



# **Spring Cloud Data Flow Server for Mesos**

1.0.0.RC1

---

Copyright © 2013-2016 Pivotal Software, Inc.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

---

# Table of Contents

I. Introduction .....	1
1. Introducing Spring Cloud Data Flow for Mesos and Marathon .....	2
2. Spring Cloud Data Flow .....	3
3. Spring Cloud Stream .....	4
4. Spring Cloud Task .....	5
II. Getting Started .....	6
5. Deploying Streams on Mesos and Marathon .....	7
III. Streams .....	10
6. Introduction .....	11
7. Stream DSL .....	12
8. Register a Stream App .....	13
8.1. Whitelisting application properties .....	14
9. Creating a Stream .....	15
10. Destroying a Stream .....	16
11. Deploying and Undeploying Streams .....	17
12. Other Source and Sink Application Types .....	18
13. Simple Stream Processing .....	19
14. Stateful Stream Processing .....	20
15. Tap a Stream .....	21
16. Using Labels in a Stream .....	22
17. Explicit Broker Destinations in a Stream .....	23
18. Directed Graphs in a Stream .....	24
18.1. Common application properties .....	24
IV. Dashboard .....	25
19. Introduction .....	26
20. Apps .....	27
21. Runtime .....	28
22. Streams .....	29
23. Create Stream .....	30
24. Tasks .....	31
24.1. Apps .....	31
Create a Task Definition from a selected Task App .....	31
View Task App Details .....	32
24.2. Definitions .....	32
Launching Tasks .....	32
24.3. Executions .....	32
25. Jobs .....	33
25.1. List job executions .....	33
Job execution details .....	34
Step execution details .....	34
Step Execution Progress .....	34
26. Analytics .....	36
V. Appendices .....	37
A. Test Cluster .....	38
A.1. Create Vagrant file with 64-bit Ubuntu .....	38
A.2. Install Mesos, Marathon and Docker .....	38
B. Building .....	40

B.1. Documentation .....	40
B.2. Working with the code .....	40
Importing into eclipse with m2eclipse .....	40
Importing into eclipse without m2eclipse .....	41
C. Contributing .....	42
C.1. Sign the Contributor License Agreement .....	42
C.2. Code Conventions and Housekeeping .....	42

---

# Part I. Introduction

# 1. Introducing Spring Cloud Data Flow for Mesos and Marathon

This project provides support for orchestrating long-running (*streaming*) and short-lived (*task/batch*) data microservices to Marathon on Mesos.

## 2. Spring Cloud Data Flow

Spring Cloud Data Flow is a cloud-native orchestration service 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.

The Spring Cloud Data Flow architecture consists of a server that deploys [Streams](#) and [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 while Tasks are based on the [Spring Cloud Task](#) programming model. The sections below describe more information about creating your own custom Streams and Tasks

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.

### 3. 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.



## 4. Spring Cloud Task

Spring Cloud Task makes it easy to create short-lived microservices. We provide capabilities that allow short-lived JVM processes to be executed on demand in a production environment.

For more details about the core framework components and the supported features, please review Spring Cloud Task's [reference guide](#).

There's a rich ecosystem of Spring Cloud Task [Application-Starters](#) that can be used either as standalone data microservice applications or in Spring Cloud Data Flow. For convenience, the generated application-starters are available for use from [Maven Repo](#). There are several [samples](#) available for reference.

---

## Part II. Getting Started

---

## 5. Deploying Streams on Mesos and Marathon

In this getting started guide, the Data Flow Server is run as a standalone application outside the Mesos cluster. A future version will provide support for the Data Flow Server itself to run on Mesos.

### 1. Deploy a Mesos and Marathon cluster.

The [Mesosphere getting started guide](#) provides a number of options for you to deploy a cluster. Many of the options listed there need some additional work to get going. For example, many Vagrant provisioned VMs are using deprecated versions of the Docker client. We have included some brief instructions for setting up a single-node cluster with Vagrant in [Appendix A, Test Cluster](#). In addition to this we have also used the [Playa Mesos](#) Vagrant setup. For those that want to setup a distributed cluster quickly, there is also an option to spin up a cluster on AWS using [Mesosphere's Datacenter Operation System on Amazon Web Services](#).

The rest of this getting started guide assumes that you have a working Mesos and Marathon cluster and know the Marathon endpoint URL.

### 2. Create a Rabbit MQ service on the Mesos cluster.

The `rabbitmq` service will be used for messaging between applications in the stream. There is a sample [application JSON file for Rabbit MQ](#) in the `spring-cloud-dataflow-server-mesos` repository that you can use as a starting point. The service discovery mechanism is currently disabled so you need to look up the host and port to use for the connection. Depending on how large your cluster is, you may want to tweak the CPU and/or memory values.

Using the above JSON file and an Mesos and Marathon cluster installed you can deploy a Rabbit MQ application instance by issuing the following command

```
curl -X POST http://192.168.33.10:8080/v2/apps -d @rabbitmq.json -H "Content-type: application/json"
```

Note the `@` symbol to reference a file and that we are using the Marathon endpoint URL of `192.168.33.10:8080`. Your endpoint might be different based on the configuration used for your installation of Mesos and Marathon. Using the Marathon and Mesos UIs you can verify that `rabbitmq` service is running on the cluster.

### 3. Download the Spring Cloud Data Flow Server for Mesos and Marathon.

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

### 4. Using the Marathon GUI, look up the host and port for the `rabbitmq` application. In our case it was `192.168.33.10:31916`. For the deployed apps to be able to connect to Rabbit MQ we need to provide the following property when we start the server:

```
--
spring.cloud.deployer.mesos.marathon.environmentVariables='SPRING_RABBITMQ_HOST=192.168.33.10,SPRING_RABBITMQ_PORT=31916'
```

### 5. Now, run the Spring Cloud Data Flow Server for Mesos and Marathon passing in this host/port configuration.

```
$ java -jar spring-cloud-dataflow-server-mesos-1.0.0.RC1.jar --
spring.cloud.deployer.mesos.marathon.apiEndpoint=http://192.168.33.10:8080 --
spring.cloud.deployer.mesos.marathon.memory=768 --
spring.cloud.deployer.mesos.marathon.environmentVariables='SPRING_RABBITMQ_HOST=192.168.33.10,SPRING_RABBITMQ_PORT=31916'
```

You can pass in properties to set default values for memory and cpu resource request. For example `--spring.cloud.deployer.mesos.marathon.memory=768` will by default allocate additional memory for the application vs. the default value of 512. You can see all the available options in the [MarathonAppDeployerProperties.java](#) file.

## 6. Download and run the Spring Cloud Data Flow shell.

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

## 7. 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-docker
```

## 8. Deploy a simple stream in the shell



### Note

If you need to specify any of the app specific configuration properties then you must use "long-form" of them including the app specific prefix like `--jdbc.tableName=TEST_DATA`. This is due to the server not being able to access the metadata for the Docker based starter apps. You will also not see the configuration properties listed when using the `app info` command or in the Dashboard GUI.

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

In the Mesos UI you can then look at the logs for the log sink.

```
2016-04-26 18:13:03.001 INFO 1 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer :
Tomcat started on port(s): 8080 (http)
2016-04-26 18:13:03.004 INFO 1 --- [main] o.s.c.s.a.l.s.r.LogSinkRabbitApplication :
Started LogSinkRabbitApplication in 7.766 seconds (JVM running for 8.24)
2016-04-26 18:13:54.443 INFO 1 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] :
Initializing Spring FrameworkServlet 'dispatcherServlet'
2016-04-26 18:13:54.445 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet :
FrameworkServlet 'dispatcherServlet': initialization started
2016-04-26 18:13:54.459 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet :
FrameworkServlet 'dispatcherServlet': initialization completed in 14 ms
2016-04-26 18:14:09.088 INFO 1 --- [time.ticktock-1] log.sink :
04/26/16 18:14:09
2016-04-26 18:14:10.077 INFO 1 --- [time.ticktock-1] log.sink :
04/26/16 18:14:10
2016-04-26 18:14:11.080 INFO 1 --- [time.ticktock-1] log.sink :
04/26/16 18:14:11
2016-04-26 18:14:12.083 INFO 1 --- [time.ticktock-1] log.sink :
04/26/16 18:14:12
2016-04-26 18:14:13.090 INFO 1 --- [time.ticktock-1] log.sink :
04/26/16 18:14:13
2016-04-26 18:14:14.091 INFO 1 --- [time.ticktock-1] log.sink :
04/26/16 18:14:14
2016-04-26 18:14:15.093 INFO 1 --- [time.ticktock-1] log.sink :
04/26/16 18:14:15
2016-04-26 18:14:16.095 INFO 1 --- [time.ticktock-1] log.sink :
04/26/16 18:14:16
```

## 9. Destroy the stream

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

---

# Part III. Streams

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

## 6. 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.

## 7. 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.



## 8. 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](http://bit.ly/stream-applications-rabbit-maven)
- Maven based Stream Applications with Kafka Binder: [bit.ly/stream-applications-kafka-maven](http://bit.ly/stream-applications-kafka-maven)
- Maven based Task Applications: [bit.ly/task-applications-maven](http://bit.ly/task-applications-maven)
- Docker based Stream Applications with RabbitMQ Binder: [bit.ly/stream-applications-rabbit-docker](http://bit.ly/stream-applications-rabbit-docker)
- Docker based Stream Applications with Kafka Binder: [bit.ly/stream-applications-kafka-docker](http://bit.ly/stream-applications-kafka-docker)
- Docker based Task Applications: [bit.ly/task-applications-docker](http://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.

## 8.1 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
```

## 9. 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 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_tbdlvj28c8vj37n900m0000gn/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_tbdlvj28c8vj37n900m0000gn/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_tbdlvj28c8vj37n900m0000gn/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 16, Using Labels in a Stream](#).

## 10. 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.

## 11. 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
```

## 12. 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_tbdlvj28c8vj37n900m0000gn/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_tbdlvj28c8vj37n900m0000gn/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.

## 13. 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
```

## 14. 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 :
```

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.



## 15. 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.

## 16. 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
```

## 17. 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.

## 18. 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](#).

### 18.1 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. Dashboard

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

## 19. 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	<a href="http://cloud.spring.io/spring-cloud-dataflow/">http://cloud.spring.io/spring-cloud-dataflow/</a>
Sources	<a href="https://github.com/spring-cloud/spring-cloud-dataflow">https://github.com/spring-cloud/spring-cloud-dataflow</a>
Documentation	<a href="http://docs.spring.io/spring-cloud-dataflow/docs/current/reference/html/">http://docs.spring.io/spring-cloud-dataflow/docs/current/reference/html/</a>
API Docs	<a href="http://docs.spring.io/spring-cloud-dataflow/docs/current/api/">http://docs.spring.io/spring-cloud-dataflow/docs/current/api/</a>
Support Forum	<a href="http://stackoverflow.com/questions/tagged/spring-cloud">http://stackoverflow.com/questions/tagged/spring-cloud</a>
Issue Tracker	<a href="https://github.com/spring-cloud/spring-cloud-dataflow/issues">https://github.com/spring-cloud/spring-cloud-dataflow/issues</a>

Figure 19.1. The Spring Cloud Data Flow Dashboard

## 20. 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).

All Applications

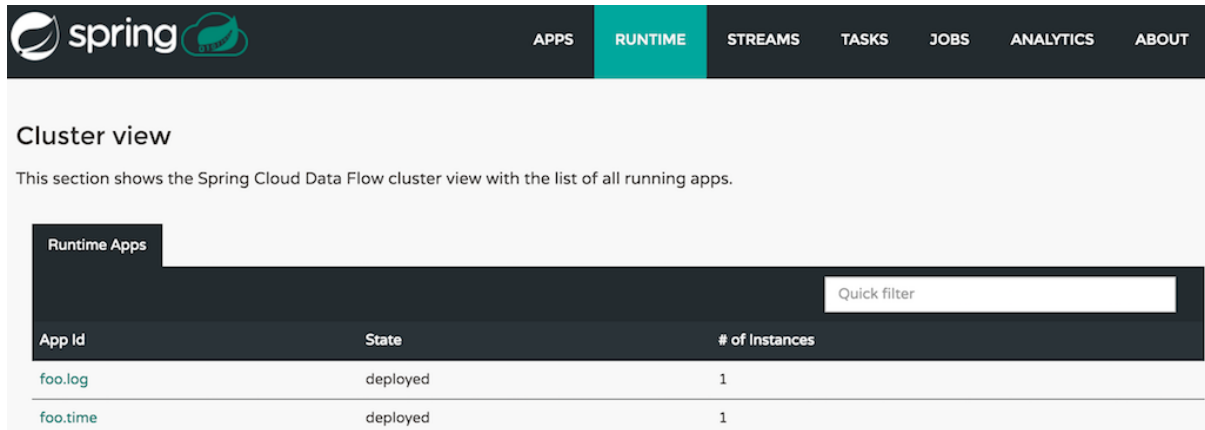
+ Register Application(s) Unregister Application(s) Quick filter

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

Figure 20.1. List of Available Applications

## 21. 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.



The screenshot shows the 'Runtime' tab selected in the dashboard. Below the navigation bar, the 'Cluster view' section contains a description and a table of runtime apps. The table has columns for 'App Id', 'State', and '# of Instances'. Two apps are listed: 'foo.log' and 'foo.time', both in a 'deployed' state with 1 instance each. A 'Quick filter' input field is located to the right of the table header.

App Id	State	# of Instances
<a href="#">foo.log</a>	deployed	1
<a href="#">foo.time</a>	deployed	1

Figure 21.1. List of Running Applications



## 22. 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**.

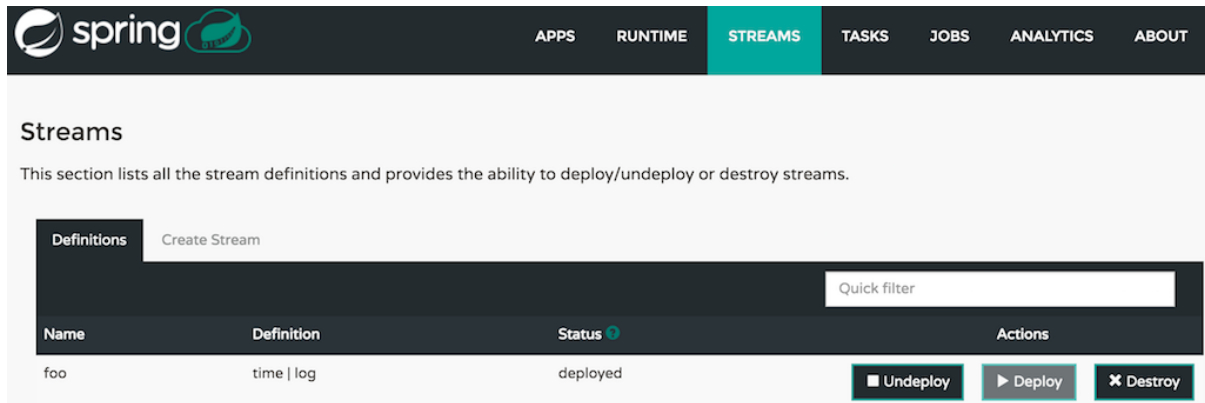


Figure 22.1. List of Stream Definitions

## 23. Create Stream

The *Create Stream* section of the Dashboard includes the [Spring Flo](#) designer tab that provides the canvas application, offering an 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.

The screenshot displays the Spring Flo dashboard. At the top, the navigation bar includes 'APPS', 'RUNTIME', 'STREAMS' (active), 'TASKS', 'JOBS', 'ANALYTICS', and 'ABOUT'. Below the navigation bar, the 'Streams' section is active, with a sub-tab 'Create Stream'. The interface features a DSL editor with the following code:

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

Below the DSL editor, there is a 'source' panel with connectors for 'file', 'ftp', 'http', 'jdbc', 'jms', and 'load-gener...'. The main canvas, titled 'STREAM\_1', shows a visual representation of the pipeline. It starts with a 'time' source connector, which branches into three parallel paths. Each path consists of a 'λ scriptable-t...' transform node followed by a 'log' sink connector.

Figure 23.1. Flo for Spring Cloud Data Flow

## 24. Tasks

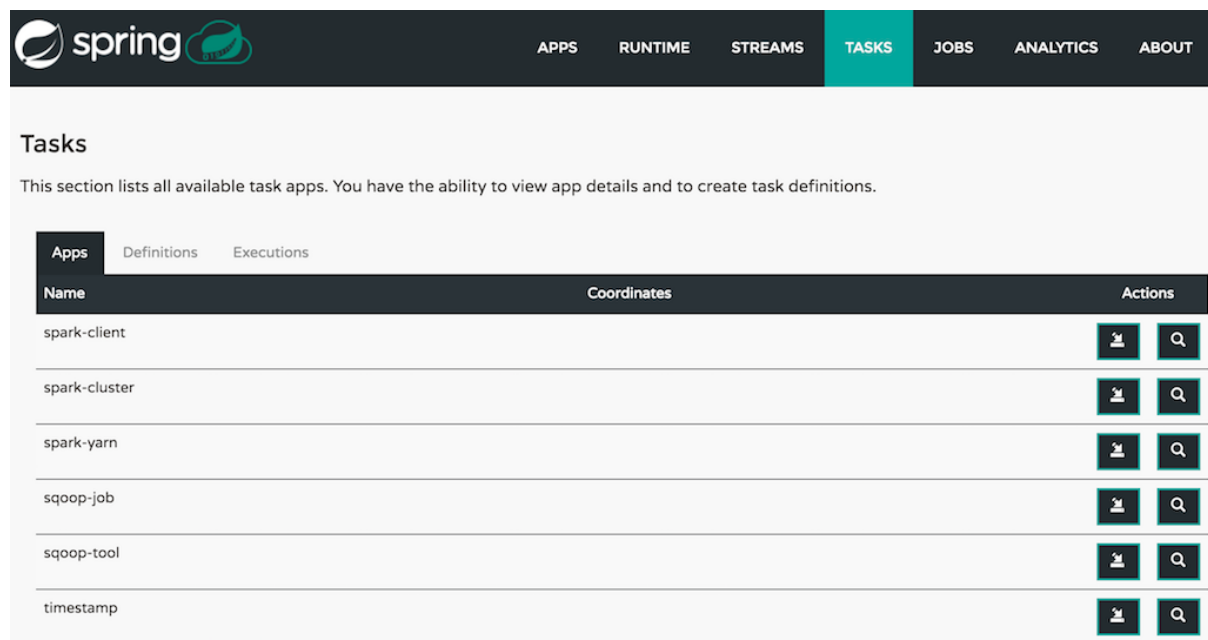
The *Tasks* section of the Dashboard currently has three tabs:

- Apps
- Definitions
- Executions

### 24.1 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.



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

Figure 24.1. 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.

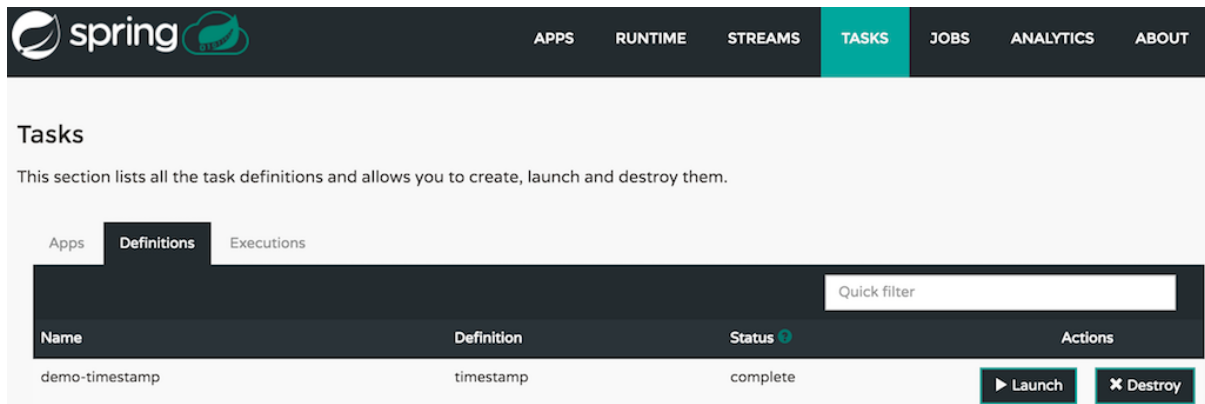
**Note:** Each parameter is only included if the *Include* checkbox is selected.

## 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.

## 24.2 Definitions

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



The screenshot shows the 'Tasks' page with the 'Definitions' tab selected. The table lists task definitions with columns: Name, Definition, Status, and Actions. A single task 'demo-timestamp' is shown with status 'complete'. The 'Actions' column contains 'Launch' and 'Destroy' buttons.

Name	Definition	Status	Actions
demo-timestamp	timestamp	complete	<a href="#">▶ Launch</a> <a href="#">✕ Destroy</a>

Figure 24.2. List of Task Definitions

## Launching Tasks

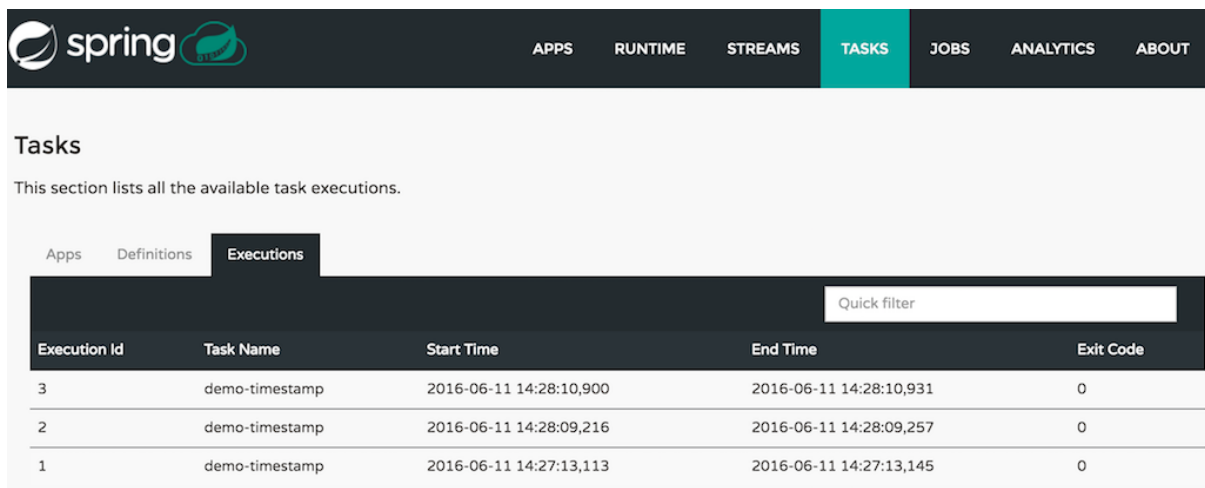
Once the task definition is created, they can be launched through the Dashboard as well. Navigate to the **Definitions** tab. Select the Task you want to launch by pressing Launch.

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

- Parameter Key
- Parameter Value

Task parameters are not typed.

## 24.3 Executions



The screenshot shows the 'Tasks' page with the 'Executions' tab selected. The table lists task executions with columns: Execution Id, Task Name, Start Time, End Time, and Exit Code. Three executions are shown for the 'demo-timestamp' task.

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 24.3. List of Task Executions

## 25. 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.

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	[Restart] [Stop] [Details]
job1	1	1	1	2016-06-13 13:57:58,241	1	COMPLETED	[Restart] [Stop] [Details]

Figure 25.1. List of Job Executions

### 25.1 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** [Back](#)

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	<a href="#">Q</a>

Figure 25.2. 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.

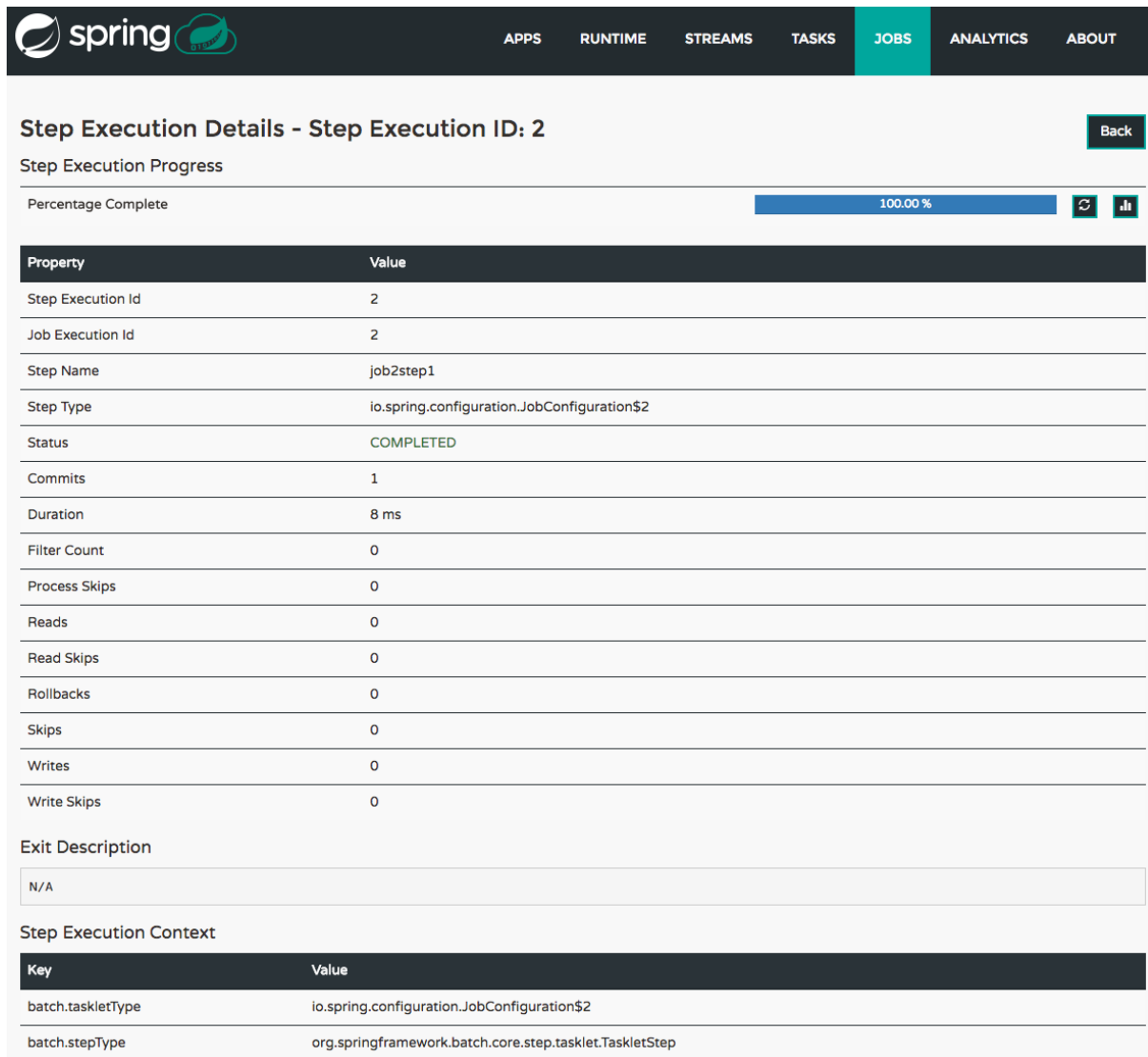


Figure 25.3. Step Execution History

## 26. 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-arrange the order of created dashboards or remove data visualizations.



---

## **Part V. Appendices**

# Appendix A. Test Cluster

Here are brief setup instructions for setting up a local Vagrant single-node cluster. The Mesos endpoint will be [192.168.33.10:5050](http://192.168.33.10:5050) and the Marathon endpoint will be [192.168.33.10:8080](http://192.168.33.10:8080).

## A.1 Create Vagrant file with 64-bit Ubuntu

First create the Vagrant file with necessary customizations:

```
$ vi Vagrantfile
```

Add the following content and save the file:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/trusty64"

  config.vm.network "private_network", ip: "192.168.33.10"
  config.vm.hostname = "mesos"

  config.vm.provider "virtualbox" do |vb|
    vb.memory = "4096"
    vb.cpus = 4
  end
end
```

Next, update the box to the latest version and start it:

```
$ vagrant box update
$ vagrant up
```

## A.2 Install Mesos, Marathon and Docker

We can now ssh to the instance to install the necessary bits:

```
$ vagrant ssh
```

The rest of these instructions are run from within this ssh shell.

1. Refresh the apt repo and install Docker:

```
vagrant@mesos:~$ sudo apt-get -y update
vagrant@mesos:~$ wget -qO- https://get.docker.com/ | sh
vagrant@mesos:~$ sudo usermod -aG docker vagrant
```

2. Install needed repos:

```
vagrant@mesos:~$ echo "deb http://repos.mesosphere.io/${lsb_release -is | tr '[:upper:]' '[:lower:]'}
${lsb_release -cs} main" | sudo tee /etc/apt/sources.list.d/mesosphere.list
vagrant@mesos:~$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF
vagrant@mesos:~$ sudo add-apt-repository ppa:webupd8team/java -y
vagrant@mesos:~$ sudo apt-get -y update
```

3. Install Java:

```
vagrant@mesos:~$ sudo apt-get install oracle-java8-installer
```

#### 4. Install Mesos and Marathon:

```
vagrant@mesos:~$ sudo apt-get -y install mesos marathon
```

#### 5. Add Docker as a containerizer:

```
vagrant@mesos:~$ echo 'docker,mesos' | sudo tee /etc/mesos-slave/containerizers
```

#### 6. Set the IP address as the hostname used for the slave:

```
vagrant@mesos:~$ echo $(/sbin/ifconfig eth1 | grep 'inet addr:' | cut -d: -f2 | awk '{ print $1}') |  
sudo tee /etc/mesos-slave/hostname
```

#### 7. Reboot the server

```
vagrant@mesos:~$ sudo reboot
```

# Appendix B. 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 building. See below for more information on how to 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 middleware 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.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 {project-artifactId} -am
```

## B.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 [m2eclipse](#) eclipse plugin for maven support. Other IDEs and tools should also work without issue.

### Importing into eclipse with m2eclipse

We recommend the [m2eclipse](#) 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 C. 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.

## C.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.

## C.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).