Spring Cloud Data Flow Reference Guide

1.0.0.M3

Copyright © 2013-2016Pivotal 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. Preface 1
- 1. About the documentation 2
- 2. Getting help 3

II. Spring Cloud Data Flow Overview 4
- 3. Introducing Spring Cloud Data Flow 5
  - 3.1. Features 5
- 4. Spring Cloud Data Flow Architecture 6
  - 4.1. Components 6
- 5. System Requirements 7
- 6. Deploying Spring Cloud Data Flow 8
  - 6.1. Deploying 'local' 8

III. Streams 10
- 7. Introduction 11
- 8. Creating a Simple Stream 12
- 9. Deleting a Stream 13
- 10. Deploying and Undeploying Streams 14
- 11. Other Source and Sink Types 15
- 12. Simple Stream Processing 16
- 13. DSL Syntax 17
  - 13.1. Register a Stream App 17
- 14. Advanced Features 18
- 15. Module Labels 19
- 16. Tap DSL 20
- 17. Connecting to explicit destination names at the broker 21

IV. Tasks 22
- 18. Introducing Spring Cloud Task 23
- 19. The Lifecycle of a task 24
  - 19.1. Register a Task App 24
  - 19.2. Create a Task Definition 25
  - 19.3. Launch an ad-hoc Task 25
  - 19.4. Task Execution 25
  - 19.5. Destroy a Task Definition 25
- 20. Task Repository 27
  - 20.1. Configuring the Task Execution Repository 27
    - Local 27
  - 20.2. Datasource 27

V. Dashboard 29
- 21. Introduction 30
- 22. Apps 31
- 23. Runtime 32
- 24. Streams 33
- 25. Tasks 34
  - 25.1. Modules 34
    - Create a Task Definition from a selected Job Module 34
    - View Task Module Details 34
  - 25.2. Definitions 35
    - Launching Tasks 35

25.3. Executions 35
26. Jobs 36
26.1. List job executions 36
Job execution details 37
Step execution details 37
Step Execution Progress 37
27. Analytics 39
VI. Appendices 40
A. Building 41
A.1. Documentation 41
A.2. Working with the code 41
Importing into eclipse with m2eclipse 41
Importing into eclipse without m2eclipse 42
B. Contributing 43
B.1. Sign the Contributor License Agreement 43
B.2. Code Conventions and Housekeeping 43

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/.

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.

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`.
- 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

A cloud native programming and operating model for composable data microservices on a structured platform. With Spring Cloud Data Flow, developers can create, orchestrate and refactor data pipelines through single programming model 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 integration and batch modules from Spring XD are refactored into Spring Boot data microservices applications that are now autonomous deployment units – thus enabling them to take full advantage of platform capabilities "natively", and they can independently evolve in isolation.

Spring Cloud Data Flow defines best practices for distributed stream and batch microservice design patterns.

3.1 Features

- Orchestrate applications across a variety of distributed runtime platforms including: Cloud Foundry, Apache YARN, Apache Mesos, and Kubernetes
- Separate runtime dependencies backed by ‘spring profiles’
- Consume stream and batch data-microservices as maven dependency
- Develop using: DSL, Shell, REST-APIs, Admin-UI, and Flo
- Take advantage of metrics, health checks and remote management of data-microservices
- 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 also need to have Redis installed and running if you plan on running a local system, or to run the included tests.

6. Deploying Spring Cloud Data Flow

6.1 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.M3/spring-cloud-dataflow-server-local-1.0.0.M3.jar

wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-shell/1.0.0.M3/
spring-cloud-dataflow-shell-1.0.0.M3.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.M3.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.M3.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.M3.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:>
```

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.

Part III. Streams

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

7. 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 modules 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 --port=8091 | file --dir=/tmp/httpdata/
```

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

8. Creating a Simple 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 Admin Server resolves `time` and `log` to maven coordinates and uses those to launch the `time` and `log` applications of the stream. In this simple example, the time source simply sends the current time as a message each second, and the log sink outputs it using the logging framework.

```
2016-01-13 10:41:15.398  INFO 65275 --- [nio-9393-exec-1] o.s.c.d.a.s.l.OutOfProcessModuleDeployer :
 deploying module org.springframework.cloud.stream.module:log-sink:jar:exec:1.0.0.BUILD-SNAPSHOT
 instance 0
   Logs will be in /var/folders/hs/h87zy7z17qs6mcnl4hj8_dp00000gp/T/spring-cloud-data-
flow-3652850284472151116/ticktock.log
2016-01-13 10:41:15.433  INFO 65275 --- [nio-9393-exec-1] o.s.c.d.a.s.l.OutOfProcessModuleDeployer :
 deploying module org.springframework.cloud.stream.module:time-source:jar:exec:1.0.0.BUILD-SNAPSHOT
 instance 0
   Logs will be in /var/folders/hs/h87zy7z17qs6mcnl4hj8_dp00000gp/T/spring-cloud-data-
flow-3652850284472151116/ticktock.time
```

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

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

> **Important**
>
> See Chapter 15, *Module Labels*.

9. Deleting a Stream

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

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

10. 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
```

11. Other Source and Sink 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

```
2016-01-13 18:42:19.162  INFO 19463 --- [nio-9393-exec-7] o.s.c.d.a.s.l.OutOfProcessModuleDeployer :
 deploying module org.springframework.cloud.stream.module:log-sink:jar:exec:1.0.0.BUILD-SNAPSHOT
 instance 0
   Logs will be in /var/folders/hs/h87zy7z17qs6mcnl4hj8_dp00000gp/T/spring-cloud-data-
flow-2185888994718649403/myhttpstream.log
2016-01-13 18:42:19.180  INFO 19463 --- [nio-9393-exec-7] o.s.c.d.a.s.l.OutOfProcessModuleDeployer :
 deploying module org.springframework.cloud.stream.module:http-source:jar:exec:1.0.0.BUILD-SNAPSHOT
 instance 0
   Logs will be in /var/folders/hs/h87zy7z17qs6mcnl4hj8_dp00000gp/T/spring-cloud-data-
flow-2185888994718649403/myhttpstream.http
```

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

```
dataflow:> runtime modules
```

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

```
2016-01-13 21:15:34.825  INFO 54348 --- [hannel-adapter1] log.sink                                 :
 hello
2016-01-13 21:17:36.544  INFO 54348 --- [hannel-adapter1] 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 modules which are provided. You can also define your own modules.

12. 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 shell command)

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

Will result in an uppercased 'hello' in the log

```
15:18:21,345  WARN ThreadPoolTaskScheduler-1 logger.myprocstream:141 - HELLO
```

See the Processors section for more information.

13. DSL Syntax

In the examples above, we connected a source to a sink using the pipe symbol |. You can also pass parameters to the source and sink configurations. The parameter names will depend on the individual module implementations, but as an example, the `http` source module 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 module parameters and also the shell command `module info` provides some additional documentation. For more comprehensive documentation on module parameters, please see the Modules chapter.

13.1 Register a Stream App

Register a Stream App with the App Registry using the Spring Cloud Data Flow Shell `module register` command. You must provide a unique name 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:>module register --name mysource --type source --uri maven://com.example:mysource:0.0.1-
SNAPSHOT

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

dataflow:>module 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>
```

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:

```
source.foo=file:///tmp/foo.jar
sink.bar=file:///tmp/bar.jar
```

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

```
module import --uri file:///tmp/stream-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 `module register` or `module 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.

14. Advanced Features

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`. To learn more, refer to then section on Named Destinations. Second, you may need to determine the output channel of a stream based on some information that is only known at runtime. To learn about such content-based routing, refer to the Dynamic Router section.

15. Module Labels

When a stream is comprised of multiple modules 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
```

16. Tap DSL

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 lets the parser parse this as the destination name instead of app name.

17. Connecting to explicit destination names at the broker

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 `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 `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.

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 modules.

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

18. 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.

19. 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

19.1 Register a Task App

Register a Task App with the App Registry using the Spring Cloud Data Flow Shell module 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:>module register --name task1 --type task --uri maven://com.example:mytask:1.0.2

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

dataflow:>module 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 module import command and provide the location of the properties file via --uri:

```
module 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 module register or module 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.

19.2 Create a Task Definition

Create a Task Definition from a Task Module 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.

19.3 Launch an ad-hoc 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'
```

19.4 Task Execution

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 view 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 view` command with the id of the task execution , for example `task view --id 549`.

19.5 Destroy a Task Definition

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.

20. Task Repository

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

20.1 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 Dataflow will need to be rebuilt. Since Spring Cloud Dataflow 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 Dataflow

20.2 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
```

Part V. Dashboard

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

21. 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 Dataflow 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 modules

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

```
http://<adminHost>:<adminPort>/admin-ui
```

For example: http://localhost:9393/admin-ui

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

Note: The default Dashboard server port is `9393`

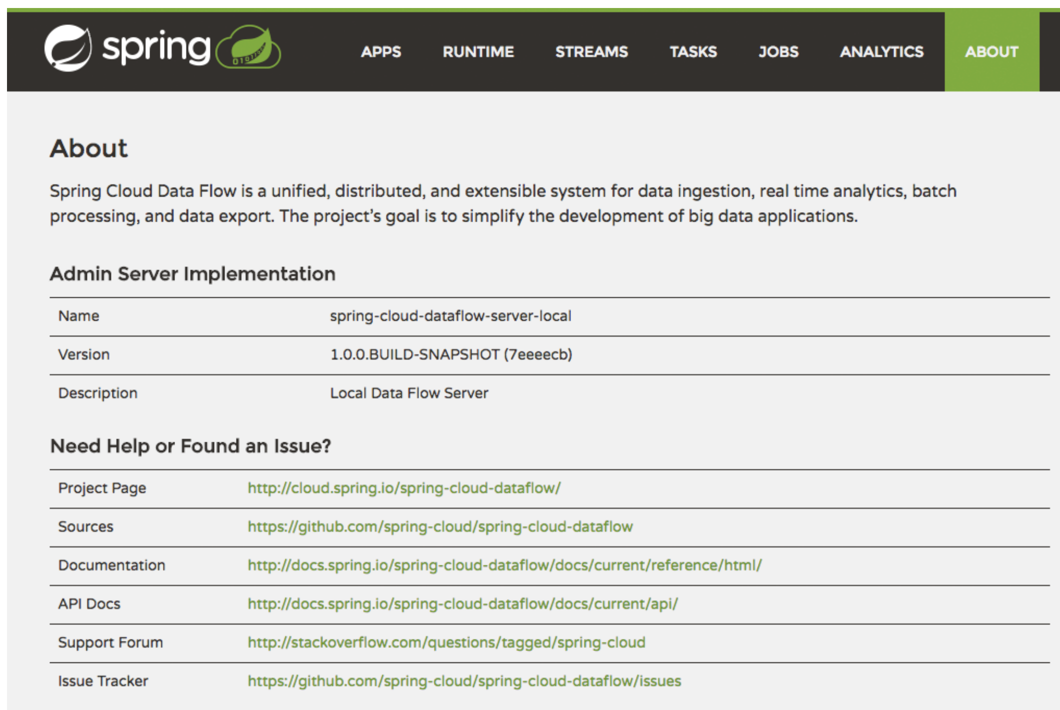

Figure 21.1. The Spring Cloud Data Flow Dashboard

22. Apps

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

Figure 22.1. List of Available Applications

23. 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 module 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 module id.

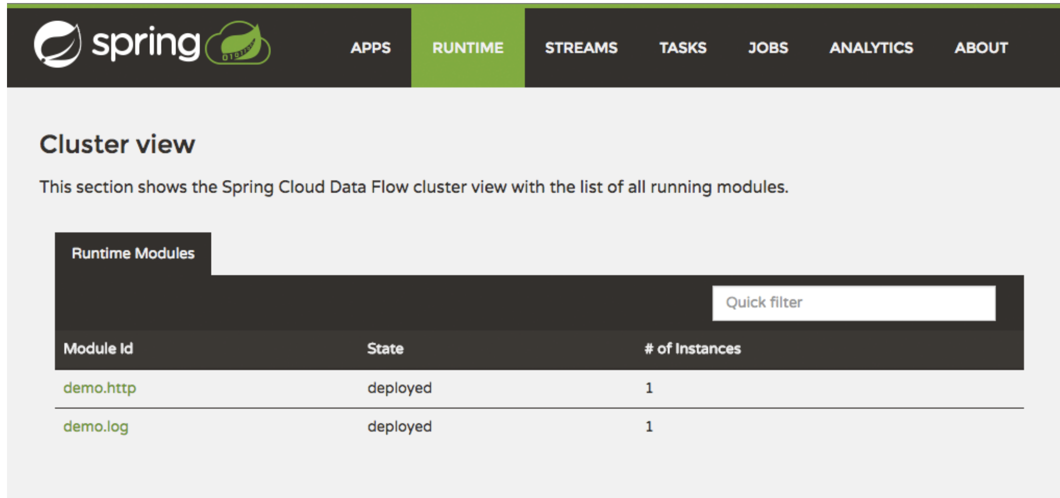

Figure 23.1. List of Running Applications

24. 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**.

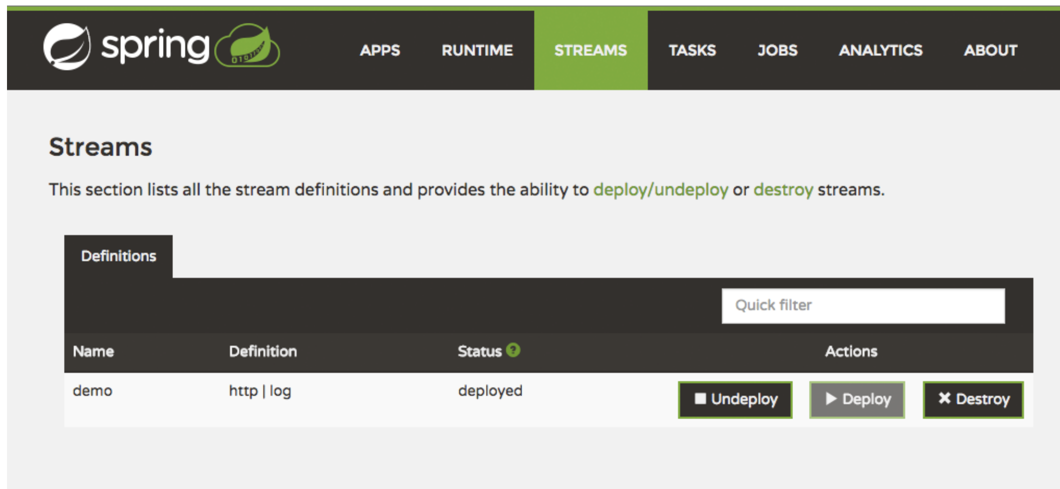

Figure 24.1. List of Stream Definitions

25. Tasks

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

- Modules
- Definitions
- Executions

25.1 Modules

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

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

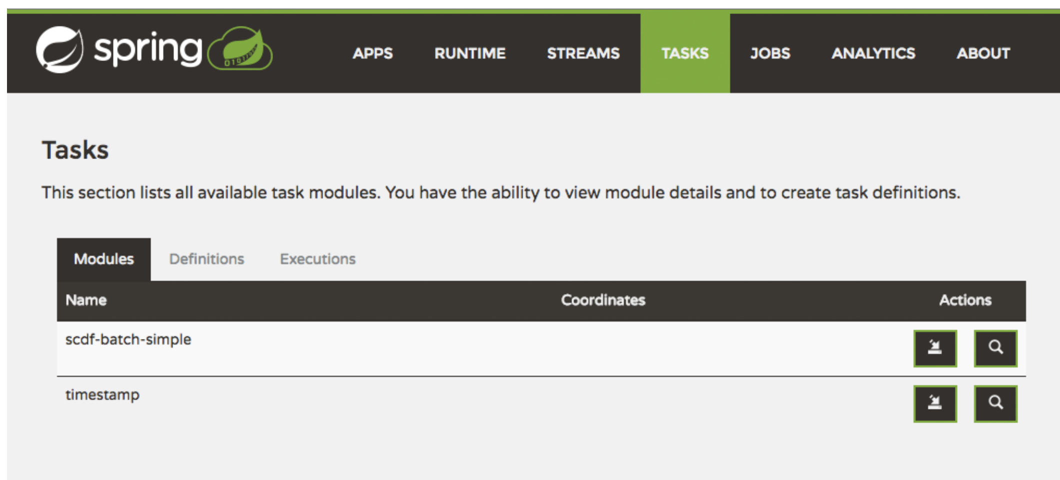

Figure 25.1. List of Task Modules

On this screen you can perform the following actions:

- View details such as the task module options.
- Create a Task Definition from the respective Module.

Create a Task Definition from a selected Job Module

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 parameters that are used during the deployment of the definition.

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

View Task Module Details

On this page you can view the details of a selected task module. The pages lists the available options (properties) of the modules.

25.2 Definitions

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

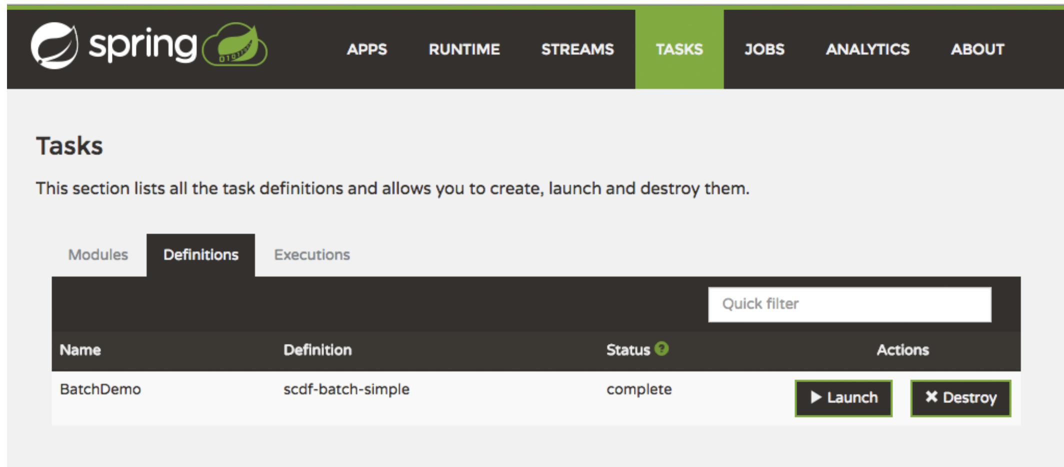

Figure 25.2. List of Task Definitions

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.

25.3 Executions

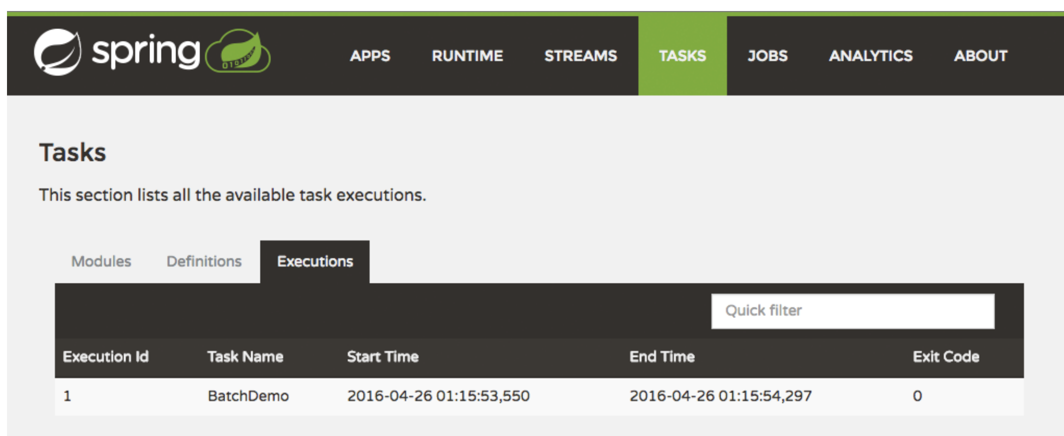

Figure 25.3. List of Task Executions

26. 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.

Figure 26.1. List of Job Executions

26.1 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

spring APPS RUNTIME STREAMS TASKS JOBS ANALYTICS ABOUT

Back

Job Execution Details - Execution ID: 1

Property	Value
Id	1
Job Name	job-1421673801
Job Instance	1
Task Execution Id	1
Composed Job	✖
Job Parameters	random=0.0017610404862123952
Start Time	2016-04-26 01:15:54,164
End Time	2016-04-26 01:15:54,290
Duration	126 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
1	step1	0	0	1	0	45 ms	COMPLETED	

Figure 26.2. 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.

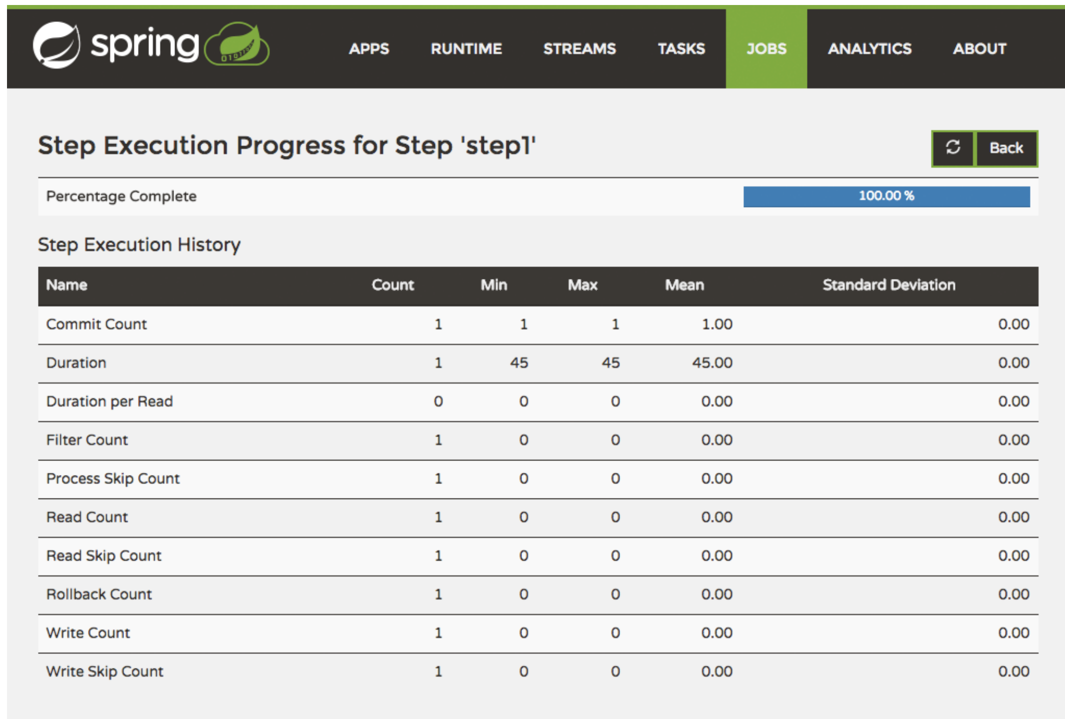

Figure 26.3. Step Execution History

27. Analytics

The *Analytics* section of the Dashboard provided data visualization capabilities for the various analytics modules 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).