

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

1.0.0.M2

Copyright © 2013-2015Pivotal 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
II. Spring Cloud Data Flow Overview .....	3
2. Introducing Spring Cloud Data Flow .....	4
2.1. Features .....	4
3. Spring Cloud Data Flow Architecture .....	5
3.1. Components .....	5
III. Getting Started .....	6
4. Deploying Streams on Mesos and Marathon .....	7
IV. Appendices .....	9
A. Test Cluster .....	10
A.1. Create Vagrant file with 64-bit Ubuntu .....	10
A.2. Install Mesos, Marathon and Docker .....	10
B. Building .....	12
B.1. Documentation .....	12
B.2. Working with the code .....	12
Importing into eclipse with m2eclipse .....	12
Importing into eclipse without m2eclipse .....	13
C. Contributing .....	14
C.1. Sign the Contributor License Agreement .....	14
C.2. Code Conventions and Housekeeping .....	14

---

# Part I. Introduction

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

This project provides support for deploying Spring Cloud Dataflow Stream definitions to Marathon on Mesos.

---

# **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.

## 2. 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.

### 2.1 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

## 3. Spring Cloud Data Flow Architecture

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

### 3.1 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](#)

---

## **Part III. Getting Started**



## 4. 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. This also requires running a local instance of Redis to store available modules. 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 modules 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. Run a local redis-server.

```
$ redis-server
```

This is used by the locally running Data Flow Server to store the state of available module versions for stream definitions.

### 4. 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.M2/spring-cloud-dataflow-server-mesos-1.0.0.M2.jar
```

### 5. 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'
```

6. 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.M2.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.

7. 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.M3/spring-cloud-dataflow-shell-1.0.0.M3.jar

$ java -jar spring-cloud-dataflow-shell-1.0.0.M3.jar
```

8. Register the Rabbit MQ version of the `time` and `log` app modules using the shell

```
dataflow:>module register --type source --name time --uri docker:springcloudstream/time-source-rabbit
dataflow:>module register --type sink --name log --uri docker:springcloudstream/log-sink-rabbit
```

9. Deploy a simple stream in the shell

```
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
```

- 10 Destroy the stream

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

---

## **Part IV. 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).