



Spring Cloud Data Flow Server for Cloud Foundry

1.2.1.RELEASE

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

Copyright © 2013-2017 Pivotal Software, Inc.

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

Table of Contents

I. Spring Cloud Data Flow for Cloud Foundry	1
1. Spring Cloud Data Flow	2
2. Spring Cloud Stream	3
3. Spring Cloud Task	4
II. Architecture	5
4. Introduction	6
5. Microservice Architectural Style	8
5.1. Comparison to other Platform architectures	8
6. Streaming Applications	10
6.1. Imperative Programming Model	10
6.2. Functional Programming Model	10
7. Streams	11
7.1. Topologies	11
7.2. Concurrency	11
7.3. Partitioning	11
7.4. Message Delivery Guarantees	12
8. Analytics	14
9. Task Applications	15
10. Data Flow Server	16
10.1. Endpoints	16
10.2. Customization	16
10.3. Security	17
11. Runtime	18
11.1. Fault Tolerance	18
11.2. Resource Management	18
11.3. Scaling at runtime	18
11.4. Application Versioning	18
III. Getting started	19
12. Deploying on Cloud Foundry	20
12.1. Provision a Redis service instance on Cloud Foundry	20
12.2. Provision a Rabbit service instance on Cloud Foundry	20
12.3. Provision a MySQL service instance on Cloud Foundry	20
12.4. Download the Spring Cloud Data Flow Server and Shell apps	21
12.5. Running the Server	21
Deploying and Running the Server app on Cloud Foundry	21
Configuring Defaults for Deployed Apps	22
Running the Server app locally	23
Sample Manifest Template	24
12.6. Running Spring Cloud Data Flow Shell locally	24
13. Security	26
13.1. Authentication and Cloud Foundry	26
Pivotal Single Sign-On Service	26
Cloud Foundry UAA	27
14. Application Names and Prefixes	28
14.1. Using Custom Routes	28
15. Deploying Docker Applications	29
16. Configuration Reference	30

16.1. Understanding what's going on	31
16.2. Using Spring Cloud Config Server	31
Stream, Task, and Spring Cloud Config Server	31
Sample Manifest Template	32
Self-signed SSL Certificate and Spring Cloud Config Server	32
17. Application Level Service Bindings	34
18. A Note About User Provided Services	35
19. Application Rolling Upgrades	36
20. Maximum Disk Quota Configuration	39
20.1. PCF's Operations Manager Configuration	39
20.2. Scale Application	39
20.3. Configuring target free disk percentage	39
IV. Server Configuration	41
21. Feature Toggles	42
22. Database Configuration	43
23. Security	44
23.1. Enabling HTTPS	44
Using Self-Signed Certificates	45
Self-Signed Certificates and the Shell	45
23.2. Traditional Authentication	46
Single User Authentication	46
File based authentication	48
LDAP Authentication	48
LDAP Transport Security	49
Shell Authentication	50
Customizing authorization	50
Authorization - Shell and Dashboard Behavior	53
Authorization with Ldap	53
23.3. OAuth 2.0	53
OAuth REST Endpoint Authorization	54
OAuth Authentication using the Spring Cloud Data Flow Shell	54
OAuth2 Authentication Examples	55
Local OAuth2 Server	55
Authentication using GitHub	55
23.4. Securing the Spring Boot Management Endpoints	56
24. Monitoring and Management	58
24.1. Spring Boot Admin	58
24.2. Monitoring Deployed Applications	59
24.3. Log and DataDog MetricWriter	62
V. Streams	63
25. Introduction	64
26. Stream DSL	65
27. Register a Stream App	66
27.1. Whitelisting application properties	68
27.2. Creating and using a dedicated metadata artifact	68
Using the companion artifact	69
28. Creating custom applications	71
29. Creating a Stream	72
29.1. Application properties	72
Passing application properties when creating a stream	72

29.2. Deployment properties	74
Application properties versus Deployer properties	74
Passing instance count as deployment property	74
Inline vs file reference properties	75
Passing application properties when deploying a stream	75
Passing Spring Cloud Stream properties for the application	76
Passing per-binding producer consumer properties	76
Passing stream partition properties during stream deployment	77
Passing application content type properties	77
Overriding application properties during stream deployment	78
29.3. Common application properties	78
30. Destroying a Stream	80
31. Deploying and Undeploying Streams	81
32. Other Source and Sink Application Types	82
33. Simple Stream Processing	83
34. Stateful Stream Processing	84
35. Tap a Stream	85
36. Using Labels in a Stream	86
37. Explicit Broker Destinations in a Stream	87
38. Directed Graphs in a Stream	88
39. Stream applications with multiple binder configurations	89
VI. Tasks	90
40. Introducing Spring Cloud Task	91
41. The Lifecycle of a task	92
41.1. Creating a custom Task Application	92
41.2. Registering a Task Application	92
41.3. Creating a Task	93
41.4. Launching a Task	94
Common application properties	94
41.5. Reviewing Task Executions	94
41.6. Destroying a Task	95
42. Task Repository	96
42.1. Configuring the Task Execution Repository	96
Local	96
Task Application Repository	96
42.2. Datasource	96
43. Subscribing to Task/Batch Events	98
44. Launching Tasks from a Stream	99
44.1. TriggerTask	99
44.2. TaskLaunchRequest-transform	100
45. Composed Tasks	101
45.1. Configuring the Composed Task Runner in Spring Cloud Data Flow	101
Registering the Composed Task Runner application	101
Configuring the Composed Task Runner application	101
45.2. Creating, Launching, and Destroying a Composed Task	101
Creating a Composed Task	101
Task Application Parameters	102
Launching a Composed Task	102
Exit Statuses	102
Destroying a Composed Task	103

Stopping a Composed Task	103
Restarting a Composed Task	103
45.3. Composed Task DSL	103
Conditional Execution	103
Transitional Execution	105
Basic Transition	105
Transition With a Wildcard	106
Transition With a Following Conditional Execution	107
Split Execution	108
Split Containing Conditional Execution	109
VII. Tasks on Cloud Foundry	111
46. Version Compatibility	112
47. Tooling	113
48. Task Database Schema	114
49. Running Task Applications	115
49.1. Create a Task	115
49.2. Launch a Task	115
49.3. View Task Logs	115
49.4. List Tasks	116
49.5. List Task Executions	116
49.6. Destroy a Task	116
49.7. Deleting Task From Cloud Foundry	116
VIII. Dashboard	117
50. Introduction	118
51. Apps	119
51.1. Bulk Import of Applications	119
52. Runtime	121
53. Streams	122
54. Create Stream	124
55. Tasks	125
55.1. Apps	125
Create a Task Definition from a selected Task App	125
View Task App Details	126
55.2. Definitions	126
Creating Task Definitions using the bulk define interface	126
Creating Composed Task Definitions	127
Launching Tasks	128
55.3. Executions	129
56. Jobs	130
56.1. List job executions	130
Job execution details	131
Step execution details	131
Step Execution Progress	131
57. Analytics	133
IX. 'How-to' guides	134
58. Configure Maven Properties	135
59. Logging	137
59.1. Deployment Logs	137
59.2. Application Logs	137
60. Frequently asked questions	139

60.1. Advanced SpEL expressions	139
60.2. How to use JDBC-sink?	139
60.3. How to use multiple message-binders?	140
X. Appendices	142
A. Migrating from Spring XD to Spring Cloud Data Flow	143
A.1. Terminology Changes	143
A.2. Modules to Applications	143
Custom Applications	143
Application Registration	143
Application Properties	144
A.3. Message Bus to Binders	144
Message Bus	144
Binders	144
Named Channels	145
Directed Graphs	145
A.4. Batch to Tasks	145
A.5. Shell/DSL Commands	146
A.6. REST-API	146
A.7. UI / Flo	146
A.8. Architecture Components	147
ZooKeeper	147
RDBMS	147
Redis	147
Cluster Topology	147
A.9. Central Configuration	147
A.10. Distribution	147
A.11. Hadoop Distribution Compatibility	148
A.12. YARN Deployment	148
A.13. Use Case Comparison	148
Use Case #1	148
Use Case #2	149
Use Case #3	149
B. Building	151
B.1. Basic Compile and Test	151
B.2. Documentation	151
B.3. Working with the code	151
Importing into eclipse with m2eclipse	151
Importing into eclipse without m2eclipse	152

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 orchestration service for composable microservice applications on modern runtimes. With Spring Cloud Data Flow, developers can create and orchestrate data pipelines for common use cases such as data ingest, real-time analytics, and data import/export.

The Spring Cloud Data Flow architecture consists of a server that deploys [Streams](#) and [Tasks](#). Streams and Tasks are defined using a [DSL](#) or visually through the browser based designer UI. Streams and Tasks are based on [Spring Cloud Stream](#) and [Spring Cloud Task](#) programming models respectively.

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 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 microservice applications or in Spring Cloud Data Flow. For convenience, the generated application-starters are available for use from [Maven Repo](#). There are several [samples](#) available for reference.

Part II. Architecture

4. Introduction

Spring Cloud Data Flow simplifies the development and deployment of applications focused on data processing use-cases. The major concepts of the architecture are Applications, the Data Flow Server, and the target runtime.

Applications come in two flavors

- Long lived Stream applications where an unbounded amount of data is consumed or produced via messaging middleware.
- Short lived Task applications that process a finite set of data and then terminate.

Depending on the runtime, applications can be packaged in two ways

- Spring Boot uber-jar that is hosted in a maven repository, file, http or any other Spring resource implementation.
- Docker

The runtime is the place where applications execute. The target runtimes for applications are platforms that you may already be using for other application deployments.

The supported runtimes are

- Cloud Foundry
- Apache YARN
- Kubernetes
- Apache Mesos
- Local Server for development

There is a deployer Service Provider Interface (SPI) that enables you to extend Data Flow to deploy onto other runtimes, for example to support Docker Swarm. There are community implementations of Hashicorp's Nomad and RedHat Openshift is available. We look forward to working with the community for further contributions!

The component that is responsible for deploying applications to a runtime is the Data Flow Server. There is a Data Flow Server executable jar provided for each of the target runtimes. The Data Flow server is responsible for interpreting

- A stream DSL that describes the logical flow of data through multiple applications.
- A deployment manifest that describes the mapping of applications onto the runtime. For example, to set the initial number of instances, memory requirements, and data partitioning.

As an example, the DSL to describe the flow of data from an http source to an Apache Cassandra sink would be written as "http | cassandra". These names in the DSL are registered with the Data Flow Server and map onto application artifacts that can be hosted in Maven or Docker repositories. Many source, processor, and sink applications for common use-cases (e.g. jdbc, hdfs, http, router) are provided by the Spring Cloud Data Flow team. The pipe symbol represents the communication between the two applications via messaging middleware. The two messaging middleware brokers that are supported are

- Apache Kafka
- RabbitMQ

In the case of Kafka, when deploying the stream, the Data Flow server is responsible to create the topics that correspond to each pipe symbol and configure each application to produce or consume from the topics so the desired flow of data is achieved.

The interaction of the main components is shown below

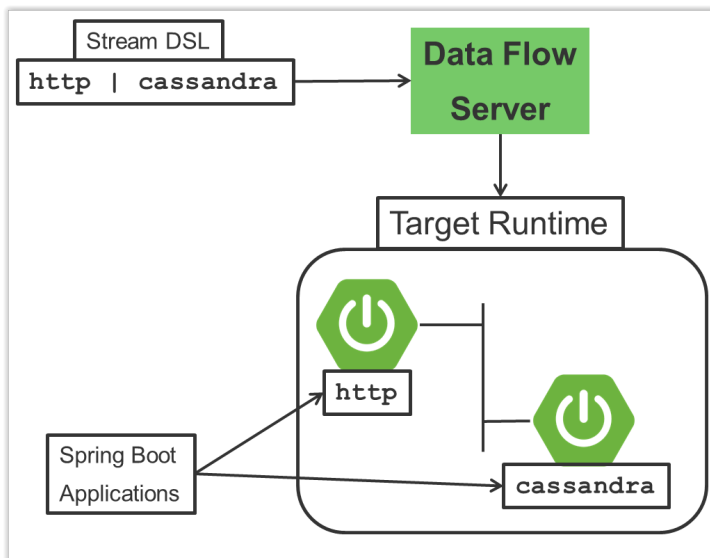


Figure 4.1. The Spring Cloud Data High Level Architecture

In this diagram a DSL description of a stream is POSTed to the Data Flow Server. Based on the mapping of DSL application names to Maven and Docker artifacts, the http-source and cassandra-sink applications are deployed on the target runtime.

5. Microservice Architectural Style

The Data Flow Server deploys applications onto the target runtime that conform to the microservice architectural style. For example, a stream represents a high level application that consists of multiple small microservice applications each running in their own process. Each microservice application can be scaled up or down independent of the other and each has their own versioning lifecycle.

Both Streaming and Task based microservice applications build upon Spring Boot as the foundational library. This gives all microservice applications functionality such as health checks, security, configurable logging, monitoring and management functionality, as well as executable JAR packaging.

It is important to emphasise that these microservice applications are ‘just apps’ that you can run by yourself using ‘java -jar’ and passing in appropriate configuration properties. We provide many common microservice applications for common operations so you don’t have to start from scratch when addressing common use-cases which build upon the rich ecosystem of Spring Projects, e.g Spring Integration, Spring Data, Spring Hadoop and Spring Batch. Creating your own microservice application is similar to creating other Spring Boot applications, you can start using the Spring Initializr web site or the UI to create the basic scaffolding of either a Stream or Task based microservice.

In addition to passing in the appropriate configuration to the applications, the Data Flow server is responsible for preparing the target platform’s infrastructure so that the application can be deployed. For example, in Cloud Foundry it would be binding specified services to the applications and executing the ‘cf push’ command for each application. For Kubernetes it would be creating the replication controller, service, and load balancer.

The Data Flow Server helps simplify the deployment of multiple applications onto a target runtime, but one could also opt to deploy each of the microservice applications manually and not use Data Flow at all. This approach might be more appropriate to start out with for small scale deployments, gradually adopting the convenience and consistency of Data Flow as you develop more applications. Manual deployment of Stream and Task based microservices is also a useful educational exercise that will help you better understand some of the automatic applications configuration and platform targeting steps that the Data Flow Server provides.

5.1 Comparison to other Platform architectures

Spring Cloud Data Flow’s architectural style is different than other Stream and Batch processing platforms. For example in Apache Spark, Apache Flink, and Google Cloud Dataflow applications run on a dedicated compute engine cluster. The nature of the compute engine gives these platforms a richer environment for performing complex calculations on the data as compared to Spring Cloud Data Flow, but it introduces complexity of another execution environment that is often not needed when creating data centric applications. That doesn’t mean you cannot do real time data computations when using Spring Cloud Data Flow. Refer to the analytics section which describes the integration of Redis to handle common counting based use-cases as well as the RxJava integration for functional API driven analytics use-cases, such as time-sliding-window and moving-average among others.

Similarly, Apache Storm, Hortonworks DataFlow and Spring Cloud Data Flow’s predecessor, Spring XD, use a dedicated application execution cluster, unique to each product, that determines where your code should execute on the cluster and perform health checks to ensure that long lived applications are restarted if they fail. Often, framework specific interfaces are required to be used in order to correctly “plug in” to the cluster’s execution framework.

As we discovered during the evolution of Spring XD, the rise of multiple container frameworks in 2015 made creating our own runtime a duplication of efforts. There is no reason to build your own resource management mechanics, when there are multiple runtime platforms that offer this functionality already. Taking these considerations into account is what made us shift to the current architecture where we delegate the execution to popular runtimes, runtimes that you may already be using for other purposes. This is an advantage in that it reduces the cognitive distance for creating and managing data centric applications as many of the same skills used for deploying other end-user/web applications are applicable.

6. Streaming Applications

While Spring Boot provides the foundation for creating DevOps friendly microservice applications, other libraries in the Spring ecosystem help create Stream based microservice applications. The most important of these is Spring Cloud Stream.

The essence of the Spring Cloud Stream programming model is to provide an easy way to describe multiple inputs and outputs of an application that communicate over messaging middleware. These input and outputs map onto Kafka topics or Rabbit exchanges and queues. Common application configuration for a Source that generates data, a Process that consumes and produces data and a Sink that consumes data is provided as part of the library.

6.1 Imperative Programming Model

Spring Cloud Stream is most closely integrated with Spring Integration's imperative "event at a time" programming model. This means you write code that handles a single event callback. For example,

```
@EnableBinding(Sink.class)
public class LoggingSink {

    @StreamListener(Sink.INPUT)
    public void log(String message) {
        System.out.println(message);
    }
}
```

In this case the String payload of a message coming on the input channel, is handed to the log method. The `@EnableBinding` annotation is what is used to tie together the input channel to the external middleware.

6.2 Functional Programming Model

However, Spring Cloud Stream can support other programming styles. The use of reactive APIs where incoming and outgoing data is handled as continuous data flows and it defines how each individual message should be handled. You can also use operators that describe functional transformations from inbound to outbound data flows. The upcoming versions will support Apache Kafka's KStream API in the programming model.

7. Streams

7.1 Topologies

The Stream DSL describes linear sequences of data flowing through the system. For example, in the stream definition `http | transformer | cassandra`, each pipe symbol connects the application on the left to the one on the right. Named channels can be used for routing and to fan out data to multiple messaging destinations.

Taps can be used to ‘listen in’ to the data that is flowing across any of the pipe symbols. Taps can be used as sources for new streams with an independent life cycle.

7.2 Concurrency

For an application that will consume events, Spring Cloud Stream exposes a concurrency setting that controls the size of a thread pool used for dispatching incoming messages. See the [Consumer properties](#) documentation for more information.

7.3 Partitioning

A common pattern in stream processing is to partition the data as it moves from one application to the next. Partitioning is a critical concept in stateful processing, for either performance or consistency reasons, to ensure that all related data is processed together. For example, in a time-windowed average calculation example, it is important that all measurements from any given sensor are processed by the same application instance. Alternatively, you may want to cache some data related to the incoming events so that it can be enriched without making a remote procedure call to retrieve the related data.

Spring Cloud Data Flow supports partitioning by configuring Spring Cloud Stream’s output and input bindings. Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion across different types of middleware. Partitioning can thus be used whether the broker itself is naturally partitioned (e.g., Kafka topics) or not (e.g., RabbitMQ). The following image shows how data could be partitioned into two buckets, such that each instance of the average processor application consumes a unique set of data.

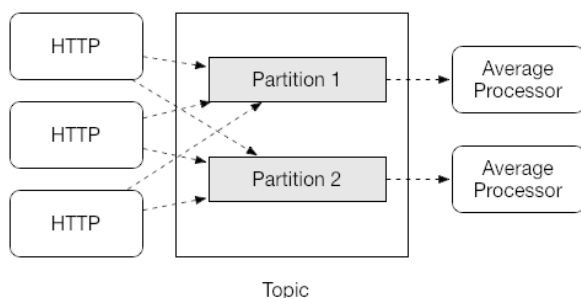


Figure 7.1. Spring Cloud Stream Partitioning

To use a simple partitioning strategy in Spring Cloud Data Flow, you only need set the instance count for each application in the stream and a `partitionKeyExpression` producer property when deploying the stream. The `partitionKeyExpression` identifies what part of the message will be used as the key to partition data in the underlying middleware. An `ingest` stream can be defined as `http | averageprocessor | cassandra` (Note that the Cassandra sink isn’t shown in the diagram above).

Suppose the payload being sent to the http source was in JSON format and had a field called `sensorId`. Deploying the stream with the shell command `stream deploy ingest --propertiesFile ingestStream.properties` where the contents of the file `ingestStream.properties` are

```
deployer.http.count=3
deployer.averageprocessor.count=2
app.http.producer.partitionKeyExpression=payload.sensorId
```

will deploy the stream such that all the input and output destinations are configured for data to flow through the applications but also ensure that a unique set of data is always delivered to each `averageprocessor` instance. In this case the default algorithm is to evaluate `payload.sensorId % partitionCount` where the `partitionCount` is the application count in the case of RabbitMQ and the partition count of the topic in the case of Kafka.

Please refer to [the section called “Passing stream partition properties during stream deployment”](#) for additional strategies to partition streams during deployment and how they map onto the underlying [Spring Cloud Stream Partitioning properties](#).

Also note, that you can’t currently scale partitioned streams. Read the section [Section 11.3, “Scaling at runtime”](#) for more information.

7.4 Message Delivery Guarantees

Streams are composed of applications that use the Spring Cloud Stream library as the basis for communicating with the underlying messaging middleware product. Spring Cloud Stream also provides an opinionated configuration of middleware from several vendors, in particular providing [persistent publish-subscribe semantics](#).

The [Binder abstraction](#) in Spring Cloud Stream is what connects the application to the middleware. There are several configuration properties of the binder that are portable across all binder implementations and some that are specific to the middleware.

For consumer applications there is a retry policy for exceptions generated during message handling. The retry policy is configured using the [common consumer properties](#) `maxAttempts`, `backOffInitialInterval`, `backOffMaxInterval`, and `backOffMultiplier`. The default values of these properties will retry the callback method invocation 3 times and wait one second for the first retry. A backoff multiplier of 2 is used for the second and third attempts.

When the number of retry attempts has exceeded the `maxAttempts` value, the exception and the failed message will become the payload of a message and be sent to the application’s error channel. By default, the default message handler for this error channel logs the message. You can change the default behavior in your application by creating your own message handler that subscribes to the error channel.

Spring Cloud Stream also supports a configuration option for both Kafka and RabbitMQ binder implementations that will send the failed message and stack trace to a dead letter queue. The dead letter queue is a destination and its nature depends on the messaging middleware (e.g in the case of Kafka it is a dedicated topic). To enable this for RabbitMQ set the [consumer properties](#) `republishToDlq` and `autoBindDlq` and the [producer property](#) `autoBindDlq` to true when deploying the stream. To always apply these producer and consumer properties when deploying streams, configure them as [common application properties](#) when starting the Data Flow server.

Additional messaging delivery guarantees are those provided by the underlying messaging middleware that is chosen for the application for both producing and consuming applications. Refer to the Kafka

[Consumer](#) and [Producer](#) and Rabbit [Consumer](#) and [Producer](#) documentation for more details. You will find extensive declarative support for all the native QOS options.

8. Analytics

Spring Cloud Data Flow is aware of certain Sink applications that will write counter data to Redis and provides an REST endpoint to read counter data. The types of counters supported are

- [Counter](#) - Counts the number of messages it receives, optionally storing counts in a separate store such as redis.
- [Field Value Counter](#) - Counts occurrences of unique values for a named field in a message payload
- [Aggregate Counter](#) - Stores total counts but also retains the total count values for each minute, hour day and month.

It is important to note that the timestamp that is used in the aggregate counter can come from a field in the message itself so that out of order messages are properly accounted.

9. Task Applications

The Spring Cloud Task programming model provides:

- Persistence of the Task's lifecycle events and exit code status.
- Lifecycle hooks to execute code before or after a task execution.
- Emit task events to a stream (as a source) during the task lifecycle.
- Integration with Spring Batch Jobs.

10. Data Flow Server

10.1 Endpoints

The Data Flow Server uses an embedded servlet container and exposes REST endpoints for creating, deploying, undeploying, and destroying streams and tasks, querying runtime state, analytics, and the like. The Data Flow Server is implemented using Spring's MVC framework and the [Spring HATEOAS](#) library to create REST representations that follow the HATEOAS principle.

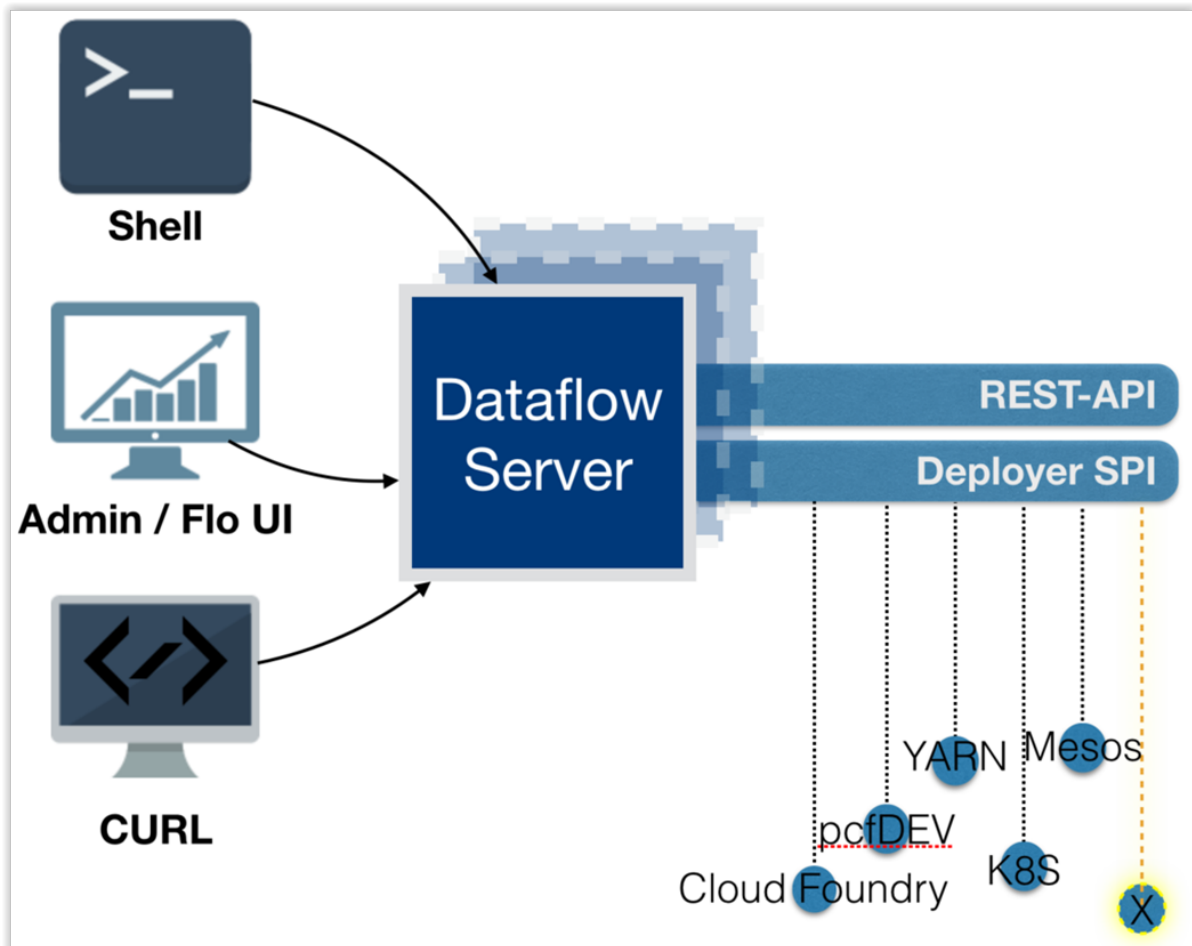


Figure 10.1. The Spring Cloud Data Flow Server

10.2 Customization

Each Data Flow Server executable jar targets a single runtime by delegating to the implementation of the deployer Service Provider Interface found on the classpath.

We provide a Data Flow Server executable jar that targets a single runtime. The Data Flow server delegates to the implementation of the deployer Service Provider Interface found on the classpath. In the current version, there are no endpoints specific to a target runtime, but may be available in future releases as a convenience to access runtime specific features

While we provide a server executable for each of the target runtimes you can also create your own customized server application using Spring Initializr. This lets you add or remove functionality relative to the executable jar we provide. For example, adding additional security implementations, custom

endpoints, or removing Task or Analytics REST endpoints. You can also enable or disable some features through the use of feature toggles.

10.3 Security

The Data Flow Server executable jars support basic http, LDAP(S), File-based, and OAuth 2.0 authentication to access its endpoints. Refer to the [security section](#) for more information.

Authorization via groups is planned for a future release.

11. Runtime

11.1 Fault Tolerance

The target runtimes supported by Data Flow all have the ability to restart a long lived application should it fail. Spring Cloud Data Flow sets up whatever health probe is required by the runtime environment when deploying the application.

The collective state of all applications that comprise the stream is used to determine the state of the stream. If an application fails, the state of the stream will change from 'deployed' to 'partial'.

11.2 Resource Management

Each target runtime lets you control the amount of memory, disk and CPU that is allocated to each application. These are passed as properties in the deployment manifest using key names that are unique to each runtime. Refer to the each platforms server documentation for more information.

11.3 Scaling at runtime

When deploying a stream, you can set the instance count for each individual application that comprises the stream. Once the stream is deployed, each target runtime lets you control the target number of instances for each individual application. Using the APIs, UIs, or command line tools for each runtime, you can scale up or down the number of instances as required. Future work will provide a portable command in the Data Flow Server to perform this operation.

Currently, this is not supported with the Kafka binder (based on the 0.8 simple consumer at the time of the release), as well as partitioned streams, for which the suggested workaround is redeploying the stream with an updated number of instances. Both cases require a static consumer set up based on information about the total instance count and current instance index, a limitation intended to be addressed in future releases. For example, Kafka 0.9 and higher provides good infrastructure for scaling applications dynamically and will be available as an alternative to the current Kafka 0.8 based binder in the near future. One specific concern regarding scaling partitioned streams is the handling of local state, which is typically reshuffled as the number of instances is changed. This is also intended to be addressed in the future versions, by providing first class support for local state management.

11.4 Application Versioning

Application versioning, that is upgrading or downgrading an application from one version to another, is not directly supported by Spring Cloud Data Flow. You must rely on specific target runtime features to perform these operational tasks.

The roadmap for Spring Cloud Data Flow will deploy applications that are compatible with Spinnaker to manage the complete application lifecycle. This also includes automated canary analysis backed by application metrics. Portable commands in the Data Flow server to trigger pipelines in Spinnaker are also planned.

Part III. Getting started

12. 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.

12.1 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.

12.2 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).

12.3 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.

12.4 Download the Spring Cloud Data Flow Server and Shell apps

```
wget http://repo.spring.io/release/org/springframework/cloud/spring-cloud-dataflow-server-cloudfoundry/1.2.1.RELEASE/spring-cloud-dataflow-server-cloudfoundry-1.2.1.RELEASE.jar
wget http://repo.spring.io/release/org/springframework/cloud/spring-cloud-dataflow-shell/1.2.1.RELEASE/spring-cloud-dataflow-shell-1.2.1.RELEASE.jar
```

12.5 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 -b java_buildpack -m 2G -k 2G --no-start -p spring-cloud-dataflow-server-cloudfoundry-1.2.1.RELEASE.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 application 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 [Section 13.1, "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, set it as the remote repository as listed below.

```
cf set-env dataflow-server SPRING_APPLICATION_JSON '{"maven": { "remote-repositories": { "repo1": { "url": "https://repo.spring.io/libs-release" } } } }'
```

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

**Note**

If you need to configure multiple Maven repositories, a proxy, or authorization for a private repository, see [Maven Configuration](#).

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.

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 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-1.2.1.RELEASE.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 Toggles](#) for how to disable task support in Data Flow.

Sample Manifest Template

As an alternative to setting environment variables via `cf set-env` command, you can curate all the relevant env-var's in `manifest.yml` file and use `cf push` command to provision the server.

Following is a sample template to provision the server on PCFDev.

```
---
applications:
- name: data-flow-server
  host: data-flow-server
  memory: 2G
  disk_quota: 2G
  instances: 1
  path: {PATH TO SERVER UBER-JAR}
  env:
    SPRING_APPLICATION_NAME: data-flow-server
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL: https://api.local.pcfdev.io
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG: pcfdev-org
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE: pcfdev-space
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN: local.pcfdev.io
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME: admin
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD: admin
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES: rabbit
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES: mysql
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION: true
    SPRING_APPLICATION_JSON: {"maven": { "remote-repositories": { "repol": { "url": "https://
repo.spring.io/libs-release" } } } }
  services:
  - mysql
```

Once you're ready with the relevant properties in this file, you can issue `cf push` command from the directory where this file is stored.

12.6 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-1.2.1.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-SR1-stream-applications-rabbit-maven
```



A Note about application URIs

While Spring Cloud Data Flow for Cloud Foundry leverages the core Data Flow project, and as such theoretically supports registering apps using any scheme, the use of `file://` URIs does not really make sense on Cloud Foundry. Indeed, the local filesystem of the Data Flow server is ephemeral and chances are that you don't want to manually upload your apps there.

When deploying apps using Data Flow for Cloud Foundry, a typical choice is to use `maven://` coordinates, or maybe `http://` URIs.

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 2017 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.

13. 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. For 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.

13.1 Authentication and Cloud Foundry

Spring Cloud Data Flow can either integrate with *Pivotal Single Sign-On Service* (E.g. on PWS) or *Cloud Foundry User Account and Authentication (UAA) Server*.

Pivotal Single Sign-On Service

When deploying Spring Cloud Data Flow to Cloud Foundry you can simply bind the application to the *Pivotal Single Sign-On Service*. By doing so, Spring Cloud Data Flow takes advantage of the [Spring Cloud Single Sign-On Connector](#), which provides Cloud Foundry specific auto-configuration support for OAuth 2.0.

Simply bind the *Pivotal Single Sign-On Service* to your Data Flow Server app and Single Sign-On (SSO) via OAuth2 will be enabled by default.

Authorization is similarly support as for non-Cloud Foundry security scenarios. Please refer to the security section from the core Data Flow [reference guide](#).

As the provisioning of roles can vary widely across environments, we assign by default all Spring Cloud Data Flow roles to users.

This can be customized by providing your own [AuthoritiesExtractor](#).

One possible approach to set the custom `AuthoritiesExtractor` on the `UserInfoTokenServices` could be this:

```
public class MyUserInfoTokenServicesPostProcessor
    implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) {
        if (bean instanceof UserInfoTokenServices) {
            final UserInfoTokenServices userInfoTokenServices = (UserInfoTokenServices) bean;
            userInfoTokenServices.setAuthoritiesExtractor(ctx.getBean(AuthoritiesExtractor.class));
        }
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) {
        return bean;
    }
}
```

And you simply declare it in your configuration class:

```
@Bean
public BeanPostProcessor myUserInfoTokenServicesPostProcessor() {
    BeanPostProcessor postProcessor = new MyUserInfoTokenServicesPostProcessor();
}
```

```
return postProcessor;
}
```

Cloud Foundry UAA

The availability of this option depends on the used Cloud Foundry environment. In order to provide UAA integration, you have to manually provide the necessary OAuth2 configuration properties, for instance via the `SPRING_APPLICATION_JSON` property.

```
{
  "security.oauth2.client.client-id": "scdf",
  "security.oauth2.client.client-secret": "scdf-secret",
  "security.oauth2.client.access-token-uri": "https://login.cf.myhost.com/oauth/token",
  "security.oauth2.client.user-authorization-uri": "https://login.cf.myhost.com/oauth/authorize",
  "security.oauth2.resource.user-info-uri": "https://login.cf.myhost.com/userinfo"
}
```

By default, the property `spring.cloud.dataflow.security.cf-use-uaa` is set to `true`. This property will activate a special

[AuthoritiesExtractor](#) **CloudFoundryDataflowAuthoritiesExtractor**.

If CloudFoundry UAA is not used, then make sure to set `spring.cloud.dataflow.security.cf-use-uaa` to `false`.

Under the covers this *AuthoritiesExtractor* will call out to the [Cloud Foundry Apps API](#) and ensure that users are in fact *Space Developers*.

If the authenticated user is verified as *Space Developer*, all roles will be assigned, otherwise no roles whatsoever will be assigned. In that case you may see the following Dashboard screen:

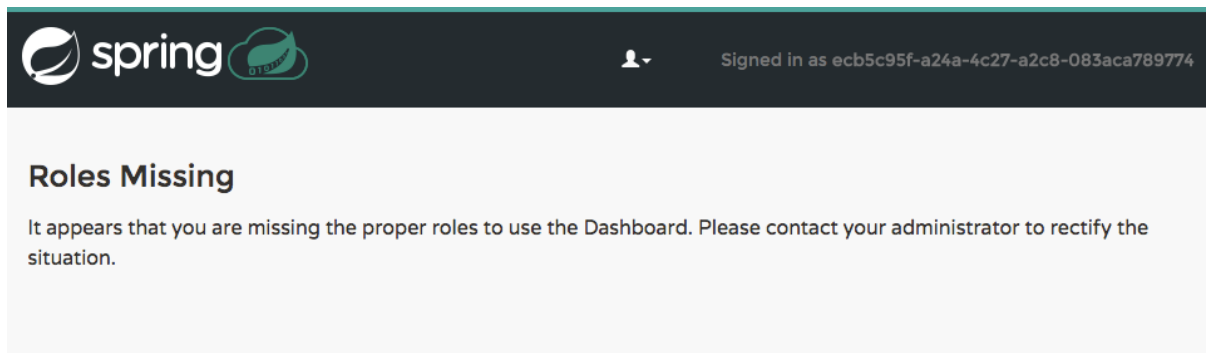


Figure 13.1. Accessing the Data Flow Dashboard without Roles

14. 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
```

14.1 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:

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

dataflow:>stream deploy foo --properties "deployer.http.cloudfoundry.domain=mydomain.com,
                                         deployer.http.cloudfoundry.host=myhost,
                                         deployer.http.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.

15. Deploying Docker Applications

Starting with version 1.2, it is possible to register and deploy Docker based apps as part of streams and tasks using Data Flow for Cloud Foundry.

If you are using Spring Boot and RabbitMQ based Docker images you can provide a common deployment property to facilitate the apps binding to the RabbitMQ service. Assuming your RabbitMQ service is named `rabbit` you can provide the following:

```
cf set-env dataflow-server SPRING_APPLICATION_JSON
'{"spring.cloud.dataflow.applicationProperties.stream.spring.rabbitmq.addresses":
"${vcap.services.rabbit.credentials.protocols.amqp.uris}"}'
```

For Spring Cloud Task apps, something similar to the following could be used, if using a database service instance named `mysql`:

```
cf set-env SPRING_DATASOURCE_URL '${vcap.services.mysql.credentials.jdbcUrl}'
cf set-env SPRING_DATASOURCE_USERNAME '${vcap.services.mysql.credentials.username}'
cf set-env SPRING_DATASOURCE_PASSWORD '${vcap.services.mysql.credentials.password}'
cf set-env SPRING_DATASOURCE_DRIVER_CLASS_NAME 'org.mariadb.jdbc.Driver'
```

For non-Java or non-Boot apps, your Docker app would have to parse the `VCAP_SERVICES` variable in order to bind to any available services.



Passing application properties

When using non-boot apps, chances are that you want the application properties passed to your app using traditional environment variables, as opposed to using the special `SPRING_APPLICATION_JSON` variable. To achieve this, set the following variables for streams and tasks, respectively:

```
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_USE_SPRING_APPLICATION_JSON=false
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_USE_SPRING_APPLICATION_JSON=false
```

16. 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

# Timeout to use, in milliseconds, when querying the Cloud Foundry API to compute app status.
# (for setting env var use SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_STATUS_TIMEOUT
# and SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_STATUS_TIMEOUT)
spring.cloud.deployer.cloudfoundry.stream.statusTimeout=5000
spring.cloud.deployer.cloudfoundry.task.statusTimeout=5000
```

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 by using the `deployer.<app-name>` shortcut. For example

```
>stream create --name ticktock --definition "time | log"
>stream deploy --name ticktock --properties "deployer.time.memory=2g"
```

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

16.1 Understanding what's going on

If you want to get better insights into what is happening when your streams and tasks are being deployed, you may want to turn on the following features:

- Reactor "stacktraces", showing which operators were involved before an error occurred. This is helpful as the deployer relies on project reactor and regular stacktraces may not always allow understanding the flow before an error happened. Note that this comes with a performance penalty, so is disabled by default.

```
spring.cloud.dataflow.server.cloudfoundry.debugReactor = true
```

- Deployer and Cloud Foundry client library request/response logs. This allows seeing detailed conversation between the Data Flow server and the Cloud Foundry Cloud Controller.

```
logging.level.cloudfoundry-client = DEBUG
```

16.2 Using Spring Cloud Config Server

Spring Cloud Config Server can be used to centralize configuration properties for Spring Boot applications. Likewise, both Spring Cloud Data Flow and the applications orchestrated using Spring Cloud Data Flow can be integrated with config-server to leverage the same capabilities.

Stream, Task, and Spring Cloud Config Server

Similar to Spring Cloud Data Flow server, it is also possible to configure both the stream and task applications to resolve the centralized properties from config-server. Setting the property `spring.cloud.config.uri` for the deployed applications is a common way to bind to the Config Server. See the [Spring Cloud Config Client](#) reference guide for more information. Since this property is likely to be used across all applications deployed by the Data Flow server, the Data Flow Server's property `spring.cloud.dataflow.applicationProperties.stream` for stream apps and `spring.cloud.dataflow.applicationProperties.task` for task apps can be used to pass the `uri` of the Config Server to each deployed stream or task application. Refer to the section on Common application properties for more information.

If you're using applications from the [App Starters project](#), note that these applications already embed the `spring-cloud-services-starter-config-client` dependency. If you're building your application from scratch and want to add the client side support for config server, simply add a reference dependency reference to the config server client library. A maven example snippet follows:

```
...
<dependency>
  <groupId>io.pivotal.spring.cloud</groupId>
  <artifactId>spring-cloud-services-starter-config-client</artifactId>
  <version>CONFIG_CLIENT_VERSION</version>
</dependency>
...
```

Where, `CONFIG_CLIENT_VERSION` can be the latest release of [Spring Cloud Config Server](#) client for Pivotal Cloud Foundry.



Note

You will observe a `WARN` logging message if the application that uses this library can not connect to the config server when the application starts and whenever the `/health` endpoint is accessed. You can disable the client library if you know that you are not using config server functionality by setting the environment variable `SPRING_CLOUD_CONFIG_ENABLED=false`. Another, more drastic option, is to disable the platform health check with the environment variable `SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_HEALTH_CHECK=none`

Sample Manifest Template

Following `manifest.yml` template includes the required env-var's for the Spring Cloud Data Flow server and deployed apps/tasks to successfully run on Cloud Foundry and automatically resolve centralized properties from `my-config-server` at the runtime.

```
---
applications:
- name: data-flow-server
  host: data-flow-server
  memory: 2G
  disk_quota: 2G
  instances: 1
  path: {PATH TO SERVER UBER-JAR}
  env:
    SPRING_APPLICATION_NAME: data-flow-server
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL: https://api.local.pcfdev.io
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG: pcfdev-org
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE: pcfdev-space
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN: local.pcfdev.io
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME: admin
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD: admin
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES: rabbit,my-config-server
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES: mysql,my-config-server
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION: true
    SPRING_APPLICATION_JSON: { "maven": { "remote-repositories": { "repol": { "url": "https://
repo.spring.io/libs-release" } } } }
  services:
  - mysql
  - my-config-server
```

Where, `my-config-server` is the name of the Spring Cloud Config Service instance running on Cloud Foundry. By binding the service to both Spring Cloud Data Flow server as well as all the Spring Cloud Stream and Spring Cloud Task applications respectively, we can now resolve centralized properties backed by this service.

Self-signed SSL Certificate and Spring Cloud Config Server

Often, in a development environment, we may not have a valid certificate to enable SSL communication between clients and the backend services. However, the config-server for Pivotal Cloud Foundry uses HTTPS for all client-to-service communication, so it is necessary to add a self-signed SSL certificate in environments with no valid certificates.

Using the same `manifest.yml` template listed in the previous section, for the server, we can provide the self-signed SSL certificate via: `TRUST_CERTS: <API_ENDPOINT>`.

However, the deployed applications *also* require `TRUST_CERTS` as a *flat* env-var (as opposed to being wrapped inside `SPRING_APPLICATION_JSON`), so

we will have to instruct the server with yet another set of tokens
 SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_USE_SPRING_APPLICATION_JSON:
 false and
 SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_USE_SPRING_APPLICATION_JSON: false
 for stream and task applications respectively. With this setup, the applications will receive their
 application properties as regular environment variables

Let's review the updated `manifest.yml` with the required changes. Both the Data Flow server
 and deployed applications would get their config from the `my-config-server` Cloud Config server
 (deployed as a Cloud Foundry service)

```
---
applications:
- name: test-server
  host: test-server
  memory: 1G
  disk_quota: 1G
  instances: 1
  path: spring-cloud-dataflow-server-cloudfoundry-VERSION.jar
  env:
    SPRING_APPLICATION_NAME: test-server
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL: <URL>
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG: <ORG>
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE: <SPACE>
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN: <DOMAIN>
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME: <USER>
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD: <PASSWORD>
    MAVEN_REMOTE_REPOSITORIES_REPO1_URL: https://repo.spring.io/libs-release
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES: my-config-server #this is so all stream
applications bind to my-config-server
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES: config-server #this for so all task
applications bind to my-config-server
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_USE_SPRING_APPLICATION_JSON: false #this is for all the
stream applications
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_USE_SPRING_APPLICATION_JSON: false #this is for all the task
applications
    TRUST_CERTS: <API_ENDPOINT> #this is for the server
    spring.cloud.dataflow.applicationProperties.stream.TRUST_CERTS: <API_ENDPOINT> #this propagates to
all streams
    spring.cloud.dataflow.applicationProperties.task.TRUST_CERTS: <API_ENDPOINT> #this propagates to
all tasks
  services:
- mysql
- my-config-server #this is for the server
```


17. 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 httpjdbc --definition "http | jdbc"
dataflow:>stream deploy --name httpjdbc --properties
"deployer.jdbc.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: `deployer.jdbc.cloudfoundry.services=mysqlService,someService`).

18. A Note About User Provided Services

In addition to marketplace services, Cloud Foundry supports [User Provided Services](#) (UPS). 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).

Let's review an example of extracting and supplying the connection credentials from an UPS.

- A sample UPS setup for Apache Kafka.

```
cf create-user-provided-service kafkacups -p '{"brokers":"HOST:PORT","zkNodes":"HOST:PORT"}'
```

- The UPS credentials will be wrapped within VCAP_SERVICES and it can be supplied directly in the stream definition like the following.

```
stream create fooz --definition "time | log"
stream deploy fooz --properties "app.time.spring.cloud.stream.kafka.binder.brokers=
${vcap.services.kafkacups.credentials.brokers},app.time.spring.cloud.stream.kafka.binder.zkNodes=
${vcap.services.kafkacups.credentials.zkNodes},app.log.spring.cloud.stream.kafka.binder.brokers=
${vcap.services.kafkacups.credentials.brokers},app.log.spring.cloud.stream.kafka.binder.zkNodes=
${vcap.services.kafkacups.credentials.zkNodes}"
```

19. 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 -b java_buildpack -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.group":"foo.time"}
```

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.

20. 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.

20.1 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.

20.2 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
```

20.3 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
```

Part IV. Server Configuration

In this section you will learn how to configure Spring Cloud Data Flow server's features such as the relational database to use and security.

21. Feature Toggles

Data Flow server offers specific set of features that can be enabled/disabled when launching. These features include all the lifecycle operations, REST endpoints (server, client implementations including Shell and the UI) for:

1. Streams

2. Tasks

3. Analytics

One can enable, disable these features by setting the following boolean properties when launching the Data Flow server:

- `spring.cloud.dataflow.features.streams-enabled`
- `spring.cloud.dataflow.features.tasks-enabled`
- `spring.cloud.dataflow.features.analytics-enabled`

By default, all the features are enabled. Note: Since analytics feature is enabled by default, the Data Flow server is expected to have a valid Redis store available as analytic repository as we provide a default implementation of analytics based on Redis. This also means that the Data Flow server's health depends on the redis store availability as well. If you do not want to enabled HTTP endpoints to read analytics data written to Redis, then disable the analytics feature using the property mentioned above.

The REST endpoint `/features` provides information on the features enabled/disabled.

22. Database Configuration

Spring Cloud Data Flow provides schemas for H2, HSQLDB, MySQL, Oracle, Postgresql, DB2 and SqlServer that will be automatically created when the server starts.

The JDBC drivers for **MySQL** (via MariaDB driver), **HSQLDB**, **PostgreSQL** along with embedded **H2** are available out of the box. If you are using any other database, then the corresponding JDBC driver jar needs to be on the classpath of the server.

The database properties can be passed as command-line arguments to the Data Flow Server.

For instance, If you are using **MySQL**:

```
java -jar spring-cloud-dataflow-server-local/target/spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar \
  --spring.datasource.url=jdbc:mysql:<db-info> \
  --spring.datasource.username=<user> \
  --spring.datasource.password=<password> \
  --spring.datasource.driver-class-name=org.mariadb.jdbc.Driver &
```

For **PostgreSQL**:

```
java -jar spring-cloud-dataflow-server-local/target/spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar \
  --spring.datasource.url=jdbc:postgresql:<db-info> \
  --spring.datasource.username=<user> \
  --spring.datasource.password=<password> \
  --spring.datasource.driver-class-name=org.postgresql.Driver &
```

For **HSQLDB**:

```
java -jar spring-cloud-dataflow-server-local/target/spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar \
  --spring.datasource.url=jdbc:hsqldb:<db-info> \
  --spring.datasource.username=SA \
  --spring.datasource.driver-class-name=org.hsqldb.jdbc.JDBCdriver &
```



Note

There is a schema update to the Spring Cloud Data Flow datastore when upgrading from version 1.0.x to 1.1.x. Migration scripts for specific database types can be found [here](#).



Note

If you wish to use an external H2 database instance instead of the one embedded with Spring Cloud Data Flow set the `spring.dataflow.embedded.database.enabled` property to false. If `spring.dataflow.embedded.database.enabled` is set to false or a database other than h2 is specified as the datasource the embedded database will not start.

23. 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 using either:

- [OAuth 2.0](#)
- Traditional Authentication (Incl. Basic Authentication)

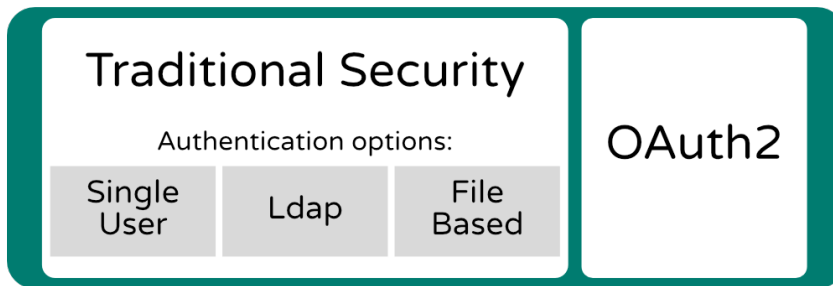


Figure 23.1. Authentication Options

When choosing *traditional authentication*, the Spring Cloud Data Flow server will be the main authentication point, using Spring Security under the covers. When selecting this option, users then need to further define their preferred authentication mechanism aka select the desired authentication backing store:

- [Single User Authentication](#)
- [Ldap Authentication](#)
- [File-based authentication](#)

When choosing between *traditional authentication* or *OAuth2*, keep in mind that both options are mutually exclusive. Please refer to the sub-chapters below for a more detailed discussion.



Note

By default, the REST endpoints (administration, management and health), as well as the Dashboard UI do not require authenticated access.

23.1 Enabling HTTPS

By default, the dashboard, management, and health endpoints use HTTP as a transport. You can switch to HTTPS easily, by adding a certificate to your configuration in `application.yml`.

```

server:
  port: 8443
  ssl:
    key-alias: yourKeyAlias
    key-store: path/to/keystore
    key-store-password: yourKeyStorePassword
    key-password: yourKeyPassword
    trust-store: path/to/trust-store
    trust-store-password: yourTrustStorePassword
  
```

- ❶ As the default port is 9393, you may choose to change the port to a more common HTTPs-typical port.
- ❷ The alias (or name) under which the key is stored in the keystore.
- ❸ The path to the keystore file. Classpath resources may also be specified, by using the classpath prefix: `classpath:path/to/keystore`
- ❹ The password of the keystore.
- ❺ The password of the key.
- ❻ The path to the truststore file. Classpath resources may also be specified, by using the classpath prefix: `classpath:path/to/trust-store`
- ❼ The password of the trust store.



Note

If HTTPS is enabled, it will completely replace HTTP as the protocol over which the REST endpoints and the Data Flow Dashboard interact. Plain HTTP requests will fail - therefore, make sure that you configure your Shell accordingly.

Using Self-Signed Certificates

For testing purposes or during development it might be convenient to create self-signed certificates. To get started, execute the following command to create a certificate:

```
$ keytool -genkey -alias dataflow -keyalg RSA -keystore dataflow.keystore \
  -validity 3650 -storetype JKS \
  -dname "CN=localhost, OU=Spring, O=Pivotal, L=Kailua-Kona, ST=HI, C=US" ❶
  -keypass dataflow -storepass dataflow
```

- ❶ CN is the only important parameter here. It should match the domain you are trying to access, e.g. localhost.

Then add the following to your `application.yml` file:

```
server:
  port: 8443
  ssl:
    enabled: true
    key-alias: dataflow
    key-store: "/your/path/to/dataflow.keystore"
    key-store-type: jks
    key-store-password: dataflow
    key-password: dataflow
```

This is all that's needed for the Data Flow Server. Once you start the server, you should be able to access it via <https://localhost:8443/>. As this is a self-signed certificate, you will hit a warning in your browser, that you need to ignore.

Self-Signed Certificates and the Shell

By default self-signed certificates are an issue for the Shell and additional steps are necessary to make the Shell work with self-signed certificates. Two options are available:

1. Add the self-signed certificate to the JVM truststore
2. Skip certificate validation

Add the self-signed certificate to the JVM truststore

In order to use the JVM truststore option, we need to export the previously created certificate from the keystore:

```
$ keytool -export -alias dataflow -keystore dataflow.keystore -file dataflow_cert -storepass dataflow
```

Next, we need to create a truststore which the Shell will use:

```
$ keytool -importcert -keystore dataflow.truststore -alias dataflow -storepass dataflow -file dataflow_cert -noprompt
```

Now, you are ready to launch the Data Flow Shell using the following JVM arguments:

```
$ java -Djavax.net.ssl.trustStorePassword=dataflow \
-Djavax.net.ssl.trustStore=/path/to/dataflow.truststore \
-Djavax.net.ssl.trustStoreType=jks \
-jar spring-cloud-dataflow-shell-1.2.1.RELEASE.jar
```



Tip

In case you run into trouble establishing a connection via SSL, you can enable additional logging by using and setting the `javax.net.debug` JVM argument to `ssl`.

Don't forget to target the Data Flow Server with:

```
dataflow:> dataflow config server https://localhost:8443/
```

Skip Certificate Validation

Alternatively, you can also bypass the certification validation by providing the optional command-line parameter `--dataflow.skip-ssl-validation=true`.

Using this command-line parameter, the shell will accept any (self-signed) SSL certificate.



Warning

If possible you should avoid using this option. Disabling the trust manager defeats the purpose of SSL and makes you vulnerable to man-in-the-middle attacks.

23.2 Traditional Authentication

When using *traditional authentication* Spring Cloud Data Flow will be the sole authentication provider. In that case Dataflow REST API users would use [Basic Authentication](#) to access the endpoints.

When using that option, users have a choice of 3 backing stores for authentication details:

- **Single User Authentication** using Spring Boot properties
- **File-based authentication** for multiple users using a Yaml file
- **Ldap Authentication**

Single User Authentication

This is the simplest option and mimics the behavior of the default Spring Boot user user-experience. It can be enabled by adding the following to `application.yml` or via environment variables:

```
security:
  basic:
    enabled: true
    realm: Spring Cloud Data Flow
```

❶
❷

- ❶ Enables basic authentication. Must be set to true for security to be enabled.
- ❷ (Optional) The realm for Basic authentication. Will default to *Spring* if not explicitly set.



Note

Current versions of Chrome do not display the *realm*. Please see the following [Chromium issue ticket](#) for more information.

In this use-case, the underlying Spring Boot will auto-create a user called *user* with an auto-generated password which will be printed out to the console upon startup.

With this setup, the generated user will have all main roles assigned:

- VIEW
- CREATE
- MANAGE

```
2017-04-20 16:02:57.935 INFO 38883 --- [main] o.s.b.a.s.SimpleJobServiceFactoryBean : No database type set, using meta data indicating: H2
2017-04-20 16:03:00.312 INFO 38883 --- [main] c.d.s.c.s.BasicAuthSecurityConfiguration :
Using default security password: bdee2af2-1629-40d6-a839-d571b94ae54 with roles 'VIEW,CREATE,MANAGE'
2017-04-20 16:03:00.595 INFO 38883 --- [main] c.d.s.c.s.BasicAuthSecurityConfiguration : Authorization 'GET' | 'hasRole('ROLE_VIEW')' | '/metrics/streams'
2017-04-20 16:03:00.598 INFO 38883 --- [main] c.d.s.c.s.BasicAuthSecurityConfiguration : Authorization 'GET' | 'hasRole('ROLE_VIEW')' | '/about'
2017-04-20 16:03:00.599 INFO 38883 --- [main] c.d.s.c.s.BasicAuthSecurityConfiguration : Authorization 'GET' | 'hasRole('ROLE_VIEW')' | '/metrics/**'
2017-04-20 16:03:00.599 INFO 38883 --- [main] c.d.s.c.s.BasicAuthSecurityConfiguration : Authorization 'DELETE' | 'hasRole('ROLE_CREATE')' | '/metrics/**'
2017-04-20 16:03:00.599 INFO 38883 --- [main] c.d.s.c.s.BasicAuthSecurityConfiguration : Authorization 'GET' | 'hasRole('ROLE_MANAGE')' | '/management/**'
2017-04-20 16:03:00.599 INFO 38883 --- [main] c.d.s.c.s.BasicAuthSecurityConfiguration : Authorization 'GET' | 'hasRole('ROLE_VIEW')' | '/apps'
2017-04-20 16:03:00.600 INFO 38883 --- [main] c.d.s.c.s.BasicAuthSecurityConfiguration : Authorization 'GET' | 'hasRole('ROLE_VIEW')' | '/apps/**'
```

Figure 23.2. Default Spring Boot user credentials

You can customize the user yourself using the following properties:

```
security.user.name=user # Default user name.
security.user.password= # Password for the default user name. A random password is logged on startup by default.
security.user.role=VIEW, CREATE, MANAGE # Granted roles for the default user name.
```



Note

Please be aware of inherent issues of Basic Authentication and *logging out*, since the credentials are cached by the browser and simply browsing back to application pages will log you back in.

Of course, you can also pass in credentials using system properties, environment variables or command-line arguments as this is standard Spring Boot behavior. For instance in the following example, command-line arguments are used to specify the user credentials:

```
$ java -jar spring-cloud-dataflow-server-local-1.2.1.RELEASE.jar \
  --security.basic.enabled=true \
  --security.user.name=test \
  --security.user.password=pass \
  --security.user.role=VIEW
```

If you need to define more than one file-based user account, please take a look at [File based authentication](#).

File based authentication

By default Spring Boot allows you to only specify one single user. Spring Cloud Data Flow also supports the listing of more than one user in a configuration file, as described below. Each user must be assigned a password and one or more roles:

```
security:
  basic:
    enabled: true
    realm: Spring Cloud Data Flow
  spring:
    cloud:
      dataflow:
        security:
          authentication:
            file:
              enabled: true
              users:
                bob: bobspassword, ROLE_MANAGE
                alice: alicepwd, ROLE_VIEW, ROLE_CREATE
```

- ❶ Enables file based authentication
- ❷ This is a yaml map of username to password
- ❸ Each map value is made of a corresponding password and role(s), comma separated

LDAP Authentication

Spring Cloud Data Flow also supports authentication against an LDAP server (Lightweight Directory Access Protocol), providing support for the following 2 modes:

- Direct bind
- Search and bind

When the LDAP authentication option is activated, the default single user mode is turned off.

In *direct bind mode*, a pattern is defined for the user's distinguished name (DN), using a placeholder for the username. The authentication process derives the distinguished name of the user by replacing the placeholder and use it to authenticate a user against the LDAP server, along with the supplied password. You can set up LDAP direct bind as follows:

```
security:
  basic:
    enabled: true
    realm: Spring Cloud Data Flow
  spring:
    cloud:
      dataflow:
        security:
          authentication:
            ldap:
              enabled: true
              url: ldap://ldap.example.com:3309
              userDnPattern: uid={0},ou=people,dc=example,dc=com
```

- ❶ Enables LDAP authentication
- ❷ The URL for the LDAP server
- ❸ The distinguished name (DN) pattern for authenticating against the server

The *search and bind* mode involves connecting to an LDAP server, either anonymously or with a fixed account, and searching for the distinguished name of the authenticating user based on its username, and then using the resulting value and the supplied password for binding to the LDAP server. This option is configured as follows:

```
security:
  basic:
    enabled: true
    realm: Spring Cloud Data Flow
  spring:
    cloud:
      dataflow:
        security:
          authentication:
            ldap:
              enabled: true
              url: ldap://localhost:10389
              managerDn: uid=admin,ou=system
              managerPassword: secret
              userSearchBase: ou=otherpeople,dc=example,dc=com
              userSearchFilter: uid={0}
```

- ❶ Enables LDAP integration
- ❷ The URL of the LDAP server
- ❸ A DN for to authenticate to the LDAP server, if anonymous searches are not supported (optional, required together with next option)
- ❹ A password to authenticate to the LDAP server, if anonymous searches are not supported (optional, required together with previous option)
- ❺ The base for searching the DN of the authenticating user (serves to restrict the scope of the search)
- ❻ The search filter for the DN of the authenticating user



Tip

For more information, please also see the chapter [LDAP Authentication](#) of the Spring Security reference guide.

LDAP Transport Security

When connecting to an LDAP server, you typically (In the LDAP world) have 2 options in order to establish a connection to an LDAP server securely:

- LDAP over SSL (LDAPS)
- Start Transport Layer Security (Start TLS is defined in [RFC2830](#))

As of *Spring Cloud Data Flow 1.1.0* only LDAPS is supported out-of-the-box. When using official certificates no special configuration is necessary, in order to connect to an LDAP Server via LDAPS. Just change the url format to **ldaps**, e.g. `ldaps://localhost:636`.

In case of using self-signed certificates, the setup for your Spring Cloud Data Flow server becomes slightly more complex. The setup is very similar to [the section called “Using Self-Signed Certificates”](#) (Please read first) and Spring Cloud Data Flow needs to reference a *trustStore* in order to work with your self-signed certificates.



Important

While useful during development and testing, please never use self-signed certificates in production!

Ultimately you have to provide a set of system properties to reference the trustStore and its credentials when starting the server:

```
$ java -Djavax.net.ssl.trustStorePassword=dataflow \
-Djavax.net.ssl.trustStore=/path/to/dataflow.truststore \
-Djavax.net.ssl.trustStoreType=jks \
-jar spring-cloud-starter-dataflow-server-local-1.2.1.RELEASE.jar
```

As mentioned above, another option to connect to an LDAP server securely is via *Start TLS*. In the LDAP world, LDAPs is technically even considered deprecated in favor of Start TLS. However, this option is currently not supported out-of-the-box by Spring Cloud Data Flow.

Please follow the following [issue tracker ticket](#) to track its implementation. You may also want to look at the Spring LDAP reference documentation chapter on [Custom DirContext Authentication Processing](#) for further details.

Shell Authentication

When using traditional authentication with the Data Flow Shell, you typically provide a *username* and *password* using command-line arguments, e.g.:

```
$ java -jar target/spring-cloud-dataflow-shell-1.2.1.RELEASE.jar \
--dataflow.username=myuser \ ❶
--dataflow.password=mysecret ❷
```

- ❶ If authentication is enabled the username must be provided
- ❷ If the password is not provided, the shell will prompt for it

Alternatively, you can *target* a Data Flow server also from within the Shell:

```
server-unknown:>dataflow config server
--uri http://localhost:9393 \ ❶
--username myuser \ ❷
--password mysecret \ ❸
--skip-ssl-validation true \ ❹
```

- ❶ Optional, defaults to [localhost:9393](#)
- ❷ Mandatory if security is enabled
- ❸ If security is enabled, and the password is not provided, the user will be prompted for it
- ❹ Optional, ignores certificate errors (When using self-signed certificates). Use cautiously!

```
server-unknown:>dataflow config server --uri http://localhost:9393 --username myuser
Password:
Successfully targeted http://localhost:9393
dataflow:>
```

Figure 23.3. Target and Authenticate with the Data Flow Server from within the Shell

Customizing authorization

All of the above deals with authentication, *i.e.* how to assess the identity of the user. Irrespective of the option chosen, you can also customize **authorization** *i.e.* who can do what.

The default scheme uses three roles to protect the [REST endpoints](#) that Spring Cloud Data Flow exposes:

- **ROLE_VIEW** for anything that relates to retrieving state
- **ROLE_CREATE** for anything that involves creating, deleting or mutating the state of the system
- **ROLE_MANAGE** for boot management endpoints.

All of those defaults are specified in `dataflow-server-defaults.yml` which is part of the Spring Cloud Data Flow Core Module. Nonetheless, you can override those, if desired, e.g. in `application.yml`. The configuration takes the form of a YAML **list** (as some rules may have precedence over others) and so you'll need to copy/paste the whole list and tailor it to your needs (as there is no way to merge lists). Always refer to your version of `application.yml`, as the snippet reproduced below may be out-dated. The default rules are as such:

```
spring:
  cloud:
    dataflow:
      security:
        authorization:
          enabled: true
          rules:
            # Metrics

            - GET      /metrics/streams          => hasRole('ROLE_VIEW')

            # About

            - GET      /about                    => hasRole('ROLE_VIEW')

            # Metrics

            - GET      /metrics/**               => hasRole('ROLE_VIEW')
            - DELETE   /metrics/**               => hasRole('ROLE_CREATE')

            # Boot Endpoints

            - GET      /management/**            => hasRole('ROLE_MANAGE')

            # Apps

            - GET      /apps                     => hasRole('ROLE_VIEW')
            - GET      /apps/**                  => hasRole('ROLE_VIEW')
            - DELETE   /apps/**                  => hasRole('ROLE_CREATE')
            - POST     /apps                     => hasRole('ROLE_CREATE')
            - POST     /apps/**                  => hasRole('ROLE_CREATE')

            # Completions

            - GET      /completions/**           => hasRole('ROLE_CREATE')

            # Job Executions & Batch Job Execution Steps && Job Step Execution Progress

            - GET      /jobs/executions          => hasRole('ROLE_VIEW')
            - PUT      /jobs/executions/**       => hasRole('ROLE_CREATE')
            - GET      /jobs/executions/**       => hasRole('ROLE_VIEW')

            # Batch Job Instances

            - GET      /jobs/instances           => hasRole('ROLE_VIEW')
            - GET      /jobs/instances/*         => hasRole('ROLE_VIEW')

            # Running Applications

            - GET      /runtime/apps             => hasRole('ROLE_VIEW')
            - GET      /runtime/apps/**          => hasRole('ROLE_VIEW')

            # Stream Definitions
```

```

- GET      /streams/definitions           => hasRole('ROLE_VIEW')
- GET      /streams/definitions/*         => hasRole('ROLE_VIEW')
- GET      /streams/definitions/*/related => hasRole('ROLE_VIEW')
- POST     /streams/definitions           => hasRole('ROLE_CREATE')
- DELETE   /streams/definitions/*         => hasRole('ROLE_CREATE')
- DELETE   /streams/definitions           => hasRole('ROLE_CREATE')

# Stream Deployments

- DELETE   /streams/deployments/*         => hasRole('ROLE_CREATE')
- DELETE   /streams/deployments           => hasRole('ROLE_CREATE')
- POST     /streams/deployments/*         => hasRole('ROLE_CREATE')

# Task Definitions

- POST     /tasks/definitions              => hasRole('ROLE_CREATE')
- DELETE   /tasks/definitions/*            => hasRole('ROLE_CREATE')
- GET      /tasks/definitions              => hasRole('ROLE_VIEW')
- GET      /tasks/definitions/*            => hasRole('ROLE_VIEW')

# Task Executions

- GET      /tasks/executions               => hasRole('ROLE_VIEW')
- GET      /tasks/executions/*             => hasRole('ROLE_VIEW')
- POST     /tasks/executions               => hasRole('ROLE_CREATE')
- DELETE   /tasks/executions/*             => hasRole('ROLE_CREATE')

```

The format of each line is the following:

```
HTTP_METHOD URL_PATTERN '=>' SECURITY_ATTRIBUTE
```

where

- HTTP_METHOD is one http method, capital case
- URL_PATTERN is an Ant style URL pattern
- SECURITY_ATTRIBUTE is a SpEL expression (see docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#el-access)
- each of those separated by one or several blank characters (spaces, tabs, etc.)

Be mindful that the above is indeed a YAML list, not a map (thus the use of '-' dashes at the start of each line) that lives under the `spring.cloud.dataflow.security.authorization.rules` key.



Tip

In case you are solely interested in authentication but not authorization, for instance every user shall have access to all endpoints, then you can also set `spring.cloud.dataflow.security.authorization.enabled=false`.

If you are using basic security configuration by using security properties then it is important to set the roles for the users.

For instance,

```

java -jar spring-cloud-dataflow-server-local/target/spring-cloud-dataflow-server-local-1.2.1.RELEASE.jar \
  --security.basic.enabled=true \
  --security.user.name=test \
  --security.user.password=pass \
  --security.user.role=VIEW

```

Authorization - Shell and Dashboard Behavior

When authorization is enabled, the *Dashboard* and the *Shell* will be *role-aware*, meaning that depending on the assigned role(s), not all functionality may be visible.

For instance, Shell commands, for which the user does not have the necessary roles for, will be marked as unavailable.



Important

Currently, the Shell's `help` command will list commands that are unavailable. Please track the following issue: github.com/spring-projects/spring-shell/issues/115

Similarly for the *Dashboard*, the UI will not show pages, or page elements, for which the user is not authorized for.

Authorization with Ldap

When configuring Ldap for authentication, you can also specify the `group-role-attribute` in conjunction with `group-search-base` and `group-search-filter`.

The *group role attribute* contains the name of the role. If not specified, the `ROLE_MANAGE` role is populated by default.

For further information, please refer to [Configuring an LDAP Server](#) of the Spring Security reference guide.

23.3 OAuth 2.0

[OAuth 2.0](#) allows you to integrate Spring Cloud Data Flow into Single Sign On (SSO) environments. The following 3 OAuth2 Grant Types will be used:

- *Authorization Code* - Used for the GUI (Browser) integration. You will be redirected to your OAuth Service for authentication
- *Password* - Used by the shell (And the REST integration), so you can login using username and password
- *Client Credentials* - Retrieve an Access Token directly from your OAuth provider and pass it to the Dataflow server using the *Authorization* Http header.

The REST endpoints can be accessed using 2 ways:

- Basic Authentication which will use the *Password Grant Type* under the covers to authenticate with your OAuth2 service
- Access Token which will use the *Client Credentials Grant Type* under the covers



Note

When authentication is set up, it is strongly recommended to enable HTTPS as well, especially in production environments.

You can turn on OAuth2 authentication by adding the following to `application.yml` or via environment variables:

```
security:
  oauth2:
    client:
      client-id: myclient
      client-secret: mysecret
      access-token-uri: http://127.0.0.1:9999/oauth/token
      user-authorization-uri: http://127.0.0.1:9999/oauth/authorize
    resource:
      user-info-uri: http://127.0.0.1:9999/me
```

- ❶ Providing the Client Id in the OAuth Configuration Section will activate OAuth2 security

You can verify that basic authentication is working properly using *curl*:

```
$ curl -u myusername:mypassword http://localhost:9393/
```

As a result you should see a list of available REST endpoints.

Besides Basic Authentication, you can also provide an Access Token in order to access the REST Api. In order to make that happen, you would retrieve an OAuth2 Access Token from your OAuth2 provider first, and then pass that Access Token to the REST Api using the **Authorization** Http header:

```
$ curl -H "Authorization: Bearer <ACCESS_TOKEN>" http://localhost:9393/
```

OAuth REST Endpoint Authorization

The OAuth2 authentication option uses the same authorization rules as used by the [Traditional Authentication](#) option.



Tip

The authorization rules are defined in `dataflow-server-defaults.yml` (Part of the Spring Cloud Data Flow Core Module). Please see the chapter on [customizing authorization](#) for more details.

Due to fact that the determination of security roles is very environment-specific, *Spring Cloud Data Flow* will by default assign all roles to authenticated OAuth2 users using the `DefaultDataflowAuthoritiesExtractor` class.

You can customize that behavior by providing your own Spring bean definition that extends Spring Security OAuth's `AuthoritiesExtractor` interface. In that case, the custom bean definition will take precedence over the default one provided by *Spring Cloud Data Flow*

OAuth Authentication using the Spring Cloud Data Flow Shell

If your OAuth2 provider supports the *Password Grant Type* you can start the *Data Flow Shell* with:

```
$ java -jar spring-cloud-dataflow-shell-1.2.1.RELEASE.jar \
--dataflow.uri=http://localhost:9393 \
--dataflow.username=my_username --dataflow.password=my_password
```



Note

Keep in mind that when authentication for Spring Cloud Data Flow is enabled, the underlying OAuth2 provider **must** support the *Password* OAuth2 Grant Type, if you want to use the Shell.

From within the Data Flow Shell you can also provide credentials using:

```
dataflow config server --uri http://localhost:9393 --username my_username --password my_password
```

Once successfully targeted, you should see the following output:

```
dataflow:>dataflow config info
dataflow config info

#####
#Credentials#[username='my_username, password=****']#
#####
#Result      #
#Target      #http://localhost:9393
#####
```

OAuth2 Authentication Examples

Local OAuth2 Server

With [Spring Security OAuth](#) you can easily create your own OAuth2 Server with the following 2 simple annotations:

- `@EnableResourceServer`
- `@EnableAuthorizationServer`

A working example application can be found at:

<https://github.com/ghillert/oauth-test-server/>

Simply clone the project, build and start it. Furthermore configure Spring Cloud Data Flow with the respective *Client Id* and *Client Secret*.

Authentication using GitHub

If you rather like to use an existing OAuth2 provider, here is an example for GitHub. First you need to **Register a new application** under your GitHub account at:

<https://github.com/settings/developers>

When running a default version of Spring Cloud Data Flow locally, your GitHub configuration should look like the following:

Application name

Something users will recognize and trust

Homepage URL

The full URL to your application homepage

Application description

This is displayed to all potential users of your application

Authorization callback URL

Your application's callback URL. Read our [OAuth documentation](#) for more information.

Update application **Delete application**

Figure 23.4. Register an OAuth Application for GitHub



Note

For the *Authorization callback URL* you will enter Spring Cloud Data Flow's Login URL, e.g. localhost:9393/login.

Configure Spring Cloud Data Flow with the GitHub relevant Client Id and Secret:

```
security:
  oauth2:
    client:
      client-id: your-github-client-id
      client-secret: your-github-client-secret
      access-token-uri: https://github.com/login/oauth/access_token
      user-authorization-uri: https://github.com/login/oauth/authorize
    resource:
      user-info-uri: https://api.github.com/user
```



Important

GitHub does not support the OAuth2 password grant type. As such you cannot use the Spring Cloud Data Flow Shell in conjunction with GitHub.

23.4 Securing the Spring Boot Management Endpoints

When enabling security, please also make sure that the [Spring Boot HTTP Management Endpoints](#) are secured as well. You can enable security for the management endpoints by adding the following to `application.yml`:

```
management:
  contextPath: /management
```

```
security:  
  enabled: true
```



Important

If you don't explicitly enable security for the management endpoints, you may end up having unsecured REST endpoints, despite `security.basic.enabled` being set to `true`.

24. Monitoring and Management

The Spring Cloud Data Flow server is a Spring Boot application that includes the [Actuator library](#), which adds several production ready features to help you monitor and manage your application.

The Actuator library adds http endpoints under the context path `/management` that is also a discovery page for available endpoints. For example, there is a `health` endpoint that shows application health information and an `env` that lists properties from Spring's `ConfigurableEnvironment`. By default only the health and application info endpoints are accessible. The other endpoints are considered to be *sensitive* and need to be [enabled explicitly via configuration](#). If you are enabling *sensitive* endpoints then you should also [secure the Data Flow server's endpoints](#) so that information is not inadvertently exposed to unauthenticated users. The local Data Flow server has security disabled by default, so all actuator endpoints are available.

The Data Flow server requires a relational database and if the feature toggled for analytics is enabled, a Redis server is also required. The Data Flow server will autoconfigure the [DataSourceHealthIndicator](#) and [RedisHealthIndicator](#) if needed. The health of these two services is incorporated to the overall health status of the server through the `health` endpoint.

24.1 Spring Boot Admin

A nice way to visualize and interact with actuator endpoints is to incorporate the [Spring Boot Admin](#) client library into the Spring Cloud Data Flow server. You can create the Spring Boot Admin application by following [a few simple steps](#).

A simple way to have the Spring Cloud Data Flow server be a client to the Spring Boot Admin Server is by adding a dependency to the Data Flow server's Maven pom.xml file and an additional configuration property as documented in [Registering Client Applications](#). You will need to clone the github repository for the Spring Cloud Data Flow server in order to modify the Maven pom. There are tags in the repository for each release.

Adding this dependency will result in a UI with tabs for each of the actuator endpoints.

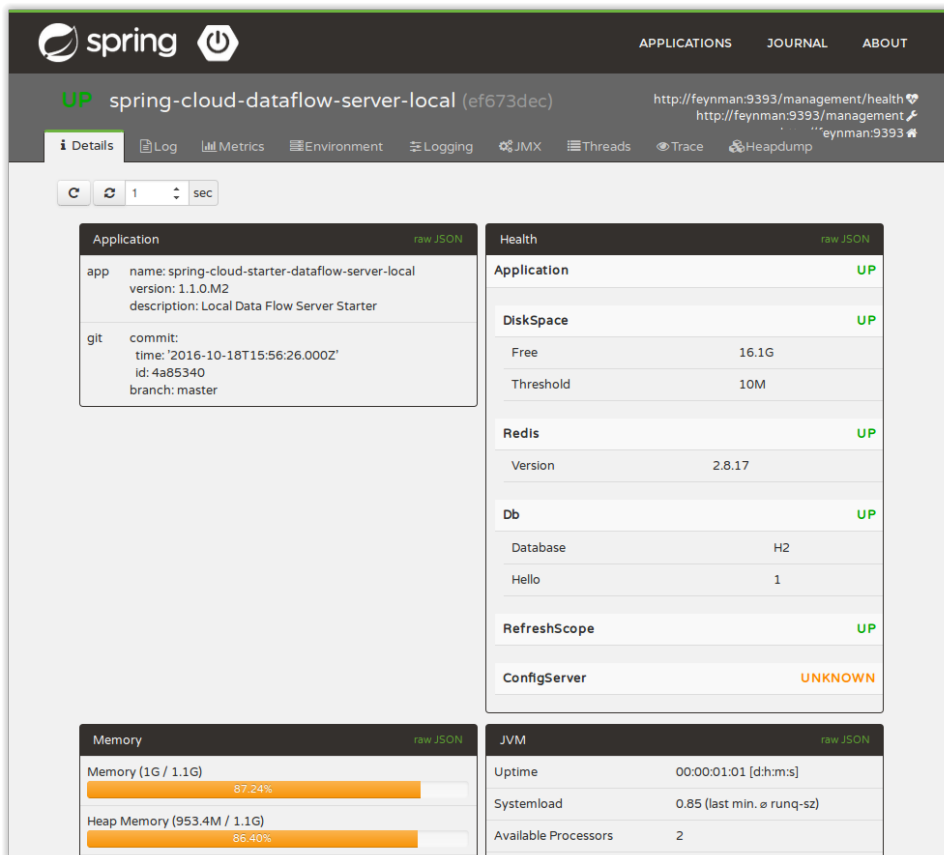


Figure 24.1. Spring Boot Admin UI

Additional configuration is required to interact with JMX beans and logging levels. Refer to the Spring Boot admin documentation for more information. As only the `info` and `health` endpoints are available to unauthenticated users, you should enable security on the Data Flow Server and also [configure Spring Boot Admin server's security](#) so that it can securely access the actuator endpoints.

24.2 Monitoring Deployed Applications

The applications that are deployed by Spring Cloud Data Flow are based on Spring Boot which contains several features for monitoring your application in production. Each deployed application contains [several web endpoints](#) for monitoring and interacting with Stream and Task applications.

In particular, the `/metrics` endpoint contains counters and gauges for HTTP requests, [System Metrics](#) (such as JVM stats), [DataSource Metrics](#) and [Message Channel Metrics](#) (such as message rates). Spring Boot lets you [add your own metrics](#) to the `/metrics` endpoint either by registering an implementation of the `PublicMetrics` interface or through its integration with [Dropwizard](#).

The Spring Boot interfaces `MetricWriter` and `Exporter` are used to send the metrics data to a place where they can be displayed and analyzed. There are implementations in Spring Boot to export metrics to Redis, Open TSDB, Statsd, and JMX.

There are a few additional Spring projects that provide support for sending metrics data to external systems.

- [Spring Cloud Stream](#) provides `ApplicationMetricsExporter` which publishes metrics via an [Emitter](#) to a messaging middleware destination.

- [Spring Cloud Data Flow Metrics Collector](#) subscribes to the metrics destination and aggregates metric messages published by the Spring Cloud Stream applications. It has an HTTP endpoint to access the aggregated metrics.
- [Spring Cloud Data Flow Metrics](#) provides `LogMetricWriter` that writes to the log.
- [Spring Cloud Data Flow Metrics Datadog Metrics](#) provides `DatadogMetricWriter` that writes to [Datadog](#).

The Spring Cloud Stream [Emitter](#) is used by the [Spring Cloud Stream App Starters](#) project that provides the most commonly used applications when creating Data Flow Streams.

The architecture when using Spring Cloud Stream's `Emitter`, the Data Flow Metrics Collector, and the Data Flow server is shown below.

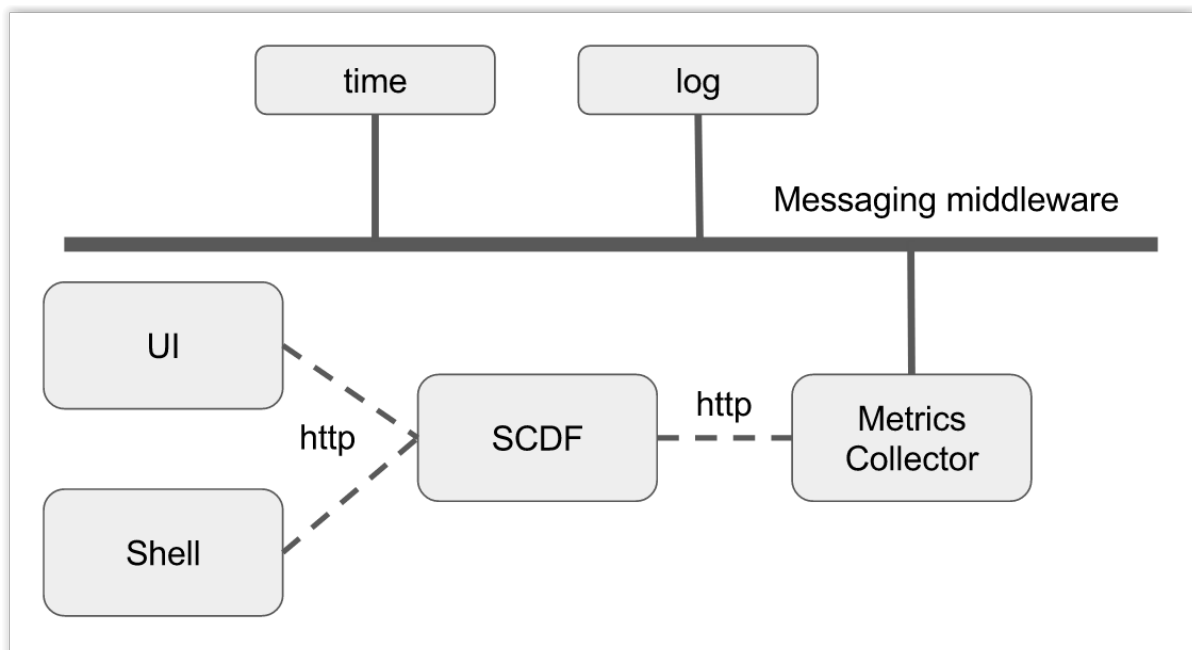


Figure 24.2. Spring Cloud Data Flow Metrics Architecture

As with the App Starters, there is a Spring Boot uber jar artifact of the Metrics Collector for all of the supported binders. You can find more information on building and running the Metrics Collector on its [project page](#).

The dataflow server now accepts an optional property `spring.cloud.dataflow.metrics.collector.uri`, this property should point to the URI of your deployed metrics collector app. For example, if you are running the metrics collector locally on port 8080 then start the server (local example) with the following command:

```
$ java -jar spring-cloud-dataflow-server-local-1.2.1.RELEASE.jar --
spring.cloud.dataflow.metrics.collector.uri=http://localhost:8080
```

The Metrics Collector can be secured with 'basic' authentication that requires a username and password. To set the username and password, use the properties `spring.cloud.dataflow.metrics.collector.username` and `spring.cloud.dataflow.metrics.collector.password`.

The metrics for each application are published when the property `spring.cloud.stream.bindings.applicationMetrics.destination` is set. This can be set

as any other application property when deploying an application in Data Flow. Since it is quite common to want all applications in a stream to emit metrics, setting it at the Data Flow server level is a good way to achieve that.

```
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.bindings.applicationMetrics.destination=metrics
```

Using the destination name `metrics` is a good choice as the Metrics Collector subscribes to that name by default.

The next most common way to configure the metrics destination is using deployment properties. Here is an example for the `ticktock` stream that uses the App Starters `time` and `log` applications.

```
app register --name time --type source --uri maven://org.springframework.cloud.stream.app:time-source-rabbit:1.2.0.RELEASE

app register --name log --type sink --uri maven://org.springframework.cloud.stream.app:log-sink-rabbit:1.2.0.RELEASE

stream create --name foostream --definition "time | log"

stream deploy --name foostream --
properties "app.*.spring.cloud.stream.bindings.applicationMetrics.destination=metrics,deployer.*.count=2"
```

The Metrics Collector exposes aggregated metrics under the HTTP endpoint `/collector/metrics` in JSON format. The Data Flow server accesses this endpoint in two distinct ways. The first is by exposing a `/metrics/streams` HTTP endpoint that acts as a proxy to the Metrics Collector endpoint. This is accessed by the UI when overlaying message rates on the Flo diagrams for each stream. It is also accessed to enrich the Data Flow `/runtime/apps` endpoint that is exposed in the UI via the `Runtime` tab and in the shell through the `runtime apps` command with message rates.

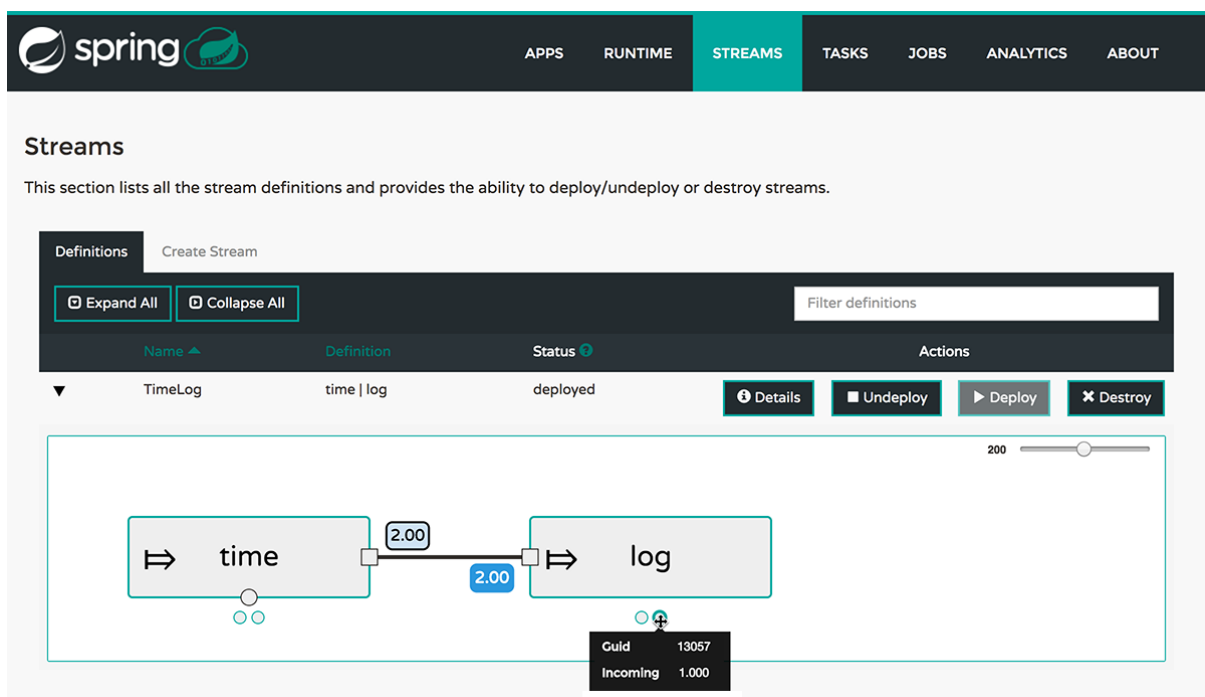


Figure 24.3. Stream Message Rates

By default, Data Flow will set the property

```
spring.cloud.stream.metrics.properties=spring.application.name,spring.application.index,spring.cloud.application.*,spring.c
```

Which is the set of application properties values needed to perform aggregation. It will also set the property

```
spring.metrics.export.triggers.application.includes=integration**`
```

since Data Flow will only display instantaneous input and output channel message rates. By default, all metric values in the `/metric` endpoint are sent so restricting it reduces the size of the message payload without impacting the functionality. Data Flow also exposes a `guid` property when displaying metric data which is used track back to the specific application instance that generated the metric. The `guid` value is platform dependent.

Note that you can override these defaults by setting them as you would any application property value.

Data Flow will not provide its own implementation to store and visualize historical metrics data. We will integrate with existing ALM system by providing an Exporter application that consumes messages from the same destination as the Metrics Collector and writes them to an existing ALM system. Which specific ALM system we will support is driven by user demand. However, to serve as an example, we will develop an Elastic Search exporter with a Grafana front end since it is open source.

24.3 Log and DataDog MetricWriter

If you prefer to have deployed applications bypass the centralized collection of metrics via the Metrics Collector, you can use the MetricWriters in [Spring Cloud Data Flow Metrics](#) and [Spring Cloud Data Flow Metrics Datadog Metrics](#).

The Data Flow Metrics project provides the foundation for exporting Spring Boot metrics via MetricWriters. It provides Spring Boots AutoConfiguration to setup the writing process and common functionality such as defining a metric name prefix appropriate for your environment. For example, you may want to include the region where the application is running in addition to the application's name and stream/task to which it belongs. It also includes a `LogMetricWriter` so that metrics can be stored into the log file. While very simple in approach, log files are often ingested into application monitoring tools (such as Splunk) where they can be further processed to create dashboards of an application's performance.

To make use of this functionality, you will need to add additional dependencies into your Stream and Task applications. To customize the "out of the box" Task and Stream applications you can use the [Data Flow Initializr](#) to generate a project and then add to the generated Maven pom file the MetricWriter implementation you want to use. The documentation on the Data Flow Metrics project pages provides the additional information you need to get started.

Part V. Streams

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

25. Introduction

In Spring Cloud Data Flow, a basic stream defines the ingestion of event data from a *source* to a *sink* that passes through any number of *processors*. Streams are composed of [Spring Cloud Stream](#) applications and the deployment of stream definitions is done via the Data Flow Server (REST API). The [Getting Started](#) section shows you how to start the server 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 UNIX pipes and filters syntax. Default values for ports and filenames are used in this example but can be overridden using `--` options, such as

```
http --server.port=8091 | file --directory=/tmp/httpdata/
```

To create these stream definitions you use the shell or make an HTTP POST request to the Spring Cloud Data Flow Server. For more information on making HTTP request directly to the server, consult the [REST API Guide](#).

26. Stream DSL

In the example above, we connected a source to a sink using the pipe symbol `|`. You can also pass properties to the source and sink configurations. The property names will depend on the individual app implementations, but as an example, the `http` source app exposes a `server.port` setting and it allows you to change the data ingestion port from the default value. To create the stream using port 8000, we would use

```
dataflow:> stream create --definition "http --server.port=8000 | log" --name myhttpstream
```

The shell provides tab completion for application properties and also the shell command `app info <appType> : <appName>` provides additional documentation for all the supported properties.



Note

Supported Stream `<appType>`'s are: source, processor, and sink

27. Register a Stream App

Register a Stream App with the App Registry using the Spring Cloud Data Flow Shell `app register` command. You must provide a unique name, application type, and a URI that can be resolved to the app artifact. For the type, specify "source", "processor", or "sink". Here are a few examples:

```
dataflow:>app register --name mysource --type source --uri maven://com.example:mysource:0.0.1-SNAPSHOT

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

dataflow:>app register --name mysink --type sink --uri http://example.com/mysink-2.0.1.jar
```

When providing a URI with the `maven` scheme, the format should conform to the following:

```
maven://<groupId>:<artifactId>[:<extension>[:<classifier>]]:<version>
```

For example, if you would like to register the snapshot versions of the `http` and `log` applications built with the RabbitMQ binder, you could do the following:

```
dataflow:>app register --name http --type source --uri maven://
org.springframework.cloud.stream.app:http-source-rabbit:1.2.1.BUILD-SNAPSHOT
dataflow:>app register --name log --type sink --uri maven://org.springframework.cloud.stream.app:log-
sink-rabbit:1.2.1.BUILD-SNAPSHOT
```

If you would like to register multiple apps at one time, you can store them in a properties file where the keys are formatted as `<type>.<name>` and the values are the URIs.

For example, if you would like to register the snapshot versions of the `http` and `log` applications built with the RabbitMQ binder, you could have the following in a properties file [eg: *stream-apps.properties*]:

```
source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.1.BUILD-SNAPSHOT
sink.log=maven://org.springframework.cloud.stream.app:log-sink-rabbit:1.2.1.BUILD-SNAPSHOT
```

Then to import the apps in bulk, use the `app import` command and provide the location of the properties file via `--uri`:

```
dataflow:>app import --uri file:///<YOUR_FILE_LOCATION>/stream-apps.properties
```

For convenience, we have the static files with application-URIs (for both maven and docker) available for all the out-of-the-box stream and task/batch app-starters. You can point to this file and import all the application-URIs in bulk. Otherwise, as explained in previous paragraphs, you can register them individually or have your own custom property file with only the required application-URIs in it. It is recommended, however, to have a "focused" list of desired application-URIs in a custom property file.

List of available Stream Application Starters:

Artifact Type	Stable Release	SNAPSHOT Release
RabbitMQ + Maven	bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven	bit.ly/Bacon-BUILD-SNAPSHOT-stream-applications-rabbit-maven
RabbitMQ + Docker	bit.ly/Bacon-RELEASE-stream-applications-rabbit-docker	N/A

Artifact Type	Stable Release	SNAPSHOT Release
Kafka 0.9 + Maven	bit.ly/Bacon-RELEASE-stream-applications-kafka-09-maven	bit.ly/Bacon-BUILD-SNAPSHOT-stream-applications-kafka-09-maven
Kafka 0.9 + Docker	bit.ly/Bacon-RELEASE-stream-applications-kafka-09-docker	N/A
Kafka 0.10 + Maven	bit.ly/Bacon-RELEASE-stream-applications-kafka-10-maven	bit.ly/Bacon-BUILD-SNAPSHOT-stream-applications-kafka-10-maven
Kafka 0.10 + Docker	bit.ly/Bacon-RELEASE-stream-applications-kafka-10-docker	N/A

List of available Task Application Starters:

Artifact Type	Stable Release	SNAPSHOT Release
Maven	bit.ly/Belmont-GA-task-applications-maven	bit.ly/Belmont-BUILD-SNAPSHOT-task-applications-maven
Docker	bit.ly/Belmont-GA-task-applications-docker	N/A

You can find more information about the available task starters in the [Task App Starters Project Page](#) and related reference documentation. For more information about the available stream starters look at the [Stream App Starters Project Page](#) and related reference documentation.

As an example, if you would like to register all out-of-the-box stream applications built with the RabbitMQ binder in bulk, you can with the following command.

```
dataflow:>app import --uri http://bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven
```

You can also pass the `--local` option (which is `true` by default) to indicate whether the properties file location should be resolved within the shell process itself. If the location should be resolved from the Data Flow Server process, specify `--local false`.



Warning

When using either `app register` or `app import`, if an 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 app coordinates, then include the `--force` option.

Note however that once downloaded, applications may be cached locally on the Data Flow server, based on the resource location. If the resource location doesn't change (even though the actual resource *bytes* may be different), then it won't be re-downloaded. When using `maven://resources` on the other hand, using a constant location still may circumvent caching (if using `-SNAPSHOT` versions).

Moreover, if a stream is already deployed and using some version of a registered app, then (forcibly) re-registering a different app will have no effect until the stream is deployed anew.

**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.

27.1 Whitelisting application properties

Stream and Task applications are Spring Boot applications which are aware of many [Section 29.3, “Common application properties”](#), e.g. `server.port` but also families of properties such as those with the prefix `spring.jmx` and `logging`. When creating your own application it is desirable to whitelist properties so that the shell and the UI can display them first as primary properties when presenting options via TAB completion or in drop-down boxes.

To whitelist application properties create a file named `spring-configuration-metadata-whitelist.properties` in the `META-INF` resource directory. There are two property keys that can be used inside this file. The first key is named `configuration-properties.classes`. The value is a comma separated list of fully qualified `@ConfigurationProperty` class names. The second key is `configuration-properties.names` whose value is a comma separated list of property names. This can contain the full name of property, such as `server.port` or a partial name to whitelist a category of property names, e.g. `spring.jmx`.

The [Spring Cloud Stream application starters](#) are a good place to look for examples of usage. Here is a simple example of the file sink's `spring-configuration-metadata-whitelist.properties` file

```
configuration-properties.classes=org.springframework.cloud.stream.app.file.sink.FileSinkProperties
```

If we also wanted to add `server.port` to be white listed, then it would look like this:

```
configuration-properties.classes=org.springframework.cloud.stream.app.file.sink.FileSinkProperties
configuration-properties.names=server.port
```

**Important**

Make sure to add 'spring-boot-configuration-processor' as an optional dependency to generate configuration metadata file for the properties.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>
```

27.2 Creating and using a dedicated metadata artifact

You can go a step further in the process of describing the main properties that your stream or task app supports by creating a so-called metadata companion artifact. This simple jar file contains only the Spring boot JSON file about configuration properties metadata, as well as the whitelisting file described in the previous section.

Here is the contents of such an artifact, for the canonical `log` sink:

```
$ jar tvf log-sink-rabbit-1.2.1.BUILD-SNAPSHOT-metadata.jar
373848 META-INF/spring-configuration-metadata.json
```

174 META-INF/spring-configuration-metadata-whitelist.properties

Note that the `spring-configuration-metadata.json` file is quite large. This is because it contains the concatenation of *all* the properties that are available at runtime to the log sink (some of them come from `spring-boot-actuator.jar`, some of them come from `spring-boot-autoconfigure.jar`, even some more from `spring-cloud-starter-stream-sink-log.jar`, etc.) Data Flow always relies on all those properties, even when a companion artifact is not available, but here all have been merged into a single file.

To help with that (as a matter of fact, you don't want to try to craft this giant JSON file by hand), you can use the following plugin in your build:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-app-starter-metadata-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>aggregate-metadata</id>
      <phase>compile</phase>
      <goals>
        <goal>aggregate-metadata</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```



Note

This plugin comes in *addition* to the `spring-boot-configuration-processor` that creates the individual JSON files. Be sure to configure the two!

The benefits of a companion artifact are manifold:

1. being way lighter (usually a few kilobytes, as opposed to megabytes for the actual app), they are quicker to download, allowing quicker feedback when using e.g. `app info` or the Dashboard UI
2. as a consequence of the above, they can be used in resource constrained environments (such as PaaS) when metadata is the only piece of information needed
3. finally, for environments that don't deal with boot uberjars directly (for example, Docker-based runtimes such as Kubernetes or Mesos), this is the only way to provide metadata about the properties supported by the app.

Remember though, that this is entirely optional when dealing with uberjars. The uberjar itself *also* includes the metadata in it already.

Using the companion artifact

Once you have a companion artifact at hand, you need to make the system aware of it so that it can be used.

When registering a single app *via* `app register`, you can use the optional `--metadata-uri` option in the shell, like so:

```
dataflow:>app register --name log --type sink
--uri maven://org.springframework.cloud.stream.app:log-sink-kafka-10:1.2.1.BUILD-SNAPSHOT
--metadata-uri=maven://org.springframework.cloud.stream.app:log-sink-
kafka-10:jar:metadata:1.2.1.BUILD-SNAPSHOT
```

When registering several files using the `app import` command, the file should contain a `<type>.<name>.metadata` line in addition to each `<type>.<name>` line. This is optional (*i.e.* if some apps have it but some others don't, that's fine).

Here is an example for a Dockerized app, where the metadata artifact is being hosted in a Maven repository (but retrieving it *via* `http://` or `file://` would be equally possible).

```
...
source.http=docker:springcloudstream/http-source-rabbit:latest
source.http.metadata=maven://org.springframework.cloud.stream.app:http-source-
rabbit:jar:metadata:1.2.1.BUILD-SNAPSHOT
...
```

28. Creating custom applications

While there are out of the box source, processor, sink applications available, one can extend these applications or write a custom [Spring Cloud Stream](#) application.

The process of creating Spring Cloud Stream applications via Spring Initializr is detailed in the Spring Cloud Stream [documentation](#). It is possible to include multiple binders to an application. If doing so, refer the instructions in [the section called “Passing Spring Cloud Stream properties for the application”](#) on how to configure them.

For supporting property whitelisting, Spring Cloud Stream applications running in Spring Cloud Data Flow may include the Spring Boot `configuration-processor` as an optional dependency, as in the following example.

```
<dependencies>
  <!-- other dependencies -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
  </dependency>
</dependencies>
```



Note

Make sure that the `spring-boot-maven-plugin` is included in the POM. The plugin is necessary for creating the executable jar that will be registered with Spring Cloud Data Flow. Spring Initializr will include the plugin in the generated POM.

Once a custom application has been created, it can be registered as described in [Chapter 27, Register a Stream App](#).

29. Creating a Stream

The Spring Cloud Data Flow Server exposes a full RESTful API for managing the lifecycle of stream definitions, but the easiest way to use it is via the Spring Cloud Data Flow shell. Start the shell as described in the [Getting Started](#) section.

New streams are created by with the help of stream definitions. The definitions are built from a simple DSL. For example, let's walk through what happens if we execute the following shell command:

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

This defines a stream named `ticktock` based off the DSL expression `time | log`. The DSL uses the "pipe" symbol `|`, to connect a source to a sink.

Then to deploy the stream execute the following shell command (or alternatively add the `--deploy` flag when creating the stream so that this step is not needed):

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

The Data Flow Server resolves `time` and `log` to maven coordinates and uses those to launch the `time` and `log` applications of the stream.

```
2016-06-01 09:41:21.728 INFO 79016 --- [nio-9393-exec-6] o.s.c.d.spi.local.LocalAppDeployer :
    deploying app ticktock.log instance 0
    Logs will be in /var/folders/wn/8jxm_tbdlvj28c8vj37n900m0000gn/T/spring-cloud-
dataflow-912434582726479179/ticktock-1464788481708/ticktock.log
2016-06-01 09:41:21.914 INFO 79016 --- [nio-9393-exec-6] o.s.c.d.spi.local.LocalAppDeployer :
    deploying app ticktock.time instance 0
    Logs will be in /var/folders/wn/8jxm_tbdlvj28c8vj37n900m0000gn/T/spring-cloud-
dataflow-912434582726479179/ticktock-1464788481910/ticktock.time
```

In this example, the `time` source simply sends the current time as a message each second, and the `log` sink outputs it using the logging framework. You can tail the `stdout` log (which has an "`_<instance>`" suffix). The log files are located within the directory displayed in the Data Flow Server's log output, as shown above.

```
$ tail -f /var/folders/wn/8jxm_tbdlvj28c8vj37n900m0000gn/T/spring-cloud-dataflow-912434582726479179/
ticktock-1464788481708/ticktock.log/stdout_0.log
2016-06-01 09:45:11.250 INFO 79194 --- [ kafka-binder-] log.sink : 06/01/16 09:45:11
2016-06-01 09:45:12.250 INFO 79194 --- [ kafka-binder-] log.sink : 06/01/16 09:45:12
2016-06-01 09:45:13.251 INFO 79194 --- [ kafka-binder-] log.sink : 06/01/16 09:45:13
```

29.1 Application properties

Application properties are the properties associated with each application in the stream. When the application is deployed, the application properties are applied to the application via command line arguments or environment variables based on the underlying deployment implementation.

Passing application properties when creating a stream

The following stream

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

can have application properties defined at the time of stream creation.

The shell command `app info <appType>:<appName>` displays the white-listed application properties for the application. For more info on the property white listing refer to [Section 27.1, “Whitelisting application properties”](#)

Below are the white listed properties for the app `time`:

```
dataflow:> app info source:time
#####
#      Option Name      #      Description      #      Default      #
#      Type             #
#####
#trigger.time-unit      #The TimeUnit to apply to delay#<none>
#java.util.concurrent.TimeUnit #
#      #              #values.              #              #
#      #              #
#trigger.fixed-delay    #Fixed delay for periodic    #1
#java.lang.Integer      #
#      #              #triggers.              #              #
#      #              #
#trigger.cron           #Cron expression value for the #<none>
#java.lang.String       #
#      #              #Cron Trigger.              #              #
#      #              #
#trigger.initial-delay  #Initial delay for periodic  #0
#java.lang.Integer      #
#      #              #triggers.              #              #
#      #              #
#trigger.max-messages   #Maximum messages per poll, -1 #1
#java.lang.Long         #
#      #              #means infinity.              #              #
#      #              #
#trigger.date-format    #Format for the date value.  #<none>
#java.lang.String       #
#####
```

Below are the white listed properties for the app `log`:

```
dataflow:> app info sink:log
#####
#      Option Name      #      Description      #      Default      #
#      Type             #
#####
#log.name               #The name of the logger to use.#<none>
#java.lang.String       #
#log.level              #The level at which to log    #<none>
#org.springframework.integration#
#      #              #messages.              #              #
#      #              #
#n.handler.LoggingHandler$Level#
#log.expression         #A SpEL expression (against the#payload
#java.lang.String       #
#      #              #incoming message) to evaluate #              #
#      #              #
#      #              #as the logged message.              #              #
#      #              #
#####
```

The application properties for the `time` and `log` apps can be specified at the time of stream creation as follows:

```
dataflow:> stream create --definition "time --fixed-delay=5 / log --level=WARN" --name ticktock
```

Note that the properties `fixed-delay` and `level` defined above for the apps `time` and `log` are the 'short-form' property names provided by the shell completion. These 'short-form' property names are applicable only for the white-listed properties and in all other cases, only *fully qualified* property names should be used.

29.2 Deployment properties

When deploying the stream, properties that control the deployment of the apps into the target platform are known as `deployment` properties. For instance, one can specify how many instances need to be deployed for the specific application defined in the stream using the deployment property called `count`.

Application properties versus Deployer properties

Starting with version 1.2, the distinction between properties that are meant for the *deployed app* and properties that govern *how* this app is deployed (thanks to some implementation of a [spring cloud deployer](#)) is more explicit. The former should be passed using the syntax `app.<app-name>.<property-name>=<value>` while the latter use the `deployer.<app-name>.<short-property-name>=<value>`

The following table recaps the difference in behavior between the two.

	Application Properties	Deployer Properties
Example Syntax	<code>app.filter.expression=foo</code>	<code>deployer.filter.count=3</code>
What the application "sees"	<code>expression=foo</code> or <code><some-prefix>.expression=foo</code> if <code>expression</code> is one of the whitelisted properties	Nothing
What the deployer "sees"	Nothing	<code>spring.cloud.deployer.count=3</code> The <code>spring.cloud.deployer</code> prefix is automatically and always prepended to the property name
Typical usage	Passing/Overriding application properties, passing Spring Cloud Stream binder or partitioning properties	Setting the number of instances, memory, disk, etc.

Passing instance count as deployment property

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

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

Note that `count` is the **reserved** property name used by the underlying deployer. Hence, if the application also has a custom property named `count`, it is **not** supported when specified in 'short-form' form during stream *deployment* as it could conflict with the *instance* count deployer property. Instead, the `count` as a custom application property can be specified in its *fully qualified* form (example: `app.foo.bar.count`) during stream *deployment* or it can be specified using 'short-form' or *fully qualified* form during the stream *creation* where it will be considered as an app property.



Important

See [Chapter 36, Using Labels in a Stream](#).

Inline vs file reference properties

When using the Spring Cloud Data Flow Shell, there are two ways to provide deployment properties: either **inline** or via a **file reference**. Those two ways are exclusive and documented below:

Inline properties

use the `--properties` shell option and list properties as a comma separated list of key=value pairs, like so:

```
stream deploy foo
  --properties "deployer.transform.count=2,app.transform.producer.partitionKeyExpression=payload"
```

Using a file reference

use the `--propertiesFile` option and point it to a local `.properties`, `.yaml` or `.yml` file (i.e. that lives in the filesystem of the machine running the shell). Being read as a `.properties` file, normal rules apply (ISO 8859-1 encoding, `=`, `<space>` or `:` delimiter, etc.) although we recommend using `=` as a key-value pair delimiter for consistency:

```
stream deploy foo --propertiesFile myprops.properties
```

where `myprops.properties` contains:

```
deployer.transform.count=2
app.transform.producer.partitionKeyExpression=payload
```

Both the above properties will be passed as deployment properties for the stream `foo` above.

In case of using YAML as the format for the deployment properties, use the `.yaml` or `.yml` file extension when deploying the stream,

```
stream deploy foo --propertiesFile myprops.yaml
```

where `myprops.yaml` contains:

```
deployer:
  transform:
    count: 2
app:
  transform:
    producer:
      partitionKeyExpression: payload
```

Passing application properties when deploying a stream

The application properties can also be specified when deploying a stream. When specified during deployment, these application properties can either be specified as 'short-form' property names (applicable for white-listed properties) or *fully qualified* property names. The application properties should have the prefix `"app.<appName/label>"`.

For example, the stream

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

can be deployed with application properties using the 'short-form' property names:

```
dataflow:>stream deploy ticktock --properties "app.time.fixed-delay=5,app.log.level=ERROR"
```

When using the app label,

```
stream create ticktock --definition "a: time / b: log"
```

the application properties can be defined as:

```
stream deploy ticktock --properties "app.a.fixed-delay=4,app.b.level=ERROR"
```

Passing Spring Cloud Stream properties for the application

Spring Cloud Data Flow sets the required Spring Cloud Stream properties for the applications inside the stream. Most importantly, the `spring.cloud.stream.bindings.<input/output>.destination` is set internally for the apps to bind.

If someone wants to override any of the Spring Cloud Stream properties, they can be set via deployment properties.

For example, for the below stream

```
dataflow:> stream create --definition "http / transform --
expression=payload.getValue('hello').toUpperCase() / log" --name ticktock
```

if there are multiple binders available in the classpath for each of the applications and the binder is chosen for each deployment then the stream can be deployed with the specific Spring Cloud Stream properties as:

```
dataflow:>stream deploy ticktock --
properties "app.time.spring.cloud.stream.bindings.output.binder=kafka,app.transform.spring.cloud.stream.bindings.input.binder=kafka"
```



Note

Overriding the destination names is not recommended as Spring Cloud Data Flow takes care of setting this internally.

Passing per-binding producer consumer properties

A Spring Cloud Stream application can have producer and consumer properties set per-binding basis. While Spring Cloud Data Flow supports specifying short-hand notation for per binding producer properties such as `partitionKeyExpression`, `partitionKeyExtractorClass` as described in [the section called “Passing stream partition properties during stream deployment”](#), all the supported Spring Cloud Stream producer/consumer properties can be set as Spring Cloud Stream properties for the app directly as well.

The consumer properties can be set for the inbound channel name with the prefix `app.[app/label name].spring.cloud.stream.bindings.<channelName>.consumer.` and the producer properties can be set for the outbound channel name with the prefix `app.[app/label name].spring.cloud.stream.bindings.<channelName>.producer..` For example, the stream

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

can be deployed with producer/consumer properties as:

```
dataflow:>stream deploy ticktock --
properties "app.time.spring.cloud.stream.bindings.output.producer.requiredGroups=myGroup,app.time.spring.cloud.stream.bindings.output.producer.groupId=myGroup"
```

The binder specific producer/consumer properties can also be specified in a similar way.

For instance

```
dataflow:>stream deploy ticktock --
properties "app.time.spring.cloud.stream.rabbit.bindings.output.producer.autoBindDlq=true,app.log.spring.cloud.stream.rabbit"
```

Passing stream partition properties during stream deployment

A common pattern in stream processing is to partition the data as it is streamed. This entails deploying multiple instances of a message consuming app and using content-based routing so that messages with a given key (as determined at runtime) are always routed to the same app instance. You can pass the partition properties during stream deployment to declaratively configure a partitioning strategy to route each message to a specific consumer instance.

See below for examples of deploying partitioned streams:

app.[app/label name].producer.partitionKeyExtractorClass

The class name of a PartitionKeyExtractorStrategy (default `null`)

app.[app/label name].producer.partitionKeyExpression

A SpEL expression, evaluated against the message, to determine the partition key; only applies if `partitionKeyExtractorClass` is null. If both are null, the app is not partitioned (default `null`)

app.[app/label name].producer.partitionSelectorClass

The class name of a PartitionSelectorStrategy (default `null`)

app.[app/label name].producer.partitionSelectorExpression

A SpEL expression, evaluated against the partition key, to determine the partition index to which the message will be routed. The final partition index will be the return value (an integer) modulo `[nextModule].count`. If both the class and expression are null, the underlying binder's default PartitionSelectorStrategy will be applied to the key (default `null`)

In summary, an app is partitioned if its count is `> 1` and the previous app has a `partitionKeyExtractorClass` or `partitionKeyExpression` (class takes precedence). When a partition key is extracted, the partitioned app instance is determined by invoking the `partitionSelectorClass`, if present, or the `partitionSelectorExpression` % `partitionCount`, where `partitionCount` is application count in the case of RabbitMQ, and the underlying partition count of the topic in the case of Kafka.

If neither a `partitionSelectorClass` nor a `partitionSelectorExpression` is present the result is `key.hashCode() % partitionCount`.

Passing application content type properties

In a stream definition you can specify that the input or the output of an application need to be converted to a different type. You can use the `inputType` and `outputType` properties to specify the content type for the incoming data and outgoing data, respectively.

For example, consider the following stream:

```
dataflow:>stream create tuple --definition "http | filter --inputType=application/x-spring-tuple
--expression=payload.hasFieldName('hello') | transform --
expression=payload.getValue('hello').toUpperCase()
| log" --deploy
```

The `http` app is expected to send the data in JSON and the `filter` app receives the JSON data and processes it as a Spring Tuple. In order to do so, we use the `inputType` property on the filter app

to convert the data into the expected Spring Tuple format. The `transform` application processes the Tuple data and sends the processed data to the downstream `log` application.

When sending some data to the `http` application:

```
dataflow:>http post --data {"hello":"world","foo":"bar"} --contentType application/json --target http://localhost:<http-port>
```

At the log application you see the content as follows:

```
INFO 18745 --- [transform.tuple-1] log.sink : WORLD
```

Depending on how applications are chained, the content type conversion can be specified either as via the `--outputType` in the upstream app or as an `--inputType` in the downstream app. For instance, in the above stream, instead of specifying the `--inputType` on the 'transform' application to convert, the option `--outputType=application/x-spring-tuple` can also be specified on the 'http' application.

For the complete list of message conversion and message converters, please refer to Spring Cloud Stream [documentation](#).

Overriding application properties during stream deployment

Application properties that are defined during deployment override the same properties defined during the stream creation.

For example, the following stream has application properties defined during stream creation:

```
dataflow:> stream create --definition "time --fixed-delay=5 / log --level=WARN" --name ticktock
```

To override these application properties, one can specify the new property values during deployment:

```
dataflow:>stream deploy ticktock --properties "app.time.fixed-delay=4,app.log.level=ERROR"
```

29.3 Common application properties

In addition to configuration via DSL, Spring Cloud Data Flow provides a mechanism for setting common properties to all the streaming applications that are launched by it. This can be done by adding properties prefixed with `spring.cloud.dataflow.applicationProperties.stream` when starting the server. When doing so, the server will pass all the properties, without the prefix, to the instances it launches.

For example, all the launched applications can be configured to use a specific Kafka broker by launching the Data Flow server with the following options:

```
--
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.brokers=192.168.1.100:9092
--
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.zkNodes=192.168.1.100:2181
```

This will cause the properties `spring.cloud.stream.kafka.binder.brokers` and `spring.cloud.stream.kafka.binder.zkNodes` to be passed to all the launched applications.



Note

Properties configured using this mechanism have lower precedence than stream deployment properties. They will be overridden if a property with the same key is specified at stream

deployment time (e.g. `app.http.spring.cloud.stream.kafka.binder.brokers` will override the common property).

30. Destroying a Stream

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

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

If the stream was deployed, it will be undeployed before the stream definition is deleted.

31. 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
```


32. Other Source and Sink Application Types

Let's try something a bit more complicated and swap out the `time` source for something else. Another supported source type is `http`, which accepts data for ingestion over HTTP POSTs. Note that the `http` source accepts data on a different port from the Data Flow Server (default 8080). By default the port is randomly assigned.

To create a stream using an `http` source, but still using the same `log` sink, we would change the original command above to

```
dataflow:> stream create --definition "http | log" --name myhttpstream --deploy
```

which will produce the following output from the server

```
2016-06-01 09:47:58.920 INFO 79016 --- [io-9393-exec-10] o.s.c.d.spi.local.LocalAppDeployer :
    deploying app myhttpstream.log instance 0
    Logs will be in /var/folders/wn/8jxm_tbdlvj28c8vj37n900m0000gn/T/spring-cloud-
dataflow-912434582726479179/myhttpstream-1464788878747/myhttpstream.log
2016-06-01 09:48:06.396 INFO 79016 --- [io-9393-exec-10] o.s.c.d.spi.local.LocalAppDeployer :
    deploying app myhttpstream.http instance 0
    Logs will be in /var/folders/wn/8jxm_tbdlvj28c8vj37n900m0000gn/T/spring-cloud-
dataflow-912434582726479179/myhttpstream-1464788886383/myhttpstream.http
```

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

```
dataflow:> runtime apps
```

You should see that the corresponding `http` source has a `url` property containing the host and port information on which it is listening. You are now ready to post to that url, e.g.:

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

and the stream will then funnel the data from the `http` source to the output log implemented by the `log` sink

```
2016-06-01 09:50:22.121 INFO 79654 --- [ kafka-binder-] log.sink : hello
2016-06-01 09:50:26.810 INFO 79654 --- [ kafka-binder-] log.sink : goodbye
```

Of course, we could also change the sink implementation. You could pipe the output to a file (`file`), to `hadoop` (`hdfs`) or to any of the other sink apps which are available. You can also define your own apps.

33. Simple Stream Processing

As an example of a simple processing step, we can transform the payload of the HTTP posted data to upper case using the stream definitions

```
http | transform --expression=payload.toUpperCase() | log
```

To create this stream enter the following command in the shell

```
dataflow:> stream create --definition "http | transform --expression=payload.toUpperCase() | log" --name  
mystream --deploy
```

Posting some data (using a shell command)

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

Will result in an uppercased 'HELLO' in the log

```
2016-06-01 09:54:37.749 INFO 80083 --- [ kafka-binder-] log.sink : HELLO
```

34. Stateful Stream Processing

To demonstrate the data partitioning functionality, let's deploy the following stream with Kafka as the binder.

```
dataflow:>stream create --name words --definition "http --server.port=9900 | splitter --
expression=payload.split(' ') | log"
Created new stream 'words'

dataflow:>stream deploy words --properties
"app.splitter.producer.partitionKeyExpression=payload,deployer.log.count=2"
Deployed stream 'words'

dataflow:>http post --target http://localhost:9900 --data "How much wood would a woodchuck chuck if a
woodchuck could chuck wood"
> POST (text/plain;Charset=UTF-8) http://localhost:9900 How much wood would a woodchuck chuck if a
woodchuck could chuck wood
> 202 ACCEPTED
```

You'll see the following in the server logs.

```
2016-06-05 18:33:24.982 INFO 58039 --- [nio-9393-exec-9] o.s.c.d.spi.local.LocalAppDeployer :
  deploying app words.log instance 0
  Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-
dataflow-694182453710731989/words-1465176804970/words.log
2016-06-05 18:33:24.988 INFO 58039 --- [nio-9393-exec-9] o.s.c.d.spi.local.LocalAppDeployer :
  deploying app words.log instance 1
  Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-
dataflow-694182453710731989/words-1465176804970/words.log
```

Review the `words.log instance 0` logs:

```
2016-06-05 18:35:47.047 INFO 58638 --- [ kafka-binder-] log.sink : How
2016-06-05 18:35:47.066 INFO 58638 --- [ kafka-binder-] log.sink :
chuck
2016-06-05 18:35:47.066 INFO 58638 --- [ kafka-binder-] log.sink :
```

Review the `words.log instance 1` logs:

```
2016-06-05 18:35:47.047 INFO 58639 --- [ kafka-binder-] log.sink :
much
2016-06-05 18:35:47.066 INFO 58639 --- [ kafka-binder-] log.sink :
wood
2016-06-05 18:35:47.066 INFO 58639 --- [ kafka-binder-] log.sink :
would
2016-06-05 18:35:47.066 INFO 58639 --- [ kafka-binder-] log.sink : a
2016-06-05 18:35:47.066 INFO 58639 --- [ kafka-binder-] log.sink :
woodchuck
2016-06-05 18:35:47.067 INFO 58639 --- [ kafka-binder-] log.sink : if
2016-06-05 18:35:47.067 INFO 58639 --- [ kafka-binder-] log.sink : a
2016-06-05 18:35:47.067 INFO 58639 --- [ kafka-binder-] log.sink :
woodchuck
2016-06-05 18:35:47.067 INFO 58639 --- [ kafka-binder-] log.sink :
could
2016-06-05 18:35:47.067 INFO 58639 --- [ kafka-binder-] log.sink :
wood
```

This shows that payload splits that contain the same word are routed to the same application instance.

35. Tap a Stream

Taps can be created at various producer endpoints in a stream. For a stream like this:

```
stream create --definition "http | step1: transform --expression=payload.toUpperCase() | step2:
transform --expression=payload+'!' | log" --name mainstream --deploy
```

taps can be created at the output of `http`, `step1` and `step2`.

To create a stream that acts as a 'tap' on another stream requires to specify the `source destination name` for the tap stream. The syntax for source destination name is:

```
`:<streamName>.<label/appName>`
```

To create a tap at the output of `http` in the stream above, the source destination name is `mainstream.http` To create a tap at the output of the first transform app in the stream above, the source destination name is `mainstream.step1`

The tap stream DSL looks like this:

```
stream create --definition ":mainstream.http > counter" --name tap_at_http --deploy

stream create --definition ":mainstream.step1 > jdbc" --name tap_at_step1_transformer --deploy
```

Note the colon (:) prefix before the destination names. The colon allows the parser to recognize this as a destination name instead of an app name.

36. Using Labels in a Stream

When a stream is comprised of multiple apps with the same name, they must be qualified with labels:

```
stream create --definition "http | firstLabel: transform --expression=payload.toUpperCase() |  
secondLabel: transform --expression=payload+'!' | log" --name myStreamWithLabels --deploy
```

37. Explicit Broker Destinations in a Stream

One can connect to a specific destination name located in the broker (Rabbit, Kafka etc.,) either at the `source` or at the `sink` position.

The following stream has the destination name at the `source` position:

```
stream create --definition ":myDestination > log" --name ingest_from_broker --deploy
```

This stream receives messages from the destination `myDestination` located at the broker and connects it to the `log` app.

The following stream has the destination name at the `sink` position:

```
stream create --definition "http > :myDestination" --name ingest_to_broker --deploy
```

This stream sends the messages from the `http` app to the destination `myDestination` located at the broker.

From the above streams, notice that the `http` and `log` apps are interacting with each other via the broker (through the destination `myDestination`) rather than having a pipe directly between `http` and `log` within a single stream.

It is also possible to connect two different destinations (`source` and `sink` positions) at the broker in a stream.

```
stream create --definition ":destination1 > :destination2" --name bridge_destinations --deploy
```

In the above stream, both the destinations (`destination1` and `destination2`) are located in the broker. The messages flow from the source destination to the sink destination via a `bridge` app that connects them.

38. Directed Graphs in a Stream

If directed graphs are needed instead of the simple linear streams described above, two features are relevant.

First, named destinations may be used as a way to combine the output from multiple streams or for multiple consumers to share the output from a single stream. This can be done using the DSL syntax `http > :mydestination or :mydestination > log`.

Second, you may need to determine the output channel of a stream based on some information that is only known at runtime. In that case, a router may be used in the sink position of a stream definition. For more information, refer to the Router Sink starter's [README](#).

39. Stream applications with multiple binder configurations

In some cases, a stream can have its applications bound to multiple spring cloud stream binders when they are required to connect to different messaging middleware configurations. In those cases, it is important to make sure the applications are configured appropriately with their binder configurations. For example, let's consider the following stream:

```
http | transform --expression=payload.toUpperCase() | log
```

and in this stream, each application connects to messaging middleware in the following way:

```
Http source sends events to RabbitMQ (rabbit1)
Transform processor receives events from RabbitMQ (rabbit1) and sends the processed events into Kafka
(kafkal)
Log sink receives events from Kafka (kafkal)
```

Here, `rabbit1` and `kafkal` are the binder names given in the spring cloud stream application properties. Based on this setup, the applications will have the following binder(s) in their classpath with the appropriate configuration:

```
Http - Rabbit binder
Transform - Both Kafka and Rabbit binders
Log - Kafka binder
```

The `spring-cloud-stream` binder configuration properties can be set within the applications themselves. If not, they can be passed via deployment properties when the stream is deployed.

For example,

```
dataflow:>stream create --definition "http | transform --expression=payload.toUpperCase() | log" --name
mystream
```

```
dataflow:>stream deploy mystream --properties
"app.http.spring.cloud.stream.bindings.output.binder=rabbit1,app.transform.spring.cloud.stream.bindings.input.binder=rabbit1,
app.transform.spring.cloud.stream.bindings.output.binder=kafkal,app.log.spring.cloud.stream.bindings.input.binder=kafkal"
```

One can override any of the binder configuration properties by specifying them via deployment properties.

Part VI. Tasks

This section goes into more detail about how you can work with [Spring Cloud Task](#). It covers topics such as creating and running task applications.

If you're just starting out with Spring Cloud Data Flow, you should probably read the [Getting Started](#) guide before diving into this section.

40. 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.

41. 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

41.1 Creating a custom Task Application

While Spring Cloud Task does provide a number of out of the box applications (via the [spring-cloud-task-app-starters](#)), most task applications will be custom developed. In order to create a custom task application:

1. Create a new project via [Spring Initializer](#) via either the website or your IDE making sure to select the following starters:
 - a. Cloud Task - This dependency is the `spring-cloud-starter-task`.
 - b. JDBC - This is the dependency for the `spring-jdbc` starter.
2. Within your new project, create a new class that will serve as your main class:

```
@EnableTask
@SpringBootApplication
public class MyTask {

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

3. With this, you'll need one or more `CommandLineRunner` or `ApplicationRunner` within your application. You can either implement your own or use the ones provided by Spring Boot (there is one for running batch jobs for example).
4. Packaging your application up via Spring Boot into an über jar is done via the standard Boot conventions.
5. The packaged application can be registered and deployed as noted below.

41.2 Registering a Task Application

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

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

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

```
app import --uri file:///tmp/task-apps.properties
```

For convenience, we have the static files with application-URIs (for both maven and docker) available for all the out-of-the-box Task app-starters. You can point to this file and import all the application-URIs in bulk. Otherwise, as explained in previous paragraphs, you can register them individually or have your own custom property file with only the required application-URIs in it. It is recommended, however, to have a "focused" list of desired application-URIs in a custom property file.

List of available static property files:

Artifact Type	Stable Release	SNAPSHOT Release
Maven	http://bit.ly/Belmont-GA-task-applications-maven	http://bit.ly/Belmont-BUILD-SNAPSHOT-task-applications-maven
Docker	http://bit.ly/Belmont-GA-task-applications-docker	http://bit.ly/Belmont-BUILD-SNAPSHOT-task-applications-docker

For example, if you would like to register all out-of-the-box task applications in bulk, you can with the following command.

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

You can also pass the `--local` option (which is `TRUE` by default) to indicate whether the properties file location should be resolved within the shell process itself. If the location should be resolved from the Data Flow Server process, specify `--local false`.

When using either `app register` or `app import`, if a 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.

41.3 Creating a Task

Create a Task Definition from a Task App 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.

41.4 Launching a 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'
```

When a task is launched, any properties that need to be passed as the command line arguments to the task application can be set when launching the task as follows:

```
dataflow:>task launch mytask --arguments "--server.port=8080,--foo=bar"
```

Additional properties meant for a `TaskLauncher` itself can be passed in using a `--properties` option. Format of this option is a comma delimited string of properties prefixed with `app.<task definition name>.<property>`. Properties are passed to `TaskLauncher` as application properties and it is up to an implementation to choose how those are passed into an actual task application. If the property is prefixed with `deployer` instead of `app` it is passed to `TaskLauncher` as a deployment property and its meaning may be `TaskLauncher` implementation specific.

```
dataflow:>task launch mytask --properties "deployer.timestamp.foo1=bar1,app.timestamp.foo2=bar2"
```

Common application properties

In addition to configuration via DSL, Spring Cloud Data Flow provides a mechanism for setting common properties to all the task applications that are launched by it. This can be done by adding properties prefixed with `spring.cloud.dataflow.applicationProperties.task` when starting the server. When doing so, the server will pass all the properties, without the prefix, to the instances it launches.

For example, all the launched applications can be configured to use the properties `foo` and `fizz` by launching the Data Flow server with the following options:

```
--spring.cloud.dataflow.applicationProperties.task.foo=bar
--spring.cloud.dataflow.applicationProperties.task.fizz=bar2
```

This will cause the properties `foo=bar` and `fizz=bar2` to be passed to all the launched applications.



Note

Properties configured using this mechanism have lower precedence than task deployment properties. They will be overridden if a property with the same key is specified at task launch time (e.g. `app.trigger.fizz` will override the common property).

41.5 Reviewing Task Executions

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

41.6 Destroying a Task

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, instead it just removes the task definition from the database.

42. Task Repository

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

42.1 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 Data Flow will need to be rebuilt. Since Spring Cloud Data Flow 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 Data Flow](#)

Task Application Repository

When launching a task application be sure that the database driver that is being used by Spring Cloud Data Flow is also a dependency on the task application. For example if your Spring Cloud Data Flow is set to use `Postgresql`, be sure that the task application *also* has `Postgresql` as a dependency.



Note

When executing tasks externally (i.e. command line) and you wish for Spring Cloud Data Flow to show the TaskExecutions in its UI, be sure that common `datasource` settings are shared among the both. By default Spring Cloud Task will use a local H2 instance and the execution will not be recorded to the database used by Spring Cloud Data Flow.

42.2 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
```


43. Subscribing to Task/Batch Events

You can also tap into various task/batch events when the task is launched. If the task is enabled to generate task and/or batch events (with the additional dependencies `spring-cloud-task-stream` and `spring-cloud-stream-binder-kafka`, in the case of Kafka as the binder), those events are published during the task lifecycle. By default, the destination names for those published events on the broker (rabbit, kafka etc.,) are the event names themselves (for instance: `task-events`, `job-execution-events` etc.,).

```
dataflow:>task create myTask --definition "myBatchJob"
dataflow:>task launch myTask
dataflow:>stream create task-event-subscriber1 --definition ":task-events > log" --deploy
```

You can control the destination name for those events by specifying explicit names when launching the task such as:

```
dataflow:>task launch myTask --properties "spring.cloud.stream.bindings.task-
events.destination=myTaskEvents"
dataflow:>stream create task-event-subscriber2 --definition ":myTaskEvents > log" --deploy
```

The default Task/Batch event and destination names on the broker are enumerated below:

Table 43.1. Task/Batch Event Destinations

Event	Destination
Task events	task-events
Job Execution events	job-execution-events
Step Execution events	step-execution-events
Item Read events	item-read-events
Item Process events	item-process-events
Item Write events	item-write-events
Skip events	skip-events

44. Launching Tasks from a Stream

You can launch a task from a stream by using one of the available `task-launcher` sinks. Currently the platforms supported via the `task-launcher` sinks are [local](#), [Cloud Foundry](#), and [Yarn](#).



Note

`task-launcher-local` is meant for development purposes only.

A `task-launcher` sink expects a message containing a [TaskLaunchRequest](#) object in its payload. From the `TaskLaunchRequest` object the `task-launcher` will obtain the URI of the artifact to be launched as well as the environment properties, command line arguments, deployment properties and application name to be used by the task.

The [task-launcher-local](#) can be added to the available sinks by executing the app register command as follows (for the Rabbit Binder):

```
app register --name task-launcher-local --type sink --uri maven://
org.springframework.cloud.stream.app:task-launcher-local-sink-rabbit:jar:1.2.0.RELEASE
```

In the case of a maven based task that is to be launched, the `task-launcher` application is responsible for downloading the artifact. You **must** configure the `task-launcher` with the appropriate configuration of [Maven Properties](#) such as `--maven.remote-repositories.repo1.url=http://repo.spring.io/libs-milestone"` to resolve artifacts, in this case against a milestone repo. Note that this repo can be different than the one used to register the `task-launcher` application itself.

44.1 TriggerTask

One way to launch a task using the `task-launcher` is to use the [triggertask](#) source. The `triggertask` source will emit a message with a `TaskLaunchRequest` object containing the required launch information. The `triggertask` can be added to the available sources by executing the app register command as follows (for the Rabbit Binder):

```
app register --type source --name triggertask --uri maven://
org.springframework.cloud.stream.app:triggertask-source-rabbit:1.2.0.RELEASE
```

An example of this would be to launch the timestamp task once every 60 seconds, the stream to implement this would look like:

```
stream create foo --definition "triggertask --triggertask.uri=maven://
org.springframework.cloud.task.app:timestamp-task:jar:1.2.0.RELEASE --trigger.fixed-
delay=60 --triggertask.environment-properties=spring.datasource.url=jdbc:h2:tcp://
localhost:19092/mem:dataflow,spring.datasource.username=sa | task-launcher-local --maven.remote-
repositories.repo1.url=http://repo.spring.io/libs-release" --deploy
```

If you execute `runtime apps` you can find the log file for the task launcher sink. Tailing that file you can find the log file for the launched tasks. The setting of `triggertask.environment-properties` is so that all the task executions can be collected in the same H2 database used in the local version of the Data Flow Server. You can then see the list of task executions using the shell command `task execution list`

```
dataflow:>task execution list
#####
# Task Name      #ID#      Start Time      #      End Time      #Exit Code#
#####
```

```
#timestamp-task_26176#4 #Tue May 02 12:13:49 EDT 2017#Tue May 02 12:13:49 EDT 2017#0 #
#timestamp-task_32996#3 #Tue May 02 12:12:49 EDT 2017#Tue May 02 12:12:49 EDT 2017#0 #
#timestamp-task_58971#2 #Tue May 02 12:11:50 EDT 2017#Tue May 02 12:11:50 EDT 2017#0 #
#timestamp-task_13467#1 #Tue May 02 12:10:50 EDT 2017#Tue May 02 12:10:50 EDT 2017#0 #
#####
```

44.2 TaskLaunchRequest-transform

Another option to start a task using the `task-launcher` would be to create a stream using the [Tasklaunchrequest-transform](#) processor to translate a message payload to a `TaskLaunchRequest`.

The `tasklaunchrequest-transform` can be added to the available processors by executing the `app register` command as follows (for the Rabbit Binder):

```
app register --type processor --name tasklaunchrequest-transform --uri maven://
org.springframework.cloud.stream.app:tasklaunchrequest-transform-processor-rabbit:1.2.0.RELEASE
```

For example:

```
stream create task-stream --definition "http --port=9000 | tasklaunchrequest-transform --uri=maven://
org.springframework.cloud.task.app:timestamp-task:jar:1.2.0.RELEASE | task-launcher-local --
maven.remote-repositories.repo1.url=http://repo.spring.io/libs-release"
```

45. Composed Tasks

Spring Cloud Data Flow allows a user to create a directed graph where each node of the graph is a task application. This is done by using the DSL for composed tasks. A composed task can be created via the RESTful API, the Spring Cloud Data Flow Shell, or the Spring Cloud Data Flow UI.

45.1 Configuring the Composed Task Runner in Spring Cloud Data Flow

Composed tasks are executed via a task application called the [Composed Task Runner](#).

Registering the Composed Task Runner application

Out of the box the Composed Task Runner application is not registered with Spring Cloud Data Flow. So, to launch composed tasks we must first register the Composed Task Runner as an application with Spring Cloud Data Flow as follows:

```
app register --name composed-task-runner --type task --uri maven://
org.springframework.cloud.task.app:composedtaskrunner-task:<DESIRED_VERSION>
```

You can also configure Spring Cloud Data Flow to use a different task definition name for the composed task runner. This can be done by setting the `spring.cloud.dataflow.task.composedTaskRunnerName` property to the name of your choice. You can then register the composed task runner application with the name you set using that property.

Configuring the Composed Task Runner application

The Composed Task Runner application has a `dataflow.server.uri` property that is used for validation and for launching child tasks. This defaults to `localhost:9393`. If you run a distributed Spring Cloud Data Flow server, like you would do if you deploy the server on Cloud Foundry, YARN or Kubernetes, then you need to provide the URI that can be used to access the server. You can either provide this `dataflow.server.uri` property for the Composed Task Runner application when launching a composed task, or you can provide a `spring.cloud.dataflow.server.uri` property for the Spring Cloud Data Flow server when it is started. For the latter case the `dataflow.server.uri` Composed Task Runner application property will be automatically set when a composed task is launched.

45.2 Creating, Launching, and Destroying a Composed Task

Creating a Composed Task

The DSL for the composed tasks is used when creating a task definition via the task create command. For example:

```
dataflow:> app register --name timestamp --type task --uri maven://
org.springframework.cloud.task.app:timestamp-task:<DESIRED_VERSION>
dataflow:> app register --name mytaskapp --type task --uri file:///home/tasks/mytask.jar
dataflow:> task create my-composed-task --definition "mytaskapp && timestamp"
dataflow:> task launch my-composed-task
```

In the example above we assume that the applications to be used by our composed task have not been registered yet. So the first two steps we register two task applications. We then create our composed task definition by using the task create command. The composed task DSL in the example above will, when launched, execute mytaskapp and then execute the timestamp application.

But before we launch the my-composed-task definition, we can view what Spring Cloud Data Flow generated for us. This can be done by executing the task list command.

```
dataflow:>task list
#####
#      Task Name      #      Task Definition
#####
#my-composed-task      #mytaskapp && timestamp
#my-composed-task-mytaskapp#mytaskapp
#my-composed-task-timestamp#timestamp
```

Spring Cloud Data Flow created three task definitions, one for each of the applications that comprises our composed task (my-composed-task-mytaskapp and my-composed-task-timestamp) as well as the composed task (my-composed-task) definition. We also see that each of the generated names for the child tasks is comprised of the name of the composed task and the name of the application separated by a dash -. i.e. *my-composed-task - mytaskapp*.

Task Application Parameters

The task applications that comprise the composed task definition can also contain parameters. For example:

```
dataflow:> task create my-composed-task --definition "mytaskapp --displayMessage=hello && timestamp --format=YYYY"
```

Launching a Composed Task

Launching a composed task is done the same way as launching a stand-alone task. i.e.

```
task launch my-composed-task
```

Once the task is launched and assuming all the tasks complete successfully you will see three task executions when executing a task execution list. For example:

```
dataflow:>task execution list
#####
#      Task Name      #ID #      Start Time      #      End Time      #Exit Code#
#####
#my-composed-task-timestamp#713#Wed Apr 12 16:43:07 EDT 2017#Wed Apr 12 16:43:07 EDT 2017#0      #
#my-composed-task-mytaskapp#712#Wed Apr 12 16:42:57 EDT 2017#Wed Apr 12 16:42:57 EDT 2017#0      #
#my-composed-task      #711#Wed Apr 12 16:42:55 EDT 2017#Wed Apr 12 16:43:15 EDT 2017#0      #
#####
```

In the example above we see that my-compose-task launched and it also launched the other tasks in sequential order and all of them executed successfully with "Exit Code" as 0.

Exit Statuses

The following list shows how the Exit Status will be set for each step (task) contained in the composed task following each step execution.

- If the TaskExecution has an ExitMessage that will be used as the ExitStatus
- If no ExitMessage is present and the ExitCode is set to zero then the ExitStatus for the step will be COMPLETED.
- If no ExitMessage is present and the ExitCode is set to any non zero number then the ExitStatus for the step will be FAILED.

Destroying a Composed Task

The same command used to destroy a stand-alone task is the same as destroying a composed task. The only difference is that destroying a composed task will also destroy the child tasks associated with it. For example

```
dataflow:>task list
#####
#      Task Name      #      Task Definition
#####
#my-composed-task      #mytaskapp && timestamp
#my-composed-task-mytaskapp#mytaskapp
#my-composed-task-timestamp#timestamp
...
dataflow:>task destroy my-composed-task
dataflow:>task list
#####
#      Task Name      #      Task Definition
#####
```

Stopping a Composed Task

In cases where a composed task execution needs to be stopped. This can be done via the:

- RESTful API
- Spring Cloud Data Flow Dashboard by selecting the Job's tab and then clicking the stop button by the job execution that needs to be stopped.

The composed task run will be stopped when the currently running child task completes. The step associated with the child task that was running at the time that the composed task was stopped will be marked as `STOPPED` as well as the composed task job execution.

Restarting a Composed Task

In cases where a composed task fails during execution and the status of the composed task is `FAILED` then the task can be restarted. This can be done via the:

- RESTful API
- Shell by launching the task using the same parameters
- Spring Cloud Data Flow Dashboard by selecting the Job's tab and then clicking the restart button by the job execution that needs to be restarted.



Note

Restarting a Composed Task job that has been stopped (via the Spring Cloud Data Flow Dashboard or RESTful API), will relaunch the `STOPPED` child task, and then launch the remaining (unlaunched) child tasks in the specified order.

45.3 Composed Task DSL

Conditional Execution

Conditional execution is expressed using a double ampersand symbol `&&`. This allows each task in the sequence to be launched only if the previous task successfully completed. For example:

```
task create my-composed-task --definition "foo && bar"
```

When the composed task `my-composed-task` is launched, it will launch the task `foo` and if it completes successfully, then the task `bar` will be launched. If the `foo` task fails, then the task `bar` will not launch.

You can also use the Spring Cloud Data Flow Dashboard to create your conditional execution. By using the designer to drag and drop applications that are required, and connecting them together to create your directed graph. For example:

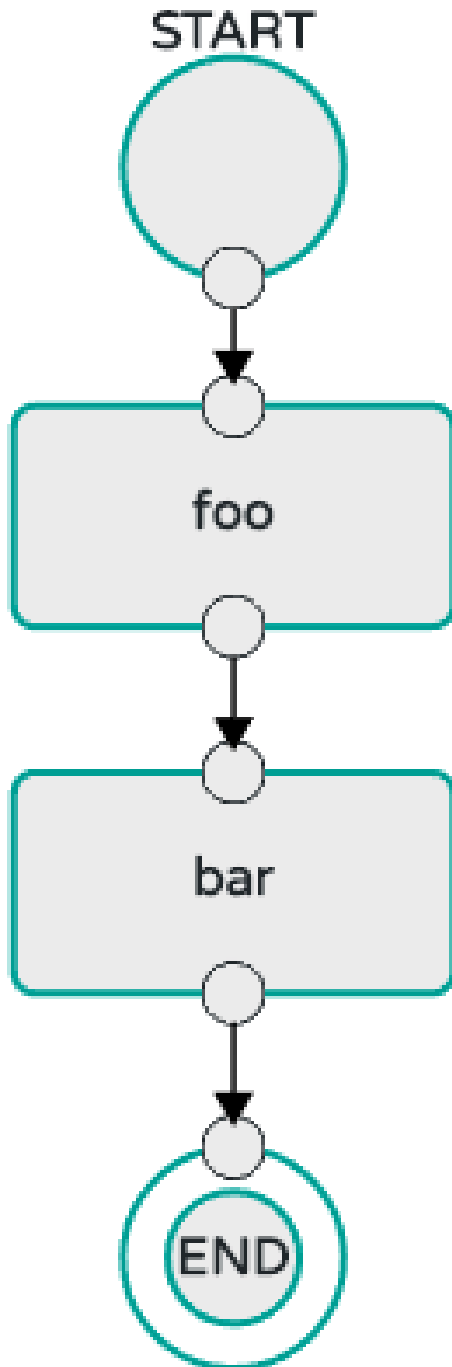


Figure 45.1. Conditional Execution

The diagram above is a screen capture of the directed graph as it being created using the Spring Cloud Data Flow Dashboard. We see that are 4 components in the diagram that comprise a conditional execution:

- Start icon - All directed graphs start from this symbol. There will only be one.
- Task icon - Represents each task in the directed graph.
- End icon - Represents the termination of a directed graph.
- Solid line arrow - Represents the flow conditional execution flow between:
 - Two applications
 - The start control node and an application
 - An application and the end control node

**Note**

You can view a diagram of your directed graph by clicking the detail button next to the composed task definition on the definitions tab.

Transitional Execution

The DSL supports fine grained control over the transitions taken during the execution of the directed graph. Transitions are specified by providing a condition for equality based on the exit status of the previous task. A task transition is represented by the following symbol `->`.

Basic Transition

A basic transition would look like the following:

```
task create my-transition-composed-task --definition "foo 'FAILED' -> bar 'COMPLETED' -> baz"
```

In the example above `foo` would launch and if it had an exit status of `FAILED`, then the `bar` task would launch. If the exit status of `foo` was `COMPLETED` then `baz` would launch. All other statuses returned by `foo` will have no effect and task would terminate normally.

Using the Spring Cloud Data Flow Dashboard to create the same "basic transition" would look like:

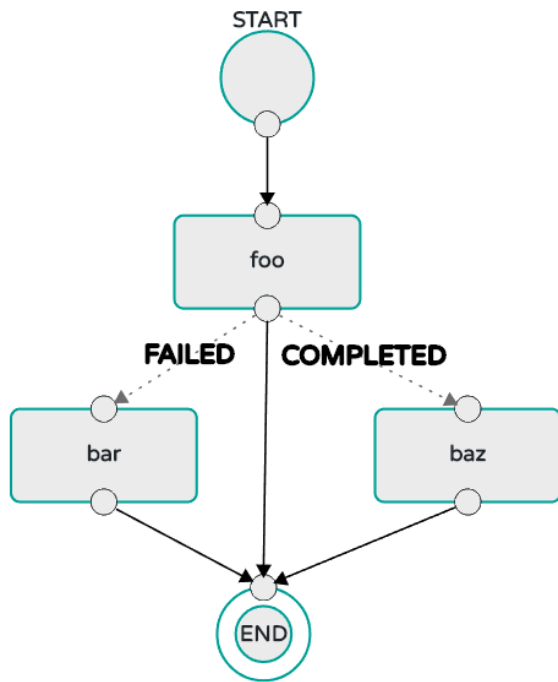


Figure 45.2. Basic Transition

The diagram above is a screen capture of the directed graph as it being created using the Spring Cloud Data Flow Dashboard. Notice that there are 2 different types of connectors:

- Dashed line - Is the line used to represent transitions from the application to one of the possible destination applications.
- Solid line - Used to connect applications in a conditional execution or a connection between the application and a control node (end, start).

When creating a transition, link the application to each of possible destination using the connector. Once complete go to each connection and select it by clicking it. A bolt icon should appear, click that icon and enter the exit status required for that connector. The solid line for that connector will turn to a dashed line.

Transition With a Wildcard

Wildcards are supported for transitions by the DSL for example:

```
task create my-transition-composed-task --definition "foo 'FAILED' -> bar '*' -> baz"
```

In the example above `foo` would launch and if it had an exit status of `FAILED`, then the `bar` task would launch. Any exit status of `foo` other than `FAILED` then `baz` would launch.

Using the Spring Cloud Data Flow Dashboard to create the same "transition with wildcard" would look like:

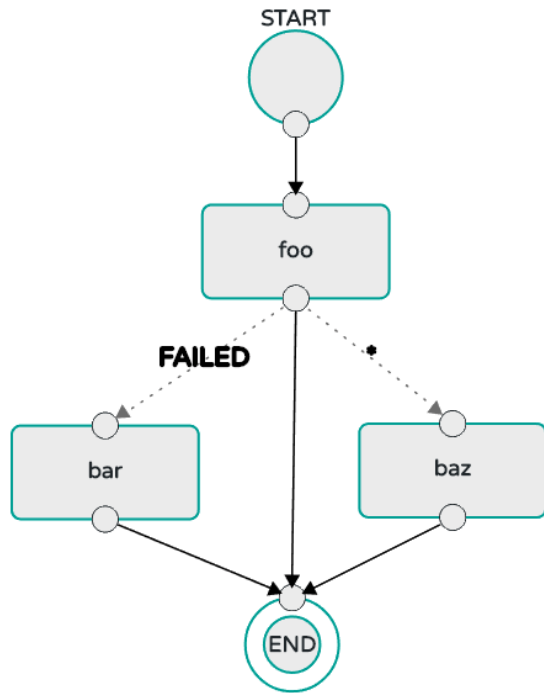


Figure 45.3. Basic Transition With Wildcard

Transition With a Following Conditional Execution

A transition can be followed by a conditional execution so long as the wildcard is not used. For example:

```
task create my-transition-conditional-execution-task --definition "foo 'FAILED' -> bar 'UNKNOWN' -> baz
&& qux && quux"
```

In the example above `foo` would launch and if it had an exit status of `FAILED`, then the `bar` task would launch. If `foo` had an exit status of `UNKNOWN` then `baz` would launch. Any exit status of `foo` other than `FAILED` or `UNKNOWN` then `qux` would launch and upon successful completion `quux` would launch.

Using the Spring Cloud Data Flow Dashboard to create the same "transition with conditional execution" would look like:

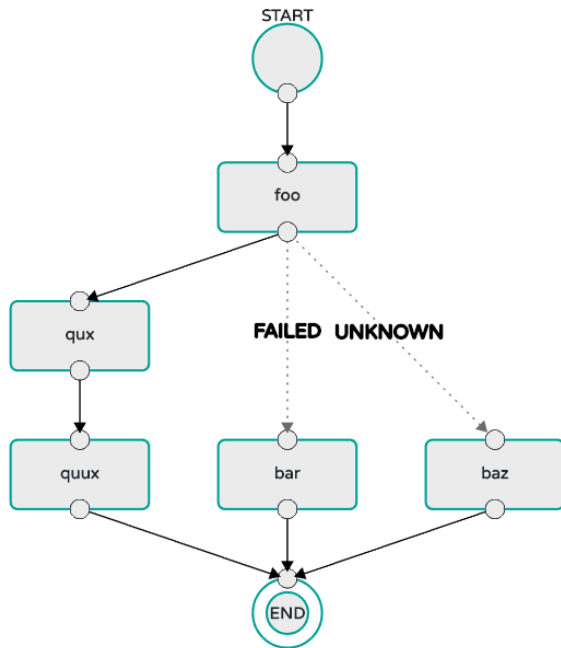


Figure 45.4. Transition With Conditional Execution



Note

In this diagram we see the dashed line (transition) connecting the `foo` application to the target applications, but a solid line connecting the conditional executions between `foo`, `quux`, and `quux`.

Split Execution

Splits allow for multiple tasks within a composed task to be run in parallel. It is denoted by using angle brackets `<>` to group tasks and flows that are to be run in parallel. These tasks and flows are separated by the double pipe `||`. For example:

```
task create my-split-task --definition "<foo || bar || baz>"
```

The example above will launch tasks `foo`, `bar` and `baz` in parallel.

Using the Spring Cloud Data Flow Dashboard to create the same "split execution" would look like:

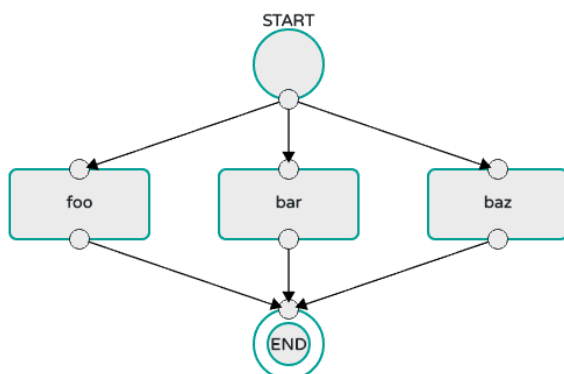


Figure 45.5. Split

With the task DSL a user may also execute multiple split groups in succession. For example:

```
task create my-split-task --definition "<foo || bar || baz> && <qux || quux>"
```

In the example above tasks `foo`, `bar` and `baz` will be launched in parallel, once they all complete then tasks `qux`, `quux` will be launched in parallel. Once they complete the composed task will end. However if `foo`, `bar`, or `baz` fails then, the split containing `qux` and `quux` will not launch.

Using the Spring Cloud Data Flow Dashboard to create the same "split with multiple groups" would look like:

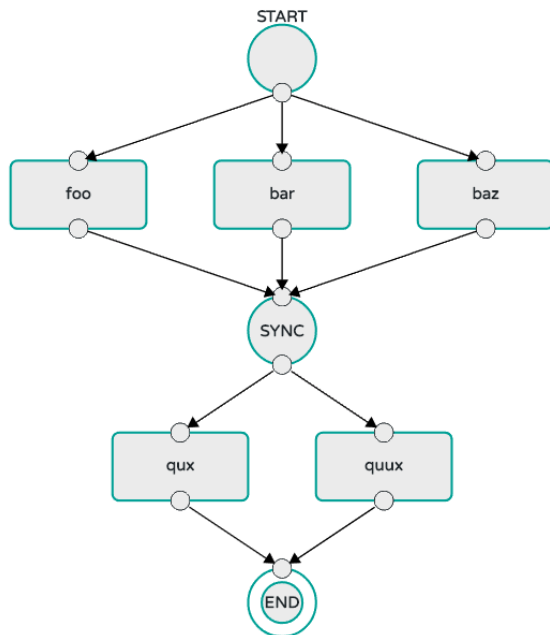


Figure 45.6. Split as a part of a conditional execution

Notice that there is a `SYNC` control node that is by the designer when connecting two consecutive splits.

Split Containing Conditional Execution

A split can also have a conditional execution within the angle brackets. For example:

```
task create my-split-task --definition "<foo && bar || baz>"
```

In the example above we see that `foo` and `baz` will be launched in parallel, however `bar` will not launch until `foo` completes successfully.

Using the Spring Cloud Data Flow Dashboard to create the same "split containing conditional execution" would look like:

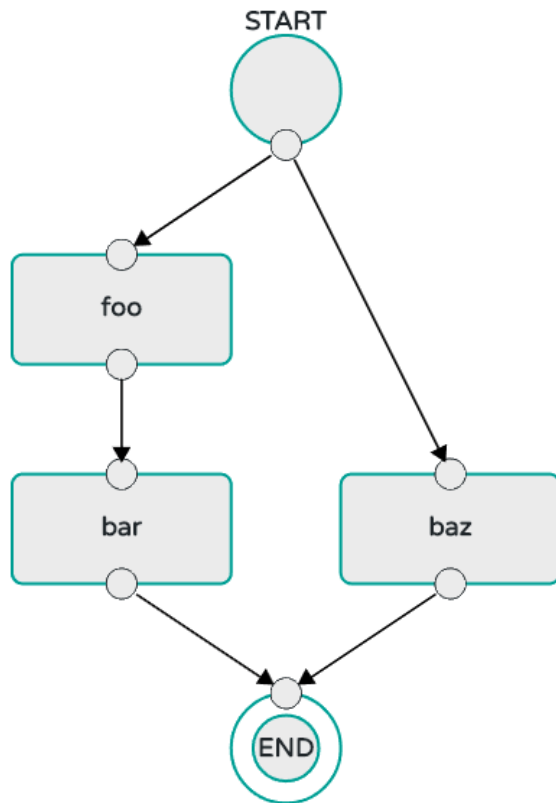


Figure 45.7. Split with conditional execution

Part VII. 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 [Chapter 21, Feature Toggles](#) for how to disable task support in Data Flow.

46. 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.

47. Tooling

It is 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 is to view them within the context of CF CLI.

48. Task Database Schema

The database schema for Task applications was changed slightly 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.

49. 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.

49.1 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.

49.2 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'
```

49.3 View Task Logs

As previously mentioned, the CL CLI 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 CF CLI as the app is running. Historic logs are not available.

49.4 List Tasks

Listing tasks is as simple as:

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

49.5 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      #
#####
```

49.6 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#
#####
```

49.7 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.

Part VIII. Dashboard

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

50. Introduction

Spring Cloud Data Flow provides a browser-based GUI and it currently includes 6 tabs:

- **Apps** Lists all available applications and provides the control to register/unregister them
- **Runtime** Provides the Data Flow cluster view with the list of all running applications
- **Streams** List, create, deploy, and destroy Stream Definitions
- **Tasks** List, create, launch and destroy Task Definitions
- **Jobs** Perform Batch Job related functions
- **Analytics** Create data visualizations for the various analytics applications

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

`http://<host>:<port>/dashboard`

For example: <http://localhost:9393/dashboard>

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



Note

The default Dashboard server port is 9393

About

Spring Cloud Data Flow is a unified, distributed, and extensible system for data ingestion, real time analytics, batch processing, and data export. The project's goal is to simplify the development of big data applications.

Dataflow Server Implementation	
Name	spring-cloud-dataflow-server-local
Version	1.0.0.BUILD-SNAPSHOT (7188a69)
Description	Local Data Flow Server

Need Help or Found an Issue?

Project Page	http://cloud.spring.io/spring-cloud-dataflow/
Sources	https://github.com/spring-cloud/spring-cloud-dataflow
Documentation	http://docs.spring.io/spring-cloud-dataflow/docs/current/reference/html/
API Docs	http://docs.spring.io/spring-cloud-dataflow/docs/current/api/
Support Forum	http://stackoverflow.com/questions/tagged/spring-cloud
Issue Tracker	https://github.com/spring-cloud/spring-cloud-dataflow/issues

Figure 50.1. The Spring Cloud Data Flow Dashboard

51. Apps

The *Apps* section of the Dashboard lists all the available applications and provides the control to register/unregister them (if applicable). It is possible to import a number of applications at once using the **Bulk Import Applications** action.

The screenshot shows the 'Apps' section of the Spring Cloud Data Flow Server dashboard. At the top, there's a navigation bar with tabs: APPS (selected), RUNTIME, STREAMS, TASKS, JOBS, ANALYTICS, and ABOUT. Below the navigation bar, the 'Apps' section is titled, and a subtitle states: 'This section lists all the available applications and provides the control to register/unregister them (if applicable).' Below this, there's a 'Quick filter' input field. The main content area shows a table of available applications. The table has columns: Name, Type, URI, and Actions. The 'Bulk Import Applications' button is highlighted in the top right of the table area.

Name	Type	URI	Actions
file	source	maven://org.springframework.cloud.stream.app:file-source-rabbit:1.0.2.RELEASE	[Search] [Delete]
ftp	source	maven://org.springframework.cloud.stream.app:ftp-source-rabbit:1.0.2.RELEASE	[Search] [Delete]
gemfire	source	maven://org.springframework.cloud.stream.app:gemfire-source-rabbit:1.0.2.RELEASE	[Search] [Delete]
gemfire-cq	source	maven://org.springframework.cloud.stream.app:gemfire-cq-source-rabbit:1.0.2.RELEASE	[Search] [Delete]
http	source	maven://org.springframework.cloud.stream.app:http-source-rabbit:1.0.2.RELEASE	[Search] [Delete]
jdbc	source	maven://org.springframework.cloud.stream.app:jdbc-source-rabbit:1.0.2.RELEASE	[Search] [Delete]
jms	source	maven://org.springframework.cloud.stream.app:jms-source-rabbit:1.0.2.RELEASE	[Search] [Delete]

Figure 51.1. List of Available Applications

51.1 Bulk Import of Applications

The bulk import applications page provides numerous options for defining and importing a set of applications in one go. For bulk import the application definitions are expected to be expressed in a properties style:

```
<type>.<name> = <coordinates>
```

For example:

```
task.timestamp=maven://org.springframework.cloud.task.app:timestamp-
task:1.2.0.RELEASE
```

```
processor.transform=maven://org.springframework.cloud.stream.app:transform-
processor-rabbit:1.2.0.RELEASE
```

At the top of the bulk import page an *Uri* can be specified that points to a properties file stored elsewhere, it should contain properties formatted as above. Alternatively, using the textbox labeled *Apps as Properties* it is possible to directly list each property string. Finally, if the properties are stored in a local file the *Select Properties File* option will open a local file browser to select the file. After setting your definitions via one of these routes, click **Import**.

At the bottom of the page there are quick links to the property files for common groups of stream apps and task apps. If those meet your needs, simply select your appropriate variant (rabbit, kafka, docker, etc) and click the **Import** action on those lines to immediately import all those applications.

The screenshot shows the 'Bulk Import Applications' page of the Spring Cloud Data Flow Server. The page has a dark header with the Spring logo and navigation links: APPS, RUNTIME, STREAMS, TASKS, JOBS, ANALYTICS, and ABOUT. The main content area is titled 'Bulk Import Applications' and contains a description: 'Import and register applications in bulk. Simply provide a URI that points to the location of the properties file where the keys are formatted as type.name and the values are the URIs of the apps. For convenience, a list of out-of-the-box Stream and Task app starters is provided below, as well.'

There are three main input methods:

- Uri:** A text input field with the placeholder '<http://url.to.properties>'. Below it, a note says 'Please provide a valid URI pointing to the respective properties file.'
- OR:** A section separator.
- Apps as Properties:** A text area with an example: 'task.timestamp=maven://o.s.cloud.task.app:timestamp-task:1.2.3.RELEASE' and 'task.spark-client=maven://o.s.cloud.task.app:spark-client-task:1.2.3.RELEASE'. Below it, a note says 'Please provide a valid properties where the keys are formatted as type.name and the values are the URIs of the apps.'

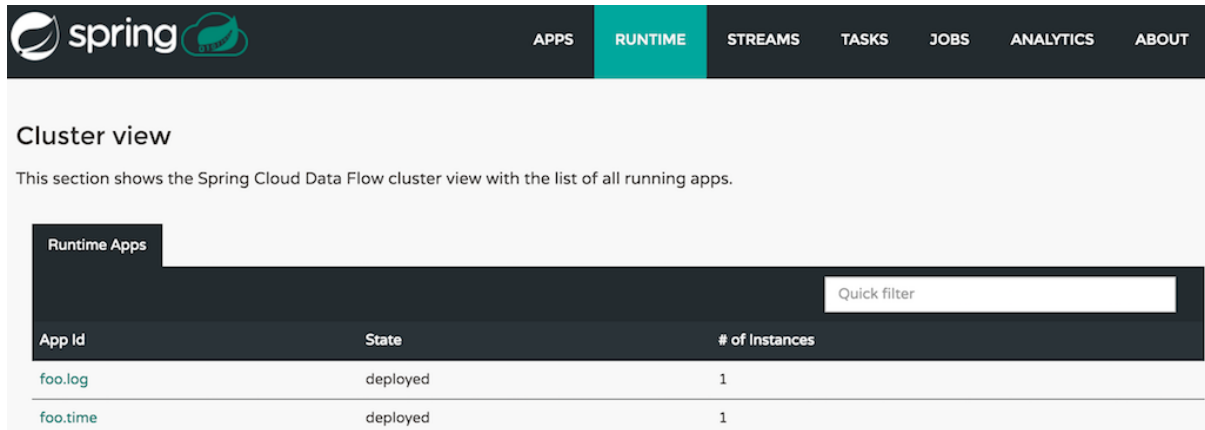
At the bottom, there is a 'Select Properties File' section with a 'Choose File' button and the text 'No file chosen'. Below this, a note says 'Please provide a text file containing properties. This will be used to populate the text area above.' There is also a checkbox labeled 'Force' with a question mark icon.

At the bottom right, there are two buttons: 'Cancel' and 'Import'.

Figure 51.2. Bulk Import Applications

52. Runtime

The *Runtime* section of the Dashboard application shows the Spring Cloud Data Flow cluster view with the list of all running applications. For each runtime app the state of the deployment and the number of deployed instances is shown. A list of the used deployment properties is available by clicking on the app id.



The screenshot shows the 'Runtime' tab selected in the dashboard. Below the navigation bar, there's a 'Cluster view' section with a description. A 'Runtime Apps' tab is active, displaying a table of applications. A 'Quick filter' input field is present on the right. The table lists two applications: 'foo.log' and 'foo.time', both in a 'deployed' state with 1 instance each.

App Id	State	# of Instances
foo.log	deployed	1
foo.time	deployed	1

Figure 52.1. List of Running Applications

53. 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**. Each row includes an arrow on the left, which can be clicked to see a visual representation of the definition. Hovering over the boxes in the visual representation will show more details about the apps including any options passed to them. In this screenshot the timer stream has been expanded to show the visual representation:

The screenshot shows the 'Definitions' tab in the Spring Cloud Data Flow Dashboard. At the top, there are buttons for 'Expand All' and 'Collapse All', and a 'Quick filter' input field. Below this is a table with columns: Name, Definition, Status, and Actions.

Name	Definition	Status	Actions
▶ minutes	:timer.time > transform --expression=payload.substring(2,4) log	deployed	Details Undeploy Deploy Destroy
▶ seconds	:timer.time > transform --expression=payload.substring(4) log	deployed	Details Undeploy Deploy Destroy
▼ timer	time --date-format=h:mm:ss log	deployed	Details Undeploy Deploy Destroy

Below the table, the 'timer' stream is expanded to show its visual representation. It consists of a 'time' component connected to a 'log' component. A zoom slider is visible in the top right corner of the visual representation area, set to 173.

Figure 53.1. List of Stream Definitions

If the **details** button is clicked the view will change to show a visual representation of that stream and also any related streams. In the above example, if clicking **details** for the timer stream, the view will change to the one shown below which clearly shows the relationship between the three streams (two of them are tapping into the timer stream).

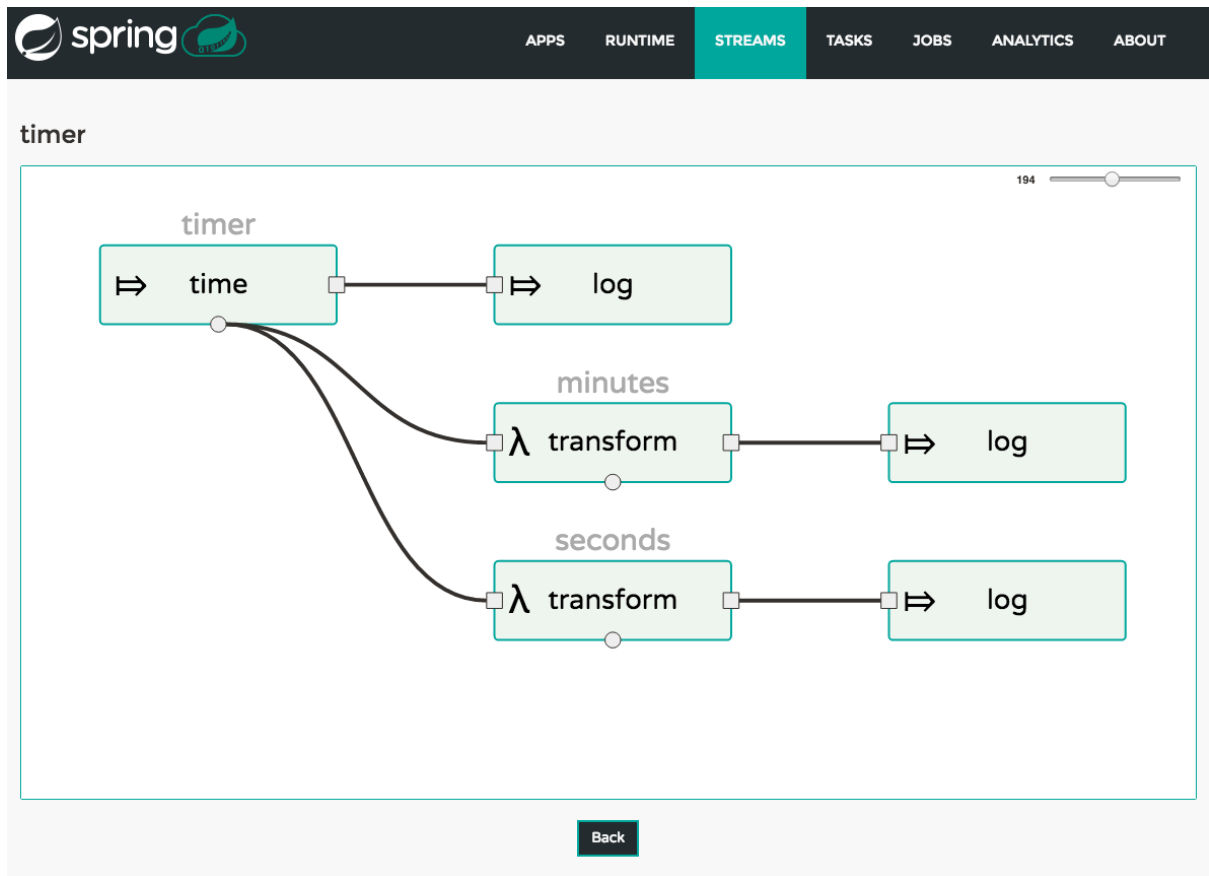


Figure 53.2. Stream Details Page

54. Create Stream

The *Create Stream* section of the Dashboard includes the [Spring Flo](#) designer tab that provides the canvas application, offering an interactive graphical interface for creating data pipelines.

In this tab, you can:

- Create, manage, and visualize stream pipelines using DSL, a graphical canvas, or both
- Write pipelines via DSL with content-assist and auto-complete
- Use auto-adjustment and grid-layout capabilities in the GUI for simpler and interactive organization of pipelines

Watch this [screencast](#) that highlights some of the "Flo for Spring Cloud Data Flow" capabilities. Spring Flo [wiki](#) includes more detailed content on core Flo capabilities.

The screenshot displays the Spring Cloud Data Flow Dashboard's 'Streams' section. The 'Create Stream' tab is active, showing a DSL editor with the following code:

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

On the left, a 'source' panel lists components: file, ftp, http, jdbc, jms, and load-gener... The main canvas, titled 'STREAM_1', shows a visual representation of the pipeline. A 'time' source component connects to three parallel 'scriptable-transform' components, each of which then connects to a 'log' sink component.

Figure 54.1. Flo for Spring Cloud Data Flow

55. Tasks

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

- Apps
- Definitions
- Executions

55.1 Apps

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



Note

You will also use this tab to create Batch Jobs.

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

Figure 55.1. List of Task Apps

On this screen you can perform the following actions:

- View details such as the task app options.
- Create a Task Definition from the respective App.

Create a Task Definition from a selected Task App

On this screen you can create a new Task Definition. As a minimum you must provide a name for the new definition. You will also have the option to specify various properties that are used during the deployment of the app.



Note

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

View Task App Details

On this page you can view the details of a selected task app, including the list of available options (properties) for that app.

55.2 Definitions

This page lists the Data Flow Task definitions and provides actions to **launch** or **destroy** those tasks. It also provides a shortcut operation to define one or more tasks using simple textual input, indicated by the **bulk define tasks** button.

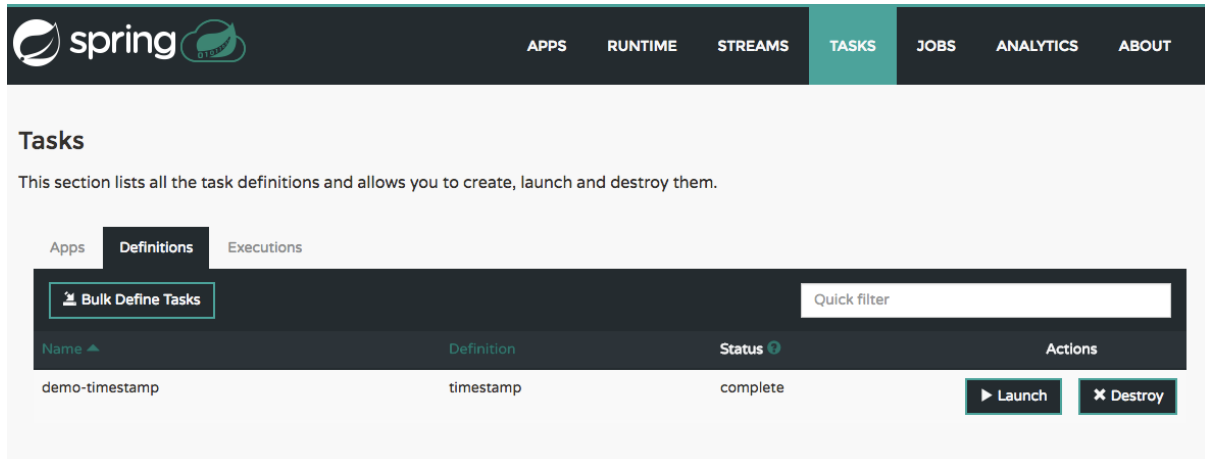


Figure 55.2. List of Task Definitions

Creating Task Definitions using the bulk define interface

After pressing **bulk define tasks**, the following screen will be shown.

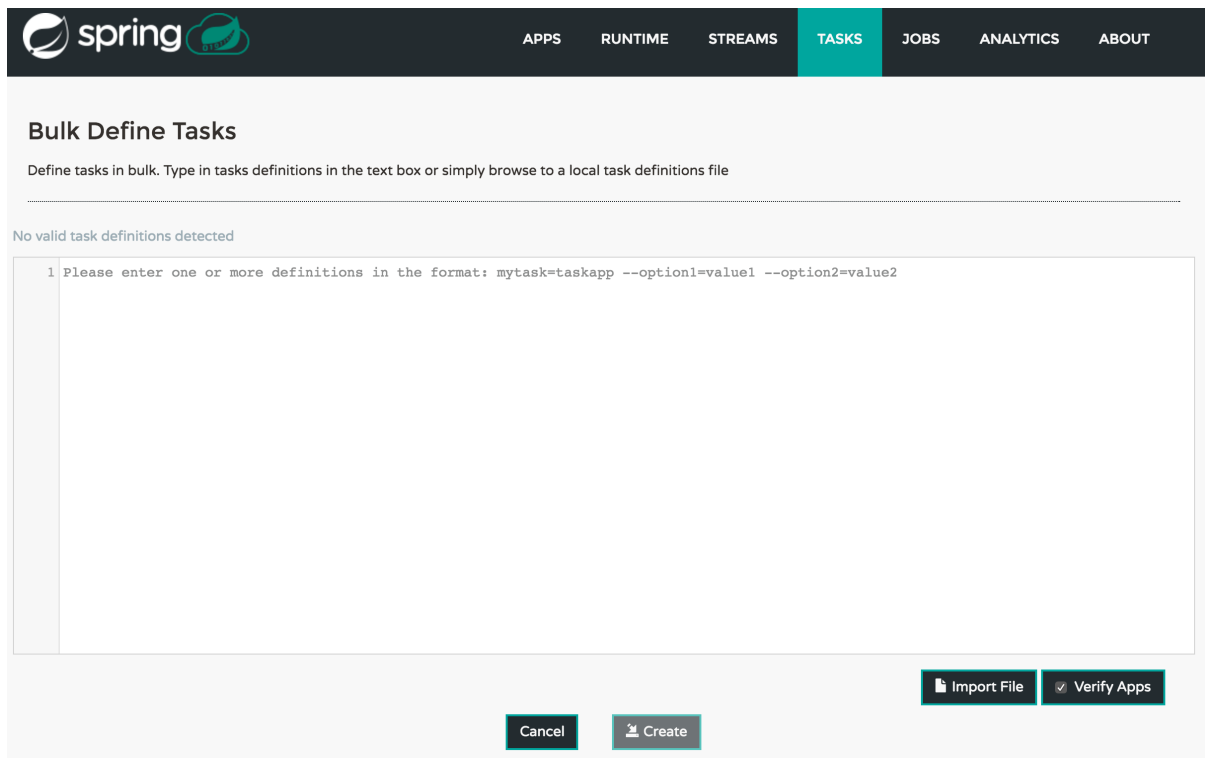


Figure 55.3. Bulk Define Tasks

It includes a textbox where one or more definitions can be entered and then various actions performed on those definitions. The required input text format for task definitions is very basic, each line should be of the form:

```
<task-definition-name> = <task-application> <options>
```

For example:

```
demo-timestamp = timestamp --format=hhmmss
```

After entering any data a validator will run asynchronously to verify both the syntax and that the application name entered is a valid application and it supports the options specified. If validation fails the editor will show the errors with more information via tooltips.

To make it easier to enter definitions into the text area, content assist is supported. Pressing **Ctrl+Space** will invoke content assist to suggest simple task names (based on the line on which it is invoked), task applications and task application options. Press ESCape to close the content assist window without taking a selection.

If the validator should not verify the applications or the options (for example if specifying non-whitelisted options to the applications) then turn off that part of validation by toggling the checkbox off on the **Verify Apps** button - the validator will then only perform syntax checking. When correctly validated, the **create** button will be clickable and on pressing it the UI will proceed to create each task definition. If there are any errors during creation then after creation finishes the editor will show any lines of input, as it cannot be used in task definitions. These can then be fixed up and creation repeated. There is an **import file** button to open a file browser on the local file system if the definitions are in a file and it is easier to import than copy/paste.



Note

Bulk loading of composed task definitions is not currently supported.

Creating Composed Task Definitions

The dashboard includes the Create Composed Task tab that provides the canvas application, offering a interactive graphical interface for creating composed tasks.

In this tab, you can:

- Create and visualize composed tasks using DSL, a graphical canvas, or both
- Use auto-adjustment and grid-layout capabilities in the GUI for simpler and interactive organization of the composed task

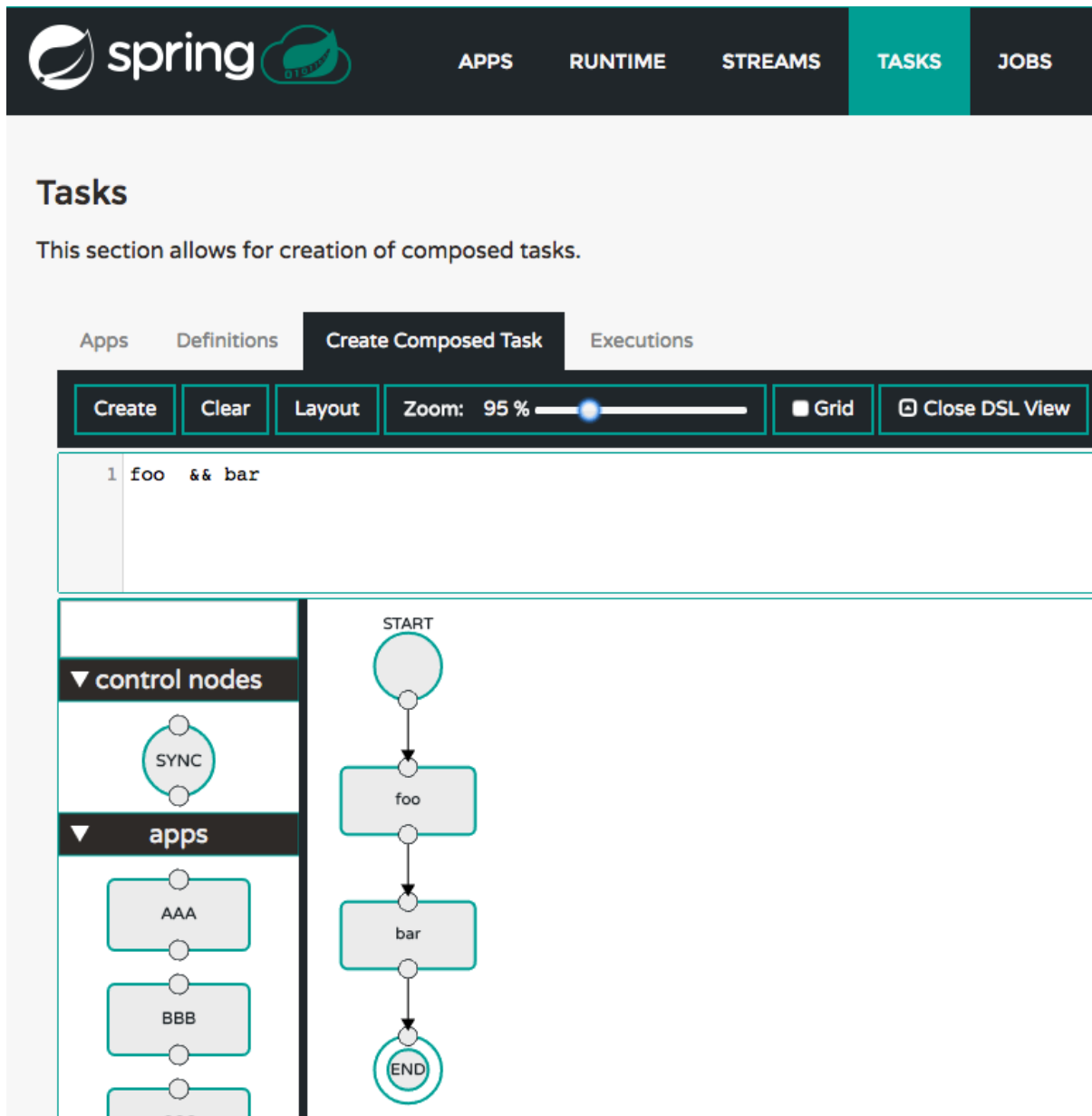


Figure 55.4. Composed Task Designer

Launching Tasks

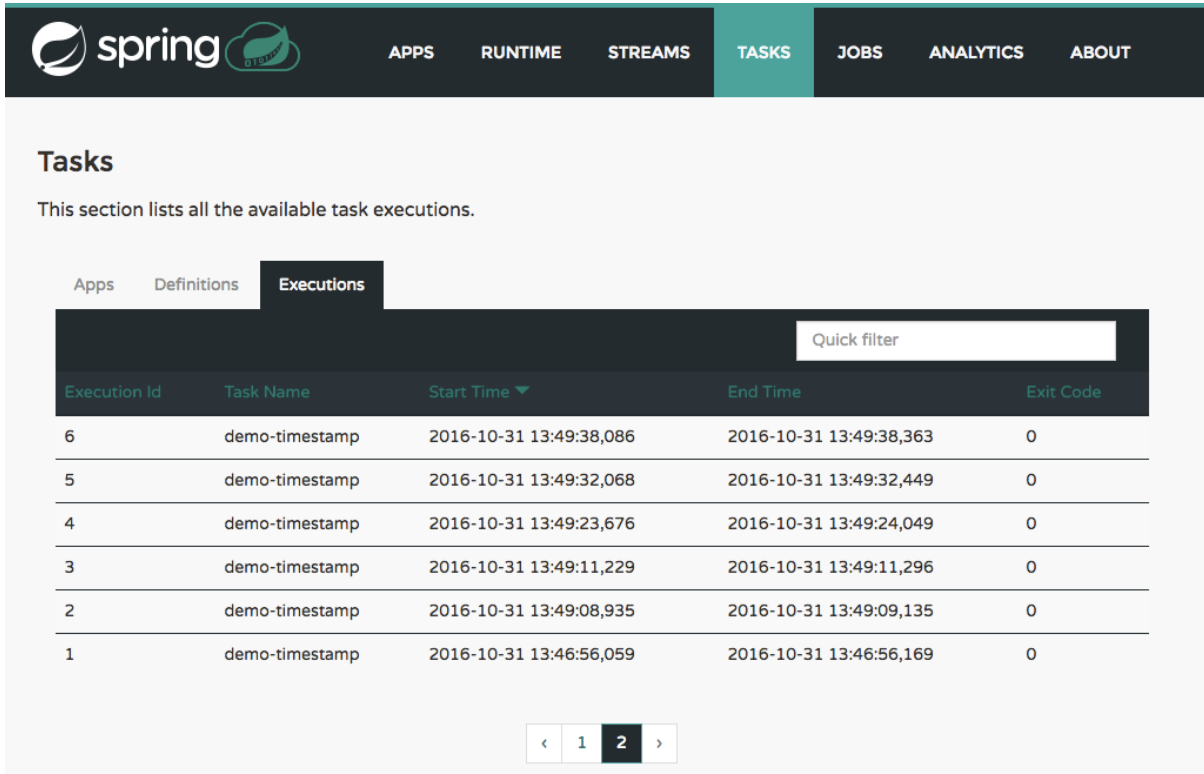
Once the task definition is created, they can be launched through the Dashboard as well. Navigate to the **Definitions** tab. Select the Task you want to launch by pressing **Launch**.

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

- Parameter Key
- Parameter Value

Task parameters are not typed.

55.3 Executions



The screenshot shows the 'Tasks' section of the Spring Cloud Data Flow Server interface. The top navigation bar includes 'spring' logo, 'APPS', 'RUNTIME', 'STREAMS', 'TASKS' (highlighted), 'JOBS', 'ANALYTICS', and 'ABOUT'. Below the navigation bar, the 'Tasks' section is titled 'Tasks' and includes a subtitle: 'This section lists all the available task executions.' There are three tabs: 'Apps', 'Definitions', and 'Executions' (selected). A 'Quick filter' input field is located above the table. The table lists task executions with columns: 'Execution Id', 'Task Name', 'Start Time', 'End Time', and 'Exit Code'. The data shows six executions, all for 'demo-timestamp', with start times ranging from 2016-10-31 13:46:56,059 to 2016-10-31 13:49:38,086 and end times ranging from 2016-10-31 13:46:56,169 to 2016-10-31 13:49:38,363. All exit codes are 0. A pagination control at the bottom shows '< 1 2 >' with '2' highlighted.

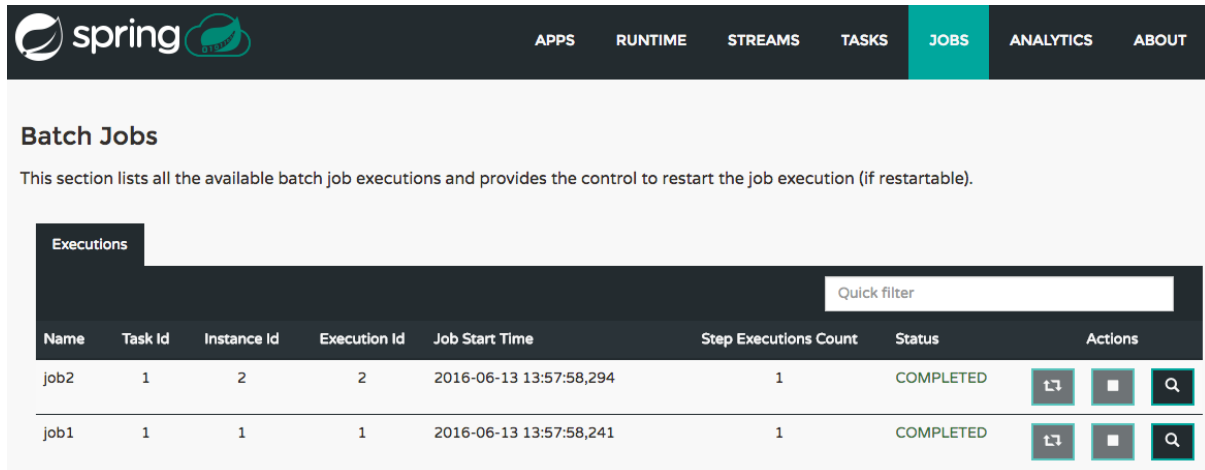
Execution Id	Task Name	Start Time	End Time	Exit Code
6	demo-timestamp	2016-10-31 13:49:38,086	2016-10-31 13:49:38,363	0
5	demo-timestamp	2016-10-31 13:49:32,068	2016-10-31 13:49:32,449	0
4	demo-timestamp	2016-10-31 13:49:23,676	2016-10-31 13:49:24,049	0
3	demo-timestamp	2016-10-31 13:49:11,229	2016-10-31 13:49:11,296	0
2	demo-timestamp	2016-10-31 13:49:08,935	2016-10-31 13:49:09,135	0
1	demo-timestamp	2016-10-31 13:46:56,059	2016-10-31 13:46:56,169	0

Figure 55.5. List of Task Executions

56. Jobs

The *Jobs* section of the Dashboard allows you to inspect **Batch Jobs**. The main section of the screen provides a list of Job Executions. **Batch Jobs** are **Tasks** that were executing one or more **Batch Job**. As such each Job Execution has a back reference to the **Task Execution Id** (Task Id).

In case of a failed job, you can also restart the task. When dealing with long-running Batch Jobs, you can also request to stop it.









Name	Task Id	Instance Id	Execution Id	Job Start Time	Step Executions Count	Status	Actions
job2	1	2	2	2016-06-13 13:57:58,294	1	COMPLETED	  
job1	1	1	1	2016-06-13 13:57:58,241	1	COMPLETED	  

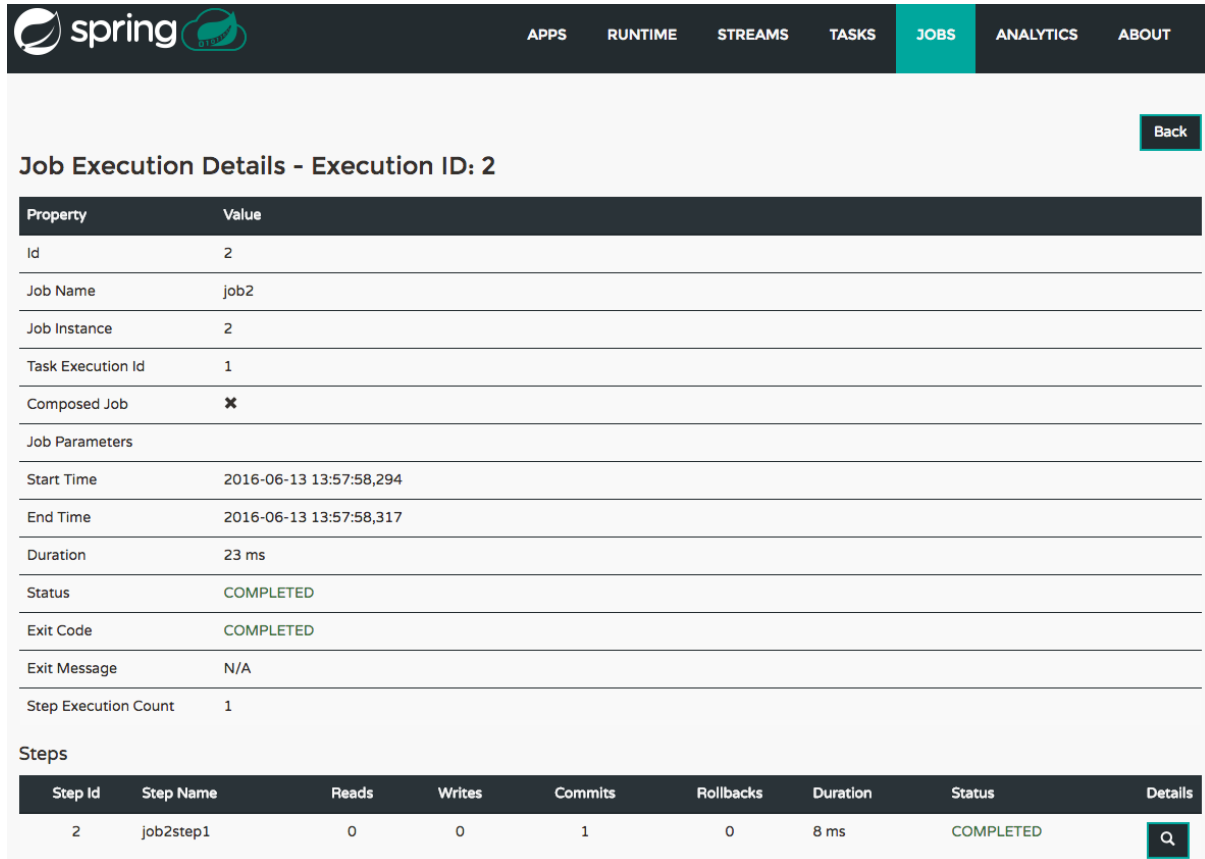
Figure 56.1. List of Job Executions

56.1 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



The screenshot shows the 'Jobs' tab in the Spring Cloud Data Flow Server interface. The 'Job Execution Details' section for 'Execution ID: 2' is displayed. It includes a 'Back' button and a table of properties. Below the properties table, there is a 'Steps' section with a table of executed steps. The first step, 'job2step1', is shown as completed.

Property	Value
Id	2
Job Name	job2
Job Instance	2
Task Execution Id	1
Composed Job	✖
Job Parameters	
Start Time	2016-06-13 13:57:58,294
End Time	2016-06-13 13:57:58,317
Duration	23 ms
Status	COMPLETED
Exit Code	COMPLETED
Exit Message	N/A
Step Execution Count	1

Step Id	Step Name	Reads	Writes	Commits	Rollbacks	Duration	Status	Details
2	job2step1	0	0	1	0	8 ms	COMPLETED	

Figure 56.2. 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.

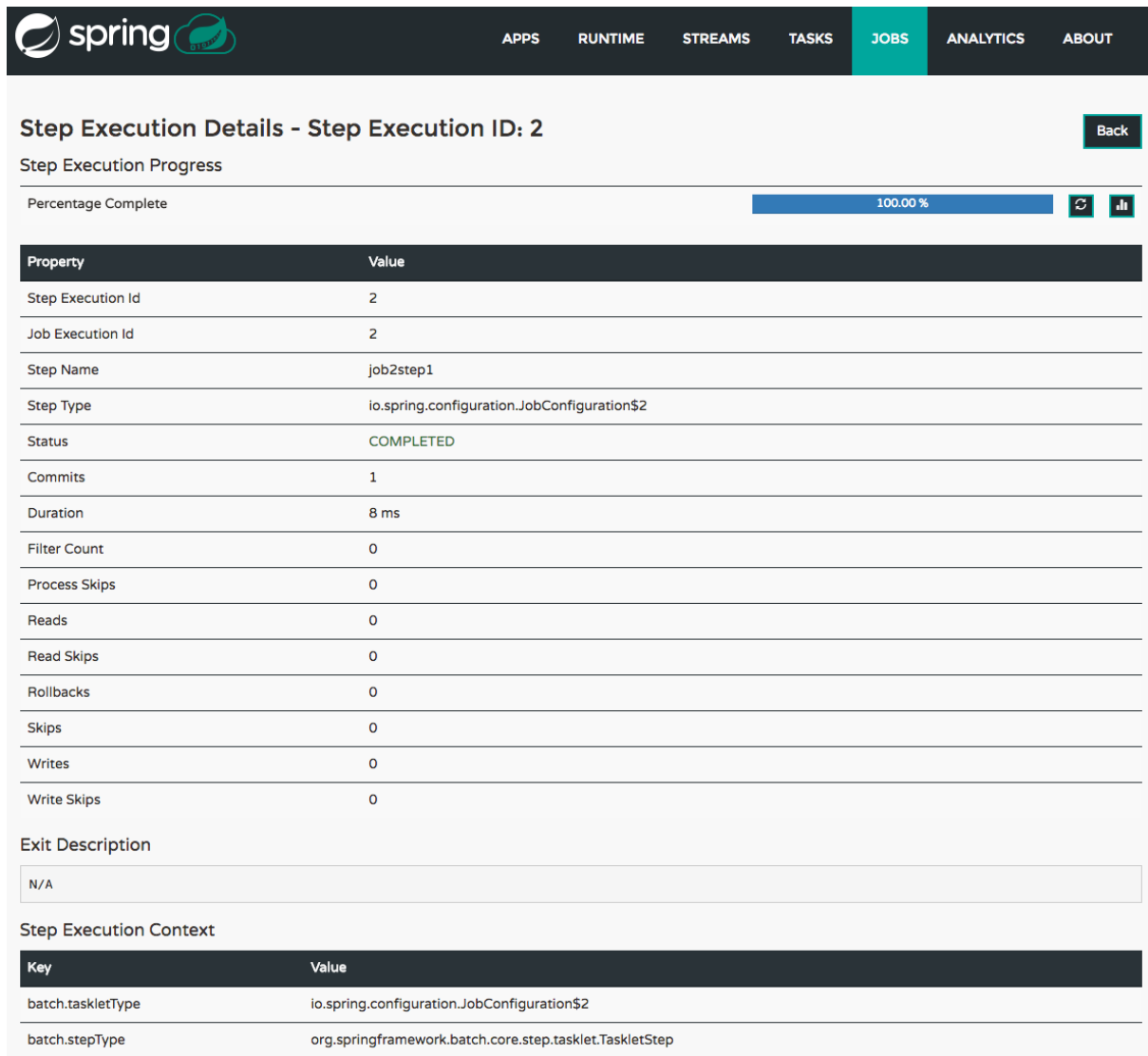


Figure 56.3. Step Execution History

57. Analytics

The *Analytics* section of the Dashboard provided data visualization capabilities for the various analytics applications available in *Spring Cloud Data Flow*:

- Counters
- Field-Value Counters
- Aggregate Counters

For example, if you create a stream with a [Counter](#) application, you can now easily create the corresponding graph from within the **Dashboard** tab:

1. Under `Metric Type`, select `Counters` from the select box
2. Under `Stream`, select `tweetcount`
3. Under `Visualization`, select the desired chart option, `Bar Chart`

Using the icons to the right, you can add additional charts to the Dashboard, re-arrange the order of created dashboards or remove data visualizations.

Part IX. ‘How-to’ guides

This section provides answers to some common ‘how do I do that...’ type of questions that often arise when using Spring Cloud Data Flow.

If you are having a specific problem that we don’t cover here, you might want to check out stackoverflow.com to see if someone has already provided an answer; this is also a great place to ask new questions (please use the `spring-cloud-dataflow` tag).

We’re also more than happy to extend this section; If you want to add a ‘how-to’ you can send us a [pull request](#).

58. Configure Maven Properties

You can set the maven properties such as local maven repository location, remote maven repositories and their authentication credentials including the proxy server properties via commandline properties when starting the Dataflow server or using the `SPRING_APPLICATION_JSON` environment property for the Dataflow server.

The remote maven repositories need to be configured explicitly if the apps are resolved using maven repository except for local Data Flow server. The other Data Flow server implementations (that use maven resources for app artifacts resolution) have no default value for remote repositories. The local server has repo.spring.io/libs-snapshot as the default remote repository.

To pass the properties as commandline options:

```
$ java -jar <dataflow-server>.jar --maven.localRepository=mylocal
--maven.remote-repositories.repo1.url=https://repo1
--maven.remote-repositories.repo1.auth.username=repoluser
--maven.remote-repositories.repo1.auth.password=repolpass
--maven.remote-repositories.repo2.url=https://repo2 --maven.proxy.host=proxyhost
--maven.proxy.port=9018 --maven.proxy.auth.username=proxyuser
--maven.proxy.auth.password=proxypass
```

or, using the `SPRING_APPLICATION_JSON` environment property:

```
export SPRING_APPLICATION_JSON='{ "maven": { "local-repository": "local", "remote-repositories":
{ "repo1": { "url": "https://repo1", "auth": { "username": "repoluser", "password": "repolpass" } },
"repo2": { "url": "https://repo2" } }, "proxy": { "host": "proxyhost", "port":
9018, "auth": { "username": "proxyuser", "password": "proxypass" } } }'}
```

Formatted JSON:

```
SPRING_APPLICATION_JSON='{
  "maven": {
    "local-repository": "local",
    "remote-repositories": {
      "repo1": {
        "url": "https://repo1",
        "auth": {
          "username": "repoluser",
          "password": "repolpass"
        }
      },
      "repo2": {
        "url": "https://repo2"
      }
    },
    "proxy": {
      "host": "proxyhost",
      "port": 9018,
      "auth": {
        "username": "proxyuser",
        "password": "proxypass"
      }
    }
  }
}'
```



Note

Depending on Spring Cloud Data Flow server implementation, you may have to pass the environment properties using the platform specific environment-setting

capabilities. For instance, in Cloud Foundry, you'd be passing them as `cf set-env SPRING_APPLICATION_JSON`.

59. Logging

Spring Cloud Data Flow is built upon several Spring projects, but ultimately the dataflow-server is a Spring Boot app, so the logging techniques that apply to any [Spring Boot](#) application are applicable here as well.

While troubleshooting, following are the two primary areas where enabling the DEBUG logs could be useful.

59.1 Deployment Logs

Spring Cloud Data Flow builds upon [Spring Cloud Deployer](#) SPI and the platform specific dataflow-server uses the respective [SPI implementations](#). Specifically, if we were to troubleshoot deployment specific issues; such as the network errors, it'd be useful to enable the DEBUG logs at the underlying deployer and the libraries used by it.

1. For instance, if you'd like to enable DEBUG logs for the [local-deployer](#), you'd be starting the server with following.

```
$ java -jar <dataflow-server>.jar --logging.level.org.springframework.cloud.deployer.spi.local=DEBUG
```

(where, `org.springframework.cloud.deployer.spi.local` is the global package for everything local-deployer related)

2. For instance, if you'd like to enable DEBUG logs for the [cloudfoundry-deployer](#), you'd be setting the following environment variable and upon restaging the dataflow-server, we will see more logs around request, response and the elaborate stack traces (*upon failures*). The cloudfoundry-deployer uses [cf-java-client](#), so we will have to enable DEBUG logs for this library.

```
$ cf set-env dataflow-server JAVA_OPTS '-Dlogging.level.cloudfoundry-client=DEBUG'
$ cf restage dataflow-server
```

(where, `cloudfoundry-client` is the global package for everything `cf-java-client` related)

3. If there's a need to review Reactor logs, which is used by the `cf-java-client`, then the following would be helpful.

```
$ cf set-env dataflow-server JAVA_OPTS '-Dlogging.level.cloudfoundry-client=DEBUG -Dlogging.level.reactor.ipc.netty=DEBUG'
$ cf restage dataflow-server
```

(where, `reactor.ipc.netty` is the global package for everything `reactor-netty` related)



Note

Similar to the `local-deployer` and `cloudfoundry-deployer` options as discussed above, there are equivalent settings available for Apache YARN, Apache Mesos and Kubernetes variants, too. Check out the respective [SPI implementations](#) to find out more details about the packages to configure for logging.

59.2 Application Logs

The streaming applications in Spring Cloud Data Flow are Spring Boot applications and they can be independently setup with logging configurations.

For instance, if you'd have to troubleshoot the header and payload specifics that are being passed around source, processor and sink channels, you'd be deploying the stream with the following options.

```
dataflow:>stream create foo --definition "http --logging.level.org.springframework.integration=DEBUG  
/ transform --logging.level.org.springframework.integration=DEBUG / log --  
logging.level.org.springframework.integration=DEBUG" --deploy
```

(where, *org.springframework.integration* is the global package for everything Spring Integration related, which is responsible for messaging channels)

These properties can also be specified via deployment properties when deploying the stream.

```
dataflow:>stream deploy foo --properties "app.*.logging.level.org.springframework.integration=DEBUG"
```

60. Frequently asked questions

In this section, we will review the frequently discussed questions in Spring Cloud Data Flow.

60.1 Advanced SpEL expressions

One of the powerful features of SpEL expressions is [functions](#). Spring Integration provides `jsonPath()` and `xpath()` out-of-the-box [SpEL-functions](#), if appropriate libraries are in the classpath. All the provided Spring Cloud Stream application starters are supplied with the `json-path` and `spring-integration-xml` jars, thus we can use those SpEL-functions in Spring Cloud Data Flow streams whenever expressions are possible. For example we can transform JSON-aware payload from the HTTP request using some `jsonPath()` expression:

```
dataflow:>stream create jsonPathTransform --definition "http | transform --
expression=#jsonPath(payload,'$.price') | log" --deploy
...
dataflow:> http post --target http://localhost:8080 --data {"symbol":"SCDF","price":72.04}
dataflow:> http post --target http://localhost:8080 --data {"symbol":"SCDF","price":72.06}
dataflow:> http post --target http://localhost:8080 --data {"symbol":"SCDF","price":72.08}
```

In this sample we apply `jsonPath` for the incoming payload to extract just only the `price` field value. Similar syntax can be used with `splitter` or `filter` expression options. Actually any available SpEL-based option has access to the built-in SpEL-functions. For example we can extract some value from JSON data to calculate the `partitionKey` before sending output to the Binder:

```
dataflow:>stream deploy foo --
properties "deployer.transform.count=2,app.transform.producer.partitionKeyExpression=#jsonPath(payload,'$.symbol')"
```

The same syntax can be applied for `xpath()` SpEL-function when you deal with XML data. Any other custom SpEL-function can also be used, but for this purpose you should build a library with the `@Configuration` class containing an appropriate `SpelFunctionFactoryBean` `@Bean` definition. The target Spring Cloud Stream application starter should be re-packaged to supply such a custom extension via built-in Spring Boot `@ComponentScan` mechanism or auto-configuration hook.

60.2 How to use JDBC-sink?

The JDBC-sink can be used to insert message payload data into a relational database table. By default, it inserts the entire payload into a table named after the `jdbc.table-name` property, and if it is not set, by default the application expects to use a table with the name `messages`. To alter this behavior, the JDBC sink accepts [several options](#) that you can pass using the `--foo=bar` notation in the stream, or change globally. The JDBC sink has a `jdbc.initialize` property that if set to `true` will result in the sink creating a table based on the specified configuration when the it starts up. If that `initialize` property is `false`, which is the default, you will have to make sure that the table to use is already available.

A stream definition using `jdbc` sink relying on all defaults with MySQL as the backing database looks like the following. In this example, the system time is persisted in MySQL for every second.

```
dataflow:>stream create --name mydata --definition "time | jdbc --spring.datasource.url=jdbc:mysql://
localhost:3306/test --spring.datasource.username=root --spring.datasource.password=root --
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver" --deploy
```

For this to work, you'd have to have the following table in the MySQL database.

```
CREATE TABLE test.messages
(
```

```
payload varchar(255)
);
```

```
mysql> desc test.messages;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| payload | varchar(255) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> select * from test.messages;
+-----+-----+
| payload |
+-----+-----+
| 04/25/17 09:10:04 |
| 04/25/17 09:10:06 |
| 04/25/17 09:10:07 |
| 04/25/17 09:10:08 |
| 04/25/17 09:10:09 |
| ..... |
| ..... |
| ..... |
+-----+-----+
```

60.3 How to use multiple message-binders?

For situations where the data is consumed and processed between two different message brokers, Spring Cloud Data Flow provides easy to override global configurations, out-of-the-box [bridge-processor](#), and DSL primitives to build these type of topologies.

Let's assume we have data queueing up in RabbitMQ (e.g., *queue = fooRabbit*) and the requirement is to consume all the payloads and publish them to Apache Kafka (e.g., *topic = barKafka*), as the destination for downstream processing.

Follow the global application of [configurations](#) to define multiple binder configurations.

```
# Apache Kafka Global Configurations (i.e., identified by "kafka")
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafka.type=kafka
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafka.environment.spring.cloud.stream.kafka
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafka.environment.spring.cloud.stream.kafka

# RabbitMQ Global Configurations (i.e., identified by "rabbit1")
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.rabbit1.type=rabbit
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.host=localhost
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.port=5672
```



Note

In this example, both the message brokers are running locally and reachable at `localhost` with respective ports.

These properties can be supplied in a ".properties" file that is accessible to the server directly or via `config-server`.

```
java -jar spring-cloud-dataflow-server-local/target/spring-cloud-dataflow-server-local-1.1.4.RELEASE.jar
--spring.config.location=<PATH-TO-FILE>/foo.properties
```

Spring Cloud Data Flow internally uses `bridge-processor` to directly connect different named channel destinations. Since we are publishing and subscribing from two different messaging systems, you'd have to build the `bridge-processor` with both RabbitMQ and Apache Kafka binders in the classpath. To do that, head over to start-scs.cfapps.io/ and select `Bridge Processor, Kafka binder`

starter, and Rabbit binder starter as the dependencies and follow the patching procedure described in the [reference guide](#). Specifically, for the bridge-processor, you'd have to import the BridgeProcessorConfiguration provided by the starter.

Once you have the necessary adjustments, you can build the application. Let's register the name of the application as multiBinderBridge.

```
dataflow:>app register --type processor --name multiBinderBridge --uri file:///<PATH-TO-FILE>/multipleBinderBridge-0.0.1-SNAPSHOT.jar
```

It is time to create a stream definition with the newly registered processor application.

```
dataflow:>stream create fooRabbitToBarKafka --definition ":fooRabbit > multiBinderBridge --spring.cloud.stream.bindings.input.binder=rabbit1 --spring.cloud.stream.bindings.output.binder=kafka1 > :barKafka" --deploy
```



Note

Since we are to consume messages from RabbitMQ (*i.e., identified by `rabbit1`*) and then publish the payload to Apache Kafka (*i.e., identified by `kafka1`*), we are supplying them as input and output channel settings respectively.



Note

The queue `fooRabbit` in RabbitMQ is where the stream is consuming events from and the topic `barKafka` in Apache Kafka is where the data is finally landing.

Part X. Appendices

Appendix A. Migrating from Spring XD to Spring Cloud Data Flow

A.1 Terminology Changes

Old	New
XD-Admin	Server (<i>implementations</i> : local, cloud foundry, apache yarn, kubernetes, and apache mesos)
XD-Container	N/A
Modules	Applications
Admin UI	Dashboard
Message Bus	Binders
Batch / Job	Task

A.2 Modules to Applications

If you have custom Spring XD modules, you'd have to refactor them to use Spring Cloud Stream and Spring Cloud Task annotations, with updated dependencies and built as normal Spring Boot "applications".

Custom Applications

- Spring XD's stream and batch modules are refactored into [Spring Cloud Stream](#) and [Spring Cloud Task](#) application-starters, respectively. These applications can be used as the reference while refactoring Spring XD modules
- There are also some samples for [Spring Cloud Stream](#) and [Spring Cloud Task](#) applications for reference
- If you'd like to create a brand new custom application, use the getting started guide for [Spring Cloud Stream](#) and [Spring Cloud Task](#) applications and as well as review the development [guide](#)
- Alternatively, if you'd like to patch any of the out-of-the-box stream applications, you can follow the procedure [here](#)

Application Registration

- Custom Stream/Task application requires being installed to a maven repository for Local, YARN, and CF implementations or as docker images, when deploying to Kubernetes and Mesos. Other than maven and docker resolution, you can also resolve application artifacts from `http`, `file`, or as `hdfs` coordinates
- Unlike Spring XD, you do not have to upload the application bits while registering custom applications anymore; instead, you're expected to [register](#) the application coordinates that are hosted in the maven repository or by other means as discussed in the previous bullet

- By default, none of the out-of-the-box applications are preloaded already. It is intentionally designed to provide the flexibility to register app(s), as you find appropriate for the given use-case requirement
- Depending on the binder choice, you can manually add the appropriate binder dependency to build applications specific to that binder-type. Alternatively, you can follow the Spring Initializr [procedure](#) to create an application with binder embedded in it

Application Properties

- counter-sink:
 - The peripheral `redis` is not required in Spring Cloud Data Flow. If you intend to use the `counter-sink`, then `redis` becomes required, and you're expected to have your own running `redis` cluster
- field-value-counter-sink:
 - The peripheral `redis` is not required in Spring Cloud Data Flow. If you intend to use the `field-value-counter-sink`, then `redis` becomes required, and you're expected to have your own running `redis` cluster
- aggregate-counter-sink:
 - The peripheral `redis` is not required in Spring Cloud Data Flow. If you intend to use the `aggregate-counter-sink`, then `redis` becomes required, and you're expected to have your own running `redis` cluster

A.3 Message Bus to Binders

Terminology wise, in Spring Cloud Data Flow, the message bus implementation is commonly referred to as binders.

Message Bus

Similar to Spring XD, there's an abstraction available to extend the binder interface. By default, we take the opinionated view of [Apache Kafka](#) and [RabbitMQ](#) as the production-ready binders and are available as GA releases.

Binders

Selecting a binder is as simple as providing the right binder dependency in the classpath. If you're to choose Kafka as the binder, you'd register stream applications that are pre-built with Kafka binder in it. If you were to create a custom application with Kafka binder, you'd add the following dependency in the classpath.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka</artifactId>
  <version>1.0.2.RELEASE</version>
</dependency>
```

- Spring Cloud Stream supports [Apache Kafka](#), [RabbitMQ](#) and experimental [Google PubSub](#) and [Solace JMS](#). All binder implementations are maintained and managed in their individual repositories
- Every Stream/Task application can be built with a binder implementation of your choice. All the out-of-the-box applications are pre-built for both Kafka and Rabbit and they're readily available for use as

maven artifacts [[Spring Cloud Stream](#) / [Spring Cloud Task](#) or docker images [[Spring Cloud Stream](#) / [Spring Cloud Task](#) Changing the binder requires selecting the right binder [dependency](#). Alternatively, you can download the pre-built application from this version of [Spring Initializr](#) with the desired “binder-starter” dependency

Named Channels

Fundamentally, all the messaging channels are backed by pub/sub semantics. Unlike Spring XD, the messaging channels are backed only by `topics` or `topic-exchange` and there's no representation of `queues` in the new architecture.

- `${xd.module.index}` is not supported anymore; instead, you can directly interact with named destinations
- `stream.index` changes to `:<stream-name>.<label/app-name>`
 - *for instance:* `ticktock.0` changes to `:ticktock.time`
- “topic/queue” prefixes are not required to interact with named-channels
 - *for instance:* `topic:foo` changes to `:foo`
 - *for instance:* `stream create stream1 --definition ":foo > log"`

Directed Graphs

If you're building non-linear streams, you could take advantage of named destinations to build directed graphs.

for instance, in Spring XD:

```
stream create f --definition "queue:foo > transform --expression=payload+'-foo' | log" --deploy
stream create b --definition "queue:bar > transform --expression=payload+'-bar' | log" --deploy
stream create r --definition "http | router --expression=payload.contains('a')?'queue:foo':'queue:bar'"
--deploy
```

for instance, in Spring Cloud Data Flow:

```
stream create f --definition ":foo > transform --expression=payload+'-foo' | log" --deploy
stream create b --definition ":bar > transform --expression=payload+'-bar' | log" --deploy
stream create r --definition "http | router --expression=payload.contains('a')?'foo':'bar'" --deploy
```

A.4 Batch to Tasks

A Task by definition, is any application that does not run forever, including Spring Batch jobs, and they end/stop at some point. Task applications can be majorly used for on-demand use-cases such as database migration, machine learning, scheduled operations etc. Using [Spring Cloud Task](#), users can build Spring Batch jobs as microservice applications.

- Spring Batch [jobs](#) from Spring XD are being refactored to Spring Boot applications a.k.a Spring Cloud Task [applications](#)
- Unlike Spring XD, these “Tasks” don't require explicit deployment; instead, a task is ready to be launched directly once the definition is declared

A.5 Shell/DSL Commands

Old Command	New Command
module upload	app register / app import
module list	app list
module info	app info
admin config server	dataflow config server
job create	task create
job launch	task launch
job list	task list
job status	task status
job display	task display
job destroy	task destroy
job execution list	task execution list
runtime modules	runtime apps

A.6 REST-API

Old API	New API
/modules	/apps
/runtime/modules	/runtime/apps
/runtime/modules/{moduleId}	/runtime/apps/{appId}
/jobs/definitions	/task/definitions
/jobs/deployments	/task/deployments

A.7 UI / Flo

The Admin-UI is now renamed as Dashboard. The URI for accessing the Dashboard is changed from localhost:9393/admin-ui to localhost:9393/dashboard

- (New) Apps: Lists all the registered applications that are available for use. This view includes informational details such as the URI and the properties supported by each application. You can also register/unregister applications from this view
- Runtime: Container changes to Runtime. The notion of `xd-container` is gone, replaced by out-of-the-box applications running as autonomous Spring Boot applications. The Runtime tab displays the applications running in the runtime platforms (*implementations*: cloud foundry, apache yarn, apache mesos, or kubernetes). You can click on each application to review relevant details about the application such as where it is running with, and what resources etc.

- [Spring Flo](#) is now an OSS product. Flo for Spring Cloud Data Flow's "Create Stream", the designer-tab comes pre-built in the Dashboard
- (New) Tasks:
 - The sub-tab "Modules" is renamed to "Apps"
 - The sub-tab "Definitions" lists all the Task definitions, including Spring Batch jobs that are orchestrated as Tasks
 - The sub-tab "Executions" lists all the Task execution details similar to Spring XD's Job executions

A.8 Architecture Components

Spring Cloud Data Flow comes with a significantly simplified architecture. In fact, when compared with Spring XD, there are less peripherals that are necessary to operationalize Spring Cloud Data Flow.

ZooKeeper

ZooKeeper is not used in the new architecture.

RDBMS

Spring Cloud Data Flow uses an RDBMS instead of Redis for stream/task definitions, application registration, and for job repositories. The default configuration uses an embedded H2 instance, but Oracle, DB2, SqlServer, MySQL/MariaDB, PostgreSQL, H2, and HSQLDB databases are supported. To use Oracle, DB2 and SqlServer you will need to create your own Data Flow Server using [Spring Initializr](#) and add the appropriate JDBC driver dependency.

Redis

Running a Redis cluster is only required for analytics functionality. Specifically, when the `counter-sink`, `field-value-counter-sink`, or `aggregate-counter-sink` applications are used, it is expected to also have a running instance of Redis cluster.

Cluster Topology

Spring XD's `xd-admin` and `xd-container` server components are replaced by stream and task applications themselves running as autonomous Spring Boot applications. The applications run natively on various platforms including Cloud Foundry, Apache YARN, Apache Mesos, or Kubernetes. You can develop, test, deploy, scale +/-, and interact with (Spring Boot) applications individually, and they can evolve in isolation.

A.9 Central Configuration

To support centralized and consistent management of an application's configuration properties, [Spring Cloud Config](#) client libraries have been included into the Spring Cloud Data Flow server as well as the Spring Cloud Stream applications provided by the Spring Cloud Stream App Starters. You can also [pass common application properties](#) to all streams when the Data Flow Server starts.

A.10 Distribution

Spring Cloud Data Flow is a Spring Boot application. Depending on the platform of your choice, you can download the respective release uber-jar and deploy/push it to the runtime platform (cloud foundry,

apache yarn, kubernetes, or apache mesos). For example, if you're running Spring Cloud Data Flow on Cloud Foundry, you'd download the Cloud Foundry server implementation and do a `cf push` as explained in the [reference guide](#).

A.11 Hadoop Distribution Compatibility

The `hdfs-sink` application builds upon Spring Hadoop 2.4.0 release, so this application is compatible with following Hadoop distributions.

- Cloudera - cdh5
- Pivotal Hadoop - phd30
- Hortonworks Hadoop - hdp24
- Hortonworks Hadoop - hdp23
- Vanilla Hadoop - hadoop26
- Vanilla Hadoop - 2.7.x (default)

A.12 YARN Deployment

Spring Cloud Data Flow can be deployed and used with Apache YARN in two different ways.

- Deploy the server [directly](#) in a YARN cluster
- Leverage Apache Ambari [plugin to provision](#) Spring Cloud Data Flow as a service

A.13 Use Case Comparison

Let's review some use-cases to compare and contrast the differences between Spring XD and Spring Cloud Data Flow.

Use Case #1

(It is assumed both XD and SCDF distributions are already downloaded)

Description: Simple `ticktock` example using local/singlenode.

Spring XD	Spring Cloud Data Flow
Start <code>xd-singlenode</code> server from CLI # <code>xd-singlenode</code>	Start a binder of your choice Start <code>local-server</code> implementation of SCDF from the CLI # <code>java -jar spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar</code>
Start <code>xd-shell</code> server from the CLI # <code>xd-shell</code>	Start <code>dataflow-shell</code> server from the CLI

Spring XD	Spring Cloud Data Flow
	<pre># java -jar spring-cloud-dataflow-shell-1.0.0.BUILD-SNAPSHOT.jar</pre>
Create ticktock stream <code>xd:>stream create ticktock --definition "time log" --deploy</code>	Create ticktock stream <code>dataflow:>stream create ticktock --definition "time log" --deploy</code>
Review ticktock results in the xd-singlenode server console	Review ticktock results by tailing the ticktock.log/stdout_log application logs

Use Case #2

(It is assumed both XD and SCDF distributions are already downloaded)

Description: Stream with custom module/application.

Spring XD	Spring Cloud Data Flow
Start xd-singlenode server from CLI <pre># xd-singlenode</pre>	Start a binder of your choice Start local-server implementation of SCDF from the CLI <pre># java -jar spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar</pre>
Start xd-shell server from the CLI <pre># xd-shell</pre>	Start dataflow-shell server from the CLI <pre># java -jar spring-cloud-dataflow-shell-1.0.0.BUILD-SNAPSHOT.jar</pre>
Register custom "processor" module to transform payload to a desired format <code>xd:>module upload --name toupper --type processor --file <CUSTOM_JAR_FILE_LOCATION></code>	Register custom "processor" application to transform payload to a desired format <code>dataflow:>app register --name toupper --type processor --uri <MAVEN_URI_COORDINATES></code>
Create a stream with custom module <code>xd:>stream create testupper --definition "http toupper log" --deploy</code>	Create a stream with custom application <code>dataflow:>stream create testupper --definition "http toupper log" --deploy</code>
Review results in the xd-singlenode server console	Review results by tailing the testupper.log/stdout_log application logs

Use Case #3

(It is assumed both XD and SCDF distributions are already downloaded)

Description: Simple batch-job.

Spring XD	Spring Cloud Data Flow
Start xd-singlenode server from CLI <pre># xd-singlenode</pre>	Start local-server implementation of SCDF from the CLI <pre># java -jar spring-cloud-dataflow-server-local-1.0.0.BUILD-SNAPSHOT.jar</pre>
Start xd-shell server from the CLI <pre># xd-shell</pre>	Start dataflow-shell server from the CLI <pre># java -jar spring-cloud-dataflow-shell-1.0.0.BUILD-SNAPSHOT.jar</pre>
Register custom “batch-job” module <pre>xd:>module upload --name simple-batch --type job --file <CUSTOM_JAR_FILE_LOCATION></pre>	Register custom “batch-job” as task application <pre>dataflow:>app register --name simple-batch --type task --uri <MAVEN_URI_COORDINATES></pre>
Create a job with custom batch-job module <pre>xd:>job create batchtest --definition "simple-batch"</pre>	Create a task with custom batch-job application <pre>dataflow:>task create batchtest --definition "simple-batch"</pre>
Deploy job <pre>xd:>job deploy batchtest</pre>	NA
Launch job <pre>xd:>job launch batchtest</pre>	Launch task <pre>dataflow:>task launch batchtest</pre>
Review results in the xd-singlenode server console as well as Jobs tab in UI (executions sub-tab should include all step details)	Review results by tailing the batchtest/stdout_log application logs as well as Task tab in UI (executions sub-tab should include all step details)

Appendix B. Building

B.1 Basic Compile and Test

To build the source you will need to install JDK 1.8.

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 ($\geq 3.3.3$) 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.2 Documentation

There is a "full" profile that will generate documentation. You can build just the documentation by executing

```
$ ./mvnw package -DskipTests=true -P full -pl spring-cloud-dataflow-server-cloudfoundry-docs -am
```

B.3 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.