# Spring Cloud Data Flow Server for Kubernetes

1.0.0.BUILD-SNAPSHOT

# Table of Contents

# Part I. Introduction

# 1. Introducing Spring Cloud Data Flow for Kubernetes

This project provides support for deploying Spring Cloud Dataflow Stream definitions to Kubernetes.

# 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 Kubernetes

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

1. Deploy a Kubernetes cluster.

   The Kubernetes Getting Started guide lets you choose among many deployment options so you can pick one that you are most comfortable using. We have successfully used the Vagrant option from a downloaded Kubernetes release.

   Of note, the docker-compose-kubernetes is not among those options, but it was also used by the developers of this project to run a local Kubernetes cluster using Docker Compose.

   The rest of this getting started guide assumes that you have a working Kubernetes cluster and a `kubectl` command line.

2. Create a Kafka service on the Kubernetes cluster.

   The Kafka service will be used for messaging between modules in the stream. There are sample replication controller and service YAML files in the `spring-cloud-dataflow-server-kubernetes` repository that you can use as a starting point as they have the required metadata set for service discovery by the modules.

   ```
   $ git clone https://github.com/spring-cloud/spring-cloud-dataflow-server-kubernetes
   $ cd spring-cloud-dataflow-server-kubernetes
   $ kubectl create -f src/etc/kubernetes/kafka-controller.yml
   $ kubectl create -f src/etc/kubernetes/kafka-service.yml
   ```

   You can use the command `kubectl get pods` to verify that the controller is running. Note that it can take a minute or so until there is an external IP address for the kafka server. Use the command `kubectl get services` to check on the state of the service and look for when there is a value under the EXTERNAL_IP column. Use the commands `kubectl delete svc kafka` and `kubectl delete rc kafka` to clean up afterwards.

3. Determine the location of your Kubernetes Master URL, for example:

   ```
   $ kubectl cluster-info

   Kubernetes master is running at https://10.245.1.2

      ...other output omitted...
   ```

4. Export environment variables to connect to Kubernetes.

   The Data Flow Server uses the fabric8 Java client library to connect to the Kubernetes cluster. It can be configured using system properties, environment variables, and the Kube config file. In testing using the Google Container Engine, only setting the environment variables `KUBERNETES_MASTER` and `KUBERNETES_NAMESPACE` were required. Other configuration values were read from the Kube config file.

   ```
   $ export KUBERNETES_MASTER=https://10.245.1.2/
   $ export KUBERNETES_NAMESPACE=default
   ```

   This approach supports using one Data Flow Server instance per Kubernetes namespace.

5. Run a local Redis server.

```
$ cd <redis-install-dir>
$ ./src/redis-server
```

This is used by the locally running Data Flow Server to store the state of registered stream app module URIs to be used for stream definitions.

6. Download and run the Spring Cloud Data Flow Server for Kubernetes.

```
$ wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-server-
kubernetes/1.0.0.M2/spring-cloud-dataflow-server-kubernetes-1.0.0.M2.jar

$ java -jar spring-cloud-dataflow-server-kubernetes-1.0.0.M2.jar --
spring.cloud.deployer.kubernetes.memory=768Mi
```

> **Note**
>
> We haven't tuned the memory use of the OOTB apps yet, so to be on the safe side we are increasing the memory for the pods by providing the following property: `--spring.cloud.deployer.kubernetes.memory=768Mi`

> **Note**
>
> If you are running Kubernetes using vagrant locally, then you might need to increase the CPU for the deployed apps using the following property: `--spring.cloud.deployer.kubernetes.cpu=1`

Ensure that the Data Flow Server is running in the same terminal session that has the Kubernetes environment variables set.

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 Kafka version of the `time` and `log` app modules using the shell

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

9. Deploy a simple stream in the shell

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

You can use the command `kubectl get pods` to check on the state of the pods corresponding to this stream. We can run this from the shell by running it as an OS command by adding a "!" before the command.

```
dataflow:>! kubectl get pods
command is:kubectl get pods
NAME                  READY     STATUS    RESTARTS   AGE
kafka-d207a           1/1       Running   0          50m
ticktock-log-qnk72    1/1       Running   0          2m
ticktock-time-r65cn   1/1       Running   0          2m
```

Look at the logs for the pod deployed for the log sink.

```
$ kubectl logs -f ticktock-log-qnk72
...
2015-12-28 18:50:02.897  INFO 1 --- [           main] o.s.c.s.module.log.LogSinkApplication    :
 Started LogSinkApplication in 10.973 seconds (JVM running for 50.055)
2015-12-28 18:50:08.561  INFO 1 --- [hannel-adapter1] log.sink                                 :
 2015-12-28 18:50:08
2015-12-28 18:50:09.556  INFO 1 --- [hannel-adapter1] log.sink                                 :
 2015-12-28 18:50:09
2015-12-28 18:50:10.557  INFO 1 --- [hannel-adapter1] log.sink                                 :
 2015-12-28 18:50:10
2015-12-28 18:50:11.558  INFO 1 --- [hannel-adapter1] log.sink                                 :
 2015-12-28 18:50:11
```

**Note**

If you need to be able to connect from outside of the Kubernetes cluster to an app that you deploy, like the `http-source`, then you can provide a deployment property of `spring.cloud.deployer.kubernetes.createLoadBalancer=true` for the app module to specify that you want to have a LoadBalancer with an external IP address created for your app's service.

To register the `http-source`, deploy it so you can post data to it you can use the following commands:

```
dataflow:>module register --type source --name http --uri docker:springcloudstream/http-source-kafka
dataflow:>stream create --name test --definition "http | log"
dataflow:>stream deploy test --properties
 "module.http.spring.cloud.deployer.kubernetes.createLoadBalancer=true"
```

Now, look up the external IP address for the `http` app (it can sometimes take a minute or two for the external IP to get assigned):

```
dataflow:>! kubectl get service
command is:kubectl get service
NAME         CLUSTER-IP       EXTERNAL-IP      PORT(S)    AGE
kafka        10.103.240.92    <none>           9092/TCP   7m
kubernetes   10.103.240.1     <none>           443/TCP    4h
test-http    10.103.251.157   130.211.200.96   8080/TCP   58s
test-log     10.103.240.28    <none>           8080/TCP   59s
zk           10.103.247.25    <none>           2181/TCP   7m
```

Next, post some data to the `test-http` app:

```
dataflow:>http post --target http://130.211.200.96:8080 --data "Hello"
```

Finally, look at the logs for the `test-log` pod:

```
dataflow:>! kubectl get pods
command is:kubectl get pods
NAME              READY     STATUS             RESTARTS   AGE
kafka-o20qq       1/1       Running            0          9m
test-http-9obkq   1/1       Running            0          2m
test-log-ysiz3    1/1       Running            0          2m
dataflow:>! kubectl logs test-log-ysiz3
command is:kubectl logs test-log-ysiz3
...
2016-04-27 16:54:29.789  INFO 1 --- [           main] o.s.c.s.b.k.KafkaMessageChannelBinder$3  :
 started inbound.test.http.test
2016-04-27 16:54:29.799  INFO 1 --- [           main] o.s.c.support.DefaultLifecycleProcessor  :
 Starting beans in phase 0
2016-04-27 16:54:29.799  INFO 1 --- [           main] o.s.c.support.DefaultLifecycleProcessor  :
 Starting beans in phase 2147482647
2016-04-27 16:54:29.895  INFO 1 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer :
 Tomcat started on port(s): 8080 (http)
2016-04-27 16:54:29.896  INFO 1 --- [  kafka-binder-] log.sink                                 :
 Hello
```

A useful command to help in troubleshooting issues, such as a container that has a fatal error starting up, add the options `--previous` to view last terminated container log. You can also get more detailed information about the pods by using the `kubctl describe` like:

```
kubectl describe pods/ticktock-log-qnk72
```

10. Destroy the stream

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

> **Warning**
>
> If you stop and restart the Data Flow Server when streams are deployed, you will not be able to destroy them via shell commands. You would have to destroy the services and replication containers using the `kubectl` command. This is a bug that is being addressed in a future release.

# Part IV. Server Implementation

# 5. Server Properties

The Spring Data Flow Kubernetes Server has several properties you can configure that let you control the default values to set the `cpu` and `memory` requirements for the pods. The configuration is controlled by configuration properties under the `spring.cloud.deployer.kubernetes` prefix. For example you might declare the following section in an `application.properties` file or pass them as command line arguments when starting the Server.

```
spring.cloud.deployer.kubernetes.memory=512Mi
spring.cloud.deployer.kubernetes.cpu=500m
```

See KubernetesAppDeployerProperties for more of the supported options.

Data Flow Server properties that are common across all of the Data Flow Server implementations that concern maven repository settings can also be set in a similar manner. See the section on Common Data Flow Server Properties for more information.

# Part V. 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 {project-artifactId} -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).