, to avoid running the tests.

Note

You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.

Note

Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

The projects that require middleware generally include a `docker-compose.yml`, so consider using Docker Compose to run the middeware servers in Docker containers. See the README in the scripts demo repository for specific instructions about the common cases of mongo, rabbit and redis.

B.2 Documentation

There is a "full" profile that will generate documentation. You can build just the documentation by executing

```
$ ./mvnw package -DskipTests=true -P full -pl spring-cloud-dataflow-server-cloudfoundry-docs -am
```

B.3 Working with the code

If you don't have an IDE preference we would recommend that you use Spring Tools Suite or Eclipse when working with the code. We use the m2eclipe eclipse plugin for maven support. Other IDEs and tools should also work without issue.

Importing into eclipse with m2eclipse

We recommend the m2eclipe eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".

Unfortunately m2e does not yet support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the `.settings.xml` file for the projects. If you do not do this you may see many different errors related to the POMs in the projects. Open your Eclipse preferences, expand the Maven preferences, and select User Settings. In the User Settings field click Browse and navigate to the Spring Cloud project you imported selecting the `.settings.xml` file in that project. Click Apply and then OK to save the preference changes.

Note

Alternatively you can copy the repository settings from [`.settings.xml`] into your own `~/.m2/settings.xml`.

Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.