Spring Cloud Data Flow Reference Guide

1.0.0.RC1

Sabby Anandan, Marius Bogoevici, Eric Bottard, Mark Fisher, Ilayaperumal Gopinathan, Gunnar Hillert, Mark Pollack, Patrick Peralta, Glenn Renfro, Thomas Risberg, Dave Syer, David Turanski, Janne Valkealahti

Table of Contents

Part I. Preface

1. About the documentation

The Spring Cloud Data Flow reference guide is available as html, pdf and epub documents. The latest copy is available at docs.spring.io/spring-cloud-dataflow/docs/current-SNAPSHOT/reference/html/.

2. Getting help

Having trouble with Spring Cloud Data Flow, We’d like to help!

- Ask a question - we monitor stackoverflow.com for questions tagged with `spring-cloud-dataflow`.
- Report bugs with Spring Cloud Data Flow at github.com/spring-cloud/spring-cloud-dataflow/issues.

Note

All of Spring Cloud Data Flow is open source, including the documentation! If you find problems with the docs; or if you just want to improve them, please get involved.

Part II. Spring Cloud Data Flow Overview

This section provides a brief overview of the Spring Cloud Data Flow reference documentation. Think of it as map for the rest of the document. You can read this reference guide in a linear fashion, or you can skip sections if something doesn’t interest you.

3. Introducing Spring Cloud Data Flow

Spring Cloud Data Flow is a cloud native programming and operating model for composable data microservices on modern runtimes. 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.

Spring Cloud Data Flow is the cloud native redesign of Spring XD – a project that aimed to simplify development of Big Data applications. The streaming and batch modules from Spring XD are refactored into Spring Boot data microservice applications that are now autonomous deployment units and they can "natively" run in modern runtimes such as Cloud Foundry, Apache YARN, Apache Mesos, and Kubernetes.

Spring Cloud Data Flow offers a collection of patterns and best practices for microservices-based distributed streaming and batch data pipelines.

3.1 Features

- Orchestrate applications across a variety of distributed modern runtimes including: Cloud Foundry, Apache YARN, Apache Mesos, and Kubernetes
- Separate runtime dependencies backed by **spring profiles**
- Consume stream and batch data-microservices as maven dependencies
- Develop using: DSL, Shell, REST-APIs, Data Flow Server UI, and Flo
- Take advantage of metrics, health checks and remote management of each data microservice
- Scale stream and batch pipelines without interrupting data flows

4. Spring Cloud Data Flow Architecture

The architecture for Spring Cloud Data Flow is separated into a number of distinct components.

4.1 Components

The Core domain model includes the concept of a **stream** that is a composition of spring-cloud-stream apps in a linear pipeline from a **source** to a **sink**, optionally including **processor** apps in between. The domain also includes the concept of a **task**, which may be any process that does not run indefinitely, including Spring Batch jobs.

The App Registry maintains the set of available apps, and their mappings to a URI. For example, if relying on Maven coordinates, the URI would be of the format: `maven://<groupId>:<artifactId>:<version>`

The Data Flow Server Core provides the REST API and UI to be used in combination with an implementation of the Deployer SPI when creating a Data Flow Server for a given deployment environment.

The Shell connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream and managing its lifecycle.

Several Data Flow Server implementations exist, covering a range of runtime environments:

- Local (intended for development only)
- Cloud Foundry
- Apache Yarn
- Apache Mesos
- Kubernetes

As mentioned above, the Spring Cloud Data Flow Server implementations all rely upon corresponding implementations of the Spring Cloud Deployer SPI, which provides the abstraction layer for deploying the apps of a given stream or task. The following are links to the deployer SPI projects that correspond to the Data Flow Servers listed above:

- Local
- Cloud Foundry
- Apache Yarn
- Apache Mesos
- Kubernetes = Getting started

5. System Requirements

You need Java installed (Java 7 or better, we recommend Java 8), and to build, you need to have Maven installed as well.

You need to have an RDBMS for storing stream, task and app states in the database. The `local` dataflow server by default uses embedded H2 database for this.

You also need to have Redis running if you are running any streams that involve analytics applications. Redis may also be required run the unit/integration tests.

For the deployed streams and tasks to communicate, either RabbitMQ or Kafka needs to be installed. The local server registers sources, sink, processors and tasks the are published from the Spring Cloud Stream App Starters and Spring Cloud Task App Starters repository. By default the server registers these applications that use Kafka, but setting the property `binding` to `rabbit` will register a list of applications that use RabbitMQ as the message broker.

6. Controlling features with Dataflow server

Dataflow server offers specific set of features that can be enabled/disabled when launching. These features include all the lifecycle operations, REST endpoints (server, client implementations including Shell and the UI) for:

1. Streams
2. Tasks
3. Analytics

One can enable, disable these features by setting the following boolean properties when launching the dataflow server:

- `spring.cloud.dataflow.features.streams-enabled`
- `spring.cloud.dataflow.features.tasks-enabled`
- `spring.cloud.dataflow.features.analytics-enabled`

By default, all these features are enabled for `local` dataflow server.

The REST endpoint `/features` provides information on the features enabled/disabled.

7. Deploying Spring Cloud Data Flow

7.1 Deploying 'local'

1. Download the Spring Cloud Data Flow Server and Shell apps:

```
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-server-
local/1.0.0.RC1/spring-cloud-dataflow-server-local-1.0.0.RC1.jar

wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-shell/1.0.0.RC1/
spring-cloud-dataflow-shell-1.0.0.RC1.jar
```

2. Launch the Data Flow Server

   a. Since the Data Flow Server is a Spring Boot application, you can run it just by using `java -jar`.

   ```
   $ java -jar spring-cloud-dataflow-server-local-1.0.0.RC1.jar
   ```

   b. Running with Custom Maven Settings and/or Behind a Proxy If you want to override specific maven configuration properties (remote repositories, etc.) and/or run the Data Flow Server behind a proxy, you need to specify those properties as command line arguments when starting the Data Flow Server. For example:

   ```
   $ java -jar spring-cloud-dataflow-server-local-1.0.0.RC1.jar --maven.localRepository=mylocal --
   maven.remoteRepositories=repo1,repo2 --maven.offline=true
   --maven.proxy.protocol=https --maven.proxy.host=host1 --maven.proxy.port=8090 --
   maven.proxy.non_proxy_hosts='host2|host3' --maven.proxy.auth.username=user1 --
   maven.proxy.auth.password=passwd
   ```

   By default, the protocol is set to `http`. You can omit the auth properties if the proxy doesn't need a username and password.

   By default, the maven `localRepository` is set to `${user.home}/.m2/repository/`, and `repo.spring.io/libs-snapshot` will be the only remote repository.

   You can also use environment variables to specify the maven/proxy properties:

   ```
   export MAVEN_LOCAL_REPOSITORY=mylocalMavenRepo
   export MAVEN_REMOTE_REPOSITORIES=repo1,repo2
   export MAVEN_OFFLINE=true
   export MAVEN_PROXY_PROTOCOL=https
   export MAVEN_PROXY_HOST=host1
   export MAVEN_PROXY_PORT=8090
   export MAVEN_PROXY_NON_PROXY_HOSTS='host2|host3'
   export MAVEN_PROXY_AUTH_USERNAME=user1
   export MAVEN_PROXY_AUTH_PASSWORD=passwd
   ```

3. Launch the shell:

```
$ java -jar spring-cloud-dataflow-shell-1.0.0.RC1.jar
```

If the Data Flow Server and shell are not running on the same host, point the shell to the Data Flow server:

```
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 Kafka 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-kafka-maven
```

4. 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 --server.port=9000 | log" --deploy
```

Note

You will need to wait a little while until the apps are actually deployed successfully before posting data. Look in the log file of the Data Flow server for the location of the log files for the `http` and `log` applications. Tail the log file for each application to verify the application has started.

Now post some data

```
dataflow:> http post --target http://localhost:9000 --data "hello world"
```

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

8. 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.

8.1 Enabling HTTPS

By default, the dashboard, management, and health endpoints use HTTP as a transport. You can switch to HTTPS easily, by adding a certificate to your configuration in `application.yml`.

```
server:
  port: 8443                                        ❶
  ssl:
    key-alias: yourKeyAlias                         ❷
    key-store: path/to/keystore                     ❸
    key-store-password: yourKeyStorePassword        ❹
    key-password: yourKeyPassword                   ❺
    trust-store: path/to/trust-store                ❻
    trust-store-password: yourTrustStorePassword    ❼
```

❶ As the default port is `9393`, you may choose to change the port to a more common HTTPs-typical port.

❷ The alias (or name) under which the key is stored in the keystore.

❸ The path to the keystore file. Classpath resources may also be specified, by using the classpath prefix: `classpath:path/to/keystore`

❹ The password of the keystore.

❺ The password of the key.

❻ The path to the truststore file. Classpath resources may also be specified, by using the classpath prefix: `classpath:path/to/trust-store`

❼ The password of the trust store.

Note

If HTTPS is enabled, it will completely replace HTTP as the protocol over which the REST endpoints and the Data Flow Dashboard interact. Plain HTTP requests will fail - therefore, make sure that you configure your Shell accordingly.

Using Self-Signed Certificates

For testing purposes or during development it might be convenient to create self-signed certificates. To get started, execute the following command to create a certificate:

```
$ keytool -genkey -alias dataflow -keyalg RSA -keystore dataflow.keystore \
          -validity 3650 -storetype JKS \
          -dname "CN=localhost, OU=Spring, O=Pivotal, L=Kailua-Kona, ST=HI, C=US"  ❶
          -keypass dataflow -storepass dataflow
```

❶ *CN* is the only important parameter here. It should match the domain you are trying to access, e.g. `localhost`.

Then add the following to your `application.yml` file:

```
server:
  port: 8443
  ssl:
    enabled: true
    key-alias: dataflow
    key-store: "/your/path/to/dataflow.keystore"
    key-store-type: jks
    key-store-password: dataflow
    key-password: dataflow
```

This is all that’s needed for the Data Flow Server. Once you start the server, you should be able to access it via https://localhost:8443/. As this is a self-signed certificate, you will hit a warning in your browser, that you need to ignore.

This issue also is relevant for the Data Flow Shell. Therefore additional steps are necessary to make the Shell work with self-signed certificates. First, we need to export the previously created certificate from the keystore:

```
$ keytool -export -alias dataflow -keystore dataflow.keystore -file dataflow_cert -storepass dataflow
```

Next, we need to create a truststore which the Shell will use:

```
$ keytool -importcert -keystore dataflow.truststore -alias dataflow -storepass dataflow -file
 dataflow_cert -noprompt
```

Now, you are ready to launch the Data Flow Shell using the following JVM arguments:

```
$ java -Djavax.net.ssl.trustStorePassword=dataflow \
       -Djavax.net.ssl.trustStore=/path/to/dataflow.truststore \
       -Djavax.net.ssl.trustStoreType=jks \
       -jar spring-cloud-dataflow-shell-1.0.0.RC1.jar
```

Tip

In case you run into trouble establishing a connection via SSL, you can enable additional logging by using and setting the `javax.net.debug` JVM argument to `ssl`.

Don’t forget to target the Data Flow Server with:

```
dataflow:> dataflow config server https://localhost:8443/
```

8.2 Enabling Authentication

By default, the REST endpoints (administration, management and health), as well as the Dashboard UI do not require authenticated access. However, authentication can be provided via OAuth 2.0, thus allowing you to also integrate Spring Cloud Data Flow into Single Sign On (SSO) environments. The following 2 OAuth2 Grant Types will be used:

- *Authorization Code* - Used for the GUI (Browser) integration. You will be redirected to your OAuth Service for authentication
- *Password* - Used by the shell (And the REST integration), so you can login using username and password

The REST endpoints are secured via Basic Authentication but will use the Password Grand Type under the covers to authenticate with your OAuth2 service.

Note

When authentication is set up, it is strongly recommended to enable HTTPS as well, especially in production environments.

You can turn on authentication by adding the following to `application.yml` or via environment variables:

```
security:
  basic:
    enabled: true                                                  ❶
    realm: Spring Cloud Data Flow                                  ❷
  oauth2:                                                          ❸
    client:
      client-id: myclient
      client-secret: mysecret
      access-token-uri: http://127.0.0.1:9999/oauth/token
      user-authorization-uri: http://127.0.0.1:9999/oauth/authorize
    resource:
      user-info-uri: http://127.0.0.1:9999/me
```

❶ Must be set to `true` for security to be enabled.

❷ The realm for Basic authentication

❸ OAuth Configuration Section

Note

As of version 1.0 Spring Cloud Data Flow does not provide finer-grained authorization. Thus, once you are logged in, you have full access to all functionality.

You can verify that basic authentication is working properly using *curl*:

```
$ curl -u myusername:mypassword http://localhost:9393/
```

As a result you should see a list of available REST endpoints.

Authentication and Cloud Foundry

When deploying Spring Cloud Data Flow to Cloud Foundry, we 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.

Part III. Streams

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

9. 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.

10. 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.

11. 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:http-
log-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 and Task 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
- Maven based Task Applications: bit.ly/task-applications-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
- Docker based Task Applications: bit.ly/task-applications-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.

11.1 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
```

12. 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
```

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"
```

Important

See Chapter 19, *Using Labels in a Stream*.

13. 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.

14. 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
```

15. 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.

16. 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
```

17. 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.

18. 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.

19. 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
```

20. 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.

21. 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.

21.1 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 `stream.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 IV. Tasks

This section goes into more detail about how you can work with Spring Cloud Tasks. It covers topics such as creating and running task applications.

If you’re just starting out with Spring Cloud Data Flow, you should probably read the *Getting Started* guide before diving into this section.

22. Introducing Spring Cloud Task

A task executes a process on demand. In this case a task is a Spring Boot application that is annotated with `@EnableTask`. Hence a user launches a task that performs a certain process, and once complete the task ends. An example of a task would be a boot application that exports data from a JDBC repository to an HDFS instance. Tasks record the start time and the end time as well as the boot exit code in a relational database. The task implementation is based on the Spring Cloud Task project.

23. The Lifecycle of a task

Before we dive deeper into the details of creating Tasks, we need to understand the typical lifecycle for tasks in the context of Spring Cloud Data Flow:

1. Register a Task App
2. Create a Task Definition
3. Launch a Task
4. Task Execution
5. Destroy a Task Definition

23.1 Registering a Task Application

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

```
dataflow:>app register --name task1 --type task --uri maven://com.example:mytask:1.0.2

dataflow:>app register --name task2 --type task --uri file:///Users/example/mytask-1.0.2.jar

dataflow:>app register --name task3 --type task --uri http://example.com/mytask-1.0.2.jar
```

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

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

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, this would be a valid properties file:

```
task.foo=file:///tmp/foo.jar
task.bar=file:///tmp/bar.jar
```

Then use the `app import` command and provide the location of the properties file via `--uri`:

```
app import --uri file:///tmp/task-apps.properties
```

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 task app is already registered with the provided name, it will not be overridden by default. If you would like to override the pre-existing task 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.

23.2 Creating a Task

Create a Task Definition from a Task App by providing a definition name as well as properties that apply to the task execution. Creating a task definition can be done via the restful API or the shell. To create a task definition using the shell, use the `task create` command to create the task definition. For example:

```
dataflow:>task create mytask --definition "timestamp --format=\"yyyy\""
 Created new task 'mytask'
```

A listing of the current task definitions can be obtained via the restful API or the shell. To get the task definition list using the shell, use the `task list` command.

23.3 Launching a Task

An adhoc task can be launched via the restful API or via the shell. To launch an ad-hoc task via the shell use the `task launch` command. For Example:

```
dataflow:>task launch mytask
 Launched task 'mytask'
```

23.4 Reviewing Task Executions

Once the task is launched the state of the task is stored in a relational DB. The state includes:

- Task Name
- Start Time
- End Time
- Exit Code
- Exit Message
- Last Updated Time
- Parameters

A user can check the status of their task executions via the restful API or by the shell. To display the latest task executions via the shell use the `task execution list` command.

To get a list of task executions for just one task definition, add `--name` and the task definition name, for example `task execution list --name foo`. To retrieve full details for a task execution use the `task display` command with the id of the task execution , for example `task display --id 549`.

23.5 Destroying a Task

Destroying a Task Definition will remove the definition from the definition repository. This can be done via the restful API or via the shell. To destroy a task via the shell use the `task destroy` command. For Example:

```
dataflow:>task destroy mytask
 Destroyed task 'mytask'
```

The task execution information for previously launched tasks for the definition will remain in the task repository.

Note: This will not stop any currently executing tasks for this definition, this just removes the definition.

24. Task Repository

Out of the box Spring Cloud Data Flow offers an embedded instance of the H2 database. The H2 is good for development purposes but is not recommended for production use.

24.1 Configuring the Task Execution Repository

To add a driver for the database that will store the Task Execution information, a dependency for the driver will need to be added to a maven pom file and the Spring Cloud Data Flow will need to be rebuilt. Since Spring Cloud Data Flow is comprised of an SPI for each environment it supports, please review the SPI's documentation on which POM should be updated to add the dependency and how to build. This document will cover how to setup the dependency for local SPI.

Local

1. Open the spring-cloud-dataflow-server-local/pom.xml in your IDE.
2. In the `dependencies` section add the dependency for the database driver required. In the sample below postgresql has been chosen.

```
<dependencies>
...
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
    </dependency>
...
</dependencies>
```

3. Save the changed pom.xml
4. Build the application as described here: Building Spring Cloud Data Flow

24.2 Datasource

To configure the datasource Add the following properties to the dataflow-server.yml or via environment variables:

a. spring.datasource.url

b. spring.datasource.username

c. spring.datasource.password

d. spring.datasource.driver-class-name

For example adding postgres would look something like this:

- Environment variables:

```
export spring_datasource_url=jdbc:postgresql://localhost:5432/mydb
export spring_datasource_username=myuser
export spring_datasource_password=mypass
export spring_datasource_driver-class-name="org.postgresql.Driver"
```

- dataflow-server.yml

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/mydb
    username: myuser
    password: mypass
    driver-class-name:org.postgresql.Driver
```

25. Subscribing to Task/Batch Events

You can also tap into various task/batch events when the task is launched. If the task is enabled to generate task and/or batch events (with the additional dependencies `spring-cloud-task-stream` and `spring-cloud-stream-binder-kafka`, in the case of Kafka as the binder), those events are published during the task lifecycle. By default, the destination names for those published events on the broker (rabbit, kafka etc.,) are the event names themselves (for instance: `task-events`, `job-execution-events` etc.,).

```
dataflow:>task create myTask --definition “myBatchJob"
dataflow:>task launch myTask
dataflow:>stream create task-event-subscriber1 --definition ":task-events > log" --deploy
```

You can control the destination name for those events by specifying explicit names when launching the task such as:

```
dataflow:>task launch myTask --properties "spring.cloud.stream.bindings.task-
events.destination=myTaskEvents"
dataflow:>stream create task-event-subscriber2 --definition ":myTaskEvents > log" --deploy
```

The default Task/Batch event and destination names on the broker are enumerated below:

Table 25.1. Task/Batch Event Destinations

Event	Destination
Task events	`task-events`
Job Execution events	`job-execution-events`
Step Execution events	`step-execution-events`
Item Read events	`item-read-events`
Item Process events	`item-process-events`
Item Write events	`item-write-events`
Skip events	`skip-events`

Part V. Dashboard

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

26. 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
- **Tasks** List, create, launch and destroy Task Definitions
- **Jobs** Perform Batch Job related functions
- **Analytics** Create data visualizations for the various analytics applications

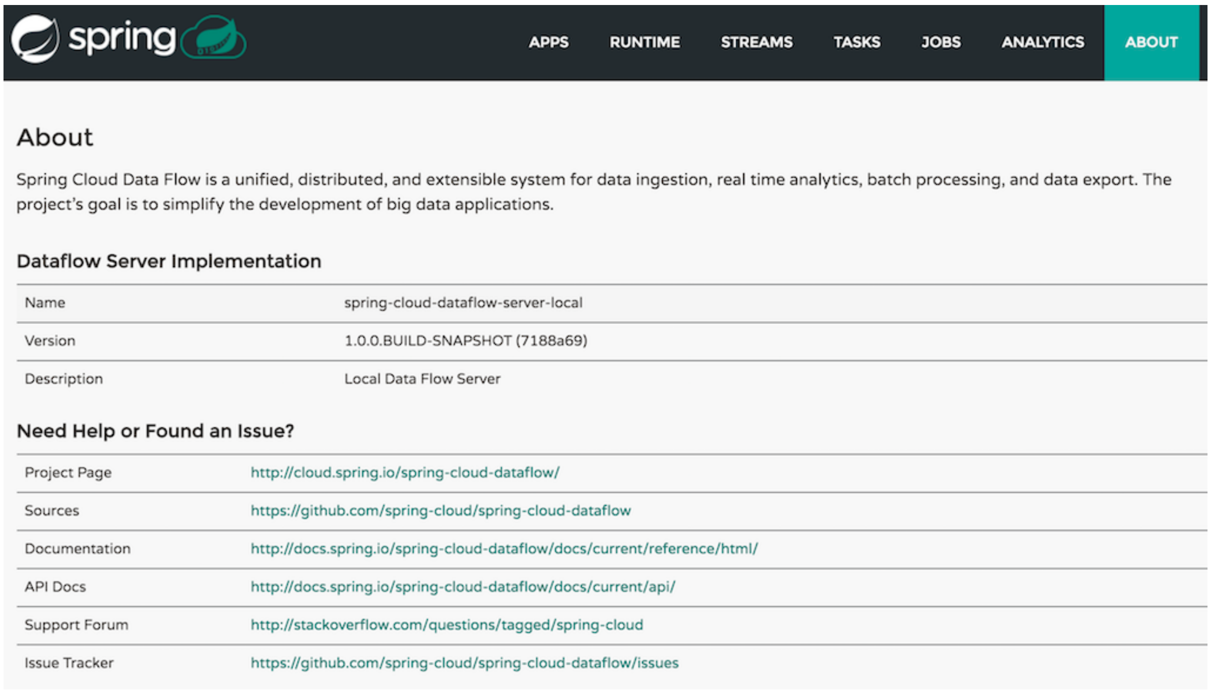
Upon starting Spring Cloud Data Flow, the Dashboard is available at:

`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



About

Spring Cloud Data Flow is a unified, distributed, and extensible system for data ingestion, real time analytics, batch processing, and data export. The project's goal is to simplify the development of big data applications.

Dataflow Server Implementation

Name	spring-cloud-dataflow-server-local
Version	1.0.0.BUILD-SNAPSHOT (7188a69)
Description	Local Data Flow Server

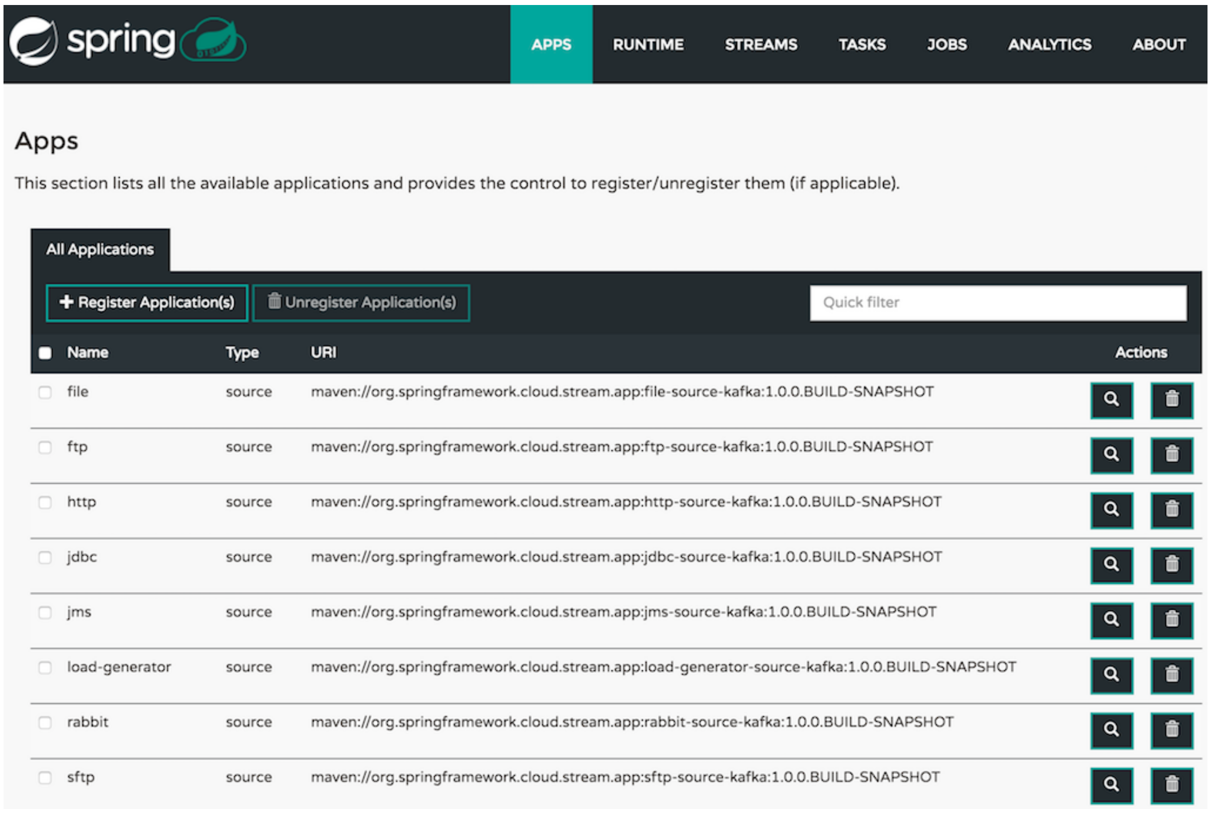
Need Help or Found an Issue?

Project Page	http://cloud.spring.io/spring-cloud-dataflow/
Sources	https://github.com/spring-cloud/spring-cloud-dataflow
Documentation	http://docs.spring.io/spring-cloud-dataflow/docs/current/reference/html/
API Docs	http://docs.spring.io/spring-cloud-dataflow/docs/current/api/
Support Forum	http://stackoverflow.com/questions/tagged/spring-cloud
Issue Tracker	https://github.com/spring-cloud/spring-cloud-dataflow/issues

Figure 26.1. The Spring Cloud Data Flow Dashboard

27. 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.



Apps

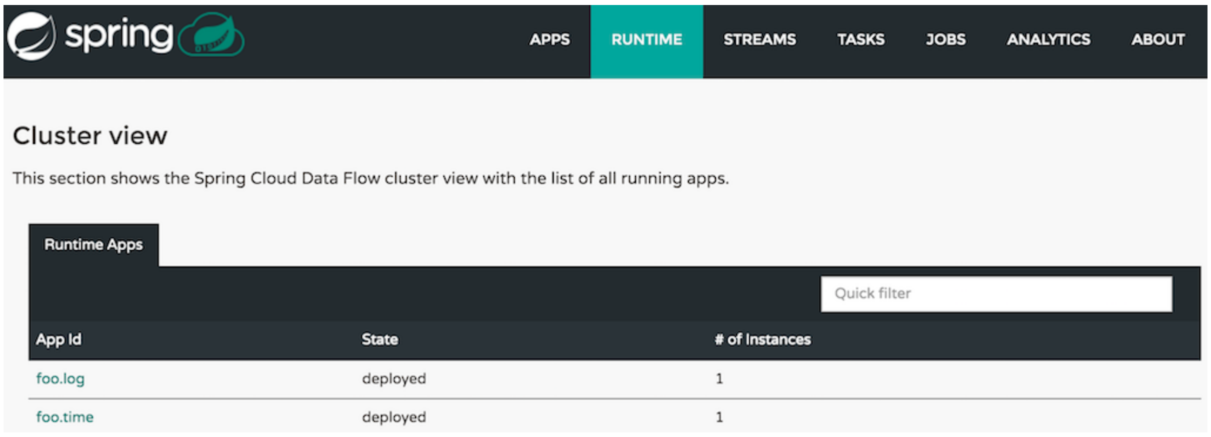
This section lists all the available applications and provides the control to register/unregister them (if applicable).

Name	Type	URI	Actions
file	source	maven://org.springframework.cloud.stream.app:file-source-kafka:1.0.0.BUILD-SNAPSHOT	
ftp	source	maven://org.springframework.cloud.stream.app:ftp-source-kafka:1.0.0.BUILD-SNAPSHOT	
http	source	maven://org.springframework.cloud.stream.app:http-source-kafka:1.0.0.BUILD-SNAPSHOT	
jdbc	source	maven://org.springframework.cloud.stream.app:jdbc-source-kafka:1.0.0.BUILD-SNAPSHOT	
jms	source	maven://org.springframework.cloud.stream.app:jms-source-kafka:1.0.0.BUILD-SNAPSHOT	
load-generator	source	maven://org.springframework.cloud.stream.app:load-generator-source-kafka:1.0.0.BUILD-SNAPSHOT	
rabbit	source	maven://org.springframework.cloud.stream.app:rabbit-source-kafka:1.0.0.BUILD-SNAPSHOT	
sftp	source	maven://org.springframework.cloud.stream.app:sftp-source-kafka:1.0.0.BUILD-SNAPSHOT	

Figure 27.1. List of Available Applications

28. 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.



Cluster view

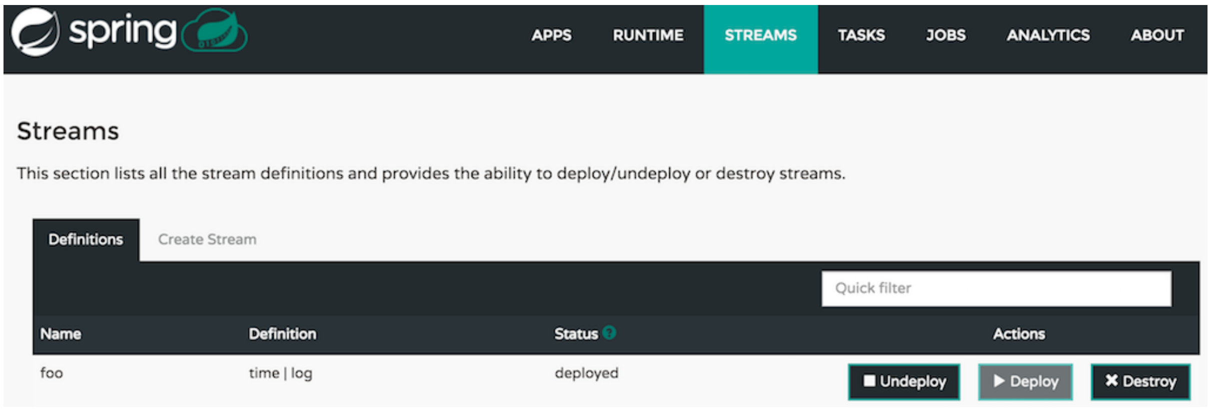
This section shows the Spring Cloud Data Flow cluster view with the list of all running apps.

App Id	State	# of Instances
foo.log	deployed	1
foo.time	deployed	1

Figure 28.1. List of Running Applications

29. 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**.



Streams

This section lists all the stream definitions and provides the ability to deploy/undeploy or destroy streams.

Name	Definition	Status	Actions
foo	time | log	deployed	Undeploy Deploy Destroy

Figure 29.1. List of Stream Definitions

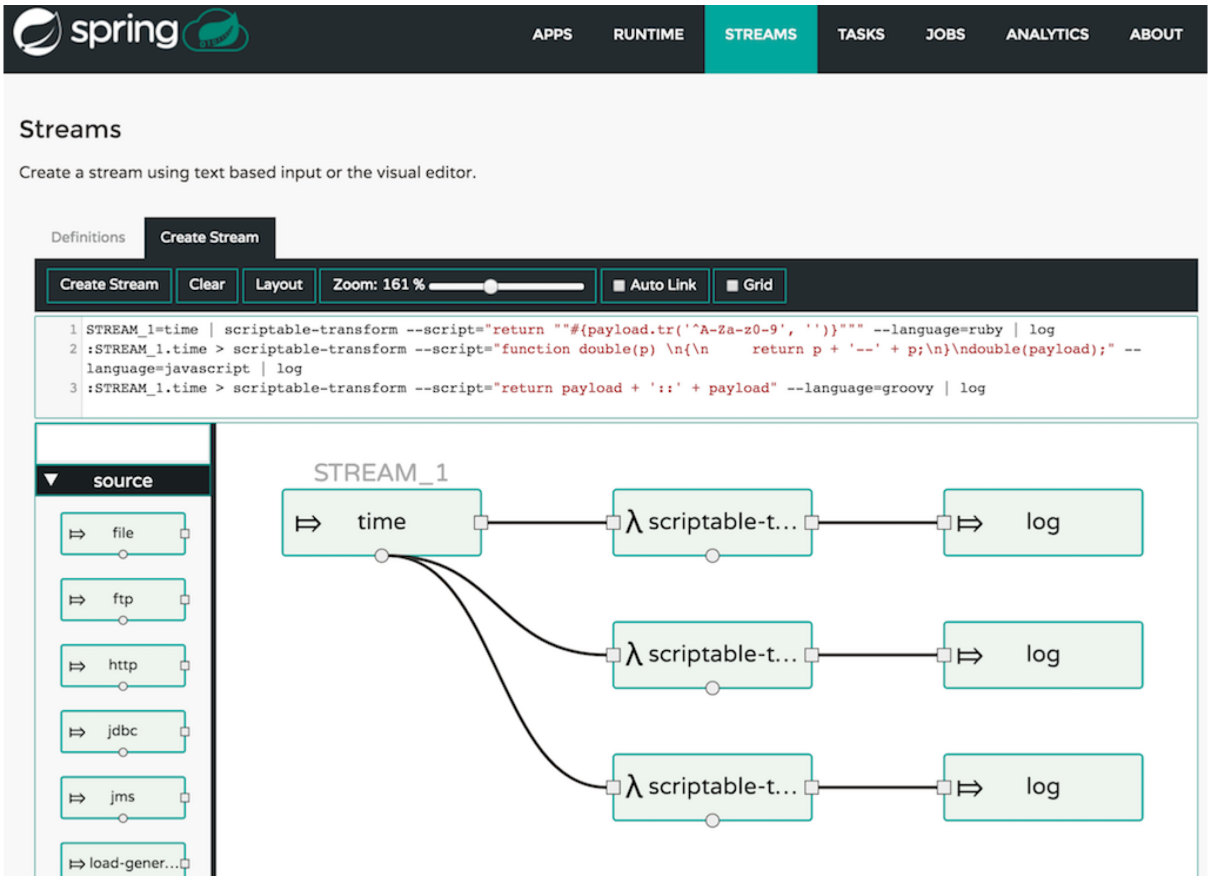
30. Create Stream

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.



```
STREAM_1=time | scriptable-transform --script="return ""#{payload.tr('^A-Za-z0-9', '')}""" --language=ruby | log
:STREAM_1.time > scriptable-transform --script="function double(p) \n{\n     return p + '--' + p;\n}\ndouble(payload);" --
language=javascript | log
:STREAM_1.time > scriptable-transform --script="return payload + '::' + payload" --language=groovy | log
```

Figure 30.1. Flo for Spring Cloud Data Flow

31. Tasks

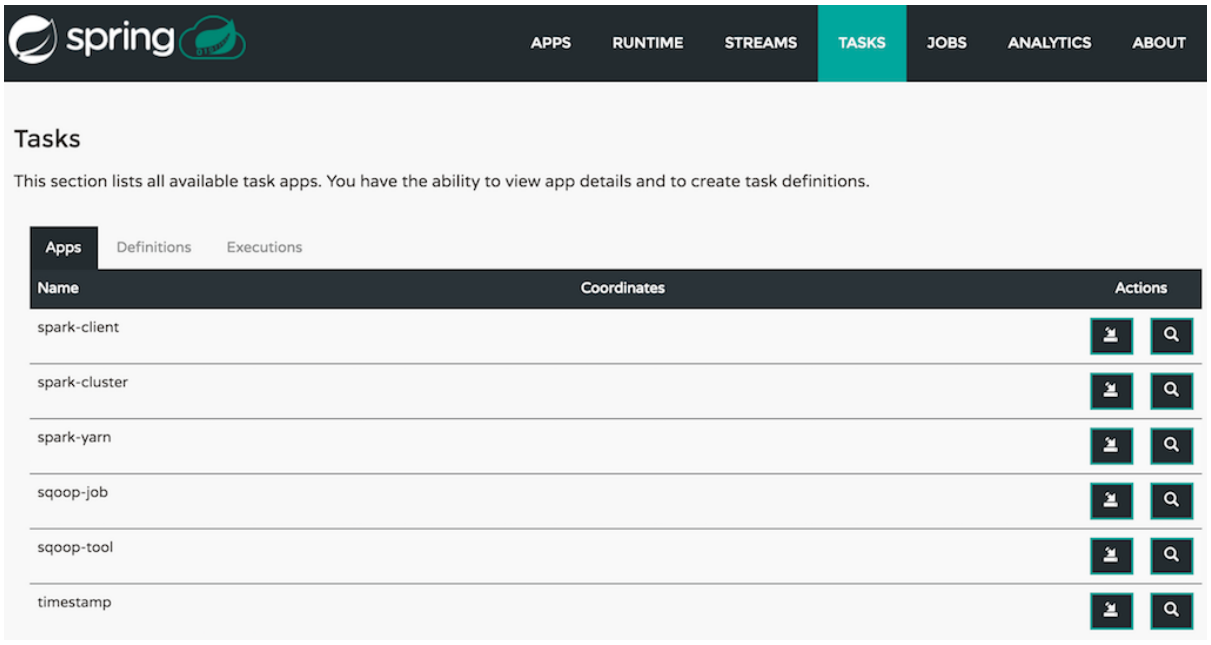
The *Tasks* section of the Dashboard currently has three tabs:

- Apps
- Definitions
- Executions

31.1 Apps

Apps encapsulate a unit of work into a reusable component. Within the Data Flow runtime environment *Apps* allow users to create definitions for *Streams* as well as *Tasks*. Consequently, the *Apps* tab within the *Tasks* section allows users to create *Task* definitions.

Note: You will also use this tab to create Batch Jobs.



Tasks

This section lists all available task apps. You have the ability to view app details and to create task definitions.

Name	Coordinates	Actions
spark-client		
spark-cluster		
spark-yarn		
sqoop-job		
sqoop-tool		
timestamp		

Figure 31.1. List of Task Apps

On this screen you can perform the following actions:

- View details such as the task app options.
- Create a Task Definition from the respective App.

Create a Task Definition from a selected Task App

On this screen you can create a new Task Definition. As a minimum you must provide a name for the new definition. You will also have the option to specify various properties that are used during the deployment of the app.

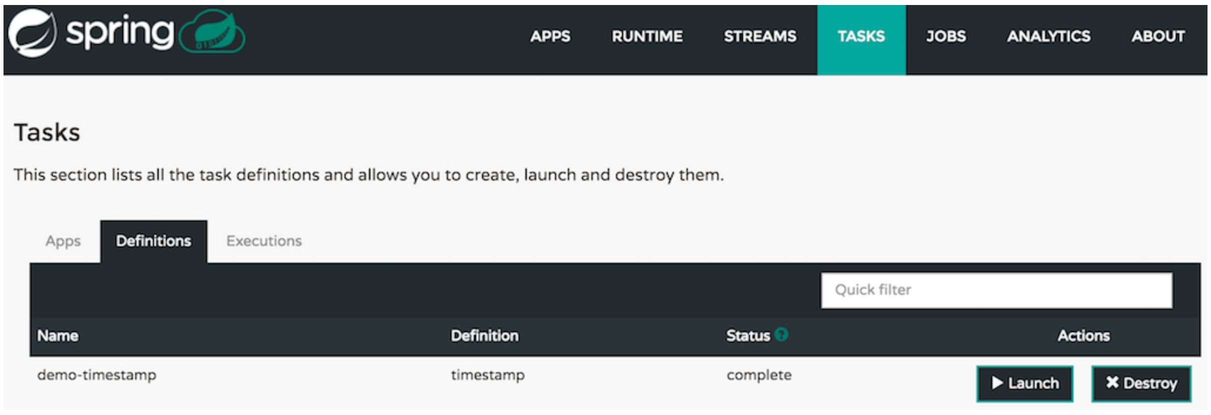
Note: Each parameter is only included if the *Include* checkbox is selected.

View Task App Details

On this page you can view the details of a selected task app, including the list of available options (properties) for that app.

31.2 Definitions

This page lists the Data Flow Task definitions and provides actions to **launch** or **destroy** those tasks.



Tasks

This section lists all the task definitions and allows you to create, launch and destroy them.

Name	Definition	Status	Actions
demo-timestamp	timestamp	complete	▶ Launch ✖ Destroy

Figure 31.2. List of Task Definitions

Launching Tasks

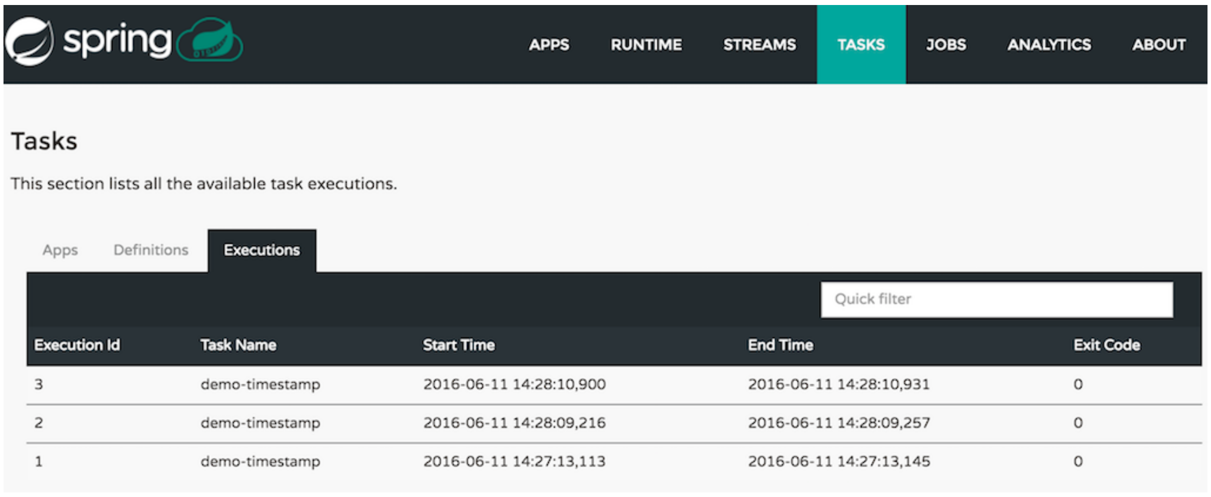
Once the task definition is created, they can be launched through the Dashboard as well. Navigate to the **Definitions** tab. Select the Task you want to launch by pressing `Launch`.

On the following screen, you can define one or more Task parameters by entering:

- Parameter Key
- Parameter Value

Task parameters are not typed.

31.3 Executions



Tasks

This section lists all the available task executions.

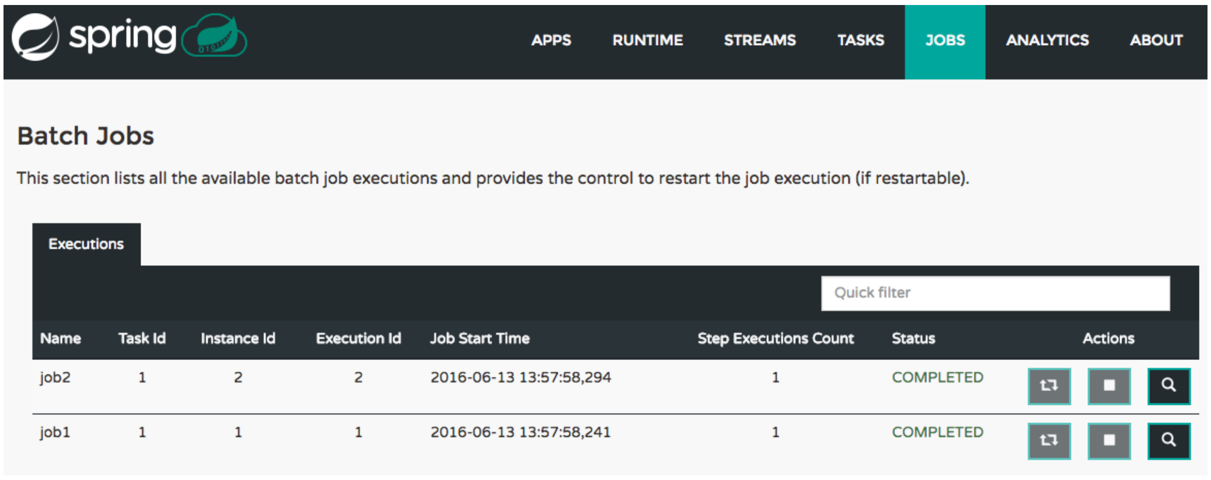
Execution Id	Task Name	Start Time	End Time	Exit Code
3	demo-timestamp	2016-06-11 14:28:10,900	2016-06-11 14:28:10,931	0
2	demo-timestamp	2016-06-11 14:28:09,216	2016-06-11 14:28:09,257	0
1	demo-timestamp	2016-06-11 14:27:13,113	2016-06-11 14:27:13,145	0

Figure 31.3. List of Task Executions

32. Jobs

The *Jobs* section of the Dashboard allows you to inspect **Batch Jobs**. The main section of the screen provides a list of Job Executions. **Batch Jobs** are **Tasks** that were executing one or more **Batch Job**. As such each Job Execution has a back reference to the **Task Execution Id** (Task Id).

In case of a failed job, you can also restart the task. When dealing with long-running Batch Jobs, you can also request to stop it.



Batch Jobs

This section lists all the available batch job executions and provides the control to restart the job execution (if restartable).

Name	Task Id	Instance Id	Execution Id	Job Start Time	Step Executions Count	Status	Actions
job2	1	2	2	2016-06-13 13:57:58,294	1	COMPLETED	
job1	1	1	1	2016-06-13 13:57:58,241	1	COMPLETED	

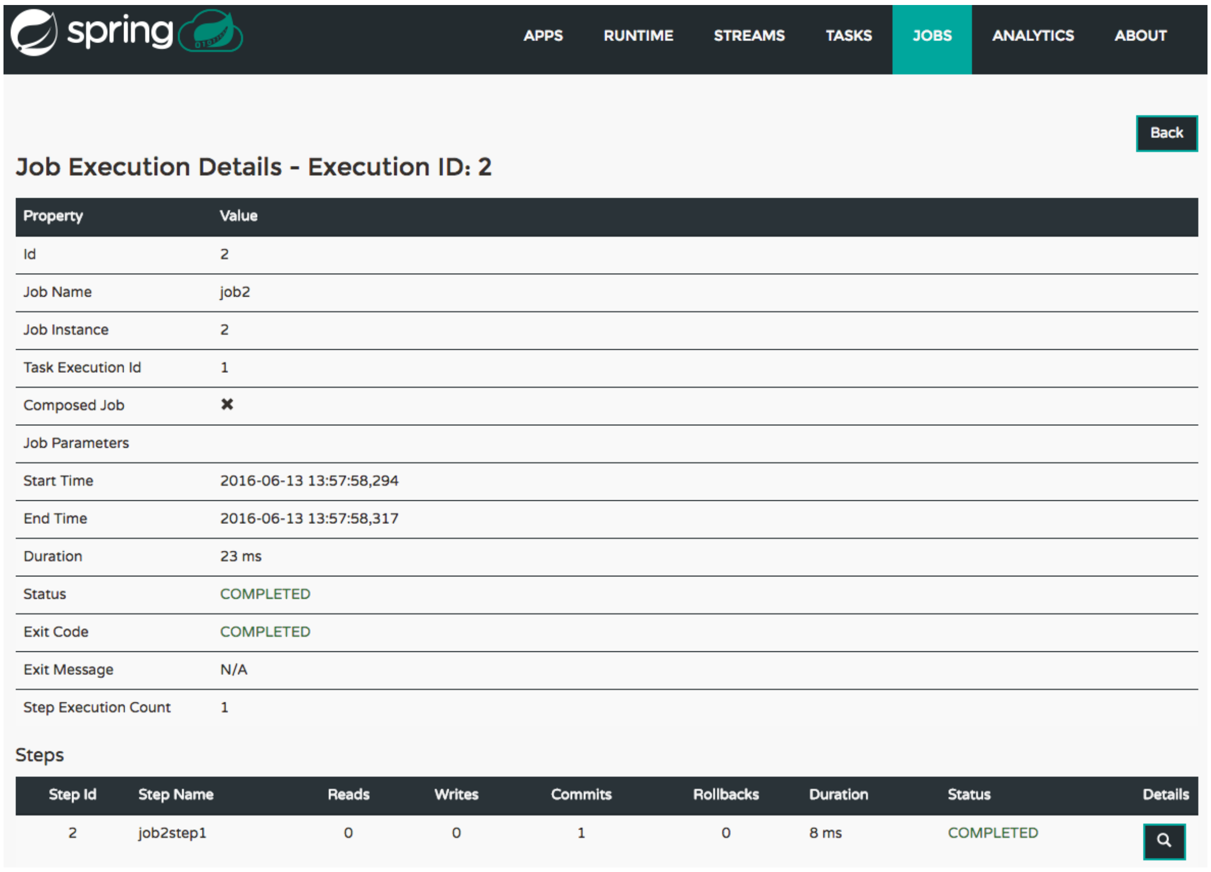
Figure 32.1. List of Job Executions

32.1 List job executions

This page lists the Batch Job Executions and provides the option to **restart** or **stop** a specific job execution, provided the operation is available. Furthermore, you have the option to view the Job execution details.

The list of Job Executions also shows the state of the underlying Job Definition. Thus, if the underlying definition has been deleted, *deleted* will be shown.

Job execution details



Job Execution Details - Execution ID: 2

Property	Value
Id	2
Job Name	job2
Job Instance	2
Task Execution Id	1
Composed Job	✖
Job Parameters	
Start Time	2016-06-13 13:57:58,294
End Time	2016-06-13 13:57:58,317
Duration	23 ms
Status	COMPLETED
Exit Code	COMPLETED
Exit Message	N/A
Step Execution Count	1

Steps

Step Id	Step Name	Reads	Writes	Commits	Rollbacks	Duration	Status	Details
2	job2step1	0	0	1	0	8 ms	COMPLETED	🔍

Figure 32.2. Job Execution Details

The Job Execution Details screen also contains a list of the executed steps. You can further drill into the *Step Execution Details* by clicking onto the magnifying glass.

Step execution details

On the top of the page, you will see progress indicator the respective step, with the option to refresh the indicator. Furthermore, a link is provided to view the *step execution history*.

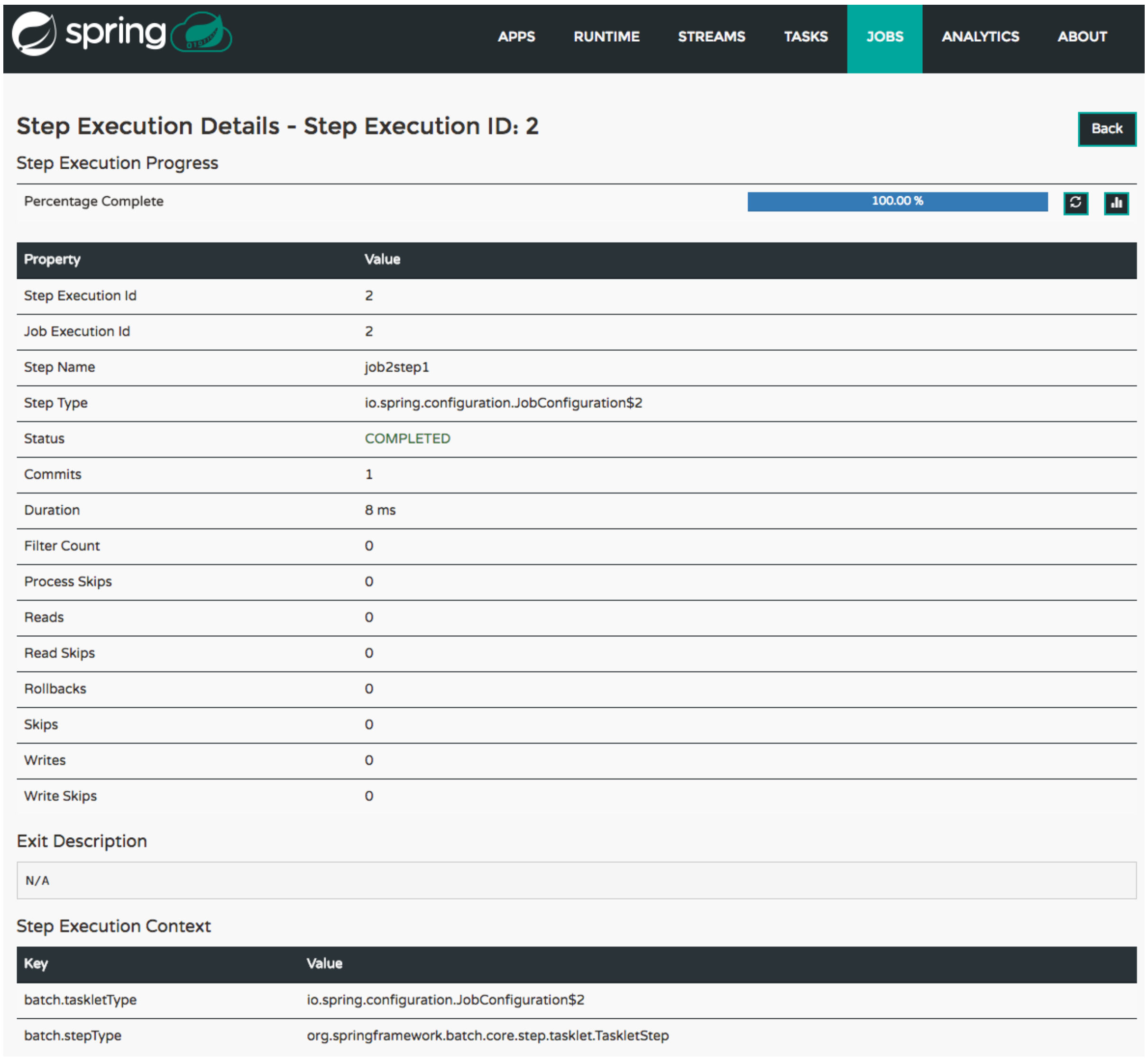
The Step Execution details screen provides a complete list of all Step Execution Context key/value pairs.

Important

In case of exceptions, the *Exit Description* field will contain additional error information. Please be aware, though, that this field can only have a maximum of **2500 characters**. Therefore, in case of long exception stacktraces, trimming of error messages may occur. In that case, please refer to the server log files for further details.

Step Execution Progress

On this screen, you can see a progress bar indicator in regards to the execution of the current step. Under the **Step Execution History**, you can also view various metrics associated with the selected step such as **duration**, **read counts**, **write counts** etc.



Step Execution Details - Step Execution ID: 2

Step Execution Progress

Percentage Complete

Property	Value
Step Execution Id	2
Job Execution Id	2
Step Name	job2step1
Step Type	io.spring.configuration.JobConfiguration$2
Status	COMPLETED
Commits	1
Duration	8 ms
Filter Count	0
Process Skips	0
Reads	0
Read Skips	0
Rollbacks	0
Skips	0
Writes	0
Write Skips	0

Exit Description

N/A

Step Execution Context

Key	Value
batch.taskletType	io.spring.configuration.JobConfiguration$2
batch.stepType	org.springframework.batch.core.step.tasklet.TaskletStep

Figure 32.3. Step Execution History

33. 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. Appendices

Appendix A. Building

To build the source you will need to install JDK 1.7.

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.

A.1 Documentation

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

```
$ ./mvnw clean package -DskipTests -P full -pl spring-cloud-dataflow-docs -am
```

A.2 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.

Appendix B. Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

B.1 Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the contributor's agreement. Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

B.2 Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the Spring Cloud Build project. If using IntelliJ, you can use the Eclipse Code Formatter Plugin to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the .java files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow these conventions, if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).