Spring Cloud Data Flow Reference Guide

Sabby Anandan · Marius Bogoevici · Eric Bottard · Mark Fisher · Ilayaperumal Gopinathan · Gunnar Hillert · Mark Pollack · Patrick Peralta · Glenn Renfro · Thomas Risberg · Dave Syer · David Turanski · Janne Valkealahti · Oleg Zhurakousky · Jay Bryant · Vinicius Carvalho · Chris Schaefer

Table of Contents

Version 2.0.0.BUILD-SNAPSHOT

Preface

1. About the documentation

The documentation for this release is available in HTML.

The latest copy of the Spring Cloud Data Flow reference guide can be found here.

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

---

2. Getting help

Having trouble with Spring Cloud Data Flow, We would like to help!

- Ask a question. We monitor stackoverflow.com for questions tagged with `spring-cloud-dataflow`.
- Report bugs with Spring Cloud Data Flow at github.com/spring-cloud/spring-cloud-dataflow/issues.

All of Spring Cloud Data Flow is open source, including the documentation! If you find problems with the docs or if you just want to improve them, please get involved.

Getting Started

3. Getting Started - Local

3.1. System Requirements

You need Java 8 to run and to build you need to have Maven.

You need to have an RDBMS for storing stream definition and deployment properties and task/batch job states. By default, the Data Flow server uses embedded H2 database for this purpose but you can easily configure the server to use another external database.

You also need to have Redis running if you are running any streams that involve analytics applications. Redis may also be required to run the unit/integration tests.

For the deployed streams applications communicate, either RabbitMQ or Kafka needs to be installed.

For deploying, upgrading and rolling back applications in Streams at runtime, you should install the Spring Cloud Skipper server as well.

3.2. Getting Started with Docker Compose

A Docker Compose file is provided to quickly bring up Spring Cloud Data Flow and its dependencies without having to obtain them manually. When running, a composed system includes the latest GA release of Spring Cloud Data Flow Server using the Kafka binder for communication and Skipper for stream deployment. Docker Compose is required and it's recommended to use the latest version.

1. Download the Spring Cloud Data Flow Server Docker Compose file:

```
$ wget https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/master/spring-cloud-dataflow-server/docker-compose.yml
```

If the `wget` command is unavailable, `curl` or another platform specific utility may be used. Alternatively navigate to https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/master/spring-cloud-dataflow-server/docker-compose.yml in a web browser and save the contents. Ensure the downloaded filename is `docker-compose.yml`.

1. Start Docker Compose

   In the directory where you downloaded `docker-compose.yml`, start the system, as follows:

```
$ export DATAFLOW_VERSION=latest
$ export SKIPPER_VERSION=2.0.0.M1
$ docker-compose up
```

   The `docker-compose.yml` file defines `DATAFLOW_VERSION` and `SKIPPER_VERSION` variables so that those can be changed easily. The above commands first set the `DATAFLOW_VERSION` and `SKIPPER_VERSION` to use in the environment and then `docker-compose` is started.

   A shorthand version which only exposes the `DATAFLOW_VERSION` and `SKIPPER_VERSION` variables to the `docker-compose` process rather than setting it in the environment would be as follows:

```
$ DATAFLOW_VERSION=latest SKIPPER_VERSION=2.0.0.M1 docker-compose up
```

   When using Windows, environment variables are defined by using the `set` command. To start the system on Windows, enter the following commands:

```
C:\ set DATAFLOW_VERSION=latest
C:\ set SKIPPER_VERSION=2.0.0.M1
C:\ docker-compose up
```

By default Docker Compose will use locally available images. For example when using the `latest` tag, execute `docker-compose pull` prior to `docker-compose up` to ensure the latest image is downloaded.

2. Launch the Spring Cloud Data Flow Dashboard

   Spring Cloud Data Flow will be ready for use once the `docker-compose` command stops emitting log messages. At this time, in your browser navigate to the Spring Cloud Data Flow Dashboard. By default the latest GA releases of Stream and Task applications will be imported automatically.

3. Create a Stream

   To create a stream, first navigate to the "Streams" menu link then click the "Create Stream" link. Enter `time | log` into the "Create Stream" textarea then click the "CREATE STREAM" button. Enter "ticktock" for the stream name and click the "Deploy Stream(s) checkbox as show in the following image:

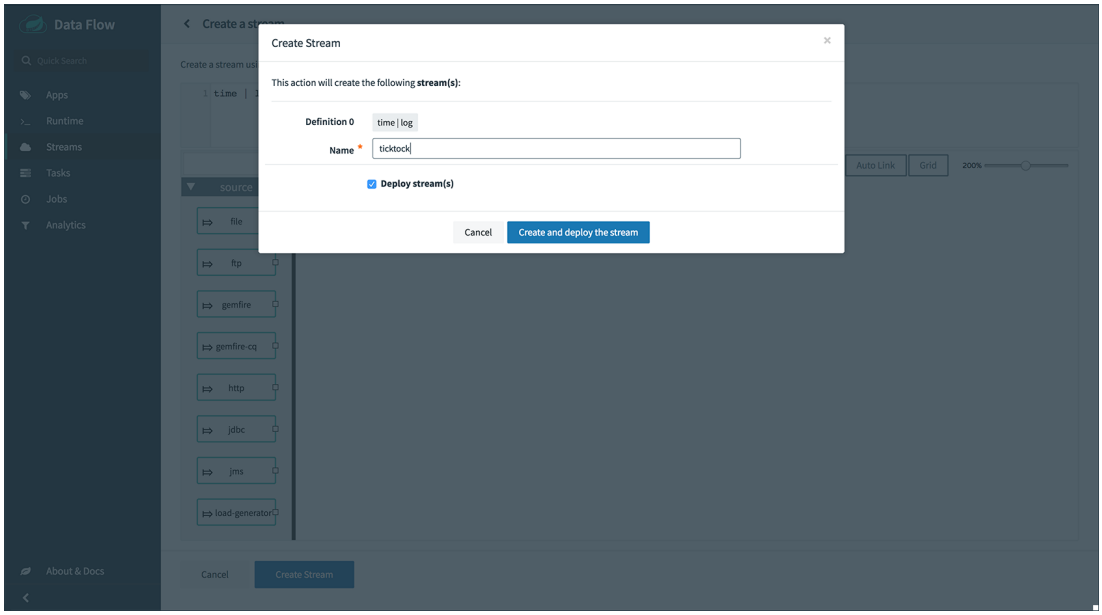


Figure 1. Creating a Stream

Then click "OK" which will return back to the Definitions page. The stream will be in "deploying" status and move to "deployed" when finished. You may need to refresh your browser to see the updated status.

4. View Stream Logs

   To view the stream logs, navigate to the "Runtime" menu link and click the "ticktock.log" link. Copy the path in the "stdout" text box on the dashboard and in another console type:

   ```
   $ docker exec -it skipper tail -f /path/from/stdout/textbox/in/dashboard
   ```

   You should now see the output of the log sink, printing a timestamp once per second. Press CTRL+c to end the `tail`.

5. Delete a Stream

   To delete the stream, first navigate to the "Streams" menu link in the dashboard then click the checkbox on the "ticktock" row. Click the "DESTROY ALL 1 SELECTED STREAMS" button and then "YES" to destroy the stream.

6. Destroy the Quick Start environment

   To destroy the Quick Start environment, in another console from where the `docker-compose.yml` is located, type as follows:

   ```
   $ docker-compose down
   ```

Some stream applications may open a port, for example `http --server.port=`. By default, a port range of `9000-9010` is exposed from the container to the host. If you would like to change this range, modify the `ports` block of the `dataflow-server` service in the `docker-compose.yml` file.

3.2.1. Docker Compose Customization

Out of the box Spring Cloud Data Flow will use the H2 embedded database for storing state, Kafka for communication and no analytics. Customizations can be made to these components by editing the `docker-compose.yml` file as described below.

1. To use MySQL rather than the H2 embedded database, add the following configuration under the `services` section:

   ```
   mysql:
     image: mariadb:10.2
     environment:
       MYSQL_DATABASE: dataflow
       MYSQL_USER: root
       MYSQL_ROOT_PASSWORD: rootpw
     expose:
       - 3306
   ```

   The following entries need to be added to the `environment` block of the `dataflow-server` service definition:

   ```
       - spring.datasource.url=jdbc:mysql://mysql:3306/dataflow
       - spring.datasource.username=root
       - spring.datasource.password=rootpw
       - spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
   ```

2. To use RabbitMQ instead of Kafka for communication, replace the following configuration under the `services` section:

```
kafka:
  image: wurstmeister/kafka:1.1.0
  expose:
    - "9092"
  environment:
    - KAFKA_ADVERTISED_PORT=9092
    - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
    - KAFKA_ADVERTISED_HOST_NAME=kafka
zookeeper:
  image: wurstmeister/zookeeper
  expose:
    - "2181"
```

   With:

```
rabbitmq:
  image: rabbitmq:3.7
  expose:
    - "5672"
```

   In the `dataflow-server` services configuration block, add the following `environment` entry:

```
    - spring.cloud.dataflow.applicationProperties.stream.spring.rabbitmq.host=rabbitmq
```

   And finally, modify the `app-import` service definition `command` attribute to replace `bit.ly/Celsius-SR3-stream-applications-kafka-10-maven` with `bit.ly/Celsius-SR3-stream-applications-rabbit-maven`.

3. To enable analytics using redis as a backend, add the following configuration under the `services` section:

```
redis:
  image: redis:2.8
  expose:
    - "6379"
```

   Then add the following entries to the `environment` block of the `dataflow-server` service definition:

```
    - spring.cloud.dataflow.applicationProperties.stream.spring.redis.host=redis
    - spring.redis.host=redis
```

4. To enable Metrics support, the following modifications need to be made.

   First, after the `zookeeper` service definition add Redis as described inEnable analytics using Redis.

   Then add the metrics collector after the `skipper-server` service definition:

```
metrics-collector:
  image: springcloud/metrics-collector-kafka:2.0.0.RELEASE
  environment:
    - spring.cloud.stream.kafka.binder.brokers=kafka:9092
    - spring.cloud.stream.kafka.binder.zkNodes=zookeeper:2181
    - spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration
  expose:
    - "8080"
```

5. To enable `app starters` registration directly from the host machine you have to mount the source host folders to the `dataflow-server` container. For example, if the `my-app.jar` is in the `/foo/bar/apps` folder on your host machine, then add the following `volumes` block to the `dataflow-server` service definition:

```
dataflow-server:
  image: springcloud/spring-cloud-dataflow-server:${DATAFLOW_VERSION}
  container_name: dataflow-server
  ports:
    - "9393:9393"
  environment:
    - spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.brokers=kafka:9092
    - spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.zkNodes=zookeeper:2181
  volumes:
    - /foo/bar/apps:/root/apps
```

   Later provides access to the `my-app.jar` (and the other files in the folder) from within container's `/root/apps/` folder. Check the compose-file reference for furthether configuration details.

> The explicit volume mounting couples the docker-compose to your host's file system, limiting the portability to other machines and OS-es. Unlike `docker`, the `docker-compose` doesn't allow volume mounting from the command line (e.g. no `-v` like parameter). Instead you can define a placeholder environment variable such as `HOST_APP_FOLDER` in place of the hardcoded path: `-${HOST_APP_FOLDER}:/root/apps` and set this variable before starting the docker-compose.

Once the host folder is mounted, you can register the app starters (from `/root/apps`), with the SCDF Shell or Dashboard, using the `file://` URI schema:

```
dataflow:>app register --type source --name my-app --uri file://root/apps/my-app.jar
```

Use also `--metadata-uri` if the metadata jar is available in the /root/apps.

To access host's local maven repository from within the `dataflow-server` container, you should mount host maven local repository (defaults to `~/.m2` for OSX and Linux and `C:\Documents and Settings{your-username}\.m2` for Windows) to a `dataflow-server` volume called `/root/.m2/`. For MacOS or Linux host machines this looks like this:

```
  dataflow-server:
  .........
    volumes:
      - ~/.m2:/root/.m2
```

Now you can use the `maven://` URI schema and maven coordinates to resolve jars installed in the host's maven repository:

```
dataflow:>app register --type processor --name pose-estimation --uri
maven://org.springframework.cloud.stream.app:pose-estimation-processor-rabbit:2.0.2.BUILD-SNAPSHOT --metadata-uri
maven://org.springframework.cloud.stream.app:pose-estimation-processor-rabbit:jar:metadata:2.0.2.BUILD-SNAPSHOT --
force
```

This approach allow you to share jars build and installed on the host machine (e.g. `mvn clean install`) directly with the dataflow-server container.

One can also pre-register the apps directly in the docker-compose. For every pre-registered app starer, add an additional `wget` statement to the `app-import` block configuration:

```
  app-import:
    image: alpine:3.7
    command: >
      /bin/sh -c "
        ....
        wget -qO- 'http://dataflow-server:9393/apps/source/my-app' --post-data='uri=file:/root/apps/my-
app.jar&metadata-uri=file:/root/apps/my-app-metadata.jar';
        echo 'My custom apps imported'"
```

Check the SCDF REST API for further details.

3.3. Getting Started with Manual Installation

1. Download the Spring Cloud Data Flow Server and Shell apps:

```
wget https://repo.spring.io/snapshot/org/springframework/cloud/spring-cloud-dataflow-server/2.0.0.BUILD-
SNAPSHOT/spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar

wget https://repo.spring.io/snapshot/org/springframework/cloud/spring-cloud-dataflow-shell/2.0.0.BUILD-
SNAPSHOT/spring-cloud-dataflow-shell-2.0.0.BUILD-SNAPSHOT.jar
```

2. Download Skipper when Stream features are enabled, since Data Flow delegates to Skipper for those features.

```
wget https://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-skipper-server/2.0.0.M1/spring-cloud-skipp

wget https://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-skipper-shell/2.0.0.M1/spring-cloud-skippe
```

3. Launch Skipper (Required unless the Stream features are disabled and the Spring Cloud Data Flow runs in Task mode only)

   In the directory where you downloaded Skipper, run the server using `java -jar`, as follows:

```
$ java -jar spring-cloud-skipper-server-2.0.0.M1.jar
```

4. Launch the Data Flow Server

   In the directory where you downloaded Data Flow, run the server using `java -jar`, as follows:

```
$ java -jar spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar
```

   If Skipper and the Data Flow server are not running on the same host, set the configuration property `spring.cloud.skipper.client.serverUri` to the location of Skipper, e.g.

```
$ java -jar spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar --
spring.cloud.skipper.client.serverUri=http://192.51.100.1:7577/api
```

5. Launch the Data Flow Shell, as follows:

```
$ java -jar spring-cloud-dataflow-shell-2.0.0.BUILD-SNAPSHOT.jar
```

If the Data Flow Server and shell are not running on the same host, you can also point the shell to the Data Flow server URL using the `dataflow config server` command when in the shell's interactive mode.

If the Data Flow Server and shell are not running on the same host, point the shell to the Data Flow server URL, as follows:

```
server-unknown:>dataflow config server http://198.51.100.0
Successfully targeted http://198.51.100.0
dataflow:>
```

Alternatively, pass in the command line option `--dataflow.uri`. The shell's command line option `--help` shows what is available.

3.4. Deploying Streams

1. Register Stream Apps

By default, the application registry is empty. As an example, register two applications, `http` and `log`, that communicate by using RabbitMQ.

```
dataflow:>app register --name http --type source --uri maven://org.springframework.cloud.stream.app:http-source-
rabbit:1.2.0.RELEASE
Successfully registered application 'source:http'

dataflow:>app register --name log --type sink --uri maven://org.springframework.cloud.stream.app:log-sink-
rabbit:1.1.0.RELEASE
Successfully registered application 'sink:log'
```

For more details, such as how to register applications that are based on docker containers or use Kafka as the messaging middleware, review the section on how to register applications.

Depending on your environment, you may need to configure the Data Flow Server to point to a custom Maven repository location or configure proxy settings. See [configuration-maven] for more information.

2. Create a stream

Use the `stream create` command to create a stream with a `http` source and a `log` sink and deploy it:

```
dataflow:> stream create --name httptest --definition "http --server.port=9000 | log" --deploy
```

You need to wait a little while, until the apps are actually deployed successfully, before posting data. Look in the log file of the Data Flow server for the location of the log files for the `http` and `log` applications. Use the `tail` command on the log file for each application to verify that the application has started.

Now post some data, as shown in the following example:

```
dataflow:> http post --target http://localhost:9000 --data "hello world"
```

Check to see if `hello world` ended up in log files for the `log` application. The location of the log file for the `log` application will be shown in the Data Flow server's log.

When deploying locally, each app (and each app instance, in case of `count > 1`) gets a dynamically assigned `server.port`, unless you explicitly assign one with `--server.port=x`. In both cases, this setting is propagated as a configuration property that overrides any lower-level setting that you may have used (for example, in `application.yml` files).

Following sections show Streams can be updated and rolled back by using the Local Data Flow server and Skipper. If you execute the Unix `jps` command you can see the two java processes running, as shown in the following listing:

```
$ jps | grep rabbit
12643 log-sink-rabbit-1.1.0.RELEASE.jar
12645 http-source-rabbit-1.2.0.RELEASE.jar
```

3.4.1. Upgrading

Before we start upgrading the log-sink version to 1.2.0.RELEASE, we will have to register that version in the app registry.

```
dataflow:>app register --name log --type sink --uri maven://org.springframework.cloud.stream.app:log-sink-
rabbit:1.2.0.RELEASE
Successfully registered application 'sink:log'
```

Since we are using the local server, we need to set the port to a different value (9002) than the currently running log sink's value of 9000 to avoid a conflict. While we are at it, we update log level to be `ERROR`. To do so, we create a YAML file, named `local-log-update.yml`, with the following contents:

```
version:
  log: 1.2.0.RELEASE
app:
  log:
    server.port: 9002
    log.level: ERROR
```

Now we update the Stream, as follows:

```
dataflow:> stream update --name httptest --propertiesFile /home/mpollack/local-log-update.yml
Update request has been sent for the stream 'httptest'
```

By executing the Unix `jps` command, you can see the two java processes running, but now the log application is version 1.2.0.RELEASE, as shown in the following listing:

```
$ jps | grep rabbit
22034 http-source-rabbit-1.2.0.RELEASE.jar
22031 log-sink-rabbit-1.1.0.RELEASE.jar
```

Now you can look in the log file of the Skipper server. To do so, use the following command:

`cd` to the directory `/tmp/spring-cloud-dataflow-5262910238261867964/httptest-1511749222274/httptest.log-v2` and `tail -f stdout_0.log`

You should see log entries similar to the following:

```
INFO 12591 --- [  StateUpdate-1] o.s.c.d.spi.local.LocalAppDeployer       : Deploying app with deploymentId httptest.log-v2
   Logs will be in /tmp/spring-cloud-dataflow-5262910238261867964/httptest-1511749222274/httptest.log-v2
INFO 12591 --- [  StateUpdate-1] o.s.c.s.s.d.strategies.HealthCheckStep   : Waiting for apps in release httptest-v2 to be h
INFO 12591 --- [  StateUpdate-1] o.s.c.s.s.d.s.HandleHealthCheckStep      : Release httptest-v2 has been DEPLOYED
INFO 12591 --- [  StateUpdate-1] o.s.c.s.s.d.s.HandleHealthCheckStep      : Apps in release httptest-v2 are healthy.
```

Now you can post a message to the http source at port `9000`, as follows:

```
dataflow:> http post --target http://localhost:9000 --data "hello world upgraded"
```

The log message is now at the error level, as shown in the following example:

```
ERROR 22311 --- [http.httptest-1] log-sink  : hello world upgraded
```

If you query the `/info` endpoint of the application, you can also see that it is at version `1.2.0.RELEASE`, as shown in the following example:

```
$ curl http://localhost:9002/info
{"app":{"description":"Spring Cloud Stream Log Sink Rabbit Binder Application","name":"log-sink-
rabbit","version":"1.2.0.RELEASE"}}
```

Force upgrade of a Stream

When upgrading a stream, the --force option can be used to deploy new instances of currently deployed applications even if no applicaton or deployment properties have changed. This behavior is needed in the case when configuration information is obtained by the application itself at startup time, for example from Spring Cloud Config Server. You can specify which applications to force upgrade by using the option --app-names. If you do not specify any application names, all the applications will be force upgraded. You can specify --force and --app-names options together with --properties or --propertiesFile options.

Overriding properties during Stream update

The properties that are passed during stream update are added on top of the existing properties for the same stream.

For instance, the stream `ticktock` is deployed without any explicit properties as follows:

```
dataflow:>stream create --name ticktock --definition "time | log --name=mylogger"
Created new stream 'ticktock'

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

```
Deployment request has been sent for stream 'ticktock'
```

```
dataflow:>stream manifest --name ticktock
"apiVersion": "skipper.spring.io/v1"
"kind": "SpringCloudDeployerApplication"
"metadata":
  "name": "time"
"spec":
  "resource": "maven://org.springframework.cloud.stream.app:time-source-rabbit"
  "resourceMetadata": "maven://org.springframework.cloud.stream.app:time-source-rabbit:jar:metadata:1.3.1.RELEASE"
  "version": "1.3.1.RELEASE"
  "applicationProperties":
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "time"
    "spring.cloud.stream.metrics.key": "ticktock.time.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.output.producer.requiredGroups": "ticktock"
    "spring.cloud.stream.metrics.properties":
"spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*"
    "spring.cloud.stream.bindings.output.destination": "ticktock.time"
    "spring.cloud.dataflow.stream.name": "ticktock"
    "spring.cloud.dataflow.stream.app.type": "source"
  "deploymentProperties":
    "spring.cloud.deployer.group": "ticktock"
---
"apiVersion": "skipper.spring.io/v1"
"kind": "SpringCloudDeployerApplication"
"metadata":
  "name": "log"
"spec":
  "resource": "maven://org.springframework.cloud.stream.app:log-sink-rabbit"
  "resourceMetadata": "maven://org.springframework.cloud.stream.app:log-sink-rabbit:jar:metadata:1.3.1.RELEASE"
  "version": "1.3.1.RELEASE"
  "applicationProperties":
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "log"
    "spring.cloud.stream.metrics.key": "ticktock.log.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.input.group": "ticktock"
    "log.name": "mylogger"
    "spring.cloud.stream.metrics.properties":
"spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*"
    "spring.cloud.dataflow.stream.name": "ticktock"
    "spring.cloud.dataflow.stream.app.type": "sink"
    "spring.cloud.stream.bindings.input.destination": "ticktock.time"
  "deploymentProperties":
    "spring.cloud.deployer.group": "ticktock"
```

In the second update, we try to add a new property for `log` application `foo2=bar2`.

```
dataflow:>stream update --name ticktock --properties app.log.foo2=bar2
Update request has been sent for the stream 'ticktock'

dataflow:>stream manifest --name ticktock
"apiVersion": "skipper.spring.io/v1"
"kind": "SpringCloudDeployerApplication"
"metadata":
  "name": "time"
"spec":
  "resource": "maven://org.springframework.cloud.stream.app:time-source-rabbit"
  "resourceMetadata": "maven://org.springframework.cloud.stream.app:time-source-rabbit:jar:metadata:1.3.1.RELEASE"
  "version": "1.3.1.RELEASE"
  "applicationProperties":
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "time"
    "spring.cloud.stream.metrics.key": "ticktock.time.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.output.producer.requiredGroups": "ticktock"
    "spring.cloud.stream.metrics.properties":
"spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*"
    "spring.cloud.stream.bindings.output.destination": "ticktock.time"
    "spring.cloud.dataflow.stream.name": "ticktock"
    "spring.cloud.dataflow.stream.app.type": "source"
  "deploymentProperties":
    "spring.cloud.deployer.group": "ticktock"
---
"apiVersion": "skipper.spring.io/v1"
"kind": "SpringCloudDeployerApplication"
"metadata":
  "name": "log"
"spec":
  "resource": "maven://org.springframework.cloud.stream.app:log-sink-rabbit"
  "resourceMetadata": "maven://org.springframework.cloud.stream.app:log-sink-rabbit:jar:metadata:1.3.1.RELEASE"
  "version": "1.3.1.RELEASE"
  "applicationProperties":
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "log"
    "spring.cloud.stream.metrics.key": "ticktock.log.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.input.group": "ticktock"
    "log.name": "mylogger"
    "spring.cloud.stream.metrics.properties":
"spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*"
    "spring.cloud.dataflow.stream.name": "ticktock"
    "spring.cloud.dataflow.stream.app.type": "sink"
    "foo2": "bar2" (1)
    "spring.cloud.stream.bindings.input.destination": "ticktock.time"
  "deploymentProperties":
    "spring.cloud.deployer.count": "1"
    "spring.cloud.deployer.group": "ticktock"
```

```
dataflow:>stream list
╔═══════════╤═══════════════════════════════════════╤═══════════════════════════════════════╗
║Stream Name│           Stream Definition           │                Status                 ║
╠═══════════╪═══════════════════════════════════════╪═══════════════════════════════════════╣
║ticktock   │time | log --log.name=mylogger --foo2=bar2│The stream has been successfully deployed║
╚═══════════╧═══════════════════════════════════════╧═══════════════════════════════════════╝
```

1. Property `foo2=bar2` is applied for the `log` application.

Now, when we add another property `foo3=bar3` to `log` application, this new property is added on top of the existing properties for the stream `ticktock`.

```
dataflow:>stream update --name ticktock --properties app.log.foo3=bar3
Update request has been sent for the stream 'ticktock'

dataflow:>stream manifest --name ticktock
"apiVersion": "skipper.spring.io/v1"
"kind": "SpringCloudDeployerApplication"
"metadata":
  "name": "time"
"spec":
  "resource": "maven://org.springframework.cloud.stream.app:time-source-rabbit"
  "resourceMetadata": "maven://org.springframework.cloud.stream.app:time-source-rabbit:jar:metadata:1.3.1.RELEASE"
  "version": "1.3.1.RELEASE"
  "applicationProperties":
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "time"
    "spring.cloud.stream.metrics.key": "ticktock.time.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.output.producer.requiredGroups": "ticktock"
    "spring.cloud.stream.metrics.properties":
"spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*"
    "spring.cloud.stream.bindings.output.destination": "ticktock.time"
    "spring.cloud.dataflow.stream.name": "ticktock"
    "spring.cloud.dataflow.stream.app.type": "source"
  "deploymentProperties":
    "spring.cloud.deployer.group": "ticktock"
---
"apiVersion": "skipper.spring.io/v1"
"kind": "SpringCloudDeployerApplication"
"metadata":
  "name": "log"
"spec":
  "resource": "maven://org.springframework.cloud.stream.app:log-sink-rabbit"
  "resourceMetadata": "maven://org.springframework.cloud.stream.app:log-sink-rabbit:jar:metadata:1.3.1.RELEASE"
  "version": "1.3.1.RELEASE"
  "applicationProperties":
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "log"
    "spring.cloud.stream.metrics.key": "ticktock.log.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.input.group": "ticktock"
    "log.name": "mylogger"
    "spring.cloud.stream.metrics.properties":
"spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*"
    "spring.cloud.dataflow.stream.name": "ticktock"
    "spring.cloud.dataflow.stream.app.type": "sink"
    "foo2": "bar2" (1)
    "spring.cloud.stream.bindings.input.destination": "ticktock.time"
    "foo3": "bar3" (1)
  "deploymentProperties":
    "spring.cloud.deployer.count": "1"
    "spring.cloud.deployer.group": "ticktock"
```

1. The property `foo3=bar3` is added along with the existing `foo2=bar2` for the `log` application.

We can still override the existing properties as follows:

```
dataflow:>stream update --name ticktock --properties app.log.foo3=bar4
Update request has been sent for the stream 'ticktock'

dataflow:>stream manifest ticktock
"apiVersion": "skipper.spring.io/v1"
"kind": "SpringCloudDeployerApplication"
"metadata":
  "name": "time"
"spec":
  "resource": "maven://org.springframework.cloud.stream.app:time-source-rabbit"
  "resourceMetadata": "maven://org.springframework.cloud.stream.app:time-source-rabbit:jar:metadata:1.3.1.RELEASE"
  "version": "1.3.1.RELEASE"
  "applicationProperties":
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "time"
    "spring.cloud.stream.metrics.key": "ticktock.time.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.output.producer.requiredGroups": "ticktock"
    "spring.cloud.stream.metrics.properties":
"spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*"
    "spring.cloud.stream.bindings.output.destination": "ticktock.time"
    "spring.cloud.dataflow.stream.name": "ticktock"
    "spring.cloud.dataflow.stream.app.type": "source"
  "deploymentProperties":
    "spring.cloud.deployer.group": "ticktock"
```

```
---
"apiVersion": "skipper.spring.io/v1"
"kind": "SpringCloudDeployerApplication"
"metadata":
  "name": "log"
"spec":
  "resource": "maven://org.springframework.cloud.stream.app:log-sink-rabbit"
  "resourceMetadata": "maven://org.springframework.cloud.stream.app:log-sink-rabbit:jar:metadata:1.3.1.RELEASE"
  "version": "1.3.1.RELEASE"
  "applicationProperties":
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "log"
    "spring.cloud.stream.metrics.key": "ticktock.log.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.input.group": "ticktock"
    "log.name": "mylogger"
    "spring.cloud.stream.metrics.properties":
"spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*"
    "spring.cloud.dataflow.stream.name": "ticktock"
    "spring.cloud.dataflow.stream.app.type": "sink"
    "foo2": "bar2" (1)
    "spring.cloud.stream.bindings.input.destination": "ticktock.time"
    "foo3": "bar4" (1)
  "deploymentProperties":
    "spring.cloud.deployer.count": "1"
    "spring.cloud.deployer.group": "ticktock"
```

(1) The property `foo3` is replaced with the new value` bar4` and the existing property `foo2=bar2` remains.

Stream History

The history of the stream can be viewed by running the `stream history` command, as shown (with its output), in the following example:

```
dataflow:>stream history --name httptest
╔═══════╤═══════════════════════════╤═══════╤════════════╤═══════════════╤═══════════════╗
║Version│        Last updated       │ Status│Package Name│Package Version│  Description  ║
╠═══════╪═══════════════════════════╪═══════╪════════════╪═══════════════╪═══════════════╣
║2      │Mon Nov 27 22:41:16 EST 2017│DEPLOYED│httptest    │1.0.0          │Upgrade complete║
║1      │Mon Nov 27 22:40:41 EST 2017│DELETED │httptest    │1.0.0          │Delete complete ║
╚═══════╧═══════════════════════════╧═══════╧════════════╧═══════════════╧═══════════════╝
```

Stream Manifest

The manifest is a YAML document that represents the final state of what was deployed to the platform. You can view the manifest for any stream version by using the `stream manifest --name <name-of-stream> --releaseVersion <optional-version>` command. If the `--releaseVersion` is not specified, the manifest for the last version is returned. The following listing shows a typical `stream manifest` command and its output:

```
dataflow:>stream manifest --name httptest

---
# Source: log.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  name: log
spec:
  resource: maven://org.springframework.cloud.stream.app:log-sink-rabbit
  version: 1.2.0.RELEASE
  applicationProperties:
    spring.metrics.export.triggers.application.includes: integration**
    spring.cloud.dataflow.stream.app.label: log
    spring.cloud.stream.metrics.key: httptest.log.${spring.cloud.application.guid}
    spring.cloud.stream.bindings.input.group: httptest
    spring.cloud.stream.metrics.properties:
spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*
    spring.cloud.dataflow.stream.name: httptest
    spring.cloud.dataflow.stream.app.type: sink
    spring.cloud.stream.bindings.input.destination: httptest.http
  deploymentProperties:
    spring.cloud.deployer.indexed: true
    spring.cloud.deployer.group: httptest
    spring.cloud.deployer.count: 1

---
# Source: http.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  name: http
spec:
  resource: maven://org.springframework.cloud.stream.app:http-source-rabbit
  version: 1.2.0.RELEASE
  applicationProperties:
    spring.metrics.export.triggers.application.includes: integration**
    spring.cloud.dataflow.stream.app.label: http
    spring.cloud.stream.metrics.key: httptest.http.${spring.cloud.application.guid}
    spring.cloud.stream.bindings.output.producer.requiredGroups: httptest
    spring.cloud.stream.metrics.properties:
spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*
    server.port: 9000
    spring.cloud.stream.bindings.output.destination: httptest.http
```

```
    spring.cloud.dataflow.stream.name: httptest
    spring.cloud.dataflow.stream.app.type: source
  deploymentProperties:
    spring.cloud.deployer.group: httptest
```

The majority of the deployment and application properties were set by Data Flow in order to enable the applications to talk to each other and send application metrics with identifying labels.

If you compare this YAML document to the one for `--releaseVersion=1` you will see the difference in the log application version.

3.4.2. Rolling back

To go back to the previous version of the stream, use the `stream rollback` command, as shown (with its output) in the following example:

```
dataflow:>stream rollback --name httptest
Rollback request has been sent for the stream 'httptest'
```

By executing the Unix `jps` command, you can see the two java processes running, but now the log application is back to 1.1.0.RELEASE. The http source process remains unchanged. The following listing shows the `jps` command and typical output:

```
$ jps | grep rabbit
22034 http-source-rabbit-1.2.0.RELEASE.jar
23939 log-sink-rabbit-1.1.0.RELEASE.jar
```

Now look in the log file for the skipper server, by using the following command:

`cd` to the directory `/tmp/spring-cloud-dataflow-3784227772192239992/httptest-1511755751505/httptest.log-v3` and `tail -f stdout_0.log`

You should see log entries similar to the following:

```
INFO 21487 --- [  StateUpdate-2] o.s.c.d.spi.local.LocalAppDeployer       : Deploying app with deploymentId httptest.log-v3
   Logs will be in /tmp/spring-cloud-dataflow-3784227772192239992/httptest-1511755751505/httptest.log-v3
INFO 21487 --- [  StateUpdate-2] o.s.c.s.s.d.strategies.HealthCheckStep   : Waiting for apps in release httptest-v3 to be h
INFO 21487 --- [  StateUpdate-2] o.s.c.s.s.d.s.HandleHealthCheckStep      : Release httptest-v3 has been DEPLOYED
INFO 21487 --- [  StateUpdate-2] o.s.c.s.s.d.s.HandleHealthCheckStep      : Apps in release httptest-v3 are healthy.
```

Now post a message to the http source at port `9000`, as follows:

```
dataflow:> http post --target http://localhost:9000 --data "hello world upgraded"
```

The log message in the log sink is now back at the info error level, as shown in the following example:

```
INFO 23939 --- [http.httptest-1] log-sink  : hello world rollback
```

The `history` command now shows that the third version of the stream has been deployed, as shown (with its output) in the following listing:

```
dataflow:>stream history --name httptest
╔═══════╤═══════════════════════════╤════════╤════════════╤═══════════════╤═══════════════╗
║Version│        Last updated       │ Status │Package Name│Package Version│  Description  ║
╠═══════╪═══════════════════════════╪════════╪════════════╪═══════════════╪═══════════════╣
║3      │Mon Nov 27 23:01:13 EST 2017│DEPLOYED│httptest    │1.0.0          │Upgrade complete║
║2      │Mon Nov 27 22:41:16 EST 2017│DELETED │httptest    │1.0.0          │Delete complete ║
║1      │Mon Nov 27 22:40:41 EST 2017│DELETED │httptest    │1.0.0          │Delete complete ║
╚═══════╧═══════════════════════════╧════════╧════════════╧═══════════════╧═══════════════╝
```

If you look at the manifest for version 3, you can see that it shows version 1.1.0.RELEASE for the log sink.

3.5. Deploying Tasks

In this getting started section, we show how to register a task, create a task definition and then launch it. We will then also review information about the task executions.

Launching Spring Cloud Task applications are not delegated to Skipper since they are short lived applications. Tasks are alwasy deployed directly via the Data Flow Server.

1. Register a Task App

   By default, the application registry is empty. As an example, we will register one task application, `timestamp` which simply prints the current time to the log.

```
dataflow:>app register --name timestamp --type task --uri maven://org.springframework.cloud.task.app:timestamp-
task:1.3.0.RELEASE
Successfully registered application 'task:timestamp'
```

Depending on your environment, you may need to configure the Data Flow Server to point to a custom Maven repository location or configure proxy settings. See [configuration-maven] for more information.

2. Create a Task Definition

   Use the `task create` command to create a task definition using the previously registered `timestamp` application. In this example, no additional properties are used to configure the `timestamp` application.

```
dataflow:> task create --name printTimeStamp --definition "timestamp"
```

3. Launch a Task

   The launching of task definitions is done through the shell's `task launch` command.

```
dataflow:> task launch printTimeStamp
```

   Check to see if the a timestamp ended up in log file for the timestamp task. The location of the log file for the task application will be shown in the Data Flow server's log. You should see a log entry similar to

```
TimestampTaskConfiguration$TimestampTask : 2018-02-28 16:42:21.051
```

4. Review task execution

   Information about the task execution can be obtained using the command `task execution list`.

```
dataflow:>task execution list
╔═══════════════╤══╤════════════════════════════╤════════════════════════════╤═════════╗
║   Task Name   │ID│         Start Time         │          End Time          │Exit Code║
╠═══════════════╪══╪════════════════════════════╪════════════════════════════╪═════════╣
║printTimeStamp │1 │Wed Feb 28 16:42:21 EST 2018│Wed Feb 28 16:42:21 EST 2018│0        ║
╚═══════════════╧══╧════════════════════════════╧════════════════════════════╧═════════╝
```

   Additional information can be obtained using the command `task execution status`.

```
dataflow:>task execution status --id 1
╔═══════════════════╤═══════════════════════════════════════════════════════╗
║        Key        │                         Value                         ║
╠═══════════════════╪═══════════════════════════════════════════════════════╣
║Id                 │1                                                      ║
║Name               │printTimeStamp                                         ║
║Arguments          │[--spring.cloud.task.executionid=1]                    ║
║Job Execution Ids  │[]                                                     ║
║Start Time         │Wed Feb 28 16:42:21 EST 2018                           ║
║End Time           │Wed Feb 28 16:42:21 EST 2018                           ║
║Exit Code          │0                                                      ║
║Exit Message       │                                                       ║
║Error Message      │                                                       ║
║External Execution Id │printTimeStamp-ab86b2cc-0508-4c1e-b33d-b3896d17fed7 ║
╚═══════════════════╧═══════════════════════════════════════════════════════╝
```

   The Tasks section has more information on the lifecycle of Tasks and also how to use Composed Tasks which let you create a directed graph where each node of the graph is a task application.

---

4. Getting Started - Cloud Foundry

4.1. System Requirements

The Spring Cloud Data Flow server deploys task tasks (short-lived applications) and via Skipper deploys streams (long-lived applications) to Cloud Foundry. The server is a lightweight Spring Boot application. It can run on Cloud Foundry or your laptop, but it is more common to run the server in Cloud Foundry.

Spring Cloud Data Flow requires a few data services to perform streaming, task/batch processing, and analytics. You have two options when you provision Spring Cloud Data Flow and related services on Cloud Foundry:

- The simplest (and automated) method is to use the Spring Cloud Data Flow for PCF tile. This is an opinionated tile for Pivotal Cloud Foundry. It automatically provisions the server and the required data services, thus simplifying the overall getting-started experience. You can read more about the installation here.
- Alternatively, you can provision all the components manually. The following section goes into the specifics of how to do so.

4.1.1. Provision a Redis Service Instance on Cloud Foundry

A Redis instance is required for analytics apps and is typically bound to such apps when you create an analytics stream by using setting deployment properties set on a per-application basis

You can use `cf marketplace` to discover which plans are available to you, depending on the details of your Cloud Foundry setup. For example, you can use Pivotal Web Services, as the following example shows:

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

4.1.2. Provision a Rabbit Service Instance on Cloud Foundry

RabbitMQ is used as a messaging middleware between streaming apps and is bound to each deployed streaming app. Apache Kafka is the other option. We can use the `SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[pws]_DEPLOYMENT_SERVICES` setting in `Skipper Server` configuration which automatically binds RabbitMQ to the deployed streaming applications.

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

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

4.1.3. Provision a MySQL Service Instance on Cloud Foundry

An RDBMS is used to persist Data Flow state, such as stream and task definitions, deployments, and executions.

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

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

4.2. Cloud Foundry Installation

Starting with 2.0.x, the Data Flow Server requires a `Skipper` server for managing the Streams lifecycle.

1. Download the Data Flow server and shell applications, as the following example shows:

```
wget https://repo.spring.io/snapshot/org/springframework/cloud/spring-cloud-dataflow-server/2.0.0.BUILD-
SNAPSHOT/spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar

wget https://repo.spring.io/snapshot/org/springframework/cloud/spring-cloud-dataflow-shell/2.0.0.BUILD-
SNAPSHOT/spring-cloud-dataflow-shell-2.0.0.BUILD-SNAPSHOT.jar
```

2. Download Skipper to which Data Flow delegate Streams lifecycle operations such as deployment, upgrading and rolling back. The following example shows how to do so:

```
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-skipper-server/2.0.0.M1/spring-cloud-s
```

   Push Skipper to Cloud Foundry. The following example shows a manifest for Skipper.

```
---
applications:
- name: skipper-server
  host: skipper-server
  memory: 1G
  disk_quota: 1G
  instances: 1
  timeout: 180
  buildpack: java_buildpack
  path: <PATH TO THE DOWNLOADED SKIPPER SERVER UBER-JAR>
  env:
    SPRING_APPLICATION_NAME: skipper-server
    SPRING_APPLICATION_JSON: '{ "spring": { "cloud": { "skipper": { "server": { "enableLocalPlatform": "false"} }
    SPRING_CLOUD_SKIPPER_SERVER_STRATEGIES_HEALTHCHECK_TIMEOUTINMILLIS: 300000
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL: https://api.run.pivotal.io
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG: {org}
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE: {space}
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_DOMAIN: cfapps.io
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME: {email}
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD: {password}
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SKIP_SSL_VALIDATION: false
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_DELETE_ROUTES: false
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES: {middlewareServiceNam
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_STREAM_ENABLE_RANDOM_APP_NAME_P
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_MEMORY: 2048m
services:
- {services}
```

You need to fill in `{org}`, `{space}`, `{email}`, `{password}`, `{middlewareServiceName}` (e.g. rabbit or kafka) and {services} (such as mysql) before running these commands. Once you have the desired config values in `manifest.yml`, you can run the `cf push` command to provision the skipper-server.

Only set 'Skip SSL Validation' to `true` if you run on a Cloud Foundry instance by using self-signed certificates (for example, in development). Do not use self-signed certificates for production.

When specifying the `buildpack`, our examples typically specify `java_buildpack` or `java_buildpack_offline`. Use the CF command `cf buildpacks` to get a listing of available relevant buildpacks for your environment.

3. Configure and run the Data Flow Server

   One of the most important configuration details is providing credentials to the Cloud Foundry instance so that the server can itself spawn applications. You can use any Spring Boot-compatible configuration mechanism (passing program arguments, editing configuration files before building the application, using Spring Cloud Config, using environment variables, and others), although some may prove more practicable than others, depending on how you typically deploy applications to Cloud Foundry.

In later sections, we show how to deploy Data Flow by usingenvironment variables or a Cloud Foundry manifest. However, there are some general configuration details you should be aware of in either approach.

4.2.1. General Configuration

You must use a unique name for your app. An application with the same name in the same organization causes your deployment to fail.

The recommended minimum memory setting for the server is 2G. Also, to push apps to PCF and obtain application property metadata, the server downloads applications to a 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, you can increase this 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.

If you are pushing to a space with multiple users (for example, on PWS), the route you chose for your application name may already be taken. You can use the `--random-route` option to avoid this when you push the server application.

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

If you need to configure multiple Maven repositories, a proxy, or authorization for a private repository,

> see Maven Configuration.

4.2.2. Deploying by Using Environment Variables

The following configuration is for Pivotal Web Services. You need to fill in `{org}`, `{space}`, `{email}` and `{password}` before running these commands. Tasks are deployed directly from the Data Flow Server. In the future, you will be able to deploy tasks to multiple platforms, but for 2.0.0.M1 you can deploy only to a single platform and the name must be `default`.

```
cf set-env dataflow-server SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL:
https://api.run.pivotal.io
cf set-env dataflow-server SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG: {org}
cf set-env dataflow-server SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE:
{space}
cf set-env dataflow-server SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_DOMAIN:
cfapps.io
cf set-env dataflow-server SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME:
{email}
cf set-env dataflow-server SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD:
{password}
cf set-env dataflow-server
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SKIP_SSL_VALIDATION: true
cf set-env dataflow-server SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES:
mysql
cf set-env dataflow-server SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_MEMORY:
2048m
```

Deploy Skipper first and then configure the URI location where the Skipper server runs.

The Spring Cloud Data Flow server does not have any default remote maven repository configured. This is intentionally designed to provide the flexibility, so you can override and point to a remote repository of your choice. The out-of-the-box applications that are supported by Spring Cloud Data Flow are available in Spring's repository. If you want to use them, set it as the remote repository, as the following example shows:

```
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

or using the environment variable `MAVEN_REMOTEREPOSITORIES[REPO1]_URL:`.

Only set 'Skip SSL Validation' to true if you run on a Cloud Foundry instance using self-signed certificates (for example, in development). Do not use self-signed certificates for production.

If you are deploying in an environment that requires you to sign on using the Pivotal Single Sign-On Service, see [getting-started-security-cloud-foundry] for information on how to configure the server.

You can now issue a `cf push` command and reference the Data Flow server .jar file, as the following example shows:

```
cf push dataflow-server -b java_buildpack -m 2G -k 2G --no-start -p spring-cloud-dataflow-server-2.0.0.BUILD-
SNAPSHOT.jar
cf bind-service dataflow-server redis
cf bind-service dataflow-server my_mysql
```

4.2.3. Deploying by Using a Manifest

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

The following example template provisions 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
```

```
    MAVEN_REMOTEREPOSITORIES[REPO1]_URL: https://repo.spring.io/libs-snapshot
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL: https://api.sys.huron.cf-
app.com
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG: sabby20
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE: sabby20
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_DOMAIN: apps.huron.cf-app.com
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME: admin
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD: ***
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SKIP_SSL_VALIDATION: true
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES: mysql
    SPRING_CLOUD_SKIPPER_CLIENT_SERVER_URI: https://<skipper-host-name>/api
services:
- mysql
```

Deploy Skipper first and then configure the URI location where the Skipper server runs.

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

4.3. Local Installation

To run the server application locally (on your laptop or desktop) and target your Cloud Foundry installation, configure the Data Flow server by setting the following environment variables.

```
export SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL=https://api.run.pivotal.io
export SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG={org}
export SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE={space}
export SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_DOMAIN=cfapps.io
export SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME={email}
export SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD={password}
export SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SKIP_SSL_VALIDATION=false

# 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_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES=my_mysql
export SKIPPER_CLIENT_HOST https://<skipper-host-name>/api
```

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

Only set 'Skip SSL Validation' to true if you run on a Cloud Foundry instance using self-signed certificates (for example, in development). Do not use self-signed certificates for production.

Deploy Skipper first and then configure the URI location of where the Skipper server is running.

Now we are ready to start the server application, as follows:

```
java -jar spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar
```

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. To use them, substitute `cf set-env` syntax with `export`.

4.4. Data Flow Shell

The following example shows how to start the Data Flow Shell:

```
$ java -jar spring-cloud-dataflow-shell-{dataflow-project-version}.jar
```

4.5. Deploying Streams

By default, the application registry is empty. If you would like to register all out-of-the-box stream applications built with the RabbitMQ binder in bulk, run the following command:

```
dataflow:>app import --uri http://bit.ly/Darwin-SR1-stream-applications-rabbit-maven
```

For more details, review how to register applications.

Data Flow delegates the Streams deployment to Skipper which provide support for features such as Streams update and rollback.

4.5.1. Creating Streams

> Make sure the Skipper server is deployed and have configured the Data Flow server's `SPRING_CLOUD_SKIPPER_CLIENT_SERVER_URI` property to reference the Skipper server.

The following example shows how to create and deploy a stream:

```
dataflow:> stream create --name httptest --definition "http | log"
dataflow:> stream deploy --name httptest --platformName pws
```

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

Now you can post some data. The URL is unique to your deployment. The following example shows how to post data:

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

Now you can see whether `hello world` is in the log files for the `log` application.

> Skipper includes the concept of platforms, so it is important to define the "accounts" based on the project preferences. In the preceding YAML file, the accounts map to `pws` as the platform. You can modify this, and you can have any number of platform definitions. The Spring Cloud Skipper reference guide has more details.

You can read more about the general features of using Skipper to deploy streams in theStream Lifecycle section and how to upgrade a streams in the Updating a Stream section.

4.6. Deploying Streams

This section proceeds with the assumption that Spring Cloud Data Flow, Spring Cloud Skipper, RDBMS, and your desired messaging middleware are all running in PWS. The following listing shows the apps running in a sample org and space:

```
$ cf apps                                                                                    ★
Getting apps in org ORG / space SPACE as email@pivotal.io...
OK

name                          requested state    instances    memory    disk    urls
skipper-server                started            1/1          1G        1G      skipper-server.cfapps.io
dataflow-server               started            1/1          1G        1G      dataflow-server.cfapps.io
```

The following example shows how to start the Data Flow shell for the Data Flow server:

```
$ java -jar spring-cloud-dataflow-shell-{dataflow-project-version}.jar
```

If the Data Flow Server and shell are not running on the same host, you can point the shell to the Data Flow server URL, as follows:

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

Alternatively, you can pass in the `--dataflow.uri` command line option. The shell'sx `--help` command line option shows what options are available.

You can verify the available platforms in Skipper, as follows:

```
dataflow:>stream platform-list
╔═══════╤═══════════╤═══════════════════════════════════════════════════════════════════════════╗
║ Name  │    Type   │                                Description                                ║
╠═══════╪═══════════╪═══════════════════════════════════════════════════════════════════════════╣
║pws    │cloudfoundry│org == [scdf-ci], space == [space-sabby], url == [https://api.run.pivotal.io]  ‖
╚═══════╧═══════════╧═══════════════════════════════════════════════════════════════════════════╝
```

We start by deploying a stream with the `time-source` pointing to `1.2.0.RELEASE` and `log-sink` pointing to `1.1.0.RELEASE`. The goal is to perform a rolling upgrade of the `log-sink` application to `1.2.0.RELEASE`.

```
dataflow:>app register --name time --type source --uri maven://org.springframework.cloud.stream.app:time-source-rabbi
Successfully registered application 'source:time'

dataflow:>app register --name log --type sink --uri maven://org.springframework.cloud.stream.app:log-sink-rabbit:1.1.(
Successfully registered application 'sink:log'

dataflow:>app info source:time
Information about source application 'time':
Resource URI: maven://org.springframework.cloud.stream.app:time-source-rabbit:1.2.0.RELEASE

dataflow:>app info sink:log
Information about sink application 'log':
Resource URI: maven://org.springframework.cloud.stream.app:log-sink-rabbit:1.1.0.RELEASE
```

When you create a stream, use a unique name (one that might not be taken by another application on PCF/PWS).

The following example shows how to create a deploy a stream

```
dataflow:>stream create ticker-314 --definition "time | log"
Created new stream 'ticker-314'
dataflow:>stream deploy ticker-314 --platformName pws
Deployment request has been sent for stream 'ticker-314'
```

While deploying the stream, we supply `--platformName`, which indicates the platform repository (`pws`) to use when deploying the stream applications with Skipper.

Now you can list the running applications again and see your applications in the list, as the following example shows:

```
$ cf apps
Getting apps in org ORG / space SPACE as email@pivotal.io...

name                        requested state   instances   memory   disk   urls
ticker-314-log-v1           started           1/1         1G       1G     ticker-314-log-v1.cfapps.io
ticker-314-time-v1          started           1/1         1G       1G     ticker-314-time-v1.cfapps.io
skipper-server              started           1/1         1G       1G     skipper-server.cfapps.io
dataflow-server             started           1/1         1G       1G     dataflow-server.cfapps.io
```

Now you an verify the logs, as the following example shows:

```
$ cf logs ticker-314-log-v1
...
...
2017-11-20T15:39:43.76-0800 [APP/PROC/WEB/0] OUT 2017-11-20 23:39:43.761  INFO 12 --- [ ticker-314.time.ticker-314-1]
2017-11-20T15:39:44.75-0800 [APP/PROC/WEB/0] OUT 2017-11-20 23:39:44.757  INFO 12 --- [ ticker-314.time.ticker-314-1]
2017-11-20T15:39:45.75-0800 [APP/PROC/WEB/0] OUT 2017-11-20 23:39:45.757  INFO 12 --- [ ticker-314.time.ticker-314-1]
```

Now you can verify the stream history, as the following example shows:

```
dataflow:>stream history --name ticker-314
╔═══════╤═══════════════════════════╤═══════╤════════════╤═══════════════╤═══════════════╗
║Version│        Last updated       │ Status │Package Name│Package Version│  Description  ‖
╠═══════╪═══════════════════════════╪═══════╪════════════╪═══════════════╪═══════════════╣
‖1      │Mon Nov 20 15:34:37 PST 2017│DEPLOYED│ticker-314  │1.0.0          │Install complete‖
╚═══════╧═══════════════════════════╧═══════╧════════════╧═══════════════╧═══════════════╝
```

Now you can verify the package manifest in Skipper. The `log-sink` should be at `1.1.0.RELEASE`. The following

example shows both the command to use and its output:

```
dataflow:>stream manifest --name ticker-314

---
# Source: log.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  name: log
spec:
  resource: maven://org.springframework.cloud.stream.app:log-sink-rabbit
  version: 1.1.0.RELEASE
  applicationProperties:
    spring.cloud.dataflow.stream.app.label: log
    spring.cloud.stream.metrics.properties: spring.application.name,spring.application.index,spring.cloud.application
    spring.cloud.stream.bindings.applicationMetrics.destination: metrics
    spring.cloud.dataflow.stream.name: ticker-314
    spring.metrics.export.triggers.application.includes: integration**
    spring.cloud.stream.metrics.key: ticker-314.log.${spring.cloud.application.guid}
    spring.cloud.stream.bindings.input.group: ticker-314
    spring.cloud.dataflow.stream.app.type: sink
    spring.cloud.stream.bindings.input.destination: ticker-314.time
  deploymentProperties:
    spring.cloud.deployer.indexed: true
    spring.cloud.deployer.group: ticker-314

---
# Source: time.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  name: time
spec:
  resource: maven://org.springframework.cloud.stream.app:time-source-rabbit
  version: 1.2.0.RELEASE
  applicationProperties:
    spring.cloud.dataflow.stream.app.label: time
    spring.cloud.stream.metrics.properties: spring.application.name,spring.application.index,spring.cloud.application
    spring.cloud.stream.bindings.applicationMetrics.destination: metrics
    spring.cloud.dataflow.stream.name: ticker-314
    spring.metrics.export.triggers.application.includes: integration**
    spring.cloud.stream.metrics.key: ticker-314.time.${spring.cloud.application.guid}
    spring.cloud.stream.bindings.output.producer.requiredGroups: ticker-314
    spring.cloud.stream.bindings.output.destination: ticker-314.time
    spring.cloud.dataflow.stream.app.type: source
  deploymentProperties:
    spring.cloud.deployer.group: ticker-314
```

Now you can update `log-sink` from `1.1.0.RELEASE` to `1.2.0.RELEASE`. First we need to register the version 1.2.0.RELEASE. The following example shows how to do so:

```
dataflow:>app register --name log --type sink --uri maven://org.springframework.cloud.stream.app:log-sink-rabbit:1.1.(
Successfully registered application 'sink:log'
```

If you run the `app list` command for the log sink, you can now see that two versions are registered, as the following example shows:

```
dataflow:>app list --id sink:log
╔══════╤═════════╤══════════════════════╤════╗
║source│processor│         sink         │task║
╠══════╪═════════╪══════════════════════╪════╣
║      │         │> log-1.1.0.RELEASE <│    ║
║      │         │log-1.2.0.RELEASE     │    ║
╚══════╧═════════╧══════════════════════╧════╝
```

The greater-than and less-than signs around `> log-1.1.0.RELEASE <` indicate that this is the default version that is used when matching `log` in the DSL for a stream definition. You can change the default version by using the `app default` command.

```
dataflow:>stream update --name ticker-314 --properties version.log=1.2.0.RELEASE
Update request has been sent for stream 'ticker-314'
```

Now you can list the applications again to see the two versions of the `ticker-314-log` application, as the following example shows:

```
± cf apps
Getting apps in org ORG / space SPACE as email@pivotal.io...

Getting apps in org scdf-ci / space space-sabby as sanandan@pivotal.io...
OK

name                        requested state   instances   memory   disk   urls
ticker-314-log-v2           started           1/1         1G       1G     ticker-314-log-v2.cfapps.io
ticker-314-log-v1           stopped           0/1         1G       1G
ticker-314-time-v1          started           1/1         1G       1G     ticker-314-time-v1.cfapps.io
skipper-server              started           1/1         1G       1G     skipper-server.cfapps.io
dataflow-server             started           1/1         1G       1G     dataflow-server.cfapps.io
```

There are two versions of the `log-sink` applications. The `ticker-314-log-v1` application instance is going down (route already removed) and the newly spawned `ticker-314-log-v2` application is bootstrapping. The version number is incremented and the version-number (`v2`) is included in the new application name.

1. Once the new application is up and running, you can verify the logs, as the following example shows:

```
$ cf logs ticker-314-log-v2
...
...
2017-11-20T18:38:35.00-0800 [APP/PROC/WEB/0] OUT 2017-11-21 02:38:35.003  INFO 18 --- [ticker-314.time.ticker-314-1] t
2017-11-20T18:38:36.00-0800 [APP/PROC/WEB/0] OUT 2017-11-21 02:38:36.004  INFO 18 --- [ticker-314.time.ticker-314-1] t
2017-11-20T18:38:37.00-0800 [APP/PROC/WEB/0] OUT 2017-11-21 02:38:37.005  INFO 18 --- [ticker-314.time.ticker-314-1] t
```

Now you can look at the updated package manifest persisted in Skipper. You should now be seeing `log-sink` at 1.2.0.RELEASE. The following example shows the command to use and its output:

```
skipper:>stream manifest --name ticker-314
---
# Source: log.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  name: log
spec:
  resource: maven://org.springframework.cloud.stream.app:log-sink-rabbit
  version: 1.2.0.RELEASE
  applicationProperties:
    spring.cloud.dataflow.stream.app.label: log
    spring.cloud.stream.metrics.properties: spring.application.name,spring.application.index,spring.cloud.application
    spring.cloud.stream.bindings.applicationMetrics.destination: metrics
    spring.cloud.dataflow.stream.name: ticker-314
    spring.metrics.export.triggers.application.includes: integration**
    spring.cloud.stream.metrics.key: ticker-314.log.${spring.cloud.application.guid}
    spring.cloud.stream.bindings.input.group: ticker-314
    spring.cloud.dataflow.stream.app.type: sink
    spring.cloud.stream.bindings.input.destination: ticker-314.time
  deploymentProperties:
    spring.cloud.deployer.indexed: true
    spring.cloud.deployer.group: ticker-314
    spring.cloud.deployer.count: 1

---
# Source: time.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  name: time
spec:
  resource: maven://org.springframework.cloud.stream.app:time-source-rabbit
  version: 1.2.0.RELEASE
  applicationProperties:
    spring.cloud.dataflow.stream.app.label: time
    spring.cloud.stream.metrics.properties: spring.application.name,spring.application.index,spring.cloud.application
    spring.cloud.stream.bindings.applicationMetrics.destination: metrics
    spring.cloud.dataflow.stream.name: ticker-314
    spring.metrics.export.triggers.application.includes: integration**
    spring.cloud.stream.metrics.key: ticker-314.time.${spring.cloud.application.guid}
    spring.cloud.stream.bindings.output.producer.requiredGroups: ticker-314
    spring.cloud.stream.bindings.output.destination: ticker-314.time
    spring.cloud.dataflow.stream.app.type: source
  deploymentProperties:
    spring.cloud.deployer.group: ticker-314
```

Now you can verify stream history for the latest updates.

```
dataflow:>stream history --name ticker-314
╔═══════╤═══════════════════════════╤═══════╤═══════════╤═══════════════╤═══════════════╗
║Version│        Last updated       │ Status│Package Name│Package Version│  Description  ║
╠═══════╪═══════════════════════════╪═══════╪═══════════╪═══════════════╪═══════════════╣
║2      │Mon Nov 20 15:39:37 PST 2017│DEPLOYED│ticker-314 │1.0.0          │Upgrade complete║
║1      │Mon Nov 20 15:34:37 PST 2017│DELETED │ticker-314 │1.0.0          │Delete complete ║
╚═══════╧═══════════════════════════╧═══════╧═══════════╧═══════════════╧═══════════════╝
```

Rolling-back to the previous version is just a command away. The following example shows how to do so and the resulting output:

```
dataflow:>stream rollback --name ticker-314
Rollback request has been sent for the stream 'ticker-314'

...
...

dataflow:>stream history --name ticker-314
╔═══════╤═══════════════════════════╤═══════╤═══════════╤═══════════════╤═══════════════╗
║Version│        Last updated       │ Status│Package Name│Package Version│  Description  ║
╠═══════╪═══════════════════════════╪═══════╪═══════════╪═══════════════╪═══════════════╣
║3      │Mon Nov 20 15:41:37 PST 2017│DEPLOYED│ticker-314 │1.0.0          │Upgrade complete║
║2      │Mon Nov 20 15:39:37 PST 2017│DELETED │ticker-314 │1.0.0          │Delete complete ║
║1      │Mon Nov 20 15:34:37 PST 2017│DELETED │ticker-314 │1.0.0          │Delete complete ║
╚═══════╧═══════════════════════════╧═══════╧═══════════╧═══════════════╧═══════════════╝
```

4.7. Deploying Tasks

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/Dearborn-GA-task-applications-maven
```

Now you can create a simple timestamp task, as the following example shows:

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

Now you can examine the tail of the logs (for example, `cf logs mytask`) and then launch the task in the UI or in the Data Flow Shell, as the following example shows:

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

You will see the year (`2018` at the time of this writing) printed in the logs. The execution status of the task is stored in the database, and you can retrieve information about the task execution by using the `task execution list` and `task execution status --id <ID_OF_TASK>` shell commands or though the Data Flow UI.

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

5. Getting Started - Kubernetes

Spring Cloud Data Flow is a toolkit for building data integration and real-time data-processing pipelines.

Pipelines consist of Spring Boot apps, built with the Spring Cloud Stream or Spring Cloud Task microservice frameworks. This makes Spring Cloud Data Flow suitable for a range of data-processing use cases, from import-export to event-streaming and predictive analytics.

This project provides support for using Spring Cloud Data Flow with Kubernetes as the runtime for these pipelines, with applications packaged as Docker images.

5.1. Installation

This section covers how to install the Spring Cloud Data Flow Server on a Kubernetes cluster. Spring Cloud Data Flow depends on a few services and their availability. For example, we need an RDBMS service for the application registry, stream and task repositories, and task management. For streaming pipelines, we also need Skipper server for lifecycle management and a messaging middleware option, such as Apache Kafka or RabbitMQ. In addition, if the analytics

features are in use, we need a Redis service.

This guide describes setting up an environment for testing Spring Cloud Data Flow on Google Kubernetes Engine and is not meant to be a definitive guide for setting up a production environment. Feel free to adjust the suggestions to fit your test setup. Remember that a production environment requires much more consideration for persistent storage of message queues, high availability, security, and other concerns.

Currently, only applications registered with a `--uri` property pointing to a Docker resource are supported by the Data Flow Server for Kubernetes. However, we do support Maven resources for the `--metadata-uri` property, which is used to list the properties supported by each application. For example, the following application registration is valid:

```
dataflow:>app register --type source --name time --uri docker://springcloudstream/time-source-rabbit:{docker-time-source-rabbit-version} --metadata-uri maven://org.springframework.cloud.stream.app:time-source-rabbit:jar:metadata:{docker-time-source-rabbit-version}
```

However, any application registered with a Maven, HTTP, or File resource for the executable jar (by using a `--uri` property prefixed with `maven://`, `http://` or `file://`) is ***not supported***.

5.1.1. Kubernetes Compatibility

The Spring Cloud Data Flow implementation for Kubernetes uses the Spring Cloud Deployer Kubernetes library for orchestration. Before you begin setting up a Kubernetes cluster, refer to the compatibility matrix to learn more about deployer and server compatibility against Kubernetes release versions.

The following outlines the compatibility between Spring Cloud Data Flow for Kubernetes Server and Kubernetes versions:

Versions: SCDF K8S Server - K8S Deployer \ Kubernetes	1.9.x	1.10.x	1.11.x
Server: 1.4.x - Deployer: 1.3.2	✓	✓	✓
Server: 1.5.x - Deployer: 1.3.6	✓	✓	✓
Server: 1.6.x - Deployer: 1.3.7	✓	✓	✓
Server: 1.7.x - Deployer: 1.3.9	✓	✓	✓

5.1.2. Create a Kubernetes Cluster

The Kubernetes Picking the Right Solution guide lets you choose among many options, so you can pick the one that you are most comfortable using.

All our testing is done against Google Kubernetes Engine as well as Pivotal Container Service. GKE is used as the target platform for this section. We have also successfully deployed with Minikube. We note where you need to adjust for deploying on Minikube.

When starting Minikube, you should allocate some extra resources, since we deploy several services. We start with `minikube start --cpus=4 --memory=4096`. Please note that the allocated memory and cpu for the Minikube VM directly gets assigned to the number of applications deployed in a stream/task. The more you add, the more VM resources is required.

The rest of this getting started guide assumes that you have a working Kubernetes cluster and a `kubectl` command line utility. See the docs for installation instructions: Installing and Setting up kubectl.

5.1.3. Deploying with `kubectl`

To deploy with `kubectl`

1. Get the Kubernetes configuration files.

   You can use the sample deployment and service YAML files in the https://github.com/spring-cloud/spring-cloud-dataflow repository as a starting point. They have the required metadata set for service discovery by the different applications and services deployed. To check out the code, enter the following commands:

   ```
   $ git clone https://github.com/spring-cloud/spring-cloud-dataflow
   $ cd spring-cloud-dataflow
   $ git checkout master
   ```

Choosing a Message Broker

For deployed applications to communicate with each other, you need to select a message broker. The sample deployment and service YAML files provide configurations for RabbitMQ and Kafka. You need to configure only one message broker.

- RabbitMQ

  Run the following command to start the RabbitMQ service:

  ```
  $ kubectl create -f src/kubernetes/rabbitmq/
  ```

  You can use `kubectl get all -l app=rabbitmq` to verify that the deployment, pod, and service resources are running. You can use `kubectl delete all -l app=rabbitmq` to clean up afterwards.

- Kafka

  Run the following command to start the Kafka service:

  ```
  $ kubectl create -f src/kubernetes/kafka/
  ```

  You can use `kubectl get all -l app=kafka` to verify that the deployment, pod, and service resources are running. You can use `kubectl delete all -l app=kafka` to clean up afterwards.

Deploy Services, Skipper and Data Flow

You must deploy a number of services and the Data Flow server. To do so:

1. Deploy MySQL.

   We use MySQL for this guide, but you could use a Postgres or H2 database instead. We include JDBC drivers for all three of these databases. To use a database other than MySQL, you must adjust the database URL and driver class name settings.

   > You can modify the password in the `src/kubernetes/mysql/mysql-deployment.yaml` files if you prefer to be more secure. If you do modify the password, you must also provide it as base64-encoded string in the `src/kubernetes/mysql/mysql-secrets.yaml` file.

   Run the following command to start the MySQL service:

   ```
   kubectl create -f src/kubernetes/mysql/
   ```

   You can use `kubectl get all -l app=mysql` to verify that the deployment, pod, and service resources are running. You can use `kubectl delete all,pvc,secrets -l app=mysql` to clean up afterwards.

2. Deploy Redis.

   The Redis service is used for the analytics functionality. Run the following command to start the Redis service:

   ```
   kubectl create -f src/kubernetes/redis/
   ```

   > If you do not need the analytics functionality, you can turn this feature off by setting `SPRING_CLOUD_DATAFLOW_FEATURES_ANALYTICS_ENABLED` to `false` in the `src/kubernetes/server/server-deployment.yaml` file. If you do not install the Redis service, you should also remove the Redis configuration settings from the following files depending on your message broker:
   >
   > - RabbitMQ: `src/kubernetes/server/server-config-rabbit.yaml`
   > - Kafka: `src/kubernetes/server/server-config-kafka.yaml`

   You can use `kubectl get all -l app=redis` to verify that the deployment, pod, and service resources are running. You can use `kubectl delete all -l app=redis` to clean up afterwards.

3. Deploy the Metrics Collector.

   The Metrics Collector provides message rates for all deployed stream applications. These message rates are visible in the Dashboard UI. Run one of the following commands (depending on your message broker) to start the Metrics Collector:

   - RabbitMQ: `kubectl create -f src/kubernetes/metrics/metrics-deployment-rabbit.yaml`

- Kafka: `kubectl create -f src/kubernetes/metrics/metrics-deployment-kafka.yaml`

  Create the metrics service:

  ```
  $ kubectl create -f src/kubernetes/metrics/metrics-svc.yaml
  ```

  You can use `kubectl get all -l app=metrics` to verify that the deployment, pod, and service resources are running. You can use `kubectl delete all -l app=metrics` to clean up afterwards.

4. Create Role Bindings and Service account

   Since version 1.9, the latest releases of Kubernetes have enabled RBAC on the api-server. If your target platform has RBAC enabled, you must ask a `cluster-admin` to create the `roles` and `role-bindings` for you before deploying the Data Flow server. They associate the Data Flow service account with the roles it needs to be run with.

   To create Role Bindings and Service account:

   ```
   kubectl create -f src/kubernetes/server/server-roles.yaml
   kubectl create -f src/kubernetes/server/server-rolebinding.yaml
   kubectl create -f src/kubernetes/server/service-account.yaml
   ```

   You can use `kubectl get roles` and `kubectl get sa` to list the available roles and service accounts.

   To cleanup roles, bindings and the service account, use the following commands:

   ```
   $ kubectl delete role scdf-role
   $ kubectl delete rolebinding scdf-rb
   $ kubectl delete serviceaccount scdf-sa
   ```

5. Deploy Skipper

   Data Flow delegates to Skipper the streams lifecycle management. Deploy Skipper to enable the stream management features. For more details, see Spring Cloud Skipper Reference Guide for a complete overview.

   Specify the version of Skipper that you want to deploy.

   The deployment is defined in the `src/kubernetes/skipper/skipper-deployment.yaml` file. To control what version of Skipper gets deployed, modify the tag used for the Docker image in the container specification, as the following example shows:

   ```
   spec:
     containers:
     - name: skipper
       image: springcloud/spring-cloud-skipper-server:2.0.0.M1   # <1>
       imagePullPolicy: Always
   ```

   1 You may change the version as you like.

   Skipper includes the concept of platforms, so it is important to define the “accounts” based on the project preferences. In the preceding YAML file, the accounts map to `minikube` as the platform. You can modify this, and you can have any number of platform definitions. More details are in the Spring Cloud Skipper Reference Guide.

   If you want to orchestrate stream processing pipelines with Apache Kafka as the messaging middleware by using Skipper, you must change the `SPRING_APPLICATION_JSON` environment variable value in the `src/kubernetes/skipper/skipper-deployment.yaml` file as follows:

   ```
   "{\"spring.cloud.skipper.server.enableLocalPlatform\" : false,
   \"spring.cloud.skipper.server.platform.kubernetes.accounts.minikube.environmentVariables\":
   \"SPRING_CLOUD_STREAM_KAFKA_BINDER_BROKERS=${KAFKA_SERVICE_HOST}:${KAFKA_SERVICE_PORT},
   SPRING_CLOUD_STREAM_KAFKA_BINDER_ZK_NODES=${KAFKA_ZK_SERVICE_HOST}:${KAFKA_ZK_SERVICE_PORT}\",\"spring.cloud.ski
   pper.server.platform.kubernetes.accounts.minikube.memory\" : \"1024Mi\"}"
   ```

   Additionally, if you want to use the Apache Kafka Streams Binder, configure the `SPRING_APPLICATION_JSON`

environment variable in `src/kubernetes/skipper/skipper-deployment.yaml` as follows:

```
"{\"spring.cloud.skipper.server.enableLocalPlatform\" : false,
\"spring.cloud.skipper.server.platform.kubernetes.accounts.minikube.environmentVariables\":
\"SPRING_CLOUD_STREAM_KAFKA_BINDER_BROKERS=${KAFKA_SERVICE_HOST}:${KAFKA_SERVICE_PORT},
SPRING_CLOUD_STREAM_KAFKA_BINDER_ZK_NODES=${KAFKA_ZK_SERVICE_HOST}:${KAFKA_ZK_SERVICE_PORT},
SPRING_CLOUD_STREAM_KAFKA_STREAMS_BINDER_BROKERS=${KAFKA_SERVICE_HOST}:${KAFKA_SERVICE_PORT},
SPRING_CLOUD_STREAM_KAFKA_STREAMS_BINDER_ZK_NODES=${KAFKA_ZK_SERVICE_HOST}:${KAFKA_ZK_SERVICE_PORT}\",\"spring.c
loud.skipper.server.platform.kubernetes.accounts.minikube.memory\" : \"1024Mi\"}"
```

Run the following commands to start Skipper as the companion server for Spring Cloud Data Flow:

```
kubectl create -f src/kubernetes/skipper/skipper-deployment.yaml
kubectl create -f src/kubernetes/skipper/skipper-svc.yaml
```

You can use the command `kubectl get all -l app=skipper` to verify that the deployment, pod, and service resources are running. You can use `kubectl delete all -l app=skipper` to clean up afterwards.

6. Deploy the Data Flow Server.

Specify the version of Spring Cloud Data Flow server that you want to deploy.

The deployment is defined in the `src/kubernetes/server/server-deployment.yaml` file. To control which version of Spring Cloud Data Flow server gets deployed, modify the tag used for the Docker image in the container specification, as follows:

```
    spec:
      containers:
      - name: scdf-server
        image: springcloud/spring-cloud-dataflow-server:2.0.0.BUILD-SNAPSHOT      # <1>
        imagePullPolicy: Always
```

1 Change the version as you like. This document is based on the `2.0.0.BUILD-SNAPSHOT` release. The docker tag `latest` can be used for `BUILD-SNAPSHOT` releases.

The Skipper service should be running and the `SPRING_CLOUD_SKIPPER_CLIENT_SERVER_URI` property in `src/kubernetes/server/server-deployment.yaml` should point to it.

The Data Flow Server uses the Fabric8 Java client library to connect to the Kubernetes cluster. We use environment variables to set the values needed when deploying the Data Flow server to Kubernetes. We also use the Fabric8 Spring Cloud integration with Kubernetes library to access the Kubernetes ConfigMap and Secrets settings. The ConfigMap settings for RabbitMQ are specified in the `src/kubernetes/server/server-config-rabbit.yaml` file and for Kafka in the `src/kubernetes/server/server-config-kafka.yaml` file. MySQL secrets are located in the `src/kubernetes/mysql/mysql-secrets.yaml` file. If you modified the password for MySQL, you should change it in the `src/kubernetes/mysql/mysql-secrets.yaml` file. Any secrets have to be provided in base64 encoding.

We now configure the Data Flow server with file-based security, and the default user is 'user' with a password of 'password'. You should change these values in `src/kubernetes/server/server-config-rabbit.yaml` for RabbitMQ or `src/kubernetes/server/server-config-kafka.yaml` for Kafka.

The default memory for the pods is 1024Mi. If you expect most of your applications to require more memory, update the value in the `src/kubernetes/server/server-deployment.yaml` file .

- RabbitMQ: `kubectl create -f src/kubernetes/server/server-config-rabbit.yaml`
- Kafka: `kubectl create -f src/kubernetes/server/server-config-kafka.yaml`

To create a server deployment:

```
kubectl create -f src/kubernetes/server/server-svc.yaml
kubectl create -f src/kubernetes/server/server-deployment.yaml
```

You can use `kubectl get all -l app=scdf-server` to verify that the deployment, pod, and service resources are running. You can use `kubectl delete all,cm -l app=scdf-server` to clean up afterwards.

You can use the `kubectl get svc scdf-server` command to locate the `EXTERNAL_IP` address assigned to `scdf-`

`server`. We use that later to connect from the shell. The following example (with output) shows how to do so:

```
$ kubectl get svc scdf-server
NAME          CLUSTER-IP       EXTERNAL-IP       PORT(S)    AGE
scdf-server   10.103.246.82    130.211.203.246   80/TCP     4m
```

The URL you need to use is in this case is `130.211.203.246`.

If you use Minikube, you do not have an external load balancer and the `EXTERNAL_IP` shows as `<pending>`. You need to use the `NodePort` assigned for the `scdf-server` service. You can use the following command to look up the URL to use:

```
$ minikube service --url scdf-server
http://192.168.99.100:31991
```

5.2. Helm Installation

Spring Cloud Data Flow offers a Helm Chart for deploying the Spring Cloud Data Flow server and its required services to a Kubernetes Cluster.

The Helm chart is available since the 1.2 GA release of Spring Cloud Data Flow for Kubernetes.

The following instructions cover how to initialize `Helm` and install Spring Cloud Data Flow on a Kubernetes cluster.

1. Installing Helm

   `Helm` is comprised of two components: the client (Helm) and the server (Tiller). The `Helm` client runs on your local machine and can be installed by following the instructions found here. If Tiller has not been installed on your cluster, run the following `Helm` client command:

   ```
   $ helm init
   ```

   To verify that the `Tiller` pod is running, use the following command: `kubectl get pod --namespace kube-system`. You should see the `Tiller` pod running.

2. Installing the Spring Cloud Data Flow Server and required services.

   Update the `Helm` repository and install the chart:

   ```
   $ helm repo update
   $ helm install --name my-release stable/spring-cloud-data-flow
   ```

As of Spring Cloud Data Flow 1.7.0, the `Helm` chart has been promoted to the `Stable` repository. If you would like to install a previous version, you need access to the incubator repository. To add this repository to our `Helm` set and install the chart, run the following commands:

```
$ helm repo add incubator https://kubernetes-charts-incubator.storage.googleapis.com
$ helm repo update
$ helm install --name my-release incubator/spring-cloud-data-flow
```

If you run on a Kubernetes cluster without a load balancer, such as in Minikube, you should override the service type to use `NodePort`. To do so, add the `--set server.service.type=NodePort` override, as follows:

```
helm install --name my-release --set server.service.type=NodePort \
    stable/spring-cloud-data-flow
```

If you run on a Kubernetes cluster without RBAC, such as in Minikube, you should override `rbac.create` to `false`. By default, it is set to `true` (based on best practices). To do so, add the `--set rbac.create=false` override, as follows:

```
helm install --name my-release --set server.service.type=NodePort \
    --set rbac.create=false \
    stable/spring-cloud-data-flow
```

If you wish to specify a version of Spring Cloud Data Flow other than the current GA release, you can set the `server.version`, as follows (replacing `stable` with `incubator` if needed):

```
helm install --name my-release stable/spring-cloud-data-flow --set server.version=<version-you-want>
```

To see all of the settings that can be configured on the Spring Cloud Data Flow chart, view the README.

The following listing shows Spring Cloud Data Flow's Kubernetes version compatibility with the respective Helm Chart releases:

```
| SCDF-K8S-Server Version \ Chart Version | 0.1.x | 0.2.x | 1.0.x |
|-----------------------------------------|-------|---------------|
|1.2.x                                    |✓      |×      |×      |
|1.3.x                                    |×      |✓      |×      |
|1.4.x                                    |×      |✓      |×      |
|1.5.x                                    |×      |✓      |×      |
|1.6.x                                    |×      |✓      |×      |
|1.7.x                                    |×      |×      |✓      |
|-------------------------------------------------------------------|
```

You should see the following output:

```
NAME:   my-release
LAST DEPLOYED: Sat Mar 10 11:33:29 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME                   TYPE    DATA  AGE
my-release-mysql       Opaque  2     1s
my-release-data-flow   Opaque  2     1s
my-release-redis       Opaque  1     1s
my-release-rabbitmq    Opaque  2     1s

==> v1/ConfigMap
NAME                           DATA  AGE
my-release-data-flow-server    1     1s
my-release-data-flow-skipper   1     1s

==> v1/PersistentVolumeClaim
NAME                  STATUS   VOLUME                                     CAPACITY  ACCESSMODES  STORAGECLASS  AGE
my-release-rabbitmq   Bound    pvc-e9ed7f55-2499-11e8-886f-08002799df04   8Gi       RWO          standard      1s
my-release-mysql      Pending  standard                                   1s
my-release-redis      Pending  standard                                   1s

==> v1/ServiceAccount
NAME                   SECRETS  AGE
my-release-data-flow   1        1s

==> v1/Service
NAME                           CLUSTER-IP      EXTERNAL-IP  PORT(S)                                AGE
my-release-mysql               10.110.98.253   <none>       3306/TCP                               1s
my-release-data-flow-server    10.105.216.155  <pending>    80:32626/TCP                           1s
my-release-redis               10.111.63.33    <none>       6379/TCP                               1s
my-release-data-flow-metrics   10.107.157.1    <none>       80/TCP                                 1s
my-release-rabbitmq            10.106.76.215   <none>       4369/TCP,5672/TCP,25672/TCP,15672/TCP  1s
my-release-data-flow-skipper   10.100.28.64    <none>       80/TCP                                 1s

==> v1beta1/Deployment
NAME                           DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
my-release-mysql               1        1        1           0          1s
my-release-rabbitmq            1        1        1           0          1s
my-release-data-flow-metrics   1        1        1           0          1s
my-release-data-flow-skipper   1        1        1           0          1s
my-release-redis               1        1        1           0          1s
my-release-data-flow-server    1        1        1           0          1s


NOTES:
1. Get the application URL by running these commands:
     NOTE: It may take a few minutes for the LoadBalancer IP to be available.
           You can watch the status of the server by running 'kubectl get svc -w my-release-data-flow-server'
  export SERVICE_IP=$(kubectl get svc --namespace default my-release-data-flow-server -o jsonpath='{.status.loadBalan
  echo http://$SERVICE_IP:80
```

You have just created a new release in the default namespace of your Kubernetes cluster. The `NOTES` section gives instructions for connecting to the newly installed server. It takes a couple of minutes for the application and its required services to start up. You can check on the status by issuing a `kubectl get pod -w` command. Wait for the READY column to show `1/1` for all pods. Once that is done, you can connect to the Data Flow server with the external IP listed by the `kubectl get svc my-release-data-flow-server` command. The default username is `user`, and its password is `password`.

If you run on Minikube, you can use the following command to get the URL for the server:

```
minikube service --url my-release-data-flow-server
```

To see what `Helm` releases you have running, you can use the `helm list` command. When it is time to delete the release, run `helm delete my-release`. This removes any resources created for the release but keeps release information so that you can rollback any changes by using a `helm rollback my-release 1` command. To completely delete the release and purge any release metadata, use `helm delete my-release --purge`.

There is an issue with generated secrets used for the required services getting rotated on chart upgrades. To avoid this issue, set the password for these services when installing the chart. You can use the following command:

```
helm install --name my-release \
    --set rabbitmq.rabbitmqPassword=rabbitpwd \
    --set mysql.mysqlRootPassword=mysqlpwd \
    --set redis.redisPassword=redispwd incubator/spring-cloud-data-flow
```

5.3. Deploying Streams

This section covers how to deploy streams with Spring Cloud Data Flow and Skipper. For more about Skipper, see cloud.spring.io/spring-cloud-skipper.

We assume that Spring Cloud Data Flow, Spring Cloud Skipper, an RDBMS, and your desired messaging middleware is up and running in minikube. We use RabbitMQ as the messaging middleware.

Before you get started, you can see what applications are running. The following example (with output) shows how to do so:

```
$ kubectl get all
NAME                                READY     STATUS    RESTARTS   AGE
po/mysql-777890292-z0dsw            1/1       Running   0          38m
po/rabbitmq-317767540-2qzrr         1/1       Running   0          38m
po/redis-4054078334-37m0l           1/1       Running   0          38m
po/scdf-server-2734071167-bjd3g     1/1       Running   0          48s
po/skipper-2408247821-50z31         1/1       Running   0          3m

...
...
```

5.3.1. Create Streams

This section describes how to create streams (using Skipper). To do so:

1. Download and run the Spring Cloud Data Flow shell.

```
wget http://repo.spring.io/{dataflow-version-type-lowercase}/org/springframework/cloud/spring-cloud-dataflow-
shell/{dataflow-project-version}/spring-cloud-dataflow-shell-{dataflow-project-version}.jar

java -jar spring-cloud-dataflow-shell-{dataflow-project-version}.jar
```

You should see the following startup message from the shell:

```
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/

{dataflow-project-version}

Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help".
server-unknown:>
```

You can connect the Shell to a Data Flow Server running on different host. Use the `kubectl get svc scdf-server` command to retrieve the `EXTERNAL-IP` assigned to `scdf-server` and use that to connect from the shell. The following example shows how to get the external IP address:

```
kubectl get svc scdf-server
NAME         CLUSTER-IP       EXTERNAL-IP       PORT(S)    AGE
scdf-server  10.103.246.82    130.211.203.246   80/TCP     4m
```

In the preceding example, the URL to use is 130.211.203.246

If you use Minikube, you do not have an external load balancer and the EXTERNAL-IP column shows `<pending>`. You need to use the NodePort assigned for the `skipper` service. The following example (with output) shows how to look up the URL to use:

```
$ minikube service --url scdf-server
http://192.168.99.100:31991
```

The following example (with output) shows how to configure the Data Flow server URI (with the default user and password settings):

```
server-unknown:>dataflow config server --username user --password password --uri http://130.211.203.246
Successfully targeted http://130.211.203.246
dataflow:>
```

Alternatively, you can use the `--dataflow.uri` command line option. The shell's `--help` command line option shows what is available.

2. Verify the registered platforms in Skipper, as the following example (with output) shows:

```
dataflow:>stream platform-list
╔═══════╤═══════╤═══════════════════════════════════════════════════════════════════════════════════════════════╗
║ Name  │ Type  │                                          Description                                          ║
╠═══════╪═══════╪═══════════════════════════════════════════════════════════════════════════════════════════════╣
║minikube│kubernetes│master url == [https://kubernetes.default.svc/], namespace == [default], api version == [v1]  ║
╚═══════╧═══════╧═══════════════════════════════════════════════════════════════════════════════════════════════╝
```

3. Register the Docker images of the Rabbit binder based `time` and `log` apps by using the shell.

   We start by deploying a stream with the `time-source` pointing to the 1.3.0.RELEASE and `log-sink` pointing to the 1.2.0.RELEASE. The goal is to perform a rolling upgrade of the `log-sink` application to 1.3.0.RELEASE. The following multi-step example (with output after each command) shows how to do so:

```
dataflow:>app register --type source --name time --uri docker://springcloudstream/time-source-rabbit:1.3.0.RELEASE
Successfully registered application 'source:time'

dataflow:>app register --type sink --name log --uri docker://springcloudstream/log-sink-rabbit:1.2.0.RELEASE --met
Successfully registered application 'sink:log'

dataflow:>app info time --type source
Information about source application 'time':
Version: '1.3.0.RELEASE':
Default application version: 'true':
Resource URI: docker://springcloudstream/time-source-rabbit:1.3.0.RELEASE
╔═══════════════════════╤═══════════════════════╤═══════════════════════╤═══════════════════════╗
║      Option Name      │      Description      │        Default        │          Type         ║
╠═══════════════════════╪═══════════════════════╪═══════════════════════╪═══════════════════════╣
║trigger.time-unit      │The TimeUnit to apply to delay│<none>                 │java.util.concurrent   ║
║                       │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       ║
╚═══════════════════════╧═══════════════════════╧═══════════════════════╧═══════════════════════╝

dataflow:>app info log --type sink
Information about sink application 'log':
Version: '1.2.0.RELEASE':
Default application version: 'true':
Resource URI: docker://springcloudstream/log-sink-rabbit:1.2.0.RELEASE
╔═══════════════════════╤═══════════════════════╤═══════════════════════╤═══════════════════════╗
║      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.   ║
║                       │messages.              │                       │n.handler.LoggingHan   ║
║log.expression         │A SpEL expression (against the│payload                │java.lang.String       ║
║                       │incoming message) to evaluate │                       │                       ║
║                       │as the logged message.        │                       │                       ║
╚═══════════════════════╧═══════════════════════╧═══════════════════════╧═══════════════════════╝
```

For Kafka binder application registration may look like this:

```
dataflow:>app register --type source --name time --uri docker://springcloudstream/time-
source-kafka-10:{docker-time-source-kafka-version} --metadata-uri
maven://org.springframework.cloud.stream.app:time-source-kafka-10:jar:metadata:{docker-
time-source-kafka-version}
dataflow:>app register --type sink --name log --uri docker://springcloudstream/log-sink-
kafka-10:{docker-log-sink-kafka-version} --metadata-uri
maven://org.springframework.cloud.stream.app:log-sink-kafka-10:jar:metadata:{docker-log-
sink-kafka-version}
```

Alternatively, if you want register all out-of-the-box stream applications for a particular binder in bulk, you can use one of the following commands:

- RabbitMQ: `dataflow:>app import --uri bit.ly/Celsius-SR3-stream-applications-rabbit-docker`
- Kafka: `dataflow:>app import --uri bit.ly/Celsius-SR3-stream-applications-kafka-10-docker`

For more details, review how to register applications.

4. Create a simple stream in the shell, by running the following command:

   The following example shows how to create a stream:

```
dataflow:>stream create mystream --definition "time | log"
Created new stream 'mystream'
```

5. Deploy the stream.

   The following example shows how to deploy the stream:

```
dataflow:>stream deploy mystream --platformName minikube
Deployment request has been sent for stream 'mystream'
```

While deploying the stream, we supply `--platformName`, which indicates the platform repository (in this case, `minikube`) to use when deploying the stream applications with Skipper.

6. List the pods.

   The following command (with output) shows how to list the pods. You can run this from the shell by by adding a "!" before the command (which makes a command run as an OS command):

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
mystream-log-v1-0-2k4r8             1/1       Running   0          2m
mystream-time-v1-qhdqq              1/1       Running   0          2m
mysql-777890292-z0dsw          1/1       Running   0          49m
rabbitmq-317767540-2qzrr       1/1       Running   0          49m
redis-4054078334-37m0l         1/1       Running   0          49m
scdf-server-2734071167-bjd3g   1/1       Running   0          12m
skipper-2408247821-50z31       1/1       Running   0          15m

...
...
```

7. Verify the logs.

   The followig example shows how to make sure that the values you expect appear in the logs:

```
$ kubectl logs -f mystream-log-v1-0-2k4r8
...
...
2017-10-30 22:59:04.966  INFO 1 --- [ mystream.time.mystream-1] log-sink                                : 10/30/1
2017-10-30 22:59:05.968  INFO 1 --- [ mystream.time.mystream-1] log-sink                                : 10/30/1
2017-10-30 22:59:07.000  INFO 1 --- [ mystream.time.mystream-1] log-sink                                : 10/30/1
```

8. Verify the stream history.

   The following example (with output) shows how to display the stream history, so that you can verify its content:

```
dataflow:>stream history --name mystream
╔═══════╤═══════════════════════════╤════════╤════════════╤═══════════════╤═══════════════╗
║Version│        Last updated       │ Status │Package Name│Package Version│  Description  ║
╠═══════╪═══════════════════════════╪════════╪════════════╪═══════════════╪═══════════════╣
║1      │Mon Oct 30 16:18:28 PDT 2017│DEPLOYED│mystream    │1.0.0          │Install complete║
╚═══════╧═══════════════════════════╧════════╧════════════╧═══════════════╧═══════════════╝
```

9. Verify the package manifest.

   The `log-sink` should be at 1.2.0.RELEASE. The following example (with output) shows how to display the package manifest so that you can ensure the version of the `log-sink` application:

```
dataflow:>stream manifest --name mystream

---
# Source: log.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  "name": "log"
spec:
  resource: "docker:springcloudstream/log-sink-rabbit"
  resourceMetadata: "docker:springcloudstream/log-sink-rabbit:jar:metadata:1.2.0.RELEASE"
  version: "1.2.0.RELEASE"
  applicationProperties:
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "log"
    "spring.cloud.stream.metrics.key": "mystream.log.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.input.group": "mystream"
    "spring.cloud.stream.metrics.properties": "spring.application.name,spring.application.index,spring.cloud.appli
    "spring.cloud.stream.bindings.applicationMetrics.destination": "metrics"
    "spring.cloud.dataflow.stream.name": "mystream"
    "spring.cloud.dataflow.stream.app.type": "sink"
    "spring.cloud.stream.bindings.input.destination": "mystream.time"
  deploymentProperties:
    "spring.cloud.deployer.group": "mystream"

---
# Source: time.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  "name": "time"
spec:
  resource: "docker:springcloudstream/time-source-rabbit"
  resourceMetadata: "docker:springcloudstream/time-source-rabbit:jar:metadata:1.3.0.RELEASE"
  version: "1.3.0.RELEASE"
  applicationProperties:
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "time"
    "spring.cloud.stream.metrics.key": "mystream.time.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.output.producer.requiredGroups": "mystream"
    "spring.cloud.stream.metrics.properties": "spring.application.name,spring.application.index,spring.cloud.appli
    "spring.cloud.stream.bindings.applicationMetrics.destination": "metrics"
    "spring.cloud.stream.bindings.output.destination": "mystream.time"
    "spring.cloud.dataflow.stream.name": "mystream"
    "spring.cloud.dataflow.stream.app.type": "source"
  deploymentProperties:
    "spring.cloud.deployer.group": "mystream"
```

10. Register the `log-sink` application version 1.3.0.RELEASE and update your stream to use it

    The following example (with output after each command) shows how to register the `log-sink` application and update its version:

```
dataflow:>app register --name log --type sink --uri docker:springcloudstream/log-sink-rabbit:1.3.0.RELEASE --force
Successfully registered application 'sink:log'

dataflow:>stream update --name mystream --properties version.log=1.3.0.RELEASE
Update request has been sent for stream 'mystream'
```

11. List the pods again.

    The following example (with output) shows how to list the pods, so that you can see your application in the list:

```
$ kubectl get pods
NAME                                READY     STATUS        RESTARTS   AGE
mystream-log-v1-0-2k4r8         1/1       Terminating   0          3m
mystream-log-v2-0-fjnlt         0/1       Running       0          9s
mystream-time-v1-qhdqq          1/1       Running       0          3m
mysql-777890292-zOdsw           1/1       Running       0          51m
rabbitmq-317767540-2qzrr        1/1       Running       0          51m
redis-4054078334-37m0l          1/1       Running       0          51m
scdf-server-2734071167-bjd3g    1/1       Running       0          14m
skipper-2408247821-50z31        1/1       Running       0          16m

...
...
```

The list shows two versions of the `log-sink` applications. The `mystream-log-v1-0-2k4r8` pod is going down and the newly spawned `mystream-log-v2-0-fjnlt` pod is bootstrapping. The version number is incremented and the version-number (`v2`) is included in the new application name.

12. Once the new pod is up and running, you can verify the logs.

The following example shows how to display the logs so that you can verify their content:

```
$ kubectl logs -f mystream-log-v2-0-fjnlt
...
...
2017-10-30 23:24:30.016  INFO 1 --- [ mystream.time.mystream-1] log-sink                                 : 10/30/1
2017-10-30 23:24:31.017  INFO 1 --- [ mystream.time.mystream-1] log-sink                                 : 10/30/1
2017-10-30 23:24:32.018  INFO 1 --- [ mystream.time.mystream-1] log-sink                                 : 10/30/1
```

13. View the updated package manifest persisted in Skipper. You should now see `log-sink` at 1.3.0.RELEASE.

    The following example (with output) shows how to view the updated package manifest:

```
dataflow:>stream manifest --name mystream

---
# Source: log.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  "name": "log"
spec:
  resource: "docker:springcloudstream/log-sink-rabbit"
  resourceMetadata: "docker:springcloudstream/log-sink-rabbit:jar:metadata:1.3.0.RELEASE"
  version: "1.3.0.RELEASE"
  applicationProperties:
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "log"
    "spring.cloud.stream.metrics.key": "mystream.log.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.input.group": "mystream"
    "spring.cloud.stream.metrics.properties": "spring.application.name,spring.application.index,spring.cloud.appli
    "spring.cloud.stream.bindings.applicationMetrics.destination": "metrics"
    "spring.cloud.dataflow.stream.name": "mystream"
    "spring.cloud.dataflow.stream.app.type": "sink"
    "spring.cloud.stream.bindings.input.destination": "mystream.time"
  deploymentProperties:
    "spring.cloud.deployer.group": "mystream"
    "spring.cloud.deployer.count": "1"

---
# Source: time.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  "name": "time"
spec:
  resource: "docker:springcloudstream/time-source-rabbit"
  resourceMetadata: "docker:springcloudstream/time-source-rabbit:jar:metadata:1.3.0.RELEASE"
  version: "1.3.0.RELEASE"
  applicationProperties:
    "spring.metrics.export.triggers.application.includes": "integration**"
    "spring.cloud.dataflow.stream.app.label": "time"
    "spring.cloud.stream.metrics.key": "mystream.time.${spring.cloud.application.guid}"
    "spring.cloud.stream.bindings.output.producer.requiredGroups": "mystream"
    "spring.cloud.stream.metrics.properties": "spring.application.name,spring.application.index,spring.cloud.appli
    "spring.cloud.stream.bindings.applicationMetrics.destination": "metrics"
    "spring.cloud.stream.bindings.output.destination": "mystream.time"
    "spring.cloud.dataflow.stream.name": "mystream"
    "spring.cloud.dataflow.stream.app.type": "source"
  deploymentProperties:
    "spring.cloud.deployer.group": "mystream"
```

14. Verify stream history for the latest updates.

    The following example (with output) shows how to display the version history of your stream so that you can verify the version:

```
dataflow:>stream history --name mystream
╔═══════╤════════════════════════════╤═══════╤════════════╤═══════════════╤════════════════╗
║Version│        Last updated        │ Status│Package Name│Package Version│  Description   ║
╠═══════╪════════════════════════════╪═══════╪════════════╪═══════════════╪════════════════╣
║2      │Mon Oct 30 16:21:55 PDT 2017│DEPLOYED│mystream    │1.0.0          │Upgrade complete║
║1      │Mon Oct 30 16:18:28 PDT 2017│DELETED │mystream    │1.0.0          │Delete complete ║
╚═══════╧════════════════════════════╧═══════╧════════════╧═══════════════╧════════════════╝
```

5.3.2. Rolling back to a Previous Version

Skipper includes a `rollback` command so that you can roll back to a previous version. The following example (with output) shows how to use it:

```
dataflow:>stream rollback --name mystream
Rollback request has been sent for the stream 'mystream'

...
...

dataflow:>stream history --name mystream
╔═══════╤═══════════════════════════╤═══════╤════════════╤═══════════════╤════════════════╗
║Version│        Last updated       │ Status│Package Name│Package Version│  Description   ║
╠═══════╪═══════════════════════════╪═══════╪════════════╪═══════════════╪════════════════╣
║3      │Mon Oct 30 16:22:51 PDT 2017│DEPLOYED│mystream    │1.0.0          │Upgrade complete║
║2      │Mon Oct 30 16:21:55 PDT 2017│DELETED │mystream    │1.0.0          │Delete complete ║
║1      │Mon Oct 30 16:18:28 PDT 2017│DELETED │mystream    │1.0.0          │Delete complete ║
╚═══════╧═══════════════════════════╧═══════╧════════════╧═══════════════╧════════════════╝
```

5.3.3. Destroy a Stream

Destroy the stream, by using the following command:

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

5.3.4. Troubleshoot Stream Deployment

To troubleshoot issues such as a container that has a fatal error starting up, add the `--previous` option to view the last terminated container log. You can also get more detailed information about the pods by using the `kubctl describe`, as the following example shows:

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

If you need to specify any of the application-specific configuration properties, you might use the "long form" of them by including the application-specific prefix (for example, `--jdbc.tableName=TEST_DATA`). If you did not register the `--metadata-uri` for the Docker based starter applications, this form is **required**. In this case, you also do not see the configuration properties listed when using the `app info` command or in the Dashboard GUI.

5.3.5. Accessing an Application from outside the Cluster

If you need to be able to connect from outside of the Kubernetes cluster to an application that you deploy (such as the `http-source`), you need to use either an external load balancer for the incoming connections or you need to use a NodePort configuration that exposes a proxy port on each Kubetnetes node. If your cluster does not support external load balancers (such as Minikube), you must use the NodePort approach. You can use deployment properties to configure the access. To specify that you want to have a load balancer with an external IP address created for your application's service, use `deployer.http.kubernetes.createLoadBalancer=true` for the application. For the NodePort configuration, use `deployer.http.kubernetes.createNodePort=<port>`, where `<port>` is a number between 30000 and 32767.

1. Register the `http-source` by using one of the following commands:

   RabbitMQ:

   ```
   dataflow:>app register --type source --name http --uri docker//springcloudstream/http-source-rabbit:{docker-
   http-source-rabbit-version} --metadata-uri maven://org.springframework.cloud.stream.app:http-source-
   rabbit:jar:metadata:{docker-http-source-rabbit-version}
   ```

   Kafka:

   ```
   dataflow:>app register --type source --name http --uri docker//springcloudstream/http-source-kafka:{docker-
   http-source-kafka-version} --metadata-uri maven://org.springframework.cloud.stream.app:http-source-
   kafka:jar:metadata:{docker-http-source-kafka-version}
   ```

2. Create the `http | log` stream without deploying it by using the following command:

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

   If your cluster supports an External LoadBalancer for the `http-source`, you can use the following command to deploy the stream:

```
dataflow:>stream deploy test --properties "deployer.http.kubernetes.createLoadBalancer=true"
```

3. Check whether the pods have started by using the following command:

```
dataflow:>! kubectl get pods -l role=spring-app
command is:kubectl get pods -l role=spring-app
NAME               READY     STATUS    RESTARTS   AGE
test-http-2bqx7    1/1       Running   0          3m
test-log-0-tg1m4   1/1       Running   0          3m
```

Pods that are ready show `1/1` in the `READY` column. Now you can look up the external IP address for the `http` application (it can sometimes take a minute or two for the external IP to get assigned) by using the following command:

```
dataflow:>! kubectl get service test-http
command is:kubectl get service test-http
NAME         CLUSTER-IP       EXTERNAL-IP      PORT(S)    AGE
test-http    10.103.251.157   130.211.200.96   8080/TCP   58s
```

If you use Minikube or any cluster that does not support an external load balancer, you should deploy the stream with a NodePort in the range of 30000-32767. You can use the following command to deploy it:

```
dataflow:>stream deploy test --properties "deployer.http.kubernetes.createNodePort=32123"
```

4. Check whether the pods have started by using the following command:

```
dataflow:>! kubectl get pods -l role=spring-app
command is:kubectl get pods -l role=spring-app
NAME               READY     STATUS    RESTARTS   AGE
test-http-9obkq    1/1       Running   0          3m
test-log-0-ysiz3   1/1       Running   0          3m
```

Pods that are ready show `1/1` in the `READY` column. Now you can look up the URL to use with the following command:

```
dataflow:>! minikube service --url test-http
command is:minikube service --url test-http
http://192.168.99.100:32123
```

5. Post some data to the `test-http` application either by using the `EXTERNAL_IP` address (mentioned earlier) with port 8080 or by using the URL provided by the following Minikube command:

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

6. View the logs for the `test-log` pod, by using the following command:

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

7. Destroy the stream

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

5.4. Deploying Tasks

This section covers how to deploy tasks. To do so:

1. Create a task and launch it. To do so, register the `timestamp` task app and create a simple task definition and launch it, as follows:

```
dataflow:>app register --type task --name timestamp --uri docker:springcloudtask/timestamp-task:{docker-
timestamp-task-version} --metadata-uri maven://org.springframework.cloud.task.app:timestamp-task:jar:metadata:
{docker-timestamp-task-version}
dataflow:>task create task1 --definition "timestamp"
dataflow:>task launch task1
```

   You can now list the tasks and executions byusing the following commands:

```
dataflow:>task list
╔═══════════╤═══════════════╤═══════════╗
║Task Name  │Task Definition│Task Status║
╠═══════════╪═══════════════╪═══════════╣
║task1      │timestamp      │running    ║
╚═══════════╧═══════════════╧═══════════╝

dataflow:>task execution list
╔═══════════╤══╤═══════════════════════════╤═══════════════════════════╤═══════════╗
║Task Name  │ID│         Start Time        │          End Time         │Exit Code  ║
╠═══════════╪══╪═══════════════════════════╪═══════════════════════════╪═══════════╣
║task1      │1 │Fri May 05 18:12:05 EDT 2017│Fri May 05 18:12:05 EDT 2017│0          ║
╚═══════════╧══╧═══════════════════════════╧═══════════════════════════╧═══════════╝
```

2. Destroy the task, by using the following command:

```
dataflow:>task destroy --name task1
```

5.5. Application and Server Properties

This section covers how you can customize the deployment of your applications. You can use a number of properties to influence settings for the applications that are deployed. Properties can be applied on a per-application basis or in the server configuration for all deployed applications.

Properties set on a per-application basis always take precedence over properties set as the server configuration. This arrangement allows for the ability to override global server level properties on a per-application basis.

See `KubernetesDeployerProperties` for more on the supported options.

5.5.1. Using Deployments

The deployer uses `ReplicationController` by default. To use deployments instead, you can set the following option as part of the container `env` section in a deployment YAML file:

```
env:
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_CREATE_DEPLOYMENT
  value: 'true'
```

This is now the preferred setting and will be the default in future releases of the deployer.

5.5.2. Memory and CPU Settings

The apps are deployed by default with the following `Limits` and `Requests` settings:

```
Limits:
  cpu: 500m
  memory: 512Mi
Requests:
  cpu: 500m
  memory: 512Mi
```

You might find that the 512Mi memory limit is too low. To increase it, you can provide a common `spring.cloud.deployer.memory` deployer property, as the following example shows (replace `<app>` with the name of the app for which you want to set the memory):

```
deployer.<app>.memory=640m
```

This property affects both the `Requests` and `Limits` memory value set for the container.

If you want to set the `Requests` and `Limits` values separately, you can use the deployer properties that are specific to the Kubernetes deployer. The following example shows how to set `Limits` to `1000m` for `CPU` and `1024Mi` for memory and `Requests` to `800m` for CPU and `640Mi` for memory:

```
deployer.<app>.kubernetes.limits.cpu=1000m
deployer.<app>.kubernetes.limits.memory=1024Mi
deployer.<app>.kubernetes.requests.cpu=800m
deployer.<app>.kubernetes.requests.memory=640Mi
```

Those values results in the following container settings being used:

```
Limits:
  cpu: 1
  memory: 1Gi
Requests:
  cpu: 800m
  memory: 640Mi
```

When using the common memory property, you should use an `m` suffix for the value. When using the Kubernetes-specific properties, you should use the Kubernetes `Mi` style suffix.

You can also control the default values to which to set the `cpu` and `memory` requirements for the pods that are created as part of application deployments. You can declare the following as part of the container `env` section in a deployment YAML file:

```
env:
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_CPU
  value: 500m
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_MEMORY
  value: 640Mi
```

The settings we have used so far only affect the settings for the container. They do not affect the memory setting for the JVM process in the container. If you would like to set JVM memory settings, you can provide an environment variable to do so. See the next section for details.

5.5.3. Environment Variables

To influence the environment settings for a given application, you can take advantage of the `spring.cloud.deployer.kubernetes.environmentVariables` deployer property. For example, a common requirement in production settings is to influence the JVM memory arguments. You can achieve this by using the `JAVA_TOOL_OPTIONS` environment variable, as the following example shows:

```
deployer.<app>.kubernetes.environmentVariables=JAVA_TOOL_OPTIONS=-Xmx1024m
```

The `environmentVariables` property accepts a comma-delimited string. If an environment variable contains a value which is also a comma-delimited string, it must be enclosed in single quotation marks — for example, `spring.cloud.deployer.kubernetes.environmentVariables=spring.cloud.stream.kafka.binder.brokers='somehost:9092 anotherhost:9093'`

This overrides the JVM memory setting for the desired `<app>` (replace `<app>` with the name of your application).

5.5.4. Liveness and Readiness Probes

The `liveness` and `readiness` probes use paths called `/health` and `/info` respectively. They use a `delay` of `10` for both and a `period` of `60` and `10` respectively. You can change these defaults when you deploy the stream by using deployer properties.

The following example changes the `liveness` probe (replace `<app>` with the name of your application) by setting deployer properties:

```
deployer.<app>.kubernetes.livenessProbePath=/health
deployer.<app>.kubernetes.livenessProbeDelay=120
deployer.<app>.kubernetes.livenessProbePeriod=20
```

The same can be declared as part of the container env section in a deployment YAML file:

```
env:
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_LIVENESS_PROBE_PATH
  value: '/health'
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_LIVENESS_PROBE_DELAY
  value: '120'
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_LIVENESS_PROBE_PERIOD
  value: '20'
```

Similarly, you can swap `liveness` for `readiness` to override the default `readiness` settings.

By default, port 8080 is used as the probe port. You can change the defaults for both `liveness` and `readiness` probe ports by using deployer properties, as the following example shows:

```
deployer.<app>.kubernetes.readinessProbePort=7000
deployer.<app>.kubernetes.livenessProbePort=7000
```

You can also set the port values in the container `env` section of a deployment YAML file:

```
env:
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_READINESS_PROBE_PORT
  value: '7000'
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_LIVENESS_PROBE_PORT
  value: '7000'
```

By default `liveness` and `readiness` probe paths use Spring Boot 2.x+ actuator endpoints. To use Spring Boot 1.x actuator endpoint paths, you must adjust the `liveness` and `readiness` values, for example (replace `<app>` with the name of your application):

```
deployer.<app>.kubernetes.livenessProbePath=/health
deployer.<app>.kubernetes.readinessProbePath=/info
```

To automatically set both `liveness` and `readiness` endpoints on a per-application basis to the default Spring Boot 1.x paths, you can set the following property:

```
deployer.<app>.kubernetes.bootMajorVersion=1
```

You can access secured probe endpoints by using credentials stored in a[Kubernetes secret](). You can use an existing secret, provided the credentials are contained under the `credentials` key name of the secret's `data` block. You can configure probe authentication on a per-application basis. When enabled, it is applied to both the `liveness` and `readiness` probe endpoints by using the same credentials and authentication type. Currently, only `Basic` authentication is supported.

To create a new secret:

1. First generate the base64 string with the credentials used to access the secured probe endpoints.

   Basic authentication encodes a username and password as a base64 string in the format of `username:password`.

   The following example (which includes output and in which you should replace `user` and `pass` with your values) shows how to generate a base64 string:

   ```
   $ echo -n "user:pass" | base64
   dXNlcjpwYXNz
   ```

```
$
```

2. With the encoded credentials, create a file (for example, `myprobesecret.yml`) with the following contents:

```
apiVersion: v1
kind: Secret
metadata:
  name: myprobesecret
type: Opaque
data:
  credentials: GENERATED_BASE64_STRING
```

3. Replace `GENERATED_BASE64_STRING` with the base64-encoded value generated earlier.
4. Create the secret by using `kubectl`, as the following example shows:

```
$ kubectl create -f ./myprobesecret.yml
secret "myprobesecret" created
$
```

5. Set the following deployer properties to use authentication when accessing probe endpoints, as the following example shows:

```
deployer.<app>.kubernetes.probeCredentialsSecret=myprobesecret
```

   Replace `<app>` with the name of the application to which to apply authentication.

5.5.5. Using `SPRING_APPLICATION_JSON`

You can use a `SPRING_APPLICATION_JSON` environment variable to set Data Flow server properties (including the configuration of maven repository settings) that are common across all of the Data Flow server implementations. These settings go at the server level in the container `env` section of a deployment YAML. The following example shows how to do so:

```
env:
- name: SPRING_APPLICATION_JSON
  value: "{ \"maven\": { \"local-repository\": null, \"remote-repositories\": { \"repo1\": { \"url\": \"https://repo.s
```

5.5.6. Private Docker Registry

You can pull Docker images from a private registry on a per-application basis. First, you must create a secret in the cluster. Follow the Pull an Image from a Private Registry guide to create the secret.

Once you have created the secret, use the `imagePullSecret` property to set the secret to use, as the following example shows:

```
deployer.<app>.kubernetes.imagePullSecret=mysecret
```

Replace `<app>` with the name of your application and `mysecret` with the name of the secret you created earlier.

You can also configure the image pull secret at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_IMAGE_PULL_SECRET
  value: mysecret
```

Replace `mysecret` with the name of the secret you created earlier.

5.5.7. Annotations

You can add annotations to Kubernetes objects on a per-application basis. The supported object types are pod `Deployment`, `Service` and `Job`. Annotations are defined in a `key:value` format allowing for multiple annotations separated by a comma. For more information and use cases on annotations see Annotations.

The following example shows how you can configure applications to use annotations:

```
deployer.<app>.kubernetes.podAnnotations=annotationName:annotationValue
deployer.<app>.kubernetes.serviceAnnotations=annotationName:annotationValue,annotationName2:annotationValue2
deployer.<app>.kubernetes.jobAnnotations=annotationName:annotationValue
```

Replace `<app>` with the name of your application and the value of your annotations.

5.5.8. Entry Point Style

An Entry Point Style affects how application properties are passed to the container to be deployed. Currently, three styles are supported:

- `exec`: (default) Passes all application properties and command line arguments in the deployment request as container args. Application properties are transformed into the format of `--key=value`.
- `shell`: Passes all application properties as environment variables. Command line arguments from the deployment request are not converted into environment variables nor set on the container. Application properties are transformed into an uppercase string and `.` characters are replaced with `_`.
- `boot`: Creates an environment variable called `SPRING_APPLICATION_JSON` that contains a JSON representation of all application properties. Command line arguments from the deployment request are set as container args.

In all cases, environment variables defined at the server level configuration and on a per-application basis are set onto the container as-is.

You can configure applications as follows:

```
deployer.<app>.kubernetes.entryPointStyle=<Entry Point Style>
```

Replace `<app>` with the name of your application and `<Entry Point Style>` with your desired Entry Point Style.

You can also configure the Entry Point Style at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_ENTRY_POINT_STYLE
  value: entryPointStyle
```

Replace `entryPointStye` with the desired Entry Point Style.

You should choose an Entry Point Style of either `exec` or `shell`, to correspond to how the `ENTRYPOINT` syntax is defined in the container's `Dockerfile`. For more information and uses cases on `exec` vs `shell`, see the ENTRYPOINT section of the Docker documentation.

Using the `boot` Entry Point Style corresponds to using the `exec` style `ENTRYPOINT`. Command line arguments from the deployment request are passed to the container, with the addition of application properties mapped into the `SPRING_APPLICATION_JSON` environment variable rather than command line arguments.

When you use the `boot` Entry Point Style, the `deployer.<app>.kubernetes.environmentVariables` property must not contain `SPRING_APPLICATION_JSON`.

5.5.9. Deployment Service Account

You can configure a custom service account for application deployments through properties. You can use an existing service account or create a new one. One way to create a service account is by using `kubectl`, as the following example shows:

```
$ kubectl create serviceaccount myserviceaccountname
serviceaccount "myserviceaccountname" created
```

Then you can configure individual applications as follows:

```
deployer.<app>.kubernetes.deploymentServiceAccountName=myserviceaccountname
```

Replace `<app>` with the name of your application and `myserviceaccountname` with your service account name.

You can also configure the service account name at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_DEPLOYMENT_SERVICE_ACCOUNT_NAME
  value: myserviceaccountname
```

Replace `myserviceaccountname` with the service account name to be applied to all deployments.

5.5.10. Image Pull Policy

An image pull policy defines when a Docker image should be pulled to the local registry. Currently, three policies are supported:

- `IfNotPresent` : (default) Do not pull an image if it already exists.
- `Always` : Always pull the image regardless of whether it already exists.
- `Never` : Never pull an image. Use only an image that already exists.

The following example shows how you can individually configure applications:

```
deployer.<app>.kubernetes.imagePullPolicy=Always
```

Replace `<app>` with the name of your application and `Always` with your desired image pull policy.

You can configure an image pull policy at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
        env:
        - name: SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_KUBERNETES_ACCOUNTS[default]_IMAGE_PULL_POLICY
          value: Always
```

Replace `Always` with your desired image pull policy.

5.5.11. Deployment Labels

Custom labels can be set on Deployment related objects. See Labels for more information on labels. Labels are specified in `key:value` format.

The following example shows how you can individually configure applications:

```
deployer.<app>.kubernetes.deploymentLabels=myLabelName:myLabelValue
```

Replace `<app>` with the name of your application, `myLabelName` with your label name and `myLabelValue` with the value of your label.

Additionally, multiple labels can be applied, for example:

```
deployer.<app>.kubernetes.deploymentLabels=myLabelName:myLabelValue,myLabelName2:myLabelValue2
```

Applications

A selection of pre-built stream and task/batch starter apps for various data integration and processing scenarios to facilitate learning and experimentation. The table below includes the pre-built applications at a glance. For more details, review how to register supported applications.

6. Available Applications

Source	Processor	Sink	Task

Source	Processor	Sink	Task
jms	scriptable-transform	log	spark-cluster
ftp	transform	task-launcher-yarn	spark-client
time	header-enricher	throughput	timestamp
load-generator	python-http	task-launcher-local	jdbchdfs-local
syslog	twitter-sentiment	mongodb	composed-task-runner
s3	splitter	hdfs-dataset	timestamp-batch
loggregator	bridge	ftp	
triggertask	pmml	jdbc	
twitterstream	python-jython	aggregate-counter	
mongodb	groovy-transform	cassandra	
gemfire-cq	httpclient	router	
http	filter	redis-pubsub	
rabbit	groovy-filter	file	
tcp	aggregator	websocket	
trigger	tensorflow	s3	
mqtt	tasklaunchrequest-transform	rabbit	
tcp-client		counter	
mail		pgcopy	
jdbc		gpfdist	
gemfire		sftp	
file		field-value-counter	
		hdfs	
		tcp	
		gemfire	
		task-launcher-cloudfoundry	

Architecture

7. 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 through 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, or HTTP(S).
- Docker image.

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 platforms are:

- Cloud Foundry
- Kubernetes
- Local Server

The local server is supported in production for Task deployment as a replacement for the Spring Batch Admin project. The local server is not supported in production for Stream deployments.

There is a deployer Service Provider Interface (SPI) that lets you extend Data Flow to deploy onto other runtimes. There are community implementations

- HashiCorp Nomad
- OpenShift
- Apache Mesos

The Apache YARN implementation has reached end-of-line status. Let us know atGitter if youare interested in forking the project to continue developing and maintaining it.

The Spring Cloud Data Flow Server delegates the deployment and runtime status of stream applications to the Spring Cloud Skipper Server. Later has the capabilities to update and rollback applications in a Stream at runtime as well as the ability to deploy applications to multiple platforms.

For managing task applications you need to select a Spring Cloud Data Flow Server executable jar that targets a single platform.

The Data Flow server is also responsible for:

- Interpreting and executing a stream DSL that describes the logical flow of data through multiple long-lived applications.
- Launching a long-lived task application.
- Interpreting and executing a composed task DSL that describes the logical flow of data through multiple short-lived applications.
- Applying 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.
- Providing the runtime status of deployed applications.

As an example, the stream DSL to describe the flow of data from an HTTP source to an Apache Cassandra sink would be written using a Unix pipes and filter syntax " `http | cassandra` ". Each name in the DSL is mapped to an application that can that Maven or Docker repositories. You can also register an application to an `http` location. Many source, processor, and sink applications for common use cases (such as JDBC, HDFS, HTTP, and router) are provided by the Spring Cloud Data Flow team. The pipe symbol represents the communication between the two applications through 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 for creating 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. Similarly for RabbitMQ, exchanges and queues are created as needed to achieve the desired flow.

The interaction of the main components is shown in the following image:

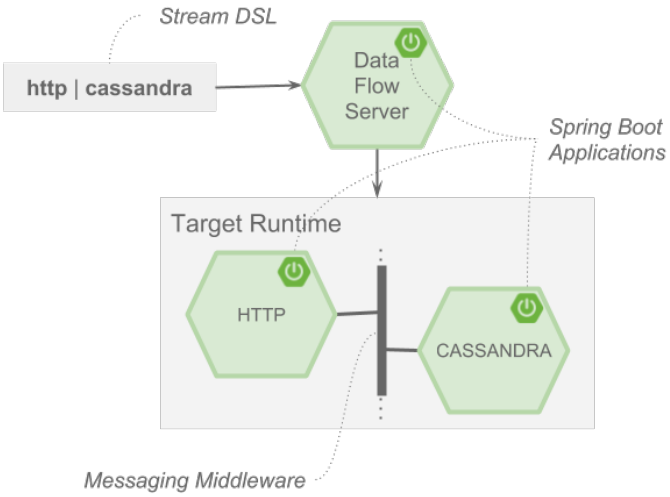


Figure 2. The Spring Cloud Data High Level Architecure

In the preceding 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. Data that is posted to the HTTP application will then be stored in Cassandra. The Samples Repository shows this use case in full detail.

8. 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 independently of the other and each has its own versioning lifecycle. Using Data Flow with Skipper enables you to independently upgrade or rollback each application at runtime.

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 emphasize that these microservice applications are 'just apps' that you can run by yourself by using `java -jar` and passing in appropriate configuration properties. We provide many common microservice applications for common operations so you need not start from scratch when addressing common use cases that build upon the rich ecosystem of Spring Projects, such as Spring Integration, Spring Data, and Spring Batch. Creating your own microservice application is similar to creating other Spring Boot applications. You can start by using the Spring Initializr web site to create the basic scaffolding of either a Stream or Task-based microservice.

In addition to passing the appropriate application properties to each applications, the Data Flow server is responsible for preparing the target platform's infrastructure so that the applications can be deployed. For example, in Cloud Foundry, it would bind specified services to the applications and execute the `cf push` command for each application. For Kubernetes, it would create the replication controller, service, and load balancer.

The Data Flow Server helps simplify the deployment of multiple, relatated, applications onto a target runtime, setting up necessary input and output topics, partitions, and metrics functionality. However, 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 can help you better understand some of the automatic application configuration and platform targeting steps that the Data Flow Server provides.

8.1. 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 the complexity of another execution environment that is often not needed when creating data-centric applications. That does not mean you cannot do real-time data computations when using Spring Cloud Data Flow. Refer to the section Analytics, which describes the integration of Redis to handle common counting-based use cases. Spring Cloud Stream also supports using Reactive APIs such as Project Reactor and RxJava which can be useful for creating functional style applications that contain

time-sliding-window and moving-average functionality. Similarly, Spring Cloud Stream also supports the development of applications in that use the Kafka Streams API.

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 run on the cluster and performs health checks to ensure that long-lived applications are restarted if they fail. Often, framework-specific interfaces are required 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 effort. 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, which 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.

---

9. Data Flow Server

The Data Flow Server provides the following functionality:

- Endpoints
- Security

9.1. 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 by using Spring's MVC framework and the Spring HATEOAS library to create REST representations that follow the HATEOAS principle, as shown in the following image:

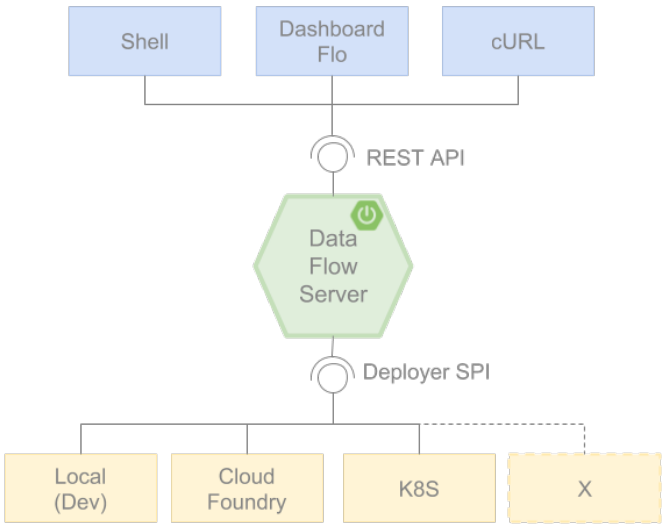


Figure 3. The Spring Cloud Data Flow Server

[NOTE] The Data Flow Server that deploys applications to the local machine is not intended to be used in production for streaming use cases but for the development and testing of stream based applications. The local Data Flow **is** intended to be used in production for batch use cases as a replacement for the Spring Batch Admin project. Both streaming and batch use cases are intended to be used in production when deploying to Cloud Foundry or Kuberenetes.

9.2. 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.

---

10. Streams

10.1. 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 in/fan out data to multiple messaging destinations.

The concept of a tap can be used to 'listen' to the data that is flowing across any of the pipe symbols. "Taps" are just other streams that use an input any one of the "pipes" in a target stream and have an independent life cycle from the target stream.

10.2. Concurrency

For an application that consumes 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.

10.3. 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 (for example, Kafka topics) or not (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.

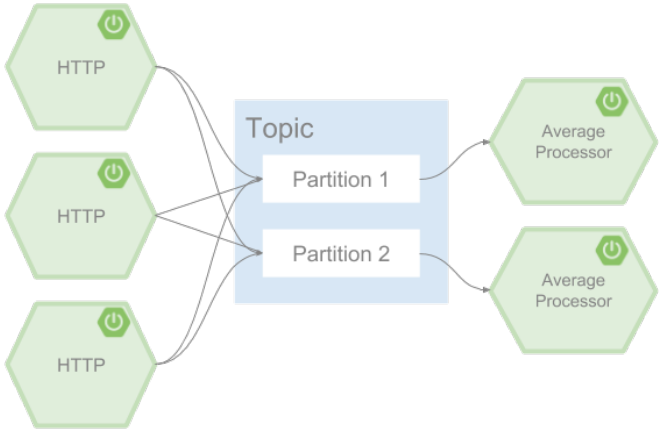


Figure 4. Spring Cloud Stream Partitioning

To use a simple partitioning strategy in Spring Cloud Data Flow, you need only 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 is 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 is not shown in the diagram above.) Suppose the payload being sent to the HTTP source was in JSON format and had a field called `sensorId`. For example, consider the case of deploying the stream with the shell command `stream deploy ingest --propertiesFile ingestStream.properties` where the contents of the `ingestStream.properties` file are as follows:

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

The result is to 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 Passing Stream Partition Properties for additional strategies to partition streams during deployment and how they map onto the underlying Spring Cloud Stream Partitioning properties.

Also note that you cannot currently scale partitioned streams. ReadScaling at Runtime for more information.

10.4. 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 by using the common consumer properties `maxAttempts`, `backOffInitialInterval`, `backOffMaxInterval`, and `backOffMultiplier`. The default values of these properties 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 become the payload of a message and are 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 sends 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 (for example, in the case of Kafka, it is a dedicated topic). To enable this for RabbitMQ set the `republishtoDlq` and `autoBindDlq` consumer properties and the `autoBindDlq` producer property 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 can find extensive declarative support for all the native QOS options.

---

11. Stream Programming Models

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 as well as the KStream/KTable programming model. Common application configuration for a Source that generates data, a Processor that consumes and produces data, and a Sink that consumes data is provided as part of the library.

11.1. Imperative Programming Model

Spring Cloud Stream is most closely integrated with Spring Integration's imperative "one event at a time" programming model. This means you write code that handles a single event callback, as shown in the following 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 used to tie the input channel to the external middleware.

11.2. Functional Programming Model

However, Spring Cloud Stream can support other programming styles, such as reactive APIs, where incoming and outgoing data is handled as continuous data flows and how each individual message should be handled is defined. With many reactive AOIs, you can also use operators that describe functional transformations from inbound to outbound data flows. Here is an example:

```
@EnableBinding(Processor.class)
public static class UppercaseTransformer {

  @StreamListener
  @Output(Processor.OUTPUT)
  public Flux<String> receive(@Input(Processor.INPUT) Flux<String> input) {
    return input.map(s -> s.toUpperCase());
  }
}
```

12. Application Versioning

Application versioning within a Stream is now supported when using Data Flow together with Skipper. You can update application and deployment properties as well as the version of the application. Rolling back to a previous application version is also supported.

13. Task Programming Model

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.
- The ability to emit task events to a stream (as a source) during the task lifecycle.
- Integration with Spring Batch Jobs.

See the Tasks section for more information.

14. Analytics

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

- 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.

Note that the timestamp used in the aggregate counter can come from a field in the message itself so that out-of-order messages are properly accounted.

15. Runtime

The Data Flow Server relies on the target platform for the following runtime functionality:

- Fault Tolerance
- Resource Management

15.1. Fault Tolerance

The target runtimes supported by Data Flow all have the ability to restart a long-lived application. Spring Cloud Data Flow sets up health probes are required by the runtime environment when deploying the application. You also have the ability to customize the health probes.

The collective state of all applications that make up the stream is used to determine the state of the stream. If an application fails, the state of the stream changes from 'deployed' to 'partial'.

15.2. Resource Management

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

15.3. Scaling at Runtime

When deploying a stream, you can set the instance count for each individual application that makes up 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.

Currently, scaling at runtime is not supported with the Kafka binder, as well as with partitioned streams, for which the suggested workaround is redeploying the stream with an updated number of instances. Both cases require a static consumer to be set up, based on information about the total instance count and current instance index.

Configuration

16. Configuration - Local

16.1. Feature Toggles

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

- Streams (requires Skipper)
- Tasks
- Analytics
- Task Scheduler

One can enable and 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`
- `spring.cloud.dataflow.features.schedules-enabled`

By default, stream (requires Skipper), tasks, and analytics are enabled and Task Scheduler is disabled by default.

Since the analytics feature is enabled by default, the Data Flow Server expects to have a valid Redis store available to server as the analytic repository, because Spring Cloud Data Flow provides 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 Data Flow's endpoints to read analytics data written to Redis, then disable the analytics feature by setting the property mentioned above.

The REST `/about` endpoint provides information on the features that have been enabled and disabled.

16.2. Database

A relational database is used to store stream and task definitions as well as the state of executed tasks. Spring Cloud Data Flow provides schemas for H2, HSQLDB, MySQL, Oracle, Postgresql, DB2, and SqlServer. The schema is automatically created when the server starts.

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

The JDBC drivers for **MySQL** (through the MariaDB driver), **HSQLDB**, **PostgreSQL**, and embedded **H2** are available without additional configuration. If you are using any other database, then you need to put the corresponding JDBC driver jar on the classpath of the server.

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

The following example shows how to define a database connection with 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"
```

The following example shows how to define a MySQL database connection with command Line arguments

```
java -jar spring-cloud-dataflow-server/target/spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar \
    --spring.datasource.url=jdbc:mysql://localhost:3306/mydb \
    --spring.datasource.username= \
    --spring.datasource.password= \
    --spring.datasource.driver-class-name=org.mariadb.jdbc.Driver &
```

The following example shows how to define a PostgreSQL database connection with command line arguments:

```
java -jar spring-cloud-dataflow-server/target/spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar \
    --spring.datasource.url=jdbc:postgresql://localhost:5432/mydb \
    --spring.datasource.username= \
    --spring.datasource.password= \
```

```
    --spring.datasource.driver-class-name=org.postgresql.Driver &
```

The following example shows how to define a HSQLDB database connection with command line arguments:

```
java -jar spring-cloud-dataflow-server/target/spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar \
    --spring.datasource.url=jdbc:hsqldb://localhost:9001/mydb \
    --spring.datasource.username=SA \
    --spring.datasource.driver-class-name=org.hsqldb.jdbc.JDBCDriver &
```

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 does not start.

16.2.1. Disabling database schema creation and migration strategies

You can control whether or not Data Flow bootstraps your database on startup. On most production environments chances are you will not have enough privileges to do so. If that's the case you can disable it by setting the property `spring.cloud.dataflow.rdbms.initialize.enable` to `false`. The scripts that the server uses to bootstrap the database can be found under `spring-cloud-dataflow-core/src/main/resources/` folder.

For new installations run the corresponding database script located under `/schemas` and `/migrations.1.x.x`, for upgrades from version `1.2.0` you only need to run the `/migrations.1.x.x` scripts.

16.2.2. Adding a Custom JDBC Driver

To add a custom driver for the database (for example, Oracle), you should rebuild the Data Flow Server and add the dependency to the Maven `pom.xml` file. Since there is a Spring Cloud Data Flow Server for each target platform, you need to modify the appropriate maven `pom.xml` for each platform. There are tags in each GitHub repository for each server version.

To add a custom JDBC driver dependency for the local server implementation:

1. Select the tag that corresponds to the version of the server you want to rebuild and clone the github repository.
2. Edit the spring-cloud-dataflow-server/pom.xml and, in the `dependencies` section, add the dependency for the database driver required. In the following example , an Oracle driver has been chosen:

```
<dependencies>
...
  <dependency>
    <groupId>com.oracle.jdbc</groupId>
    <artifactId>ojdbc8</artifactId>
    <version>12.2.0.1</version>
  </dependency>
...
</dependencies>
```

3. Build the application as described in Building Spring Cloud Data Flow

You can also provide default values when rebuilding the server by adding the necessary properties to the dataflow-server.yml file, as shown in the following example for PostgreSQL:

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/mydb
    username: myuser
    password: mypass
    driver-class-name:org.postgresql.Driver
```

16.3. Local Deployer

You can use the following configuration properties of the Data Flow Local server's deployer to customize how applications are deployed:

```
spring.cloud.deployer.local.workingDirectoriesRoot=java.io.tmpdir # Directory in which all created processes will run and
create log files.

spring.cloud.deployer.local.deleteFilesOnExit=true # Whether to delete created files and directories on JVM exit.

spring.cloud.deployer.local.envVarsToInherit=TMP,LANG,LANGUAGE,"LC_.*. # Array of regular expression patterns for
environment variables that are passed to launched applications.

spring.cloud.deployer.local.javaCmd=java # Command to run java.

spring.cloud.deployer.local.shutdownTimeout=30 # Max number of seconds to wait for app shutdown.

spring.cloud.deployer.local.javaOpts== # The Java options to pass to the JVM

spring.cloud.deployer.local.freeDiskSpacePercentage=5 # The target percentage of free disk space to always aim for when
```

```
cleaning downloaded resources (typically via the local maven repository). Specify as an integer greater than zero and
less than 100. Default is 5.
```

Data Flow Local server itself overrides `spring.cloud.deployer.local.freeDiskSpacePercentage` to `0` from its default value.

When deploying the application, you can also set deployer properties prefixed with `deployer.<name of application>`. For example, to set Java options for the time application in the `ticktock` stream, use the following stream deployment properties.

```
dataflow:> stream create --name ticktock --definition "time --server.port=9000 | log"
dataflow:> stream deploy --name ticktock --properties "deployer.time.local.javaOpts=-Xmx2048m -Dtest=foo"
```

As a convenience, you can set the `deployer.memory` property to set the Java option `-Xmx`, as shown in the following example:

```
dataflow:> stream deploy --name ticktock --properties "deployer.time.memory=2048m"
```

At deployment time, if you specify an `-Xmx` option in the `deployer.<app>.local.javaOpts` property in addition to a value of the `deployer.<app>.local.memory` option, the value in the `javaOpts` property has precedence. Also, the `javaOpts` property set when deploying the application has precedence over the Data Flow Server's `spring.cloud.deployer.local.javaOpts` property.

16.4. Maven

If you want to override specific maven configuration properties (remote repositories, proxies, and others) or run the Data Flow Server behind a proxy, you need to specify those properties as command line arguments when starting the Data Flow Server, as shown in the following example:

```
$ java -jar spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar --spring.config.additional-
location=/home/joe/maven.yml
```

where `maven.yaml` is

```
maven
  localRepository: mylocal
  remote-repositories:
    repo1:
      url: https://repo1
      auth:
        username: user1
        password: pass1
      snapshot-policy:
        update-policy: daily
        checksum-policy: warn
      release-policy:
        update-policy: never
        checksum-policy: fail
    repo2:
      url: https://repo2
      policy:
        update-policy: always
        checksum-policy: fail
  proxy:
    host: proxy1
    port: "9010"
    auth:
      username: proxyuser1
      password: proxypass1
```

By default, the protocol is set to `http`. You can omit the auth properties if the proxy does not need a username and password. Also, the maven `localRepository` is set to `${user.home}/.m2/repository/` by default. As shown in the preceding example, the remote repositories can be specified along with their authentication (if needed). If the remote repositories are behind a proxy, then the proxy properties can be specified as shown in the preceding example.

The repository policies can be specified for each remote repository configuration as shown in the preceding example. The key `policy` is applicable to both `snapshot` and the `release` repository policies.

You can refer to Repository Policies for the list of supported repository policies.

As these are Spring Boot `@ConfigurationProperties`, that you need to specify by adding them to the `SPRING_APPLICATION_JSON` environment variable. The following example shows how the JSON is structured:

```
$ SPRING_APPLICATION_JSON='{ "maven": { "local-repository": null,
"remote-repositories": { "repo1": { "url": "https://repo1", "auth": { "username": "repo1user", "password": "repo1pass" }
}, "repo2": { "url": "https://repo2" } },
"proxy": { "host": "proxyhost", "port": 9018, "auth": { "username": "proxyuser", "password": "proxypass" } } } }' java -
jar spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar
```

16.5. Local Data Flow Server Logging configuration

Spring Cloud Data Flow `local` server is configured to use `RollingFileAppender` for logging and you should enable the spring profile named 'local'. You need to pass the property `--spring.profiles.active=local` when starting the uber jar to trigger the configuration of logging in local mode. The profile is needed since writing to a log file is not the recommended way to gather logs when running on Cloud Platforms such as Cloud Foundry and Kubernetes.

```
logging:
  config: classpath:logback-scdf-local.xml
```

By default, the log file is configured to use:

```
<property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdir:-/tmp}}}/spring-cloud-dataflow-
server}"/>
```

with the logback configuration for the `RollingPolicy`:

```
<appender name="FILE"
     class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>${LOG_FILE}.log</file>
  <rollingPolicy
    class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
   <!-- daily rolling -->
   <fileNamePattern>${LOG_FILE}.${LOG_FILE_ROLLING_FILE_NAME_PATTERN:-%d{yyyy-MM-dd}}.%i.gz</fileNamePattern>
   <maxFileSize>${LOG_FILE_MAX_SIZE:-100MB}</maxFileSize>
   <maxHistory>${LOG_FILE_MAX_HISTORY:-30}</maxHistory>
   <totalSizeCap>${LOG_FILE_TOTAL_SIZE_CAP:-500MB}</totalSizeCap>
  </rollingPolicy>
  <encoder>
   <pattern>${FILE_LOG_PATTERN}</pattern>
  </encoder>
 </appender>
```

To check the `java.io.tmpdir` for the current Spring Cloud Data Flow Server `local` server,

```
jinfo <pid> | grep "java.io.tmpdir"
```

If you want to change or override any of the properties `LOG_FILE`, `LOG_PATH`, `LOG_TEMP`, `LOG_FILE_MAX_SIZE`, `LOG_FILE_MAX_HISTORY` and `LOG_FILE_TOTAL_SIZE_CAP`, please set them as system properties.

16.6. Skipper

Data Flow Server delegates to the Skipper server the management of the Stream's lifecycle. Set the configuration property `spring.cloud.skipper.client.serverUri` to the location of Skipper, e.g.

```
$ java -jar spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar --
spring.cloud.skipper.client.serverUri=http://192.51.100.1:7577/api
```

16.7. 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 OAuth 2.0.

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

While you can theoretically choose any OAuth provider in conjunction with Spring Cloud Data Flow, we recommend using the CloudFoundry User Account and Authentication (UAA) Server.

Not only is the UAA OpenID certified and is used by Cloud Foundry but it can also be used in local stand-alone deployment scenarios.

Furthermore, the UAA not only provides its own user store, but also provides comprehensive LDAP integration.

16.7.1. Enabling HTTPS

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

```
server:
  port: 8443                                    (1)
  ssl:
    key-alias: yourKeyAlias                     (2)
    key-store: path/to/keystore                 (3)
```

```
    key-store-password: yourKeyStorePassword          4
    key-password: yourKeyPassword                     5
    trust-store: path/to/trust-store                  6
    trust-store-password: yourTrustStorePassword      7
```

1. As the default port is `9393`, you may choose to change the port to a more common HTTPs-typical port.
2. The alias (or name) under which the key is stored in the keystore.
3. The path to the keystore file. Classpath resources may also be specified, by using the classpath prefix - for example: `classpath:path/to/keystore`.
4. The password of the keystore.
5. The password of the key.
6. The path to the truststore file. Classpath resources may also be specified, by using the classpath prefix - for example: `classpath:path/to/trust-store`
7. The password of the trust store.

If HTTPS is enabled, it completely replaces 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"  1
         -keypass dataflow -storepass dataflow
```

1. `CN` is the important parameter here. It should match the domain you are trying to access - for example, `localhost`.

Then add the following lines 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 is needed for the Data Flow Server. Once you start the server, you should be able to access it at `localhost:8443/`. As this is a self-signed certificate, you should hit a warning in your browser, which 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:

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

Adding 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, as follows:

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

Next, we need to create a truststore which the shell can use, as follows:

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

Now, you are ready to launch the Data Flow Shell by 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-2.0.0.BUILD-SNAPSHOT.jar
```

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

Do not forget to target the Data Flow Server with the following:

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

Skipping Certificate Validation

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

If you set this command-line parameter, the shell accepts any (self-signed) SSL certificate.

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.

16.7.2. Authentication using OAuth 2.0

In order to support authentication and authorization, Spring Cloud Data Flow is usingOAuth 2.0 and OpenID Connect. It lets you integrate Spring Cloud Data Flow into Single Sign On (SSO) environments.

As of Spring Cloud Data Flow 2.0, OAuth2 is the only mechanism for providing authentication and authorization.

The following OAuth2 Grant Types are used:

- **Authorization Code**: Used for the GUI (browser) integration. Visitors are redirected to your OAuth Service for authentication
- **Password**: Used by the shell (and the REST integration), so visitors can log in with username and password
- **Client Credentials**: Retrieve an access token directly from your OAuth provider and pass it to the Data Flow server by using the Authorization HTTP header

Currently, Spring Cloud Data Flow uses opaque tokens and not transparent tokens (JWT).

The REST endpoints can be accessed in two ways:

- **Basic authentication**, which uses the Password Grant Type under the covers to authenticate with your OAuth2 service
- **Access token**, which uses the Client Credentials Grant Type under the covers

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 by setting environment variables:

```
security:
  oauth2:
    client:
      client-id: myclient                                          (1)
      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
```

(1) Providing the Client ID in the OAuth Configuration Section activates OAuth2 security

You can verify that basic authentication is working properly by using curl, as follows:

```
$ curl -u myusername:mypassword http://localhost:9393/ -H 'Accept: application/json'
```

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

Please be aware that when accessing the Root URL with a web browser and enabled security, you are redirected to the Dashboard UI. In order to see the list of REST endpoints, specify the `application/json`. Also be sure to add the Accept header using tools such as Postman (Chrome) or

> RESTClient (Firefox).

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/ -H 'Accept: application/json'
```

OAuth REST Endpoint Authorization

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

> 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.

Because the determination of security roles is environment-specific, Spring Cloud Data Flow, by default, assigns all roles to authenticated OAuth2 users by 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 takes precedence over the default one provided by Spring Cloud Data Flow.

OAuth Authentication using the Spring Cloud Data Flow Shell

When using the Shell, the credentials can either be provided via username and password or by specifying a *credentials-provider* command.

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

```
$ java -jar spring-cloud-dataflow-shell-2.0.0.BUILD-SNAPSHOT.jar \
  --dataflow.uri=http://localhost:9393 \
  --dataflow.username=my_username --dataflow.password=my_password
```

> 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 via username/password authentication.

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

```
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                      ║
╚═══════════╧═══════════════════════════════════════════╝
```

Alternatively, you can specify the *credentials-provider* command in order to pass-in a bearer token directly, instead of providing a username and password. This works from within the shell or by providing the `--dataflow.credentials-provider-command` command-line argument when starting the Shell.

> When using the *credentials-provider* command, please be aware that your specified command**must** return a *Bearer token* (Access Token prefixed with *Bearer*). For instance, in Unix environments the following simplistic command can be used:
>
> ```
> $ java -jar spring-cloud-dataflow-shell-2.0.0.BUILD-SNAPSHOT.jar \
>   --dataflow.uri=http://localhost:9393 \
>   --dataflow.credentials-provider-command="echo Bearer 123456789"
> ```

OAuth2 Authentication Examples

This section documents following authentication scenarios:

- Local OAuth2 Server
- Authentication with GitHub

Local OAuth2 Server

For local deployment scenarios, we recommend using the CloudFoundry User Account and Authentication (UAA) Server, which is OpenID certified.

While the UAA is also used by Cloud Foundry, it is also a fully featured stand alone OAuth2 server with enterprise features such as LDAP integration.

For local testing and development, you can also use the support provided by Spring Security OAuth. It allows you to easily create your own OAuth2 Server with the following simple annotations:

- `@EnableResourceServer`
- `@EnableAuthorizationServer`

A working example application can be found at:

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

Clone the project and configure Spring Cloud Data Flow with the respective Client ID and Client Secret. Then build and start the project.

Authentication with GitHub

If you 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 image:

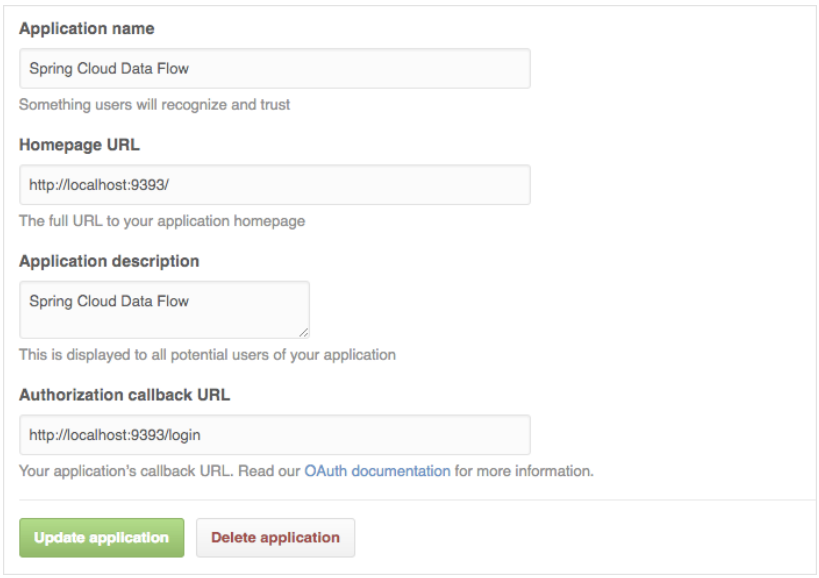


Figure 5. Register an OAuth Application for GitHub

For the authorization callback URL, enter Spring Cloud Data Flow's Login URL - for example, `localhost:9393/login`.

Configure Spring Cloud Data Flow with the GitHub relevant Client ID and Secret, as follows:

```
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
```

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

16.7.3. 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
```

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

Setting up a CloudFoundry User Account and Authentication (UAA) Server

Checkout, Build and Run UAA:

```
$ git clone https://github.com/cloudfoundry/uaa.git
$ cd uaa/
$ ./gradlew  run
```

In a separate window, check if the UAA server is running:

The configuration of the UAA is driven by a[uaa.yml](uaa.yml) file. You can provide custom configuration but in this case we will use the embedded default configuration that provides also a default user:

```
curl -v -d"username=marissa&password=koala&client_id=app&grant_type=password" -u "app:appclientsecret"
http://localhost:8080/uaa/oauth/token -d 'token_format=opaque'
```

This should produce output similar to the following:

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
* Server auth using Basic with user 'app'
> POST /uaa/oauth/token HTTP/1.1
> Host: localhost:8080
> Authorization: Basic YXBwOmFwcGNsaWVudHNlY3JldA==
> User-Agent: curl/7.54.0
> Accept: */*
> Content-Length: 85
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 85 out of 85 bytes
< HTTP/1.1 200
< Cache-Control: no-store
< Pragma: no-cache
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: DENY
< X-Content-Type-Options: nosniff
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Mon, 03 Dec 2018 23:58:41 GMT
<
* Connection #0 to host localhost left intact
{"access_token":"0f935cea42fd4516bb36e4088f6d7c44","token_type":"bearer","id_token":"eyJhbGciOiJIUzI1NiIsImprdSI6Imh0dHBz
Oi8vbG9jYWxob3NOOjgwODAvdWFhL3Rva2VuX2tleXMiLCJraWQiOiJsZWdhY3ktdG9rZW4ta2V5IiwidHlwIjoiSldUIn0.eyJzdWIiOiI1ODExMWNlNC02N
GRlLTQ3ZDYtYjZiZi0wZDRhNDFhNGFlMDAiLCJhdWQiOlsiYXBwIl0sImlzcyI6Imh0dHA6Ly9sb2NhbGhvc3Q6ODA4MC91YWEvb2F1dGgvdG9rZW4iLCJleH
AiOjE1NDM5MjQ3MjEsImlhdCI6MTUOMzg4MTUyMSwiYW1yIjpbInB3ZCJdLCJhenAiOiJhcHAiLCJzY29wZSI6WyJvcGVuaWQiXSwiZW1haWwiOiJtYXJpc3N
hQHRlc3Qub3JnIiwiemlkIjoidWFhIiwib3JpZ2luIjoidWFhIiwianRpIjoiMGY5MzVjZWE0MmZkNDUxNmJiMzZlNDA4OGY2ZDdjNDQiLCJwcmV2aW91c19s
b2dvbl90aW1lIjoxNTQzODgxMTk1NzU2LCJlbWFpbF92ZXJpZmllZCI6dHJ1ZSwiY2xpZW50X2lkIjoiYXBwIiwiY2lkIjoiYXBwIiwiZ3JhbnRfdHlwZSI6I
nBhc3N3b3JkIiwidXNlcl9uYW1lIjoibWFyaXNzYSIsInJldl9zaWciOiI1ZmI4N2M1NiIsInVzZXJfaWQiOiI1ODExMWNlNC02NGRlLTQ3ZDYtYjZiZiOwZD
RhNDFhNGFlMDAiLCJhdXRoX3RpbWUiOjE1NDM4ODE1MjF9.qSxoygd7LUUJSENOwGI3-
UOx2tYZzHnTDuzk6AJUfBk","refresh_token":"d106513b98804a5O8c35d3f7b656a7fe-r","expires_in":43199,"scope":"scim.userids
openid cloud_controller.read password.write cloud_controller.write","jti":"0f935cea42fd4516bb36e4088f6d7c44"}
```

Using `token_format` parameter you can requested token to be either:

- opaque
- jwt

In order to configure the UAA, it is recommended to use the `uaac` command\ line tool and target the UAA server:

```
$ gem install cf-uaac
$ uaac target http://localhost:8080/uaa
```

Check out user detail of user `marissa`:

```
$ uaac token owner get cf marissa -s "" -p koala
$ uaac contexts
```

This should also provide the relevant token details. Next, switch to the admin user (password is `adminsecret`):

```
$ uaac token client get admin -s adminsecret
```

Get a list of all configured clients:

```
$ uaac clients
```

This returns the interesting client app among others:

```
…
  app
    scope: cloud_controller.read cloud_controller.write openid password.write scim.userids organizations.acme
    resource_ids: none
    authorized_grant_types: password implicit authorization_code client_credentials refresh_token
    redirect_uri: http://localhost:8080/** http://localhost:8080/app/
    autoapprove: openid
    authorities: uaa.resource
    name: The Ultimate Oauth App
    signup_redirect_url: http://localhost:8080/app/
    change_email_redirect_url: http://localhost:8080/app/
    lastmodified: 1543879731285
…
```

We are going to use that `app` for Spring Cloud Data Flow but need to update the `redirect_uri`. We will also update the client secret:

```
$ uaac client update app --redirect_uri http://localhost:9393/login
$ uaac secret set app --secret dataflow
```

Now we can update the relavant OAuth configuration in Spring Cloud Data Flow:

```
security:
  oauth2:
    client:
      client-id: app
      client-secret: dataflow
      scope: openid, dataflow.view, dataflow.create, dataflow.manage    (1)
      access-token-uri: http://localhost:8080/uaa/oauth/token
      user-authorization-uri: http://localhost:8080/uaa/oauth/authorize
    resource:
      user-info-uri: http://localhost:8080/uaa/userinfo
```

(1) If you use scopes to identify roles, please make sure to also request the relevant roles

Role Mappings

By default all roles are assigned to users that login to Spring Cloud Data Flow. However, you can set the property:

`spring.cloud.dataflow.security.authorization.map-oauth-scopes: true`

This will instruct the the underlying `DefaultAuthoritiesExtractor` to map OAuth scopes to the respective authorities. The following scopes are supported:

- Scope `dataflow.view` maps to the `VIEW` role
- Scope `dataflow.create` maps to the `CREATE` role
- Scope `dataflow.manage` maps to the `MANAGE` role

Additionally you can also map arbitrary scopes to each of the Data Flow roles:

```
spring:
  cloud:
    deployer:
      local:
        free-disk-space-percentage: 0
    dataflow:
      security:
        authorization:
          map-oauth-scopes: true                              (1)
          role-mappings:
            ROLE_VIEW: dataflow.view                          (2)
            ROLE_CREATE: dataflow.create
            ROLE_MANAGE: dataflow.manage
```

(1) Enables explicit mapping support from OAuth scopes to Data Flow roles

(2) When role mapping support is enabled, you must provide a mapping for all 3 Spring Cloud Data Flow roles **ROLE_MANAGE**, **ROLE_VIEW**, **ROLE_CREATE**.

LDAP Authentication

LDAP Authentication (Lightweight Directory Access Protocol) is indirectly provided by Spring Cloud Data Flow using the UAA. The UAA itself provides comprehensive LDAP support.

While you may use your own OAuth2 authentication server, the LDAP support documented here requires using the UAA as authentication server. For any other provider, please consult the documentation for that particular provider.

The UAA supports authentication against an LDAP (Lightweight Directory Access Protocol) server using the following modes:

- Direct bind
- Search and bind
- Search and Compare

When integrating with an external identity provider such as LDAP, authentication within the UAA becomes **chained**. UAA first attempts to authenticate with a user's credentials against the UAA user store before the external provider, LDAP. For more information, see Chained Authentication in the *User Account and Authentication LDAP Integration* GitHub documentation.

LDAP Role Mapping

The OAuth2 authentication server (UAA), provides comprehensive support for mapping LDAP groups to OAuth scopes.

The following options exist:

- `ldap/ldap-groups-null.xml` No groups will be mapped
- `ldap/ldap-groups-as-scopes.xml` Group names will be retrieved from an LDAP attribute. E.g. `CN`
- `ldap/ldap-groups-map-to-scopes.xml` Groups will be mapped to UAA groups using the external_group_mapping table

These values are specified via the configuration property `ldap.groups.file controls`. Under the covers these values reference a Spring XML configuration file.

During test and development it might be necessary to make frequent changes to LDAP groups and users and see those reflected in the UAA. However, user information is cached for the duration of the login. The following script helps to retrieve the updated information quickly:

```
#!/bin/bash
uaac token delete --all
uaac target http://localhost:8080/uaa
uaac token owner get cf <username> -s "" -p  <password>
uaac token client get admin -s adminsecret
uaac user get <username>
```

For more information and sample application, please see the followingGitHub repository.

Shell Authentication

When using traditional authentication with the Data Flow Shell, you typically provide a username and password by using command-line arguments, as shown in the following example:

```
$ java -jar target/spring-cloud-dataflow-shell-2.0.0.BUILD-SNAPSHOT.jar  \
  --dataflow.username=myuser                                             \  (1)
  --dataflow.password=mysecret                                              (2)
```

(1) If authentication is enabled, the username must be provided.

(2) If the password is not provided, the shell prompts for it.

Alternatively, you can target a Data Flow Server also from within the shell, as follows:

```
server-unknown:>dataflow config server
  --uri  http://localhost:9393                                           \  (1)
  --username myuser                                                      \  (2)
  --password mysecret                                                    \  (3)
  --skip-ssl-validation  true                                            \  (4)
```

(1) Optional, defaults to localhost:9393.

(2) Mandatory if security is enabled.

(3) If security is enabled, and the password is not provided, the user is prompted for it.

(4) Optional, ignores certificate errors (when using self-signed certificates). Use cautiously!

The following image shows a typical shell command to connect to and authenticate a Data Flow Server:

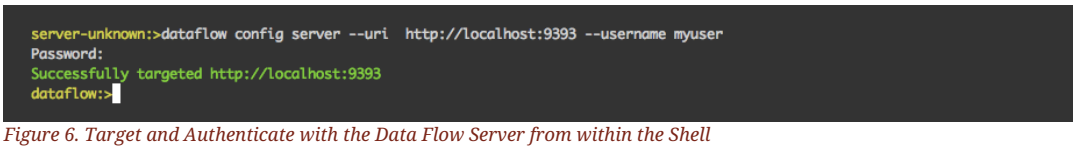


Figure 6. Target and Authenticate with the Data Flow Server from within the Shell

Customizing Authorization

The preceding content deals with mostly with authentication - that is, how to assess the identity of the user. Irrespective of the option chosen, you can also customize **authorization** - that is, who can do what.

The default scheme uses three roles to protect theREST 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 - for example, in `application.yml`. The configuration takes the form of a YAML list (as some rules may have precedence over others). Consequently, you need to copy and 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 following snippet may be outdated.

The default rules are as follows:

```
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')

            # Stream Validations

            - GET /streams/validation/              => hasRole('ROLE_VIEW')
            - GET /streams/validation/*             => hasRole('ROLE_VIEW')

            # 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')

            # Task Schedules

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

            # Task Validations

            - GET    /tasks/validation/              => hasRole('ROLE_VIEW')
            - GET    /tasks/validation/*             => hasRole('ROLE_VIEW')

            # Tools

            - POST   /tools/**                      => 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 Expression-Based Access Control.
- Each of those separated by one or several blank characters (spaces, tabs, and so on)

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.

Authorization - Shell and Dashboard Behavior

When security is enabled, the dashboard and the shell are role-aware, meaning that, depending on the assigned roles, not all functionality may be visible.

For instance, shell commands for which the user does not have the necessary roles are marked as unavailable.

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

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

16.8. Monitoring and Management

The Spring Cloud Data Flow server is a Spring Boot 1.5 application that includes theActuator 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` which is a discovery page for available managerment 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 enable sensitive endpoints, you should alsosecure 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 toggle for analytics is enabled, a Redis server is also required. The Data Flow server autoconfigures 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.

16.8.1. Monitoring Deployed Stream Applications

The stream applications deployed by Spring Cloud Data Flow can be based on Spring Boot 1.5 or Spring Boot 2.0. Both versions contains several features for monitoring your application in production. However, Spring Boot 1.x and 2.x as well as Spring Cloud Stream 1.x and 2.x differ in how monitoring is implemented. Since Spring Cloud Data Flow supports deploying 1.x and 2.x applications, we will cover both cases individually.

What is common across 1.x and 2.x applications is that Spring Cloud Stream apps can be configured to publish metrics to a messaging middleware destination. The Spring Cloud Data Flow Metrics Collector subscribes to this destination and aggregates metrics into a stream based view. The Metrics Collector 2.0 server supports collecting metrics from streams that contain only Boot 1.x or 2.x apps as well as streams that contain a mixture of Boot versions. The Data Flow UI queries the Metrics collector over HTTP to display messages rates next to each deployed application.

The following image shows the architecture when using Spring Cloud Stream's metrics publisher, the Metrics Collector, and the Data Flow server:

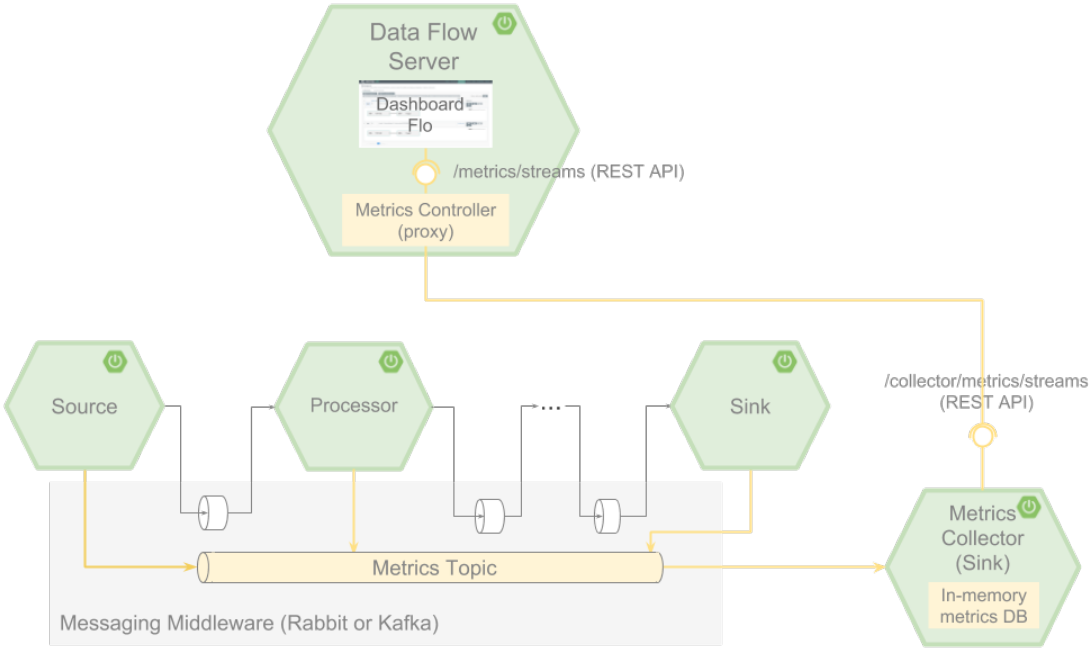


Figure 7. Spring Cloud Data Flow Metrics Architecture

Using the Metrics Collector

There are two versions of the Metrics Collector, a 1.0 version based on Spring Boot 1.0 that understands how to aggregate metrics from Spring Boot 1.x applications and a 2.0 version based on Spring Boot 2.0 that understands how to aggregate metrics from Spring Boot 1.x and 2.x. There is a Metrics Collector server for Rabbit and Kafka. You can find more information on downloading and running the Metrics Collector on its project page.

The Data Flow server property: `spring.cloud.dataflow.metrics.collector.uri` references the URI the Metrics Collector. For example, if you run the Metrics Collector locally on port `8080`, this is how you start local Data Flow server to reference the Metrics Collector.

```
$ java -jar spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.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` when starting the Data Flow server.

The metrics for each application are published when the property `spring.cloud.stream.bindings.applicationMetrics.destination` is set. Using a destination name of `metrics` is a good choice as the Metrics Collector subscribes to that name by default.

Since it is quite common to want all stream applications deployed by Data Flow to emit metrics, setting the property:

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

on the Data Flow server will let you configure support for metrics publication in one central location.

Using a destination name of `metrics` is a good choice as the Metrics Collector subscribes to that name by default, which of course, can be overridden to be different than the default as needed.

The next most common way to configure the metrics destination is to use deployment properties. The following example shows 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"
```

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 SCDF's Flo and it's visual representation of streaming pipelines. It is also accessed to enrich the Data Flow `/runtime/apps` endpoint that is exposed in the UI in the `Runtime` tab and in the shell through the `runtime apps` command with message rates.

The following image shows the message rates as they appear in the Streams tab of the UI:

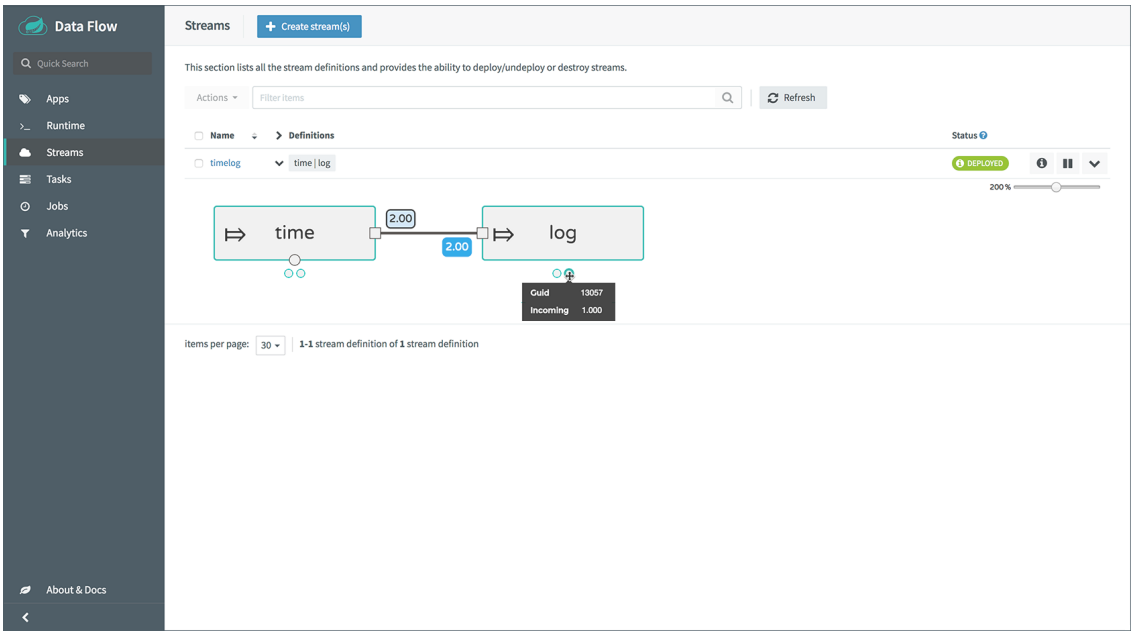


Figure 8. Stream Message Rates

When deploying applications, Data Flow sets the `spring.cloud.stream.metrics.properties` property, as shown in the following example:

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

The values of these keys are used as the tags to perform aggregation. In the case of 2.x applications, these key-values map directly onto tags in the Micrometer library. The property `spring.cloud.application.guid` can be used to track back to the specific application instance that generated the metric. The `guid` value is platform-dependent.

Data Flow also sets the application property that controls which metric values are exported. For 1.x applications, the property is `spring.metrics.export.triggers.application.includes` and the default value is is shown below:

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

For 2.x applications, the property is `spring.cloud.stream.metrics.meter-filter` and it does not have a default value, so all metrics are exported.

Note that the Data Flow UI only displays instantaneous input and output channel message rates. Data Flow does not provide its own implementation to store and visualize historical metrics data, instead we integrate with existing monitoring systems. For Boot 1.x, there is support for sending metrics to the application log and Datadog. For Boot 2.x, metrics is backed by the Micrometer library that provides a wide range of monitoring systems.

Spring Boot 2.x

We have developed a sample application that show how to modify the `time` and `log` applications and export metrics to InfluxDB using the micrometer library. A Grafana front end is also provided. This is a WIP, so check the Spring Cloud Data Flow Samples Repository for the latest status.

Spring Boot 1.x

Each deployed application contains 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. 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.

A few additional Spring projects provide support for sending metrics data to external systems:

- 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.

To make use of this functionality, you need to build the Stream application with the additional pom dependency of the MetricWriter implementation you want to use. To customize the "out of the box" Stream applications, use the Spring Cloud Stream Initializr to generate a project and then modify the pom. The documentation on the Data Flow Metrics project pages provides the additional information you need to get started.

16.9. About Configuration

The Spring Cloud Data Flow About Restful API result contains a display name, version, and, if specified, a URL for each of the major dependencies that comprise Spring Cloud Data Flow. The result (if enabled) also contains the sha1 and or sha256 checksum values for the shell dependency. The information that is returned for each of the dependencies is configurable by setting the following properties:

- spring.cloud.dataflow.version-info.spring-cloud-dataflow-core.name: the name to be used for the core.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-core.version: the version to be used for the core.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-dashboard.name: the name to be used for the dashboard.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-dashboard.version: the version to be used for the dashboard.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-implementation.name: the name to be used for the implementation.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-implementation.version: the version to be used for the implementation.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.name: the name to be used for the shell.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.version: the version to be used for the shell.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.url: the URL to be used for downloading the shell dependency.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.checksum-sha1: the sha1 checksum value that is returned with the shell dependency info.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.checksum-sha256: the sha256 checksum value that is returned with the shell dependency info.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.checksum-sha1-url: if the `spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.checksum-sha1` is not specified, SCDF uses the contents of the file specified at this URL for the checksum.
- spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.checksum-sha256-url: if the `spring.cloud.dataflow.version-info.spring-cloud-dataflow-shell.checksum-sha256` is not specified, SCDF uses the contents of the file specified at this URL for the checksum.

16.9.1. Enabling Shell Checksum values

By default, checksum values are not displayed for the shell dependency. If you need this feature enabled, set the `spring.cloud.dataflow.version-info.dependency-fetch.enabled` property to true.

16.9.2. Reserved Values for URLs

There are reserved values (surrounded by curly braces) that you can insert into the URL that will make sure that the

links are up to date:

- repository: if using a build-snapshot, milestone, or release candidate of Data Flow, the repository refers to the repo-spring-io repository. Otherwise, it refers to Maven Central.
- version: Inserts the version of the jar/pom.

For example, `myrepository/org/springframework/cloud/spring-cloud-dataflow-shell/{version}/spring-cloud-dataflow-shell-{version}.jar` produces `myrepository/org/springframework/cloud/spring-cloud-dataflow-shell/1.2.3.RELEASE/spring-cloud-dataflow-shell-1.2.3.RELEASE.jar` if you were using the 1.2.3.RELEASE version of the Spring Cloud Data Flow Shell

---

17. Configuration - Cloud Foundry

This section describes how to configure Spring Cloud Data Flow server's features, such as security and which relational database to use. It also describes how to configure Spring Cloud Data Flow shell's features.

17.1. Feature Toggles

Data Flow server offers a specific set of features that you can enable or disable when launching. These features include all the lifecycle operations and REST endpoints (server, client implementations including Shell and the UI) for:

- Streams
- Tasks
- Analytics

You can enable or disable these features by setting the following boolean properties when you launch 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 features are enabled. Note: Since the analytics feature is enabled by default, the Data Flow server is expected to have a valid Redis store available as its analytic repository (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 enable HTTP endpoints to read analytics data written to Redis, you can disable the analytics feature by using the `spring.cloud.dataflow.features.analytics-enabled` property to `false`.

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

17.2. Deployer Properties

You can also set other optional properties that alter the way Spring Cloud Data Flow deploys task apps to Cloud Foundry:

- You can configure the default memory and disk sizes for a deployed application. By default, they are 1024 MB memory and 1024 MB disk. To change these to (for example) 512 and 2048 respectively, use the following commands:

  [source]\

  ```
  e.g.
  ```

- You can set the buildpack that is 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_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_BUILDPACK java_buildpack_offline
  ```

- You can customize the health check mechanism used by Cloud Foundry to assert whether apps are running by using the `SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_HEALTH_CHECK` environment variable. The current supported options are `http` (the default), `port`, and `none`.

  You can also set environment variables that specify the the HTTP-based health check endpoint and timeout: `SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_HEALTH_CHECK_ENDPOINT`

and `SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_HEALTH_CHECK_TIMEOUT`, respectively. These default to `/health` (the Spring Boot default location) and `120` seconds.

- You can also specify deployment properties by using the DSL. For instance, if you want to set the allocated memory for the `http` application to 512m and also bind a mysql service to the `jdbc` application, you can run the following commands:

```
dataflow:> stream create --name mysqlstream --definition "http | jdbc --tableName=names --columns=name"
dataflow:> stream deploy --name mysqlstream --properties "deployer.http.memory=512,
deployer.jdbc.cloudfoundry.services=mysql"
```

You can configure these settings separately for stream and task apps. To alter settings for tasks, substitute `TASK` for `STREAM` in the property name, as the following example shows:

```
cf set-env dataflow-server
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_TASK_MEMORY
512
```

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

17.3. Application Names and Prefixes

To help avoid clashes with routes across spaces in Cloud Foundry, a naming strategy that provides a random prefix to a deployed application is available and is enabled by default. You can override the default configurations and set the respective properties by using `cf set-env` commands.

For instance, if you want to disable the randomization, you can override it by using the following command:

```
cf set-env dataflow-server
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_ENABLE_RANDOM_APP_NAME_PREFIX false
```

17.4. Custom Routes

As an alternative to a random name or to get even more control over the hostname used by the deployed apps, you can use custom deployment properties, as the following example shows:

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

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

The preceding example binds the `http` app to the `myhost.mydomain.com/my-path` URL. Note that this example shows **all** of the available customization options. In practice, you can use only one or two out of the three.

17.5. Docker Applications

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

If you use Spring Boot and RabbitMQ-based Docker images, you can provide a common deployment property to facilitate binding the apps 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, you can use something similar to the following, if you use 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 applications, your Docker app must parse the `VCAP_SERVICES` variable in order to bind to any available services.

Passing application properties

When using non-Boot applications, chances are that you want to pass the application properties by using traditional environment variables, as opposed to using the special `SPRING_APPLICATION_JSON` variable. To do so, set the following variables for streams and tasks, respectively:

```
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_USE_SPRING_APPLICATION_JSON=false
```

17.6. 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 want to provide a `mysql` service binding only for the `jdbc` application in the following stream definition, you can pass the service binding as a deployment property:

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

where `mysqlService` is the name of the service specifically bound only to the `jdbc` application and the `http` application does not get the binding by this method.

If you have more than one service to bind, they can be passed as comma-separated items (for example: `deployer.jdbc.cloudfoundry.services=mysqlService,someService`).

17.7. 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 User-provided Services as well, whether for use as the messaging middleware (for example, if you want to use an external Apache Kafka installation) or for use by some of the stream applications (for example, an Oracle Database).

Now we review an example of extracting and supplying the connection credentials from a UPS.

The following example shows a sample UPS setup for Apache Kafka:

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

The UPS credentials are wrapped within `VCAP_SERVICES`, and they can be supplied directly in the stream definition, as the following example shows.

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

17.8. Database Connection Pool

The Data Flow server uses the Spring Cloud Connector library to create the DataSource with a default connection pool size of 4. To change the connection pool size and maximum wait time, set the following two properties `spring.cloud.dataflow.server.cloudfoundry.maxPoolSize` and `"spring.cloud.dataflow.server.cloudfoundry.maxWaitTime`. The wait time is specified in milliseconds.

17.9. Maximum Disk Quota

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 by using Pivotal Cloud Foundry's (PCF) Ops Manager GUI.

This configuration is relevant for Spring Cloud Data Flow because every 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 and reuse. With this happening in the background, the default disk quota (1G) can fill up rapidly, especially when we experiment 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.

17.9.1. PCF's Operations Manager

From PCF's Ops Manager, select the "Pivotal Elastic Runtime" tile and navigate to the "Application Developer Controls" tab. Change the "Maximum Disk Quota per App (MB)" setting from 2048 (2G) to 10240 (10G). Save the disk quota update and click "Apply Changes" to complete the configuration override.

17.10. Scale Application

Once the disk quota change has been successfully applied and assuming you have a running application, you can scale the application with a new `disk_limit` through the CF CLI, as the following example shows:

```
→ 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
```

You can then list the applications and see the new maximum disk space, as the following example shows:

```
→ 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
```

17.11. Managing Disk Use

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. If you deploy the Data Flow server by using the default `port` health check type, you must explicitly monitor the disk space on the server in order to avoid running out space. If you deploy the server by using the `http` health check type (see the next example), the Data Flow server is restarted if there is low disk space. This is due to Spring Boot's Disk Space Health Indicator. You can configure the settings of the Disk Space Health Indicator by using the properties that have the `management.health.diskspace` prefix.

For version 1.7, we are investigating the use of Volume Services for the Data Flow server to store `.jar` artifacts before pushing them to Cloud Foundry.

The following example shows how to deploy the `http` health check type to an endpoint called `/management/health`:

```
---
  ...
  health-check-type: http
  health-check-http-endpoint: /management/health
```

17.12. Application Resolution Alternatives

Though we highly recommend using Maven Repository for application resolution and registration in Cloud Foundry, there might be situations where an alternative approach would make sense. The following alternative options could help you resolve applications when running on Cloud Foundry.

- With the help of Spring Boot, we can serve static content in Cloud Foundry. A simple Spring Boot application can bundle all the required stream and task applications. By having it run on Cloud Foundry, the static application can

then serve the über-jar's. From the shell, you can, for example, register the application with the name `http-source.jar` by using `--uri=http://<Route-To-StaticApp>/http-source.jar`.

- The über-jar's can be hosted on any external server that's reachable over HTTP. They can be resolved from raw GitHub URIs as well. From the shell, you can, for example, register the app with the name `http-source.jar` by using `--uri=http://<Raw_GitHub_URI>/http-source.jar`.
- Static Buildpack support in Cloud Foundry is another option. A similar HTTP resolution works on this model, too.
- Volume Services is another great option. The required über-jars can be hosted in an external file system. With the help of volume-services, you can, for example, register the application with the name `http-source.jar` by using `--uri=file://<Path-To-FileSystem>/http-source.jar`.

17.13. Database Connection Pool

The Data Flow server uses the Spring Cloud Connector library to create the DataSource with a default connection pool size of `4`. To change the connection pool size and maximum wait time, set the following two properties `spring.cloud.dataflow.server.cloudfoundry.maxPoolSize` and `spring.cloud.dataflow.server.cloudfoundry.maxWaitTime`. The wait time is specified in milliseconds.

17.14. 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 (UAA and SSO running on Cloud Foundry), see the security section from the core reference guide. You can configure the security details in `dataflow-server.yml` or pass them as environment variables through `cf set-env` commands.

17.14.1. Authentication and Cloud Foundry

Spring Cloud Data Flow can either integrate with Pivotal Single Sign-On Service (for example, 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 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.

To do so, bind the Pivotal Single Sign-On Service to your Data Flow Server application and provide the following properties:

```
SPRING_CLOUD_DATAFLOW_SECURITY_CFUSEUAA: false                                    (1)
SECURITY_OAUTH2_CLIENT_CLIENTID: "${security.oauth2.client.clientId}"
SECURITY_OAUTH2_CLIENT_CLIENTSECRET: "${security.oauth2.client.clientSecret}"
SECURITY_OAUTH2_CLIENT_ACCESSTOKENURI: "${security.oauth2.client.accessTokenUri}"
SECURITY_OAUTH2_CLIENT_USERAUTHORIZATIONURI: "${security.oauth2.client.userAuthorizationUri}"
SECURITY_OAUTH2_RESOURCE_USERINFOURI: "${security.oauth2.resource.userInfoUri}"
```

(1) It is important that the property `spring.cloud.dataflow.security.cf-use-uaa` is set to `false`

Authorization is similarly supported for non-Cloud Foundry security scenarios. See the security section from the core Data Flow reference guide.

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

You can customize this behavior by providing your own `AuthoritiesExtractor`.

The following example shows one possible approach to set the custom `AuthoritiesExtractor` on the `UserInfoTokenServices`:

```
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;
 }
}
```

Then you can declare it in your configuration class as follows:

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

Cloud Foundry UAA

The availability of Cloud Foundry User Account and Authentication (UAA) depends on the Cloud Foundry environment. In order to provide UAA integration, you have to provide the necessary OAuth2 configuration properties (for example, by setting the `SPRING_APPLICATION_JSON` property).

The following JSON example shows how to create a security configuration:

```
{
  "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 `spring.cloud.dataflow.security.cf-use-uaa` property is set to `true`. This property activates a special `AuthoritiesExtractor` called `CloudFoundryDataflowAuthoritiesExtractor`.

If you do not use CloudFoundry UAA, you should set `spring.cloud.dataflow.security.cf-use-uaa` to `false`.

Under the covers, this `AuthoritiesExtractor` calls out to the Cloud Foundry Apps API and ensure that users are in fact Space Developers.

If the authenticated user is verified as a Space Developer, all roles are assigned. Otherwise, no roles whatsoever are assigned. In that case, you may see the following Dashboard screen:

Figure 9. Accessing the Data Flow Dashboard without Roles

17.15. Configuration Reference

You must provide several pieces of configuration. These are Spring Boot `@ConfigurationProperties`, so you can set them as environment variables or by any other means that Spring Boot supports. The following listing is in environment variable format, as that is an easy way to get started configuring Boot applications in Cloud Foundry. Note that in the future, you will be able to deploy tasks to multiple platforms, but for 2.0.0.M1 you can deploy only to a single platform and the name must be `default`.

```
# Default values appear after the equal signs.
# Example values, typical for Pivotal Web Services, are included as comments.

# URL of the CF API (used when using cf login -a for example) - for example, https://api.run.pivotal.io
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL=

# The name of the organization that owns the space above - for example, youruser-org
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG=

# The name of the space into which modules will be deployed - for example, development
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE=

# The root domain to use when mapping routes - for example, cfapps.io
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_DOMAIN=

# The user name and password of the user to use to create applications
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME=
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD

# Whether to allow self-signed certificates during SSL validation (you should NOT do so in production)
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SKIP_SSL_VALIDATION

# A comma-separated set of service instance names to bind to every deployed task application.
# Among other things, this should include an RDBMS service that is used
# for Spring Cloud Task execution reporting, such as my_mysql
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES
spring.cloud.deployer.cloudfoundry.task.services=

# Timeout, in seconds, to use when doing blocking API calls to Cloud Foundry
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_API_TIMEOUT=

# Timeout, in milliseconds, to use when querying the Cloud Foundry API to compute app status
```

```
SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_STATUS_TIMEOUT
```

Note that you can set `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, as the following example shows:

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

The commands in the preceding example deploy the time source with 2048MB of memory, while the log sink uses the default 1024MB.

When you deploy a stream, you can also pass `JAVA_OPTS` as a deployment property, as the following example shows:

```
stream deploy --name ticktock --properties "deployer.time.cloudfoundry.javaOpts=-Duser.timezone=America/New_York"
```

17.16. Debugging

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 feature 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 it is disabled by default.

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

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

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

17.17. Spring Cloud Config Server

You can use Spring Cloud Config Server to centralize configuration properties for Spring Boot applications. Likewise, both Spring Cloud Data Flow and the applications orchestrated by Spring Cloud Data Flow can be integrated with a configuration server to use the same capabilities.

17.17.1. Stream, Task, and Spring Cloud Config Server

Similar to Spring Cloud Data Flow server, you can configure both the stream and task applications to resolve the centralized properties from the configuration server. Setting the `spring.cloud.config.uri` property for the deployed applications is a common way to bind to the configuration 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 `spring.cloud.dataflow.applicationProperties.stream` property for stream applications and `spring.cloud.dataflow.applicationProperties.task` property for task applications can be used to pass the `uri` of the Config Server to each deployed stream or task application. See the section on common application properties for more information.

Note that, if you use applications from the App Starters project, these applications already embed the `spring-cloud-services-starter-config-client` dependency. If you build your application from scratch and want to add the client side support for config server, you can add a dependency reference to the config server client library. The following snippet shows a Maven example:

```
...
<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 the Spring Cloud Config Server client for Pivotal

Cloud Foundry.

You may see a `WARN` logging message if the application that uses this library cannot connect to the configuration server when the application starts and whenever the `/health` endpoint is accessed. If you know that you are not using config server functionality, you can disable the client library by setting the `SPRING_CLOUD_CONFIG_ENABLED` environment variable to `false`.

17.17.2. Sample Manifest Template

The following SCDF and Skipper `manifest.yml` templates includes the required environment variables for the Skipper and Spring Cloud Data Flow server and deployed applications and tasks to successfully run on Cloud Foundry and automatically resolve centralized properties from `my-config-server` at 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
    MAVEN_REMOTE_REPOSITORIES_REPO1_URL: https://repo.spring.io/libs-snapshot
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL: https://api.sys.huron.cf-
app.com
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG: sabby20
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE: sabby20
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_DOMAIN: apps.huron.cf-app.com
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME: admin
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD: ***
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SKIP_SSL_VALIDATION: true
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES: mysql
    SPRING_CLOUD_SKIPPER_CLIENT_SERVER_URI: https://<skipper-host-name>/api
services:
- mysql
- my-config-server

---
applications:
- name: skipper-server
  host: skipper-server
  memory: 1G
  disk_quota: 1G
  instances: 1
  timeout: 180
  buildpack: java_buildpack
  path: <PATH TO THE DOWNLOADED SKIPPER SERVER UBER-JAR>
  env:
    SPRING_APPLICATION_NAME: skipper-server
    SPRING_CLOUD_SKIPPER_SERVER_ENABLE_LOCAL_PLATFORM: false
    SPRING_CLOUD_SKIPPER_SERVER_STRATEGIES_HEALTHCHECK_TIMEOUTINMILLIS: 300000
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL: https://api.local.pcfdev.io
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG: pcfdev-org
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE: pcfdev-space
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_DOMAIN: cfapps.io
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME: admin
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD: admin
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SKIP_SSL_VALIDATION: false
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_DELETE_ROUTES: false
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES: rabbit, my-config-
server
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 Spring Cloud Data Flow server, Spring Cloud Task and via Skipper to all the Spring Cloud Stream applications respectively, we can now resolve centralized properties backed by this service.

17.17.3. 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 configuration server for Pivotal Cloud Foundry uses HTTPS for all client-to-service communication, so we need to add a self-signed SSL certificate in environments with no valid certificates.

By using the same `manifest.yml` templates listed in the previous section for the server, we can provide the self-signed SSL certificate by setting `TRUST_CERTS: <API_ENDPOINT>`.

However, the deployed applications also require `TRUST_CERTS` as a flat environment variable (as opposed to being wrapped inside `SPRING_APPLICATION_JSON`), so we must instruct the server with yet another set of tokens (`SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_USE_SPRING_APPLICATION_JSON:`

`false`) for tasks. With this setup, the applications receive their application properties as regular environment variables.

The following listing shows the updated `manifest.yml` with the required changes. Both the Data Flow server and deployed applications get their configuration 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-VERSION.jar
  env:
    SPRING_APPLICATION_NAME: test-server
    MAVEN_REMOTE_REPOSITORIES_REPO1_URL: https://repo.spring.io/libs-snapshot
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL: https://api.sys.huron.cf-
app.com
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG: sabby20
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE: sabby20
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_DOMAIN: apps.huron.cf-app.com
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME: admin
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD: ***
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SKIP_SSL_VALIDATION: true
    SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES: mysql, config-server
    SPRING_CLOUD_SKIPPER_CLIENT_SERVER_URI: https://<skipper-host-name>/api
    TRUST_CERTS: <API_ENDPOINT> #this is for the server
    SPRING_CLOUD_DATAFLOW_APPLICATION_PROPERTIES_TASK_TRUST_CERTS: <API_ENDPOINT>   #this propagates to all tasks
services:
- mysql
- my-config-server #this is for the server
```

Also add the `my-config-server` service to the Skipper's manifest environment

```
---
applications:
- name: skipper-server
  host: skipper-server
  memory: 1G
  disk_quota: 1G
  instances: 1
  timeout: 180
  buildpack: java_buildpack
  path: <PATH TO THE DOWNLOADED SKIPPER SERVER UBER-JAR>
  env:
    SPRING_APPLICATION_NAME: skipper-server
    SPRING_CLOUD_SKIPPER_SERVER_ENABLE_LOCAL_PLATFORM: false
    SPRING_CLOUD_SKIPPER_SERVER_STRATEGIES_HEALTHCHECK_TIMEOUTINMILLIS: 300000
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL: <URL>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG: <ORG>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE: <SPACE>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_DOMAIN: <DOMAIN>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME: <USER>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD: <PASSWORD>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES: rabbit, my-config-
server #this is so all stream applications bind to my-config-server
services:
- mysql
  my-config-server
```

17.18. Configure Scheduling

This section discusses how to configure Spring Cloud Data Flow to connect to thePCF-Scheduler as its agent to execute tasks.

Before following these instructions, be sure to have an instance of the PCF-Scheduler service running in your Cloud Foundry space. To create a PCF-Scheduler in your space (assuming it is in your Market Place) execute the following from the CF CLI: `cf create-service scheduler-for-pcf standard <name of service>`. Name of a service is later used to bound running application in*PCF*.

For scheduling, you must add (or update) the following environment variables in your environment:

- Enable scheduling for Spring Cloud Data Flow by setting `spring.cloud.dataflow.features.schedules-enabled` to `true`.
- Bind the task deployer to your instance of PCF-Scheduler by adding the PCF-Scheduler service name to the `SPRING_CLOUD_DATAFLOW_TASK_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES` environment variable.
- Establish the URL to the PCF-Scheduler by setting the `SPRING_CLOUD_SCHEDULER_CLOUDFOUNDRY_SCHEDULER_URL`

environment variable.

After creating the preceding configurations, you must create any task definitions that need to be scheduled.

The following sample manifest shows both environment properties configured (assuming you have a PCF-Scheduler service available with the name `myscheduler`):

```
---
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_SKIPPER_SERVER_ENABLE_LOCAL_PLATFORM: false
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_URL: <URL>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_ORG: <ORG>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_SPACE: <SPACE>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_DOMAIN: <DOMAIN>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_USERNAME: <USER>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_CONNECTION_PASSWORD: <PASSWORD>
    SPRING_CLOUD_SKIPPER_SERVER_PLATFORM_CLOUDFOUNDRY_ACCOUNTS[default]_DEPLOYMENT_SERVICES: rabbit, myscheduler
    SPRING_CLOUD_DATAFLOW_FEATURES_SCHEDULES_ENABLED: true
    SPRING_CLOUD_SKIPPER_CLIENT_SERVER_URI: https://<skipper-host-name>/api
    SPRING_CLOUD_SCHEDULER_CLOUDFOUNDRY_SCHEDULER_URL: https://scheduler.local.pcfdev.io
services:
- mysql
```

Where the `SPRING_CLOUD_SCHEDULER_CLOUDFOUNDRY_SCHEDULER_URL` has the following format: `scheduler.<Domain-Name>` (for example, `scheduler.local.pcfdev.io`). Check the actual address from your *PCF* environment.

---

18. Configuration - Kubernetes

This section describes how to configure Spring Cloud Data Flow features, such as security and which relational database to use.

18.1. Feature Toggles

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

- Streams
- Tasks
- Analytics
- Schedules

You can enable or disable these features by setting the following boolean environment variables when launching the Data Flow server:

- `SPRING_CLOUD_DATAFLOW_FEATURES_STREAMS_ENABLED`
- `SPRING_CLOUD_DATAFLOW_FEATURES_TASKS_ENABLED`
- `SPRING_CLOUD_DATAFLOW_FEATURES_ANALYTICS_ENABLED`
- `SPRING_CLOUD_DATAFLOW_FEATURES_SCHEDULES_ENABLED`

By default, all the features are enabled.

Since the analytics feature is enabled by default, the Data Flow server is expected to have a valid Redis store available as an analytics repository. Consequently, 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 enable HTTP endpoints to read analytics data written to Redis, you can disable the analytics feature by setting the property mentioned earlier to `false`.

The `/features` REST endpoint provides information on the features that have been enabled and disabled.

18.2. General Configuration

The Spring Cloud Data Flow server for Kubernetes uses the `spring-cloud-kubernetes` module process both the ConfigMap and the secrets settings. To enable the ConfigMap support, pass in an environment variable of `SPRING_CLOUD_KUBERNETES_CONFIG_NAME` and set it to the name of the ConfigMap. The same is true for the secrets, where the environment variable is `SPRING_CLOUD_KUBERNETES_SECRETS_NAME`. To use the secrets, you also need to set `SPRING_CLOUD_KUBERNETES_SECRETS_ENABLE_API` to `true`.

The following example shows a snippet from a deployment script that sets these environment variables:

```
env:
- name: SPRING_CLOUD_KUBERNETES_SECRETS_ENABLE_API
  value: 'true'
- name: SPRING_CLOUD_KUBERNETES_SECRETS_NAME
  value: mysql
- name: SPRING_CLOUD_KUBERNETES_CONFIG_NAME
  value: scdf-server
```

18.2.1. Using ConfigMap and Secrets

You can pass configuration properties to the Data Flow Server by using KubernetesConfigMap and secrets.

The following example shows one possible configuration, which enables RabbitMQ, MySQL and Redis as well as basic security settings for the server:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: scdf-server
  labels:
    app: scdf-server
data:
  application.yaml: |-
    security:
      basic:
        enabled: true
        realm: Spring Cloud Data Flow
    spring:
      cloud:
        dataflow:
          security:
            authentication:
              file:
                enabled: true
                users:
                  admin: admin, ROLE_MANAGE, ROLE_VIEW
                  user: password, ROLE_VIEW, ROLE_CREATE
        deployer:
          kubernetes:
            environmentVariables:
'SPRING_RABBITMQ_HOST=${RABBITMQ_SERVICE_HOST},SPRING_RABBITMQ_PORT=${RABBITMQ_SERVICE_PORT},SPRING_REDIS_HOST=${RE
DIS_SERVICE_HOST},SPRING_REDIS_PORT=${REDIS_SERVICE_PORT}'
      datasource:
        url: jdbc:mysql://${MYSQL_SERVICE_HOST}:${MYSQL_SERVICE_PORT}/mysql
        username: root
        password: ${mysql-root-password}
        driverClassName: org.mariadb.jdbc.Driver
        testOnBorrow: true
        validationQuery: "SELECT 1"
      redis:
        host: ${REDIS_SERVICE_HOST}
        port: ${REDIS_SERVICE_PORT}
```

The preceding example assumes that RabbitMQ is deployed with `rabbitmq` as the service name. For MySQL, it assumes that the service name is `mysql`. For Redis, it assumes that the service name is `redis`. Kubernetes publishes the host and port values of these services as environment variables that we can use when configuring the apps we deploy.

We prefer to provide the MySQL connection password in a Secrets file, as the following example shows:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql
  labels:
    app: mysql
data:
  mysql-root-password: eW91cnBhc3N3b3Jk
```

The password is a base64-encoded value.

18.3. Database Configuration

Spring Cloud Data Flow provides schemas for H2, HSQLDB, MySQL, Oracle, PostgreSQL, DB2, and SQL Server. The appropriate schema is automatically created when the server starts, provided the right database driver and appropriate credentials are in the classpath.

The JDBC drivers for MySQL (via MariaDB driver), HSQLDB, PostgreSQL, and embedded H2 are available out of the box. If you use any other database, you need to put the corresponding JDBC driver jar on the classpath of the server.

For instance, if you use MySQL in addition to a password in the secrets file, you could provide the following properties in the ConfigMap:

```
data:
  application.yaml: |-
    spring:
      datasource:
        url: jdbc:mysql://${MYSQL_SERVICE_HOST}:${MYSQL_SERVICE_PORT}/mysql
        username: root
        password: ${mysql-root-password}
        driverClassName: org.mariadb.jdbc.Driver
        url: jdbc:mysql://${MYSQL_SERVICE_HOST}:${MYSQL_SERVICE_PORT}/test
        driverClassName: org.mariadb.jdbc.Driver
```

For PostgreSQL, you could use the following configuration:

```
data:
  application.yaml: |-
    spring:
      datasource:
        url: jdbc:postgresql://${PGSQL_SERVICE_HOST}:${PGSQL_SERVICE_PORT}/database
        username: root
        password: ${postgres-password}
        driverClassName: org.postgresql.Driver
```

For HSQLDB, you could use the following configuration:

```
data:
  application.yaml: |-
    spring:
      datasource:
        url: jdbc:hsqldb:hsql://${HSQLDB_SERVICE_HOST}:${HSQLDB_SERVICE_PORT}/database
        username: sa
        driverClassName: org.hsqldb.jdbc.JDBCDriver
```

You can find migration scripts for specific database types in thespring-cloud-task repo.

18.4. Security

This section covers how to secure the server application in the sample configurations file used inGetting Started - Kubernetes.

This section covers the basic configuration settings in the sample configuration. See thecore security documentation for more detailed coverage of the security configuration options for the Spring Cloud Data Flow server and shell.

When using RabbitMQ as the transport, the security settings are located in the `src/kubernetes/server/server-config-rabbit.yaml` file. For Kafka, the settings are located in the `src/kubernetes/server/server-config-kafka.yaml` file. The following example shows a security configuration in YAML:

```
security:
  basic:
    enabled: true                                    (1)
    realm: Spring Cloud Data Flow                    (2)
spring:
  cloud:
    dataflow:
      security:
        authentication:
          file:
            enabled: true
            users:
              admin: admin, ROLE_MANAGE, ROLE_VIEW   (3)
              user: password, ROLE_VIEW, ROLE_CREATE (4)
```

(1) Enable security

(2) Optionally set the realm (defaults to `Spring`)

3. Create an 'admin' user with its password set to 'admin'. It can view applications, streams, and tasks and can also view management endpoints.
4. Create a 'user' user with its password set to 'password'. It can register applications, create streams and tasks, and view them.

Feel free to change user names and passwords to suit and move the definition of user passwords to a Kubernetes secret.

18.5. Monitoring and Management

We recommend using the `kubectl` command for troubleshooting streams and tasks.

You can list all artifacts and resources used by using the following command:

```
kubectl get all,cm,secrets,pvc
```

You can list all resources used by a specific application or service by using a label to select resources. The following command lists all resources used by the `mysql` service:

```
kubectl get all -l app=mysql
```

You can get the logs for a specific pod by issuing the following command:

```
kubectl logs pod <pod-name>
```

If the pod is continuously getting restarted, you can add `-p` as an option to see the previous log, as follows:

```
kubectl logs -p <pod-name>
```

You can also tail or follow a log by adding an `-f` option, as follows:

```
kubectl logs -f <pod-name>
```

A useful command to help in troubleshooting issues, such as a container that has a fatal error when starting up, is to use the `describe` command, as the following example shows:

```
kubectl describe pod ticktock-log-0-qnk72
```

18.5.1. Inspecting Server Logs

You can access the server logs by using the following command:

```
kubectl get pod -l app=scdf=server
kubectl logs <scdf-server-pod-name>
```

18.5.2. Streams

Stream applications are deployed with the stream name followed by the name of the application. For processors and sinks, an instance index is also appended.

To see all the pods that are deployed by the Spring Cloud Data Flow server, you can specify the `role=spring-app` label, as follows:

```
kubectl get pod -l role=spring-app
```

To see details for a specific application deployment you can use the following command:

```
kubectl describe pod <app-pod-name>
```

To view the application logs, you can use the following command:

```
kubectl logs <app-pod-name>
```

If you would like to tail a log you can use the following command:

```
kubectl logs -f <app-pod-name>
```

18.5.3. Tasks

Tasks are launched as bare pods without a replication controller. The pods remain after the tasks complete, which gives you an opportunity to review the logs.

To see all pods for a specific task, use the following command:

```
kubectl get pod -l task-name=<task-name>
```

To review the task logs, use the following command:

```
kubectl logs <task-pod-name>
```

You have two options to delete completed pods. You can delete them manually once they are no longer needed or you can use the Data Flow shell `task execution cleanup` command to remove the completed pod for a task execution.

To delete the task pod manually, use the following command:

```
kubectl delete pod <task-pod-name>
```

To use the `task execution cleanup` command, you must first determine the `ID` for the task execution. To do so, use the `task execution list` command, as the following example (with output) shows:

```
dataflow:>task execution list
╔═══════════╤══╤════════════════════════════╤════════════════════════════╤═════════╗
║ Task Name │ID│         Start Time         │          End Time          │Exit Code║
╠═══════════╪══╪════════════════════════════╪════════════════════════════╪═════════╣
║task1      │1 │Fri May 05 18:12:05 EDT 2017│Fri May 05 18:12:05 EDT 2017│0        ║
╚═══════════╧══╧════════════════════════════╧════════════════════════════╧═════════╝
```

Once you have the ID, you can issue the command to cleanup the execution artifacts (the completed pod), as the following example shows:

```
dataflow:>task execution cleanup --id 1
Request to clean up resources for task execution 1 has been submitted
```

18.6. Scheduling

This section covers customization of how scheduled tasks are configured. Scheduling of tasks is enabled by default in the Spring Cloud Data Flow Kubernetes Server. Properties are used to influence settings for scheduled tasks and can be configured on a global or per-schedule basis.

Unless noted, properties set on a per-schedule basis always take precedence over properties set as the server configuration. This arrangement allows for the ability to override global server level properties for a specific schedule.

See `KubernetesSchedulerProperties` for more on the supported options.

18.6.1. Entry Point Style

An Entry Point Style affects how application properties are passed to the task container to be deployed. Currently, three styles are supported:

- `exec` : (default) Passes all application properties as command line arguments.
- `shell` : Passes all application properties as environment variables.
- `boot` : Creates an environment variable called `SPRING_APPLICATION_JSON` that contains a JSON representation of all application properties.

You can configure the entry point style as follows:

```
scheduler.kubernetes.entryPointStyle=<Entry Point Style>
```

Replace `<Entry Point Style>` with your desired Entry Point Style.

You can also configure the Entry Point Style at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_ENTRY_POINT_STYLE
  value: entryPointStyle
```

Replace `entryPointStyle` with the desired Entry Point Style.

You should choose an Entry Point Style of either `exec` or `shell` , to correspond to how the `ENTRYPOINT` syntax is defined in the container's `Dockerfile` . For more information and uses cases on `exec` vs `shell` , see the ENTRYPOINT section of the Docker documentation.

Using the `boot` Entry Point Style corresponds to using the `exec` style `ENTRYPOINT` . Command line arguments from the deployment request are passed to the container, with the addition of application properties mapped into the `SPRING_APPLICATION_JSON` environment variable rather than command line arguments.

18.6.2. Environment Variables

To influence the environment settings for a given application, you can take advantage of the `spring.cloud.scheduler.kubernetes.environmentVariables` property. For example, a common requirement in production settings is to influence the JVM memory arguments. You can achieve this by using the `JAVA_TOOL_OPTIONS` environment variable, as the following example shows:

```
scheduler.kubernetes.environmentVariables=JAVA_TOOL_OPTIONS=-Xmx1024m
```

Additionally you can configure environment variables at the server level in the container `env` section of a deployment YAML, as the following example shows:

When specifying environment variables in the server configuration and on a per-schedule basis, environment variables will be merged. This allows for the ability to set common environment variables in the server configuration and more specific at the specific schedule level.

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_ENVIRONMENT_VARIABLES
  value: myVar=myVal
```

Replace `myVar=myVal` with your desired environment variables.

18.6.3. Image Pull Policy

An image pull policy defines when a Docker image should be pulled to the local registry. Currently, three policies are supported:

- `IfNotPresent` : (default) Do not pull an image if it already exists.
- `Always` : Always pull the image regardless of whether it already exists.

- `Never` : Never pull an image. Use only an image that already exists.

The following example shows how you can individually configure containers:

```
scheduler.kubernetes.imagePullPolicy=Always
```

Replace `Always` with your desired image pull policy.

You can configure an image pull policy at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_IMAGE_PULL_POLICY
  value: Always
```

Replace `Always` with your desired image pull policy.

18.6.4. Private Docker Registry

Docker images that are private and require authentication can be pulled by configuring a Secret. First, you must create a Secret in the cluster. Follow the Pull an Image from a Private Registry guide to create the Secret.

Once you have created the secret, use the `imagePullSecret` property to set the secret to use, as the following example shows:

```
scheduler.kubernetes.imagePullSecret=mysecret
```

Replace `mysecret` with the name of the secret you created earlier.

You can also configure the image pull secret at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_IMAGE_PULL_SECRET
  value: mysecret
```

Replace `mysecret` with the name of the secret you created earlier.

18.6.5. Namespace

By default the namespace used for scheduled tasks is `default` . This value can be set at the server level configuration in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_NAMESPACE
  value: mynamespace
```

18.6.6. Service Account

You can configure a custom service account for scheduled tasks through properties. An existing service account can be used or a new one created. One way to create a service account is by using `kubectl` , as the following example shows:

```
$ kubectl create serviceaccount myserviceaccountname
serviceaccount "myserviceaccountname" created
```

Then you can configure the service account to use on a per-schedule basis as follows:

```
scheduler.kubernetes.taskServiceAccountName=myserviceaccountname
```

Replace `myserviceaccountname` with your service account name.

You can also configure the service account name at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_TASK_SERVICE_ACCOUNT_NAME
  value: myserviceaccountname
```

Replace `myserviceaccountname` with the service account name to be applied to all deployments.

For more information on scheduling tasks see Scheduling Tasks.

18.7. Debug Support

Debugging the Spring Cloud Data Flow Kubernetes Server and included components (such as the Spring Cloud Kubernetes Deployer) is supported through the Java Debug Wire Protocol (JDWP). This section outlines an approach to manually enable debugging and another approach that uses configuration files provided with Spring Cloud Data Flow Server Kubernetes to "patch" a running deployment.

JDWP itself does not use any authentication. This section assumes debugging is being done on a local development environment (such as Minikube), so guidance on securing the debug port is not provided.

18.7.1. Enabling Debugging Manually

To manually enable JDWP, first edit `src/kubernetes/server/server-deployment.yaml` and add an additional `containerPort` entry under `spec.template.spec.containers.ports` with a value of `5005`. Additionally, add the `JAVA_TOOL_OPTIONS` environment variable under `spec.template.spec.containers.env` as the following example shows:

```
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: scdf-server
        ...
        ports:
        ...
  - containerPort: 5005
        env:
        - name: JAVA_TOOL_OPTIONS
          value: '-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005'
```

The preceding example uses port 5005, but it can be any number that does not conflict with another port. The chosen port number must also be the same for the added `containerPort` value and the `address` parameter of the `JAVA_TOOL_OPTIONS` `-agentlib` flag, as shown in the preceding example.

You can now start the Spring Cloud Data Flow Kubernetes Server. Once the server is up, you can verify the configuration changes on the `scdf-server` deployment, as the following example (with output) shows:

```
kubectl describe deployment/scdf-server
...
...
Pod Template:
  ...
  Containers:
   scdf-server:
    ...
    Ports:       80/TCP, 5005/TCP
    ...
    Environment:
      JAVA_TOOL_OPTIONS:  -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
   ...
```

With the server started and JDWP enabled, you need to configure access to the port. In this example, we use the `port-forward` subcommand of `kubectl`. The following example (with output) shows how to expose a local port to your debug target by using `port-forward`:

```
$ kubectl get pod -l app=scdf-server
NAME                          READY     STATUS    RESTARTS   AGE
scdf-server-5b7cfd86f7-d8mj4   1/1       Running   0          10m
$ kubectl port-forward scdf-server-5b7cfd86f7-d8mj4 5005:5005
```

```
Forwarding from 127.0.0.1:5005 -> 5005
Forwarding from [::1]:5005 -> 5005
```

You can now attach a debugger by pointing it to `127.0.0.1` as the host and `5005` as the port. The `port-forward` subcommand runs until stopped (by pressing `CTRL+c`, for example).

You can remove debugging support by reverting the changes to `src/kubernetes/server/server-deployment.yaml`. The reverted changes are picked up on the next deployment of the Spring Cloud Data Flow Kubernetes Server. Manually adding debug support to the configuration is useful when debugging should be enabled by default each time the server is deployed.

18.7.2. Enabling Debugging with Patching

Rather than manually changing the `server-deployment.yaml`, Kubernetes objects can be "patched" in place. For convenience, patch files that provide the same configuration as the manual approach are included. To enable debugging by patching, use the following command:

```
kubectl patch deployment scdf-server -p "$(cat src/kubernetes/server/server-deployment-debug.yaml)"
```

Running the preceding command automatically adds the `containerPort` attribute and the `JAVA_TOOL_OPTIONS` environment variable. The following example (with output) shows how toverify changes to the `scdf-server` deployment:

```
$ kubectl describe deployment/scdf-server
...
...
Pod Template:
  ...
  Containers:
   scdf-server:
    ...
    Ports:       5005/TCP, 80/TCP
    ...
    Environment:
      JAVA_TOOL_OPTIONS:  -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
   ...
```

To enable access to the debug port, rather than using the `port-forward` subcommand of `kubectl`, you can patch the `scdf-server` Kubernetes service object. You must first ensure that the `scdf-server` Kubernetes service object has the proper configuration. The following example (with output) shows how to do so:

```
kubectl describe service/scdf-server
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30784/TCP
```

If the output contains `<unset>`, you must patch the service to add a name for this port. The following example shows how to do so:

```
$ kubectl patch service scdf-server -p "$(cat src/kubernetes/server/server-svc.yaml)"
```

A port name should only be missing if the target cluster had been created prior to debug functionality being added. Since multiple ports are being added to the `scdf-server` Kubernetes Service Object, each needs to have its own name.

Now you can add the debug port, as the following example shows:

```
kubectl patch service scdf-server -p "$(cat src/kubernetes/server/server-svc-debug.yaml)"
```

The following example (with output) shows how to verify the mapping:

```
$ kubectl describe service scdf-server
Name:                     scdf-server
...
```

```
...
Port:                     scdf-server-jdwp  5005/TCP
TargetPort:               5005/TCP
NodePort:                 scdf-server-jdwp  31339/TCP
...
...
Port:                     scdf-server  80/TCP
TargetPort:               80/TCP
NodePort:                 scdf-server  30883/TCP
...
...
```

The output shows that container port 5005 has been mapped to the NodePort of 31339. The following example (with output) shows how to get the IP address of the Minikube node:

```
$ minikube ip
192.168.99.100
```

With this information, you can create a debug connection by using a host of 192.168.99.100 and a port of 31339.

The following example shows how to disable JDWP:

```
$ kubectl rollout undo deployment/scdf-server
$ kubectl patch service scdf-server --type json -p='[{"op": "remove", "path": "/spec/ports/0"}]'
```

The Kubernetes deployment object is rolled back to its state before being patched. The Kubernetes service object is then patched with a `remove` operation to remove port 5005 from the `containerPorts` list.

`kubectl rollout undo` forces the pod to restart. Patching the Kubernetes Service Object does not re-create the service, and the port mapping to the `scdf-server` deployment remains the same.

See Rolling Back a Deployment for more information on deployment rollbacks, including managing history and Updating API Objects in Place Using kubectl Patch

Shell

This section covers the options for starting the shell and more advanced functionality relating to how the shell handles white spaces, quotes, and interpretation of SpEL expressions. The introductory chapters to the Stream DSL and Composed Task DSL are good places to start for the most common usage of shell commands.

19. Shell Options

The shell is built upon the Spring Shell project. There are command line options generic to Spring Shell and some specific to Data Flow. The shell takes the following command line options

```
unix:>java -jar spring-cloud-dataflow-shell-2.0.0.BUILD-SNAPSHOT.jar --help
Data Flow Options:
  --dataflow.uri=                              Address of the Data Flow Server [default: http://localhost:9393].
  --dataflow.username=                        Username of the Data Flow Server [no default].
  --dataflow.password=                     Password of the Data Flow Server [no default].
  --dataflow.credentials-provider-command= Executes an external command which must return an
                                                 OAuth Bearer Token (Access Token prefixed with 'Bearer '),
                                                 e.g. 'Bearer 12345'), [no default].
  --dataflow.skip-ssl-validation=       Accept any SSL certificate (even self-signed) [default: no].
  --dataflow.proxy.uri=                  Address of an optional proxy server to use [no default].
  --dataflow.proxy.username=        Username of the proxy server (if required by proxy server) [no default].
  --dataflow.proxy.password=        Password of the proxy server (if required by proxy server) [no default].
  --spring.shell.historySize=               Default size of the shell log file [default: 3000].
  --spring.shell.commandFile=               Data Flow Shell executes commands read from the file(s) and then exits.
  --help                                         This message.
```

The `spring.shell.commandFile` option can be used to point to an existing file that contains all the shell commands to deploy one or many related streams and tasks.
Multiple files execution is also supported, they should be passed as comma delimited string :

```
--spring.shell.commandFile=file1.txt,file2.txt
```

This is useful when creating some scripts to help automate deployment.

Also, the following shell command helps to modularize a complex script into multiple independent files:

```
dataflow:>script --file <YOUR_AWESOME_SCRIPT>
```

20. Listing Available Commands

Typing `help` at the command prompt gives a listing of all available commands. Most of the commands are for Data Flow functionality, but a few are general purpose.

```
! - Allows execution of operating system (OS) commands
clear - Clears the console
cls - Clears the console
date - Displays the local date and time
exit - Exits the shell
http get - Make GET request to http endpoint
http post - POST data to http endpoint
quit - Exits the shell
system properties - Shows the shell's properties
version - Displays shell version
```

Adding the name of the command to `help` shows additional information on how to invoke the command.

```
dataflow:>help stream create
Keyword:                   stream create
Description:               Create a new stream definition
 Keyword:                  ** default **
 Keyword:                  name
   Help:                   the name to give to the stream
   Mandatory:              true
   Default if specified:   '__NULL__'
   Default if unspecified: '__NULL__'

 Keyword:                  definition
   Help:                   a stream definition, using the DSL (e.g. "http --port=9000 | hdfs")
   Mandatory:              true
   Default if specified:   '__NULL__'
   Default if unspecified: '__NULL__'

 Keyword:                  deploy
   Help:                   whether to deploy the stream immediately
   Mandatory:              false
   Default if specified:   'true'
   Default if unspecified: 'false'
```

21. Tab Completion

The shell command options can be completed in the shell by pressing the `TAB` key after the leading `--`. For example, pressing `TAB` after `stream create --` results in

```
dataflow:>stream create --
stream create --definition    stream create --name
```

If you type `--de` and then hit tab, `--definition` will be expanded.

Tab completion is also available inside the stream or composed task DSL expression for application or task properties. You can also use `TAB` to get hints in a stream DSL expression for what available sources, processors, or sinks can be used.

22. White Space and Quoting Rules

It is only necessary to quote parameter values if they contain spaces or the `|` character. The following example passes a SpEL expression (which is applied to any data it encounters) to a transform processor:

```
transform --expression='new StringBuilder(payload).reverse()'
```

If the parameter value needs to embed a single quote, use two single quotes, as follows:

```
scan --query='Select * from /Customers where name=''Smith'''
```

22.1. Quotes and Escaping

There is a Spring Shell-based client that talks to the Data Flow Server and is responsible for **parsing** the DSL. In turn, applications may have applications properties that rely on embedded languages, such as the **Spring Expression Language**.

The shell, Data Flow DSL parser, and SpEL have rules about how they handle quotes and how syntax escaping works. When combined together, confusion may arise. This section explains the rules that apply and provides examples of the

most complicated situations you may encounter when all three components are involved.

It's not always that complicated

If you do not use the Data Flow shell (for example, you use the REST API directly) or if application properties are not SpEL expressions, then the escaping rules are simpler.

22.1.1. Shell rules

Arguably, the most complex component when it comes to quotes is the shell. The rules can be laid out quite simply, though:

- A shell command is made of keys (`--something`) and corresponding values. There is a special, keyless mapping, though, which is described later.
- A value cannot normally contain spaces, as space is the default delimiter for commands.
- Spaces can be added though, by surrounding the value with quotes (either single (`'`) or double (`"`) quotes).
- Values passed inside deployment properties (e.g. `deployment <stream-name> --properties " …"`) should not be quoted again.
- If surrounded with quotes, a value can embed a literal quote of the same kind by prefixing it with a backslash (`\`).
- Other escapes are available, such as `\t`, `\n`, `\r`, `\f` and unicode escapes of the form `\uxxxx`.
- The keyless mapping is handled in a special way such that it does not need quoting to contain spaces.

For example, the shell supports the `!` command to execute native shell commands. The `!` accepts a single keyless argument. This is why the following works:

```
dataflow:>! rm something
```

The argument here is the whole `rm something` string, which is passed as is to the underlying shell.

As another example, the following commands are strictly equivalent, and the argument value is `something` (without the quotes):

```
dataflow:>stream destroy something
dataflow:>stream destroy --name something
dataflow:>stream destroy "something"
dataflow:>stream destroy --name "something"
```

22.1.2. Property files rules

Rules are relaxed when loading the properties from files. * The special characters used in property files (both Java and YAML) needs to be escaped. For example `\` should be replaced by `\\`, '\t` by `\\t` and so forth. * For Java property files (`--propertiesFile` <FILE_PATH>.properties) the property values should not be surrounded by quotes! It is not needed even if they contain spaces.

```
filter.expression=payload > 5
```

- For YAML property files (`--propertiesFile` <FILE_PATH>.yaml), though, the values need to be surrounded by double quotes.

```
app:
    filter:
        filter:
            expression: "payload > 5"
```

22.1.3. DSL Parsing Rules

At the parser level (that is, inside the body of a stream or task definition) the rules are as follows:

- Option values are normally parsed until the first space character.
- They can be made of literal strings, though, surrounded by single or double quotes.
- To embed such a quote, use two consecutive quotes of the desired kind.

As such, the values of the `--expression` option to the filter application are semantically equivalent in the following examples:

```
filter --expression=payload>5
filter --expression="payload>5"
filter --expression='payload>5'
filter --expression='payload > 5'
```

Arguably, the last one is more readable. It is made possible thanks to the surrounding quotes. The actual expression is

`payload > 5` (without quotes).

Now, imagine that we want to test against string messages. If we want to compare the payload to the SpEL literal string, `"something"`, we could use the following:

```
filter --expression=payload=='something'           (1)
filter --expression='payload == ''something'''     (2)
filter --expression='payload == "something"'       (3)
```

(1) This works because there are no spaces. It is not very legible, though.

(2) This uses single quotes to protect the whole argument. Hence, the actual single quotes need to be doubled.

(3) SpEL recognizes String literals with either single or double quotes, so this last method is arguably the most readable.

Please note that the preceding examples are to be considered outside of the shell (for example, when calling the REST API directly). When entered inside the shell, chances are that the whole stream definition is itself inside double quotes, which would need to be escaped. The whole example then becomes the following:

```
dataflow:>stream create something --definition "http | filter --expression=payload='something' | log"

dataflow:>stream create something --definition "http | filter --expression='payload == ''something''' | log"

dataflow:>stream create something --definition "http | filter --expression='payload == \"something\"' | log"
```

22.1.4. SpEL Syntax and SpEL Literals

The last piece of the puzzle is about SpEL expressions. Many applications accept options that are to be interpreted as SpEL expressions, and, as seen above, String literals are handled in a special way there, too. The rules are as follows:

- Literals can be enclosed in either single or double quotes.
- Quotes need to be doubled to embed a literal quote. Single quotes inside double quotes need no special treatment, and the reverse is also true.

As a last example, assume you want to use the transform processor. This processor accepts an `expression` option which is a SpEL expression. It is to be evaluated against the incoming message, with a default of `payload` (which forwards the message payload untouched).

It is important to understand that the following statements are equivalent:

```
transform --expression=payload
transform --expression='payload'
```

However, they are different from the following (and variations upon them):

```
transform --expression="'payload'"
transform --expression='''payload'''
```

The first series evaluates to the message payload, while the latter examples evaluate to the literal string `payload`, (again, without quotes).

22.1.5. Putting It All Together

As a last, complete example, consider how one could force the transformation of all messages to the string literal, `hello world`, by creating a stream in the context of the Data Flow shell:

```
dataflow:>stream create something --definition "http | transform --expression='''hello world''' | log" (1)

dataflow:>stream create something --definition "http | transform --expression='\"hello world\"' | log" (2)

dataflow:>stream create something --definition "http | transform --expression=\"'hello world'\" | log" (2)
```

(1) In the first line, there are single quotes around the string (at the Data Flow parser level), but they need to be doubled because they are inside a string literal (started by the first single quote after the equals sign).

(2) The second and third lines, use single and double quotes respectively to encompass the whole string at the Data Flow parser level. Consequently, the other kind of quote can be used inside the string. The whole thing is inside the `--definition` argument to the shell, though, which uses double quotes. Consequently, double quotes are escaped (at the shell level)

Streams

This section goes into more detail about how you can create Streams, which are collections of Spring Cloud Stream

applications. It covers topics such as creating and deploying Streams.

If you are just starting out with Spring Cloud Data Flow, you should probably read theGetting Started guide before diving into this section.

23. Introduction

A Stream is are a collection of long-livedSpring Cloud Stream applications that communicate with each other over messaging middleware. A text-based DSL defines the configuration and data flow between the applications. While many applications are provided for you to implement common use-cases, you typically create a custom Spring Cloud Stream application to implement custom business logic.

The general lifecycle of a Stream is:

1. Register applications.
2. Create a Stream Definition.
3. Deploy the Stream.
4. Undeploy or Destroy the Stream.
5. Upgrade or Rollack applications in the Stream.

For deploying streams, the Data Flow Server has to be configured to delegate the deployment to a new server in the Spring Cloud ecosystem named Skipper.

Furthermore you can configure Skipper to deploy applications to one or more Cloud Foundry orgs and spaces, one or more namespaces on a Kubernetes cluster, or to the local machine. When deploying a stream in Data Flow, you can specify which platform to use at deployment time. Skipper also provides Data Flow with the ability to perform updates to deployed streams. There are many ways the applications in a stream can be updated, but one of the most common examples is to upgrade a processor application with new custom business logic while leaving the existing source and sink applications alone.

23.1. Stream Pipeline DSL

A stream is defined by using a unix-inspiredPipeline syntax. The syntax uses vertical bars, also known as “pipes” to connect multiple commands. The command `ls -l | grep key | less` in Unix takes the output of the `ls -l` process and pipes it to the input of the `grep key` process. The output of `grep` in turn is sent to the input of the `less` process. Each `|` symbol connects the standard output of the command on the left to the standard input of the command on the right. Data flows through the pipeline from left to right.

In Data Flow, the Unix command is replaced by aSpring Cloud Stream application and each pipe symbol represents connecting the input and output of applications over messaging middleware, such as RabbitMQ or Apache Kafka.

Each Spring Cloud Stream application is registered under a simple name. The registration process specifies where the application can be obtained (for example, in a Maven Repository or a Docker registry). You can find out more information on how to register Spring Cloud Stream applications in this section. In Data Flow, we classify the Spring Cloud Stream applications as Sources, Processors, or Sinks.

As a simple example, consider the collection of data from an HTTP Source writing to a File Sink. Using the DSL, the stream description is:

```
http | file
```

A stream that involves some processing would be expressed as:

```
http | filter | transform | file
```

Stream definitions can be created by using the shell’s `stream create` command, as shown in the following example:

```
dataflow:> stream create --name httpIngest --definition "http | file"
```

The Stream DSL is passed in to the `--definition` command option.

The deployment of stream definitions is done through the shell’s `stream deploy` command.

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

The Getting Started section shows you how to start the server and how to start and use the Spring Cloud Data Flow shell.

Note that the shell calls the Data Flow Servers' REST API. For more information on making HTTP requests directly to the server, consult the REST API Guide.

23.2. Stream Application DSL

The Stream Pipeline DSL described in the previous section automatically sets the input and output binding properties of each Spring Cloud Stream application. This can be done because there is only one input and/or output destination in a Spring Cloud Stream application that uses the provided binding interface of a `Source`, `Processor`, or `Sink`. However, a Spring Cloud Stream application can define a custom binding interface such as the one shown below

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

or as is common when creating a Kafka Streams application,

```
interface KStreamKTableBinding {

    @Input
    KStream<?, ?> inputStream();

    @Input
    KTable<?, ?> inputTable();
}
```

In these cases with multiple input and output bindings, Data Flow cannot make any assumptions about the flow of data from one application to another. Therefore the developer needs to set the binding properties to 'wire up' the application. The **Stream Application DSL** uses a `double pipe`, instead of the `pipe symbol`, to indicate that Data Flow should not configure the binding properties of the application. Think of `||` as meaning 'in parallel'. For example:

`dataflow:> stream create --definition "orderGeneratorApp || baristaApp || hotDrinkDeliveryApp || coldDrinkDeliveryApp" --name myCafeStream`

Breaking Change! Versions of SCDF Local, Cloud Foundry 1.7.0 to 1.7.2 and SCDF Kubernetes 1.7.0 to 1.7.1 used the `comma` character as the separator between applications. This caused breaking changes in the traditional Stream DSL. While not ideal, changing the separator character was felt to be the best solution with the least impact on existing users.

There are four applications in this stream. The baristaApp has two output destinations, `hotDrinks` and `coldDrinks` intended to be consumed by the `hotDrinkDeliveryApp` and `coldDrinkDeliveryApp` respectively. When deploying this stream, you need to set the binding properties so that the `baristaApp` sends hot drink messages to the `hotDrinkDeliveryApp` destination and cold drink messages to the `coldDrinkDeliveryApp` destination. For example

```
app.baristaApp.spring.cloud.stream.bindings.hotDrinks.destination=hotDrinksDest
app.baristaApp.spring.cloud.stream.bindings.coldDrinks.destination=coldDrinksDest
app.hotDrinkDeliveryApp.spring.cloud.stream.bindings.input.destination=hotDrinksDest
app.coldDrinkDeliveryApp.spring.cloud.stream.bindings.input.destination=coldDrinksDest
```

If you want to use consumer groups, you will need to set the Spring Cloud Stream application property `spring.cloud.stream.bindings.<channelName>.producer.requiredGroups` and `spring.cloud.stream.bindings.<channelName>.group` on the producer and consumer applications respectively.

Another common use case for the Stream Application DSL is to deploy a http gateway application that sends a synchronous request/reply message to a Kafka or RabbitMQ application. In this case both the http gateway application and the Kafka or RabbitMQ application can be a Spring Integration application that does not make use of the Spring Cloud Stream library.

It is also possible to deploy just a single application using the Stream application DSL.

23.3. Application properties

Each application takes properties to customize its behavior. As an example, the `http` source module exposes a `port` setting that allows the data ingestion port to be changed from the default value.

`dataflow:> stream create --definition "http --port=8090 | log" --name myhttpstream`

This `port` property is actually the same as the standard Spring Boot `server.port` property. Data Flow adds the ability to use the shorthand form `port` instead of `server.port`. One may also specify the longhand version as well, as shown in the following example:

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

This shorthand behavior is discussed more in the section onWhitelisting application properties. If you have registered application property metadata you can use tab completion in the shell after typing `--` to get a list of candidate property names.

The shell provides tab completion for application properties. The shell command `app info --name <appName> --type <appType>` provides additional documentation for all the supported properties.

Supported Stream `<appType>` possibilities are: source, processor, and sink.

24. Stream Lifecycle

The lifecycle of a stream, goes through the following stages:

1. Register a Stream App
2. Creating a Stream
3. Deploying a Stream
4. Destroying a Stream or Undeploying a Stream
5. Upgrade or Rollback applications in the Stream.

Skipper is a server that you discover Spring Boot applications and manage their lifecycle on multiple Cloud Platforms.

Applications in Skipper are bundled as packages that contain the application's resource location, application properties and deployment properties. You can think Skipper packages as analogous to packages found in tools such as `apt-get` or `brew`.

When Data Flow deploys a Stream, it will generate and upload a package to Skipper that represents the applications in the Stream. Subsequent commands to upgrade or rollback the applications within the Stream are passed through to Skipper. In addition, the Stream definition is reverse engineered from the package and the status of the Stream is also delegated to Skipper.

24.1. Register a Stream App

Register a versioned stream application using the `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". The version is resolved from the URI. Here are a few examples:

```
dataflow:>app register --name mysource --type source --uri maven://com.example:mysource:0.0.1
dataflow:>app register --name mysource --type source --uri maven://com.example:mysource:0.0.2
dataflow:>app register --name mysource --type source --uri maven://com.example:mysource:0.0.3

dataflow:>app list --id source:mysource
╔═══╤═══════════════════╤═════════╤════╤════╗
║app│      source       │processor│sink│task║
╠═══╪═══════════════════╪═════════╪════╪════╣
║   │> mysource-0.0.1 <│         │    │    ║
║   │mysource-0.0.2     │         │    │    ║
║   │mysource-0.0.3     │         │    │    ║
╚═══╧═══════════════════╧═════════╧════╧════╝

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
```

The application URI should conform to one the following schema formats:

- maven schema

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

- http schema

```
http://<web-path>/<artifactName>-<version>.jar
```

- file schema

```
file:///<local-path>/<artifactName>-<version>.jar
```

- docker schema

```
docker:<docker-image-path>/<imageName>:<version>
```

The URI `<version>` part is compulsory for the versioned stream applications. Skipper leverages the multi-versioned stream applications to allow upgrade or rollback of those applications at runtime using the deployment properties.

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 (for example, `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 with the `--uri` switch, as follows:

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

Registering an application using `--type app` is the same as registering a `source`, `processor` or `sink`. Applications of the type `app` are only allowed to be used in the Stream Application DSL, which uses a comma instead of the pipe symbol in the DSL, and instructs Data Flow not to configure the Spring Cloud Stream binding properties of the application. The application that is registered using `--type app` does not have to be a Spring Cloud Stream app, it can be any Spring Boot application. See the Stream Application DSL introduction for more information on using this application type.

Multiple versions can be registered for the same applications (e.g. same name and type) but only one can be set as default. The default version is used for deploying Streams.

The first time an application is registered it will be marked as default. The default application version can be altered with the `app default` command:

```
dataflow:>app default --id source:mysource --version 0.0.2
dataflow:>app list --id source:mysource
╔═══╤═══════════════════╤═════════╤════╤════╗
║app│      source       │processor│sink│task║
╠═══╪═══════════════════╪═════════╪════╪════╣
║   │mysource-0.0.1     │         │    │    ║
║   │> mysource-0.0.2 <│         │    │    ║
║   │mysource-0.0.3     │         │    │    ║
╚═══╧═══════════════════╧═════════╧════╧════╝
```

The `app list --id <type:name>` command lists all versions for a given stream application.

The `app unregister` command has an optional `--version` parameter to specify the app version to unregister.

```
dataflow:>app unregister --name mysource --type source --version 0.0.1
dataflow:>app list --id source:mysource
╔═══╤═══════════════════╤═════════╤════╤════╗
║app│      source       │processor│sink│task║
╠═══╪═══════════════════╪═════════╪════╪════╣
║   │> mysource-0.0.2 <│         │    │    ║
║   │mysource-0.0.3     │         │    │    ║
╚═══╧═══════════════════╧═════════╧════╧════╝
```

If a `--version` is not specified, the default version is unregistered.

All applications in a stream should have a default version set for the stream to be deployed. Otherwise they will be treated as unregistered application during the deployment. Use the `app default` to set the defaults.

```
app default --id source:mysource --version 0.0.3
dataflow:>app list --id source:mysource
╔═══╤═══════════════════╤═════════╤════╤════╗
```

```
╠═══╪═══════════════════╪═════════╪════╪════╣
╠   |mysource-0.0.2     |         |    |    ╠
╠   |> mysource-0.0.3 <|         |    |    ╠
╚═══╧═══════════════════╧═════════╧════╧════╝
```

The `stream deploy` necessitates default app versions to be set. The `stream update` and `stream rollback` commands though can use all (default and non-default) registered app versions.

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

This will create stream using the default mysource version (0.0.3). Then we can update the version to 0.0.2 like this:

```
dataflow:>stream update foo --properties version.mysource=0.0.2
```

Only pre-registered applications can be used to `deploy`, `update` or `rollback` a Stream.

An attempt to update the `mysource` to version `0.0.1` (not registered) will fail!

24.1.1. Register Supported Applications and Tasks

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 previously, 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.

The following table lists the bit.ly links to the available Stream Application Starters based on Spring Boot 1.5.x:

Artifact Type	Stable Release	SNAPSHOT Release
RabbitMQ + Maven	bit.ly/Celsius-SR3-stream-applications-rabbit-maven	bit.ly/Celsius-BUILD-SNAPSHOT-stream-applications-rabbit-maven
RabbitMQ + Docker	bit.ly/Celsius-SR3-stream-applications-rabbit-docker	bit.ly/Celsius-BUILD-SNAPSHOT-stream-applications-rabbit-docker
Kafka 0.10 + Maven	bit.ly/Celsius-SR3-stream-applications-kafka-10-maven	bit.ly/Celsius-BUILD-SNAPSHOT-stream-applications-kafka-10-maven
Kafka 0.10 + Docker	bit.ly/Celsius-SR3-stream-applications-kafka-10-docker	bit.ly/Celsius-BUILD-SNAPSHOT-stream-applications-kafka-10-docker

The following table lists the bit.ly links to the available Stream Application Starters based on Spring Boot 2.0.x:

App Starter actuator endpoints are secured by default. You can disable security by deploying streams with the property `app.*.spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConf`
On Kubernetes refer to the section Liveness and readiness probes to configure security for actuator endpoints.

Artifact Type	Stable Release	SNAPSHOT Release
RabbitMQ + Maven	bit.ly/Darwin-SR2-stream-applications-rabbit-maven	bit.ly/Darwin-BUILD-SNAPSHOT-stream-applications-rabbit-maven
RabbitMQ + Docker	bit.ly/Darwin-SR2-stream-applications-rabbit-docker	bit.ly/Darwin-BUILD-SNAPSHOT-stream-applications-rabbit-docker
Kafka 0.11 and above + Maven	bit.ly/Darwin-SR2-stream-applications-kafka-maven	bit.ly/Darwin-BUILD-SNAPSHOT-stream-applications-kafka-maven
Kafka 0.11 and above + Docker	bit.ly/Darwin-SR2-stream-applications-kafka-docker	bit.ly/Darwin-BUILD-SNAPSHOT-stream-applications-kafka-docker

The following table lists the available Task Application Starters:

Artifact Type	Stable Release	SNAPSHOT Release
Maven	bit.ly/Clark-SR1-task-applications-maven	bit.ly/Clark-BUILD-SNAPSHOT-task-applications-maven
Docker	bit.ly/Clark-SR1-task-applications-docker	bit.ly/Clark-BUILD-SNAPSHOT-task-applications-docker

Artifact Type	Stable Release	SNAPSHOT Release

You can find more information about the available task starters in theTask 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 Kafka binder in bulk, you can use the following command:

```
$ dataflow:>app import --uri http://bit.ly/Darwin-SR2-stream-applications-kafka-maven
```

Alternatively you can register all the stream applications with the Rabbit binder, as follows:

```
$ dataflow:>app import --uri http://bit.ly/Darwin-SR2-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`.

When using either `app register` or `app import`, if an app is already registered with the provided name and type, it is not 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 does not change (even though the actual resource *bytes* may be different), then it is not re-downloaded. When using `maven://` resources on the other hand, using a constant location may still 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 has no effect until the stream is deployed again.

In some cases, the Resource is resolved on the server side. In others, the URI is passed to a runtime container instance where it is resolved. Consult the specific documentation of each Data Flow Server for more detail.

24.1.2. Whitelisting application properties

Stream and Task applications are Spring Boot applications that are aware of manyCommon Application Properties, such as `server.port` but also families of properties such as those with the prefix `spring.jmx` and `logging`. When creating your own application, you should whitelist properties so that the shell and the UI can display them first as primary properties when presenting options through 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 the property, such as `server.port`, or a partial name to whitelist a category of property names, such as `spring.jmx`.

The Spring Cloud Stream application starters are a good place to look for examples of usage. The following example comes from the file sink's `spring-configuration-metadata-whitelist.properties` file:

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

If we also want to add `server.port` to be white listed, it would become the following line:

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

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>
```

24.1.3. 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 metadata companion artifact. This jar file contains only the Spring boot JSON file about configuration properties metadata and the whitelisting file described in the previous section.

The following example shows 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`, some more from `spring-cloud-starter-stream-sink-log.jar`, and so on). 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 (you do not 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>
```

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

The benefits of a companion artifact include:

- Being much lighter. (The companion artifact is usually a few kilobytes, as opposed to megabytes for the actual app.) Consequently, they are quicker to download, allowing quicker feedback when using, for example, `app info` or the Dashboard UI.
- As a consequence of being lighter, they can be used in resource constrained environments (such as PaaS) when metadata is the only piece of information needed.
- For environments that do not deal with Spring Boot uber jars directly (for example, Docker-based runtimes such as Kubernetes or Cloud Foundry), 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 uber jars. The uber jar itself also includes the metadata in it already.

24.1.4. 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 with `app register`, you can use the optional `--metadata-uri` option in the shell, as follows:

```
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 by using the `app import` command, the file should contain a `<type>.<name>.metadata` line in addition to each `<type>.<name>` line. Strictly speaking, doing so is optional (if some apps have it but some others do not, it works), but it is best practice.

The following example shows a Dockerized app, where the metadata artifact is being hosted in a Maven repository (retrieving it through `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
...
```

24.1.5. Creating Custom Applications

While there are out-of-the-box source, processor, sink applications available, you can extend these applications or write a custom Spring Cloud Stream application.

The process of creating Spring Cloud Stream applications with Spring Initializr is detailed in the Spring Cloud Stream documentation. It is possible to include multiple binders to an application. If doing so, see the instructions in Passing Spring Cloud Stream properties for 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 shown in the following example:

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

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

Once a custom application has been created, it can be registered as described in Register a Stream App.

24.2. 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 is it is through the Spring Cloud Data Flow shell. Start the shell as described in the Getting Started section.

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

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

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

The `stream info` command shows useful information about the stream, as shown (with its output) in the following example:

```
dataflow:>stream info ticktock
╔═══════════╤═══════════════════╤══════════╗
║Stream Name│Stream Definition  │  Status  ║
╠═══════════╪═══════════════════╪══════════╣
║ticktock   │time | log        │undeployed║
╚═══════════╧═══════════════════╧══════════╝
```

24.2.1. 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 through command line arguments or environment variables, depending on the underlying deployment implementation.

The following stream can have application properties defined at the time of stream creation:

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

The shell command `app info --name <appName> --type <appType>` displays the white-listed application properties for the application. For more info on the property white listing, refer to Whitelisting application properties

The following listing shows the white_listed properties for the `time` app:

```
dataflow:> app info --name time --type source
```

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

The following listing shows the white-listed properties for the `log` app:

```
dataflow:> app info --name log --type sink
```

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 messages.	<none>	org.springframework.integration.handler.LoggingHandler$Level
log.expression	A SpEL expression (against the incoming message) to evaluate as the logged message.	payload	java.lang.String

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, in the preceding example, the `fixed-delay` and `level` properties defined 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. In all other cases, only fully qualified property names should be used.

24.2.2. Common Application Properties

In addition to configuration through 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 passes 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
```

Doing so causes the properties `spring.cloud.stream.kafka.binder.brokers` and `spring.cloud.stream.kafka.binder.zkNodes` to be passed to all the launched applications.

Properties configured with this mechanism have lower precedence than stream deployment properties. They are overridden if a property with the same key is specified at stream deployment time (for example, `app.http.spring.cloud.stream.kafka.binder.brokers` overrides the common property).

24.3. Deploying a Stream

This section describes how to deploy a Stream when the Spring Cloud Data Flow server is responsible for deploying the stream. It covers the deployment and upgrade of Streams leveraging the Skipper service. The description of how deployment properties applies to both approaches of Stream deployment.

Give the `ticktock` stream definition:

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

To deploy the stream, use the following shell command:

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

The Data Flow Server delegates to Skipper the resolution and deployment of the `time` and `log` applications.

The `stream info` command shows useful information about the stream, including the deployment properties:

```
dataflow:>stream info --name ticktock
╔═══════════╤═══════════════════╤═══════════╗
║Stream Name│Stream Definition  │  Status   ║
╠═══════════╪═══════════════════╪═══════════╣
║ticktock   │time | log        │deploying  ║
╚═══════════╧═══════════════════╧═══════════╝

Stream Deployment properties: {
  "log" : {
    "resource" : "maven://org.springframework.cloud.stream.app:log-sink-rabbit",
    "spring.cloud.deployer.group" : "ticktock",
    "version" : "2.0.1.RELEASE"
  },
  "time" : {
    "resource" : "maven://org.springframework.cloud.stream.app:time-source-rabbit",
    "spring.cloud.deployer.group" : "ticktock",
    "version" : "2.0.1.RELEASE"
  }
}
```

There is an important optional command argument (called `--platformName`) to the `stream deploy` command. Skipper can be configured to deploy to multiple platforms. Skipper is pre-configured with a platform named `default`, which deploys applications to the local machine where Skipper is running. The default value of the command line argument `--platformName` is `default`. If you commonly deploy to one platform, when installing Skipper, you can override the configuration of the `default` platform. Otherwise, specify the `platformName` to one of the values returned by the `stream platform-list` command.

In the preceding example, the time source sends the current time as a message each second, and the log sink outputs it by 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 in the following listing:

```
$ tail -f /var/folders/wn/8jxm_tbd1vj28c8vj37n900m0000gn/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
```

You can also create and deploy the stream in one step by passing the `--deploy` flag when creating the stream, as follows:

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

However, it is not very common in real-world use cases to create and deploy the stream in one step. The reason is that when you use the `stream deploy` command, you can pass in properties that define how to map the applications onto the platform (for example, what is the memory size of the container to use, the number of each application to run, and whether to enable data partitioning features). Properties can also override application properties that were set when creating the stream. The next sections cover this feature in detail.

24.3.1. Deployment Properties

When deploying a stream, you can specify properties that fall into two groups:

- Properties that control how the apps are deployed to the target platform. These properties use a `deployer` prefix and are referred to as `deployer` properties.
- Properties that set application properties or override application properties set during stream creation and are referred to as `application` properties.

The syntax for `deployer` properties is `deployer.<app-name>.<short-property-name>=<value>`, and the syntax for `application` properties `app.<app-name>.<property-name>=<value>`. This syntax is used when passing deployment properties through the shell. You may also specify them in a YAML file, which is discussed later in this chapter.

The following table shows the difference in behavior between setting `deployer` and `application` properties when deploying an application.

	Application Properties	**Deployer Properties**
Example Syntax	`app.filter.expression=something`	`deployer.filter.count=3`
What the application "sees"	`expression=something` or, if `expression` is one of the whitelisted properties, `<some-prefix>.expression=something`	Nothing
What the deployer "sees"	Nothing	`spring.cloud.deployer.count=3`. The `spring.cloud.deployer` prefix is automatically and always prepended to

	Application Properties	Deployer Properties
Typical usage	Passing/Overriding application properties, passing Spring Cloud Stream binder or partitioning properties	Setting the number of instances, memory, disk, and others

Passing Instance Count

If you would like to have multiple instances of an application in the stream, you can include a deployer property called `count` 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. Consequently, 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 (for example, `app.something.somethingelse.count`) during stream deployment or it can be specified by using the 'short-form' or the fully qualified form during the stream creation, where it is processed as an app property.

See Using Labels in a Stream.

Inline Versus File-based Properties

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

Inline properties use the `--properties` shell option and list properties as a comma separated list of key=value pairs, as shown in the following example:

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

File references use the `--propertiesFile` option and point it to a local `.properties`, `.yaml` or `.yml` file (that is, a file that resides 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, and others), although we recommend using `=` as a key-value pair delimiter, for consistency. The following example shows a `stream deploy` command that uses the `--propertiesFile` option:

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

Assume that `myprops.properties` contains the following properties:

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

Both of the properties are passed as deployment properties for the `something` stream.

If you use YAML as the format for the deployment properties, use the `.yaml` or `.yml` file extention when deploying the stream, as shown in the following example:

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

In that case, the `myprops.yaml` file might contain the following content:

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

Passing application properties

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 as fully qualified property names. The application properties should have the prefix `app.<appName/label>`.

For example, consider the following stream command:

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

The stream in the precedig example can also be deployed with application properties by using the 'short-form' property names, as shown in the following example:

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

Consider the following example:

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

When using the app label, the application properties can be defined as follows:

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

Passing Spring Cloud Stream properties

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 you want to override any of the Spring Cloud Stream properties, they can be set with deployment properties.

For example, consider the following stream definition:

```
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 follows:

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

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

Passing Per-binding Producer and Consumer Properties

A Spring Cloud Stream application can have producer and consumer properties set on a `per-binding` basis. While Spring Cloud Data Flow supports specifying short-hand notation for per-binding producer properties such as `partitionKeyExpression` and `partitionKeyExtractorClass` (as described in Passing Stream Partition Properties), 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.`. The producer properties can be set for the `outbound` channel name with the prefix `app.[app/label name].spring.cloud.stream.bindings.<channelName>.producer.`. Consider the following example:

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

The stream can be deployed with producer and consumer properties, as follows:

```
dataflow:>stream deploy ticktock --properties
"app.time.spring.cloud.stream.bindings.output.producer.requiredGroups=myGroup,app.time.spring.cloud.stream.bindings.outpu
t.producer.headerMode=raw,app.log.spring.cloud.stream.bindings.input.consumer.concurrency=3,app.log.spring.cloud.stream.b
indings.input.consumer.maxAttempts=5"
```

The `binder`-specific producer and consumer properties can also be specified in a similar way, as shown in the following example:

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

Passing Stream Partition Properties

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.

The following list shows variations 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 is routed. The final partition index is the return value (an integer) modulo `[nextModule].count`. If both the class and expression are null, the underlying binder's default `PartitionSelectorStrategy` is 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` (`partitionKeyExtractorClass` takes precedence). When a partition key is extracted, the partitioned app instance is determined by invoking the `partitionSelectorClass`, if present, or the `partitionSelectorExpression % partitionCount`. `partitionCount` is application count, in the case of RabbitMQ, or 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 must 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.

Consider the following example of sending some data to the `http` application:

```
dataflow:>http post --data {"hello":"world","something":"somethingelse"} --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 an `--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, you can specify the new property values during deployment, as follows:

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

24.4. Destroying a Stream

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

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

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

24.5. Undeploying a Stream

Often you want to stop a stream but retain the name and definition for future use. In that case, you can `undeploy` the stream by name.

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

You can issue the `deploy` command at a later time to restart it.

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

24.6. Validating a Stream

Sometimes the one or more of the apps contained within a stream definition contain an invalid URI in its registration. This can caused by an invalid URI entered at app registration time or the app was removed from the repository from which it was to be drawn. To verify that all the apps contained in a stream are resolve-able, a user can use the `validate` command. For example:

```
dataflow:>stream validate ticktock
╔═══════════╤═══════════════════╗
║Stream Name│Stream Definition  ║
╠═══════════╪═══════════════════╣
║ticktock   │time | log         ║
╚═══════════╧═══════════════════╝


ticktock is a valid stream.
╔═══════════╤═══════════════════╗
║ App Name  │Validation Status  ║
╠═══════════╪═══════════════════╣
║source:time│valid              ║
║sink:log   │valid              ║
╚═══════════╧═══════════════════╝
```

In the example above the user validated their ticktock stream. As we see that both the `source:time` and `sink:log` are valid. Now let's see what happens if we have a stream definition with a registered app with an invalid URI.

```
dataflow:>stream validate bad-ticktock
╔═══════════╤═══════════════════╗
║Stream Name│Stream Definition  ║
╠═══════════╪═══════════════════╣
║bad-ticktock│bad-time | log    ║
╚═══════════╧═══════════════════╝


bad-ticktock is an invalid stream.
╔═══════════╤═══════════════════╗
║   App Name    │Validation Status  ║
╠═══════════╪═══════════════════╣
║source:bad-time│invalid            ║
║sink:log       │valid              ║
╚═══════════╧═══════════════════╝
```

In this case Spring Cloud Data Flow states that the stream is invalid because source:bad-time has an invalid URI.

24.7. Updating a Stream

To update the stream, use the command `stream update` which takes as a command argument either `--properties` or `--propertiesFile`. There is an important new top level prefix available when using Skipper, which is `version`. If the Stream `http | log` was deployed, and the version of `log` which registered at the time of deployment was `1.1.0.RELEASE`:

```
dataflow:> stream create --name httptest --definition "http --server.port=9000 | log"
dataflow:> stream deploy --name httptest
dataflow:>stream info httptest
╔═══════════════════════╤═══════════════════════════════╤═══════════════════════╗
║          Name         │              DSL              │         Status        ║
╠═══════════════════════╪═══════════════════════════════╪═══════════════════════╣
║httptest               │http --server.port=9000 | log  │deploying              ║
╚═══════════════════════╧═══════════════════════════════╧═══════════════════════╝
```

```
Stream Deployment properties: {
  "log" : {
    "spring.cloud.deployer.indexed" : "true",
    "spring.cloud.deployer.group" : "httptest",
    "maven://org.springframework.cloud.stream.app:log-sink-rabbit" : "1.1.0.RELEASE"
  },
  "http" : {
    "spring.cloud.deployer.group" : "httptest",
    "maven://org.springframework.cloud.stream.app:http-source-rabbit" : "1.1.0.RELEASE"
  }
}
```

Then the following command will update the Stream to use the `1.2.0.RELEASE` of the log application. Before updating the stream with the specific version of the app, we need to make sure that the app is registered with that version.

```
dataflow:>app register --name log --type sink --uri maven://org.springframework.cloud.stream.app:log-sink-
rabbit:1.2.0.RELEASE
Successfully registered application 'sink:log'
```

```
dataflow:>stream update --name httptest --properties version.log=1.2.0.RELEASE
```

Only pre-registered application versions can be used to `deploy`, `update`, or `rollback` a stream.

To verify the deployment properties and the updated version, we can use `stream info`, as shown (with its output) in the following example:

```
dataflow:>stream info httptest
╔═══════════════════════╤═══════════════════════╤═══════════════════════╗
║         Name          │          DSL          │        Status         ║
╠═══════════════════════╪═══════════════════════╪═══════════════════════╣
║httptest               │http --server.port=9000 | log │deploying              ║
╚═══════════════════════╧═══════════════════════╧═══════════════════════╝

Stream Deployment properties: {
  "log" : {
    "spring.cloud.deployer.indexed" : "true",
    "spring.cloud.deployer.count" : "1",
    "spring.cloud.deployer.group" : "httptest",
    "maven://org.springframework.cloud.stream.app:log-sink-rabbit" : "1.2.0.RELEASE"
  },
  "http" : {
    "spring.cloud.deployer.group" : "httptest",
    "maven://org.springframework.cloud.stream.app:http-source-rabbit" : "1.1.0.RELEASE"
  }
}
```

24.8. Force update of a Stream

When upgrading a stream, the `--force` option can be used to deploy new instances of currently deployed applications even if no application or deployment properties have changed. This behavior is needed in the case when configuration information is obtained by the application itself at startup time, for example from Spring Cloud Config Server. You can specify which applications to force upgrade by using the option `--app-names`. If you do not specify any application names, all the applications will be force upgraded. You can specify `--force` and `--app-names` options together with `--properties` or `--propertiesFile` options.

24.9. Stream versions

Skipper keeps a history of the streams that were deployed. After updating a Stream, there will be a second version of the stream. You can query for the history of the versions using the command `stream history --name <name-of-stream>`.

```
dataflow:>stream history --name httptest
╔═══════╤═══════════════════════════╤═══════╤════════════╤═══════════════╤═══════════════╗
║Version│        Last updated       │ Status │Package Name│Package Version│  Description  ║
╠═══════╪═══════════════════════════╪═══════╪════════════╪═══════════════╪═══════════════╣
║2      │Mon Nov 27 22:41:16 EST 2017│DEPLOYED│httptest    │1.0.0          │Upgrade complete║
║1      │Mon Nov 27 22:40:41 EST 2017│DELETED │httptest    │1.0.0          │Delete complete ║
╚═══════╧═══════════════════════════╧═══════╧════════════╧═══════════════╧═══════════════╝
```

24.10. Stream Manifests

Skipper keeps a “manifest” of the all the applications, their application properties, and their deployment properties after all values have been substituted. This represents the final state of what was deployed to the platform. You can view the manifest for any of the versions of a Stream by using the following command:

`stream manifest --name <name-of-stream> --releaseVersion <optional-version>`

If the `--releaseVersion` is not specified, the manifest for the last version is returned.

The following example shows the use of the manifest:

```
dataflow:>stream manifest --name httptest
```

Using the command results in the following output:

```
# Source: log.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  name: log
spec:
  resource: maven://org.springframework.cloud.stream.app:log-sink-rabbit
  version: 1.2.0.RELEASE
  applicationProperties:
    spring.metrics.export.triggers.application.includes: integration**
    spring.cloud.dataflow.stream.app.label: log
    spring.cloud.stream.metrics.key: httptest.log.${spring.cloud.application.guid}
    spring.cloud.stream.bindings.input.group: httptest
    spring.cloud.stream.metrics.properties:
spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*
    spring.cloud.dataflow.stream.name: httptest
    spring.cloud.dataflow.stream.app.type: sink
    spring.cloud.stream.bindings.input.destination: httptest.http
  deploymentProperties:
    spring.cloud.deployer.indexed: true
    spring.cloud.deployer.group: httptest
    spring.cloud.deployer.count: 1

---
# Source: http.yml
apiVersion: skipper.spring.io/v1
kind: SpringCloudDeployerApplication
metadata:
  name: http
spec:
  resource: maven://org.springframework.cloud.stream.app:http-source-rabbit
  version: 1.2.0.RELEASE
  applicationProperties:
    spring.metrics.export.triggers.application.includes: integration**
    spring.cloud.dataflow.stream.app.label: http
    spring.cloud.stream.metrics.key: httptest.http.${spring.cloud.application.guid}
    spring.cloud.stream.bindings.output.producer.requiredGroups: httptest
    spring.cloud.stream.metrics.properties:
spring.application.name,spring.application.index,spring.cloud.application.*,spring.cloud.dataflow.*
    server.port: 9000
    spring.cloud.stream.bindings.output.destination: httptest.http
    spring.cloud.dataflow.stream.name: httptest
    spring.cloud.dataflow.stream.app.type: source
  deploymentProperties:
    spring.cloud.deployer.group: httptest
```

The majority of the deployment and application properties were set by Data Flow to enable the applications to talk to each other and to send application metrics with identifying labels.

24.11. Rollback a Stream

You can rollback to a previous version of the stream using the command `stream rollback`.

```
dataflow:>stream rollback --name httptest
```

The optional `--releaseVersion` command argument adds the version of the stream. If not specified, the rollback goes to the previous stream version.

24.12. Application Count

The application count is a dynamic property of the system. If, due to scaling at runtime, the application to be upgraded has 5 instances running, then 5 instances of the upgraded application are deployed.

24.13. Skipper's Upgrade Strategy

Skipper has a simple 'red/black' upgrade strategy. It deploys the new version of the applications, using as many instances as the currently running version, and checks the `/health` endpoint of the application. If the health of the new application is good, then the previous application is undeployed. If the health of the new application is bad, then all new applications are undeployed and the upgrade is considered to be not successful.

The upgrade strategy is not a rolling upgrade, so if five applications of the application are running, then in a sunny-day scenario, five of the new applications are also running before the older version is undeployed.

---

25. Stream DSL

This section covers additional features of the Stream DSL not covered in the[Stream DSL introduction](#).

25.1. Tap a Stream

Taps can be created at various producer endpoints in a stream. For a stream such as that defined in the following example, taps can be created at the output of `http`, `step1` and `step2`:

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

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

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

To create a tap at the output of `http` in the preceding stream, 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 resembles the following:

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

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

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

25.2. Using Labels in a Stream

When a stream is made up 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`

25.3. Named Destinations

Instead of referencing a source or sink application, you can use a named destination. A named destination corresponds to a specific destination name in the middleware broker (Rabbit, Kafka, and others). When using the `|` symbol, applications are connected to each other with messaging middleware destination names created by the Data Flow server. In keeping with the Unix analogy, one can redirect standard input and output using the less-than (`<`) and greater-than (`>`) characters. To specify the name of the destination, prefix it with a colon (`:`). For example, the following stream has the destination name in the `source` position:

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

This stream receives messages from the destination called `myDestination`, located at the broker, and connects it to the `log` app. You can also create additional streams that consume data from the same named destination.

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

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

It is also possible to connect two different destinations (`source` and `sink` positions) at the broker in a stream, as shown in the following example:

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

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

25.4. Fan-in and Fan-out

By using named destinations, you can support fan-in and fan-out use cases. Fan-in use cases are when multiple sources all send data to the same named destination, as shown in the following example:

```
s3 > :data
ftp > :data
http > :data
```

The preceding example directs the data payloads from the Amazon S3, FTP, and HTTP sources to the same named destination called `data`. Then an additional stream created with the following DSL expression would have all the data from those three sources sent to the file sink:

```
:data > file
```

The fan-out use case is when you determine the destination of a stream based on some information that is only known at runtime. In this case, the Router Application can be used to specify how to direct the incoming message to one of N

named destinations.

A nice Video showing Fan-in and Fan-out behavior is also available.

---

26. Stream Java DSL

Instead of using the shell to create and deploy streams, you can use the Java-based DSL provided by the `spring-cloud-dataflow-rest-client` module. The Java DSL is a convenient wrapper around the `DataFlowTemplate` class that enables creating and deploying streams programmatically.

To get started, you need to add the following dependency to your project, as follows:

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-dataflow-rest-client</artifactId>
 <version>2.0.0.BUILD-SNAPSHOT</version>
</dependency>
```

A complete sample can be found in the Spring Cloud Data Flow Samples Repository.

26.1. Overview

The classes at the heart of the Java DSL are `StreamBuilder`, `StreamDefinition`, `Stream`, `StreamApplication`, and `DataFlowTemplate`. The entry point is a `builder` method on `Stream` that takes an instance of a `DataFlowTemplate`. To create an instance of a `DataFlowTemplate`, you need to provide a `URI` location of the Data Flow Server.

Spring Boot auto-configuration for `StreamBuilder` and `DataFlowTemplate` is also available. The properties in DataFlowClientProperties can be used to configure the connection to the Data Flow server. The common property to start using is `spring.cloud.dataflow.client.uri`

Consider the following example, using the `definition` style.

```
URI dataFlowUri = URI.create("http://localhost:9393");
DataFlowOperations dataFlowOperations = new DataFlowTemplate(dataFlowUri);
dataFlowOperations.appRegistryOperations().importFromResource(
                    "http://bit.ly/Darwin-SR2-stream-applications-rabbit-maven", true);
StreamDefinition streamDefinition = Stream.builder(dataFlowOperations)
                                    .name("ticktock")
                                    .definition("time | log")
                                    .create();
```

The `create` method returns an instance of a `StreamDefinition` representing a Stream that has been created but not deployed. This is called the `definition` style since it takes a single string for the stream definition, same as in the shell. If applications have not yet been registered in the Data Flow server, you can use the `DataFlowOperations` class to register them. With the `StreamDefinition` instance, you have methods available to `deploy` or `destory` the stream.

```
Stream stream = streamDefinition.deploy();
```

The `Stream` instance provides `getStatus`, `destroy` and `undeploy` methods to control and query the stream. If you are going to immediately deploy the stream, there is no need to create a separate local variable of the type `StreamDefinition`. You can just chain the calls together, as follows:

```
Stream stream = Stream.builder(dataFlowOperations)
                .name("ticktock")
                .definition("time | log")
                .create()
                .deploy();
```

The `deploy` method is overloaded to take a `java.util.Map` of deployment properties.

The `StreamApplication` class is used in the 'fluent' Java DSL style and is discussed in the next section. The `StreamBuilder` class is returned from the method `Stream.builder(dataFlowOperations)`. In larger applications, it is common to create a single instance of the `StreamBuilder` as a Spring `@Bean` and share it across the application.

26.2. Java DSL styles

The Java DSL offers two styles to create Streams.

- The `definition` style keeps the feel of using the pipes and filters textual DSL in the shell. This style is selected by using the `definition` method after setting the stream name - for example, `Stream.builder(dataFlowOperations).name("ticktock").definition(<definition goes here>)`.

- The `fluent` style lets you chain together sources, processors, and sinks by passing in an instance of a `StreamApplication`. This style is selected by using the `source` method after setting the stream name - for example, `Stream.builder(dataFlowOperations).name("ticktock").source(<stream application instance goes here>)`. You then chain together `processor()` and `sink()` methods to create a stream definition.

To demonstrate both styles, we include a simple stream that uses both approaches. A complete sample for you to get started can be found in the Spring Cloud Data Flow Samples Repository.

The following example demonstrates the definition approach:

```
public void definitionStyle() throws Exception{

  Map<String, String> deploymentProperties = createDeploymentProperties();

  Stream woodchuck = Stream.builder(dataFlowOperations)
          .name("woodchuck")
          .definition("http --server.port=9900 | splitter --expression=payload.split(' ') | log")
          .create()
          .deploy(deploymentProperties);

  waitAndDestroy(woodchuck)
}
```

The following example demonstrates the fluent approach:

```
private void fluentStyle(DataFlowOperations dataFlowOperations) throws InterruptedException {

  logger.info("Deploying stream.");

  Stream woodchuck = builder
    .name("woodchuck")
    .source(source)
    .processor(processor)
    .sink(sink)
    .create()
    .deploy();

  waitAndDestroy(woodchuck);
}
```

The `waitAndDestroy` method uses the `getStatus` method to poll for the stream's status, as shown in the following example:

```
private void waitAndDestroy(Stream stream) throws InterruptedException {

  while(!stream.getStatus().equals("deployed")){
    System.out.println("Wating for deployment of stream.");
    Thread.sleep(5000);
  }

  System.out.println("Letting the stream run for 2 minutes.");
  // Let the stream run for 2 minutes
  Thread.sleep(120000);

  System.out.println("Destroying stream");
  stream.destroy();
}
```

When using the definition style, the deployment properties are specified as a `java.util.Map` in the same manner as using the shell. The `createDeploymentProperties` method is defined as follows:

```
private Map<String, String> createDeploymentProperties() {
  DeploymentPropertiesBuilder propertiesBuilder = new DeploymentPropertiesBuilder();
  propertiesBuilder.memory("log", 512);
  propertiesBuilder.count("log",2);
  propertiesBuilder.put("app.splitter.producer.partitionKeyExpression", "payload");
  return propertiesBuilder.build();
}
```

Is this case, application properties are also overridden at deployment time in addition to setting the deployer property `count` for the log application. When using the fluent style, the deployment properties are added by using the method `addDeploymentProperty` (for example, `new StreamApplication("log").addDeploymentProperty("count", 2)`), and you do not need to prefix the property with `deployer.<app_name>`.

In order to create and deploy your streams, you need to make sure that the corresponding apps have been registered in the DataFlow server first. Attempting to create or deploy a stream that contains an unknown app throws an exception. You can register your application by using the `DataFlowTemplate`, as follows:

```
dataFlowOperations.appRegistryOperations().importFromResource(
            "http://bit.ly/Darwin-GA-stream-applications-rabbit-maven", true);
```

The Stream applications can also be beans within your application that are injected in other classes to create Streams. There are many ways to structure Spring applications, but one way is to have an `@Configuration` class define the `StreamBuilder` and `StreamApplications`, as shown in the following example:

```
@Configuration
public class StreamConfiguration {

  @Bean
  public StreamBuilder builder() {
    return Stream.builder(new DataFlowTemplate(URI.create("http://localhost:9393")));
  }

  @Bean
  public StreamApplication httpSource(){
    return new StreamApplication("http");
  }

  @Bean
  public StreamApplication logSink(){
    return new StreamApplication("log");
  }
}
```

Then in another class you can `@Autowire` these classes and deploy a stream.

```
@Component
public class MyStreamApps {

  @Autowired
  private StreamBuilder streamBuilder;

  @Autowired
  private StreamApplication httpSource;

  @Autowired
  private StreamApplication logSink;


  public void deploySimpleStream() {
    Stream simpleStream = streamBuilder.name("simpleStream")
                            .source(httpSource)
                            .sink(logSink)
                            .create()
                            .deploy();
  }
}
```

This style lets you share `StreamApplications` across multiple Streams.

26.3. Using the DeploymentPropertiesBuilder

Regardless of style you choose, the `deploy(Map<String, String> deploymentProperties)` method allows customization of how your streams will be deployed. We made it a easier to create a map with properties by using a builder style, as well as creating static methods for some properties so you don't need to remember the name of such properties. If you take the previous example of `createDeploymentProperties` it could be rewritten as:

```
private Map<String, String> createDeploymentProperties() {
 return new DeploymentPropertiesBuilder()
  .count("log", 2)
  .memory("log", 512)
  .put("app.splitter.producer.partitionKeyExpression", "payload")
  .build();
}
```

This utility class is meant to help with the creation of a Map and adds a few methods to assist with defining pre-defined properties.

26.4. Skipper Deployment Properties

In addition to Spring Cloud Data Flow, you need to pass certain Skipper specific deployment properties, for example selecting the target platform. The `SkipperDeploymentPropertiesBuilder` provides you all the properties in `DeploymentPropertiesBuilder` and adds those needed for Skipper.

```
private Map<String, String> createDeploymentProperties() {
 return new SkipperDeploymentPropertiesBuilder()
  .count("log", 2)
  .memory("log", 512)
  .put("app.splitter.producer.partitionKeyExpression", "payload")
  .platformName("pcf")
  .build();
}
```

27. 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, a multi-binder transformer that supports both Kafka and Rabbit binders is the processor in the following stream:

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

In the example above you would write your own multibindertransform application.

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

1. The HTTP source sends events to RabbitMQ (`rabbit1`).
2. The Multi-Binder Transform processor receives events from RabbitMQ (`rabbit1`) and sends the processed events into Kafka (`kafka1`).
3. The log sink receives events from Kafka (`kafka1`).

Here, `rabbit1` and `kafka1` are the binder names given in the spring cloud stream application properties. Based on this setup, the applications 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 through `deployment` properties when the stream is deployed as shown in the following example:

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

dataflow:>stream deploy mystream --properties
"app.http.spring.cloud.stream.bindings.output.binder=rabbit1,app.multibindertransform.spring.cloud.stream.bindings.input.
binder=rabbit1,
app.multibindertransform.spring.cloud.stream.bindings.output.binder=kafka1,app.log.spring.cloud.stream.bindings.input.bin
der=kafka1"
```

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

28. Examples

This chapter includes the following examples:

- Simple Stream Processing
- Stateful Stream Processing
- Other Source and Sink Application Types

You can find links to more samples in the "Samples" chapter.

28.1. Simple Stream Processing

As an example of a simple processing step, we can transform the payload of the HTTP posted data to upper case by using the following stream definition: `http | transform --expression=payload.toUpperCase() | log`

To create this stream enter the following command in the shell

```
dataflow:> stream create --definition "http --server.port=9000 | transform --expression=payload.toUpperCase() | log" --
name mystream --deploy
```

The following example uses a shell command to post some data:

```
dataflow:> http post --target http://localhost:9000 --data "hello"
```

The preceding example results in an upper-case 'HELLO' in the log, as follows:

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

28.2. Stateful Stream Processing

To demonstrate the data partitioning functionality, the following listing deploys a 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


dataflow:>runtime apps
╔════════════════════╤═══════════╤═════════════════════════════════════════════════════════════════════════════════════
══════════════════════════════════════════════════════════════════════╗
║App Id / Instance Id│Unit Status│                                                              No. of Instances /
Attributes                                                                        ║
╠════════════════════╪═══════════╪═════════════════════════════════════════════════════════════════════════════════════
══════════════════════════════════════════════════════════════════════╣
║words.log-v1        │ deployed  │                                                                            2
║
╟┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┼┈┈┈┈┈┈┈┈┈┈┈┼┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈
┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈╢
║                    │           │       guid = 24166
║
║                    │           │        pid = 33097
║
║                    │           │       port = 24166
║
║words.log-v1-0      │ deployed  │     stderr = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461063/words.log-v1/stderr_0.log     ║
║                    │           │     stdout = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461063/words.log-v1/stdout_0.log     ║
║                    │           │        url = http://192.168.0.102:24166
║
║                    │           │working.dir = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461063/words.log-v1                  ║
╟┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┼┈┈┈┈┈┈┈┈┈┈┈┼┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈
┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈╢
║                    │           │       guid = 41269
║
║                    │           │        pid = 33098
║
║                    │           │       port = 41269
║
║words.log-v1-1      │ deployed  │     stderr = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461063/words.log-v1/stderr_1.log     ║
║                    │           │     stdout = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461063/words.log-v1/stdout_1.log     ║
║                    │           │        url = http://192.168.0.102:41269
║
║                    │           │working.dir = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461063/words.log-v1                  ║
╟────────────────────┼───────────┼─────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────╢
║words.http-v1       │ deployed  │                                                                            1
║
╟┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┼┈┈┈┈┈┈┈┈┈┈┈┼┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈
┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈╢
║                    │           │       guid = 9900
║
║                    │           │        pid = 33094
║
║                    │           │       port = 9900
║
║words.http-v1-0     │ deployed  │     stderr = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461054/words.http-v1/stderr_0.log    ║
║                    │           │     stdout = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461054/words.http-v1/stdout_0.log    ║
║                    │           │        url = http://192.168.0.102:9900
║
║                    │           │working.dir = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803461054/words.http-v1                 ║
╟────────────────────┼───────────┼─────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────╢
║words.splitter-v1   │ deployed  │                                                                            1
║
╟┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┼┈┈┈┈┈┈┈┈┈┈┈┼┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈
┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈╢
║                    │           │       guid = 33963
║
║                    │           │        pid = 33093
║
║                    │           │       port = 33963
║
║words.splitter-v1-0 │ deployed  │     stderr = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803437542/words.splitter-v1/stderr_0.log║
║                    │           │     stdout = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
6467595568759190742/words-1542803437542/words.splitter-v1/stdout_0.log║
║                    │           │        url = http://192.168.0.102:33963
║
║                    │           │working.dir = /var/folders/js/7b_pn0t575l790x7j61slyxc0000gn/T/spring-cloud-deployer-
```

```
6467595568759190742/words-1542803437542/words.splitter-v1
```

When you review the `words.log-v1-0` logs, you should see the following:

```
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                                 : chuck
```

When you review the `words.log-v1-1` logs, you should see the following:

```
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 example has shown that payload splits that contain the same word are routed to the same application instance.

28.3. Other Source and Sink Application Types

This example shows something a bit more complicated: swapping 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 in the Simple Stream Processing example to the following:

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

Note that we do not see any other output this time until we actually post some data (by using a shell command). In order to see the randomly assigned port on which the http source is listening, run the following command:

```
dataflow:>runtime apps
```

App Id / Instance Id	Unit Status	No. of Instances /
myhttpstream.log-v1	deploying	1
		guid = 39628
		pid = 34403
		port = 39628
myhttpstream.log-v1-0	deploying	stderr = /var/folders/js/7b_pn0t5751790x7j61slyxc0000gn/T/spring-cloud-deployer-64
		stdout = /var/folders/js/7b_pn0t5751790x7j61slyxc0000gn/T/spring-cloud-deployer-64
		url = http://192.168.0.102:39628
		working.dir = /var/folders/js/7b_pn0t5751790x7j61slyxc0000gn/T/spring-cloud-deployer-64
myhttpstream.http-v1	deploying	1
		guid = 52143
		pid = 34401
		port = 52143
myhttpstream.http-v1-0	deploying	stderr = /var/folders/js/7b_pn0t5751790x7j61slyxc0000gn/T/spring-cloud-deployer-64
		stdout = /var/folders/js/7b_pn0t5751790x7j61slyxc0000gn/T/spring-cloud-deployer-64
		url = http://192.168.0.102:52143
		working.dir = /var/folders/js/7b_pn0t5751790x7j61slyxc0000gn/T/spring-cloud-deployer-64

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, as shown in the following example:

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

The stream then funnels the data from the http source to the output log implemented by the log sink, yielding output similar to the following:

```
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
```

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 applications that are available. You can also define your own applications.

Stream Developer Guide

This section covers how to create, test, and run Spring Cloud Stream applications on your local machine. It also shows how to map these applications into Spring Cloud Data Flow and deploy them.

29. Prebuilt Applications

The Spring Cloud Stream App Starters project provides many applications that you can start using right away. For example, there is an HTTP source application that receives messages posted to an HTTP endpoint and publishes the data to the messaging middleware. Each existing application comes in three variations, one for each type of messaging middleware that is supported. The current supported messaging middleware systems are RabbitMQ, Apache Kafka 0.9, and Apache Kafka 0.10. All the applications are based on Spring Boot and Spring Cloud Stream.

Applications are published as a Maven artifact as well as a Docker image. For GA releases, the Maven artifacts are published to Maven central and the Spring Release Repository. Milestone and snapshot releases are published to the Spring Milestone and Spring Snapshot repositories, respectively. Docker images are pushed to Docker Hub.

We use the Maven artifacts for our examples. The root location of the Spring Repository that hosts the GA artifacts of prebuilt applications is repo.spring.io/release/org/springframework/cloud/stream/app/

---

30. Running prebuilt applications

In this example, we use RabbitMQ as the messaging middleware. Follow the directions on rabbitmq.com for your platform. Then install the management plugin.

We will first run the HTTP source application and the log sink as stand alone applications using `java -jar`. The two applications use RabbitMQ to communicate.

Download each sample application, as follows:

```
wget https://repo.spring.io/libs-release/org/springframework/cloud/stream/app/http-source-rabbit/1.3.1.RELEASE//http-source-rabbit-1.3.1.RELEASE.jar

wget https://repo.spring.io/release/org/springframework/cloud/stream/app/log-sink-rabbit/1.3.1.RELEASE/log-sink-rabbit-1.3.1.RELEASE.jar
```

These are Spring Boot applications that include the Spring Boot Actuator and the Spring Security Starter. You can specify common Spring Boot properties to configure each application. The properties that are specific to each application are listed in the documentation for the Spring App Starters - for example, the HTTP source and the log sink

Now you can run the http source application. Just for fun, pass in a few Boot applications options using system properties, as shown in the following example:

```
java -Dserver.port=8123 -Dhttp.path-pattern=/data -Dspring.cloud.stream.bindings.output.destination=sensorData -jar http-source-rabbit-1.3.1.RELEASE.jar
```

The `server.port` property comes from Spring Boot's Web support, and the `http.path-pattern` property comes from the HTTP source application, HttpSourceProperties. The HTTP source app is now listening on port 8123 under the the path `/data`.

The `spring.cloud.stream.bindings.output.destination` property comes from the Spring Cloud Stream library and is the name of the messaging destination that is shared between the source and the sink. The string `sensorData` in this property is the name of the Spring Integration channel whose contents will be published to the messaging middleware.

Now you can run the log sink application and change the logging level to WARN, as follows:

```
java -Dlog.level=WARN -Dspring.cloud.stream.bindings.input.destination=sensorData -jar log-sink-rabbit-1.3.1.RELEASE.jar
```

The `log.level` property comes from the log sink application, LogSinkProperties.

The value of the property `spring.cloud.stream.bindings.input.destination` is set to `sensorData` so that the source and sink applications can communicate with each other.

You can use the following `curl` command to send data to the `http` application:

```
curl -H "Content-Type: application/json" -X POST -d '{"id":"1","temperature":"100"}' http://localhost:8123/data
```

The log sink application then shows the following output:

```
2017-03-17 15:30:17.825  WARN 22710 --- [_qquaYekbQOnA-1] log-sink                                 :
{"id":"1","temperature":"100"}
```

31. Custom Processor Application

Now you can create and test an application that does some processing on the output of the HTTP source and then sends data to the log sink. You can then use the Processor convenience class, which has both an inbound channel and an outbound channel.

To do so:

1. Visit the Spring Initialzr site.
   a. Create a new Maven project with a **Group** name of `io.spring.stream.sample` and an **Artifact** name of `transformer`.
   b. In the Dependencies text box, type `stream` to select the Spring Cloud Stream dependency.
   c. In the Dependencies text box, type either `kafka` or `rabbit` and select which middleware you want to use.
   d. Click the **Generate Project** button
2. Unzip the project and bring the project into your favorite IDE.
3. Create a class called Transformer in the `io.spring.stream.sample` package with the following contents:

```
package io.spring.stream.sample;

import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Processor;
import org.springframework.messaging.handler.annotation.Payload;

import java.util.HashMap;
import java.util.Map;

@EnableBinding(Processor.class)
public class Transformer {

    @StreamListener(Processor.INPUT)
    @Output(Processor.OUTPUT)
    public Map<String, Object> transform(@Payload Map<String, Object> doc) {
        Map<String, Object> map = new HashMap<>();
        map.put("sensor_id", doc.getOrDefault("id", "-1"));
        map.put("temp_val", doc.getOrDefault("temperature", "-999"));
        return map;
    }
}
```

   All that this processor is doing is changing the names of the keys in the map and providing a default value if one does not exist.

4. Open the `TransformerApplicationTests` class (which already exists) and create a simple unit test for the `Transformer` class, as shown in the following example:

```
package io.spring.stream.sample;


import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.HashMap;
import java.util.Map;

import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.entry;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class TransformerApplicationTests {

    @Autowired
    private Transformer transformer;

    @Test
    public void simpleTest() {
        Map<String, Object> resultMap = transformer.transform(createInputData());
        assertThat(resultMap).hasSize(2)
                .contains(entry("sensor_id", "1"))
                .contains(entry("temp_val", "100"));
    }
```

```
    private Map<String, Object> createInputData() {
        HashMap<String, Object> inputData = new HashMap<>();
        inputData.put("id", "1");
        inputData.put("temperature", "100");
        return inputData;
    }
}
```

Executing `./mvnw clean package` in the root directory of the transformer project generates the artifact `transformer-0.0.1-SNAPSHOT.jar` under the `target` directory.

Now you can run all three applications, as shown in the following listing:

```
java -Dserver.port=8123 \
     -Dhttp.path-pattern=/data \
     -Dspring.cloud.stream.bindings.output.destination=sensorData \
     -jar http-source-rabbit-1.3.1.RELEASE.jar

java -Dserver.port=8090 \
 -Dspring.cloud.stream.bindings.input.destination=sensorData \
 -Dspring.cloud.stream.bindings.output.destination=normalizedSensorData \
 -jar transformer-0.0.1-SNAPSHOT.jar

java -Dlog.level=WARN \
     -Dspring.cloud.stream.bindings.input.destination=normalizedSensorData \
     -jar log-sink-rabbit-1.3.1.RELEASE.jar
```

Now you can post some content to the http source application, as follows:

```
curl -H "Content-Type: application/json" -X POST -d '{"id":"2","temperature":"200"}' http://localhost:8123/data
```

The preceding `curl` command results in the log sink showing the following output:

```
2017-03-24 16:09:42.726  WARN 7839 --- [Raj4gYSoR_6YA-1] log-sink                                 : {sensor_id=2,
temp_val=200}
```

32. Improving the Quality of Service

Without additional configuration, RabbitMQ applications that produce data create a durable topic exchange. Similarly, a RabbitMQ application that consumes data creates an anonymous auto-delete queue. This can result in a message not being stored and forwarded by the producer if the producer application started before the consumer application. Even though the exchange is durable, there needs to be a durable queue bound to the exchange for the message to be stored for later consumption.

To pre-create durable queues and bind them to the exchange, producer applications should set the `spring.cloud.stream.bindings.<channelName>.producer.requiredGroups` property. The `requiredGroups` property accepts a comma-separated list of groups to which the producer must ensure message delivery even if they start after it has been created. The consumer applications should then specify the `spring.cloud.stream.bindings.<channelName>.group` property to consume from the durable queue. The comma-separated list of groups for both properties should generally match. Consumer groups are also the means by which multiple instances of a consuming application can participate in a competing consumer relationship with other members of the same consumer group.

The following listing shows multiple applications sharing the same groups:

```
java -Dserver.port=8123 \
     -Dhttp.path-pattern=/data \
     -Dspring.cloud.stream.bindings.output.destination=sensorData \
     -Dspring.cloud.stream.bindings.output.producer.requiredGroups=sensorDataGroup \
     -jar http-source-rabbit-1.3.1.RELEASE.jar

java -Dserver.port=8090 \
     -Dspring.cloud.stream.bindings.input.destination=sensorData \
     -Dspring.cloud.stream.bindings.input.group=sensorDataGroup \
     -Dspring.cloud.stream.bindings.output.destination=normalizedSensorData \
     -Dspring.cloud.stream.bindings.output.producer.requiredGroups=normalizedSensorDataGroup \
     -jar transformer-0.0.1-SNAPSHOT.jar

java -Dlog.level=WARN \
     -Dspring.cloud.stream.bindings.input.destination=normalizedSensorData \
     -Dspring.cloud.stream.bindings.input.group=normalizedSensorDataGroup \
     -jar log-sink-rabbit-1.3.1.RELEASE.jar
```

As before, posting data to the `http` source results in the same log message in the sink.

33. Mapping Applications onto Data Flow

Spring Cloud Data Flow (SCDF) provides a higher level way to create this group of three Spring Cloud Stream

applications by introducing the concept of a stream. A stream is defined by using Unix-like pipes and a filtering DSL. Each application is first registered with a simple name, such as `http`, `transformer`, and `log` (for the applications we are using in this example). The stream DSL to connect these three applications is `http | transformer | log`.

Spring Cloud Data Flow has server and shell components. Through the shell, you can easily register applications under a name and also create and deploy streams. You can also use the JavaDSL to perform the same actions. However, we use the shell for the examples in this chapter.

In the shell application, register the jar files you have on your local machine by using the following commands. In this example, the `http` and `log` applications are in the `/home/mpollack/temp/dev` directory and the `transformer` application is in the `/home/mpollack/dev-marketing/transformer/target` directory.

The following commands register the three applications:

```
dataflow:>app register --type source --name http --uri file://home/mpollack/temp/dev/http-source-rabbit-1.2.0.BUILD-
SNAPSHOT.jar

dataflow:>app register --type processor --name transformer --uri file://home/mpollack/dev-
marketing/transformer/target/transformer-0.0.1-SNAPSHOT.jar

dataflow:>app register --type sink --name log --uri file://home/mpollack/temp/dev/log-sink-rabbit-1.1.1.RELEASE.jar
```

Now you can create a stream definition and deploy it with the following command:

```
stream create --name httpIngest --definition "http --server.port=8123 --path-pattern=/data | transformer --
server.port=8090 | log --level=WARN" --deploy
```

Then, in the shell, you can query for the list of streams, as shown (with output) in the following listing:

```
dataflow:>stream list
╔═══════════╤═══════════════════════════════════════════════════════════════════════════════╤═══════╗
║Stream Name│                                Stream Definition                              │ Status ║
╠═══════════╪═══════════════════════════════════════════════════════════════════════════════╪═══════╣
║httpIngest │http --server.port=8123 --path-pattern=/data | transformer --server.port=8090 | log --level=WARN│Deploying║
╚═══════════╧═══════════════════════════════════════════════════════════════════════════════╧═══════╝
```

Eventually, you can see the status column say `Deployed`.

In the server log, you can see output similar to the following:

```
2017-03-24 17:12:44.071  INFO 9829 --- [nio-9393-exec-6] o.s.c.d.spi.local.LocalAppDeployer       : deploying app
httpIngest.log instance 0
   Logs will be in /tmp/spring-cloud-dataflow-4401025649434774446/httpIngest-1490389964038/httpIngest.log
2017-03-24 17:12:44.153  INFO 9829 --- [nio-9393-exec-6] o.s.c.d.spi.local.LocalAppDeployer       : deploying app
httpIngest.transformer instance 0
   Logs will be in /tmp/spring-cloud-dataflow-4401025649434774446/httpIngest-1490389964143/httpIngest.transformer
2017-03-24 17:12:44.285  INFO 9829 --- [nio-9393-exec-6] o.s.c.d.spi.local.LocalAppDeployer       : deploying app
httpIngest.http instance 0
   Logs will be in /tmp/spring-cloud-dataflow-4401025649434774446/httpIngest-1490389964264/httpIngest.http
```

You can go to each directory to see the logs of each application. In the RabbitMQ management console, you can see two exchanges and two durable queues.

The SCDF server has configured the input and output destinations, through the `requiredGroups` and `group` properties, for each application, as was done explicitly in the previous example.

Now you can post some content to the http source application, as follows:

```
curl -H "Content-Type: application/json" -X POST -d '{"id":"1","temperature":"100"}' http://localhost:8123/data
```

Using the `tail` command on the stdout_0.log file for the log sink then shows output similar to the following listing:

```
2017-03-24 17:29:55.280  WARN 11302 --- [er.httpIngest-1] log-sink                                 : {sensor_id=4,
temp_val=400}
```

If you access the Boot actuator endpoint for the applications, you can see the conventions that SCDF has made for the destination names, the consumer groups, and the requiredGroups configuration properties, as shown in the following listing:

```
# for the http source
"spring.cloud.stream.bindings.output.producer.requiredGroups": "httpIngest",
"spring.cloud.stream.bindings.output.destination": "httpIngest.http",
"spring.cloud.application.group": "httpIngest",


# For the transformer
"spring.cloud.stream.bindings.input.group": "httpIngest",
```

```
"spring.cloud.stream.bindings.output.producer.requiredGroups": "httpIngest",


"spring.cloud.stream.bindings.output.destination": "httpIngest.transformer",
"spring.cloud.stream.bindings.input.destination": "httpIngest.http",
"spring.cloud.application.group": "httpIngest",

# for the log sink
"spring.cloud.stream.bindings.input.group": "httpIngest",
"spring.cloud.stream.bindings.input.destination": "httpIngest.transformer",
"spring.cloud.application.group": "httpIngest",
```

Tasks

This section goes into more detail about how you can orchestrateSpring Cloud Task applications on Spring Cloud Data Flow.

If you are just starting out with Spring Cloud Data Flow, you should probably read the Getting Started guide for "Local" , "Cloud Foundry", "Kubernetes" before diving into this section.

34. Introduction

A task application is short lived, meaning it stops running on purpose, and can be executed on demand or scheduled for execution. A use case might be to scrape a web page and write to the database. The Spring Cloud Task framework is based on Spring Boot and adds the capability for Boot applications to record the lifecycle events of a short lived application such as when it starts, when it ends and the exit status. The TaskExecution documentation shows which information is stored in the database. The entry point for code execution in a Spring Cloud Task application is most often an implementation of Boot's CommandLineRunner interface as shown in this example.

The Spring Batch project is probably what comes to mind for Spring developers writing short lived applications. Spring Batch provides a much richer set of functionality than Spring Cloud Task and is recommended when processing large volumes of data. A use case might be to read many CSV files, transform each row of data, and write each transformed row to a database. Spring Batch provides its own database schema with a much more rich set of information about the execution of a Spring Batch job. Spring Cloud Task is integrated with Spring Batch so that if a Spring Cloud Task application defined a Spring Batch `Job`, a link between the Spring Cloud Task and Spring Cloud Batch execution tables is created.

When running Data Flow on your local machine, Tasks are launched in a separate JVM. When running on Cloud Foundry, tasks are launched using Cloud Foundry's Task functional and when running on Kubernetes, task are launched using the `Job` kind type.

---

35. 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. Creating a Task Application
2. Registering a Task Application
3. Creating a Task Definition
4. Launching a Task
5. Reviewing Task Executions
6. Destroying a Task Definition

35.1. Creating a Task Application

While Spring Cloud Task does provide a number of out-of-the-box applications (atspring-cloud-task-app-starters), most task applications require custom development. To create a custom task application:

1. Use the Spring Initializer to create a new project, making sure to select the following starters:
   a. `Cloud Task`: This dependency is the `spring-cloud-starter-task`.
   b. `JDBC`: This dependency is the `spring-jdbc` starter.
2. Within your new project, create a new class to serve as your main class, as follows:

```
@EnableTask
@SpringBootApplication
public class MyTask {

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

```
  }
}
```

3. With this class, you need one or more `CommandLineRunner` or `ApplicationRunner` implementations 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 with Spring Boot into an über jar is done through the standard Spring Boot conventions. The packaged application can be registered and deployed as noted below.

35.1.1. Task Database Configuration

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.

When you run tasks externally (that is, from the command line) and you want 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 uses a local H2 instance, and the execution is recorded to the database used by Spring Cloud Data Flow.

35.2. Registering a Task Application

You can register a Task App with the App Registry by 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". The following listing shows three 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, the followinng listing would be a valid properties file:

```
task.foo=file:///tmp/foo-1.2.1.BUILD-SNAPSHOT.jar
task.bar=file:///tmp/bar-1.2.1.BUILD-SNAPSHOT.jar
```

Then you can use the `app import` command and provide the location of the properties file by using the `--uri` option, as follows:

```
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 earlier in this chapter, 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.

The following table lists the available static property files:

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

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

```
dataflow:>app import --uri http://bit.ly/Clark-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 is not overridden by default. If you would like to override the pre-existing task app, then include the `--force` option.

In some cases, the Resource is resolved on the server side. In other cases, the URI is passed to a runtime container instance where it is resolved. Consult the specific documentation of each Data Flow Server for more detail.

35.3. Creating a Task Definition

You can 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 through the RESTful API or the shell. To create a task definition by using the shell, use the `task create` command to create the task definition, as shown in the following example:

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

A listing of the current task definitions can be obtained through the RESTful API or the shell. To get the task definition list by using the shell, use the `task list` command.

35.4. Launching a Task

An adhoc task can be launched through the RESTful API or the shell. To launch an ad-hoc task through the shell, use the `task launch` command, as shown in the following example:

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

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

```
dataflow:>task launch mytask --arguments "--server.port=8080 --custom=value"
```

The arguments need to be passed as `space` delimited values.

Additional properties meant for a `TaskLauncher` itself can be passed in by using a `--properties` option. The format of this option is a comma-separated string of properties prefixed with `app.<task definition name>.<property>`. Properties are passed to `TaskLauncher` as application properties. 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.custom1=value1,app.timestamp.custom2=value2"`

35.4.1. Application properties

Each application takes properties to customize its behavior. As an example, the `timestamp` task `format` setting establishes a output format that is different from the default value.

`dataflow:> task create --definition "timestamp --format=\"yyyy\"" --name printTimeStamp`

This `timestamp` property is actually the same as the `timestamp.format` property specified by the timestamp application. Data Flow adds the ability to use the shorthand form `format` instead of `timestamp.format`. One may also specify the longhand version as well, as shown in the following example:

`dataflow:> task create --definition "timestamp --timestamp.format=\"yyyy\"" --name printTimeStamp`

This shorthand behavior is discussed more in the section on Whitelisting application properties. If you have registered application property metadata you can use tab completion in the shell after typing `--` to get a list of candidate property names.

The shell provides tab completion for application properties. The shell command `app info --name <appName> --type <appType>` provides additional documentation for all the supported properties.

The supported Task `<appType>` is task.

35.4.2. Common application properties

In addition to configuration through 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 passes all the properties, without the prefix, to the instances it launches.

For example, all the launched applications can be configured to use the properties `prop1` and `prop2` by launching the Data Flow server with the following options:

```
--spring.cloud.dataflow.applicationProperties.task.prop1=value1
--spring.cloud.dataflow.applicationProperties.task.prop2=value2
```

This causes the properties, `prop1=value1` and `prop2=value2`, to be passed to all the launched applications.

Properties configured by using this mechanism have lower precedence than task deployment properties. They are overridden if a property with the same key is specified at task launch time (for example, `app.trigger.prop2` overrides the common property).

35.5. Limit the number concurrent task launches

Spring Cloud Data Flow allows a user establish the maximum number of concurrently running tasks to prevent the saturation of IaaS/hardware resources. This limit can be configured by setting the `spring.cloud.dataflow.task.maximum-concurrent-tasks` property. By default it is set to `20`. If the number of concurrently running tasks is equal or greater than the value set by `spring.cloud.dataflow.task.maximum-concurrent-tasks` the next task launch request will be declined and a warning message will be returned via the RESTful API, Shell or UI.

35.6. 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 through the RESTful API or the shell. To display the latest task executions through 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 execution status` command with the id of the task execution, for example `task execution status --id 549`.

35.7. Destroying a Task Definition

Destroying a Task Definition removes the definition from the definition repository. This can be done through the RESTful API or the shell. To destroy a task through the shell, use the `task destroy` command, as shown in the following example:

```
dataflow:>task destroy mytask
 Destroyed task 'mytask'
```

To destroy all tasks through the shell, use the `task all destroy` command as shown in the following example:

```
dataflow:>task all destroy
 Really destroy all tasks? [y, n]: y
 All tasks destroyed
```

Or use the force command:

```
dataflow:>task all destroy --force
 All tasks destroyed
```

The task execution information for previously launched tasks for the definition remains in the task repository.

This does not stop any currently executing tasks for this definition. Instead, it removes the task

definition from the database.

The `task destroy <task-name>` deletes only the definition and not the task deployed on Cloud Foundry. The only way to do this now is through the CLI in two steps. First, obtain a list of the apps by using the `cf apps` command. . Identify the task application to be deleted and run the `cf delete <task-name>` command.

35.8. Validating a Task

Sometimes the one or more of the apps contained within a task definition contain an invalid URI in its registration. This can be caused by an invalid URI entered at app registration time or the app was removed from the repository from which it was to be drawn. To verify that all the apps contained in a task are resolve-able, a user can use the `validate` command. For example:

```
dataflow:>task validate time-stamp
╔═══════════╤═══════════════╗
║Task Name  │Task Definition║
╠═══════════╪═══════════════╣
║time-stamp│timestamp      ║
╚═══════════╧═══════════════╝


time-stamp is a valid task.
╔═══════════════╤═══════════════╗
║   App Name    │Validation Status║
╠═══════════════╪═══════════════╣
║task:timestamp │valid          ║
╚═══════════════╧═══════════════╝
```

In the example above the user validated their time-stamp task. As we see `task:timestamp` app is valid. Now let's see what happens if we have a stream definition with a registered app with an invalid URI.

```
dataflow:>task validate bad-timestamp
╔═══════════════╤═══════════════╗
║  Task Name    │Task Definition║
╠═══════════════╪═══════════════╣
║bad-timestamp│badtimestamp   ║
╚═══════════════╧═══════════════╝


bad-timestamp is an invalid task.
╔═══════════════════╤═══════════════╗
║     App Name      │Validation Status║
╠═══════════════════╪═══════════════╣
║task:badtimestamp │invalid        ║
╚═══════════════════╧═══════════════╝
```

In this case Spring Cloud Data Flow states that the task is invalid because task:badtimestamp has an invalid URI.

36. Subscribing to Task/Batch Events

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

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

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

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

The following table lists the default task and batch event and destination names on the broker:

Table 1. Task and 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`

37. Composed Tasks

Spring Cloud Data Flow lets a user 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 through the RESTful API, the Spring Cloud Data Flow Shell, or the Spring Cloud Data Flow UI.

37.1. Configuring the Composed Task Runner

Composed tasks are executed through a task application called the Composed Task Runner.

37.1.1. Registering the Composed Task Runner

By default, the Composed Task Runner application is not registered with Spring Cloud Data Flow. Consequently, 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 by using that property.

37.1.2. Configuring the Composed Task Runner

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, as you would if you deploy the server on Cloud Foundry, YARN, or Kubernetes, 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 is automatically set when a composed task is launched.

In some cases, you may wish to execute an instance of the Composed Task Runner through the Task Launcher sink. In that case, you must configure the Composed Task Runner to use the same datasource that the Spring Cloud Data Flow instance is using. The datasource properties are set with the `TaskLaunchRequest` through the use of the `commandlineArguments` or the `environmentProperties` switches. This is because the Composed Task Runner monitors the `task_executions` table to check the status of the tasks that it is running. Using information from the table, it determines how it should navigate the graph.

Configuration Options

The ComposedTaskRunner task has the following options:

- **increment-instance-enabled** Allows a single ComposedTaskRunner instance to be re-executed without changing the parameters. Default is false which means a ComposedTaskRunner instance can only be executed once with a given set of parameters, if true it can be re-executed. (Boolean, default: false). ComposedTaskRunner is built using Spring Batch and thus upon a successful execution the batch job is considered complete. To launch the same ComposedTaskRunner definition multiple times you must set the `increment-instance-enabled` property to true or change the parameters for the definition for each launch.
- **interval-time-between-checks** The amount of time in millis that the ComposedTaskRunner will wait between checks of the database to see if a task has completed. (Integer, default: 10000). ComposedTaskRunner uses the datastore to determine the status of each child tasks. This interval indicates to ComposedTaskRunner how often it should check the status its child tasks.
- **max-wait-time** The maximum amount of time in millis that a individual step can run before the execution of the Composed task is failed (Integer, default: 0). Determines the maximum time each child task is allowed to run before the CTR will terminate with a failure. The default of `0` indicates no timeout.
- **split-thread-allow-core-thread-timeout** Specifies whether to allow split core threads to timeout. Default is false; (Boolean, default: false) Sets the policy governing whether core threads may timeout and terminate if no tasks arrive within the keep-alive time, being replaced if needed when new tasks arrive.

- **split-thread-core-pool-size** Split's core pool size. Default is 1; (Integer, default: 1) Each child task contained in a split requires a thread in order to execute. So for example a definition like: `<AAA || BBB || CCC> && <DDD || EEE>` would require a split-thread-core-pool-size of 3. This is because the largest split contains 3 child tasks. A count of 2 would mean that `AAA` and `BBB` would run in parallel but CCC would wait until either `AAA` or `BBB` to finish in order to run. Then `DDD` and `EEE` would run in parallel.
- **split-thread-keep-alive-seconds** Split's thread keep alive seconds. Default is 60. (Integer, default: 60) If the pool currently has more than corePoolSize threads, excess threads will be terminated if they have been idle for more than the keepAliveTime.
- **split-thread-max-pool-size** Split's maximum pool size. Default is {@code Integer.MAX_VALUE} (Integer, default: <none>). Establish the maximum number of threads allowed for the thread pool.
- **split-thread-queue-capacity** Capacity for Split's BlockingQueue. Default is {@code Integer.MAX_VALUE}. (Integer, default: <none>)
  - If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing.
  - If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.
  - If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.
- **split-thread-wait-for-tasks-to-complete-on-shutdown** Whether to wait for scheduled tasks to complete on shutdown, not interrupting running tasks and executing all tasks in the queue. Default is false; (Boolean, default: false)

Note when using the options above as environment variables, convert to uppercase, remove the dash character and replace with the underscore character. For example: increment-instance-enabled would be INCREMENT_INSTANCE_ENABLED.

37.2. The Lifecycle of a Composed Task

The lifecycle of a composed task has three parts:

- Creating a Composed Task
- Stopping a Composed Task
- Restarting a Composed Task

37.2.1. Creating a Composed Task

The DSL for the composed tasks is used when creating a task definition through the task create command, as shown in the following 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 preceding example, we assume that the applications to be used by our composed task have not been registered yet. Consequently, in 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 preceding example, when launched, runs mytaskapp and then runs 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, as shown (including its output) in the following example:

```
dataflow:>task list
╔═══════════════════════════╤═══════════════════════════╤═══════════╗
║         Task Name         │    Task Definition        │Task Status║
╠═══════════════════════════╪═══════════════════════════╪═══════════╣
║my-composed-task           │mytaskapp && timestamp     │unknown    ║
║my-composed-task-mytaskapp │mytaskapp                  │unknown    ║
║my-composed-task-timestamp │timestamp                  │unknown    ║
╚═══════════════════════════╧═══════════════════════════╧═══════════╝
```

In the example, Spring Cloud Data Flow created three task definitions, one for each of the applications that makes up 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 made up of the name of the composed task and the name of the application, separated by a dash `-` (as in *my-composed-task* `-` *mytaskapp*).

Task Application Parameters

The task applications that make up the composed task definition can also contain parameters, as shown in the following example:

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

37.2.2. Launching a Composed Task

Launching a composed task is done the same way as launching a stand-alone task, as follows:

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

Once the task is launched, and assuming all the tasks complete successfully, you can see three task executions when executing a `task execution list`, as shown in the following 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 preceding example, we see that `my-compose-task` launched and that it also launched the other tasks in sequential order. All of them executed successfully with `Exit Code` as `0`.

Passing properties to the child tasks

To set the properties for child tasks in a composed task graph at task launch time, you would use the following format of `app.<composed task definition name>.<child task app name>.<property>`. Using the following Composed Task definition as an example:

```
dataflow:> task create my-composed-task --definition "mytaskapp  && mytimestamp"
```

To have mytaskapp display 'HELLO' and set the mytimestamp timestamp format to 'YYYY' for the Composed Task definition, you would use the following task launch format:

```
task launch my-composed-task --properties "app.my-composed-task.mytaskapp.displayMessage=HELLO,app.my-composed-task.mytimestamp.timestamp.format=YYYY"
```

Similar to application properties, the `deployer` properties can also be set for child tasks using the format format of `deployer.<composed task definition name>.<child task app name>.<deployer-property>`.

```
task launch my-composed-task --properties "deployer.my-composed-task.mytaskapp.memory=2048m,app.my-composed-task.mytimestamp.timestamp.format=HH:mm:ss"
Launched task 'a1'
```

Passing arguments to the composed task runner

Command line arguments for the composed task runner can be passed using `--arguments` option.

For example:

```
dataflow:>task create my-composed-task --definition "<aaa: timestamp || bbb: timestamp>"
Created new task 'my-composed-task'

dataflow:>task launch my-composed-task --arguments "--increment-instance-enabled=true --max-wait-time=50000 --split-thread-core-pool-size=4" --properties "app.my-composed-task.bbb.timestamp.format=dd/MM/yyyy HH:mm:ss"
Launched task 'my-composed-task'
```

Exit Statuses

The following list shows how the Exit Status is set for each step (task) contained in the composed task following each step execution:

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

37.2.3. Destroying a Composed Task

The command used to destroy a stand-alone task is the same as the command used to destroy a composed task. The only difference is that destroying a composed task also destroys the child tasks associated with it. The following example shows the task list before and after using the `destroy` command:

```
dataflow:>task list
╔═══════════════════════╤═══════════════════════╤═══════════╗
║        Task Name      │   Task Definition     │Task Status║
╠═══════════════════════╪═══════════════════════╪═══════════╣
║my-composed-task       │mytaskapp && timestamp │COMPLETED  ║
║my-composed-task-mytask│mytaskapp              │COMPLETED  ║
║my-composed-task-timest│timestamp              │COMPLETED  ║
╚═══════════════════════╧═══════════════════════╧═══════════╝
...
dataflow:>task destroy my-composed-task
dataflow:>task list
╔═══════╤═══════════╤═══════════╗
║Task Name│Task Definition│Task Status║
╚═══════╧═══════════╧═══════════╝
```

37.2.4. Stopping a Composed Task

In cases where a composed task execution needs to be stopped, you can do so through the:

- RESTful API
- Spring Cloud Data Flow Dashboard

To stop a composed task through the dashboard, select the Jobs tab and click the Stop button next to the job execution that you want to stop.

The composed task run is 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 is marked as `STOPPED` as well as the composed task job execution.

37.2.5. Restarting a Composed Task

In cases where a composed task fails during execution and the status of the composed task is `FAILED`, the task can be restarted. You can do so through the:

- RESTful API
- The shell
- Spring Cloud Data Flow Dashboard

To restart a composed task through the shell, launch the task with the same parameters. To restart a composed task through the dashboard, select the Jobs tab and click the Restart button next to the job execution that you want to restart.

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

---

38. Composed Tasks DSL

Composed tasks can be run in three ways:

- Conditional Execution
- Transitional Execution
- Split Execution

38.1. Conditional Execution

Conditional execution is expressed by using a double ampersand symbol (`&&`). This lets each task in the sequence be launched only if the previous task successfully completed, as shown in the following example:

```
task create my-composed-task --definition "task1 && task2"
```

When the composed task called `my-composed-task` is launched, it launches the task called `task1` and, if it completes successfully, then the task called `task2` is launched. If `task1` fails, then `task2` does 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, as shown in the following image:

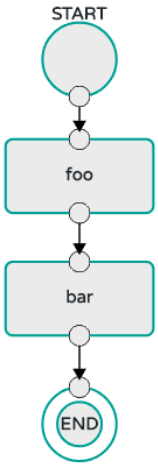


Figure 10. Conditional Execution

The preceding diagram is a screen capture of the directed graph as it being created by using the Spring Cloud Data Flow Dashboard. You can see that are four components in the diagram that comprise a conditional execution:

- Start icon: All directed graphs start from this symbol. There is only 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.
- End icon: All directed graphs end at this symbol.

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

38.2. 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 `->`.

38.2.1. Basic Transition

A basic transition would look like the following:

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

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

Using the Spring Cloud Data Flow Dashboard to create the same "`basic transition`" would resemble the following image:

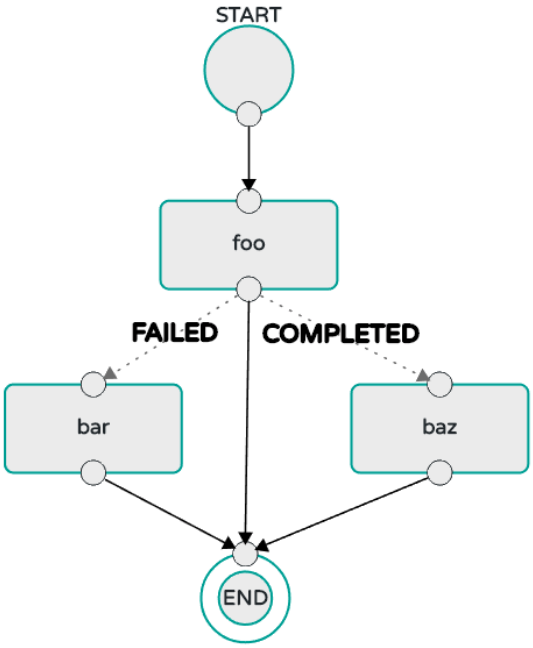


Figure 11. Basic Transition

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

- Dashed line: Represents transitions from the application to one of the possible destination applications.
- Solid line: Connects applications in a conditional execution or a connection between the application and a control node (start or end).

To create a transitional connector:

1. When creating a transition, link the application to each possible destination by using the connector.
2. Once complete, go to each connection and select it by clicking it.
3. A bolt icon appears.
4. Click that icon.
5. Enter the exit status required for that connector.
6. The solid line for that connector turns to a dashed line.

38.2.2. Transition With a Wildcard

Wildcards are supported for transitions by the DSL, as shown in the following:

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

In the preceding example, `foo` would launch, and, if it had an exit status of `FAILED`, the `bar` task would launch. For any exit status of `foo` other than `FAILED`, `baz` would launch.

Using the Spring Cloud Data Flow Dashboard to create the same “transition with wildcard” would resemble the following image:

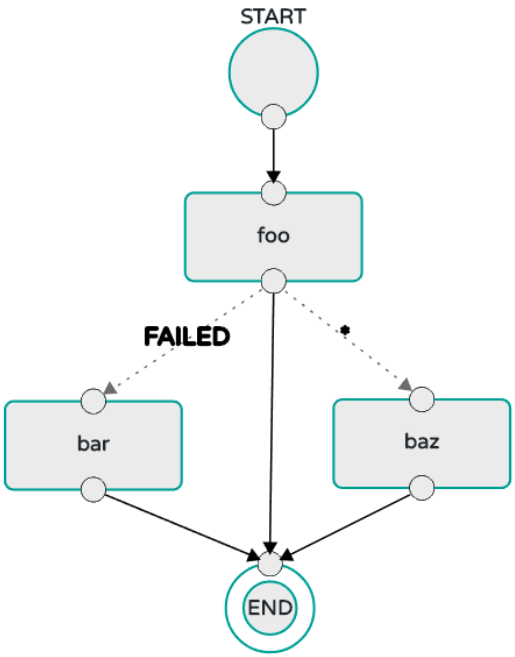


Figure 12. Basic Transition With Wildcard

38.2.3. Transition With a Following Conditional Execution

A transition can be followed by a conditional execution so long as the wildcard is not used, as shown in the following example:

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

In the preceding example, `foo` would launch, and, if it had an exit status of `FAILED`, the `bar` task would launch. If `foo` had an exit status of `UNKNOWN`, `baz` would launch. For any exit status of `foo` other than `FAILED` or `UNKNOWN`, `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 resemble the following image:

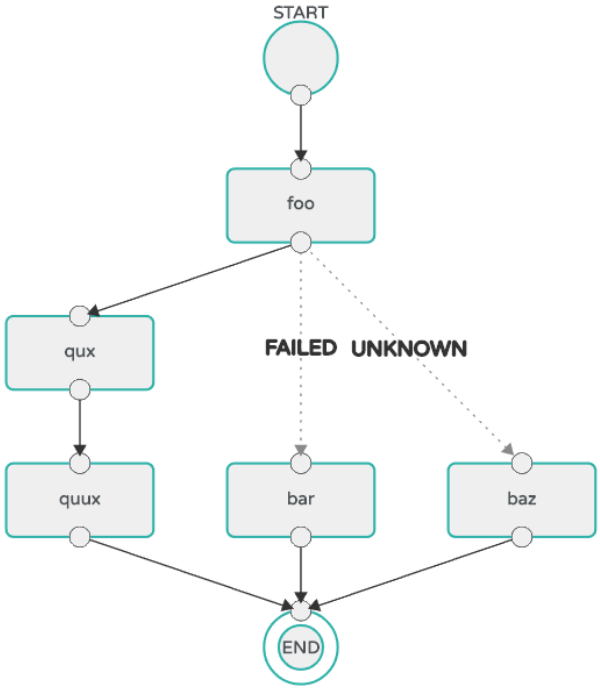


Figure 13. Transition With Conditional Execution

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`, `qux`, and `quux`.

38.3. Split Execution

Splits allow 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 `||`

symbol, as shown in the following example:

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

The preceding example above launches tasks `foo`, `bar` and `baz` in parallel.

Using the Spring Cloud Data Flow Dashboard to create the same “split execution” would resemble the following image:

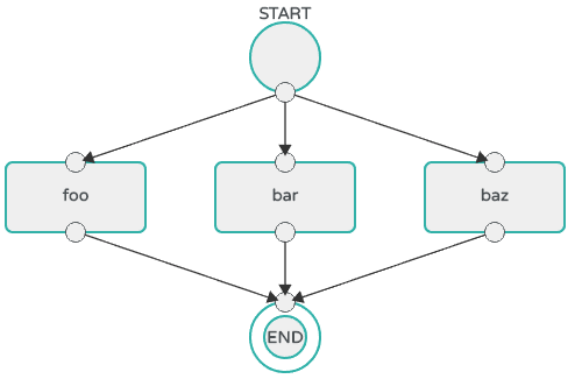


Figure 14. Split

With the task DSL, a user may also execute multiple split groups in succession, as shown in the following example:

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

In the preceding example, tasks `foo`, `bar`, and `baz` are launched in parallel. Once they all complete, then tasks `qux` and `quux` are launched in parallel. Once they complete, the composed task ends. However, if `foo`, `bar`, or `baz` fails, the split containing `qux` and `quux` does not launch.

Using the Spring Cloud Data Flow Dashboard to create the same “split with multiple groups” would resemble the following image:

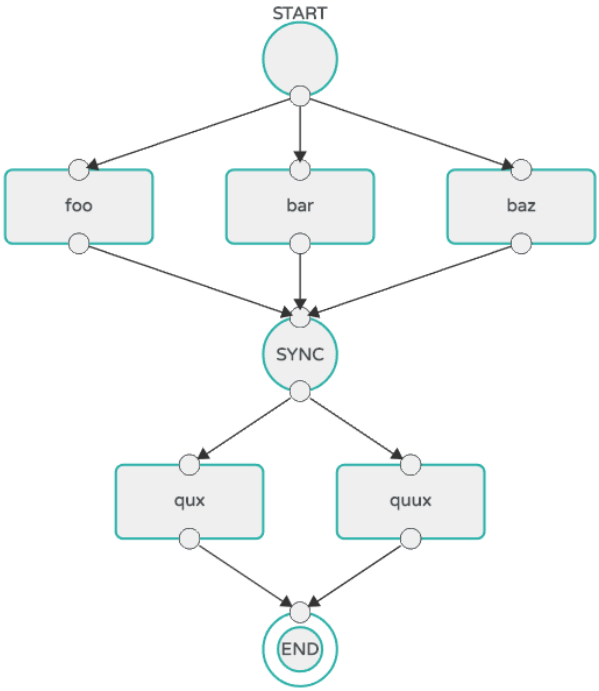


Figure 15. Split as a part of a conditional execution

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

Tasks that are used in a split should not set the their `ExitMessage`. Setting the `ExitMessage` is only to be used with transitions.

38.3.1. Split Containing Conditional Execution

A split can also have a conditional execution within the angle brackets, as shown in the following example:

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

In the preceding example, we see that `foo` and `baz` are launched in parallel. However, `bar` does not launch until `foo` completes successfully.

Using the Spring Cloud Data Flow Dashboard to create the same "`split containing conditional execution`" resembles the following image:

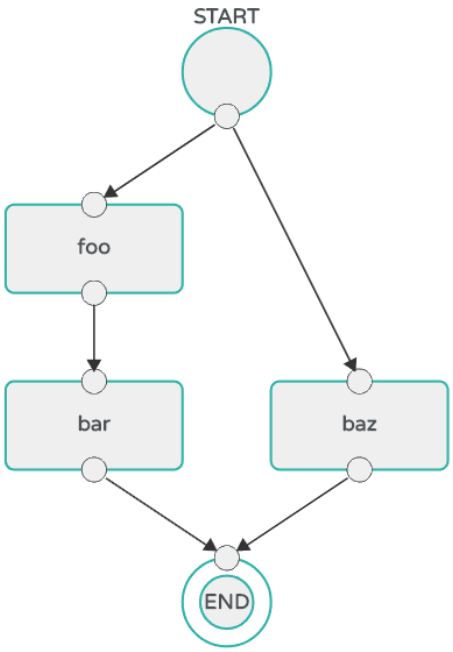


Figure 16. Split with conditional execution

38.3.2. Establishing the proper thread count for splits

Each child task contained in a split requires a thread in order to execute. To set this properly you want to look at your graph and count the split that has the largest number of child tasks, this will be the number of threads you will need to utilize. To set the thread count use the split-thread-core-pool-size property (defaults to 1). So for example a definition like: `<AAA || BBB || CCC> && <DDD || EEE>` would require a split-thread-core-pool-size of 3. This is because the largest split contains 3 child tasks. A count of 2 would mean that `AAA` and `BBB` would run in parallel but CCC would wait until either `AAA` or `BBB` to finish in order to run. Then `DDD` and `EEE` would run in parallel.

---

39. Launching Tasks from a Stream

You can launch a task from a stream by using thetasklauncher-dataflow sink. The sink connects to a Data Flow server and uses its REST API to launch any defined task. The sink accepts a JSON payload representing a `task launch request` which provides the name of the task to launch, and may include command line arguments and deployment properties.

The app-starters-task-launch-request-common component , in conjunction with Spring Cloud Streamfunctional composition, can transform the output of any source or processor to a task launch request.

Adding a dependency to `app-starters-task-launch-request-common`, auto-configures a `java.util.function.Function` implementation, registered via Spring Cloud Function as `taskLaunchRequest`.

For example, you can start with the time source, add the following dependency, build it, and register it as a custom source. We'll call it `time-tlr` in this example.

```
<dependency>
    <groupId>org.springframework.cloud.stream.app</groupId>
    <artifactId>app-starters-task-launch-request-common</artifactId>
</dependency>
```

Spring Cloud Stream Initializr provides a great starting point for creating stream applications.

Next, register the `tasklauncher-dataflow` sink, and create a task (we will use the provided timestamp task).

```
stream create --name task-every-minute --definition "time-tlr --trigger.fixed-delay=60 --
spring.cloud.stream.function.definition=taskLaunchRequest --task.launch.request.task-name=timestamp-task | tasklauncher-
dataflow" --deploy
```

The preceding stream will produce a task launch request every minute. The request provides the name of the task to launch : `{"name":"timestamp-task"}`.

The following stream definition illustrates the use of command line arguments. It will produce messages like `{"args":`:

`["foo=bar","time=12/03/18 17:44:12"],"deploymentProps":{},"name":"timestamp-task"}` to provide command line arguments to the task:

```
stream create --name task-every-second --definition "time-tlr --
spring.cloud.stream.function.definition=taskLaunchRequest --task.launch.request.task-name=timestamp-task --
task.launch.request.args=foo=bar --task.launch.request.arg-expressions=time=payload | tasklauncher-dataflow" --deploy
```

Note the use of SpEL expressions to map each message payload to the `time` command line argument, along with a static argument `foo=bar`.

You can then see the list of task executions by using the shell command `task execution list`, as shown (with its output) in the following example:

```
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        ║
╚═══════════════════╧══╧═══════════════════════════╧═══════════════════════════╧═════════╝
```

In this example, we have shown how to use the `time` source to launch a task at a fixed rate. This pattern may be applied to any source to launch a task in response to any event.

39.1. Launching a Composed Task From a Stream

A composed task can be launched with the `tasklauncher-dataflow` sink, as discussed here. Since we use the `ComposedTaskRunner` directly, we need to set up the task definitions for the composed task runner itself, along with the composed tasks, prior to the creation of the composed task launching stream. Suppose we wanted to create the following composed task definition: `AAA && BBB`. The first step would be to create the task definitions, as shown in the following example:

```
task create composed-task-runner --definition "composed-task-runner"
task create AAA --definition "timestamp"
task create BBB --definition "timestamp"
```

Releases of `ComposedTaskRunner` can be found here.

Now that the task definitions we need for composed task definition are ready, we need to create a stream that launches `ComposedTaskRunner`. So, in this case, we create a stream with

- The `time` source customized to emit task launch requests, as shown above.
- The `tasklauncher-dataflow` sink that launches the `ComposedTaskRunner`

The stream should resemble the following:

```
stream create ctr-stream --definition "time --fixed-delay=30 --task.launch.request.task-name=composed-task-launcher --
task.launch.request.args=--graph=AAA&&BBB,--increment-instance-enabled=true | tasklauncher-dataflow"
```

For now, we focus on the configuration that is required to launch the `ComposedTaskRunner`:

- **graph**: this is the graph that is to be executed by the `ComposedTaskRunner`. In this case it is `AAA&&BBB`.
- **increment-instance-enabled**: This lets each execution of `ComposedTaskRunner` be unique. `ComposedTaskRunner` is built by using Spring Batch. Thus, we want a new Job Instance for each launch of the `ComposedTaskRunner`. To do this, we set `increment-instance-enabled` to be `true`.

40. Sharing Spring Cloud Data Flow's Datastore with Tasks

As discussed in the Tasks documentation Spring Cloud Data Flow allows a user to view Spring Cloud Task App executions. So in this section we will discuss what is required by a Task Application and Spring Cloud Data Flow to share the task execution information.

40.1. A Common DataStore Dependency

Spring Cloud Data Flow supports many databases out-of-the-box, so all the user typically has to do is declare the `spring_datasource_*` environment variables to establish what data store Spring Cloud Data Flow will need. So whatever database you decide to use for Spring Cloud Data Flow make sure that the your task also includes that database dependency in its `pom.xml` or `gradle.build` file. If the database dependency that is used by Spring Cloud

Data Flow is not present in the Task Application, the task will fail and the task execution will not be recorded.

40.2. A Common Data Store

Spring Cloud Data Flow and your task application must access the same datastore instance. This is so that the task executions recorded by the task application can be read by Spring Cloud Data Flow to list them in the Shell and Dashboard views. Also the task app must have read & write privileges to the task data tables that are used by Spring Cloud Data Flow.

Given the understanding of Datasource dependency between Task apps and Spring Cloud Data Flow, let's review how to apply them in various Task orchestration scenarios.

40.2.1. Simple Task Launch

When launching a task from Spring Cloud Data Flow, Data Flow adds its datasource properties (`spring.datasource.url`, `spring.datasource.driverClassName`, `spring.datasource.username`, `spring.datasource.password`) to the app properties of the task being launched. Thus a task application will record its task execution information to the Spring Cloud Data Flow repository.

40.2.2. Task Launcher Sink

The Data Flow Task Launcher Sink always uses the Data Flow Server's configured task database when launching tasks.

Standalone Task Launcher Sink implementations are also available which allow you to store task executions in a separate database. Since these task launchers do not use the Data Flow Server, they are platform-specific and require additional configuration parameters, including data source configuration, and the resource location of the executable jar for the task application. Additionally, they do not provide a way to limit the number of concurrently running tasks, as the Data Flow Task Launcher does.

The additional configuration requires a more complex form of theTaskLaunchRequest. Requests processed by a standalone Task Launcher Sink must include the required datasource information as app properties or command line arguments. Both TaskLaunchRequest-Transform and TriggerTask Source provide examples of using a standalone Task Launcher Sink.

Currently the platforms supported by the standalone `tasklauncher` sinks are:

- local
- Cloud Foundry
- Kubernetes

`tasklauncher-local` is meant for development purposes only.

40.2.3. Composed Task Runner

Spring Cloud Data Flow allows a user to create a directed graph where each node of the graph is a task application and this is done via the Composed Task Runner. In this case the rules that applied to a Simple Task Launch or Task Launcher Sink apply to the composed task runner as well. All child apps must also have access to the datastore that is being used by the composed task runner Also, All child apps must have the same database dependency as the composed task runner enumerated in their `pom.xml` or `gradle.build` file.

40.2.4. Launching a task externally from Spring Cloud Data Flow

Users may wish to launch Spring Cloud Task applications via another method (scheduler for example) but still track the task execution via Spring Cloud Data Flow. This can be done so long as the task applications observe the rules specified here and here.

If a user wishes to use Spring Cloud Data Flow to view their Spring Batch jobs, the user must make sure that their batch application use the `@EnableTask` annotation and follow the rules enumerated here and here. More information is available here.

41. Scheduling Tasks

Spring Cloud Data Flow lets a user schedule the execution of tasks via a cron expression. A schedule can be created through the RESTful API or the Spring Cloud Data Flow UI.

41.1. The Scheduler

Spring Cloud Data Flow will schedule the execution of its tasks via a scheduling agent that is available on the cloud platform. When using the Cloud Foundry platform Spring Cloud Data Flow will use the PCF Scheduler. When using Kubernetes, a CronJob will be used.

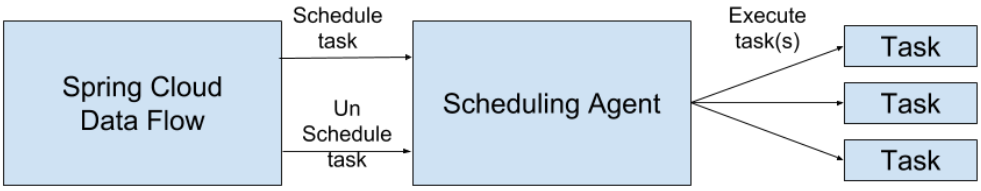


Figure 17. Architectural Overview

41.2. Enabling Scheduling

By default the Spring Cloud Data Flow leaves the scheduling feature disabled. To enable the scheduling feature the following feature properties must be set to `true`:

- `spring.cloud.dataflow.features.schedules-enabled`
- `spring.cloud.dataflow.features.tasks-enabled`

41.3. The Lifecycle of a Schedule

The lifecycle of a schedule has 2 parts:

- Scheduling a Task Execution
- Deleting a Schedule

41.3.1. Scheduling a Task Execution

You can schedule a task execution via the:

- RESTful API
- Spring Cloud Data Flow Dashboard

To schedule a task from the UI click the Tasks tab at the top of the screen, this will take you to the Task Definitions screen. Then from the Task Definition that you wish to schedule click the "clock" icon associated with task definition you wish to schedule. This will lead you to a `Create Schedule(s)` screen, where you will create a unique name for the schedule and enter the associated cron expression. Keep in mind you can always create multiple schedules for a single task definition.

41.3.2. Deleting a Schedule

You can delete a schedule via the:

- RESTful API
- Spring Cloud Data Flow Dashboard

To delete a schedule through the dashboard, select the Schedule tab under Tasks tab and click the `garbage can` icon next to the schedule you wish to delete.

Any currently running tasks that were run by the scheduling agent will not be stopped if the schedule is deleted. It only prevents future executions.

Task Developer Guide

This section covers how to create, test, and run Spring Cloud Task applications on your local machine. It also shows how to map these applications into Spring Cloud Data Flow and deploy them.

42. Prebuilt Applications

The Spring Cloud Task App Starters project provides many applications that you can start using right away. For example, there is a timestamp application that prints the timestamp to the console. All the applications are based on Spring Boot and Spring Cloud Task.

Applications are published as Maven artifacts as well as Docker images. For GA releases, the Maven artifacts are published to Maven central and the Spring Release Repository. Milestone and snapshot releases are published to the Spring Milestone and Spring Snapshot repositories, respectively. Docker images are pushed to Docker Hub

The root location of the Spring Repository that hosts the GA artifacts of prebuilt applications is repo.spring.io/release/org/springframework/cloud/task/app/

43. Running Prebuilt Applications

You can run the timestamp application by using `java -jar`.

To get started, download the sample application, as follows:

```
wget https://repo.spring.io/libs-release/org/springframework/cloud/task/app/timestamp-task/1.3.0.RELEASE/timestamp-task-1.3.0.RELEASE.jar
```

That file contains a Spring Boot applications that includes the Spring Boot Actuator and the Spring Security Starter. You can specify common Spring Boot properties to configure each application.

Now you can run the timestamp application, as follows:

```
java -jar timestamp-task-1.3.0.RELEASE.jar --logging.level.org.springframework.cloud.task=DEBUG
```

The `--logging.level.org.springframework.cloud.task=DEBUG` option lets you see output that would not otherwise be written to the console. Because Spring Cloud Task uses an in-memory database to store its results (and that in-memory database is destroyed at the end of the run), the `DEBUG` option is the only way to see the output from the timestamp task.

The timestamp application shows the following output (in the midst of much other output):

```
2018-03-12 13:45:14.583  INFO 4810 --- [           main] TimestampTaskConfiguration$TimestampTask : 2018-03-12 13:45:14.583
2018-03-12 13:45:14.609 DEBUG 4810 --- [           main] o.s.c.t.r.support.SimpleTaskRepository   : Updating: TaskExecution with executionId=1 with the following {exitCode=0, endTime=Mon Mar 12 13:45:14 CDT 2018, exitMessage='null', errorMessage='null'}
```

The first line shows the timestamp generated by the Timestamp task. The second line shows an exit code of 0 and no error. Had an error occurred, the exit code would be something other than 0, and the `errorMessage` would show the exception that was thrown.

If you have debug mode turned on, you can get even more information, in a line similar to the following (which appears just prior to the timestamp if you have debug mode turned on):

`2018-03-14 13:47:16.659 DEBUG 78382 --- [main] o.s.c.t.r.support.SimpleTaskRepository : Creating: TaskExecution{executionId=0, parentExecutionId=null, exitCode=null, taskName='application', startTime=Wed Mar 14 13:47:16 EDT 2018, endTime=null, exitMessage='null', externalExecutionId='null', errorMessage='null', arguments=[]}`

44. Building a Timestamp Task

To build your own timestamp task, follow each of these procedures:

1. Developing Your Timestamp Application
2. Recording an Error
3. Adding Pre- and Post-processing
4. Adding a MySQL Database
5. Adding and Launching Spring Cloud Tasks with Data Flow

45. Developing Your Timestamp Application

So now that we've used a pre-built timestamp task application, let's create one of our own.

45.1. Creating the Spring Task Project using Spring Initializr

Now let's create and test an application that prints the current time to the console.

To do so:

1. Visit the Spring Initialzr site.
   a. Create a new Maven or Gradle project with a **Group** name of `io.spring.task.sample` and an **Artifact** name of `timestamp`.
   b. In the Dependencies text box, type `task` to select the `Cloud Task` dependency.

c. In the Dependencies text box, type `jdbc` then select the `JDBC` dependency.

d. In the Dependencies text box, type `h2` then select the `H2`. (or your favorite database)

e. Click the **Generate Project** button

2. Unzip the timestamp.zip file and import the project into your favorite IDE.

45.2. Writing the Code

To finish our application, we need to modify the generated `TimestampApplication` code with the following contents so that it will launch a Task.

```
package io.spring.task.sample.timestamp;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.task.configuration.EnableTask;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableTask
public class TimestampApplication {

 @Bean
 public CommandLineRunner commandLineRunner() {
  return new TimestampCommandLineRunner();
 }

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

 public static class TimestampCommandLineRunner implements CommandLineRunner {

  @Override
  public void run(String... strings) throws Exception {
   DateFormat dateFormat = new SimpleDateFormat("YYYY-MM-dd HH:ss");
   System.out.println(dateFormat.format(new Date()));
  }
 }
}
```

The @EnableTask annotation sets up `TaskRepository` which stores information about the task execution such as the start and end time of the task and the exit code.

In our demo, the `TaskRepository` uses an embedded H2 database to record the results of a task. This H2 embedded database is not a practical solution for a production environment, since the H2 database goes away once the task ends. However, for a quick getting-started experience, we will use this in our example as well as echoing to the logs what is being updated in that repository.

The CommandLineRunner is a Spring Boot interface that tells Boot to execute the code in the `run` method once. When our sample application runs, Spring Boot launches our `TimestampCommandLineRunner` and outputs our timestamp message to standard out.

Any processing bootstrapped from mechanisms other than a `CommandLineRunner` or `ApplicationRunner` (by using `InitializingBean#afterPropertiesSet` for example) is not recorded by Spring Cloud Task.

Now let's open the `application.properties` file in `src/main/resources` and configure two properties, the application name and logging. The application name is also used as the name of the task. The logging level is set to `DEBUG` so we can see more information on what is going on internally.

```
logging.level.org.springframework.cloud.task=DEBUG
spring.application.name=timestamp
```

45.3. Running the Example

At this point, our application should work. Since this application is Spring Boot-based, we can run it from the command line by using `./mvnw spring-boot:run` from the root of our application, as shown (with its output) in the following example:

```
$ ./mvnw clean spring-boot:run
....... . . .
....... . . . (Maven log output here)
....... . . .

  .   ____          _            __ _ _
```

```
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.3.RELEASE)

2018-07-26 12:01:47.236  INFO 93883 --- [           main] i.s.t.s.timestamp.TimestampApplication   : Starting
TimestampApplication on Glenns-MacBook-Pro-2.local with PID 93883 (/Users/glennrenfro/project/timestamp/target/classes
started by glennrenfro in /Users/glennrenfro/project/timestamp)
2018-07-26 12:01:47.241  INFO 93883 --- [           main] i.s.t.s.timestamp.TimestampApplication   : No active profile
set, falling back to default profiles: default
2018-07-26 12:01:47.280  INFO 93883 --- [           main] s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@50b87e67: startup date [Thu Jul 26 12:01:47 EDT
2018]; root of context hierarchy
2018-07-26 12:01:47.989  INFO 93883 --- [           main] o.s.j.d.e.EmbeddedDatabaseFactory        : Starting embedded
database: url='jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=false', username='sa'
2018-07-26 12:01:48.144 DEBUG 93883 --- [           main] o.s.c.t.c.SimpleTaskConfiguration        : Using
org.springframework.cloud.task.configuration.DefaultTaskConfigurer TaskConfigurer
2018-07-26 12:01:48.145 DEBUG 93883 --- [           main] o.s.c.t.c.DefaultTaskConfigurer          : No EntityManager
was found, using DataSourceTransactionManager
2018-07-26 12:01:48.227 DEBUG 93883 --- [           main] o.s.c.t.r.s.TaskRepositoryInitializer    : Initializing task
schema for h2 database
2018-07-26 12:01:48.229  INFO 93883 --- [           main] o.s.jdbc.datasource.init.ScriptUtils     : Executing SQL
script from class path resource [org/springframework/cloud/task/schema-h2.sql]
2018-07-26 12:01:48.261  INFO 93883 --- [           main] o.s.jdbc.datasource.init.ScriptUtils     : Executed SQL script
from class path resource [org/springframework/cloud/task/schema-h2.sql] in 32 ms.
2018-07-26 12:01:48.407  INFO 93883 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans
for JMX exposure on startup
2018-07-26 12:01:48.412  INFO 93883 --- [           main] o.s.c.support.DefaultLifecycleProcessor  : Starting beans in
phase 0
2018-07-26 12:01:48.428 DEBUG 93883 --- [           main] o.s.c.t.r.support.SimpleTaskRepository   : Creating:
TaskExecution{executionId=0, parentExecutionId=null, exitCode=null, taskName='timestamp', startTime=Thu Jul 26 12:01:48
EDT 2018, endTime=null, exitMessage='null', externalExecutionId='null', errorMessage='null', arguments=[]}
2018-07-26 12:01:48.439  INFO 93883 --- [           main] i.s.t.s.timestamp.TimestampApplication   : Started
TimestampApplication in 1.464 seconds (JVM running for 4.233)
2018-07-26 12:48
2018-07-26 12:01:48.457 DEBUG 93883 --- [           main] o.s.c.t.r.support.SimpleTaskRepository   : Updating:
TaskExecution with executionId=1 with the following {exitCode=0, endTime=Thu Jul 26 12:01:48 EDT 2018,
exitMessage='null', errorMessage='null'}
```

The preceding output has three lines that of interest to us here:

- `SimpleTaskRepository` logged the creation of the entry in the `TaskRepository`.
- The execution of our `CommandLineRunner`, demonstrated by the timestamp output.
- `SimpleTaskRepository` logs the completion of the task in the `TaskRepository`.

45.4. Recording an Error

Now that we have a working task, we can intentionally create an error, to show how a Spring Cloud Task handles errors. To do so:

1. Open `src/main/java/io/spring/task/sample/timestamp/TimestampApplication.java` and insert the following line into the run method:

   ```
   throw new IllegalStateException("No Task For You!");
   ```

2. From the command line, run `./mvnw clean spring-boot:run -DskipTests`.

   We must add `-DskipTests`, because the tests would catch the Exception we added and prevent us from seeing it.

   Now we can see the Exception we added coming through in the output, as an Exception with a stack trace. Task has now captured this exception and recorded it to the database. This can be seen in the console as shown here:

   ```
   Updating: TaskExecution with executionId=1 with the following {exitCode=1, endTime=Wed Jul 25 12:42:15 EDT 2018,
   exitMessage='null', errorMessage='java.lang.IllegalStateException: Failed to execute CommandLineRunner
   ...
   ```

3. Remove or comment out the Exception so that the next lessons work correctly.

45.5. Adding Pre- and Post-processing

Spring Cloud Task includes the ability to run additional processing both before and after the task. To add both features to our current sample application:

1. Open `src/main/java/io/spring/TimestampApplication.java` and add the following code to the TimestampCommandLineRunner:

   ```
   @BeforeTask
   public void beforeTask(TaskExecution taskExecution) {
     System. out.println("Before TASK");
   }

   @AfterTask
   ```

```
public void afterTask(TaskExecution taskExecution) {
  System. out.println("After TASK");
}
```

2. From the command line, run `./mvnw clean spring-boot:run`.

   Now the output includes lines that print both `BEFORE TASK` and `AFTER TASK`.

45.6. Adding a MySQL Database

Nearly always, a real-world Spring Cloud Task needs to use a persistent (rather than an in-memory) database. In this example, we show how to add a MySQL database (MariaDB) to our Task. To do so:

1. Open the `pom.xml` file.
2. Add the following dependency:

```
<dependency>
  <groupId>org.mariadb.jdbc</groupId>
  <artifactId>mariadb-java-client</artifactId>
</dependency>
```

3. From your command line set up the database connection properties for MySql for example

```
export spring_datasource_url=jdbc:mysql://localhost:3306/practice
export spring_datasource_username=root
export spring_datasource_password=password
export spring_datasource_driverClassName=org.mariadb.jdbc.Driver
```

4. From the command line, run `./mvnw clean spring-boot:run`.

   If you examine the contents of your database, you should now see the task in the `TASK_EXECUTION` table.

---

46. Adding and Launching Spring Cloud Tasks with Data Flow

This guide walks through registering and launching a Spring Cloud Task application. It consists of the following procedures:

1. Registering and Launching Your First Task
2. Registering and Launching Your First Spring Batch-Task

46.1. Registering and Launching Your First Task

Once you have Spring Cloud Data Flow Server and Shell running, you can use the following procedure to create your first task:

1. Register a basic suite of tasks by importing their registrations through the Spring Cloud Data Flow Shell with the following command:

   `app register --name timestamp --type task --uri maven://org.springframework.cloud.task.app:timestamp-task:1.3.0.RELEASE`

   This example shows how to register a task from a Maven repository.

2. Verify that the timestamp-task app registered by running the following command in the Spring Cloud Data Flow Shell:

   `app list`

   The following output should appear:

3. Create a task definition that uses timestamp task by using the following command in the Spring Cloud Data Flow Shell:

   `task create --name myStamp --definition "timestamp"`

   You should see a message saying "Created new task 'myStamp'".

4. Launch your new task by using the following command:

   `task launch myStamp`

   You should see a message saying "Launched task `myStamp`".

5. Verify that your task was successfully run by running the following command in the Spring Cloud Data Flow Shell: `task execution list`

   You should see output similar to the following:

The exit code of 0 tells us that the task ran without errors.

46.2. Registering and Launching Your First Spring Batch-Task

Essentially, a Batch-Task is a Spring Batch application that includes the `@EnableTask` annotation, which serves as an indicator that the Spring Batch application uses Spring Cloud Task. Spring Boot takes care of the rest of the set up for us.

To register your first Spring Batch Task:

1. In Spring Cloud Data Flow Shell, register a Spring Batch Task application by using the following command:

   ```
   app register --name batch-events --type task --uri
   maven://org.springframework.cloud.task.app:timestamp-batch-task:2.0.0.RELEASE
   ```

2. To verify that your application has been registered, run the following command in the Spring Cloud Data Flow Shell:

   ```
   app list
   ```

   You should see output similar to the following:

   ```
   ```

3. Create a task definition that uses the batch-events task, by running the following command:

   ```
   task create --name myBatchTask --definition "batch-events"
   ```

   You should see a message saying "Created new task 'myBatchTask'".

4. Launch your batch-task by running the following command:

   ```
   task launch myBatchTask
   ```

   You should see a message saying "Launched task `myBatchTask`".

5. Verify that the task ran, run the following command:

   ```
   task execution list
   ```

   You should see output similar to the following:

   ```
   ```

   We can now verify that the task worked as a batch job. Thenext section describes how to do so.

46.2.1. Verifying that Your Task is a Batch

When you create and run a Batch-Task, it is both a Spring Cloud Task instance and a Spring Batch instance. In the previous section, we saw how to verify that your first batch-task worked as a task. This section steps through how to verify that it also worked as a batch. To do so:

1. Run the following command to see the list of jobs that have run:

   ```
   job execution list
   ```

   You should see output similar to the following:

   ```
   ```

2. Note the Job ID from the ID column (in this case, we want to look at 2).

3. To get the details of the job execution, we can use the Job ID in the following command:

   ```
   job execution display --id 1
   ```

   You should see output similar to the following:

   ```
   ```

47. Database Requirement for running tasks in Spring Cloud Data Flow

As previously discussed Spring Cloud Task records the state of each task execution to a relational database. And as such Spring Cloud Data Flow uses this recorded information when users request task or batch job execution information. Also Spring Batch and Spring Cloud Task offer features that allow Spring Cloud Data Flow to communicate certain start or stop behaviors. One example is when a user utilizes the Spring Cloud Data Flow UI to stop a Spring Batch app execution.

Dashboard

This section describes how to use the dashboard of Spring Cloud Data Flow.

48. Introduction

Spring Cloud Data Flow provides a browser-based GUI called the dashboard to manage the following information:

- **Apps**: The Apps tab lists all available applications and provides the controls to register/unregister them.
- **Runtime**: The Runtime tab provides the list of all running applications.
- **Streams**: The Streams tab lets you list, design, create, deploy, and destroy Stream Definitions.
- **Tasks**: The Tasks tab lets you list, create, launch, schedule and, destroy Task Definitions.
- **Jobs**: The Jobs tab lets you perform batch job related functions.
- **Analytics**: The Analytics tab lets you create data visualizations for the various analytics applications.

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

`<host>:<port>/dashboard`

For example, if Spring Cloud Data Flow is running locally, the dashboard is available at `http://localhost:9393/dashboard`.

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

The default Dashboard server port is `9393`.

The following image shows the opening page of the Spring Cloud Data Flow dashboard:

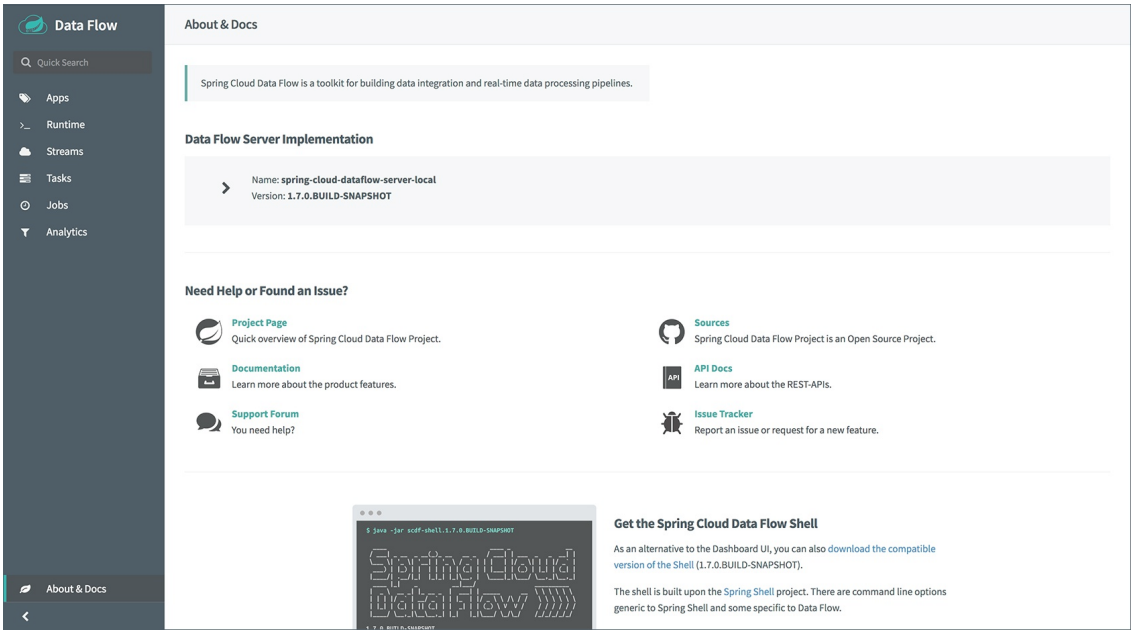


Figure 18. The Spring Cloud Data Flow Dashboard

49. Apps

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

The following image shows a typical list of available apps within the dashboard:

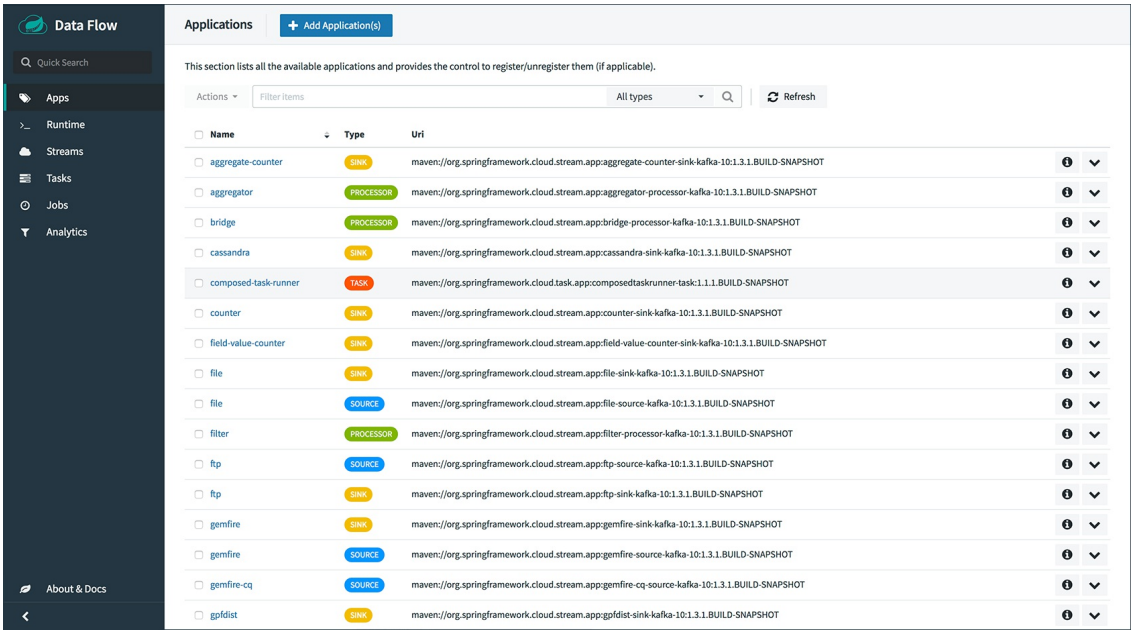


Figure 19. List of Available Applications

49.1. Bulk Import of Applications

The Bulk Import Applications page provides numerous options for defining and importing a set of applications all at once. For bulk import, the application definitions are expected to be expressed in a properties style, as follows:

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

The following examples show a typical application definitions:

```
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, a URI can be specified that points to a properties file stored elsewhere, it should contain properties formatted as shown in the previous example. Alternatively, by using the textbox labeled “Apps as Properties”, you can directly list each property string. Finally, if the properties are stored in a local file, the “Select Properties File” option opens a local file browser to select the file. After setting your definitions through one of these routes, click Import.

The following image shows the Bulk Import Applications page:

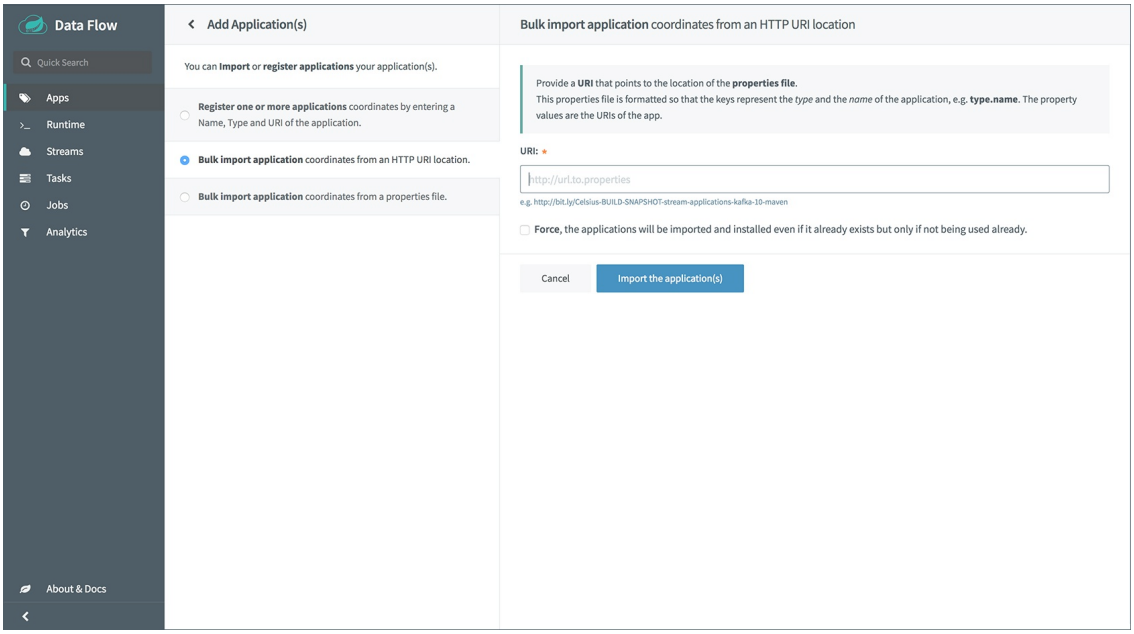


Figure 20. Bulk Import Applications

50. Runtime

The Runtime section of the Dashboard application shows 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 following image shows an example of the Runtime tab in use:

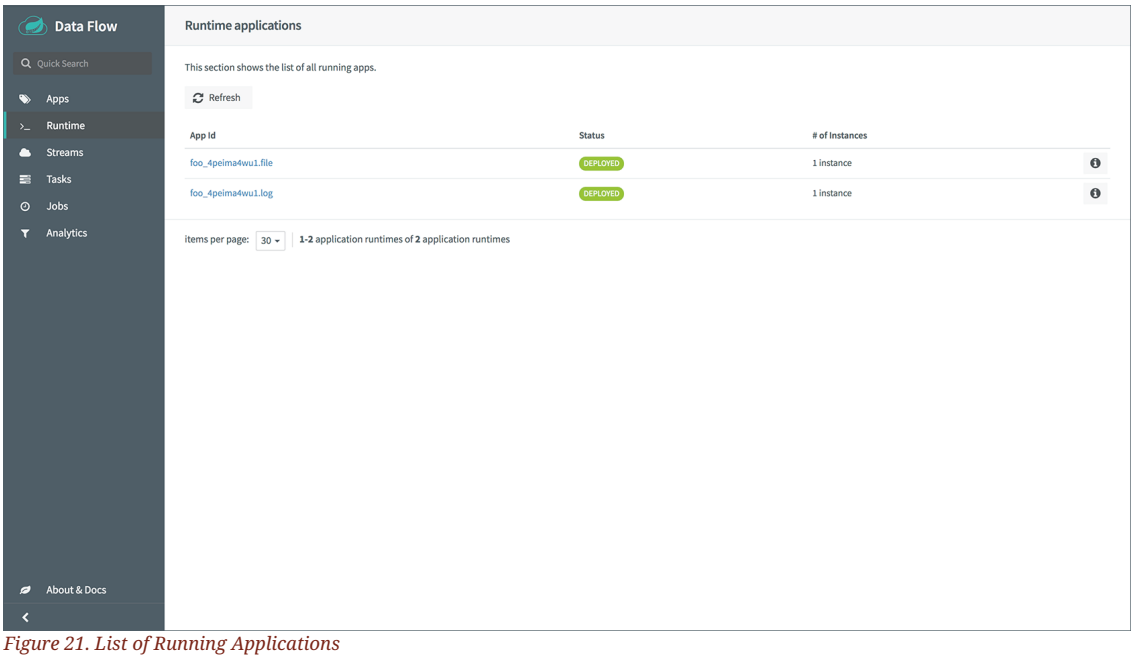


Figure 21. List of Running Applications

51. Streams

The Streams tab has two child tabs: Definitions and Create Stream. The following topics describe how to work with each one:

- Working with Stream Definitions
- Creating a Stream
- Deploying a Stream

51.1. Working with Stream Definitions

The Streams section of the Dashboard includes 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 you can click to see a visual representation of the definition. Hovering over the boxes in the visual representation shows more details about the apps, including any options passed to them.

In the following screenshot, the `timer` stream has been expanded to show the visual representation:

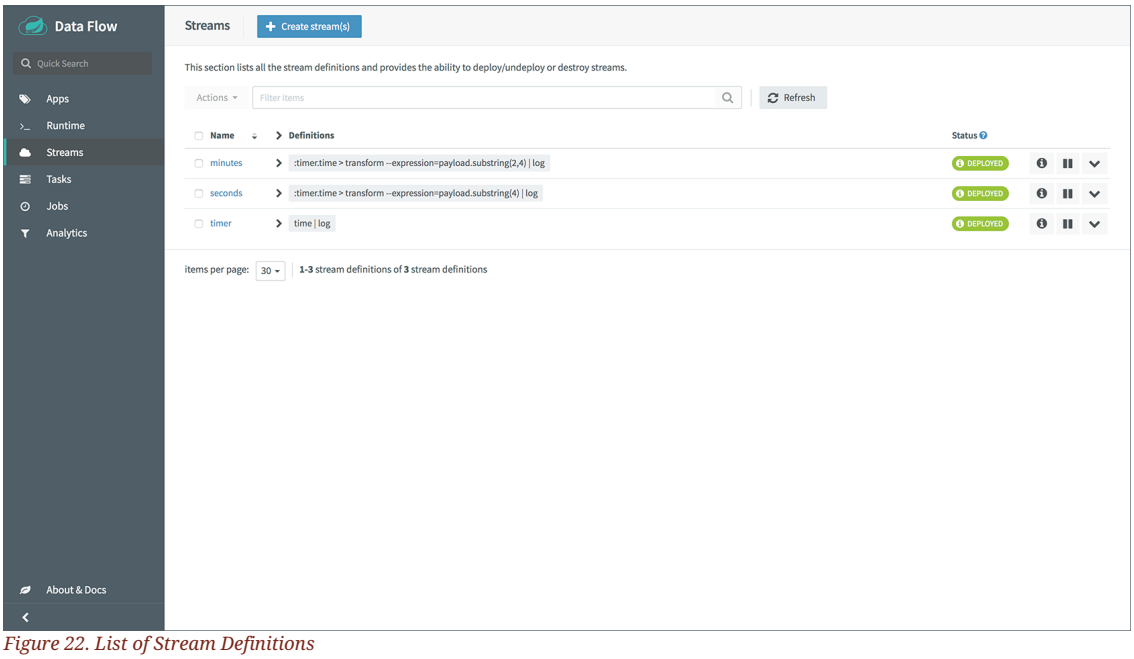


Figure 22. List of Stream Definitions

If you click the details button, the view changes to show a visual representation of that stream and any related

streams. In the preceding example, if you click details for the `timer` stream, the view changes to the following view, which clearly shows the relationship between the three streams (two of them are tapping into the `timer` stream):

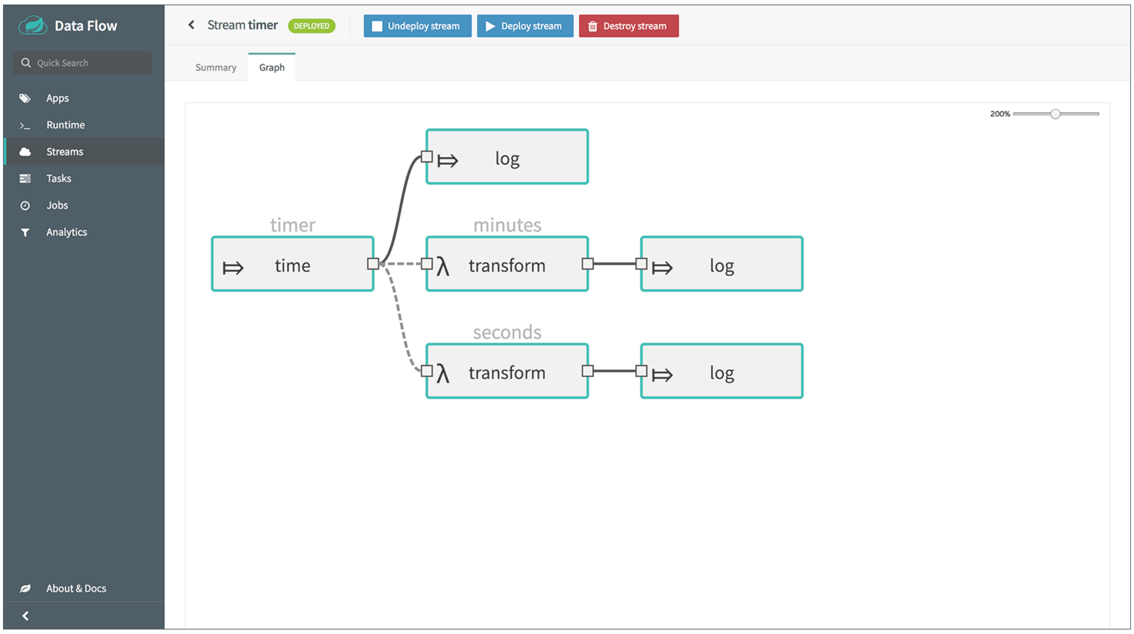


Figure 23. Stream Details Page

51.2. Creating a Stream

The Streams section of the Dashboard includes the Create Stream tab, which makes available the [Spring Flo] designer: a canvas application that offers 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

You should watch this screencast that highlights some of the "Flo for Spring Cloud Data Flow" capabilities. The Spring Flo wiki includes more detailed content on core Flo capabilities.

The following image shows the Flo designer in use:

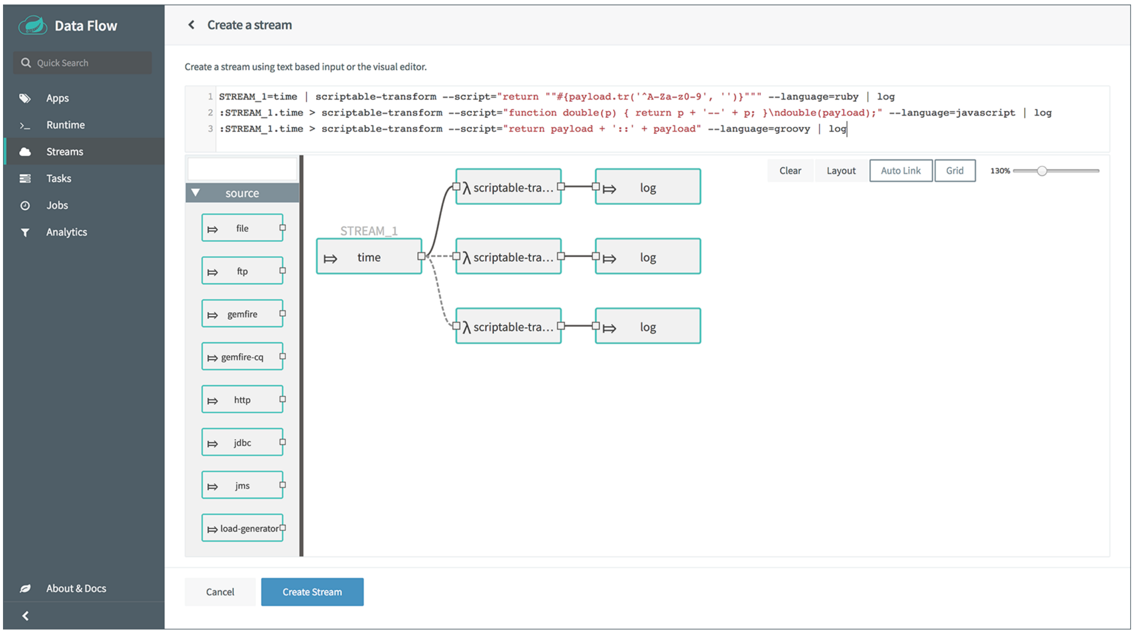


Figure 24. Flo for Spring Cloud Data Flow

51.3. Deploying a Stream

The stream deploy page includes tabs that provide different ways to setup the deployment properties and deploy the stream. The following screenshots show the stream deploy page for `foobar` (`time | log`).

You can define deployments properties using:

- Form builder tab: a builder which help you to define deployment properties (deployer, application properties…)
- Free text tab: a free textarea (key/value pairs)

You can switch between the both views, the form builder provides a more stronger validation of the inputs.

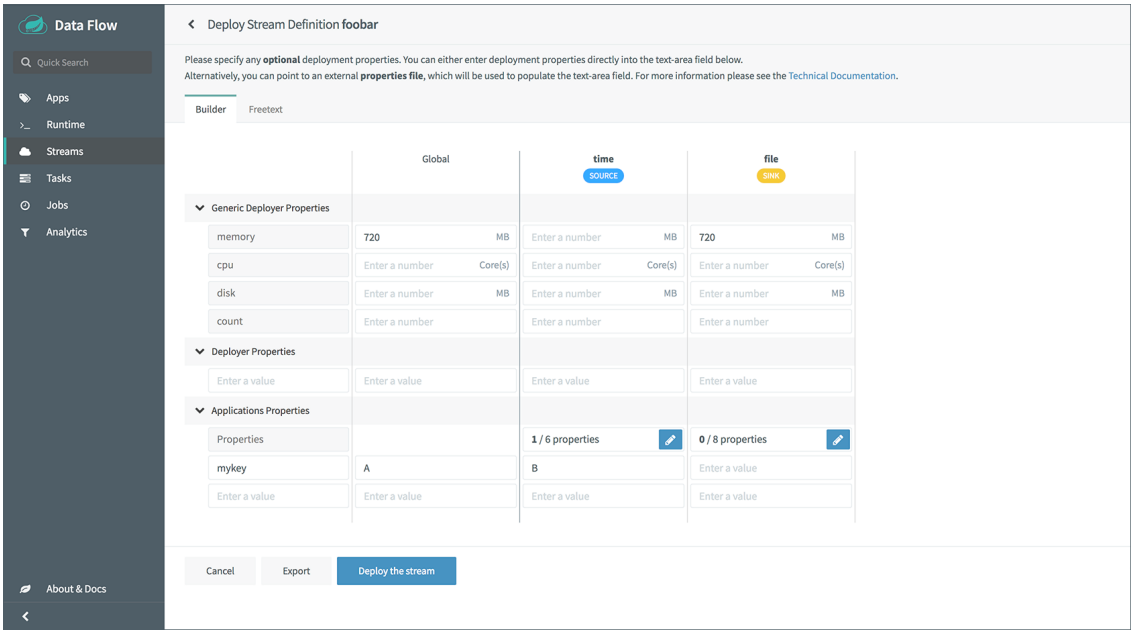


Figure 25. The following image shows the form builder

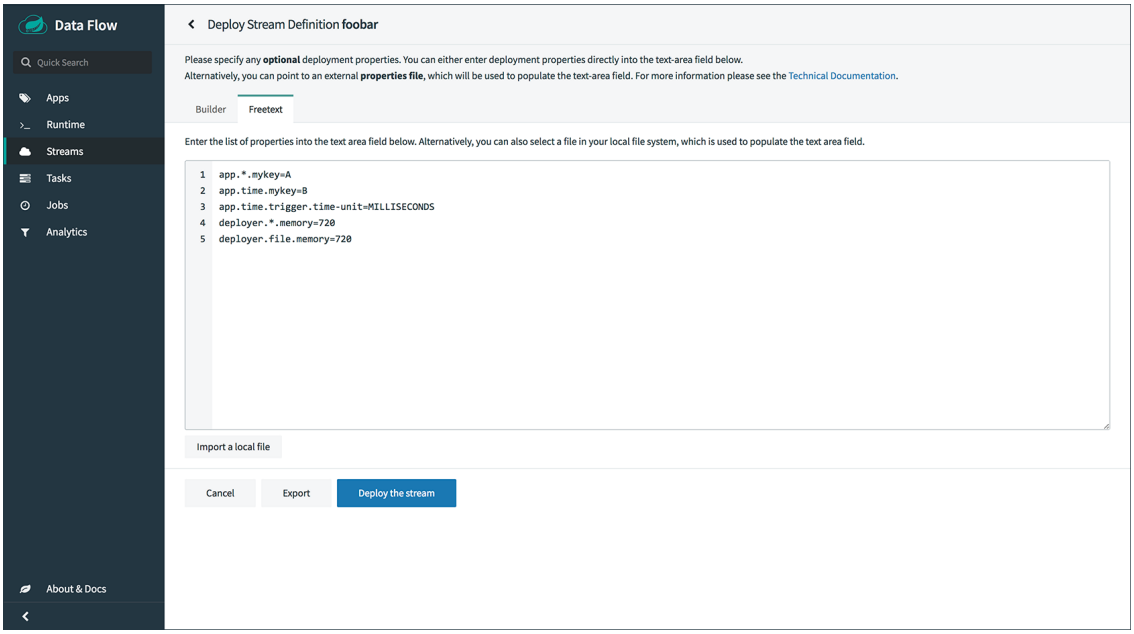


Figure 26. The following image shows the same properties in the free text

51.4. Creating Fan-In/Fan-Out Streams

In chapter Fan-in and Fan-out you learned how we can support fan-in and fan-out use cases usingnamed destinations. The UI provides dedicated support for named destinations as well:

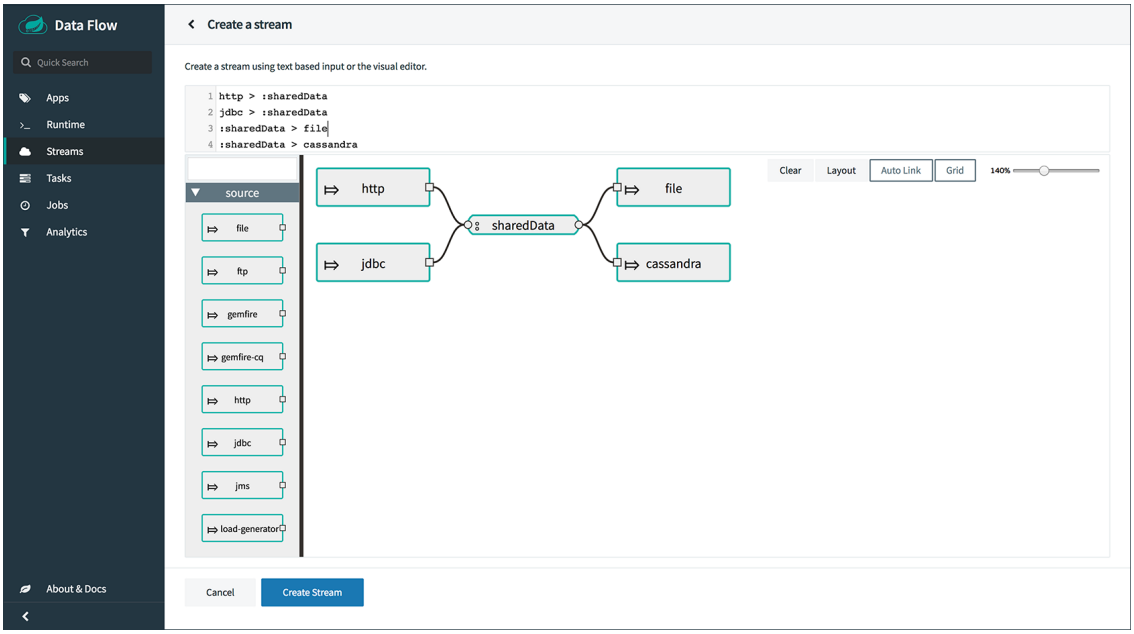


Figure 27. Flo for Spring Cloud Data Flow

In this example we have data from an *HTTP Source* and a *JDBC Source* that is being sent to the *sharedData* channel which represents a **Fan-in** use case. On the other end we have a *Cassandra Sink* and a *File Sink* subscribed to the *sharedData* channel which represents a **Fan-out** use case.

51.5. Creating a Tap Stream

Creating Taps using the Dashboard is straightforward. Let's say you have stream consisting of an *HTTP Source* and a *File Sink* and you would like to tap into the stream to also send data to a *JDBC Sink*. In order to create the tap stream simply connect the output connector of the *HTTP Source* to the *JDBC Sink*. The connection will be displayed as a dotted line, indicating that you created a tap stream.

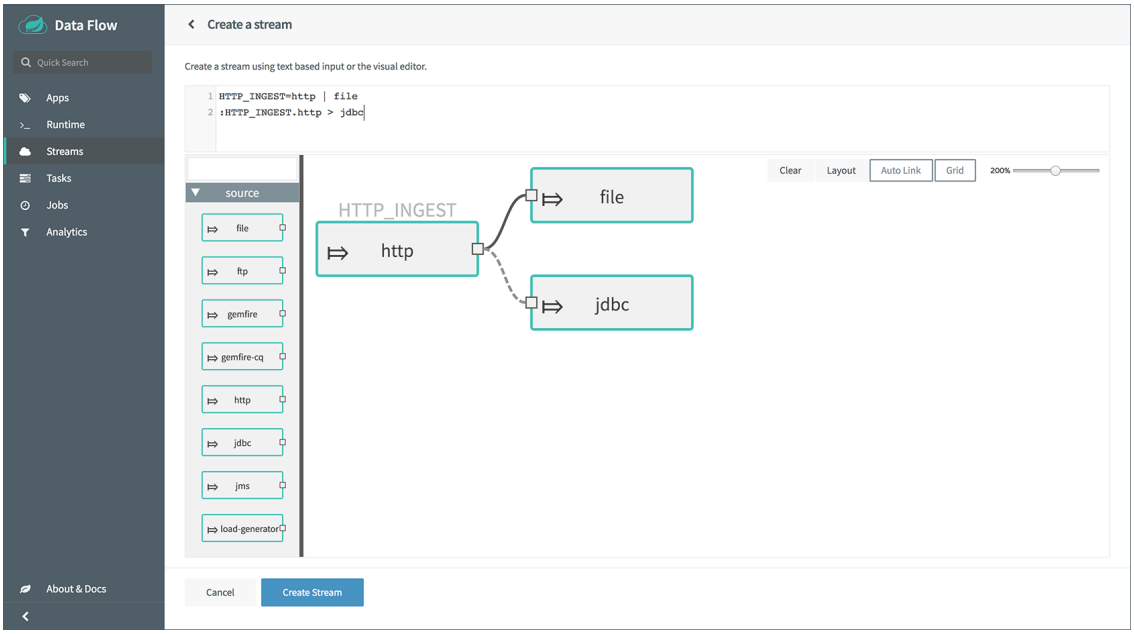


Figure 28. Creating a Tap Stream

The primary stream (*HTTP Source* to *File Sink*) will be automatically named, in case you did not provide a name for the stream, yet. When creating tap streams, the primary stream must always be explicitly named. In the picture above, the primary stream was named *HTTP_INGEST*.

Using the Dashboard, you can also switch the primary stream to become the secondary tap stream.

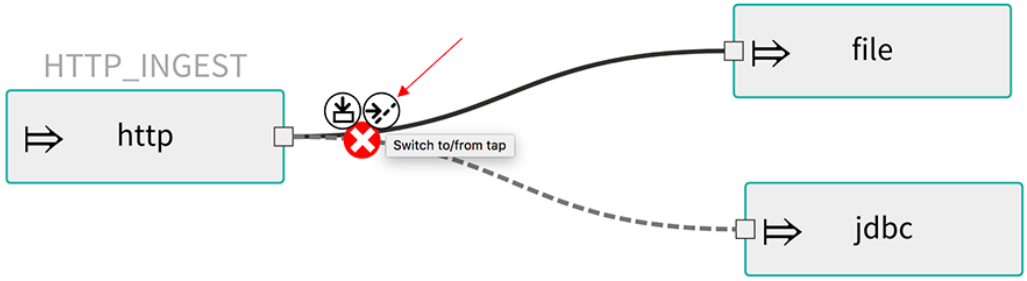


Figure 29. Change Primary Stream to Secondary Tap Stream

Simply hover over the existing primary stream, the line between *HTTP Source* and *File Sink*. Several control icons will appear, and by clicking on the icon labeled *Switch to/from tap*, you change the primary stream into a tap stream. Do the same for the tap stream and switch it to a primary stream.

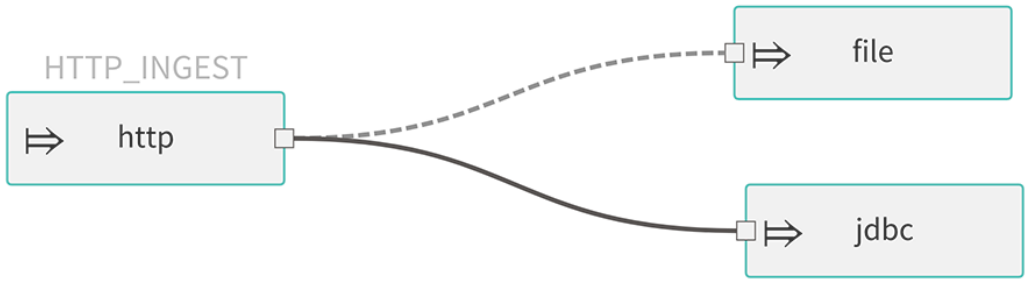


Figure 30. End Result of Switching the Primary Stream

When interacting directly with named destinations, there can be "n" combinations (Inputs/Outputs). This allows you to create complex topologies involving a wide variety of data sources and destinations.

52. Tasks

The Tasks section of the Dashboard currently has three tabs:

- Apps
- Definitions
- Executions
- Scheduling

52.1. Apps

Each app encapsulates a unit of work into a reusable component. Within the Data Flow runtime environment, apps let users create definitions for streams as well as tasks. Consequently, the Apps tab within the Tasks section lets users create task definitions.

You can also use this tab to create Batch Jobs.

The following image shows a typical list of task apps:

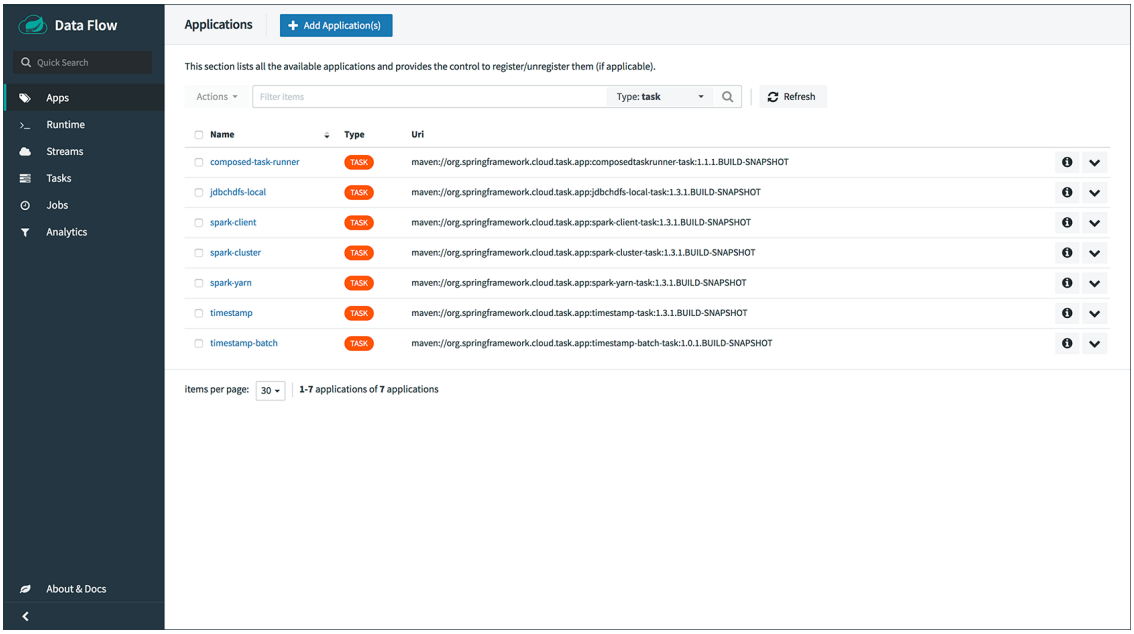


Figure 31. List of Task Apps

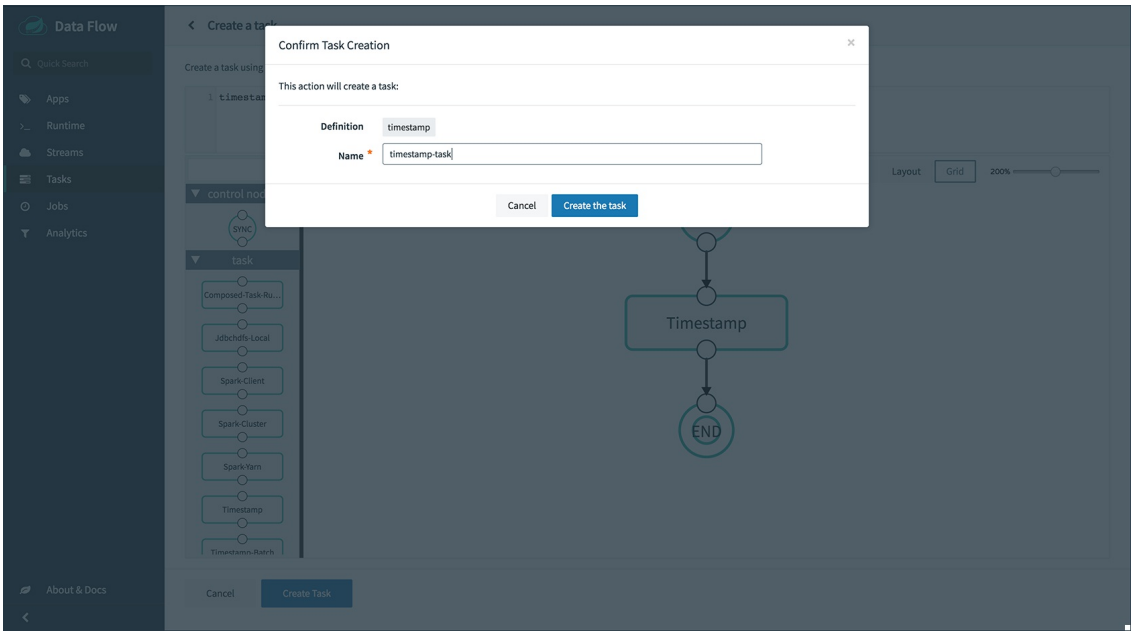
On this screen, you can perform the following actions:

- View details, such as the task app options.
- Create a task definition from the respective app.

52.1.1. 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.

52.1.2. Create a Task Definition



At a minimum, you must provide a name for the new definition. You also have the option to specify various properties that are used during the deployment of the app.

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

52.2. 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 with simple textual input, indicated by the Bulk Define Tasks button.

The following image shows the Definitions page:

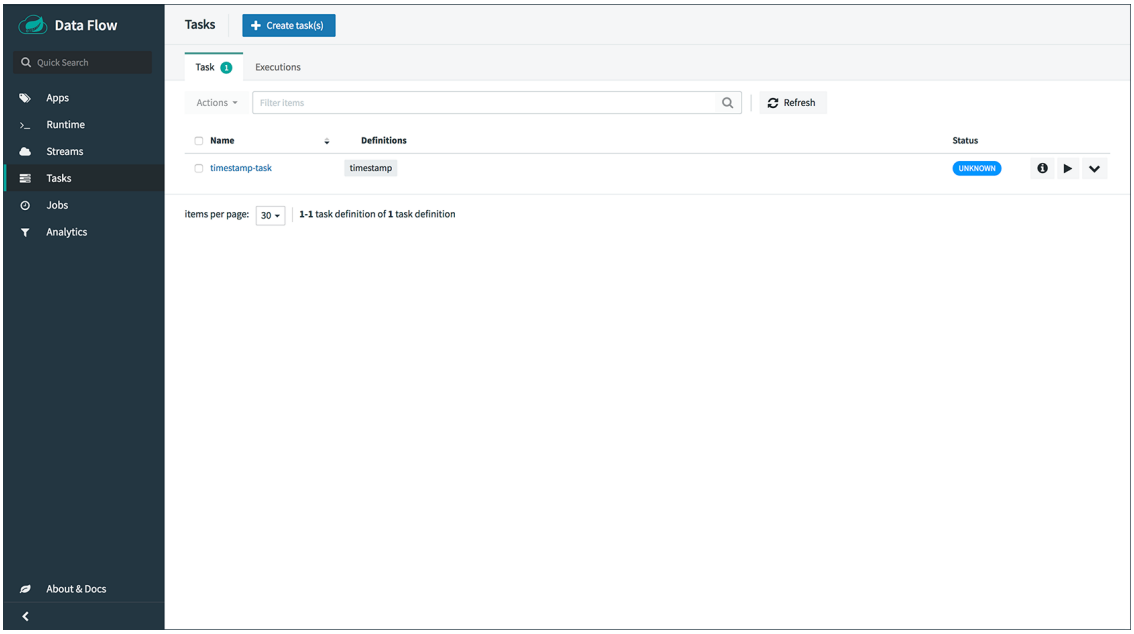


Figure 32. List of Task Definitions

52.2.1. Creating Composed Task Definitions

The dashboard includes the Create Composed Task tab, which provides an 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.

On the Create Composed Task screen, you can define one or more task parameters by entering both the parameter key and the parameter value.

Task parameters are not typed.

The following image shows the composed task designer:

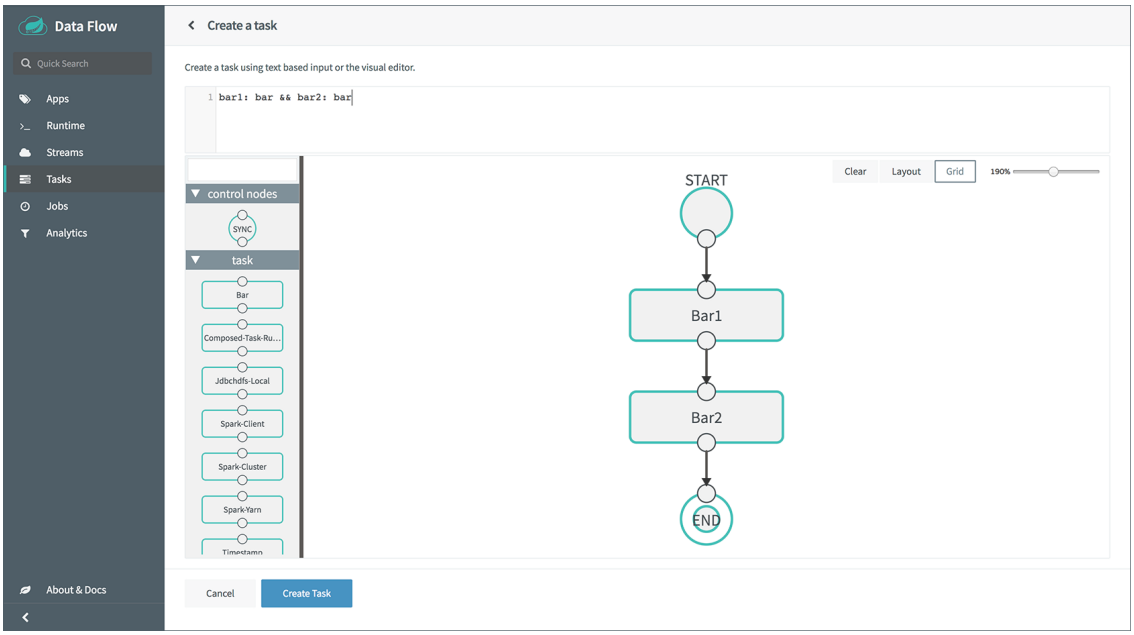


Figure 33. Composed Task Designer

52.2.2. Launching Tasks

Once the task definition has been created, the tasks can be launched through the dashboard. To do so, click the Definitions tab and select the task you want to launch by pressing `Launch`.

52.3. Executions

The Executions tab shows the current running and completed tasks.

The following image shows the Executions tab:

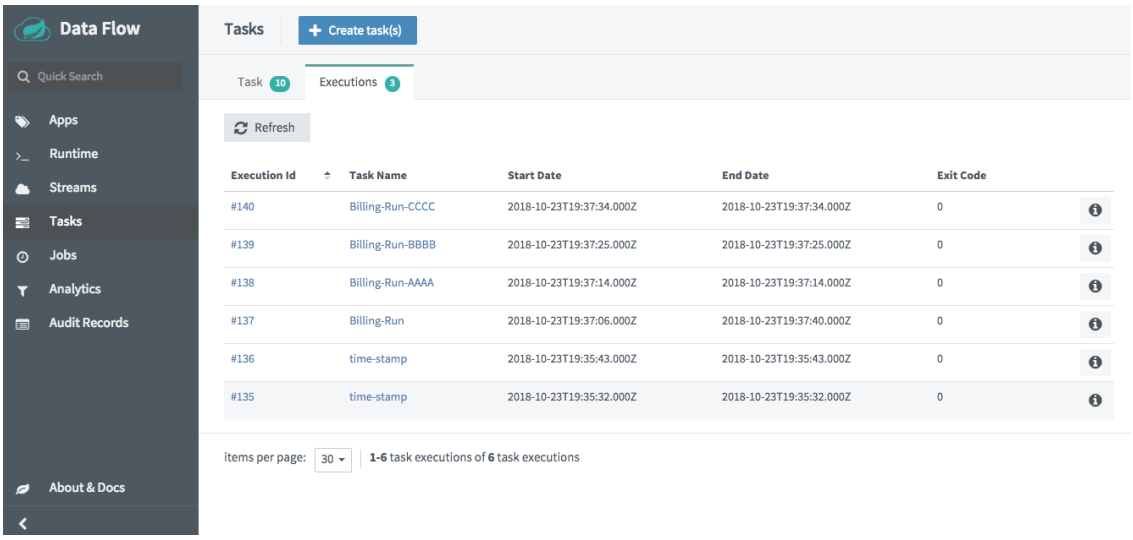
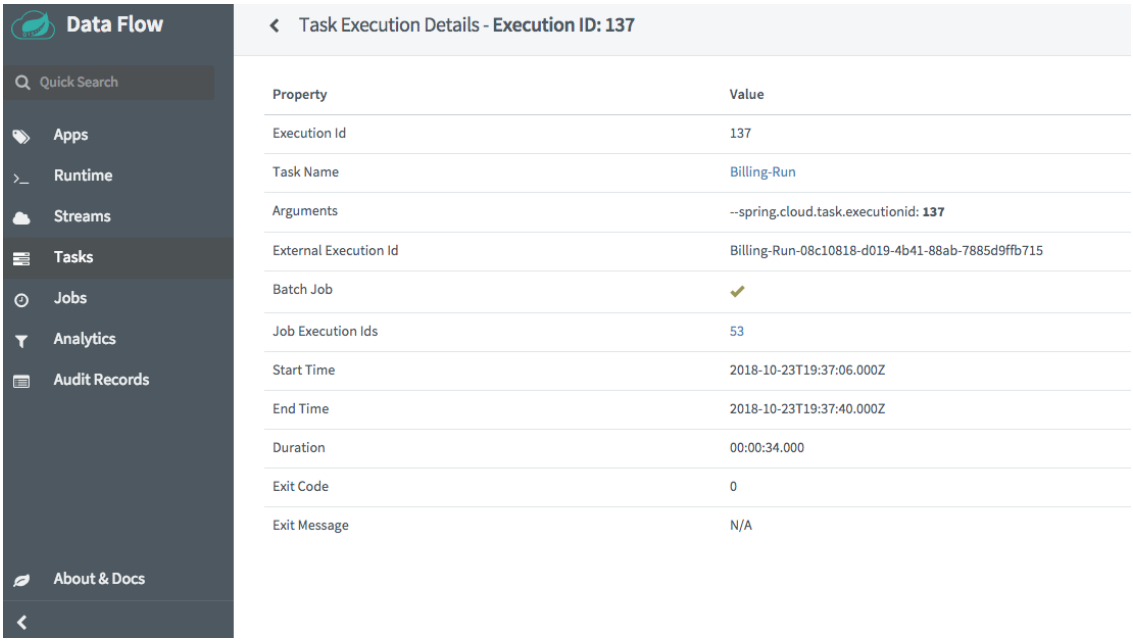


Figure 34. List of Task Executions

52.4. Execution Detail

For each task execution on the Executions page, a user can retrieve detailed information about a specific execution by clicking the information icon located to the right of the task execution.



On this screen the user can view not only the information from the Task Executions page but also:

- Task Arguments
- External Execution Id
- Batch Job Indicator (indicates if the task execution contained Spring Batch jobs.)
- Job Execution Ids links (Clicking the Job Execution Id will take you to theJob Execution Details for that Job Execution Id.)
- Task Execution Duration
- Task Execution Exit Message

53. Jobs

The Jobs section of the Dashboard lets you inspect batch jobs. The main section of the screen provides a list of job executions. Batch jobs are tasks that each execute one or more batch jobs. Each job execution has a reference to the task execution ID (in the Task Id column).

The list of Job Executions also shows the state of the underlying Job Definition. Thus, if the underlying definition has been deleted, “No definition found” appears in the Status column.

You can take the following actions for each job:

- Restart (for failed jobs).
- Stop (for running jobs).
- View execution details.

Note: Clicking the stop button actually sends a stop request to the running job, which may not immediately stop.

The following image shows the Jobs page:

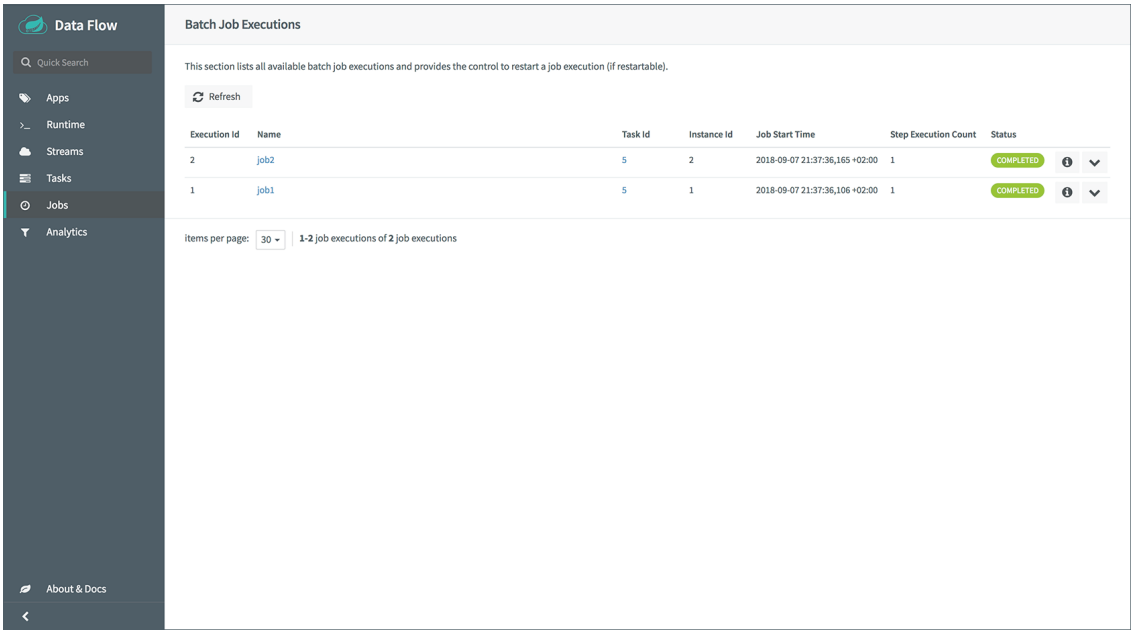


Figure 35. List of Job Executions

53.1. Job Execution Details

After having launched a batch job, the Job Execution Details page will show information about the job.

The following image shows the Job Execution Details page:

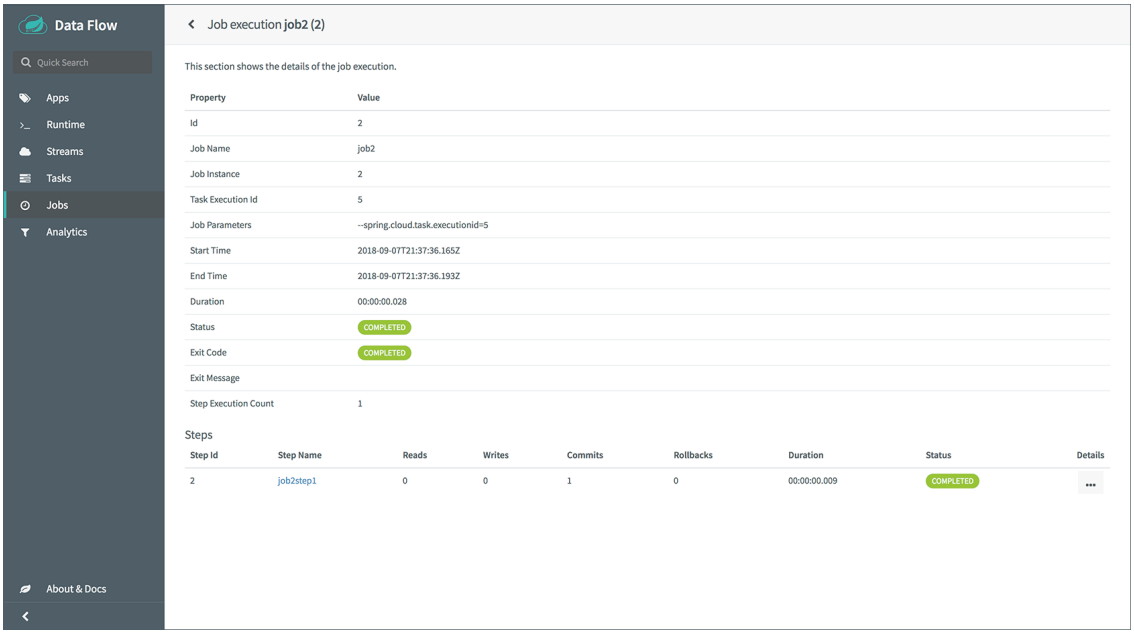


Figure 36. Job Execution Details

The Job Execution Details page contains a list of the executed steps. You can further drill into the details of each step's execution by clicking the magnifying glass icon.

53.2. Step Execution Details

The Step Execution Details page provides information about an individual step within a job.

The following image shows the Step Execution Details page:

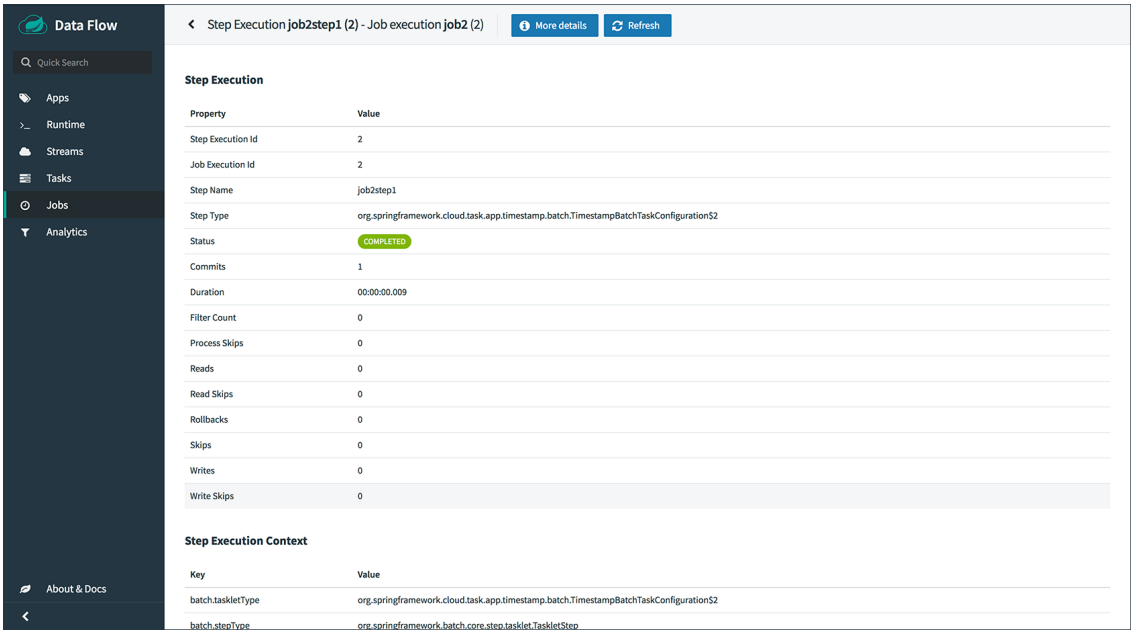


Figure 37. Step Execution Details

On the top of the page, you can see a progress indicator the respective step, with the option to refresh the indicator. 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.

For exceptions, the Exit Description field contains additional error information. However, this field can have a maximum of 2500 characters. Therefore, in the case of long exception stack traces, trimming of error messages may occur. When that happens, refer to the server log files for further details.

53.3. 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, and others.

54. Scheduling

54.1. Creating or deleting a Schedule from the Task Definition's page

From the Task Definitions page a user can create or delete a schedule for a specific task definition.

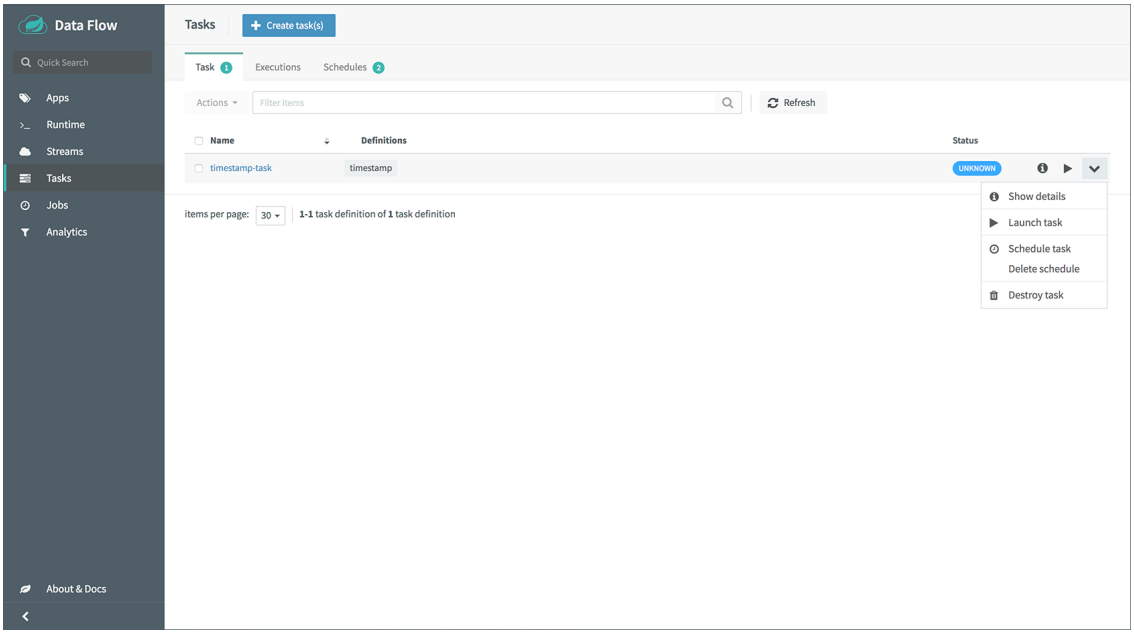


Figure 38. Task Definitions with Schedule controls

On this screen you can perform the following actions:

- The user can click the clock icon and this will take you to the Schedule Creation screen.

- The user can click the clock icon with the `x` to the upper right to delete the schedule(s) associated with the task definition.

54.2. Creating a Schedule

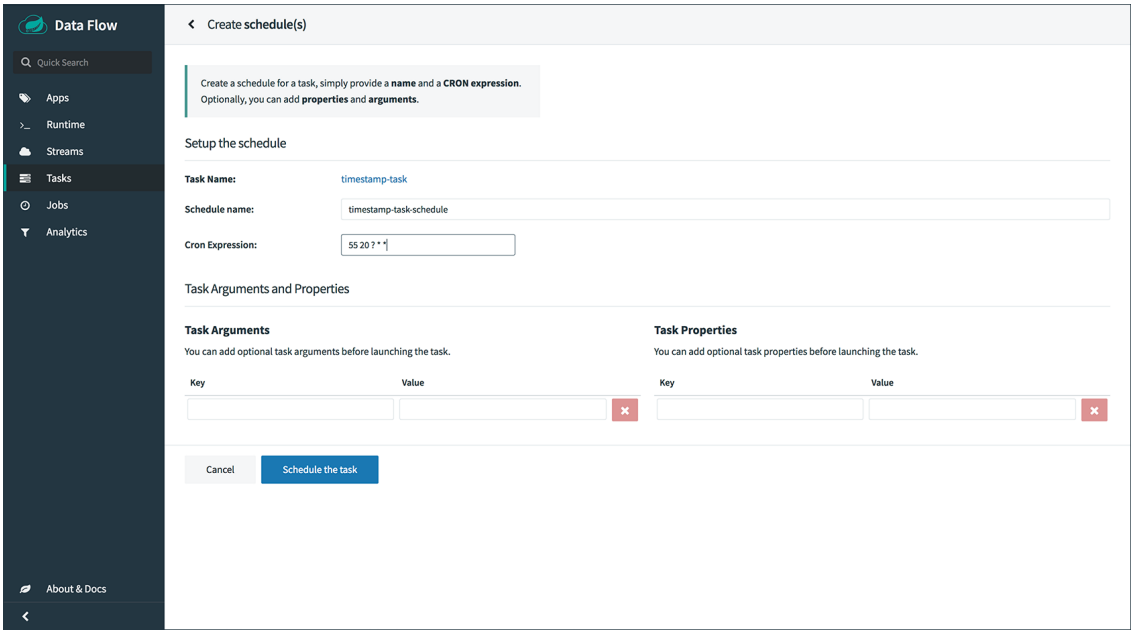


Figure 39. Create Schedule for Task Execution

Once the user clicks the clock icon on the Task Definition screen, Spring Cloud Data Flow will take the user to the Schedule Creation screen. On this screen a user can establish the schedule name, the cron expression as well as establish the properties and arguments to be used when the task is launched by this schedule.

54.3. Listing Available Schedules

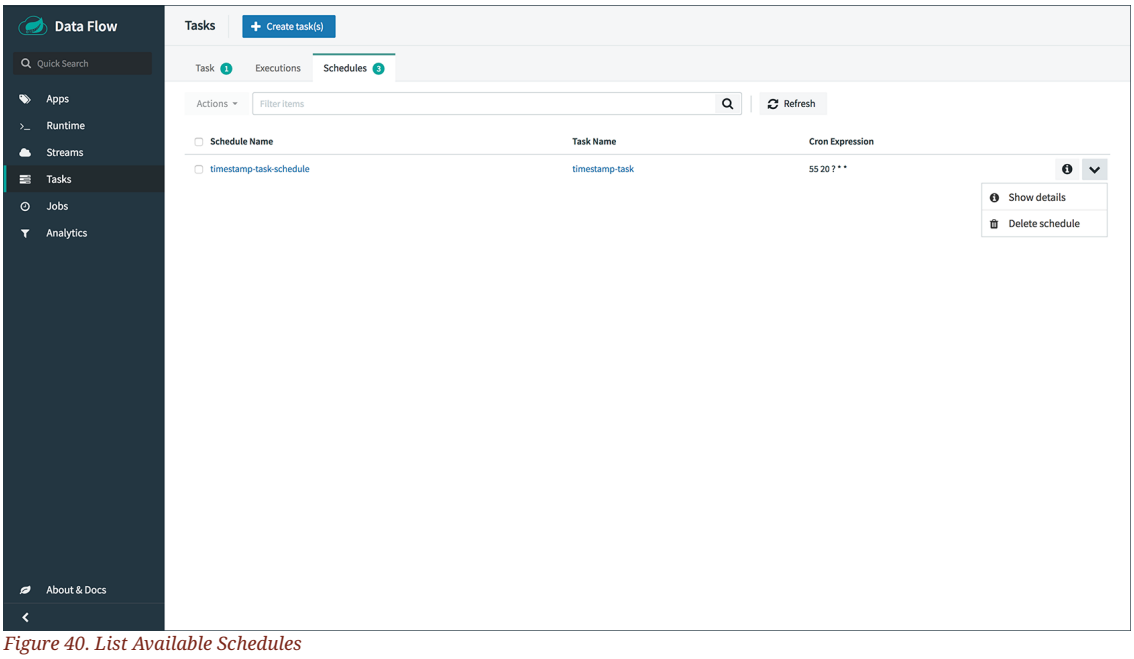


Figure 40. List Available Schedules

On this screen you can perform the following actions:

- Delete a schedule
- Get details for a schedule

55. Analytics

The Analytics page of the Dashboard provides the following 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 create the corresponding graph from within the Dashboard tab. To do so:

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

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

---

56. Auditing

The Auditing page of the Dashboard gives you access to recorded audit events. Currently audit event are recorded for:

- Streams
  - Create
  - Delete
  - Deploy
  - Undeploy
- Tasks
  - Create
  - Delete
  - Launch
- Scheduling of Tasks
  - Create Schedule
  - Delete Schedule

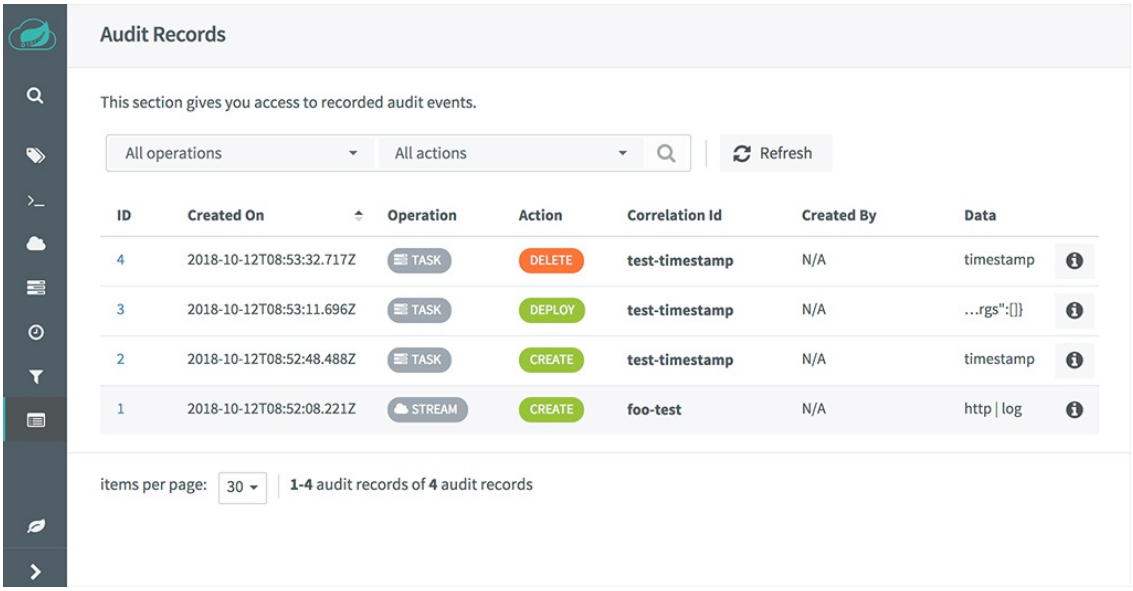


Figure 41. List Overview of Audit Records

By clicking on the *Show Details* icon, you can obtain further details regarding the auditing details:

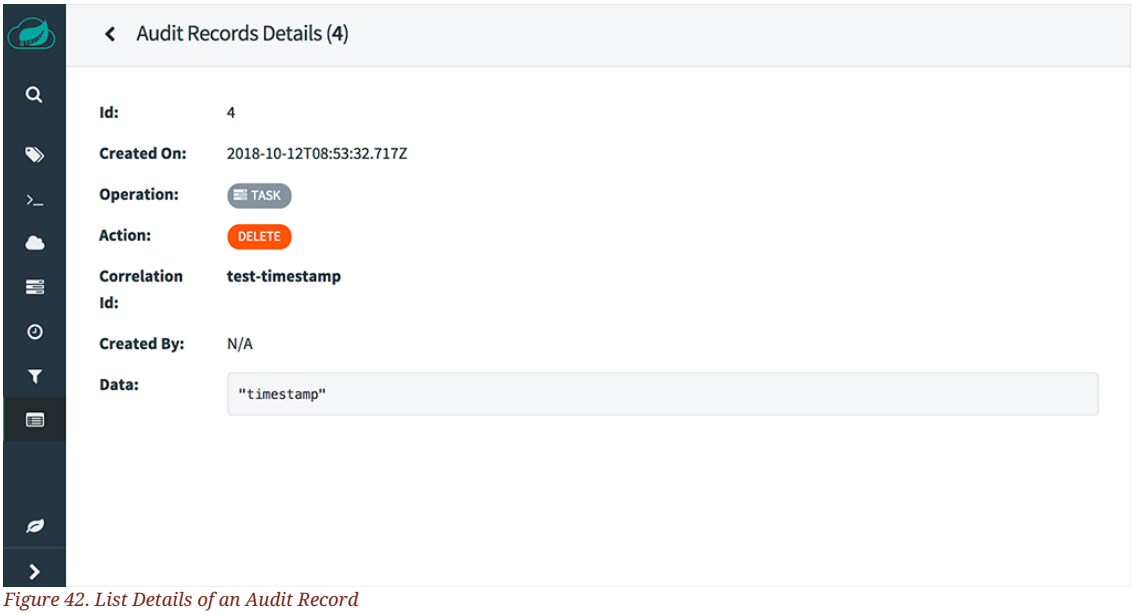


Figure 42. List Details of an Audit Record

Generally, auditing provides the following information:

- When was the record created?
- Username that triggered the audit event (if security is enabled)
- Audit operation (Schedule, Stresm, Task)
- Performed action (Create, Delete, Deploy, Rollback, Undeploy, Update)
- Correlation Id, e.g. Stream/Task name
- Audit Data

The written value of the property *Audit Data* depends on the performed *Audit Operation* and the *ActionType*. For example, when a Schedule is being created, the name of the task definition, task definition properties, deployment properties, as well as command line arguments are written to the persistence store.

Sensitive information is sanitized prior to saving the Audit Record, in an *best-effort-manner*. Any of the following keys are being detected and its sensitive values are masked:

- password
- secret
- key
- token
- .*credentials.*
- vcap_services

Samples

This section shows the available samples.

57. Links

Several samples have been created to help you get started on implementing higher-level use cases than the basic Streams and Tasks shown in the reference guide. The samples are part of a separate repository and have their own reference documentation.

- Java DSL
- HTTP to Cassandra
- HTTP to MySQL
- HTTP to Gemfire
- Gemfire CQ to Log Demo
- Gemfire to Log Demo
- Custom Processor

Task and Batch

- Batch Job on Cloud Foundry
- Batch File Ingest

Analytics

- Twitter Analytics

Data Science

- Species Prediction

Functions

- Using Spring Cloud Function

REST API Guide

This section describes the Spring Cloud Data Flow REST API.

58. Overview

Spring Cloud Data Flow provides a REST API that lets you access all aspects of the server. In fact, the Spring Cloud Data Flow shell is a first-class consumer of that API.

If you plan to use the REST API with Java, you should consider using the provided Java client (`DataflowTemplate`) that uses the REST API internally.

58.1. HTTP verbs

Spring Cloud Data Flow tries to adhere as closely as possible to standard HTTP and REST conventions in its use of HTTP verbs, as described in the following table:

Verb	Usage
`GET`	Used to retrieve a resource
`POST`	Used to create a new resource
`PUT`	Used to update an existing resource, including partial updates. Also used for resources that imply the concept of `restarts`, such as Tasks.
`DELETE`	Used to delete an existing resource.

58.2. HTTP Status Codes

Spring Cloud Data Flow tries to adhere as closely as possible to standard HTTP and REST conventions in its use of HTTP status codes, as shown in the following table:

Status code	Usage
`200 OK`	The request completed successfully.
`201 Created`	A new resource has been created successfully. The resource's URI is available from the response's `Location` header.
`204 No Content`	An update to an existing resource has been applied successfully.
`400 Bad Request`	The request was malformed. The response body includes an error description that provides further information.
`404 Not Found`	The requested resource did not exist.
`409 Conflict`	The requested resource already exists. For example, the task already exists or the stream was already being deployed
`422 Unprocessable Entity`	Returned in cases where the Job Execution cannot be stopped or restarted.

Status code	Usage

58.3. Headers

Every response has the following header(s):

Name	Description
`Content-Type`	The Content-Type of the payload, e.g. `application/hal+json`

58.4. Errors

Path	Type	Description
`error`	`String`	The HTTP error that occurred, e.g. `Bad Request`
`message`	`String`	A description of the cause of the error
`path`	`String`	The path to which the request was made
`status`	`Number`	The HTTP status code, e.g. `400`
`timestamp`	`String`	The time, in milliseconds, at which the error occurred

58.5. Hypermedia

Spring Cloud Data Flow uses hypermedia, and resources include links to other resources in their responses. Responses are in the Hypertext Application from resource to resource Language (HAL) format. Links can be found beneath the `_links` key. Users of the API should not create URIs themselves. Instead. they should use the above-described links to navigate.

59. Resources

The API includes the following resources:

- Index
- Server Meta Information
- Audit Records
- Registered Applications
- Stream Definitions
- Stream Deployments
- Stream Validation
- Task Definitions
- Task Executions
- Task Scheduler
- Task Validation
- Job Executions
- Job Instances
- Job Step Executions
- Runtime Information about Applications
- Metrics for Stream Applications

59.1. Index

The index provides the entry point into Spring Cloud Data Flow's REST API. The following topics provide more detail:

- Accessing the index
- Request Structure
- Example Request
- Response Structure

- Example Response
- Links

59.1.1. Accessing the index

Use a `GET` request to access the index.

Request Structure

```
GET / HTTP/1.1
Host: localhost:9393
```

Example Request

```
$ curl 'http://localhost:9393/' -i -X GET
```

Response Structure

Path	Type	Description
`_links`	`Object`	Links to other resources
`['api.revision']`	`Number`	Incremented each time a change is implemented in this REST API
`_links.audit-records.href`	`String`	Link to the audit records
`_links.dashboard.href`	`String`	Link to the dashboard
`_links.streams/definitions.href`	`String`	Link to the streams/definitions
`_links.streams/definitions/definition.href`	`String`	Link to the streams/definitions/definition
`_links.streams/definitions/definition.templated`	`Boolean`	Link streams/definitions/definition is templated
`_links.runtime/apps.href`	`String`	Link to the runtime/apps
`_links.runtime/apps/app.href`	`String`	Link to the runtime/apps/app
`_links.runtime/apps/app.templated`	`Boolean`	Link runtime/apps/app is templated
`_links.runtime/apps/instances.href`	`String`	Link to the runtime/apps/instances
`_links.runtime/apps/instances.templated`	`Boolean`	Link runtime/apps/instances is templated
`_links.metrics/streams.href`	`String`	Link to the metrics/streams
`_links.streams/deployments.href`	`String`	Link to the streams/deployments
`_links.streams/deployments/{name}.href`	`String`	Link to the streams/deployments/{name}
`_links.streams/deployments/{name}.templated`	`Boolean`	Link streams/deployments/{name} is templated
`_links.streams/deployments/deployment.href`	`String`	Link to the streams/deployments/deployment
`_links.streams/deployments/deployment.templated`	`Boolean`	Link streams/deployments/deployment is templated
`_links.streams/deployments/manifest/{name}/{version}.href`	`String`	Link to the streams/deployments/manifest/{name}/{version}
`_links.streams/deployments/manifest/{name}/{version}.templated`	`Boolean`	Link streams/deployments/manifest/{name}/{version} is templated
`_links.streams/deployments/history/{name}.href`	`String`	Link to the streams/deployments/history/{name}
`_links.streams/deployments/history/{name}.templated`	`Boolean`	Link streams/deployments/history is templated
`_links.streams/deployments/rollback/{name}/{version}.href`	`String`	Link to the streams/deployments/rollback/{name}/{version}
`_links.streams/deployments/rollback/{name}/{version}.templated`	`Boolean`	Link streams/deployments/rollback/{name}/{version} is templated
`_links.streams/deployments/update/{name}.href`	`String`	Link to the streams/deployments/update/{name}
`_links.streams/deployments/update/{name}.templated`	`Boolean`	Link streams/deployments/update/{name} is

templated

Path	Type	Description
_links.streams/deployments/platform/list.href	String	Link to the streams/deployments/platform/list
_links.streams/validation.href	String	Link to the streams/validation
_links.streams/validation.templated	Boolean	Link streams/validation is templated
_links.tasks/definitions.href	String	Link to the tasks/definitions
_links.tasks/definitions/definition.href	String	Link to the tasks/definitions/definition
_links.tasks/definitions/definition.templated	Boolean	Link tasks/definitions/definition is templated
_links.tasks/executions.href	String	Link to the tasks/executions
_links.tasks/executions/name.href	String	Link to the tasks/executions/name
_links.tasks/executions/name.templated	Boolean	Link tasks/executions/name is templated
_links.tasks/executions/current.href	String	Link to the tasks/executions/current
_links.tasks/executions/execution.href	String	Link to the tasks/executions/execution
_links.tasks/executions/execution.templated	Boolean	Link tasks/executions/execution is templated
_links.tasks/schedules.href	String	Link to the tasks/executions/schedules
_links.tasks/schedules/instances.href	String	Link to the tasks/schedules/instances
_links.tasks/schedules/instances.templated	Boolean	Link tasks/schedules/instances is templated
_links.tasks/validation.href	String	Link to the tasks/validation
_links.tasks/validation.templated	Boolean	Link tasks/validation is templated
_links.jobs/executions.href	String	Link to the jobs/executions
_links.jobs/thinexecutions.href	String	Link to the jobs/thinexecutions
_links.jobs/executions/name.href	String	Link to the jobs/executions/name
_links.jobs/executions/name.templated	Boolean	Link jobs/executions/name is templated
_links.jobs/thinexecutions/name.href	String	Link to the jobs/thinexecutions/name
_links.jobs/thinexecutions/name.templated	Boolean	Link jobs/executions/name is templated
_links.jobs/executions/execution.href	String	Link to the jobs/executions/execution
_links.jobs/executions/execution.templated	Boolean	Link jobs/executions/execution is templated
_links.jobs/executions/execution/steps.href	String	Link to the jobs/executions/execution/steps
_links.jobs/executions/execution/steps.templated	Boolean	Link jobs/executions/execution/steps is templated
_links.jobs/executions/execution/steps/step.href	String	Link to the jobs/executions/execution/steps/step
_links.jobs/executions/execution/steps/step.templated	Boolean	Link jobs/executions/execution/steps/step is templated
_links.jobs/executions/execution/steps/step/progress.href	String	Link to the jobs/executions/execution/steps/step/progress
_links.jobs/executions/execution/steps/step/progress.templated	Boolean	Link jobs/executions/execution/steps/step/progress is templated
_links.jobs/instances/name.href	String	Link to the jobs/instances/name
_links.jobs/instances/name.templated	Boolean	Link jobs/instances/name is templated
_links.jobs/instances/instance.href	String	Link to the jobs/instances/instance
_links.jobs/instances/instance.templated	Boolean	Link jobs/instances/instance is templated
_links.tools/parseTaskTextToGraph.href	String	Link to the tools/parseTaskTextToGraph
_links.tools/convertTaskGraphToText.href	String	Link to the tools/convertTaskGraphToText

Path	Type	Description
_links.counters.href	String	Link to the counters
_links.counters/counter.href	String	Link to the counters/counter
_links.counters/counter.templated	Boolean	Link counters/counter is templated
_links.field-value-counters.href	String	Link to the field-value-counters
_links.field-value-counters/counter.href	String	Link to the field-value-counters/counter
_links.field-value-counters/counter.templated	Boolean	Link field-value-counters/counter is templated
_links.aggregate-counters.href	String	Link to the aggregate-counters
_links.aggregate-counters/counter.href	String	Link to the aggregate-counters/counter
_links.aggregate-counters/counter.templated	Boolean	Link aggregate-counters/counter is templated
_links.apps.href	String	Link to the apps
_links.about.href	String	Link to the about
_links.completions/stream.href	String	Link to the completions/stream
_links.completions/stream.templated	Boolean	Link completions/stream is templated
_links.completions/task.href	String	Link to the completions/task
_links.completions/task.templated	Boolean	Link completions/task is templated

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 5809

{
  "_links" : {
    "dashboard" : {
      "href" : "http://localhost:9393/dashboard"
    },
    "audit-records" : {
      "href" : "http://localhost:9393/audit-records"
    },
    "streams/definitions" : {
      "href" : "http://localhost:9393/streams/definitions"
    },
    "streams/definitions/definition" : {
      "href" : "http://localhost:9393/streams/definitions/{name}",
      "templated" : true
    },
    "streams/validation" : {
      "href" : "http://localhost:9393/streams/validation/{name}",
      "templated" : true
    },
    "runtime/apps" : {
      "href" : "http://localhost:9393/runtime/apps"
    },
    "runtime/apps/app" : {
      "href" : "http://localhost:9393/runtime/apps/{appId}",
      "templated" : true
    },
    "runtime/apps/instances" : {
      "href" : "http://localhost:9393/runtime/apps/{appId}/instances",
      "templated" : true
    },
    "metrics/streams" : {
      "href" : "http://localhost:9393/metrics/streams"
    },
    "streams/deployments" : {
      "href" : "http://localhost:9393/streams/deployments"
    },
    "streams/deployments/{name}" : {
      "href" : "http://localhost:9393/streams/deployments/{name}",
      "templated" : true
    },
    "streams/deployments/history/{name}" : {
      "href" : "http://localhost:9393/streams/deployments/history/{name}",
      "templated" : true
    },
    "streams/deployments/manifest/{name}/{version}" : {
      "href" : "http://localhost:9393/streams/deployments/manifest/{name}/{version}",
      "templated" : true
    },
    "streams/deployments/platform/list" : {
      "href" : "http://localhost:9393/streams/deployments/platform/list"
    },
    "streams/deployments/rollback/{name}/{version}" : {
      "href" : "http://localhost:9393/streams/deployments/rollback/{name}/{version}",
      "templated" : true
    },
    "streams/deployments/update/{name}" : {
      "href" : "http://localhost:9393/streams/deployments/update/{name}",
```

```
    "templated" : true
  },
  "streams/deployments/deployment" : {
    "href" : "http://localhost:9393/streams/deployments/{name}",
    "templated" : true
  },
  "tasks/definitions" : {
    "href" : "http://localhost:9393/tasks/definitions"
  },
  "tasks/definitions/definition" : {
    "href" : "http://localhost:9393/tasks/definitions/{name}",
    "templated" : true
  },
  "tasks/executions" : {
    "href" : "http://localhost:9393/tasks/executions"
  },
  "tasks/executions/name" : {
    "href" : "http://localhost:9393/tasks/executions{?name}",
    "templated" : true
  },
  "tasks/executions/current" : {
    "href" : "http://localhost:9393/tasks/executions/current"
  },
  "tasks/executions/execution" : {
    "href" : "http://localhost:9393/tasks/executions/{id}",
    "templated" : true
  },
  "tasks/validation" : {
    "href" : "http://localhost:9393/tasks/validation/{name}",
    "templated" : true
  },
  "tasks/schedules" : {
    "href" : "http://localhost:9393/tasks/schedules"
  },
  "tasks/schedules/instances" : {
    "href" : "http://localhost:9393/tasks/schedules/instances/{taskDefinitionName}",
    "templated" : true
  },
  "jobs/executions" : {
    "href" : "http://localhost:9393/jobs/executions"
  },
  "jobs/executions/name" : {
    "href" : "http://localhost:9393/jobs/executions{?name}",
    "templated" : true
  },
  "jobs/executions/execution" : {
    "href" : "http://localhost:9393/jobs/executions/{id}",
    "templated" : true
  },
  "jobs/executions/execution/steps" : {
    "href" : "http://localhost:9393/jobs/executions/{jobExecutionId}/steps",
    "templated" : true
  },
  "jobs/executions/execution/steps/step" : {
    "href" : "http://localhost:9393/jobs/executions/{jobExecutionId}/steps/{stepId}",
    "templated" : true
  },
  "jobs/executions/execution/steps/step/progress" : {
    "href" : "http://localhost:9393/jobs/executions/{jobExecutionId}/steps/{stepId}/progress",
    "templated" : true
  },
  "jobs/instances/name" : {
    "href" : "http://localhost:9393/jobs/instances{?name}",
    "templated" : true
  },
  "jobs/instances/instance" : {
    "href" : "http://localhost:9393/jobs/instances/{id}",
    "templated" : true
  },
  "tools/parseTaskTextToGraph" : {
    "href" : "http://localhost:9393/tools"
  },
  "tools/convertTaskGraphToText" : {
    "href" : "http://localhost:9393/tools"
  },
  "jobs/thinexecutions" : {
    "href" : "http://localhost:9393/jobs/thinexecutions"
  },
  "jobs/thinexecutions/name" : {
    "href" : "http://localhost:9393/jobs/thinexecutions{?name}",
    "templated" : true
  },
  "counters" : {
    "href" : "http://localhost:9393/metrics/counters"
  },
  "counters/counter" : {
    "href" : "http://localhost:9393/metrics/counters/{name}",
    "templated" : true
  },
  "field-value-counters" : {
    "href" : "http://localhost:9393/metrics/field-value-counters"
  },
  "field-value-counters/counter" : {
    "href" : "http://localhost:9393/metrics/field-value-counters/{name}",
    "templated" : true
  },
  "aggregate-counters" : {
    "href" : "http://localhost:9393/metrics/aggregate-counters"
  },
  "aggregate-counters/counter" : {
    "href" : "http://localhost:9393/metrics/aggregate-counters/{name}"
```

```
      "href" : "http://localhost:9393/metrics/aggregate-counters/{name}",
      "templated" : true
    },
    "apps" : {
      "href" : "http://localhost:9393/apps"
    },
    "about" : {
      "href" : "http://localhost:9393/about"
    },
    "completions/stream" : {
      "href" : "http://localhost:9393/completions/stream{?start,detailLevel}",
      "templated" : true
    },
    "completions/task" : {
      "href" : "http://localhost:9393/completions/task{?start,detailLevel}",
      "templated" : true
    }
  },
  "api.revision" : 14
}
```


Links

The main element of the index are the links, as they let you traverse the API and execute the desired functionality:

Relation	Description
about	Access meta information, including enabled features, security info, version information
dashboard	Access the dashboard UI
audit-records	Provides audit trail information
apps	Handle registered applications
completions/stream	Exposes the DSL completion features for Stream
completions/task	Exposes the DSL completion features for Task
metrics/streams	Exposes metrics for the stream applications
jobs/executions	Provides the JobExecution resource
jobs/thinexecutions	Provides the JobExecution thin resource with no step executions included
jobs/executions/execution	Provides details for a specific JobExecution
jobs/executions/execution/steps	Provides the steps for a JobExecution
jobs/executions/execution/steps/step	Returns the details for a specific step
jobs/executions/execution/steps/step/progress	Provides progress information for a specific step
jobs/executions/name	Retrieve Job Executions by Job name
jobs/thinexecutions/name	Retrieve Job Executions by Job name with no step executions included
jobs/instances/instance	Provides the job instance resource for a specific job instance
jobs/instances/name	Provides the Job instance resource for a specific job name
runtime/apps	Provides the runtime application resource
runtime/apps/app	Exposes the runtime status for a specific app
runtime/apps/instances	Provides the status for app instances
tasks/definitions	Provides the task definition resource
tasks/definitions/definition	Provides details for a specific task definition
tasks/validation	Provides the validation for a task definition
tasks/executions	Returns Task executions and allows launching of tasks
tasks/executions/current	Provides the current count of running tasks
tasks/schedules	Provides schedule information of tasks
tasks/schedules/instances	Provides schedule information of a specific task

Relation	Description
tasks/executions/name	Returns all task executions for a given Task name
tasks/executions/execution	Provides details for a specific task execution
streams/definitions	Exposes the Streams resource
streams/definitions/definition	Handle a specific Stream definition
streams/validation	Provides the validation for a stream definition
streams/deployments	Provides Stream deployment operations
streams/deployments/{name}	Request un-deployment of an existing stream
streams/deployments/deployment	Request (un-)deployment of an existing stream definition
streams/deployments/manifest/{name}/{version}	Return a manifest info of a release version
streams/deployments/history/{name}	Get stream's deployment history as list or Releases for this release
streams/deployments/rollback/{name}/{version}	Rollback the stream to the previous or a specific version of the stream
streams/deployments/update/{name}	Update the stream.
streams/deployments/platform/list	List of supported deployment platforms
counters	Exposes the resource for dealing with Counters
counters/counter	Handle a specific counter
aggregate-counters	Provides the resource for dealing with aggregate counters
aggregate-counters/counter	Handle a specific aggregate counter
field-value-counters	Provides the resource for dealing with field-value-counters
field-value-counters/counter	Handle a specific field-value-counter
tools/parseTaskTextToGraph	Parse a task definition into a graph structure
tools/convertTaskGraphToText	Convert a graph format into DSL text format

59.2. Server Meta Information

The server meta information endpoint provides more information about the server itself. The following topics provide more detail:

- Retrieving information about the server
- Request Structure
- Example Request
- Response Structure

59.2.1. Retrieving information about the server

A `GET` request returns meta information for Spring Cloud Data Flow, including:

- Runtime environment information
- Information regarding which features are enabled
- Dependency information of Spring Cloud Data Flow Server
- Security information

Request Structure

```
GET /about HTTP/1.1
Accept: application/json
Host: localhost:9393
```

Example Request

```
$ curl 'http://localhost:9393/about' -i -X GET \
    -H 'Accept: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 2065
Content-Type: application/json;charset=UTF-8

{
  "featureInfo" : {
    "analyticsEnabled" : true,
    "streamsEnabled" : true,
    "tasksEnabled" : true,
    "schedulerEnabled" : true
  },
  "versionInfo" : {
    "implementation" : {
      "name" : "${info.app.name}",
      "version" : "${info.app.version}"
    },
    "core" : {
      "name" : "Spring Cloud Data Flow Core",
      "version" : "2.0.0.BUILD-SNAPSHOT"
    },
    "dashboard" : {
      "name" : "Spring Cloud Dataflow UI",
      "version" : "2.0.0.M2"
    },
    "shell" : {
      "name" : "Spring Cloud Data Flow Shell",
      "version" : "2.0.0.BUILD-SNAPSHOT",
      "url" : "https://repo.spring.io/libs-snapshot/org/springframework/cloud/spring-cloud-dataflow-shell/2.0.0.BUILD-SNAPS
    }
  },
  "securityInfo" : {
    "authenticationEnabled" : false,
    "formLogin" : false,
    "authenticated" : false,
    "username" : null,
    "roles" : [ ]
  },
  "runtimeEnvironment" : {
    "appDeployer" : {
      "deployerImplementationVersion" : "Test Version",
      "deployerName" : "Test Server",
      "deployerSpiVersion" : "2.0.0.M1",
      "javaVersion" : "1.8.0_144",
      "platformApiVersion" : "",
      "platformClientVersion" : "",
      "platformHostVersion" : "",
      "platformSpecificInfo" : {
        "default" : "local"
      },
      "platformType" : "Skipper Managed",
      "springBootVersion" : "2.1.1.RELEASE",
      "springVersion" : "5.1.3.RELEASE"
    },
    "taskLauncher" : {
      "deployerImplementationVersion" : "2.0.0.M2",
      "deployerName" : "LocalTaskLauncher",
      "deployerSpiVersion" : "2.0.0.M2",
      "javaVersion" : "1.8.0_144",
      "platformApiVersion" : "Linux 4.4.0-139-generic",
      "platformClientVersion" : "4.4.0-139-generic",
      "platformHostVersion" : "4.4.0-139-generic",
      "platformSpecificInfo" : { },
      "platformType" : "Local",
      "springBootVersion" : "2.1.1.RELEASE",
      "springVersion" : "5.1.3.RELEASE"
    }
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/about"
    }
  }
}
```

59.3. Registered Applications

The registered applications endpoint provides information about the applications that are registered with the Spring Cloud Data Flow server. The following topics provide more detail:

- Listing Applications
- Getting Information on a Particular Application
- Registering a New Application
- Registering a New Application with version
- Set the default application version
- Unregistering an Application
- Registering Applications in Bulk

59.3.1. Listing Applications

A `GET` request will list all applications known to Spring Cloud Data Flow. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /apps?type=source HTTP/1.1
Accept: application/json
Host: localhost:9393
```

Request Parameters

Parameter	Description
`type`	Restrict the returned apps to the type of the app. One of [app, source, processor, sink, task]

Example Request

```
$ curl 'http://localhost:9393/apps?type=source' -i -X GET \
    -H 'Accept: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 959
Content-Type: application/json;charset=UTF-8

{
  "_embedded" : {
    "appRegistrationResourceList" : [ {
      "name" : "http",
      "type" : "source",
      "uri" : "maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE",
      "version" : "1.2.0.RELEASE",
      "defaultVersion" : true,
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/apps/source/http/1.2.0.RELEASE"
        }
      }
    }, {
      "name" : "time",
      "type" : "source",
      "uri" : "maven://org.springframework.cloud.stream.app:time-source-rabbit:1.2.0.RELEASE",
      "version" : "1.2.0.RELEASE",
      "defaultVersion" : true,
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/apps/source/time/1.2.0.RELEASE"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/apps?page=0&size=20"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 2,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.3.2. Getting Information on a Particular Application

A `GET` request on `/apps/<type>/<name>` gets info on a particular application. The following topics provide more detail:

- Request Structure
- Path Parameters
- Example Request
- Response Structure

Request Structure

```
GET /apps/source/http?exhaustive=false HTTP/1.1
Accept: application/json
Host: localhost:9393
```

Request Parameters

Parameter	Description
`exhaustive`	Return all application properties, including common Spring Boot properties

Path Parameters

Table 2. /apps/{type}/{name}

Parameter	Description
`type`	The type of application to query. One of [app, source, processor, sink, task]
`name`	The name of the application to query

Example Request

```
$ curl 'http://localhost:9393/apps/source/http?exhaustive=false' -i -X GET \
    -H 'Accept: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 2446
Content-Type: application/json;charset=UTF-8

{
  "name" : "http",
  "type" : "source",
  "uri" : "maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE",
  "version" : "1.2.0.RELEASE",
  "defaultVersion" : true,
  "options" : [ {
    "id" : "http.path-pattern",
    "name" : "path-pattern",
    "type" : "java.lang.String",
    "description" : "An Ant-Style pattern to determine which http requests will be captured.",
    "shortDescription" : "An Ant-Style pattern to determine which http requests will be captured.",
    "defaultValue" : "/",
    "hints" : {
      "keyHints" : [ ],
      "keyProviders" : [ ],
      "valueHints" : [ ],
      "valueProviders" : [ ]
    },
    "deprecation" : null,
    "sourceType" : "org.springframework.cloud.stream.app.http.source.HttpSourceProperties",
    "sourceMethod" : null,
    "deprecated" : false
  }, {
    "id" : "http.mapped-request-headers",
    "name" : "mapped-request-headers",
    "type" : "java.lang.String[]",
    "description" : "Headers that will be mapped.",
    "shortDescription" : "Headers that will be mapped.",
    "defaultValue" : null,
    "hints" : {
      "keyHints" : [ ],
      "keyProviders" : [ ],
      "valueHints" : [ ],
      "valueProviders" : [ ]
    },
    "deprecation" : null,
    "sourceType" : "org.springframework.cloud.stream.app.http.source.HttpSourceProperties",
    "sourceMethod" : null,
    "deprecated" : false
  }, {
    "id" : "http.secured",
    "name" : "secured",
    "type" : "java.lang.Boolean",
    "description" : "Secure or not HTTP source path.",
    "shortDescription" : "Secure or not HTTP source path.",
    "defaultValue" : false,
    "hints" : {
      "keyHints" : [ ],
      "keyProviders" : [ ],
      "valueHints" : [ ],
      "valueProviders" : [ ]
    },
    "deprecation" : null,
    "sourceType" : "org.springframework.cloud.stream.app.http.source.HttpSourceProperties",
    "sourceMethod" : null,
    "deprecated" : false
  }, {
    "id" : "server.port",
    "name" : "port",
    "type" : "java.lang.Integer",
    "description" : "Server HTTP port.",
    "shortDescription" : "Server HTTP port.",
    "defaultValue" : null,
    "hints" : {
      "keyHints" : [ ],
      "keyProviders" : [ ],
      "valueHints" : [ ],
      "valueProviders" : [ ]
    },
    "deprecation" : null,
    "sourceType" : "org.springframework.boot.autoconfigure.web.ServerProperties",
    "sourceMethod" : null,
    "deprecated" : false
  } ],
  "shortDescription" : null
}
```

59.3.3. Registering a New Application

A `POST` request on `/apps/<type>/<name>` allows registration of a new application. The following topics provide more detail:

- Request Structure
- Request Parameters
- Path Parameters
- Example Request
- Response Structure

Request Structure

```
POST /apps/source/http HTTP/1.1
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

uri=maven%3A%2F%2Forg.springframework.cloud.stream.app%3Ahttp-source-rabbit%3A1.1.0.RELEASE
```

Request Parameters

Parameter	Description
`uri`	URI where the application bits reside
`metadata-uri`	URI where the application metadata jar can be found
`force`	Must be true if a registration with the same name and type already exists, otherwise an error will occur

Path Parameters

Table 3. /apps/{type}/{name}

Parameter	Description
`type`	The type of application to register. One of [app, source, processor, sink, task]
`name`	The name of the application to register

Example Request

```
$ curl 'http://localhost:9393/apps/source/http' -i -X POST \
    -d 'uri=maven%3A%2F%2Forg.springframework.cloud.stream.app%3Ahttp-source-rabbit%3A1.1.0.RELEASE'
```

Response Structure

```
HTTP/1.1 201 Created
```

59.3.4. Registering a New Application with version

A `POST` request on `/apps/<type>/<name>/<version>` allows registration of a new application The following topics provide more detail:

- Request Structure
- Request Parameters
- Path Parameters
- Example Request
- Response Structure

Request Structure

```
POST /apps/source/http/1.1.0.RELEASE HTTP/1.1
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

uri=maven%3A%2F%2Forg.springframework.cloud.stream.app%3Ahttp-source-rabbit%3A1.1.0.RELEASE
```

Request Parameters

Parameter	Description
`uri`	URI where the application bits reside
`metadata-uri`	URI where the application metadata jar can be found
`force`	Must be true if a registration with the same name and type already exists, otherwise an error will occur

Path Parameters

Table 4. /apps/{type}/{name}/{version:.}+

Parameter	Description
`type`	The type of application to register. One of [app, source, processor, sink, task]
`name`	The name of the application to register

Parameter	Description
version	The version of the application to register

Example Request

```
$ curl 'http://localhost:9393/apps/source/http/1.1.0.RELEASE' -i -X POST \
    -d 'uri=maven%3A%2F%2Forg.springframework.cloud.stream.app%3Ahttp-source-rabbit%3A1.1.0.RELEASE'
```

Response Structure

```
HTTP/1.1 201 Created
```

59.3.5. Set the default application version

For an application with the `name` and `type`, multiple versions can be registered. In this case, one of the versions can be chosen as the default application.

The following topics provide more detail:

- Request Structure
- Path Parameters
- Example Request
- Response Structure

Request Structure

```
PUT /apps/source/http/1.2.0.RELEASE HTTP/1.1
Accept: application/json
Host: localhost:9393
```

Path Parameters

Table 5. /apps/{type}/{name}/{version:.}+

Parameter	Description
type	The type of application. One of [app, source, processor, sink, task]
name	The name of the application
version	The version of the application

Example Request

```
$ curl 'http://localhost:9393/apps/source/http/1.2.0.RELEASE' -i -X PUT \
    -H 'Accept: application/json'
```

Response Structure

```
HTTP/1.1 202 Accepted
```

59.3.6. Unregistering an Application

A `DELETE` request on `/apps/<type>/<name>` unregisters a previously registered application. The following topics provide more detail:

- Request Structure
- Path Parameters
- Example Request
- Response Structure

Request Structure

```
DELETE /apps/source/http HTTP/1.1
Host: localhost:9393
```

Path Parameters

Table 6. /apps/{type}/{name}

Parameter	Description
type	The type of application to unregister. One of [app, source, processor, sink, task]
name	The name of the application to unregister

Example Request

```
$ curl 'http://localhost:9393/apps/source/http' -i -X DELETE
```

Response Structure

```
HTTP/1.1 200 OK
```

59.3.7. Registering Applications in Bulk

A `POST` request on `/apps` allows registering multiple applications at once. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
POST /apps HTTP/1.1
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

apps=source.http%3Dmaven%3A%2F%2Forg.springframework.cloud.stream.app%3Ahttp-source-rabbit%3A1.1.0.RELEASE&force=false
```

Request Parameters

Parameter	Description
`uri`	URI where a properties file containing registrations can be fetched. Exclusive with `apps`.
`apps`	Inline set of registrations. Exclusive with `uri`.
`force`	Must be true if a registration with the same name and type already exists, otherwise an error will occur

Example Request

```
$ curl 'http://localhost:9393/apps' -i -X POST \
    -d 'apps=source.http%3Dmaven%3A%2F%2Forg.springframework.cloud.stream.app%3Ahttp-source-
rabbit%3A1.1.0.RELEASE&force=false'
```

Response Structure

```
HTTP/1.1 201 Created
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 611

{
  "_embedded" : {
    "appRegistrationResourceList" : [ {
      "name" : "http",
      "type" : "source",
      "uri" : "maven://org.springframework.cloud.stream.app:http-source-rabbit:1.1.0.RELEASE",
      "version" : "1.1.0.RELEASE",
      "defaultVersion" : true,
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/apps/source/http/1.1.0.RELEASE"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/apps?page=0&size=20"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.4. Audit Records

59.4.1. List All Audit Records

The audit records endpoint lets you retrieve audit trail information.

The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /audit-records?page=0&size=10&operations=STREAM&actions=CREATE HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)
`operations`	Comma-separated list of Audit Operations (optional)
`actions`	Comma-separated list of Audit Actions (optional)

Example Request

```
$ curl 'http://localhost:9393/audit-records?page=0&size=10&operations=STREAM&actions=CREATE' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 651

{
  "_embedded" : {
    "auditRecordResourceList" : [ {
      "auditRecordId" : 3,
      "createdBy" : null,
      "correlationId" : "timelog",
      "auditData" : "time --format='YYYY MM DD' | log",
      "createdOn" : "2019-01-10T21:30:12.094Z",
      "auditAction" : "CREATE",
      "auditOperation" : "STREAM",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/audit-records/3"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/audit-records?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.5. Stream Definitions

The registered applications endpoint provides information about the stream definitions that are registered with the Spring Cloud Data Flow server. The following topics provide more detail:

- Creating a New Stream Definition
- List All Stream Definitions
- List Related Stream Definitions
- Delete a Single Stream Definition
- Delete All Stream Definitions
- Deploying Stream Definition
- Undeploy Stream Definition
- Undeploy All Stream Definitions

59.5.1. Creating a New Stream Definition

Creating a stream definition is achieved by creating a POST request to the stream definitions endpoint. A curl request

for a `ticktock` stream might resemble the following:

```
curl -X POST -d "name=ticktock&definition=time | log" localhost:9393/streams/definitions?deploy=false
```

A stream definition can also contain additional parameters. For instance, in the example shown under "Request Structure", we also provide the date-time format.

The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
POST /streams/definitions HTTP/1.1
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

name=timelog&definition=time+--format%3D%27YYYY+MM+DD%27+%7C+log&deploy=false
```

Request Parameters

Parameter	Description
`name`	The name for the created task definitions
`definition`	The definition for the stream, using Data Flow DSL
`deploy`	If true, the stream is deployed upon creation (default is false

Example Request

```
$ curl 'http://localhost:9393/streams/definitions' -i -X POST \
    -d 'name=timelog&definition=time+--format%3D%27YYYY+MM+DD%27+%7C+log&deploy=false'
```

Response Structure

```
HTTP/1.1 201 Created
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 288

{
  "name" : "timelog",
  "dslText" : "time --format='YYYY MM DD' | log",
  "status" : "unknown",
  "statusDescription" : "The app or group deployment is not known to the system",
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/streams/definitions/timelog"
    }
  }
}
```

59.5.2. List All Stream Definitions

The streams endpoint lets you list all the stream definitions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /streams/definitions?page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)

Example Request

```
$ curl 'http://localhost:9393/streams/definitions?page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 609
Content-Type: application/hal+json;charset=UTF-8

{
  "_embedded" : {
    "streamDefinitionResourceList" : [ {
      "name" : "timelog",
      "dslText" : "time --format='YYYY MM DD' | log",
      "status" : "unknown",
      "statusDescription" : "The app or group deployment is not known to the system",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/streams/definitions/timelog"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/streams/definitions?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.5.3. List Related Stream Definitions

The streams endpoint lets you list related stream definitions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /streams/definitions/timelog/related?nested=true HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`nested`	Should we recursively findByTaskNameLike for related stream definitions (optional)

Example Request

```
$ curl 'http://localhost:9393/streams/definitions/timelog/related?nested=true' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 625
Content-Type: application/hal+json;charset=UTF-8

{
  "_embedded" : {
    "streamDefinitionResourceList" : [ {
      "name" : "timelog",
      "dslText" : "time --format='YYYY MM DD' | log",
      "status" : "unknown",
      "statusDescription" : "The app or group deployment is not known to the system",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/streams/definitions/timelog"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/streams/definitions/timelog/related?page=0&size=20"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.5.4. Delete a Single Stream Definition

The streams endpoint lets you delete a single stream definition. (See also:Delete All Stream Definitions.) The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
DELETE /streams/definitions/timelog HTTP/1.1
Host: localhost:9393
```

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/streams/definitions/timelog' -i -X DELETE
```

Response Structure

```
HTTP/1.1 200 OK
```

59.5.5. Delete All Stream Definitions

The streams endpoint lets you delete all single stream definitions. (See also:Delete a Single Stream Definition.) The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
DELETE /streams/definitions HTTP/1.1
Host: localhost:9393
```

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/streams/definitions' -i -X DELETE
```

Response Structure

```
HTTP/1.1 200 OK
```

59.6. Stream Validation

The stream validation endpoint lets you validate the apps in a stream definition. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

59.6.1. Request Structure

```
GET /streams/validation/timelog HTTP/1.1
Host: localhost:9393
```

59.6.2. Request Parameters

There are no request parameters for this endpoint.

59.6.3. Example Request

```
$ curl 'http://localhost:9393/streams/validation/timelog' -i -X GET
```

59.6.4. Response Structure

```
HTTP/1.1 200 OK
Content-Length: 152
Content-Type: application/hal+json;charset=UTF-8

{
  "appName" : "timelog",
  "dsl" : "time --format='YYYY MM DD' | log",
  "appStatuses" : {
    "source:time" : "valid",
    "sink:log" : "valid"
  }
}
```

59.7. Stream Deployments

The deployment definitions endpoint provides information about the deployments that are registered with the Spring Cloud Data Flow server. The following topics provide more detail:

- Deploying Stream Definition
- Undeploy Stream Definition
- Undeploy All Stream Definitions
- Update Deployed Stream
- Rollback Stream Definition
- Get Manifest
- Get Deployment History
- Get Deployment Platforms

59.7.1. Deploying Stream Definition

The stream definition endpoint lets you deploy a single stream definition. Optionally, you can pass application parameters as properties in the request body. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
POST /streams/deployments/timelog HTTP/1.1
Host: localhost:9393
Content-Type: application/json
Content-Length: 36

{"app.time.timestamp.format":"YYYY"}
```

Table 7. /streams/deployments/{timelog}

Parameter	Description
`timelog`	The name of an existing stream definition (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/streams/deployments/timelog' -i -X POST \
    -H 'Content-Type: application/json' \
    -d '{"app.time.timestamp.format":"YYYY"}'
```

Response Structure

```
HTTP/1.1 201 Created
```

59.7.2. Undeploy Stream Definition

The stream definition endpoint lets you undeploy a single stream definition. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
DELETE /streams/deployments/timelog HTTP/1.1
Host: localhost:9393
```

Table 8. /streams/deployments/{timelog}

Parameter	Description
`timelog`	The name of an existing stream definition (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/streams/deployments/timelog' -i -X DELETE
```

Response Structure

```
HTTP/1.1 200 OK
```

59.7.3. Undeploy All Stream Definitions

The stream definition endpoint lets you undeploy all single stream definitions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
DELETE /streams/deployments HTTP/1.1
Host: localhost:9393
```

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/streams/deployments' -i -X DELETE
```

Response Structure

```
HTTP/1.1 200 OK
```

59.7.4. Update Deployed Stream

Thanks to Skipper, you can update deployed streams, and provide additional deployment properties.

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
POST /streams/deployments/update/timelog1 HTTP/1.1
Host: localhost:9393
Content-Length: 196
Content-Type: application/json

{"releaseName":"timelog1","packageIdentifier":{"repositoryName":"test","packageName":"timelog1","packageVersion":"1.0.0"},"
```

Table 9. /streams/deployments/update/{timelog1}

Parameter	Description
timelog1	The name of an existing stream definition (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/streams/deployments/update/timelog1' -i -X POST \
    -H 'Content-Type: application/json' \
    -d '{"releaseName":"timelog1","packageIdentifier":
{"repositoryName":"test","packageName":"timelog1","packageVersion":"1.0.0"},"updateProperties":
{"app.time.timestamp.format":"YYYYMMDD"},"force":false}'
```

Response Structure

```
HTTP/1.1 201 Created
```

59.7.5. Rollback Stream Definition

Rollback the stream to the previous or a specific version of the stream.

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
POST /streams/deployments/rollback/timelog1/1 HTTP/1.1
Host: localhost:9393
Content-Type: application/json
```

Table 10. /streams/deployments//rollback/{name}/{version}

Parameter	Description
name	The name of an existing stream definition (required)
version	The version to rollback to

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/streams/deployments/rollback/timelog1/1' -i -X POST \
    -H 'Content-Type: application/json'
```

Response Structure

```
HTTP/1.1 201 Created
```

59.7.6. Get Manifest

Return a manifest of a released version. For packages with dependencies, the manifest includes the contents of those dependencies.

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /streams/deployments/manifest/timelog1/1 HTTP/1.1
Host: localhost:9393
Content-Type: application/json
```

Table 11. /streams/deployments/manifest/{name}/{version}

Parameter	Description
`name`	The name of an existing stream definition (required)
`version`	The version of the stream

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/streams/deployments/manifest/timelog1/1' -i -X GET \
    -H 'Content-Type: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
```

59.7.7. Get Deployment History

Get the stream's deployment history.

- Request Structure
- Example Request
- Response Structure

Request Structure

```
GET /streams/deployments/history/timelog1 HTTP/1.1
Host: localhost:9393
Content-Type: application/json
```

Example Request

```
$ curl 'http://localhost:9393/streams/deployments/history/timelog1' -i -X GET \
    -H 'Content-Type: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 162
Content-Type: application/json;charset=UTF-8

[ {
  "name" : null,
  "version" : 0,
  "info" : null,
  "pkg" : null,
  "configValues" : {
    "raw" : null
  },
  "manifest" : null,
  "platformName" : null
} ]
```

59.7.8. Get Deployment Platforms

Retrieve a list of supported deployment platforms.

- Request Structure
- Example Request

- Response Structure

Request Structure

```
GET /streams/deployments/platform/list HTTP/1.1
Host: localhost:9393
Content-Type: application/json
```

Example Request

```
$ curl 'http://localhost:9393/streams/deployments/platform/list' -i -X GET \
    -H 'Content-Type: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 87
Content-Type: application/json;charset=UTF-8

[ {
  "id" : null,
  "name" : "default",
  "type" : "local",
  "description" : null
} ]
```

59.8. Task Definitions

The task definitions endpoint provides information about the task definitions that are registered with the Spring Cloud Data Flow server. The following topics provide more detail:

- Creating a New Task Definition
- List All Task Definitions
- Retrieve Task Definition Detail
- Delete Task Definition

59.8.1. Creating a New Task Definition

The task definition endpoint lets you create a new task definition. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
POST /tasks/definitions HTTP/1.1
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

name=my-task&definition=timestamp+--format%3D%27YYYY+MM+DD%27
```

Request Parameters

Parameter	Description
`name`	The name for the created task definition
`definition`	The definition for the task, using Data Flow DSL

Example Request

```
$ curl 'http://localhost:9393/tasks/definitions' -i -X POST \
    -d 'name=my-task&definition=timestamp+--format%3D%27YYYY+MM+DD%27'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 255

{
  "name" : "my-task",
  "dslText" : "timestamp --format='YYYY MM DD'",
  "composed" : false,
  "lastTaskExecution" : null,
  "status" : "UNKNOWN",
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/tasks/definitions/my-task"
    }
  }
}
```

59.8.2. List All Task Definitions

The task definition endpoint lets you get all task definitions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /tasks/definitions?page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)

Example Request

```
$ curl 'http://localhost:9393/tasks/definitions?page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 576

{
  "_embedded" : {
    "taskDefinitionResourceList" : [ {
      "name" : "my-task",
      "dslText" : "timestamp --format='YYYY MM DD'",
      "composed" : false,
      "lastTaskExecution" : null,
      "status" : "UNKNOWN",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/tasks/definitions/my-task"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/tasks/definitions?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.8.3. Retrieve Task Definition Detail

The task definition endpoint lets you get a single task definition. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request

- Response Structure

Request Structure

```
GET /tasks/definitions/my-task HTTP/1.1
Host: localhost:9393
```

Table 12. /tasks/definitions/{my-task}

Parameter	Description
`my-task`	The name of an existing task definition (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/tasks/definitions/my-task' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 255

{
  "name" : "my-task",
  "dslText" : "timestamp --format='YYYY MM DD'",
  "composed" : false,
  "lastTaskExecution" : null,
  "status" : "UNKNOWN",
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/tasks/definitions/my-task"
    }
  }
}
```

59.8.4. Delete Task Definition

The task definition endpoint lets you delete a single task definition. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
DELETE /tasks/definitions/my-task HTTP/1.1
Host: localhost:9393
```

Table 13. /tasks/definitions/{my-task}

Parameter	Description
`my-task`	The name of an existing task definition (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/tasks/definitions/my-task' -i -X DELETE
```

Response Structure

```
HTTP/1.1 200 OK
```

59.9. Task Scheduler

The task scheduler endpoint provides information about the task schedules that are registered with the Scheduler Implementation. The following topics provide more detail:

- Creating a New Task Schedule
- List All Schedules
- List Filtered Schedules

- Delete Task Schedule

59.9.1. Creating a New Task Schedule

The task schedule endpoint lets you create a new task schedule. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
POST /tasks/schedules HTTP/1.1
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

scheduleName=myschedule&taskDefinitionName=mytaskname&properties=scheduler.cron.expression%3D00+22+17+%3F+*&arguments=--foo
```

Request Parameters

Parameter	Description
`scheduleName`	The name for the created schedule
`taskDefinitionName`	The name of the task definition to be scheduled
`properties`	the properties that are required to schedule and launch the task
`arguments`	the command line arguments to be used for launching the task

Example Request

```
$ curl 'http://localhost:9393/tasks/schedules' -i -X POST \
    -d
'scheduleName=myschedule&taskDefinitionName=mytaskname&properties=scheduler.cron.expression%3D00+22+17+%3F+*&arguments=--
foo%3Dbar'
```

Response Structure

```
HTTP/1.1 201 Created
```

59.9.2. List All Schedules

The task schedules endpoint lets you get all task schedules. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /tasks/schedules?page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)

Example Request

```
$ curl 'http://localhost:9393/tasks/schedules?page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 585

{
  "_embedded" : {
    "scheduleInfoResourceList" : [ {
      "scheduleName" : "FOO",
      "taskDefinitionName" : "BAR",
      "scheduleProperties" : {
        "scheduler.AAA.spring.cloud.scheduler.cron.expression" : "00 41 17 ? * *"
      },
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/tasks/schedules/FOO"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/tasks/schedules?page=0&size=1"
    }
  },
  "page" : {
    "size" : 1,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.9.3. List Filtered Schedules

The task schedules endpoint lets you get all task schedules that have the specified task definition name. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /tasks/schedules/instances/FOO?page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Table 14. /tasks/schedules/instances/{task-definition-name}

Parameter	Description
`task-definition-name`	Filter schedules based on the specified task definition (required)

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)

Example Request

```
$ curl 'http://localhost:9393/tasks/schedules/instances/FOO?page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 599

{
  "_embedded" : {
    "scheduleInfoResourceList" : [ {
      "scheduleName" : "FOO",
      "taskDefinitionName" : "BAR",
      "scheduleProperties" : {
        "scheduler.AAA.spring.cloud.scheduler.cron.expression" : "00 41 17 ? * *"
      },
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/tasks/schedules/FOO"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/tasks/schedules/instances/FOO?page=0&size=1"
    }
  },
  "page" : {
    "size" : 1,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.9.4. Delete Task Schedule

The task schedule endpoint lets you delete a single task schedule. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
DELETE /tasks/schedules/mytestschedule HTTP/1.1
Host: localhost:9393
```

Table 15. /tasks/schedules/{scheduleName}

Parameter	Description
`scheduleName`	The name of an existing schedule (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/tasks/schedules/mytestschedule' -i -X DELETE
```

Response Structure

```
HTTP/1.1 200 OK
```

59.10. Task Validation

The task validation endpoint lets you validate the apps in a task definition. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /tasks/validation/taskC HTTP/1.1
Host: localhost:9393
```

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/tasks/validation/taskC' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 122

{
  "appName" : "taskC",
  "dsl" : "timestamp --format='yyyy MM dd'",
  "appStatuses" : {
    "task:taskC" : "valid"
  }
}
```

59.11. Task Executions

The task executions endpoint provides information about the task executions that are registered with the Spring Cloud Data Flow server. The following topics provide more detail:

- Launching a Task
- List All Task Executions
- List All Task Executions With a Specified Task Name
- Task Execution Detail
- Delete Task Execution

59.11.1. Launching a Task

Launching a task is done by requesting the creation of a new task execution. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
POST /tasks/executions HTTP/1.1
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

name=taskA&properties=app.my-task.foo%3Dbar%2Cdeployer.my-task.something-else%3D3&arguments=--server.port%3D8080+--foo%3Dba
```

Request Parameters

Parameter	Description
`name`	The name of the task definition to launch
`properties`	Application and Deployer properties to use while launching
`arguments`	Command line arguments to pass to the task

Example Request

```
$ curl 'http://localhost:9393/tasks/executions' -i -X POST \
    -d 'name=taskA&properties=app.my-task.foo%3Dbar%2Cdeployer.my-task.something-else%3D3&arguments=--
server.port%3D8080+--foo%3Dbar'
```

Response Structure

```
HTTP/1.1 201 Created
Content-Type: application/json;charset=UTF-8
Content-Length: 1

1
```

59.11.2. List All Task Executions

The task executions endpoint lets you list all task executions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request

- Response Structure

Request Structure

```
GET /tasks/executions?page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
page	The zero-based page number (optional)
size	The requested page size (optional)

Example Request

```
$ curl 'http://localhost:9393/tasks/executions?page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 1243
Content-Type: application/hal+json;charset=UTF-8

{
  "_embedded" : {
    "taskExecutionResourceList" : [ {
      "executionId" : 2,
      "exitCode" : null,
      "taskName" : "taskB",
      "startTime" : null,
      "endTime" : null,
      "exitMessage" : null,
      "arguments" : [ ],
      "jobExecutionIds" : [ ],
      "errorMessage" : null,
      "externalExecutionId" : "taskB-039dc2ca-2f84-4b3b-80aa-612c85766892",
      "taskExecutionStatus" : "UNKNOWN",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/tasks/executions/2"
        }
      }
    }, {
      "executionId" : 1,
      "exitCode" : null,
      "taskName" : "taskA",
      "startTime" : null,
      "endTime" : null,
      "exitMessage" : null,
      "arguments" : [ ],
      "jobExecutionIds" : [ ],
      "errorMessage" : null,
      "externalExecutionId" : "taskA-7df00ae4-d1cd-4447-95b4-95d6697b03d7",
      "taskExecutionStatus" : "UNKNOWN",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/tasks/executions/1"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/tasks/executions?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 2,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.11.3. List All Task Executions With a Specified Task Name

The task executions endpoint lets you list task executions with a specified task name. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /tasks/executions?name=taskB&page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
page	The zero-based page number (optional)
size	The requested page size (optional)
name	The name associated with the task execution

Example Request

```
$ curl 'http://localhost:9393/tasks/executions?name=taskB&page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 758
Content-Type: application/hal+json;charset=UTF-8

{
  "_embedded" : {
    "taskExecutionResourceList" : [ {
      "executionId" : 2,
      "exitCode" : null,
      "taskName" : "taskB",
      "startTime" : null,
      "endTime" : null,
      "exitMessage" : null,
      "arguments" : [ ],
      "jobExecutionIds" : [ ],
      "errorMessage" : null,
      "externalExecutionId" : "taskB-039dc2ca-2f84-4b3b-80aa-612c85766892",
      "taskExecutionStatus" : "UNKNOWN",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/tasks/executions/2"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/tasks/executions?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.11.4. Task Execution Detail

The task executions endpoint lets you get the details about a task execution. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /tasks/executions/1 HTTP/1.1
Host: localhost:9393
```

Table 16. /tasks/executions/{id}

Parameter	Description
id	The id of an existing task execution (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/tasks/executions/1' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 415

{
  "executionId" : 1,
  "exitCode" : null,
  "taskName" : "taskA",
  "startTime" : null,
  "endTime" : null,
  "exitMessage" : null,
  "arguments" : [ ],
  "jobExecutionIds" : [ ],
  "errorMessage" : null,
  "externalExecutionId" : "taskA-7df00ae4-d1cd-4447-95b4-95d6697b03d7",
  "taskExecutionStatus" : "UNKNOWN",
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/tasks/executions/1"
    }
  }
}
```

59.11.5. Delete Task Execution

The task executions endpoint lets you clean up resources used to deploy the task.

The cleanup implementation is platform specific.

The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
DELETE /tasks/executions/2 HTTP/1.1
Host: localhost:9393
```

Table 17. /tasks/executions/{id}

Parameter	Description
`id`	The id of an existing task execution (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/tasks/executions/2' -i -X DELETE
```

Response Structure

```
HTTP/1.1 200 OK
```

59.11.6. Task Execution Current Count

The task executions current endpoint lets you retrieve the current number of running executions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /tasks/executions/current HTTP/1.1
Host: localhost:9393
```

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/tasks/executions/current' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 65
Content-Type: application/hal+json;charset=UTF-8

{
  "maximumTaskExecutions" : 20,
  "runningExecutionCount" : 1
}
```

59.12. Job Executions

The job executions endpoint provides information about the job executions that are registered with the Spring Cloud Data Flow server. The following topics provide more detail:

- List All Job Executions
- [api-guide-resources-job-executions-job-execution-info-only-list]
- [api-guide-resources-job-executions-list-by-name]
- Job Execution Detail
- Stop Job Execution
- Restart Job Execution

59.12.1. List All Job Executions

The job executions endpoint lets you list all job executions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- [api-guide-resources-job-executions-list-response-structure]

Request Structure

```
GET /jobs/executions?page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)

Example Request

```
$ curl 'http://localhost:9393/jobs/executions?page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 3357
Content-Type: application/json;charset=UTF-8

{
  "_embedded" : {
    "jobExecutionResourceList" : [ {
      "executionId" : 2,
      "stepExecutionCount" : 0,
      "jobId" : 2,
      "taskExecutionId" : 2,
      "name" : "DOCJOB_1",
      "startDate" : "2019-01-10",
      "startTime" : "21:30:17",
      "duration" : "00:00:00",
      "jobExecution" : {
        "id" : 2,
        "version" : 1,
        "jobParameters" : {
          "parameters" : { },
          "empty" : true
        },
        "jobInstance" : {
          "id" : 2,
          "version" : null,
          "jobName" : "DOCJOB_1",
```

```
        "instanceId" : 2
      },
      "stepExecutions" : [ ],
      "status" : "STOPPED",
      "startTime" : "2019-01-10T21:30:17.566Z",
      "createTime" : "2019-01-10T21:30:17.565Z",
      "endTime" : null,
      "lastUpdated" : "2019-01-10T21:30:17.566Z",
      "exitStatus" : {
        "exitCode" : "UNKNOWN",
        "exitDescription" : "",
        "running" : true
      },
      "executionContext" : {
        "dirty" : false,
        "empty" : true,
        "values" : [ ]
      },
      "failureExceptions" : [ ],
      "jobConfigurationName" : null,
      "running" : true,
      "jobId" : 2,
      "stopping" : false,
      "allFailureExceptions" : [ ]
    },
    "jobParameters" : { },
    "jobParametersString" : "",
    "restartable" : true,
    "abandonable" : true,
    "stoppable" : false,
    "defined" : true,
    "timeZone" : "UTC",
    "_links" : {
      "self" : {
        "href" : "http://localhost:9393/jobs/executions/2"
      }
    }
  }, {
    "executionId" : 1,
    "stepExecutionCount" : 0,
    "jobId" : 1,
    "taskExecutionId" : 1,
    "name" : "DOCJOB",
    "startDate" : "2019-01-10",
    "startTime" : "21:30:17",
    "duration" : "00:00:00",
    "jobExecution" : {
      "id" : 1,
      "version" : 2,
      "jobParameters" : {
        "parameters" : { },
        "empty" : true
      },
      "jobInstance" : {
        "id" : 1,
        "version" : null,
        "jobName" : "DOCJOB",
        "instanceId" : 1
      },
      "stepExecutions" : [ ],
      "status" : "STOPPING",
      "startTime" : "2019-01-10T21:30:17.562Z",
      "createTime" : "2019-01-10T21:30:17.560Z",
      "endTime" : null,
      "lastUpdated" : "2019-01-10T21:30:17.627Z",
      "exitStatus" : {
        "exitCode" : "UNKNOWN",
        "exitDescription" : "",
        "running" : true
      },
      "executionContext" : {
        "dirty" : false,
        "empty" : true,
        "values" : [ ]
      },
      "failureExceptions" : [ ],
      "jobConfigurationName" : null,
      "running" : true,
      "jobId" : 1,
      "stopping" : true,
      "allFailureExceptions" : [ ]
    },
    "jobParameters" : { },
    "jobParametersString" : "",
    "restartable" : false,
    "abandonable" : true,
    "stoppable" : false,
    "defined" : false,
    "timeZone" : "UTC",
    "_links" : {
      "self" : {
        "href" : "http://localhost:9393/jobs/executions/1"
      }
    }
  } ]
},
"_links" : {
  "self" : {
    "href" : "http://localhost:9393/jobs/executions?page=0&size=10"
  }
}
```

```
  },
  "page" : {
    "size" : 10,
    "totalElements" : 2,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.12.2. List All Job Executions Without Step Executions Included

The job executions endpoint lets you list all job executions without step executions included. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /jobs/thinexecutions?page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)

Example Request

```
$ curl 'http://localhost:9393/jobs/thinexecutions?page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 1597
Content-Type: application/json;charset=UTF-8

{
  "_embedded" : {
    "jobExecutionThinResourceList" : [ {
      "executionId" : 2,
      "stepExecutionCount" : 0,
      "jobId" : 2,
      "taskExecutionId" : 2,
      "instanceId" : 2,
      "name" : "DOCJOB_1",
      "startDate" : "2019-01-10",
      "startTime" : "21:30:17",
      "startDateTime" : "2019-01-10T21:30:17.566Z",
      "duration" : "00:00:00",
      "jobParameters" : { },
      "jobParametersString" : "",
      "restartable" : true,
      "abandonable" : true,
      "stoppable" : false,
      "defined" : true,
      "timeZone" : "UTC",
      "status" : "STOPPED",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/jobs/thinexecutions/2"
        }
      }
    }, {
      "executionId" : 1,
      "stepExecutionCount" : 0,
      "jobId" : 1,
      "taskExecutionId" : 1,
      "instanceId" : 1,
      "name" : "DOCJOB",
      "startDate" : "2019-01-10",
      "startTime" : "21:30:17",
      "startDateTime" : "2019-01-10T21:30:17.562Z",
      "duration" : "00:00:00",
      "jobParameters" : { },
      "jobParametersString" : "",
      "restartable" : false,
      "abandonable" : false,
      "stoppable" : true,
      "defined" : false,
      "timeZone" : "UTC",
      "status" : "STARTED",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/jobs/thinexecutions/1"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/thinexecutions?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 2,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.12.3. List All Job Executions With a Specified Job Name

The job executions endpoint lets you list all job executions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /jobs/executions?name=DOCJOB&page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)

Parameter	Description
name	The name associated with the job execution

Example Request

```
$ curl 'http://localhost:9393/jobs/executions?name=DOCJOB&page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 1813
Content-Type: application/json;charset=UTF-8

{
  "_embedded" : {
    "jobExecutionResourceList" : [ {
      "executionId" : 1,
      "stepExecutionCount" : 0,
      "jobId" : 1,
      "taskExecutionId" : 1,
      "name" : "DOCJOB",
      "startDate" : "2019-01-10",
      "startTime" : "21:30:17",
      "duration" : "00:00:00",
      "jobExecution" : {
        "id" : 1,
        "version" : 2,
        "jobParameters" : {
          "parameters" : { },
          "empty" : true
        },
        "jobInstance" : {
          "id" : 1,
          "version" : null,
          "jobName" : "DOCJOB",
          "instanceId" : 1
        },
        "stepExecutions" : [ ],
        "status" : "STOPPING",
        "startTime" : "2019-01-10T21:30:17.562Z",
        "createTime" : "2019-01-10T21:30:17.560Z",
        "endTime" : null,
        "lastUpdated" : "2019-01-10T21:30:17.627Z",
        "exitStatus" : {
          "exitCode" : "UNKNOWN",
          "exitDescription" : "",
          "running" : true
        },
        "executionContext" : {
          "dirty" : false,
          "empty" : true,
          "values" : [ ]
        },
        "failureExceptions" : [ ],
        "jobConfigurationName" : null,
        "running" : true,
        "jobId" : 1,
        "stopping" : true,
        "allFailureExceptions" : [ ]
      },
      "jobParameters" : { },
      "jobParametersString" : "",
      "restartable" : false,
      "abandonable" : true,
      "stoppable" : false,
      "defined" : false,
      "timeZone" : "UTC",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/jobs/executions/1"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/executions?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.12.4. List All Job Executions With a Specified Job Name Without Step Executions Included

The job executions endpoint lets you list all job executions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request

- Response Structure

Request Structure

```
GET /jobs/thinexecutions?name=DOCJOB&page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
page	The zero-based page number (optional)
size	The requested page size (optional)
name	The name associated with the job execution

Example Request

```
$ curl 'http://localhost:9393/jobs/thinexecutions?name=DOCJOB&page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 939
Content-Type: application/json;charset=UTF-8

{
  "_embedded" : {
    "jobExecutionThinResourceList" : [ {
      "executionId" : 1,
      "stepExecutionCount" : 0,
      "jobId" : 1,
      "taskExecutionId" : 1,
      "instanceId" : 1,
      "name" : "DOCJOB",
      "startDate" : "2019-01-10",
      "startTime" : "21:30:17",
      "startDateTime" : "2019-01-10T21:30:17.562Z",
      "duration" : "00:00:00",
      "jobParameters" : { },
      "jobParametersString" : "",
      "restartable" : false,
      "abandonable" : true,
      "stoppable" : false,
      "defined" : false,
      "timeZone" : "UTC",
      "status" : "STOPPING",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/jobs/thinexecutions/1"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/thinexecutions?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.12.5. Job Execution Detail

The job executions endpoint lets you get the details about a job execution. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /jobs/executions/2 HTTP/1.1
Host: localhost:9393
```

Table 18. /jobs/executions/{id}

Parameter	Description
id	The id of an existing job execution (required)

Parameter	Description

Request Parameters

There are no request parameter for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/jobs/executions/2' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 1311
Content-Type: application/json;charset=UTF-8

{
  "executionId" : 2,
  "stepExecutionCount" : 0,
  "jobId" : 2,
  "taskExecutionId" : 2,
  "name" : "DOCJOB_1",
  "startDate" : "2019-01-10",
  "startTime" : "21:30:17",
  "duration" : "00:00:00",
  "jobExecution" : {
    "id" : 2,
    "version" : 1,
    "jobParameters" : {
      "parameters" : { },
      "empty" : true
    },
    "jobInstance" : {
      "id" : 2,
      "version" : 0,
      "jobName" : "DOCJOB_1",
      "instanceId" : 2
    },
    "stepExecutions" : [ ],
    "status" : "STOPPED",
    "startTime" : "2019-01-10T21:30:17.566Z",
    "createTime" : "2019-01-10T21:30:17.565Z",
    "endTime" : null,
    "lastUpdated" : "2019-01-10T21:30:17.566Z",
    "exitStatus" : {
      "exitCode" : "UNKNOWN",
      "exitDescription" : "",
      "running" : true
    },
    "executionContext" : {
      "dirty" : false,
      "empty" : true,
      "values" : [ ]
    },
    "failureExceptions" : [ ],
    "jobConfigurationName" : null,
    "running" : true,
    "jobId" : 2,
    "stopping" : false,
    "allFailureExceptions" : [ ]
  },
  "jobParameters" : { },
  "jobParametersString" : "",
  "restartable" : true,
  "abandonable" : true,
  "stoppable" : false,
  "defined" : true,
  "timeZone" : "UTC",
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/executions/2"
    }
  }
}
```

59.12.6. Stop Job Execution

The job executions endpoint lets you stop a job execution. The following topics provide more detail:

- Request structure
- Request parameters
- Example request
- Response structure

Request structure

```
PUT /jobs/executions/1 HTTP/1.1
Accept: application/json
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

stop=true
```

Table 19. /jobs/executions/{id}

Parameter	Description
`id`	The id of an existing job execution (required)

Request parameters

Parameter	Description
`stop`	Sends signal to stop the job if set to true

Example request

```
$ curl 'http://localhost:9393/jobs/executions/1' -i -X PUT \
    -H 'Accept: application/json' \
    -d 'stop=true'
```

Response structure

```
HTTP/1.1 200 OK
```

59.12.7. Restart Job Execution

The job executions endpoint lets you restart a job execution. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
PUT /jobs/executions/2 HTTP/1.1
Accept: application/json
Host: localhost:9393
Content-Type: application/x-www-form-urlencoded

restart=true
```

Table 20. /jobs/executions/{id}

Parameter	Description
`id`	The id of an existing job execution (required)

Request Parameters

Parameter	Description
`restart`	Sends signal to restart the job if set to true

Example Request

```
$ curl 'http://localhost:9393/jobs/executions/2' -i -X PUT \
    -H 'Accept: application/json' \
    -d 'restart=true'
```

Response Structure

```
HTTP/1.1 200 OK
```

59.13. Job Instances

The job instances endpoint provides information about the job instances that are registered with the Spring Cloud Data Flow server. The following topics provide more detail:

- List All Job Instances
- Job Instance Detail

59.13.1. List All Job Instances

The job instances endpoint lets you list all job instances. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request

- Response Structure

Request Structure

```
GET /jobs/instances?name=DOCJOB&page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
page	The zero-based page number (optional)
size	The requested page size (optional)
name	The name associated with the job instance

Example Request

```
$ curl 'http://localhost:9393/jobs/instances?name=DOCJOB&page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 2002
Content-Type: application/hal+json;charset=UTF-8

{
  "_embedded" : {
    "jobInstanceResourceList" : [ {
      "jobName" : "DOCJOB",
      "jobInstanceId" : 1,
      "jobExecutions" : [ {
        "executionId" : 1,
        "stepExecutionCount" : 0,
        "jobId" : 1,
        "taskExecutionId" : 1,
        "name" : "DOCJOB",
        "startDate" : "2019-01-10",
        "startTime" : "21:30:15",
        "duration" : "00:00:00",
        "jobExecution" : {
          "id" : 1,
          "version" : 1,
          "jobParameters" : {
            "parameters" : { },
            "empty" : true
          },
          "jobInstance" : {
            "id" : 1,
            "version" : 0,
            "jobName" : "DOCJOB",
            "instanceId" : 1
          },
          "stepExecutions" : [ ],
          "status" : "STARTED",
          "startTime" : "2019-01-10T21:30:15.670Z",
          "createTime" : "2019-01-10T21:30:15.668Z",
          "endTime" : null,
          "lastUpdated" : "2019-01-10T21:30:15.670Z",
          "exitStatus" : {
            "exitCode" : "UNKNOWN",
            "exitDescription" : "",
            "running" : true
          },
          "executionContext" : {
            "dirty" : false,
            "empty" : true,
            "values" : [ ]
          },
          "failureExceptions" : [ ],
          "jobConfigurationName" : null,
          "running" : true,
          "jobId" : 1,
          "stopping" : false,
          "allFailureExceptions" : [ ]
        },
        "jobParameters" : { },
        "jobParametersString" : "",
        "restartable" : false,
        "abandonable" : false,
        "stoppable" : true,
        "defined" : false,
        "timeZone" : "UTC"
      } ],
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/jobs/instances/1"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/instances?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.13.2. Job Instance Detail

The job instances endpoint lets you list all job instances. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /jobs/instances/1 HTTP/1.1
Host: localhost:9393
```

Table 21. /jobs/instances/{id}

Parameter	Description
`id`	The id of an existing job instance (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/jobs/instances/1' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 1487

{
  "jobName" : "DOCJOB",
  "jobInstanceId" : 1,
  "jobExecutions" : [ {
    "executionId" : 1,
    "stepExecutionCount" : 0,
    "jobId" : 1,
    "taskExecutionId" : 1,
    "name" : "DOCJOB",
    "startDate" : "2019-01-10",
    "startTime" : "21:30:15",
    "duration" : "00:00:00",
    "jobExecution" : {
      "id" : 1,
      "version" : 1,
      "jobParameters" : {
        "parameters" : { },
        "empty" : true
      },
      "jobInstance" : {
        "id" : 1,
        "version" : 0,
        "jobName" : "DOCJOB",
        "instanceId" : 1
      },
      "stepExecutions" : [ ],
      "status" : "STARTED",
      "startTime" : "2019-01-10T21:30:15.670Z",
      "createTime" : "2019-01-10T21:30:15.668Z",
      "endTime" : null,
      "lastUpdated" : "2019-01-10T21:30:15.670Z",
      "exitStatus" : {
        "exitCode" : "UNKNOWN",
        "exitDescription" : "",
        "running" : true
      },
      "executionContext" : {
        "dirty" : false,
        "empty" : true,
        "values" : [ ]
      },
      "failureExceptions" : [ ],
      "jobConfigurationName" : null,
      "running" : true,
      "jobId" : 1,
      "stopping" : false,
      "allFailureExceptions" : [ ]
    },
    "jobParameters" : { },
    "jobParametersString" : "",
    "restartable" : false,
    "abandonable" : false,
    "stoppable" : true,
    "defined" : false,
    "timeZone" : "UTC"
  } ],
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/instances/1"
    }
  }
}
```

59.14. Job Step Executions

The job step executions endpoint provides information about the job step executions that are registered with the Spring Cloud Data Flow server. The following topics provide more detail:

- List All Step Executions For a Job Execution

- Job Step Execution Detail
- Job Step Execution Progress

59.14.1. List All Step Executions For a Job Execution

The job step executions endpoint lets you list all job step executions. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /jobs/executions/1/steps?page=0&size=10 HTTP/1.1
Host: localhost:9393
```

Request Parameters

Parameter	Description
`page`	The zero-based page number (optional)
`size`	The requested page size (optional)

Example Request

```
$ curl 'http://localhost:9393/jobs/executions/1/steps?page=0&size=10' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 1669

{
  "_embedded" : {
    "stepExecutionResourceList" : [ {
      "jobExecutionId" : 1,
      "stepExecution" : {
        "id" : 1,
        "version" : 0,
        "stepName" : "DOCJOB_STEP",
        "status" : "STARTING",
        "readCount" : 0,
        "writeCount" : 0,
        "commitCount" : 0,
        "rollbackCount" : 0,
        "readSkipCount" : 0,
        "processSkipCount" : 0,
        "writeSkipCount" : 0,
        "startTime" : "2019-01-10T21:30:10.077Z",
        "endTime" : null,
        "lastUpdated" : "2019-01-10T21:30:10.077Z",
        "executionContext" : {
          "dirty" : false,
          "empty" : true,
          "values" : [ ]
        },
        "exitStatus" : {
          "exitCode" : "EXECUTING",
          "exitDescription" : "",
          "running" : true
        },
        "terminateOnly" : false,
        "filterCount" : 0,
        "failureExceptions" : [ ],
        "jobParameters" : {
          "parameters" : { },
          "empty" : true
        },
        "jobExecutionId" : 1,
        "skipCount" : 0,
        "summary" : "StepExecution: id=1, version=0, name=DOCJOB_STEP, status=STARTING, exitStatus=EXECUTING, readCount=0,
      },
      "stepType" : "",
      "_links" : {
        "self" : {
          "href" : "http://localhost:9393/jobs/executions/1/steps/1"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/executions/1/steps?page=0&size=10"
    }
  },
  "page" : {
    "size" : 10,
    "totalElements" : 1,
    "totalPages" : 1,
    "number" : 0
  }
}
```

59.14.2. Job Step Execution Detail

The job step executions endpoint lets you get details about a job step execution. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /jobs/executions/1/steps/1 HTTP/1.1
Host: localhost:9393
```

Table 22. /jobs/executions/{id}/steps/{stepid}

Parameter	Description
`id`	The id of an existing job execution (required)
`stepid`	The id of an existing step execution for a specific job execution (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/jobs/executions/1/steps/1' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 1211

{
  "jobExecutionId" : 1,
  "stepExecution" : {
    "id" : 1,
    "version" : 0,
    "stepName" : "DOCJOB_STEP",
    "status" : "STARTING",
    "readCount" : 0,
    "writeCount" : 0,
    "commitCount" : 0,
    "rollbackCount" : 0,
    "readSkipCount" : 0,
    "processSkipCount" : 0,
    "writeSkipCount" : 0,
    "startTime" : "2019-01-10T21:30:10.077Z",
    "endTime" : null,
    "lastUpdated" : "2019-01-10T21:30:10.077Z",
    "executionContext" : {
      "dirty" : false,
      "empty" : true,
      "values" : [ ]
    },
    "exitStatus" : {
      "exitCode" : "EXECUTING",
      "exitDescription" : "",
      "running" : true
    },
    "terminateOnly" : false,
    "filterCount" : 0,
    "failureExceptions" : [ ],
    "jobParameters" : {
      "parameters" : { },
      "empty" : true
    },
    "jobExecutionId" : 1,
    "skipCount" : 0,
    "summary" : "StepExecution: id=1, version=0, name=DOCJOB_STEP, status=STARTING, exitStatus=EXECUTING, readCount=0, filt
  },
  "stepType" : "",
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/executions/1/steps/1"
    }
  }
}
```

59.14.3. Job Step Execution Progress

The job step executions endpoint lets you get details about the progress of a job step execution. The following topics provide more detail:

- Request Structure
- Request Parameters
- Example Request
- Response Structure

Request Structure

```
GET /jobs/executions/1/steps/1/progress HTTP/1.1
Host: localhost:9393
```

Table 23. /jobs/executions/{id}/steps/{stepid}/progress

Parameter	Description
`id`	The id of an existing job execution (required)
`stepid`	The id of an existing step execution for a specific job execution (required)

Request Parameters

There are no request parameters for this endpoint.

Example Request

```
$ curl 'http://localhost:9393/jobs/executions/1/steps/1/progress' -i -X GET
```

Response Structure

```
HTTP/1.1 200 OK
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 2714

{
  "stepExecution" : {
    "id" : 1,
    "version" : 0,
    "stepName" : "DOCJOB_STEP",
    "status" : "STARTING",
    "readCount" : 0,
    "writeCount" : 0,
    "commitCount" : 0,
    "rollbackCount" : 0,
    "readSkipCount" : 0,
    "processSkipCount" : 0,
    "writeSkipCount" : 0,
    "startTime" : "2019-01-10T21:30:10.077Z",
    "endTime" : null,
    "lastUpdated" : "2019-01-10T21:30:10.077Z",
    "executionContext" : {
      "dirty" : false,
      "empty" : true,
      "values" : [ ]
    },
    "exitStatus" : {
      "exitCode" : "EXECUTING",
      "exitDescription" : "",
      "running" : true
    },
    "terminateOnly" : false,
    "filterCount" : 0,
    "failureExceptions" : [ ],
    "jobParameters" : {
      "parameters" : { },
      "empty" : true
    },
    "jobExecutionId" : 1,
    "skipCount" : 0,
    "summary" : "StepExecution: id=1, version=0, name=DOCJOB_STEP, status=STARTING, exitStatus=EXECUTING, readCount=0, filt
  },
  "stepExecutionHistory" : {
    "stepName" : "DOCJOB_STEP",
    "count" : 0,
    "commitCount" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    },
    "rollbackCount" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    },
    "readCount" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    },
    "writeCount" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    },
    "filterCount" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    },
    "readSkipCount" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    },
    "writeSkipCount" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
```

```
    },
    "processSkipCount" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    },
    "duration" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    },
    "durationPerRead" : {
      "count" : 0,
      "min" : 0.0,
      "max" : 0.0,
      "mean" : 0.0,
      "standardDeviation" : 0.0
    }
  },
  "percentageComplete" : 0.5,
  "finished" : false,
  "duration" : 126.0,
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/jobs/executions/1/steps/1"
    }
  }
}
```

59.15. Runtime Information about Applications

It is possible to get information about running apps known to the system, either globally or individually. The following topics provide more detail:

- Listing All Applications at Runtime
- Querying All Instances of a Single App
- Querying a Single Instance of a Single App

59.15.1. Listing All Applications at Runtime

To retrieve information about all instances of all apps, query the `/runtime/apps` endpoint by using `GET`. The following topics provide more detail:

- Request Structure
- Example Request
- Response Structure

Request Structure

```
GET /runtime/apps HTTP/1.1
Accept: application/json
Host: localhost:9393
```

Example Request

```
$ curl 'http://localhost:9393/runtime/apps' -i -X GET \
    -H 'Accept: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 209
Content-Type: application/json;charset=UTF-8

{
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/runtime/apps?page=0&size=20"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 0,
    "totalPages" : 0,
    "number" : 0
  }
}
```

59.15.2. Querying All Instances of a Single App

To retrieve information about all instances of a particular app, query the `/runtime/apps/<appId>/instances` endpoint by using `GET`. The following topics provide more detail:

- Request Structure
- Example Request
- Response Structure

Request Structure

```
GET /runtime/apps HTTP/1.1
Accept: application/json
Host: localhost:9393
```

Example Request

```
$ curl 'http://localhost:9393/runtime/apps' -i -X GET \
    -H 'Accept: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 209
Content-Type: application/json;charset=UTF-8

{
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/runtime/apps?page=0&size=20"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 0,
    "totalPages" : 0,
    "number" : 0
  }
}
```

59.15.3. Querying a Single Instance of a Single App

Finally, to retrieve information about a particular instance of a particular app, query the `/runtime/apps/<appId>/instances/<instanceId>` endpoint by using `GET`. The following topics provide more detail:

- Request Structure
- Example Request
- Response Structure

Request Structure

```
GET /runtime/apps HTTP/1.1
Accept: application/json
Host: localhost:9393
```

Example Request

```
$ curl 'http://localhost:9393/runtime/apps' -i -X GET \
    -H 'Accept: application/json'
```

Response Structure

```
HTTP/1.1 200 OK
Content-Length: 209
Content-Type: application/json;charset=UTF-8

{
  "_links" : {
    "self" : {
      "href" : "http://localhost:9393/runtime/apps?page=0&size=20"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 0,
    "totalPages" : 0,
    "number" : 0
  }
}
```

59.16. Metrics for Stream Applications

This REST endpoint exposes metrics for stream applications. It requires theMetrics Collector application to be running as a separate service. If it is not running, this endpoint returns an empty response.

In order to learn more about the Metrics Collector, please also refer to the “[configuration-monitoring-deployed-applications]” chapter.

The following topics provide more detail:

- Request Structure
- [api-guide-resources-metrics-stream-applications-example-request]
- [api-guide-resources-metrics-stream-applications-response-structure]
- [api-guide-resources-metrics-stream-applications-example-response]

59.16.1. Request Structure

For example, a typical request might resemble the following:

Unresolved directive in api-guide.adoc - include::/opt/bamboo-home/xml-data/build-dir/SCD-BMASTER-JOB1/spring-cloud-dataflow-docs/../spring-cloud-dataflow-classic-docs/target/generated-snippets/metrics-for-stream-apps-without-collector-documentation/get-metrics-without-collector-running/http-request.adoc[]

59.16.2. Example Request

Unresolved directive in api-guide.adoc - include::/opt/bamboo-home/xml-data/build-dir/SCD-BMASTER-JOB1/spring-cloud-dataflow-docs/../spring-cloud-dataflow-classic-docs/target/generated-snippets/metrics-for-stream-apps-without-collector-documentation/get-metrics-without-collector-running/curl-request.adoc[]

59.16.3. Response Structure

This REST endpoint uses Hystrix through the Spring Cloud Netflix project under the covers to make a proxy HTTP request to the Metrics Collector.

59.16.4. Example Response

The endpoint does not generate an error when the Metrics Collector is not running. Rather, it gracefully degrades and returns an empty response such as the following:

Unresolved directive in api-guide.adoc - include::/opt/bamboo-home/xml-data/build-dir/SCD-BMASTER-JOB1/spring-cloud-dataflow-docs/../spring-cloud-dataflow-classic-docs/target/generated-snippets/metrics-for-stream-apps-without-collector-documentation/get-metrics-without-collector-running/http-response.adoc[]

However, if metrics are being collected and the Metrics Collector is running, you should see a response similar to the listing below. The metrics data returned in the listing below is based on the example stream definition created in the "[configuration-monitoring-deployed-applications]" chapter, where we created `time | log` with two instances of each application deployed.

Unresolved directive in api-guide.adoc - include::/opt/bamboo-home/xml-data/build-dir/SCD-BMASTER-JOB1/spring-cloud-dataflow-docs/../spring-cloud-dataflow-classic-docs/target/generated-snippets/metrics-for-stream-apps-documentation/get-metrics-with-collector-running/http-response.adoc[]

Appendices

Having trouble with Spring Cloud Data Flow, We'd like to help!

- Ask a question - we monitor stackoverflow.com for questions tagged with `spring-cloud-dataflow`.
- Report bugs with Spring Cloud Data Flow at github.com/spring-cloud/spring-cloud-dataflow/issues.

Appendix A: Data Flow Template

As described in the previous chapter, Spring Cloud Data Flow's functionality is completely exposed through REST endpoints. While you can use those endpoints directly, Spring Cloud Data Flow also provides a Java-based API, which makes using those REST endpoints even easier.

The central entry point is the `DataFlowTemplate` class in the `org.springframework.cloud.dataflow.rest.client` package.

This class implements the `DataFlowOperations` interface and delegates to the following sub-templates that provide the specific functionality for each feature-set:

Interface	Description
`StreamOperations`	REST client for stream operations
`CounterOperations`	REST client for counter operations
`FieldValueCounterOperations`	REST client for field value counter operations
`AggregateCounterOperations`	REST client for aggregate counter operations

Interface	Description
TaskOperations	REST client for task operations
JobOperations	REST client for job operations
AppRegistryOperations	REST client for app registry operations
CompletionOperations	REST client for completion operations
RuntimeOperations	REST Client for runtime operations

When the `DataFlowTemplate` is being initialized, the sub-templates can be discovered through the REST relations, which are provided by HATEOAS.[1]

If a resource cannot be resolved, the respective sub-template results in NULL. A common cause is that Spring Cloud Data Flow offers for specific sets of features to be enabled/disabled when launching. For more information, see "[enable-disable-specific-features]".

A.1. Using the Data Flow Template

When you use the Data Flow Template, the only needed Data Flow dependency is the Spring Cloud Data Flow Rest Client, as shown in the following Maven snippet:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-dataflow-rest-client</artifactId>
  <version>2.0.0.BUILD-SNAPSHOT</version>
</dependency>
```

With that dependency, you get the `DataFlowTemplate` class as well as all the dependencies needed to make calls to a Spring Cloud Data Flow server.

When instantiating the `DataFlowTemplate`, you also pass in a `RestTemplate`. Please be aware that the needed `RestTemplate` requires some additional configuration to be valid in the context of the `DataFlowTemplate`. When declaring a `RestTemplate` as a bean, the following configuration suffices:

```
  @Bean
  public static RestTemplate restTemplate() {
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.setErrorHandler(new VndErrorResponseErrorHandler(restTemplate.getMessageConverters()));
    for(HttpMessageConverter<?> converter : restTemplate.getMessageConverters()) {
      if (converter instanceof MappingJackson2HttpMessageConverter) {
        final MappingJackson2HttpMessageConverter jacksonConverter =
            (MappingJackson2HttpMessageConverter) converter;
        jacksonConverter.getObjectMapper()
            .registerModule(new Jackson2HalModule())
            .addMixIn(JobExecution.class, JobExecutionJacksonMixIn.class)
            .addMixIn(JobParameters.class, JobParametersJacksonMixIn.class)
            .addMixIn(JobParameter.class, JobParameterJacksonMixIn.class)
            .addMixIn(JobInstance.class, JobInstanceJacksonMixIn.class)
            .addMixIn(ExitStatus.class, ExitStatusJacksonMixIn.class)
            .addMixIn(StepExecution.class, StepExecutionJacksonMixIn.class)
            .addMixIn(ExecutionContext.class, ExecutionContextJacksonMixIn.class)
            .addMixIn(StepExecutionHistory.class, StepExecutionHistoryJacksonMixIn.class);
      }
    }
    return restTemplate;
  }
```

Now you can instantiate the `DataFlowTemplate` with the following code:

```
DataFlowTemplate dataFlowTemplate = new DataFlowTemplate(
    new URI("http://localhost:9393/"), restTemplate);          (1)
```

(1) The `URI` points to the ROOT of your Spring Cloud Data Flow Server.

Depending on your requirements, you can now make calls to the server. For instance, if you want to get a list of currently available applications you can run the following code:

```
PagedResources<AppRegistrationResource> apps = dataFlowTemplate.appRegistryOperations().list();

System.out.println(String.format("Retrieved %s application(s)",
    apps.getContent().size()));

for (AppRegistrationResource app : apps.getContent()) {
  System.out.println(String.format("App Name: %s, App Type: %s, App URI: %s",
    app.getName(),
    app.getType(),
    app.getUri()));
}
```

Appendix B: “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 do not cover here, you might want to check outstackoverflow.com to see if someone has already provided an answer. That is also a great place to ask new questions (please use the `spring-cloud-dataflow` tag).

We are also more than happy to extend this section. If you want to add a “how-to”, you can send us apull request.

B.1. Configure Maven Properties

You can set the maven properties such as local maven repository location, remote maven repositories, authentication credentials, and proxy server properties through command line properties when starting the Data Flow server. Alternatively, you can set the properites using `SPRING_APPLICATION_JSON` environment property for the Data Flow server.

The remote maven repositories need to be configured explicitly if the apps are resolved using maven repository, except for a `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, run the server with a command similar to the following:

```
$ java -jar <dataflow-server>.jar --maven.localRepository=mylocal
--maven.remote-repositories.repo1.url=https://repo1
--maven.remote-repositories.repo1.auth.username=repo1user
--maven.remote-repositories.repo1.auth.password=repo1pass
--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
```

You can also use the `SPRING_APPLICATION_JSON` environment property:

```
export SPRING_APPLICATION_JSON='{ "maven": { "local-repository": "local","remote-repositories": { "repo1": { "url":
"https://repo1", "auth": { "username": "repo1user", "password": "repo1pass" } },
"repo2": { "url": "https://repo2" } }, "proxy": { "host": "proxyhost", "port": 9018, "auth": { "username": "proxyuser",
"password": "proxypass" } } } }'
```

Here is the same content in nicely formatted JSON:

```
SPRING_APPLICATION_JSON='{
  "maven": {
    "local-repository": "local",
    "remote-repositories": {
      "repo1": {
        "url": "https://repo1",
        "auth": {
          "username": "repo1user",
          "password": "repo1pass"
        }
      },
      "repo2": {
        "url": "https://repo2"
      }
    },
    "proxy": {
      "host": "proxyhost",
      "port": 9018,
      "auth": {
        "username": "proxyuser",
        "password": "proxypass"
      }
    }
  }
}'
```

Depending on the Spring Cloud Data Flow server implementation, you may have to pass the environment properties by using the platform specific environment-setting capabilities. For instance, in Cloud Foundry, you would pass them as `cf set-env SPRING_APPLICATION_JSON` .

B.2. 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, the two primary areas where enabling the DEBUG logs could be useful are

- Deployment Logs
- Application Logs

B.2.1. 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 network errors, it would be useful to enable the DEBUG logs at the underlying deployer and the libraries used by it.

To enable DEBUG logs for the local-deployer, start the server as follows:

```
$ 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.)

To enable DEBUG logs for the cloudfoundry-deployer, set the following environment variable and, after restaging the dataflow server, you can see more logs around request and response and see detailed stack traces for failures. The cloudfoundry deployer uses cf-java-client, so you must also 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.)

To review Reactor logs, which are used by the `cf-java-client`, then the following commad 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.)

Similar to the `local-deployer` and `cloudfoundry-deployer` options as discussed above, there are equivalent settings available for Kubernetes. See the respective SPI implementations for more detail about the packages to configure for logging.

B.2.2. Application Logs

The streaming applications in Spring Cloud Data Flow are Spring Cloud Stream applications, which are in turn based on Spring Boot. They can be independently setup with logging configurations.

For instance, if you must troubleshoot the `header` and `payload` specifics that are being passed around source, processor, and sink channels, you should deploy 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 with `deployment` properties when deploying the stream, as follows:

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

B.2.3. Remote Debugging

The Data Flow local server lets you debug the deployed applications. This is accomplished by enabling the remote debugging feature of the JVM through deployment properites, as shown in the following example:

```
stream deploy --name mystream --properties "deployer.fooApp.local.debugPort=9999"
```

The preceding example starts the `fooApp` application in debug mode, allowing a remote debugger to be attached on port 9999. By default, the application starts in a 'suspend' mode and waits for the remote debug session to be attached (started). Otherwise, you can provide an additional `debugSuspend` property with value `n`.

Also, when there is more then one instance of the application, the debug port for each instance is the value of `debugPort` + instanceId.

Unlike other properties you must NOT use a wildcard for the application name, since each application must use a unique debug port.

B.2.4. Log Redirect

Given that each application is a separate process that maintains its own set of logs, accessing individual logs could be a bit inconvenient, especially in the early stages of development, when logs are accessed more often. Since it is also a common pattern to rely on a local SCDF Server that deploys each application as a local JVM process, you can redirect the stdout and stdin from the deployed applications to the parent process. Thus, with a local SCDF Server, the application logs appear in the logs of the running local SCDF Server.

Typically when you deploy the stream, you see something resembling the following in the server logs:

```
017-06-28 09:50:16.372  INFO 41161 --- [nio-9393-exec-7] o.s.c.d.spi.local.LocalAppDeployer       : Deploying app with
deploymentId mystream.myapp instance 0.
   Logs will be in /var/folders/l2/63gcnd9d7g5dxxpjbgr0trpw0000gn/T/spring-cloud-dataflow-5939494818997196225/mystream-
1498661416369/mystream.myapp
```

However, by setting `local.inheritLogging=true` as a deployment property, you can see the following:

```
017-06-28 09:50:16.372  INFO 41161 --- [nio-9393-exec-7] o.s.c.d.spi.local.LocalAppDeployer       : Deploying app with
deploymentId mystream.myapp instance 0.
   Logs will be inherited.
```

After that, the application logs appear alongside the server logs, as shown in the following example:

```
stream deploy --name mystream --properties "deployer.*.local.inheritLogging=true”
```

The preceding stream definition enables log redirection for each application in the stream. The following stream definition enables log redirection for only the application named ‘my app’.

```
stream deploy --name mystream --properties "deployer.myapp.local.inheritLogging=true”
```

Log redirect is only supported with local-deployer.

B.3. Frequently Asked Questions

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

B.3.1. Advanced SpEL Expressions

One of the powerful features of SpEL expressions isfunctions. If the appropriate libraries are in the classpath, Spring Integration provides the `jsonPath()` and `xpath()` SpEL-functions. All the provided Spring Cloud Stream application starters are have with the `json-path` and `spring-integration-xml` in their uber-jar. Consequently, 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 by using a few `jsonPath()` expressions, as follows:

```
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 the preceding example, we apply `jsonPath` for the incoming payload to extract 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, as follows:

```
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. However, for this purpose, you should build a library with a `@Configuration` class containing an appropriate `SpelFunctionFactoryBean` `@Bean` definition. The target Spring Cloud Stream application starter should be repackaged to supply such a custom extension with a built-in Spring Boot `@ComponentScan` mechanism or auto-configuration hook.

B.3.2. 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. If it is not set, by default, the application

expects to use a table with a name of `messages`. To alter this behavior, the JDBC sink acceptsseveral options that you can pass by using the --param=value notation in the stream or change globally. The JDBC sink has a `jdbc.initialize` property that, if set to `true`, results in the sink creating a table based on the specified configuration when it starts. If that initialize property is `false`, which is the default, you must make sure that the table to use is already available.

A stream definition using `jdbc` sink and relying on all defaults with MySQL as the backing database looks like the following example:

```
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
```

In the preceding example, the system time is persisted in MySQL for every second. For this to work, you must 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 |
.............
.............
.............
```

B.3.3. 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, an out-of-the-box `bridge-processor`, and DSL primitives to build these type of topologies.

Assume that data is queueing up in RabbitMQ (for example, queue = `myRabbit`) and the requirement is to consume all the payloads and publish them to Apache Kafka (for example, topic = `myKafka`), as the destination for downstream processing.

In that case, you should follow the global application ofconfigurations to define multiple binder configurations, as shown in the following configuration

```
# Apache Kafka Global Configurations (that is, identified by "kafka1")
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafka1.type=kafka
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafka1.environment.spring.cloud.stream.kaf
ka.binder.brokers=localhost:9092
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafka1.environment.spring.cloud.stream.kaf
ka.binder.zkNodes=localhost:2181

# RabbitMQ Global Configurations (that is, 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=l
ocalhost
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.port=5
672
```

In the preceding example, both message brokers are running locally and are reachable at `localhost` through their respective ports.

These properties can be supplied in a `.properties` file that is accessible to the server directly or through `config-server`, as follows:

```
java -jar spring-cloud-dataflow-server/target/spring-cloud-dataflow-server-2.0.0.BUILD-SNAPSHOT.jar --
spring.config.location=/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 must 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 must import the `BridgeProcessorConfiguration` provided by the starter.

Once you have the necessary adjustments, you can build the application. The following example registers 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, as follows:

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

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

The stream consumes events from the `myRabbit` queue in RabbitMQ and sends the data to the `myKafka` topic in Apache Kafka.

Appendix C: Building

To build the source, you need to install JDK 1.8.

The build uses the Maven wrapper so that you do not have to install a specific version of Maven. To enable the tests for Redis, run the server before building. More information on how to run Redis appears later in this appendix.

The main build command is as follows:

```
$ ./mvnw clean install
```

If you like, you can add '-DskipTests' to avoid running the tests.

You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that, you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for Spring pre-release artifacts.

You might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value similar to `-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.

C.1. Documentation

There is a `full` profile that generates documentation. You can build only the documentation by using the following command:

```
$ ./mvnw clean package -DskipTests -P full -pl spring-cloud-dataflow-docs -am
```

C.2. Working with the Code

If you do not have an IDE preference, we recommend that you useSpring Tools Suite or Eclipse when working with the code. We use the m2eclipse Eclipse plugin for Maven support. Other IDEs and tools generally also work without issue.

C.2.1. Importing into Eclipse with m2eclipse

We recommend the m2eclipe eclipse plugin when working with Eclipse. If you do not already have m2eclipse installed, it is available from the Eclipse marketplace.

Unfortunately, m2e does not yet support Maven 3.3. Consequently, once the projects are imported into Eclipse, you 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. To do so:

1. Open your Eclipse preferences.
2. Expand the Maven preferences.
3. Select User Settings.
4. In the User Settings field, click Browse and navigate to the Spring Cloud project you imported.
5. Select the `.settings.xml` file in that project.
6. Click Apply.
7. Click OK.

Alternatively, you can copy the repository settings from Spring Cloud's `.settings.xml` file into your own `~/.m2/settings.xml`.

C.2.2. Importing into Eclipse without m2eclipse

If you prefer not to use m2eclipse, you can generate Eclipse project metadata by using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated Eclipse projects can be imported by selecting `Import existing projects` from the `File` menu.

Appendix D: Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial, please do not hesitate, but follow the guidelines below.

D.1. Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request, we need you to sign thecontributor's agreement. Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team and be given the ability to merge pull requests.

D.2. Code Conventions and Housekeeping

None of the following guidelines is essential for a pull request, but they all help your fellow developers understand and work with your code. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse, you can import formatter settings by using the `eclipse-code-formatter.xml` file from the Spring Cloud Build project. If using IntelliJ, you can use the Eclipse Code Formatter Plugin to import the same file.
- Make sure all new `.java` files have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph describing the class's purpose.
- Add the ASF license header comment to all new `.java` files (to do so, copy from existing files in the project).
- Add yourself as an `@author` to the .java files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well. Someone has to do it, and your fellow developers appreciate the effort.
- If no one else uses your branch, rebase it against the current master (or other target branch in the main project).
- When writing a commit message, follow these conventions. If you fix an existing issue, add `Fixes gh-XXXX` (where XXXX is the issue number) at the end of the commit message.

1. HATEOAS stands for Hypermedia as the Engine of Application State