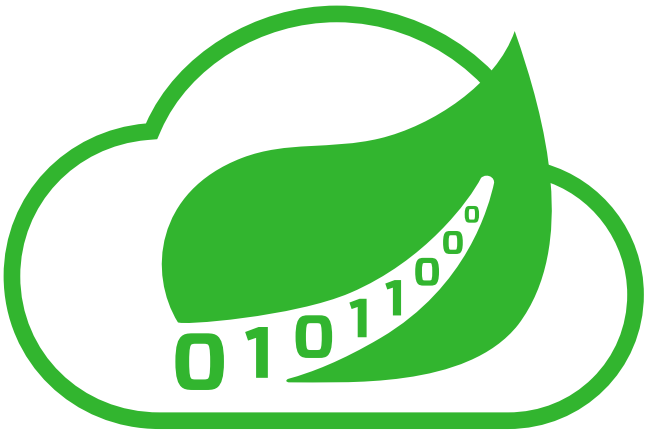

Spring Cloud Data Flow Server for Cloud Foundry

1.1.0.BUILD-SNAPSHOT

Sabby Anandan, Eric Bottard, Mark Fisher, Ilayaperumal Gopinathan, Gunnar Hillert, Mark Pollack, Thomas Risberg, Marius Bogoevici, Josh Long, Michael Minella

Table of Contents

Part I. Spring Cloud Data Flow for Cloud Foundry

This project provides support for orchestrating the deployment of Spring Cloud Stream applications to Cloud Foundry.

1. Spring Cloud Data Flow

Spring Cloud Data Flow is a cloud-native programming and operating model for composable data microservices on a structured platform. 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. A future release will also support deploying 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. The sections below describe more information about creating your own custom Streams.

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.

2. 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.

3. 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.

Unresolved directive in index.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v1.1.2.RELEASE/spring-cloud-dataflow-docs/src/main/asciidoc/architecture.adoc[]

Part II. Getting started

4. Deploying on Cloud Foundry

Spring Cloud Data Flow can be used to deploy modules in a Cloud Foundry environment. When doing so, the server application can either run itself on Cloud Foundry, or on another installation (e.g. a simple laptop).

The required configuration amounts to the same in either case, and is merely related to providing credentials to the Cloud Foundry instance so that the server can spawn applications itself. Any Spring Boot compatible configuration mechanism can be used (passing program arguments, editing configuration files before building the application, using Spring Cloud Config, using environment variables, etc.), although some may prove more practicable than others when running *on* Cloud Foundry.

Note

By default, the application registry in Spring Cloud Data Flow's Cloud Foundry server is empty. It is intentionally designed to allow users to have the flexibility of choosing and registering applications, as they find appropriate for the given use-case requirement. Depending on the message-binder of choice, users can register between RabbitMQ or Apache Kafka based maven artifacts.

4.1 Provision a Redis service instance on Cloud Foundry

Use `cf marketplace` to discover which plans are available to you, depending on the details of your Cloud Foundry setup. For example when using Pivotal Web Services:

```
cf create-service rediscloud 30mb redis
```

A redis instance is required for analytics apps, and would typically be bound to such apps when you create an analytics stream using the per-app-binding feature.

4.2 Provision a Rabbit service instance on Cloud Foundry

Use `cf marketplace` to discover which plans are available to you, depending on the details of your Cloud Foundry setup. For example when using Pivotal Web Services:

```
cf create-service cloudamqp lemur rabbit
```

Rabbit is typically used as a messaging middleware between streaming apps and would be bound to each deployed app thanks to the `SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES` setting (see below).

4.3 Provision a MySQL service instance on Cloud Foundry

Use `cf marketplace` to discover which plans are available to you, depending on the details of your Cloud Foundry setup. For example when using Pivotal Web Services:

```
cf create-service cleardb spark my_mysql
```

An RDBMS is used to persist Data Flow state, such as stream definitions and deployment ids. It can also be used for tasks to persist execution history.

4.4 Download the Spring Cloud Data Flow Server and Shell apps

```
wget http://repo.spring.io/snapshot/org/springframework/cloud/spring-cloud-dataflow-server-
cloudfoundry/1.1.0.BUILD-SNAPSHOT/spring-cloud-dataflow-server-cloudfoundry-1.1.0.BUILD-SNAPSHOT.jar
wget http://repo.spring.io/release/org/springframework/cloud/spring-cloud-dataflow-shell/1.1.2.RELEASE/
spring-cloud-dataflow-shell-1.1.2.RELEASE.jar
```

4.5 Running the Server

You can either deploy the server application on Cloud Foundry itself or on your local machine. The following two sections explain each way of running the server.

Deploying and Running the Server app on Cloud Foundry

Push the server application on Cloud Foundry, configure it (see below) and start it.

Note

You must use a unique name for your app; an app with the same name in the same organization will cause your deployment to fail

```
cf push dataflow-server -m 2G -k 2G --no-start -p spring-cloud-dataflow-server-cloudfoundry-1.1.0.BUILD-
SNAPSHOT.jar
cf bind-service dataflow-server redis
cf bind-service dataflow-server my_mysql
```

Important

The recommended minimal memory setting for the server is 2G. Also, to push apps to PCF and obtain application property metadata, the server downloads applications to Maven repository hosted on the local disk. While you can specify up to 2G as a typical maximum value for disk space on a PCF installation, this can be increased to 10G. Read the maximum disk quota section for information on how to configure this PCF property. Also, the Data Flow server itself implements a Last Recently Used algorithm to free disk space when it falls below a low water mark value.

Note

If you are pushing to a space with multiple users, for example on PWS, there may already be a route taken for the applicaiton name you have chosen. You can use the options `--random-route` to avoid this when pushing the app.

Now we can configure the app. The following configuration is for Pivotal Web Services. You need to fill in {org}, {space}, {email} and {password} before running these commands.

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL https://api.run.pivotal.io
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG {org}
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE {space}
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN cfapps.io
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES rabbit
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES my_mysql
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME {email}
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD {password}
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION false
```

Warning

Only set 'Skip SSL Validation' to true if you're running on a Cloud Foundry instance using self-signed certs (e.g. in development). Do not use for production.

Note

If you are deploying in an environment that requires you to sign on using the Pivotal Single Sign-On Service, refer to the section Chapter 7, *Authentication and Cloud Foundry* for information on how to configure the server.

Spring Cloud Data Flow server implementations (be it for Cloud Foundry, Mesos, YARN, or Kubernetes) do not have *any* default remote maven repository configured. This is intentionally designed to provide the flexibility for the users, so they can override and point to a remote repository of their choice. The out-of-the-box applications that are supported by Spring Cloud Data Flow are available in Spring's repository, so if you want to use them, you **must** set it as the remote repository as listed below.

```
cf set-env dataflow-server MAVEN_REMOTE_REPOSITORIES_REPO1_URL https://repo.spring.io/libs-snapshot
```

where `repo1` is an alias name for the remote repository.

Configuring Defaults for Deployed Apps

You can also set other optional properties that alter the way Spring Cloud Data Flow will deploy stream and task apps:

- The default memory and disk sizes for a deployed application can be configured. By default they are 1024 MB memory and 1024 MB disk. To change these, as an example to 512 and 2048 respectively, use

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_MEMORY 512
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_DISK 2048
```

- The default number of instances to deploy is set to 1, but can be overridden using

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_INSTANCES 1
```

- You can set the buildpack that will be used to deploy each application. For example, to use the Java offline buildback, set the following environment variable

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_BUILDPACK java_buildpack_offline
```

- The health check mechanism used by Cloud Foundry to assert if apps are running can be customized. Current supported options are `port` (the default) and `none`. Change the default like so:

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_HEALTH_CHECK none
```

Note

These settings can be configured separately for stream and task apps. To alter settings for tasks, simply substitute `STREAM` with `TASK` in the property name. As an example,

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_MEMORY 512
```

Tip

All the properties mentioned above are `@ConfigurationProperties` of the Cloud Foundry deployer. See CloudFoundryDeploymentProperties.java for more information.

- If you'd like to use `config-server` to manage centralized configurations for all the applications orchestrated by Spring Cloud Data Flow, you can set it up like the following.

```
cf set-env dataflow-server SPRING_APPLICATION_JSON
 '{"spring.cloud.dataflow.applicationProperties.stream.spring.cloud.config.uri": "http://
<CONFIG_SERVER_URI>"}'
```

We are now ready to start the app.

```
cf start dataflow-server
```

Alternatively, you can run the Admin application locally on your machine which is described in the next section.

Running the Server app locally

To run the server application locally, targeting your Cloud Foundry installation, you you need to configure the application either by passing in command line arguments (see below) or setting a number of environment variables.

To use environment variables set the following:

```
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL=https://api.run.pivotal.io
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG={org}
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE={space}
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN=cfapps.io
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME={email}
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD={password}
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION=false

export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES=rabbit
# The following is for letting task apps write to their db.
# Note however that when the *server* is running locally, it can't access that db
# task related commands that show executions won't work then
export SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES=my_mysql
```

You need to fill in {org}, {space}, {email} and {password} before running these commands.

Warning

Only set 'Skip SSL Validation' to true if you're running on a Cloud Foundry instance using self-signed certs (e.g. in development). Do not use for production.

Now we are ready to start the server application:

```
java -jar spring-cloud-dataflow-server-cloudfoundry-1.1.0.BUILD-SNAPSHOT.jar [--option1=value1] [--
option2=value2] [etc.]
```

Tip

Of course, all other parameterization options that were available when running the server *on* Cloud Foundry are still available. This is particularly true for configuring defaults for applications. Just substitute `cf set-env` syntax with `export`.

Note

The current underlying PCF task capabilities are considered experimental for PCF version versions less than 1.9. See Feature Togglers for how to disable task support in Data Flow.

4.6 Running Spring Cloud Data Flow Shell locally

Run the shell and optionally target the Admin application if not running on the same host (will typically be the case if deployed on Cloud Foundry as explained here)

```
$ java -jar spring-cloud-dataflow-shell-1.1.2.RELEASE.jar
```

```
server-unknown:>dataflow config server http://dataflow-server.cfapps.io
Successfully targeted http://dataflow-server.cfapps.io
dataflow:>
```

By default, the application registry will be empty. If you would like to register all out-of-the-box stream applications built with the 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/Avogadro-GA-stream-applications-rabbit-maven
```

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 | log" --deploy
```

Note

You will need to wait a little while until the apps are actually deployed successfully before posting data. Tail the log file for each application to verify the application has started.

Now post some data. The URL will be unique to your deployment, the following is just an example

```
dataflow:> http post --target http://dataflow-AxwwAhK-httptest-http.cfapps.io --data "hello world"
```

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

To run a simple task application, you can register all the out-of-the-box task applications with the following command.

```
dataflow:>app import --uri http://bit.ly/Addison-GA-task-applications-maven
```

Now create a simple timestamp task.

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

Tail the logs, e.g. `cf logs mytask` and then launch the task in the UI or in the Data Flow Shell

```
dataflow:>task launch mytask
```

You will see the year `2016` printed in the logs. The execution status of the task is stored in the database and you can retrieve information about the task execution using the shell commands `task execution list` and `task execution status --id <ID_OF_TASK>` or though the Data Flow UI.

5. Security

By default, the Data Flow server is unsecured and runs on an unencrypted HTTP connection. You can secure your REST endpoints, as well as the Data Flow Dashboard by enabling HTTPS and requiring clients to authenticate. More details about securing the REST endpoints and configuring to authenticate against an OAUTH backend (*i.e: UAA/SSO running on Cloud Foundry*), please review the security section from the core reference guide. The security configurations can be configured in `dataflow-server.yml` or passed as environment variables through `cf set-env` commands.

6. Application Names and Prefixes

To help avoid clashes with routes across spaces in Cloud Foundry, a naming strategy to provide a random prefix to a deployed application is available and is enabled by default. The default configurations are overridable and the respective properties can be set via `cf set-env` commands.

For instance, if you'd like to disable the randomization, you can override it through:

```
cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_ENABLE_RANDOM_APP_NAME_PREFIX false
```

6.1 Using Custom Routes

As an alternative to random name, or to get even more control over the hostname used by the deployed apps, one can use custom deployment properties, as such:

[[source]

```
dataflow:>stream create foo --definition "http | log"

dataflow:>stream deploy foo --properties
 "app.http.spring.cloud.deployer.cloudfoundry.domain=mydomain.com,
                                        app.http.spring.cloud.deployer.cloudfoundry.host=myhost,
                                        app.http.spring.cloud.deployer.cloudfoundry.route-path=my-
path"
```

This would result in the `http` app being bound to the URL `myhost.mydomain.com/my-path`. Note that this is an example showing **all** customization options available. One can of course only leverage one or two out of the three.

7. Authentication and Cloud Foundry

When deploying Spring Cloud Data Flow to Cloud Foundry, you can take advantage of the *Spring Cloud Single Sign-On Connector*, which provides Cloud Foundry specific auto-configuration support for OAuth 2.0, when used in conjunction with the *Pivotal Single Sign-On Service*.

Simply set `security.basic.enabled` to `true` and in Cloud Foundry bind the SSO service to your Data Flow Server app and SSO will be enabled.

8. Configuration Reference

The following pieces of configuration must be provided. These are Spring Boot `@ConfigurationProperties` so you can set them as environment variables or by any other means that Spring Boot supports. Here is a listing in environment variable format as that is an easy way to get started configuring Boot applications in Cloud Foundry.

```
# Default values cited after the equal sign.
# Example values, typical for Pivotal Web Services, cited as a comment

# url of the CF API (used when using cf login -a for example), e.g. https://api.run.pivotal.io
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL)
spring.cloud.deployer.cloudfoundry.url=

# name of the organization that owns the space above, e.g. youruser-org
# (For Setting Env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG)
spring.cloud.deployer.cloudfoundry.org=

# name of the space into which modules will be deployed, e.g. development
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE)
spring.cloud.deployer.cloudfoundry.space=

# the root domain to use when mapping routes, e.g. cfapps.io
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN)
spring.cloud.deployer.cloudfoundry.domain=

# username and password of the user to use to create apps
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME and
 SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD)
spring.cloud.deployer.cloudfoundry.username=
spring.cloud.deployer.cloudfoundry.password=

# Whether to allow self-signed certificates during SSL validation
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION)
spring.cloud.deployer.cloudfoundry.skipSslValidation=false

# Comma separated set of service instance names to bind to every stream app deployed.
# Amongst other things, this should include a service that will be used
# for Spring Cloud Stream binding, e.g. rabbit
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES)
spring.cloud.deployer.cloudfoundry.stream.services=

# Health check type to use for stream apps. Accepts 'none' and 'port'
spring.cloud.deployer.cloudfoundry.stream.health-check=


# Comma separated set of service instance names to bind to every task app deployed.
# Amongst other things, this should include an RDBMS service that will be used
# for Spring Cloud Task execution reporting, e.g. my_mysql
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES)
spring.cloud.deployer.cloudfoundry.task.services=

# Timeout to use, in seconds, when doing blocking API calls to Cloud Foundry.
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_API_TIMEOUT
and SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_API_TIMEOUT)
spring.cloud.deployer.cloudfoundry.stream.apiTimeout=360
spring.cloud.deployer.cloudfoundry.task.apiTimeout=360
```

Note that you can set the following properties `spring.cloud.deployer.cloudfoundry.services`, `spring.cloud.deployer.cloudfoundry.buildpack` or the Spring Cloud Deployer standard `spring.cloud.deployer.memory` and `spring.cloud.deployer.disk` as part of an individual deployment request prefixed by the `app.<name of application>`. For example

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

```
>stream deploy --name ticktock --properties "app.time.spring.cloud.deployer.memory=2g"
```

will deploy the time source with 2048MB of memory, while the log sink will use the default 1024MB.

8.1 Using Spring Cloud Config Server

If using Spring Cloud Config Server as a Cloud Foundry service, the easiest way to externalize the above configuration and consume it from the Data Flow server is to use the `spring-cloud-services-starter-config-client` dependency, which is included in the standard distribution of the Spring Cloud Data Flow server for Cloud Foundry.

Follow the documentation for Config Server for Pivotal Cloud Foundry. For more details, please refer to Spring Cloud Services client-dependencies documentation.

9. Application Level Service Bindings

When deploying streams in Cloud Foundry, you can take advantage of application specific service bindings, so not all services are globally configured for all the apps orchestrated by Spring Cloud Data Flow.

For instance, if you'd like to provide `mysql` service binding only for the `jdbc` application in the following stream definition, you can pass the service binding as a deployment property.

```
dataflow:>stream create --name httptojdbc --definition "http | jdbc"
dataflow:>stream deploy --name httptojdbc --properties
 "app.jdbc.spring.cloud.deployer.cloudfoundry.services=mysqlService"
```

Where, `mysqlService` is the name of the service specifically only bound to `jdbc` application and the `http` application wouldn't get the binding by this method. If you have more than one service to bind, they can be passed as comma separated items (*eg: app.jdbc.spring.cloud.deployer.cloudfoundry.services=mysqlService,someService*).

10. A Note About User Provided Services

In addition to marketplace services, Cloud Foundry supports User Provided Services. Throughout this reference manual, regular services have been mentioned, but there is nothing precluding the use of UPSs as well, whether for use as the messaging middleware (*e.g.* if you'd like to use an external Apache Kafka installation) or for *ad hoc* usage by some of the stream apps (*e.g.* an Oracle Database).

11. Application Rolling Upgrades

Similar to Cloud Foundry's blue-green deployments, you can perform rolling upgrades on the applications orchestrated by Spring Cloud Data Flow.

Let's start with the following simple stream definition.

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

List Apps.

```
# cf apps
Getting apps in org test-org / space development as test@pivotal.io...
OK

name       requested state   instances   memory   disk   urls
foo-log    started           1/1         1G       1G     foo-log.cfapps.io
foo-time   started           1/1         1G       1G     foo-time.cfapps.io
```

Let's assume you've to make an enhancement to update the "logger" to append extra text in every log statement.

- Download the `Log Sink` application starter with "Rabbit binder starter" from start-scs.cfapps.io/
- Load the downloaded project in an IDE
- Import the `LogSinkConfiguration.class`
- Adapt the handler to add extra text: `loggingHandler.setLoggerName("TEST [" + this.properties.getName() + "]");`
- Build the application locally

```
@SpringBootApplication
@Import(LogSinkConfiguration.class)
public class DemoApplication {

 @Autowired
 private LogSinkProperties properties;

 public static void main(String[] args) {
  SpringApplication.run(DemoApplication.class, args);
 }

 @Bean
 @ServiceActivator(inputChannel = Sink.INPUT)
 public LoggingHandler logSinkHandler() {
  LoggingHandler loggingHandler = new LoggingHandler(this.properties.getLevel().name());
  loggingHandler.setExpression(this.properties.getExpression());
  loggingHandler.setLoggerName("TEST [" + this.properties.getName() + "]");
  return loggingHandler;
 }
}
```

Let's deploy the locally built application to Cloud Foundry

```
# cf push foo-log-v2 -p demo-0.0.1-SNAPSHOT.jar -n foo-log-v2 --no-start
```

List Apps.

```
# cf apps
Getting apps in org test-org / space development as test@pivotal.io...
```

```
OK

name       requested state   instances   memory   disk   urls
foo-log    started           1/1         1G       1G     foo-log.cfapps.io
foo-time   started           1/1         1G       1G     foo-time.cfapps.io
foo-log-v2 stopped           1/1         1G       1G     foo-log-v2.cfapps.io
```

The stream applications do not communicate via (Go)Router, so they aren't generating HTTP traffic. Instead, they communicate via the underlying messaging middleware such as Kafka or RabbitMQ. In order to rolling upgrade to route the payload from old to the new version of the application, you'd have to replicate the `SPRING_APPLICATION_JSON` environment variable from the old application that includes `spring.cloud.stream.bindings.input.destination` and `spring.cloud.stream.bindings.input.group` credentials.

Note

You can find the `SPRING_APPLICATION_JSON` of the old application via: `"cf env foo-log"`.

```
cf set-env foo-log-v2
 SPRING_APPLICATION_JSON '{"spring.cloud.stream.bindings.input.destination":"foo.time","spring.cloud.stream.bindings.input.
```

Let's start `foo-log-v2` application.

```
cf start foo-log-v2
```

As soon as the application bootstraps, you'd now notice the payload being load balanced between two log application instances running on Cloud Foundry. Since they both share the same "destination" and "consumer group", they are now acting as competing consumers.

Old App Logs:

```
2016-08-08T17:11:08.94-0700 [APP/0]      OUT 2016-08-09 00:11:08.942  INFO 19 --- [ foo.time.foo-1]
 log.sink                                 : 08/09/16 00:11:08
2016-08-08T17:11:10.95-0700 [APP/0]      OUT 2016-08-09 00:11:10.954  INFO 19 --- [ foo.time.foo-1]
 log.sink                                 : 08/09/16 00:11:10
2016-08-08T17:11:12.94-0700 [APP/0]      OUT 2016-08-09 00:11:12.944  INFO 19 --- [ foo.time.foo-1]
 log.sink                                 : 08/09/16 00:11:12
```

New App Logs:

```
2016-08-08T17:11:07.94-0700 [APP/0]      OUT 2016-08-09 00:11:07.945  INFO 26 --- [ foo.time.foo-1] TEST
 [log.sink                       : 08/09/16 00:11:07]
2016-08-08T17:11:09.92-0700 [APP/0]      OUT 2016-08-09 00:11:09.925  INFO 26 --- [ foo.time.foo-1] TEST
 [log.sink                       : 08/09/16 00:11:09]
2016-08-08T17:11:11.94-0700 [APP/0]      OUT 2016-08-09 00:11:11.941  INFO 26 --- [ foo.time.foo-1] TEST
 [log.sink                       : 08/09/16 00:11:11]
```

Deleting the old version `foo-log` from the CF CLI would make all the payload consumed by the `foo-log-v2` application. Now, you've successfully upgraded an application in the streaming pipeline without bringing it down in entirety to do an adjustment in it.

List Apps.

```
# cf apps
Getting apps in org test-org / space development as test@pivotal.io...
OK

name       requested state   instances   memory   disk   urls
foo-time   started           1/1         1G       1G     foo-time.cfapps.io
foo-log-v2 started           1/1         1G       1G     foo-log-v2.cfapps.io
```

Note

A comprehensive canary analysis along with rolling upgrades will be supported via Spinnaker in future releases.

12. Maximum Disk Quota Configuration

By default, every application in Cloud Foundry starts with 1G disk quota and this can be adjusted to a default maximum of 2G. The default maximum can also be overridden up to 10G via Pivotal Cloud Foundry's (PCF) Ops Manager GUI.

This configuration is relevant for Spring Cloud Data Flow because every stream and task deployment is composed of applications (typically Spring Boot uber-jar's) and those applications are resolved from a remote maven repository. After resolution, the application artifacts are downloaded to the local Maven Repository for caching/reuse. With this happening in the background, there is a possibility the default disk quota (*1G*) fills up rapidly; especially, when we are experimenting with streams that are made up of unique applications. In order to overcome this disk limitation and depending on your scaling requirements,you may want to change the default maximum from 2G to 10G. Let's review the steps to change the default maximum disk quota allocation.

12.1 PCF's Operations Manager Configuration

From PCF's Ops Manager, Select "**Pivotal Elastic Runtime**" tile and navigate to "**Application Developer Controls**" tab. Change the "**Maximum Disk Quota per App (MB)**" setting from 2048 to 10240 (*10G*). Save the disk quota update and hit "Apply Changes" to complete the configuration override.

12.2 Scale Application

Once the disk quota change is applied successfully and assuming you've a running application, you may scale the application with a new `disk_limit` through CF CLI.

```
# cf scale dataflow-server -k 10GB

Scaling app dataflow-server in org ORG / space SPACE as user...
OK

....
....
....
....

     state     since                    cpu      memory           disk           details
#0   running   2016-10-31 03:07:23 PM   1.8%     497.9M of 1.1G   193.9M of 10G
```

```
# cf apps
Getting apps in org ORG / space SPACE as user...
OK

name              requested state   instances   memory   disk   urls
dataflow-server   started           1/1         1.1G     10G    dataflow-server.apps.io
```

12.3 Configuring target free disk percentage

Even when configuring the Data Flow server to use 10G of space, there is the possibility of exhausting the available space on the local disk. The server implements a least recently used (LRU) algorithm that will remove maven artifacts from the local maven repository. This is configured using the following configuration property, the default value is 25.

```
# The low water mark percentage, expressed as in integer between 0 and 100, that triggers cleanup of
# the local maven repository
```

```
# (for setting env var use SPRING_CLOUD_DATAFLOW_SERVER_CLOUDFOUNDRY_FREE_DISK_SPACE_PERCENTAGE)
spring.cloud.dataflow.server.cloudfoundry.freeDiskSpacePercentage=25
```

Unresolved directive in index.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v1.1.2.RELEASE/spring-cloud-dataflow-docs/src/main/asciidoc/configuration.adoc[]
Unresolved directive in index.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v1.1.2.RELEASE/spring-cloud-dataflow-docs/src/main/asciidoc/streams.adoc[]
Unresolved directive in index.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v1.1.2.RELEASE/spring-cloud-dataflow-docs/src/main/asciidoc/tasks.adoc[]

Part III. Tasks on Cloud Foundry

Spring Cloud Data Flow's task functionality exposes new task capabilities within the Pivotal Cloud Foundry runtime. It is important to note that the current underlying PCF task capabilities are considered experimental for PCF version versions less than 1.9. See ??? for how to disable task support in Data Flow.

13. Version Compatibility

The task functionality depends on the latest versions of PCF for runtime support. This release requires PCF version 1.7.12 or higher to run tasks. Tasks are an experimental feature in PCF 1.7 and 1.8 and a GA feature in PCF 1.9.

14. Tooling

Because the task functionality is experimental in PCF for versions less than 1.9, the tooling around it within the CF ecosystem is not complete. In order to interact with tasks via the PCF command line interface (CLI) for versions less than 1.9, you need to install a plugin: v3-cli-plugin. It's important to note that this plugin is only compatible with the PCF CLI version 6.17.0+5d0be0a-2016-04-15. You can read more about the functionality the plugin provides in its README.

It's also important to note that there is no Apps Manager support for tasks as of this release. When running applications as tasks through Spring Cloud Data Flow, the only way to view them within the context of CF is via the plugin mentioned above.

15. Task Database Schema

The database schema for Task applications was changed slighlty from the 1.0.x to 1.1.x version of Spring Cloud Task. Since Spring Cloud Data Flow automatically creates the database schema if it is not present upon server startup, you may need to update the schema if you ran a 1.0.x version of the Data Flow server and now are upgrading to the 1.1.x version. You can find the migration scripts here in the Spring Cloud Task Github repository. The documentation for Accessing Services with Diego SSH and this blog entry for connecting a GUI tools to the MySQL Service in PCF should help you to update the schema.

16. Running Task Applications

Running a task application within Spring Cloud Data Flow goes through a slightly different lifecycle than running a stream application. Both types of applications need to be registered with the appropriate artifact coordinates. Both need a definition created via the SCDF DSL. However, that's where the similarities end.

With stream based applications, you "deploy" them with the intent that they run until they are undeployed. A stream definition is only deployed once (it can be scaled, but only deployed as one instance of the stream as a whole). However, tasks are *launched*. A single task definition can be launched many times. With each launch, they will start, execute, and shut down with PCF cleaning up the resources once the shutdown has occurred. The following sections outline the process of creating, launching, destroying, and viewing tasks.

16.1 Create a Task

Similar to streams, creating a task application is done via the SCDF DSL or through the dashboard. To create a task definition in SCDF, you've to either develop a task application or use one of the out-of-the-box task app-starters. The maven coordinates of the task application should be registered in SCDF. For more details on how to register task applications, review register task applications section from the core docs.

Let's see an example that uses the out-of-the-box `timestamp` task application.

```
dataflow:>task create --name foo --definition "timestamp"
Created new task 'foo'
```

Note

Tasks in SCDF do not require explicit deployment. They are required to be launched and with that there are different ways to launch them - refer to this section for more details.

16.2 Launch a Task

Unlike streams, tasks in SCDF requires an explicit launch trigger or it can be manually kicked-off.

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

16.3 View Task Logs

As previously mentioned, the v3-cli-plugin is the way to interact with tasks on PCF, including viewing the logs. In order to view the logs as a task is executing use the following command where `foo` is the name of the task you are executing:

```
cf v3-logs foo
Tailing logs for app foo...

....
....
....
....

2016-08-19T09:44:49.11-0700 [APP/TASK/bar1/0]OUT 2016-08-19 16:44:49.111  INFO 7 --- [           main]
 o.s.c.t.a.t.TimestampTaskApplication     : Started TimestampTaskApplication in 2.734 seconds (JVM
 running for 3.288)
```

```
2016-08-19T09:44:49.13-0700 [APP/TASK/bar1/0]OUT Exit status 0
2016-08-19T09:44:49.19-0700 [APP/TASK/bar1/0]OUT Destroying container
2016-08-19T09:44:50.41-0700 [APP/TASK/bar1/0]OUT Successfully destroyed container
```

Note

Logs are only viewable through the v3-cli-plugin as the app is running. Historic logs are not available.

16.4 List Tasks

Listing tasks is as simple as:

```
dataflow:>task list
############################################################
#      Task Name       #     Task Definition     #Task Status#
############################################################
#foo                   #timestamp                #complete   #
############################################################
```

16.5 List Task Executions

If you'd like to view the execution details of the launched task, you could do the following.

```
dataflow:>task execution list
#########################################################################################################
#       Task Name        #ID#       Start Time        #        End Time         #  Exit  #
#                        #  #                         #                         #  Code  #
#########################################################################################################
#foo:cloud:              #1 # Fri Aug 19 09:44:49 PDT #Fri Aug 19 09:44:49 PDT  #0       #
#########################################################################################################
```

16.6 Destroy a Task

Destroying the task application from SCDF removes the task definition from task repository.

```
dataflow:>task destroy foo
Destroyed task 'foo'
dataflow:>task list
######################################
#Task Name#Task Definition#Task Status#
######################################
```

16.7 Deleting Task From Cloud Foundry

Currently Spring Cloud Data Flow does not delete tasks deployed on a Cloud Foundry instance once they have been pushed. The only way to do this now is via CLI on a Cloud Foundry instance version 1.9 or above. This is done in 2 steps:

1. Obtain a list of the apps via the `cf apps` command.
2. Identify the task app to be deleted and execute the `cf delete <task-name>` command.

Note

The `task destroy <task-name>` only deletes the definition and not the task deployed on Cloud Foundry.

Unresolved directive in index.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v1.1.2.RELEASE/spring-cloud-dataflow-docs/src/main/asciidoc/dashboard.adoc[]

Unresolved directive in index.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v1.1.2.RELEASE/spring-cloud-dataflow-docs/src/main/asciidoc/howto.adoc[]

Part IV. Appendices

Unresolved directive in appendix.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/v1.1.2.RELEASE/spring-cloud-dataflow-docs/src/main/asciidoc/appendix-migration-guide.adoc[]