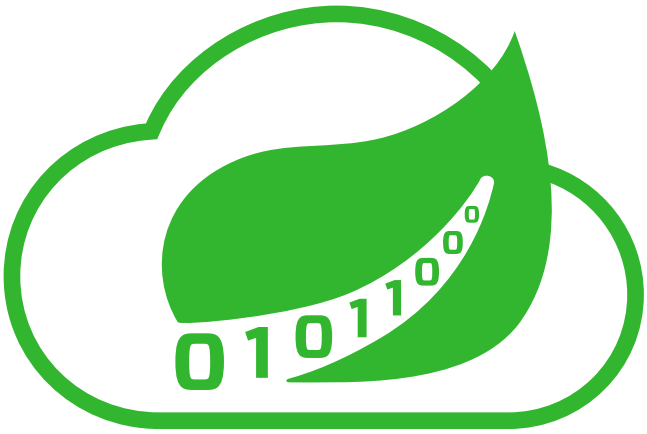


Spring Cloud Data Flow Samples

1.2.0.BUILD-SNAPSHOT

Table of Contents

1. Overview

This guide contains samples and demonstrations of how to build data microservices applications with Spring Cloud Data Flow.

2. Streaming

2.1 Cassandra Samples

HTTP to Cassandra Demo

In this demonstration, you will learn how to build a data pipeline using Spring Cloud Data Flow to consume data from an *HTTP* endpoint and write the payload to a *Cassandra* database.

We will take you through the steps to configure and Spring Cloud Data Flow server in either a local or Cloud Foundry environment.

Prerequisites

- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

Using the Local Server

Additional Prerequisites

- A running local Data Flow Server

The Local Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-local/target`

To run the Local Data Flow server Open a new terminal session:

```
$cd  <PATH/TO/SPRING-CLOUD-DATAFLOW-LOCAL-JAR>
$java -jar spring-cloud-dataflow-server-local-<VERSION>.jar
```

- Running instance of Kafka
- Running instance of Apache Cassandra
- A database utility tool such as DBeaver to connect to the Cassandra instance. You might have to provide `host`, `port`, `username` and `password` depending on the Cassandra configuration you are using.
- Create a keyspace and a `book` table in Cassandra using:

```
CREATE KEYSPACE clouddata WITH REPLICATION = { 'class' :
 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1' } AND DURABLE_WRITES =
 true;
USE clouddata;
CREATE TABLE book  (
    id          uuid PRIMARY KEY,
    isbn        text,
    author      text,
    title       text
);
```

1. Register the out-of-the-box applications for the Kafka binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

```
dataflow:>app import --uri http://bit.ly/Bacon-RELEASE-stream-applications-kafka-10-maven
```

2. Create the stream

```
dataflow:>stream create cassandrastream --definition "http --server.port=8888 --
spring.cloud.stream.bindings.output.contentType='application/json' | cassandra --
ingestQuery='insert into book (id, isbn, title, author) values (uuid(), ?, ?, ?)' --
keyspace=clouddata" --deploy

Created and deployed new stream 'cassandrastream'
```

Note

If Cassandra isn't running on default port on `localhost` or if you need username and password to connect, use one of the following options to specify the necessary connection parameters: `--username='<USERNAME>' --password='<PASSWORD>' --port=<PORT> --contact-points=<LIST-OF-HOSTS>`

3. Verify the stream is successfully deployed

```
dataflow:>stream list
```

4. Notice that `cassandrastream-http` and `cassandrastream-cassandra` Spring Cloud Stream applications are running as Spring Boot applications within the `server` as a collocated process.

```
2015-12-15 15:52:31.576  INFO 18337 --- [nio-9393-exec-1]
 o.s.c.d.a.s.l.OutOfProcessModuleDeployer : deploying module
 org.springframework.cloud.stream.module:cassandra-sink:jar:exec:1.0.0.BUILD-SNAPSHOT instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-data-
flow-2842409426977614 20/cassandrastream.cassandra
2015-12-15 15:52:31.583  INFO 18337 --- [nio-9393-exec-1]
 o.s.c.d.a.s.l.OutOfProcessModuleDeployer : deploying module
 org.springframework.cloud.stream.module:http-source:jar:exec:1.0.0.BUILD-SNAPSHOT instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-data-
flow-2842409426977614 20/cassandrastream.http
```

5. Post sample data pointing to the `http` endpoint: `localhost:8888` (`8888` is the `server.port` we specified for the `http` source in this case)

```
dataflow:>http post --contentType 'application/json' --data '{"isbn": "1599869772", "title": "The
 Art of War", "author": "Sun Tzu"}' --target http://localhost:8888
> POST (application/json;charset=UTF-8) http://localhost:8888 {"isbn": "1599869772", "title": "The
 Art of War", "author": "Sun Tzu"}
> 202 ACCEPTED
```

6. Connect to the Cassandra instance and query the table `clouddata.book` to list the persisted records

```
select * from clouddata.book;
```

7. You're done!

Using Cloud Foundry Server

Additional Prerequisites

- Cloud Foundry instance
- The Spring Cloud Data Flow Cloud Foundry Server

The Cloud Foundry Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-cloudfoundry/target`

Note

Although you can run the Data Flow Cloud Foundry Server locally and configure it to deploy to any Cloud Foundry instance, we will deploy the server to Cloud Foundry as recommended.

1. Verify that CF instance is reachable (Your endpoint urls will be different from what is shown here).

```
$ cf api
API endpoint: https://api.system.io (API version: ...)

$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

No apps found
```

2. Follow the instructions to deploy the Spring Cloud Data Flow Cloud Foundry server. Don't worry about creating a Redis service. We won't need it. If you are familiar with Cloud Foundry application manifests, we recommend creating a manifest for the the Data Flow server as shown here.

Warning

As of this writing, there is a typo on the `SPRING_APPLICATION_JSON` entry in the sample manifest. `SPRING_APPLICATION_JSON` must be followed by `:` and The JSON string must be wrapped in single quotes. Alternatively, you can replace that line with `MAVEN_REMOTE_REPOSITORIES_REPO1_URL: repo.spring.io/libs-snapshot`. If your Cloud Foundry installation is behind a firewall, you may need to install the stream apps used in this sample in your internal Maven repository and configure the server to access that repository.

3. Once you have successfully executed `cf push`, verify the dataflow server is running

```
$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

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

4. Notice that the `dataflow-server` application is started and ready for interaction via the url endpoint

5. Connect the `shell` with `server` running on Cloud Foundry, e.g., `dataflow-server.app.io`

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar


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

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

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

- Running instance of `cassandra` in Cloud Foundry or from another Cloud provider
- A database utility tool such as DBeaver to connect to the Cassandra instance. You might have to provide `host`, `port`, `username` and `password` depending on the Cassandra configuration you are using.
- Create a `book` table in your Cassandra keyspace using:

```
CREATE TABLE book  (
    id          uuid PRIMARY KEY,
    isbn        text,
    author      text,
    title       text
);
```

6. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

7. Create the stream

```
dataflow:>stream create cassandrastream --definition "http --
spring.cloud.stream.bindings.output.contentType='application/json' | cassandra --ingestQuery='insert
 into book (id, isbn, title, author) values (uuid(), ?, ?, ?)' --username='<USERNAME>' --
password='<PASSWORD>' --port=<PORT> --contact-points=<HOST> --keyspace='<KEYSPACE>'" --deploy

Created and deployed new stream 'cassandrastream'
```

8. Verify the stream is successfully deployed

```
dataflow:>stream list
```

9. Notice that `cassandrastream-http` and `cassandrastream-cassandra` Spring Cloud Stream applications are running as *cloud-native* (microservice) applications in Cloud Foundry

```
$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

name                        requested state   instances   memory   disk   urls
cassandrastream-cassandra   started           1/1         1G       1G     cassandrastream-
cassandra.app.io
cassandrastream-http        started           1/1         1G       1G     cassandrastream-http.app.io
dataflow-server             started           1/1         1G       1G     dataflow-server.app.io
```

10. Lookup the `url` for `cassandrastream-http` application from the list above. Post sample data pointing to the `http` endpoint: `<YOUR-cassandrastream-http-APP-URL>`

```
http post --contentType 'application/json' --data '{"isbn": "1599869772", "title": "The Art of War",
 "author": "Sun Tzu"}' --target http://<YOUR-cassandrastream-http-APP-URL>
> POST (application/json;charset=UTF-8) http://cassandrastream-http.app.io {"isbn": "1599869772",
 "title": "The Art of War", "author": "Sun Tzu"}
> 202 ACCEPTED
```

11. Connect to the Cassandra instance and query the table `book` to list the data inserted

```
select * from book;
```

12. Now, let's try to take advantage of Pivotal Cloud Foundry's platform capability. Let's scale the `cassandrastream-http` application from 1 to 3 instances

```
$ cf scale cassandrastream-http -i 3
Scaling app cassandrastream-http in org user-dataflow / space development as user...
OK
```

13. Verify App instances (3/3) running successfully

```
$ cf apps
Getting apps in org user-dataflow / space development as user...
OK

name                        requested state   instances   memory   disk   urls
cassandrastream-cassandra   started           1/1         1G       1G     cassandrastream-
cassandra.app.io
cassandrastream-http        started           3/3         1G       1G     cassandrastream-http.app.io
dataflow-server             started           1/1         1G       1G     dataflow-server.app.io
```

14. You're done!

Summary

In this sample, you have learned:

- How to use Spring Cloud Data Flow's `Local` and `Cloud Foundry` servers
- How to use Spring Cloud Data Flow's `shell`
- How to create streaming data pipeline to connect and write to `Cassandra`
- How to scale data microservice applications on `Pivotal Cloud Foundry`

2.2 JDBC Samples

HTTP to MySQL Demo

In this demonstration, you will learn how to build a data pipeline using Spring Cloud Data Flow to consume data from an `http` endpoint and write to MySQL database using `jdbc` sink.

We will take you through the steps to configure and Spring Cloud Data Flow server in either a local or Cloud Foundry environment.

Prerequisites

- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

Using Local Server

Additional Prerequisites

- A running local Data Flow Server

The Local Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-local/target`

To run the Local Data Flow server Open a new terminal session:

```
$cd  <PATH/TO/SPRING-CLOUD-DATAFLOW-LOCAL-JAR>
$java -jar spring-cloud-dataflow-server-local-<VERSION>.jar
```

- Running instance of Kafka
- Running instance of MySQL
- A database utility tool such as DBeaver or DbVisualizer
- Create the `test` database with a `names` table (in MySQL) using:

```
CREATE DATABASE test;
USE test;
CREATE TABLE names
(
 name varchar(255)
);
```

1. Register the out-of-the-box applications for the Kafka binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

```
dataflow:>app import --uri http://bit.ly/Bacon-RELEASE-stream-applications-kafka-10-maven
```

2. Create the stream

```
dataflow:>stream create --name mysqlstream --definition "http --server.port=8787 | jdbc --
tableName=names --columns=name --spring.datasource.driver-class-name=org.mariadb.jdbc.Driver --
spring.datasource.url='jdbc:mysql://localhost:3306/test'" --deploy

Created and deployed new stream 'mysqlstream'
```

Note

If MySQL isn't running on default port on `localhost` or if you need username and password to connect, use one of the following options to specify the

necessary connection parameters: `--spring.datasource.url='jdbc:mysql://<HOST>:<PORT>/<NAME>' --spring.datasource.username=<USERNAME> --spring.datasource.password=<PASSWORD>`

3. Verify the stream is successfully deployed

```
dataflow:>stream list
```

4. Notice that `mysqlstream-http` and `mysqlstream-jdbc` Spring Cloud Stream applications are running as Spring Boot applications within the Local `server` as collocated processes.

```
2016-05-03 09:29:55.918  INFO 65162 --- [nio-9393-exec-3] o.s.c.d.spi.local.LocalAppDeployer
  : deploying app mysqlstream.jdbc instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-
dataflow-6850863945840320040/mysqlstream1-1462292995903/mysqlstream.jdbc
2016-05-03 09:29:55.939  INFO 65162 --- [nio-9393-exec-3] o.s.c.d.spi.local.LocalAppDeployer
  : deploying app mysqlstream.http instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-
dataflow-6850863945840320040/mysqlstream-1462292995934/mysqlstream.http
```

5. Post sample data pointing to the `http` endpoint: `localhost:8787` [`8787` is the `server.port` we specified for the `http` source in this case]

```
dataflow:>http post --contentType 'application/json' --target http://localhost:8787 --data "{\"name
\": \"Foo\"}"
> POST (application/json;charset=UTF-8) http://localhost:8787 {"name": "Spring Boot"}
> 202 ACCEPTED
```

+

1. Connect to the MySQL instance and query the table `test.names` to list the new rows:

```
select * from test.names;
```

2. You're done!

Using Cloud Foundry Server

Additional Prerequisites

- Cloud Foundry instance
- The Spring Cloud Data Flow Cloud Foundry Server

The Cloud Foundry Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-cloudfoundry/target`

Note

Although you can run the Data Flow Cloud Foundry Server locally and configure it to deploy to any Cloud Foundry instance, we will deploy the server to Cloud Foundry as recommended.

1. Verify that CF instance is reachable (Your endpoint urls will be different from what is shown here).

```
$ cf api
API endpoint: https://api.system.io (API version: ...)

$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
```

```
OK

No apps found
```

2. Follow the instructions to deploy the Spring Cloud Data Flow Cloud Foundry server. Don't worry about creating a Redis service. We won't need it. If you are familiar with Cloud Foundry application manifests, we recommend creating a manifest for the the Data Flow server as shown here.

Warning

As of this writing, there is a typo on the `SPRING_APPLICATION_JSON` entry in the sample manifest. `SPRING_APPLICATION_JSON` must be followed by `:` and The JSON string must be wrapped in single quotes. Alternatively, you can replace that line with `MAVEN_REMOTE_REPOSITORIES_REPO1_URL: repo.spring.io/libs-snapshot`. If your Cloud Foundry installation is behind a firewall, you may need to install the stream apps used in this sample in your internal Maven repository and configure the server to access that repository.

3. Once you have successfully executed `cf push`, verify the dataflow server is running

```
$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

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

4. Notice that the `dataflow-server` application is started and ready for interaction via the url endpoint

5. Connect the `shell` with `server` running on Cloud Foundry, e.g., `dataflow-server.app.io`

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar


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


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

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

- Running instance of `rabbit` in Cloud Foundry
- Running instance of `mysql` in Cloud Foundry
- A database utility tool such as DBeaver or DbVisualizer
- Create the `names` table (in MySQL) using:

```
CREATE TABLE names
```

```
(
 name varchar(255)
);
```

6. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

7. Create the stream

```
dataflow:>stream create --name mysqlstream --definition "http | jdbc --tableName=names --
columns=name"
Created new stream 'mysqlstream'

dataflow:>stream deploy --name mysqlstream --properties
 "app.jdbc.spring.cloud.deployer.cloudfoundry.services=mysql"
Deployed stream 'mysqlstream'
```

Note

By supplying the `app.jdbc.spring.cloud.deployer.cloudfoundry.services=mysql` property, we are deploying the stream with `jdbc-sink` to automatically bind to `mysql` service and only this application in the stream gets the service binding. This also eliminates the requirement to supply `datasource` credentials in stream definition.

8. Verify the stream is successfully deployed

```
dataflow:>stream list
```

9. Notice that `mysqlstream-http` and `mysqlstream-jdbc` Spring Cloud Stream applications are running as *cloud-native* (microservice) applications in Cloud Foundry

```
$ cf apps
```

```
Getting apps in org user-dataflow / space development as user...
OK

name                        requested state   instances   memory   disk   urls
mysqlstream-http            started           1/1         1G       1G     mysqlstream-http.app.io
mysqlstream-jdbc            started           1/1         1G       1G     mysqlstream-jdbc.app.io
dataflow-server             started           1/1         1G       1G     dataflow-server.app.io
```

10. Lookup the `url` for `mysqlstream-http` application from the list above. Post sample data pointing to the `http` endpoint: `<YOUR-mysqlstream-http-APP-URL>`

```
http post --contentType 'application/json' --target http://mysqlstream-http.app.io --data "{\"name\":
 \"Bar"}"
> POST (application/json;charset=UTF-8) http://mysqlstream-http.app.io {"name": "Bar"}
> 202 ACCEPTED
```

11. Connect to the MySQL instance and query the table `names` to list the new rows:

```
select * from names;
```

12. Now, let's take advantage of Pivotal Cloud Foundry's platform capability. Let's scale the `mysqlstream-http` application from 1 to 3 instances

```
$ cf scale mysqlstream-http -i 3
Scaling app mysqlstream-http in org user-dataflow / space development as user...
OK
```

13. Verify App instances (3/3) running successfully

```
$ cf apps
Getting apps in org user-dataflow / space development as user...
OK

name                        requested state   instances   memory   disk   urls
mysqlstream-http            started           3/3         1G       1G     mysqlstream-http.app.io
mysqlstream-jdbc            started           1/1         1G       1G     mysqlstream-jdbc.app.io
dataflow-server             started           1/1         1G       1G     dataflow-server.app.io
```

14. You're done!

Summary

In this sample, you have learned:

- How to use Spring Cloud Data Flow's `Local` and `Cloud Foundry` servers
- How to use Spring Cloud Data Flow's `shell`
- How to create streaming data pipeline to connect and write to `MySQL`
- How to scale data microservice applications on `Pivotal Cloud Foundry`

2.3 GemFire Samples

HTTP to Gemfire Demo

In this demonstration, you will learn how to build a data pipeline using Spring Cloud Data Flow to consume data from an `http` endpoint and write to Gemfire using the `gemfire` sink.

We will take you through the steps to configure and run Spring Cloud Data Flow server in either a local or Cloud Foundry environment.

Note

For legacy reasons the `gemfire` Spring Cloud Stream Apps are named after `Pivotal GemFire`. The code base for the commercial product has since been open sourced as `Apache Geode`. These samples should work with compatible versions of Pivotal GemFire or Apache Geode. Herein we will refer to the installed IMDG simply as `Geode`.

Prerequisites

- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

- A Geode installation with a locator and cache server running Unresolved directive in streaming/gemfire/gemfire-http/overview.adoc - include::geode-setup.adoc[]

Using the Local Server

Additional Prerequisites

- A running local Data Flow Server

The Local Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-local/target`

To run the Local Data Flow server Open a new terminal session:

```
$cd  <PATH/TO/SPRING-CLOUD-DATAFLOW-LOCAL-JAR>
$java -jar spring-cloud-dataflow-server-local-<VERSION>.jar
```

- A running instance of Rabbit MQ

---

1. Use gfsh to start a locator and server

```
gfsh>start locator --name=locator1
gfsh>start server --name=server1
```

2. Create a region called `Stocks`

```
gfsh>create region --name Stocks --type=REPLICATE
```

Use the Shell to create the sample stream

1. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

2. Create the stream

This example creates an http endpoint to which we will post stock prices as a JSON document containing `symbol` and `price` fields. The property `--json=true` to enable Geode's JSON support and configures the sink to convert JSON String payloads to PdxInstance, the recommended way to store JSON documents in Geode. The `keyExpression` property is a SpEL expression used to extract the `symbol` value the PdxInstance to use as an entry key.

Note

PDX serialization is very efficient and supports OQL queries without requiring a custom domain class. Use of custom domain types requires these classes to be in the class path of both the stream apps and the cache server. For this reason, the use of custom payload types is generally discouraged.

```
dataflow:>stream create --name stocks --definition "http --port=9090 | gemfire --json=true --
regionName=Stocks --keyExpression=payload.getField('symbol')" --deploy
Created and deployed new stream 'stocks'
```

Note

If the Geode locator isn't running on default port on `localhost`, add the options `--connect-type=locator --host-addresses=<host>:<port>`. If there are multiple locators, you can provide a comma separated list of locator addresses. This is not necessary for the sample but is typical for production environments to enable fail-over.

3. Verify the stream is successfully deployed

```
dataflow:>stream list
```

4. Post sample data pointing to the `http` endpoint: `localhost:9090` (`9090` is the `port` we specified for the `http` source)

```
dataflow:>http post --target http://localhost:9090 --contentType application/json --data
 '{"symbol":"VMW","price":117.06}'
> POST (application/json) http://localhost:9090 {"symbol":"VMW","price":117.06}
> 202 ACCEPTED
```

5. Using `gfsh`, connect to the locator if not already connected, and verify the cache entry was created.

```
gfsh>get --key='VMW' --region=/Stocks
Result      : true
Key Class   : java.lang.String
Key         : VMW
Value Class : org.apache.geode.pdx.internal.PdxInstanceImpl

symbol | price
------ | ------
VMW    | 117.06
```

6. You're done!

Using the Cloud Foundry Server

Additional Prerequisites

- A Cloud Foundry instance
- Cloud Data Flow Cloud Foundry Server

The Cloud Foundry Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-cloudfoundry/target`

Note

Although you can run the Data Flow Cloud Foundry Server locally and configure it to deploy to any Cloud Foundry instance, we will deploy the server to Cloud Foundry as recommended.

1. Verify that CF instance is reachable (Your endpoint urls will be different from what is shown here).

```
$ cf api
API endpoint: https://api.system.io (API version: ...)

$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

No apps found
```

2. Follow the instructions to deploy the Spring Cloud Data Flow Cloud Foundry server. Don't worry about creating a Redis service. We won't need it. If you are familiar with Cloud Foundry application manifests, we recommend creating a manifest for the the Data Flow server as shown here.

Warning

As of this writing, there is a typo on the `SPRING_APPLICATION_JSON` entry in the sample manifest. `SPRING_APPLICATION_JSON` must be followed by `:` and The JSON string must be wrapped in single quotes. Alternatively, you can replace that line with `MAVEN_REMOTE_REPOSITORIES_REPO1_URL: repo.spring.io/libs-snapshot`. If your Cloud Foundry installation is behind a firewall, you may need to install the stream apps used in this sample in your internal Maven repository and configure the server to access that repository.

3. Once you have successfully executed `cf push`, verify the dataflow server is running

```
$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

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

4. Notice that the `dataflow-server` application is started and ready for interaction via the url endpoint

5. Connect the `shell` with `server` running on Cloud Foundry, e.g., `dataflow-server.app.io`

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar


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


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

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

```
dataflow:>
```

- Running instance of a `rabbit` service in Cloud Foundry
- Running instance of the Pivotal Cloud Cache for PCF (PCC) service `cloudcache` in Cloud Foundry.

6. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

7. Get the PCC connection information

```
$ cf service-key cloudcache my-service-key
Getting key my-service-key for service instance cloudcache as <user>...

{
 "locators": [
  "10.0.16.9[55221]",
  "10.0.16.11[55221]",
  "10.0.16.10[55221]"
 ],
 "urls": {
  "gfsh": "http://...",
  "pulse": "http://.../pulse"
 },
 "users": [
  {
   "password": <password>,
   "username": "cluster_operator"
  },
  {
   "password": <password>,
   "username": "developer"
  }
 ]
}
```

8. Using `gfsh`, connect to the PCC instance as `cluster_operator` using the service key values and create the Stocks region.

```
gfsh>connect --use-http --url=<gfsh-url> --user=cluster_operator --
password=<cluster_operator_password>
gfsh>create region --name Stocks --type=REPLICATE
```

9. Create the stream, connecting to the PCC instance as developer

   This example creates an http endpoint to which we will post stock prices as a JSON document containing `symbol` and `price` fields. The property `--json=true` to enable Geode's JSON support and configures the sink to convert JSON String payloads to PdxInstance, the recommended way to store JSON documents in Geode. The `keyExpression` property is a SpEL expression used to extract the `symbol` value the PdxInstance to use as an entry key.

Note

PDX serialization is very efficient and supports OQL queries without requiring a custom domain class. Use of custom domain types requires these classes to be in the class path of both the stream apps and the cache server. For this reason, the use of custom payload types is generally discouraged.

```
dataflow:>stream create --name stocks --definition "http --security.basic.enabled=false |
 gemfire --username=developer --password=<developer-password> --connect-type=locator --host-
addresses=10.0.16.9:55221 --regionName=Stocks --keyExpression=payload.getField('symbol')" --deploy
```

10. Verify the stream is successfully deployed

```
dataflow:>stream list
```

11. Post sample data pointing to the `http` endpoint

    Get the url of the http source using `cf apps`

```
dataflow:>http post --target http://<http source url> --contentType application/json --data
 '{"symbol":"VMW","price":117.06}'
> POST (application/json) http://... {"symbol":"VMW","price":117.06}
> 202 ACCEPTED
```

12. Using `gfsh`, connect to the PCC instance as `cluster_operator` using the service key values.

```
gfsh>connect --use-http --url=<gfsh-url> --user=cluster_operator --
password=<cluster_operator_password>
gfsh>get --key='VMW' --region=/Stocks
Result      : true
Key Class   : java.lang.String
Key         : VMW
Value Class : org.apache.geode.pdx.internal.PdxInstanceImpl

symbol | price
------ | ------
VMW    | 117.06
```

13. You're done!

Summary

In this sample, you have learned:

- How to use Spring Cloud Data Flow's `Local` and `Cloud Foundry` servers

- How to use Spring Cloud Data Flow's `shell`
- How to create streaming data pipeline to connect and write to `gemfire`

Gemfire to Log Demo

In this demonstration, you will learn how to build a data pipeline using Spring Cloud Data Flow to consume data from a `gemfire` endpoint and write to a log using the `log` sink. The `gemfire` source creates a CacheListener to monitor events for a region and publish a message whenever an entry is changed.

We will take you through the steps to configure and run Spring Cloud Data Flow server in either a local or Cloud Foundry environment.

Note

For legacy reasons the `gemfire` Spring Cloud Stream Apps are named after `Pivotal GemFire`. The code base for the commercial product has since been open sourced as `Apache Geode`. These samples should work with compatible versions of Pivotal GemFire or Apache Geode. Herein we will refer to the installed IMDG simply as `Geode`.

Prerequisites

- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

- A Geode installation with a locator and cache server running Unresolved directive in streaming/gemfire/gemfire-log/overview.adoc - include::geode-setup.adoc[]

Using the Local Server

Additional Prerequisites

- A Running Data Flow Server

The Local Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-local/target`

To run the Local Data Flow server Open a new terminal session:

```
$cd  <PATH/TO/SPRING-CLOUD-DATAFLOW-LOCAL-JAR>
$java -jar spring-cloud-dataflow-server-local-<VERSION>.jar
```

- A running instance of Rabbit MQ

---

1. Use gfsh to start a locator and server

```
gfsh>start locator --name=locator1
gfsh>start server --name=server1
```

2. Create a region called `Test`

```
gfsh>create region --name Test --type=REPLICATE
```

Use the Shell to create the sample stream.

1. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

2. Create the stream

   This example creates an gemfire source to which will publish events on a region

```
dataflow:>stream create --name events --definition " gemfire --regionName=Test | log" --deploy
Created and deployed new stream 'events'
```

Note

If the Geode locator isn't running on default port on `localhost`, add the options `--connect-type=locator --host-addresses=<host>:<port>`. If there are multiple locators, you can provide a comma separated list of locator addresses. This is not necessary for the sample but is typical for production environments to enable fail-over.

3. Verify the stream is successfully deployed

```
dataflow:>stream list
```

4. Monitor stdout for the log sink. When you deploy the stream, you will see log messages in the Data Flow server console like this

```
2017-10-28 17:28:23.275  INFO 15603 --- [nio-9393-exec-2] o.s.c.d.spi.local.LocalAppDeployer       :
 Deploying app with deploymentId events.log instance 0.
   Logs will be in /var/folders/hd/5yqz2v2d3sxd3n879f4sg4gr0000gn/T/spring-cloud-
dataflow-4093992067314402881/events-1509226103269/events.log
2017-10-28 17:28:23.277  INFO 15603 --- [nio-9393-exec-2] o.s.c.d.s.c.StreamDeploymentController
   : Downloading resource URI [maven://org.springframework.cloud.stream.app:gemfire-source-
rabbit:1.2.0.RELEASE]
2017-10-28 17:28:23.311  INFO 15603 --- [nio-9393-exec-2] o.s.c.d.s.c.StreamDeploymentController   :
 Deploying application named [gemfire] as part of stream named [events] with resource URI [maven://
org.springframework.cloud.stream.app:gemfire-source-rabbit:1.2.0.RELEASE]
2017-10-28 17:28:23.318  INFO 15603 --- [nio-9393-exec-2] o.s.c.d.spi.local.LocalAppDeployer       :
 Deploying app with deploymentId events.gemfire instance 0.
   Logs will be in /var/folders/hd/5yqz2v2d3sxd3n879f4sg4gr0000gn/T/spring-cloud-
dataflow-4093992067314402881/events-1509226103311/events.gemfire
```

   Copy the location of the `log` sink logs. This is a directory that ends in `events.log`. The log files will be in `stdout_0.log` under this directory. You can monitor the output of the log sink using `tail`, or something similar:

```
$tail -f /var/folders/hd/5yqz2v2d3sxd3n879f4sg4gr0000gn/T/spring-cloud-dataflow-4093992067314402881/
events-1509226103269/events.log/stdout_0.log
```

5. Using `gfsh`, create and update some cache entries

```
gfsh>put --region /Test --key 1  --value "value 1"
gfsh>put --region /Test --key 2  --value "value 2"
gfsh>put --region /Test --key 3  --value "value 3"
gfsh>put --region /Test --key 1  --value "new value 1"
```

6. Observe the log output You should see messages like:

```
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log sink                                 :
 value 1"
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log sink                                 :
 value 2"
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log sink                                 :
 value 3"
```

```
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log sink                                 :
 new value 1"
```

By default, the message payload contains the updated value. Depending on your application, you may need additional information. The data comes from EntryEvent. You can access any fields using the source's `cache-event-expression` property. This takes a SpEL expression bound to the EntryEvent. Try something like `--cache-event-expression='{key:'+key+',new_value:'+newValue+'}'` (HINT: You will need to destroy the stream and recreate it to add this property, an exercise left to the reader). Now you should see log messages like:

```
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log-sink                                 :
 {key:1,new_value:value 1}
2017-10-28 17:41:24.466  INFO 18986 --- [emfire.events-1] log-sink                                 :
 {key:2,new_value:value 2}
```

7. You're done!

Using the Cloud Foundry Server

Additional Prerequisites

- A Cloud Foundry instance
- The Spring Cloud Data Flow Cloud Foundry Server

The Cloud Foundry Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-cloudfoundry/target`

Note

Although you can run the Data Flow Cloud Foundry Server locally and configure it to deploy to any Cloud Foundry instance, we will deploy the server to Cloud Foundry as recommended.

1. Verify that CF instance is reachable (Your endpoint urls will be different from what is shown here).

```
$ cf api
API endpoint: https://api.system.io (API version: ...)

$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

No apps found
```

2. Follow the instructions to deploy the Spring Cloud Data Flow Cloud Foundry server. Don't worry about creating a Redis service. We won't need it. If you are familiar with Cloud Foundry application manifests, we recommend creating a manifest for the the Data Flow server as shown here.

Warning

As of this writing, there is a typo on the `SPRING_APPLICATION_JSON` entry in the sample manifest. `SPRING_APPLICATION_JSON` must be followed by `:` and The JSON string must be wrapped in single quotes. Alternatively, you can replace that line with `MAVEN_REMOTE_REPOSITORIES_REPO1_URL: repo.spring.io/libs-snapshot`. If your Cloud Foundry installation is behind a firewall, you may need to install the stream apps used in this sample in your internal Maven repository and configure the server to access that repository.

3. Once you have successfully executed `cf push`, verify the dataflow server is running

```
$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

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

4. Notice that the `dataflow-server` application is started and ready for interaction via the url endpoint

5. Connect the `shell` with `server` running on Cloud Foundry, e.g., `dataflow-server.app.io`

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar


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


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

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

- Running instance of a `rabbit` service in Cloud Foundry
- Running instance of the Pivotal Cloud Cache for PCF (PCC) service `cloudcache` in Cloud Foundry.

6. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using

whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

7. Get the PCC connection information

```
$ cf service-key cloudcache my-service-key
Getting key my-service-key for service instance cloudcache as <user>...

{
 "locators": [
  "10.0.16.9[55221]",
  "10.0.16.11[55221]",
  "10.0.16.10[55221]"
 ],
 "urls": {
  "gfsh": "http://...",
  "pulse": "http://.../pulse"
 },
 "users": [
  {
   "password": <password>,
   "username": "cluster_operator"
  },
  {
   "password": <password>,
   "username": "developer"
  }
 ]
}
```

8. Using `gfsh`, connect to the PCC instance as `cluster_operator` using the service key values and create the Test region.

```
gfsh>connect --use-http --url=<gfsh-url> --user=cluster_operator --
password=<cluster_operator_password>
gfsh>create region --name Test --type=REPLICATE
```

9. Create the stream, connecting to the PCC instance as developer. This example creates an gemfire source to which will publish events on a region

```
dataflow stream create --name events --definition " gemfire --username=developer --
password=<developer-password> --connect-type=locator --host-addresses=10.0.16.9:55221 --
regionName=Test | log" --deploy
```

10. Verify the stream is successfully deployed

```
dataflow:>stream list
```

11. Monitor stdout for the log sink

```
cf logs <log-sink-app-name>
```

12. Using `gfsh`, create and update some cache entries

```
gfsh>connect --use-http --url=<gfsh-url> --user=cluster_operator --
password=<cluster_operator_password>
gfsh>put --region /Test --key 1  --value "value 1"
gfsh>put --region /Test --key 2  --value "value 2"
gfsh>put --region /Test --key 3  --value "value 3"
gfsh>put --region /Test --key 1  --value "new value 1"
```

13. Observe the log output

You should see messages like:

```
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log sink                                :
 value 1"
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log sink                                :
 value 2"
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log sink                                :
 value 3"
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log sink                                :
 new value 1"
```

By default, the message payload contains the updated value. Depending on your application, you may need additional information. The data comes from EntryEvent. You can access any fields using the source's `cache-event-expression` property. This takes a SpEL expression bound to the EntryEvent. Try something like `--cache-event-expression='{key:'+key+',new_value:'+newValue+'}'` (HINT: You will need to destroy the stream and recreate it to add this property, an exercise left to the reader). Now you should see log messages like:

```
2017-10-28 17:28:52.893  INFO 18986 --- [emfire.events-1] log-sink                                :
 {key:1,new_value:value 1}
2017-10-28 17:41:24.466  INFO 18986 --- [emfire.events-1] log-sink                                :
 {key:2,new_value:value 2}
```

14. You're done!

Summary

In this sample, you have learned:

- How to use Spring Cloud Data Flow's `Local` and `Cloud Foundry` servers
- How to use Spring Cloud Data Flow's `shell`
- How to create streaming data pipeline to connect and publish events from `gemfire`

Gemfire CQ to Log Demo

In this demonstration, you will learn how to build a data pipeline using Spring Cloud Data Flow to consume data from a `gemfire-cq` (Continuous Query) endpoint and write to a log using the `log` sink. The `gemfire-cq` source creates a Continuous Query to monitor events for a region that match the query's result set and publish a message whenever such an event is emitted. In this example, we simulate monitoring orders to trigger a process whenever the quantity ordered is above a defined limit.

We will take you through the steps to configure and run Spring Cloud Data Flow server in either a local or Cloud Foundry environment.

Note

For legacy reasons the `gemfire` Spring Cloud Stream Apps are named after `Pivotal GemFire`. The code base for the commercial product has since been open sourced as `Apache Geode`. These samples should work with compatible versions of Pivotal GemFire or Apache Geode. Herein we will refer to the installed IMDG simply as `Geode`.

Prerequisites

- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

- A Geode installation with a locator and cache server running Unresolved directive in streaming/gemfire/gemfire-cq-log/overview.adoc - include::geode-setup.adoc[]

Using the Local Server

Additional Prerequisites

- A Running Data Flow Server

The Local Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-local/target`

To run the Local Data Flow server Open a new terminal session:

```
$cd  <PATH/TO/SPRING-CLOUD-DATAFLOW-LOCAL-JAR>
$java -jar spring-cloud-dataflow-server-local-<VERSION>.jar
```

- A running instance of Rabbit MQ

---

1. Use gfsh to start a locator and server

```
gfsh>start locator --name=locator1
gfsh>start server --name=server1
```

2. Create a region called `Orders`

```
gfsh>create region --name Orders --type=REPLICATE
```

===== Use the Shell to create the sample stream.

3. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

4. Create the stream

This example creates an gemfire-cq source to which will publish events matching a query criteria on a region. In this case we will monitor the `Orders` region. For simplicity, we will avoid creating a data structure for the order. Each cache entry contains an integer value representing the quantity of the ordered item. This stream will fire a message whenever the value>999. By default, the source emits only the value. Here we will override that using the `cq-event-expression` property. This accepts a SpEL expression bound to a CQEvent. To reference the entire CQEvent instace, we use `#this`. In order to display the contents in the log, we will invoke `toString()` on the instance.

```
dataflow:>stream create --name orders --definition " gemfire-cq --query='SELECT * from /Orders o
 where o > 999' --cq-event-expression=#this.toString() | log" --deploy
Created and deployed new stream 'events'
```

Note

If the Geode locator isn't running on default port on `localhost`, add the options `--connect-type=locator --host-addresses=<host>:<port>`. If there are multiple

locators, you can provide a comma separated list of locator addresses. This is not necessary for the sample but is typical for production environments to enable fail-over.

5. Verify the stream is successfully deployed

```
dataflow:>stream list
```

6. Monitor stdout for the log sink. When you deploy the stream, you will see log messages in the Data Flow server console like this

```
2017-10-30 09:39:36.283  INFO 8167 --- [nio-9393-exec-5] o.s.c.d.spi.local.LocalAppDeployer       :
 Deploying app with deploymentId orders.log instance 0.
   Logs will be in /var/folders/hd/5yqz2v2d3sxd3n879f4sg4gr0000gn/T/spring-cloud-
dataflow-5375107584795488581/orders-1509370775940/orders.log
```

Copy the location of the `log` sink logs. This is a directory that ends in `orders.log`. The log files will be in `stdout_0.log` under this directory. You can monitor the output of the log sink using `tail`, or something similar:

```
$tail -f /var/folders/hd/5yqz2v2d3sxd3n879f4sg4gr0000gn/T/spring-cloud-dataflow-5375107584795488581/
orders-1509370775940/orders.log/stdout_0.log
```

7. Using `gfsh`, create and update some cache entries

```
gfsh>put --region Orders  --value-class java.lang.Integer --key 01234 --value 1000
gfsh>put --region Orders  --value-class java.lang.Integer --key 11234 --value 1005
gfsh>put --region Orders  --value-class java.lang.Integer --key 21234 --value 100
gfsh>put --region Orders  --value-class java.lang.Integer --key 31234 --value 999
gfsh>put --region Orders  --value-class java.lang.Integer --key 21234 --value 1000
```

8. Observe the log output You should see messages like:

```
2017-10-30 09:53:02.231  INFO 8563 --- [ire-cq.orders-1] log-sink                                 :
 CqEvent [CqName=GfCq1; base operation=CREATE; cq operation=CREATE; key=01234; value=1000]
2017-10-30 09:53:19.732  INFO 8563 --- [ire-cq.orders-1] log-sink                                 :
 CqEvent [CqName=GfCq1; base operation=CREATE; cq operation=CREATE; key=11234; value=1005]
2017-10-30 09:53:53.242  INFO 8563 --- [ire-cq.orders-1] log-sink                                 :
 CqEvent [CqName=GfCq1; base operation=UPDATE; cq operation=CREATE; key=21234; value=1000]
```

9. Another interesting demonstration combines `gemfire-cq` with the http-gemfire example.

```
dataflow:> stream create --name stocks --definition "http --port=9090 | gemfire-json-server --
regionName=Stocks --keyExpression=payload.getField('symbol')" --deploy
dataflow:> stream create --name stock_watch --definition "gemfire-cq --query='Select * from /Stocks
 where symbol=''VMW''' | log" --deploy
```

1. You're done!

Using the Cloud Foundry Server

Additional Prerequisites

- A Cloud Foundry instance
- The Spring Cloud Data Flow Cloud Foundry Server

The Cloud Foundry Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-cloudfoundry/target`

Note

Although you can run the Data Flow Cloud Foundry Server locally and configure it to deploy to any Cloud Foundry instance, we will deploy the server to Cloud Foundry as recommended.

1. Verify that CF instance is reachable (Your endpoint urls will be different from what is shown here).

```
$ cf api
API endpoint: https://api.system.io (API version: ...)

$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

No apps found
```

2. Follow the instructions to deploy the Spring Cloud Data Flow Cloud Foundry server. Don't worry about creating a Redis service. We won't need it. If you are familiar with Cloud Foundry application manifests, we recommend creating a manifest for the the Data Flow server as shown here.

Warning

As of this writing, there is a typo on the `SPRING_APPLICATION_JSON` entry in the sample manifest. `SPRING_APPLICATION_JSON` must be followed by `:` and The JSON string must be wrapped in single quotes. Alternatively, you can replace that line with `MAVEN_REMOTE_REPOSITORIES_REPO1_URL: repo.spring.io/libs-snapshot`. If your Cloud Foundry installation is behind a firewall, you may need to install the stream apps used in this sample in your internal Maven repository and configure the server to access that repository.

3. Once you have successfully executed `cf push`, verify the dataflow server is running

```
$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

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

4. Notice that the `dataflow-server` application is started and ready for interaction via the url endpoint

5. Connect the `shell` with `server` running on Cloud Foundry, e.g., `dataflow-server.app.io`

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar


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


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

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

```
dataflow:>
```

- Running instance of a `rabbit` service in Cloud Foundry
- Running instance of the Pivotal Cloud Cache for PCF (PCC) service `cloudcache` in Cloud Foundry.

6. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

7. Get the PCC connection information

```
$ cf service-key cloudcache my-service-key
Getting key my-service-key for service instance cloudcache as <user>...

{
 "locators": [
  "10.0.16.9[55221]",
  "10.0.16.11[55221]",
  "10.0.16.10[55221]"
 ],
 "urls": {
  "gfsh": "http://...",
  "pulse": "http://.../pulse"
 },
 "users": [
  {
   "password": <password>,
   "username": "cluster_operator"
  },
  {
   "password": <password>,
   "username": "developer"
  }
 ]
}
```

8. Using `gfsh`, connect to the PCC instance as `cluster_operator` using the service key values and create the Test region.

```
gfsh>connect --use-http --url=<gfsh-url> --user=cluster_operator --
password=<cluster_operator_password>
gfsh>create region --name Orders --type=REPLICATE
```

9. Create the stream using the Data Flow Shell

   This example creates an gemfire-cq source to which will publish events matching a query criteria on a region. In this case we will monitor the `Orders` region. For simplicity, we will avoid creating a data structure for the order. Each cache entry contains an integer value representing the quantity of the ordered item. This stream will fire a message whenever the value>999. By default, the source emits only the value. Here we will override that using the `cq-event-expression` property. This accepts a SpEL expression bound to a CQEvent. To reference the entire CQEvent instance, we use `#this`. In order to display the contents in the log, we will invoke `toString()` on the instance.

```
dataflow:>stream create --name orders --definition " gemfire-cq  --username=developer --
password=<developer-password> --connect-type=locator --host-addresses=10.0.16.9:55221 --query='SELECT
 * from /Orders o where o > 999' --cq-event-expression=#this.toString()  | log" --deploy
Created and deployed new stream 'events'
```

10. Verify the stream is successfully deployed

```
dataflow:>stream list
```

11. Monitor stdout for the log sink

```
cf logs <log-sink-app-name>
```

12. Using `gfsh`, create and update some cache entries

```
gfsh>connect --use-http --url=<gfsh-url> --user=cluster_operator --
password=<cluster_operator_password>
gfsh>put --region Orders  --value-class java.lang.Integer --key 01234 --value 1000
gfsh>put --region Orders  --value-class java.lang.Integer --key 11234 --value 1005
gfsh>put --region Orders  --value-class java.lang.Integer --key 21234 --value 100
gfsh>put --region Orders  --value-class java.lang.Integer --key 31234 --value 999
gfsh>put --region Orders  --value-class java.lang.Integer --key 21234 --value 1000
```

13. Observe the log output You should see messages like:

```
2017-10-30 09:53:02.231  INFO 8563 --- [ire-cq.orders-1] log-sink                                 :
 CqEvent [CqName=GfCq1; base operation=CREATE; cq operation=CREATE; key=01234; value=1000]
2017-10-30 09:53:19.732  INFO 8563 --- [ire-cq.orders-1] log-sink                                 :
 CqEvent [CqName=GfCq1; base operation=CREATE; cq operation=CREATE; key=11234; value=1005]
2017-10-30 09:53:53.242  INFO 8563 --- [ire-cq.orders-1] log-sink                                 :
 CqEvent [CqName=GfCq1; base operation=UPDATE; cq operation=CREATE; key=21234; value=1000]
```

14. Another interesting demonstration combines `gemfire-cq` with the http-gemfire example.

```
dataflow:> stream create --name stocks --definition "http --port=9090 | gemfire-json-server --
regionName=Stocks --keyExpression=payload.getField('symbol')" --deploy
dataflow:> stream create --name stock_watch --definition "gemfire-cq --query='Select * from /Stocks
 where symbol=''VMW''' | log" --deploy
```

1. You're done!

Summary

In this sample, you have learned:

- How to use Spring Cloud Data Flow's `Local` and `Cloud Foundry` servers
- How to use Spring Cloud Data Flow's `shell`
- How to create streaming data pipeline to connect and publish CQ events from `gemfire`

2.4 Custom Stream Application Samples

Custom Spring Cloud Stream Processor

Prerequisites

- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

- A running local Data Flow Server

The Local Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-local/target`

To run the Local Data Flow server Open a new terminal session:

```
$cd  <PATH/TO/SPRING-CLOUD-DATAFLOW-LOCAL-JAR>
$java -jar spring-cloud-dataflow-server-local-<VERSION>.jar
```

- A Java IDE
- Maven Installed
- A running instance of Rabbit MQ

Create the custom stream app

We will create a custom Spring Cloud Stream application and run it on Spring Cloud Data Flow. We'll go through the steps to make a simple processor that converts temperature from Fahrenheit to Celsius. We will be running the demo locally, but all the steps will work in a Cloud Foundry environment as well.

1. Create a new spring cloud stream project
   - Create a Spring initializer project
   - Set the group to `demo.celsius.converter` and the artifact name as `celsius-converter-processor`
   - Choose a message transport binding as a dependency for the custom app There are options for choosing `Rabbit MQ` or `Kafka` as the message transport. For this demo, we will use `rabbit`. Type *rabbit* in the search bar under *Search for dependencies* and select `Stream Rabbit`.
   - Hit the generate project button and open the new project in an IDE of your choice
2. Develop the app

   We can now create our custom app. Our Spring Cloud Stream application is a Spring Boot application that runs as an executable jar. The application will include two Java classes:

   - `CelsiusConverterProcessorAplication.java` - the main Spring Boot application class, generated by Spring initializr
   - `CelsiusConverterProcessorConfiguration.java` - the Spring Cloud Stream code that we will write

     We are creating a transformer that takes a Fahrenheit input and converts it to Celsius. Following the same naming convention as the application file, create a new Java class in the same package called `CelsiusConverterProcessorConfiguration.java`.

     CelsiusConverterProcessorConfiguration.java.

```
@EnableBinding(Processor.class)
public class CelsiusConverterProcessorConfiguration {

    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public int convertToCelsius(String payload) {
        int fahrenheitTemperature = Integer.parseInt(payload);
        return (farenheitTemperature-32)*5/9;
    }
}
```

     Here we introduced two important Spring annotations. First we annotated the class with `@EnableBinding(Processor.class)`. Second we created a method and annotated it with `@Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)`. By adding these two annotations we have configured this stream app as a `Processor` (as opposed to a `Source` or a `Sink`). This means that the application receives

input from an upstream application via the `Processor.INPUT` channel and sends its output to a downstream application via the `Processor.OUTPUT` channel.

The `convertToCelsius` method takes a `String` as input for Fahrenheit and then returns the converted Celsius as an integer. This method is very simple, but that is also the beauty of this programming style. We can add as much logic as we want to this method to enrich this processor. As long as we annotate it properly and return valid output, it works as a Spring Cloud Stream Processor. Also note that it is straightforward to unit test this code.

3. Build the Spring Boot application with Maven

```
$cd <PROJECT_DIR>
$./mvnw clean package
```

4. Run the Application standalone

```
java -jar target/celsius-converter-processor-0.0.1-SNAPSHOT.jar
```

If all goes well, we should have a running standalone Spring Boot Application. Once we verify that the app is started and running without any errors, we can stop it.

Deploy the app to Spring Cloud Data Flow

1. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

2. Register the custom processor

```
app register --type processor --name convertToCelsius --uri <File URL of the jar file on the local
 filesystem where you built the project above> --force
```

3. Create the stream

We will create a stream that uses the out of the box `http` source and `log` sink and our custom transformer.

```
dataflow:>stream create --name convertToCelsiusStream --definition "http  --port=9090 |
 convertToCelsius | log" --deploy

Created and deployed new stream 'convertToCelsiusStream'
```

4. Verify the stream is successfully deployed

```
dataflow:>stream list
```

5. Verify that the apps have successfully deployed

```
dataflow:>runtime apps
```

```
2016-09-27 10:03:11.988  INFO 95234 --- [nio-9393-exec-9] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app convertToCelsiusStream.log instance 0
   Logs will be in /var/folders/2q/krqwcbhj2d58csmthyq_n1nw0000gp/T/spring-cloud-
dataflow-3236898888473815319/convertToCelsiusStream-1474984991968/convertToCelsiusStream.log
2016-09-27 10:03:12.397  INFO 95234 --- [nio-9393-exec-9] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app convertToCelsiusStream.convertToCelsius instance 0
   Logs will be in /var/folders/2q/krqwcbhj2d58csmthyq_n1nw0000gp/T/spring-
cloud-dataflow-3236898888473815319/convertToCelsiusStream-1474984992392/
convertToCelsiusStream.convertToCelsius
2016-09-27 10:03:14.445  INFO 95234 --- [nio-9393-exec-9] o.s.c.d.spi.local.LocalAppDeployer       :
 deploying app convertToCelsiusStream.http instance 0
   Logs will be in /var/folders/2q/krqwcbhj2d58csmthyq_n1nw0000gp/T/spring-cloud-
dataflow-3236898888473815319/convertToCelsiusStream-1474984994440/convertToCelsiusStream.http
```

6. Post sample data to the `http` endpoint: `localhost:9090` (`9090` is the `port` we specified for the `http` source in this case)

```
dataflow:>http post --target http://localhost:9090 --data 76
> POST (text/plain;Charset=UTF-8) http://localhost:9090 76
> 202 ACCEPTED
```

7. Open the log file for the `convertToCelsiusStream.log` app to see the output of our stream

```
tail -f /var/folders/2q/krqwcbhj2d58csmthyq_n1nw0000gp/T/spring-cloud-dataflow-7563139704229890655/
convertToCelsiusStream-1474990317406/convertToCelsiusStream.log/stdout_0.log
```

You should see the temperature you posted converted to Celsius!

```
2016-09-27 10:05:34.933  INFO 95616 --- [CelsiusStream-1] log.sink                                 : 24
```

Summary

In this sample, you have learned:

- How to write a custom `Processor` stream application
- How to use Spring Cloud Data Flow's `Local` server
- How to use Spring Cloud Data Flow's `shell` application

3. Task / Batch

3.1 Task Samples

Batch Job on Cloud Foundry

In this demonstration, you will learn how to orchestrate short-lived data processing application (*eg: Spring Batch Jobs*) using Spring Cloud Task and Spring Cloud Data Flow on Cloud Foundry.

Prerequisites

- Local PCFDev instance
- Local install of cf CLI command line tool
- Running instance of mysql in PCFDev
- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

- The Spring Cloud Data Flow Cloud Foundry Server running in PCFDev

The Cloud Foundry Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-cloudfoundry/target`

Note

Although you can run the Data Flow Cloud Foundry Server locally and configure it to deploy to any Cloud Foundry instance, we will deploy the server to Cloud Foundry as recommended.

1. Verify that CF instance is reachable (Your endpoint urls will be different from what is shown here).

```
$ cf api
API endpoint: https://api.system.io (API version: ...)

$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

No apps found
```

2. Follow the instructions to deploy the Spring Cloud Data Flow Cloud Foundry server. Don't worry about creating a Redis service. We won't need it. If you are familiar with Cloud Foundry application manifests, we recommend creating a manifest for the the Data Flow server as shown here.

Warning

As of this writing, there is a typo on the `SPRING_APPLICATION_JSON` entry in the sample manifest. `SPRING_APPLICATION_JSON` must be followed by `:` and The JSON string must be wrapped in single quotes. Alternatively, you can replace that line with `MAVEN_REMOTE_REPOSITORIES_REPO1_URL: repo.spring.io/libs-snapshot`. If your Cloud Foundry installation is behind a firewall, you may need to install the stream apps used in this sample in your internal Maven repository and configure the server to access that repository.

3. Once you have successfully executed `cf push`, verify the dataflow server is running

```
$ cf apps
Getting apps in org [your-org] / space [your-space] as user...
OK

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

4. Notice that the `dataflow-server` application is started and ready for interaction via the url endpoint

5. Connect the `shell` with `server` running on Cloud Foundry, e.g., `dataflow-server.app.io`

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar


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


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

```
server-unknown:>dataflow config server http://dataflow-server.app.io
```

```
Successfully targeted http://dataflow-server.app.io
dataflow:>
```

Note

PCF 1.7.12 or greater is required to run Tasks on Spring Cloud Data Flow. As of this writing, PCFDev and PWS supports builds upon this version.

6. Task support needs to be enabled on pcf-dev. Being logged as `admin`, issue the following command:

```
cf enable-feature-flag task_creation
Setting status of task_creation as admin...

OK

Feature task_creation Enabled.
```

Note

For this sample, all you need is the `mysql` service and in PCFDev, the `mysql` service comes with a different plan. From CF CLI, create the service by: `cf create-service p-mysql 512mb mysql` and bind this service to `dataflow-server` by: `cf bind-service dataflow-server mysql`.

Note

All the apps deployed to PCFDev start with low memory by default. It is recommended to change it to at least 768MB for `dataflow-server`. Ditto for every app spawned **by** Spring Cloud Data Flow. Change the memory by: `cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_MEMORY 512`. Likewise, we would have to skip SSL validation by: `cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION true`.

7. Tasks in Spring Cloud Data Flow require an RDBMS to host "task repository" (see here for more details), so let's instruct the Spring Cloud Data Flow server to bind the `mysql` service to each deployed task:

```
$ cf set-env dataflow-server SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES mysql
$ cf restage dataflow-server
```

Note

We only need `mysql` service for this sample.

8. As a recap, here is what you should see as configuration for the Spring Cloud Data Flow server:

```
cf env dataflow-server

....
User-Provided:
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN: local.pcfdev.io
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_MEMORY: 512
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG: pcfdev-org
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD: pass
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION: false
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE: pcfdev-space
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES: mysql
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL: https://api.local.pcfdev.io
SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME: user
```

```
No running env variables have been set

No staging env variables have been set
```

9. Notice that `dataflow-server` application is started and ready for interaction via `dataflow-server.local.pcfdev.io` endpoint

10. Build and register the batch-job example from Spring Cloud Task samples. For convenience, the final uber-jar artifact is provided with this sample.

```
dataflow:>app register --type task --name simple_batch_job --uri https://github.com/spring-cloud/
spring-cloud-dataflow-samples/raw/master/tasks/simple-batch-job/batch-job-1.3.0.BUILD-SNAPSHOT.jar
```

11. Create the task with `simple-batch-job` application

```
dataflow:>task create foo --definition "simple_batch_job"
```

Note

Unlike Streams, the Task definitions don't require explicit deployment. They can be launched on-demand, scheduled, or triggered by streams.

12. Verify there's **still** no Task applications running on PCFDev - they are listed only after the initial launch/staging attempt on PCF

```
$ cf apps
Getting apps in org pcfdev-org / space pcfdev-space as user...
OK

name              requested state   instances   memory   disk   urls
dataflow-server   started           1/1         768M     512M   dataflow-server.local.pcfdev.io
```

13. Let's launch `foo`

```
dataflow:>task launch foo
```

14. Verify the execution of `foo` by tailing the logs

```
$ cf logs foo
Retrieving logs for app foo in org pcfdev-org / space pcfdev-space as user...

2016-08-14T18:48:54.22-0700 [APP/TASK/foo/0]OUT Creating container
2016-08-14T18:48:55.47-0700 [APP/TASK/foo/0]OUT

2016-08-14T18:49:06.59-0700 [APP/TASK/foo/0]OUT 2016-08-15 01:49:06.598  INFO 14 --- [
  main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob: [name=job1]] launched with the
 following parameters: [{}]

...
...

2016-08-14T18:49:06.78-0700 [APP/TASK/foo/0]OUT 2016-08-15 01:49:06.785  INFO 14 --- [
  main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob: [name=job1]] completed with the
 following parameters: [{}] and the following status: [COMPLETED]

...
...

2016-08-14T18:49:07.36-0700 [APP/TASK/foo/0]OUT 2016-08-15 01:49:07.363  INFO 14 --- [
  main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob: [name=job2]] launched with the
 following parameters: [{}]

...
...
```

```
2016-08-14T18:49:07.53-0700 [APP/TASK/foo/0]OUT 2016-08-15 01:49:07.536  INFO 14 --- [
  main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob: [name=job2]] completed with the
 following parameters: [{}] and the following status: [COMPLETED]

...
...

2016-08-14T18:49:07.71-0700 [APP/TASK/foo/0]OUT Exit status 0
2016-08-14T18:49:07.78-0700 [APP/TASK/foo/0]OUT Destroying container
2016-08-14T18:49:08.47-0700 [APP/TASK/foo/0]OUT Successfully destroyed container
```

Note

Verify `job1` and `job2` operations embeddded in `simple-batch-job` application are launched independently and they returned with the status `COMPLETED`.

Note

Unlike LRPs in Cloud Foundry, tasks are short-lived, so the logs aren't always available. They are generated only when the Task application runs; at the end of Task operation, the container that ran the Task application is destroyed to free-up resources.

15.List Tasks in Cloud Foundry

```
$ cf apps
Getting apps in org pcfdev-org / space pcfdev-space as user...
OK

name              requested state   instances   memory   disk   urls
dataflow-server   started           1/1         768M     512M   dataflow-server.local.pcfdev.io
foo               stopped           0/1         1G       1G
```

16.Verify Task execution details

```
dataflow:>task execution list
#########################################################################################################
#        Task Name         #ID#         Start Time         #          End Time          #Exit Code#
#########################################################################################################
#foo                       #1 #Sun Aug 14 18:49:05 PDT 2016#Sun Aug 14 18:49:07 PDT 2016#0        #
#########################################################################################################
```

17.Verify Job execution details

```
dataflow:>job execution list
#########################################################################################################
#ID #Task ID#Job Name #         Start Time         #Step Execution Count #Definition Status #
#########################################################################################################
#2  #1      #job2     #Sun Aug 14 18:49:07 PDT 2016#1                    #Destroyed         #
#1  #1      #job1     #Sun Aug 14 18:49:06 PDT 2016#1                    #Destroyed         #
#########################################################################################################
```

Summary

In this sample, you have learned:

- How to register and orchestrate Spring Batch jobs in Spring Cloud Data Flow
- How to use the `cf` CLI in the context of Task applications orchestrated by Spring Cloud Data Flow
- How to verify task executions and task repository

4. Analytics

4.1 Twitter Analytics

In this demonstration, you will learn how to build a data pipeline using Spring Cloud Data Flow to consume data from *TwitterStream* and compute simple analytics over data-in-transit using *Field-Value-Counter*.

We will take you through the steps to configure Spring Cloud Data Flow's `Local` server.

Prerequisites

- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

- A running local Data Flow Server

The Local Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-local/target`

To run the Local Data Flow server Open a new terminal session:

```
$cd  <PATH/TO/SPRING-CLOUD-DATAFLOW-LOCAL-JAR>
$java -jar spring-cloud-dataflow-server-local-<VERSION>.jar
```

- Running instance of Redis
- Running instance of Kafka
- Twitter credentials from Twitter Developers site

1. Register the out-of-the-box applications for the Kafka binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

```
dataflow:>app import --uri http://bit.ly/Bacon-RELEASE-stream-applications-kafka-10-maven
```

2. Create and deploy the following streams

```
(1) dataflow:>stream create tweets --definition "twitterstream --consumerKey=<CONSUMER_KEY>
 --consumerSecret=<CONSUMER_SECRET> --accessToken=<ACCESS_TOKEN> --
accessTokenSecret=<ACCESS_TOKEN_SECRET> | log"
Created new stream 'tweets'

(2) dataflow:>stream create tweetlang  --definition ":tweets.twitterstream > field-value-counter
 --fieldName=lang --name=language" --deploy
Created and deployed new stream 'tweetlang'

(3) dataflow:>stream create tagcount --definition ":tweets.twitterstream > field-value-counter --
fieldName=entities.hashtags.text --name=hashtags" --deploy
Created and deployed new stream 'tagcount'

(4) dataflow:>stream deploy tweets
Deployed stream 'tweets'
```

Note

To get a consumerKey and consumerSecret you need to register a twitter application. If you don't already have one set up, you can create an app at the Twitter Developers site to get these credentials. The tokens <CONSUMER_KEY>, <CONSUMER_SECRET>, <ACCESS_TOKEN>, and <ACCESS_TOKEN_SECRET> are required to be replaced with your account credentials.

3. Verify the streams are successfully deployed. Where: (1) is the primary pipeline; (2) and (3) are tapping the primary pipeline with the DSL syntax `<stream-name>.<label/app name>` [e.x. `:tweets.twitterstream`]; and (4) is the final deployment of primary pipeline

```
dataflow:>stream list
```

4. Notice that `tweetlang.field-value-counter`, `tagcount.field-value-counter`, `tweets.log` and `tweets.twitterstream` Spring Cloud Stream applications are running as Spring Boot applications within the `local-server`.

```
2016-02-16 11:43:26.174  INFO 10189 --- [nio-9393-exec-2] o.s.c.d.d.l.OutOfProcessModuleDeployer
   : deploying module org.springframework.cloud.stream.module:field-value-counter-
sink:jar:exec:1.0.0.BUILD-SNAPSHOT instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-data-
flow-6990537012958280418/tweetlang-1455651806160/tweetlang.field-value-counter
2016-02-16 11:43:26.206  INFO 10189 --- [nio-9393-exec-3] o.s.c.d.d.l.OutOfProcessModuleDeployer
   : deploying module org.springframework.cloud.stream.module:field-value-counter-
sink:jar:exec:1.0.0.BUILD-SNAPSHOT instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-data-
flow-6990537012958280418/tagcount-1455651806202/tagcount.field-value-counter
2016-02-16 11:43:26.806  INFO 10189 --- [nio-9393-exec-4] o.s.c.d.d.l.OutOfProcessModuleDeployer
   : deploying module org.springframework.cloud.stream.module:log-sink:jar:exec:1.0.0.BUILD-
SNAPSHOT instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-data-
flow-6990537012958280418/tweets-1455651806800/tweets.log
2016-02-16 11:43:26.813  INFO 10189 --- [nio-9393-exec-4] o.s.c.d.d.l.OutOfProcessModuleDeployer
   : deploying module org.springframework.cloud.stream.module:twitterstream-
source:jar:exec:1.0.0.BUILD-SNAPSHOT instance 0
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gp/T/spring-cloud-data-
flow-6990537012958280418/tweets-1455651806800/tweets.twitterstream
```

5. Verify that two `field-value-counter` with the names `hashtags` and `language` is listing successfully

```
dataflow:>field-value-counter list
###########################
#Field Value Counter name#
###########################
#hashtags                #
#language                #
###########################
```

6. Verify you can query individual `field-value-counter` results successfully

```
dataflow:>field-value-counter display hashtags
Displaying values for field value counter 'hashtags'
###############################################
#                Value                #Count#
###############################################
#KCA                                   #   40#
#PENNYSTOCKS                           #   17#
#TEAMBILLIONAIRE                       #   17#
#UCL                                   #   11#
#...                                   #   ..#
#...                                   #   ..#
#...                                   #   ..#
###############################################

dataflow:>field-value-counter display language
Displaying values for field value counter 'language'
#############
#Value#Count#
#############
#en   #1,171#
#es   #  337#
#ar   #  296#
#und  #  251#
#pt   #  175#
#ja   #  137#
#..   #  ...#
#..   #  ...#
#..   #  ...#
#############
```

7. Go to `Dashboard` accessible at `localhost:9393/dashboard` and launch the `Analytics` tab. From the default `Dashboard` menu, select the following combinations to visualize real-time updates on `field-value-counter`.

   - For real-time updates on `language` tags, select:

     a. Metric Type as `Field-Value-Counters`

     b. Stream as `language`

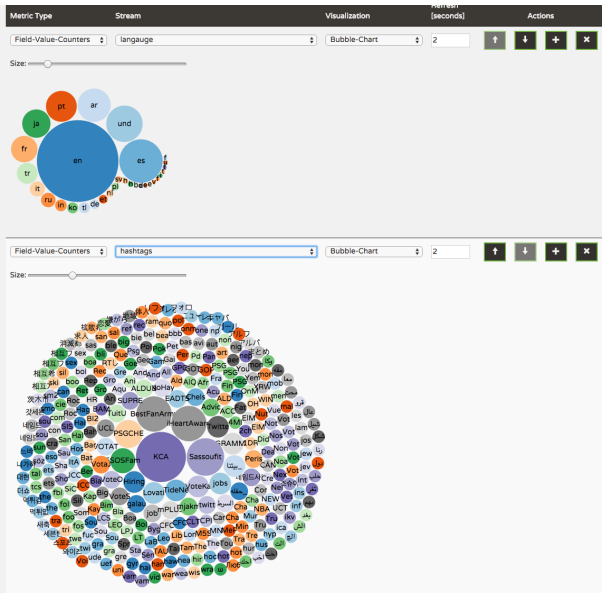
     c. Visualization as `Bubble-Chart` or `Pie-Chart`

   - For real-time updates on `hashtags` tags, select:

     a. Metric Type as `Field-Value-Counters`

     b. Stream as `hashtags`

     c. Visualization as `Bubble-Chart` or `Pie-Chart`



Summary

In this sample, you have learned:

- How to use Spring Cloud Data Flow's `Local` server
- How to use Spring Cloud Data Flow's `shell` application
- How to create streaming data pipeline to compute simple analytics using `Twitter Stream` and `Field Value Counter` applications

5. Functions

5.1 Functions in Spring Cloud Data Flow

In this sample, you will learn how to use Spring Cloud Function based streaming applications in Spring Cloud Data Flow. To learn more about Spring Cloud Function, check out the project page.

Prerequisites

- A Running Data Flow Shell

The Spring Cloud Data Flow Shell is available for download or you can build it yourself.

Note

the Spring Cloud Data Flow Shell and Local server implementation are in the same repository and are both built by running `./mvnw install` from the project root directory. If you have already run the build, use the jar in `spring-cloud-dataflow-shell/target`

To run the Shell open a new terminal session:

```
$ cd <PATH/TO/SPRING-CLOUD-DATAFLOW-SHELL-JAR>
$ java -jar spring-cloud-dataflow-shell-<VERSION>.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/


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

Note

The Spring Cloud Data Flow Shell is a Spring Boot application that connects to the Data Flow Server's REST API and supports a DSL that simplifies the process of defining a stream or task and managing its lifecycle. Most of these samples use the shell. If you prefer, you can use the Data Flow UI localhost:9393/dashboard, (or wherever it the server is hosted) to perform the same operations.

- A running local Data Flow Server

The Local Data Flow Server is Spring Boot application available for download or you can build it yourself. If you build it yourself, the executable jar will be in `spring-cloud-dataflow-server-local/target`

To run the Local Data Flow server Open a new terminal session:

```
$cd  <PATH/TO/SPRING-CLOUD-DATAFLOW-LOCAL-JAR>
$java -jar spring-cloud-dataflow-server-local-<VERSION>.jar
```

- A local build of Spring Cloud Function
- A running instance of Rabbit MQ
- General understanding of the out-of-the-box function-runner application

1. Register the out-of-the-box applications for the Rabbit binder

Note

These samples assume that the Data Flow Server can access a remote Maven repository, `repo.spring.io/libs-release` by default. If your Data Flow server is running behind a firewall, or you are using a maven proxy preventing access to public repositories, you will need to install the sample apps in your internal Maven repository and configure the server accordingly. The sample applications are typically registered using Data Flow's bulk import facility. For example, the Shell command `dataflow:>app import --uri bit.ly/Bacon-RELEASE-stream-applications-rabbit-maven` (The actual URI is release and binder specific so refer to the sample instructions for the actual URL). The bulk import URI references a plain text file containing entries for all of the publicly available Spring Cloud Stream and Task applications published to `repo.spring.io`. For example, `source.http=maven://org.springframework.cloud.stream.app:http-source-rabbit:1.2.0.RELEASE` registers the `http` source app at the corresponding Maven address, relative to the remote repository(ies) configured for the Data Flow server. The format is `maven://<groupId>:<artifactId>:<version>` You will need to download the required apps or build them and then install them in your Maven repository, using whatever group, artifact, and version you choose. If you do this, register individual apps using `dataflow:>app register…` using the `maven://` resource URI format corresponding to your installed app.

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

2. Register the out-of-the-box function-runner application (*we will use the `1.0.0.BUILD-SNAPSHOT` built by the Spring CI system*)

```
dataflow:>app register --name function-runner --type processor --uri http://repo.spring.io/
libs-snapshot/org/springframework/cloud/stream/app/function-app-rabbit/1.0.0.BUILD-SNAPSHOT/
function-app-rabbit-1.0.0.BUILD-SNAPSHOT.jar --metadata-uri http://repo.spring.io/libs-snapshot/
org/springframework/cloud/stream/app/function-app-rabbit/1.0.0.BUILD-SNAPSHOT/function-app-
rabbit-1.0.0.BUILD-SNAPSHOT-metadata.jar
```

3. Create and deploy the following stream

```
dataflow:>stream create foo --definition "http --server.port=9001 | function-runner --
function.className=com.example.functions.CharCounter --function.location=file:///<PATH/TO/SPRING-
CLOUD-FUNCTION>/spring-cloud-function-samples/function-sample/target/spring-cloud-function-
sample-1.0.0.BUILD-SNAPSHOT.jar | log" --deploy
```

Note

Replace the `<PATH/TO/SPRING-CLOUD-FUNCTION>` with the correct path.

Note

The source core of `CharCounter` function is in Spring cloud Function's samples repo.

4. Verify the stream is successfully deployed.

```
dataflow:>stream list
#########################################################################################################################################
#Stream#
 Stream Definition
         #   Status   #
# Name #

      #            #
#########################################################################################################################################
#foo   #http --server.port=9001 | function-runner --
function.className=com.example.functions.CharCounter
                                                    #All apps    #
#      #--function.location=file:///<PATH/TO/SPRING-CLOUD-FUNCTION>/spring-cloud-function-samples/
function-sample/target/spring-cloud-function-sample-1.0.0.BUILD-<SNAPSHOT class="jar"></SNAPSHOT>
     #have been   #
#      #| log

      #successfully#
#      #

      #deployed    #
#########################################################################################################################################
```

5. Notice that `foo-http`, `foo-function-runner`, and `foo-log` Spring Cloud Stream applications are running as Spring Boot applications and the log locations will be printed in the Local-server console.

```
....
....
2017-10-17 11:43:03.714  INFO 18409 --- [nio-9393-exec-7] o.s.c.d.s.s.AppDeployerStreamDeployer
   : Deploying application named [log] as part of stream named [foo] with resource URI [maven://
org.springframework.cloud.stream.app:log-sink-rabbit:jar:1.2.0.RELEASE]
2017-10-17 11:43:04.379  INFO 18409 --- [nio-9393-exec-7] o.s.c.d.spi.local.LocalAppDeployer
  : Deploying app with deploymentId foo.log instance 0.
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gs/T/spring-cloud-
dataflow-6549025456609489200/foo-1508265783715/foo.log
2017-10-17 11:43:04.380  INFO 18409 --- [nio-9393-exec-7] o.s.c.d.s.s.AppDeployerStreamDeployer
    : Deploying application named [function-runner] as part of stream named [foo] with
 resource URI [file:/var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gs/T/deployer-resource-
cache8941581850579153886/http-c73a62adae0abd7ec0dee91d891575709f02f8c9]
2017-10-17 11:43:04.384  INFO 18409 --- [nio-9393-exec-7] o.s.c.d.spi.local.LocalAppDeployer
  : Deploying app with deploymentId foo.function-runner instance 0.
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gs/T/spring-cloud-
dataflow-6549025456609489200/foo-1508265784380/foo.function-runner
2017-10-17 11:43:04.385  INFO 18409 --- [nio-9393-exec-7] o.s.c.d.s.s.AppDeployerStreamDeployer
   : Deploying application named [http] as part of stream named [foo] with resource URI [maven://
org.springframework.cloud.stream.app:http-source-rabbit:jar:1.2.0.RELEASE]
2017-10-17 11:43:04.391  INFO 18409 --- [nio-9393-exec-7] o.s.c.d.spi.local.LocalAppDeployer
  : Deploying app with deploymentId foo.http instance 0.
   Logs will be in /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gs/T/spring-cloud-
dataflow-6549025456609489200/foo-1508265784385/foo.http
....
....
```

6. Post sample data pointing to the `http` endpoint: `localhost:9001` (`9001` is the `port` we specified for the `http` source in this case)

```
dataflow:>http post --target http://localhost:9001 --data "hello world"
> POST (text/plain) http://localhost:9001 hello world
> 202 ACCEPTED


dataflow:>http post --target http://localhost:9001 --data "hmm, yeah, it works now!"
> POST (text/plain) http://localhost:9001 hmm, yeah, it works now!
> 202 ACCEPTED
```

7. Tail the log-sink's standard-out logs to see the character counts

```
$ tail -f /var/folders/c3/ctx7_rns6x30tq7rb76wzqwr0000gs/T/spring-cloud-
dataflow-6549025456609489200/foo-1508265783715/foo.log/stdout_0.log

....
....
....
....
2017-10-17 11:45:39.363  INFO 19193 --- [on-runner.foo-1] log-sink
  : 11
2017-10-17 11:46:40.997  INFO 19193 --- [on-runner.foo-1] log-sink
  : 24
....
....
```

Summary

In this sample, you have learned:

- How to use Spring Cloud Data Flow's `Local` server
- How to use Spring Cloud Data Flow's `shell` application
- How to use the out-of-the-box `function-runner` application in Spring Cloud Data Flow