My steps to learn about Spring Cloud Data Flow

Paulo Jerônimo, 2018-05-20 23:05:46 WEST

Table of Contents

Introduction

Recently I had work to produce a document with a comparison between two tools for Cloud Data Flow. I didn't have any knowledge of this kind of technology before creating this document. Spring Cloud Data Flow (SCDF) is one of the tools in my comparison document. So, here I describe my own procedures to learn about SCDF and take my own preliminary conclusions. I followed many steps on my own desktop (a MacBook Pro computer) to accomplish this task. This document shows you what I did.

Basically, to learn about SCDF in order to do a comparison with other tool:

- I saw some videos about it.
- I wrote my own labs:
  - Quick Start with Docker
  - Run SCDF in PCF Dev
  - Creating a custom processor application
  - Setting up a Windows environment to develop with SCDF
  - Connecting a source to a sink using Spring Cloud Stream
  - Manually using SCDF (server local/shell)
  - Importing existing sources, sinks, processors and tasks to SCDF

In these labs, I only used command line tools. This approach was very simple for me because I have to much experience in tools like Bash and GNU commands. I didn´t use any IDE (like Spring Tool Suite). I know that this could make all the process even simpler for developers without command line experience. But, my focus was on the technologies itself. So for me, a command line approach showed me in deeper concepts that I wanted to learn.

About this document

This document was written using Vim (my favorite text editor) and its source code is in AsciiDoc format. The generated output formats (HTML and PDF) are build (and published in GitHub Pages) with Gradle. Read more about the generation processes of this document in README.adoc.

See the online version of this document in HTML format.

About me

You can read more about me on my cv.

Videos with a technical background

Prior to starting my own labs, I saw some introductory videos (available on YouTube):

- SCDF
- Webinar Data Microservices with Spring Cloud Data Flow
- Orchestrating Data Microservices with Spring Cloud Data Flow - Mark Pollack

These was the first videos that introduce me the basic concepts. After that, I also saw many other videos (many times).

Lab 1: Quick Start with Docker

For me, the best way to start learning a new technology is by running all the stuff related to them inside a Docker container. By this way, I can abstract myself about the related installation procedures and go directly to the point.

So, I started my labs by following the steps on the Quick Start page. These steps use Docker and Docker Compose.

Prerequisites

1. Docker installed.
2. Docker Compose installed.

Step 1 - Download docker-compose.yml

My first steps were create a working directory and download the `docker-compose.yml` file:

```
$ mkdir -p ~/labs/spring-cloud-dataflow && cd $_

$ wget -c https://raw.githubusercontent.com/spring-cloud/spring-cloud-
dataflow/v1.4.0.RELEASE/spring-cloud-dataflow-server-local/docker-compose.yml
```

By default, this file is configured to use Kafka and Zookeeper:

```
$ cat docker-compose.yml
version: '3'

services:
  kafka:
    image: wurstmeister/kafka:0.10.1.0
    expose:
      - "9092"
    environment:
      - KAFKA_ADVERTISED_PORT=9092
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
    depends_on:
      - zookeeper
  zookeeper:
    image: wurstmeister/zookeeper
    expose:
      - "2181"
    environment:
      - KAFKA_ADVERTISED_HOST_NAME=zookeeper
  dataflow-server:
    image: springcloud/spring-cloud-dataflow-server-local:1.4.0.RELEASE
    container_name: dataflow-server
    ports:
      - "9393:9393"
    environment:
      -
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.br
okers=kafka:9092
      -
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.zk
Nodes=zookeeper:2181
    depends_on:
      - kafka
  app-import:
    image: alpine:3.7
    depends_on:
      - dataflow-server
    command: >
      /bin/sh -c "
        while ! nc -z dataflow-server 9393;
        do
          sleep 1;
        done;
        wget -qO- 'http://dataflow-server:9393/apps' --post
-data='uri=http://bit.ly/Celsius-SR1-stream-applications-kafka-10-maven&force=true';
        echo 'Stream apps imported'
        wget -qO- 'http://dataflow-server:9393/apps'  --post
-data='uri=http://bit.ly/Clark-GA-task-applications-maven&force=true';
        echo 'Task apps imported'"
```

I could also change these settings to use RabbitMQ, MySQL, and Redis. I show you how to do this in the step 9 of this lab.

Step 2 - Start Docker Compose

I started `docker-compose`:

```
docker-compose up
```

The above command downloaded all the necessary Docker images (configured in `docker-compose.yml` file). After that, it also created and started the containers.

The current shell remains blocked until we stop `docker-compose`.

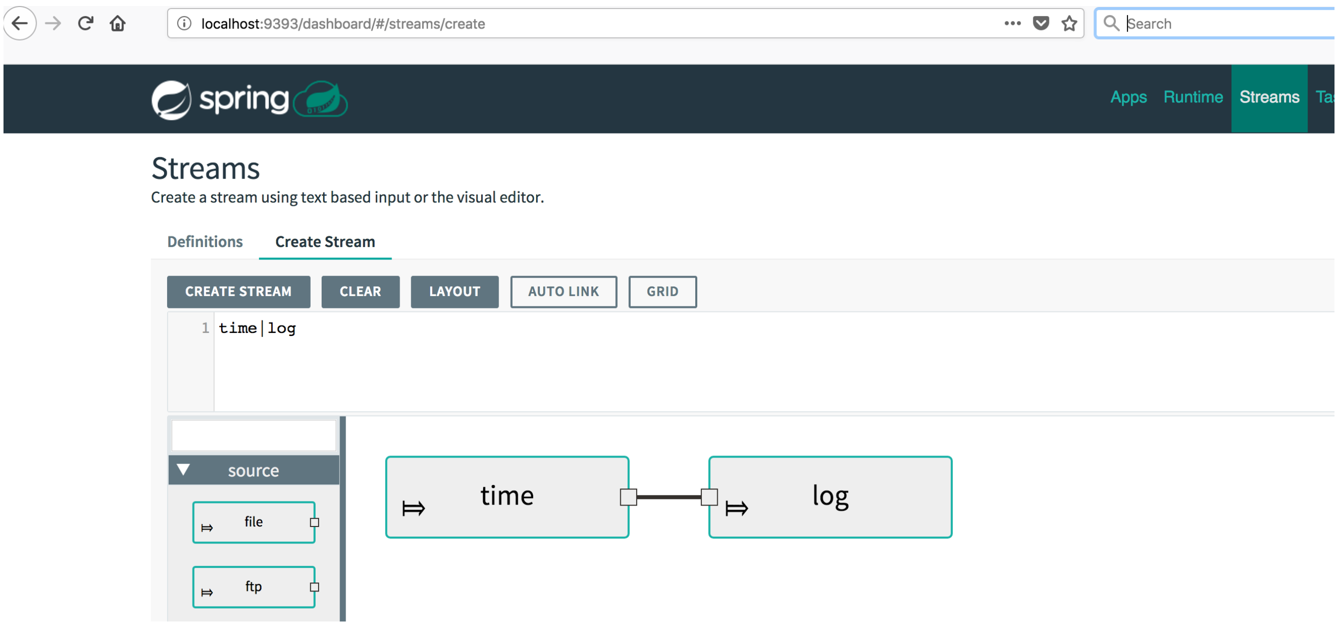
Step 3 - Launch the Spring Cloud Data Flow Dashboard

I opened the dashboard at http://localhost:9393/dashboard.

Step 4 - Create a Stream

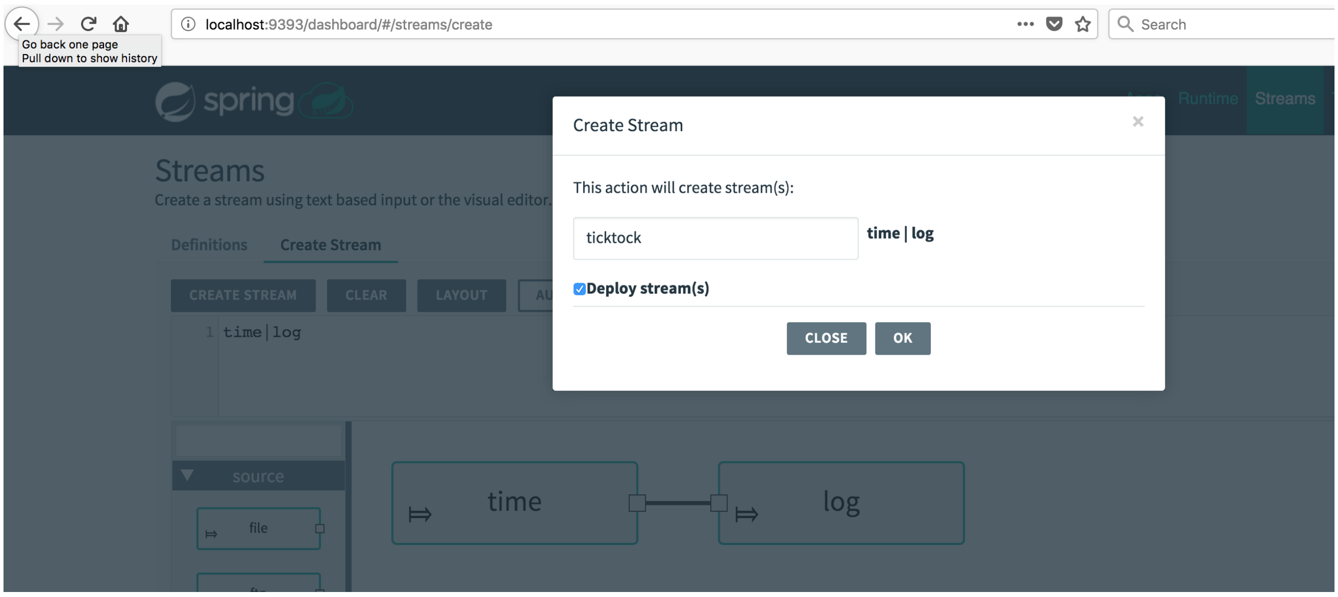
I did some steps in the UI of the dashboard.

I used `Create Stream` under `Streams` tab to define and deploy a stream `time | log` called `ticktock`. This was done in two steps:



```
time|log
```

Figure 1. Step 4.1



Streams

Create a stream using text based input or the visual editor.

```
time|log
```

Create Stream

This action will create stream(s):

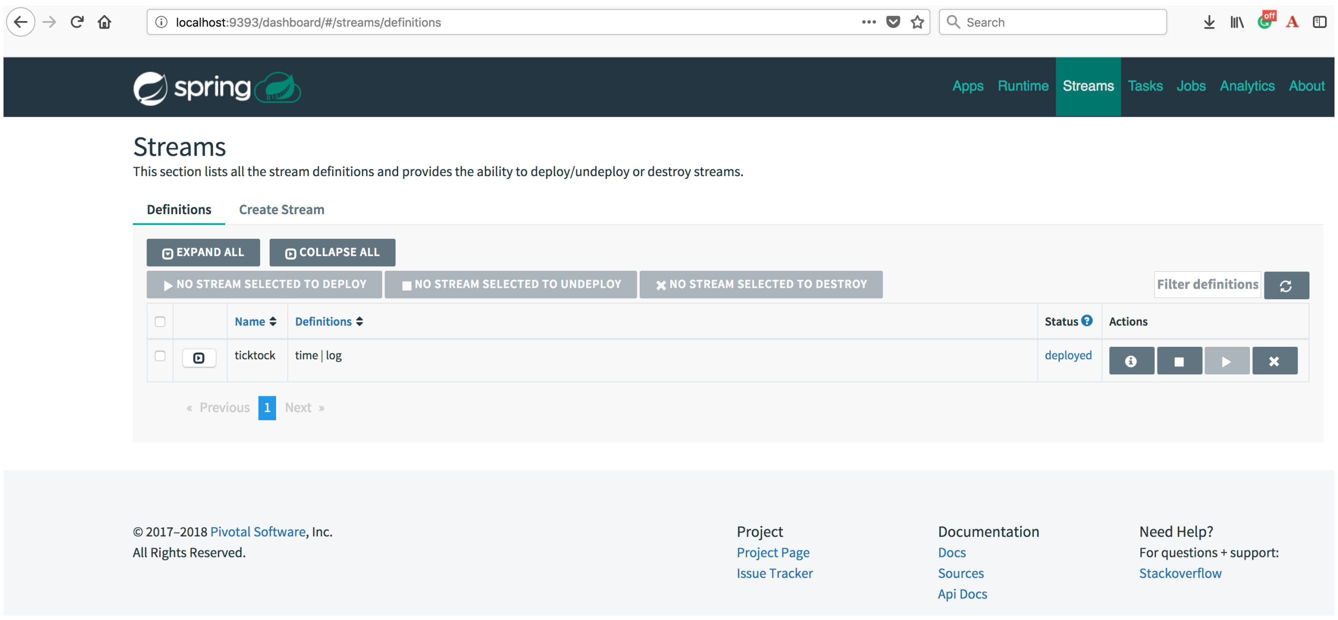
ticktock	time | log

☑ Deploy stream(s)

CLOSE OK

Figure 2. Step 4.2

After that, the `ticktock` stream was deployed:



Streams

This section lists all the stream definitions and provides the ability to deploy/undeploy or destroy streams.

Name	Definitions	Status	Actions
ticktock	time | log	deployed	

© 2017–2018 Pivotal Software, Inc.
All Rights Reserved.

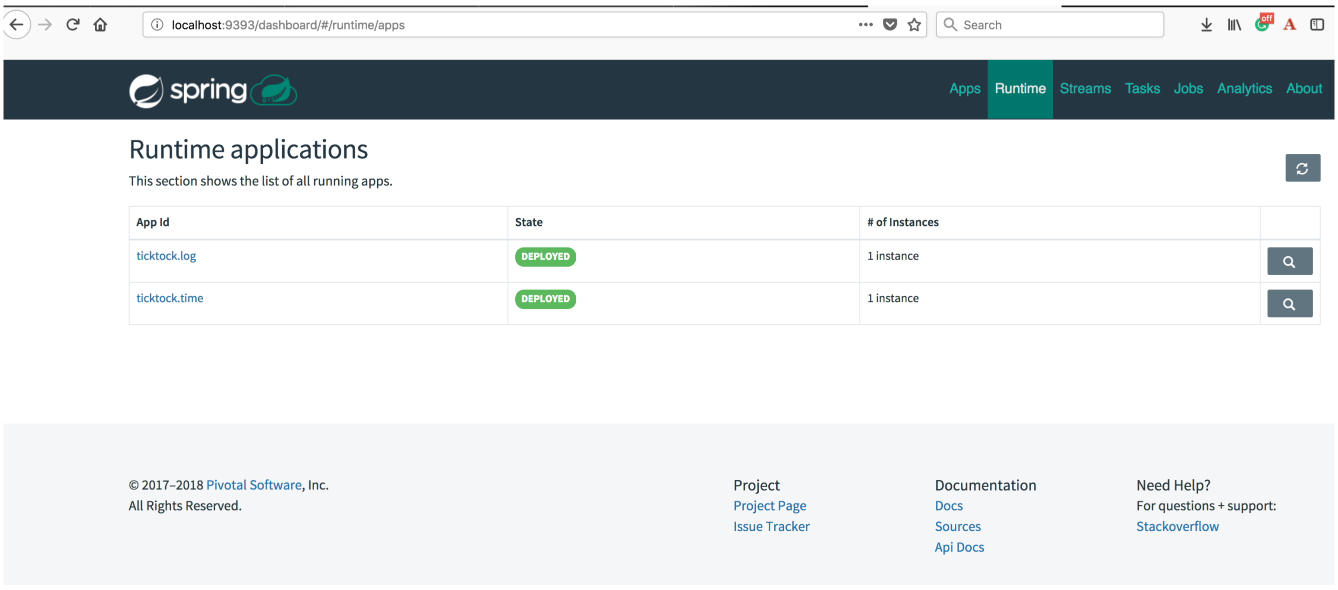
Project
Project Page
Issue Tracker

Documentation
Docs
Sources
Api Docs

Need Help?
For questions + support:
Stackoverflow

Figure 3. Step 4.3

Two running stream apps appears under `Runtime` tab:



Runtime applications

This section shows the list of all running apps.

App Id	State	# of Instances
ticktock.log	DEPLOYED	1 instance
ticktock.time	DEPLOYED	1 instance

© 2017–2018 Pivotal Software, Inc.
All Rights Reserved.

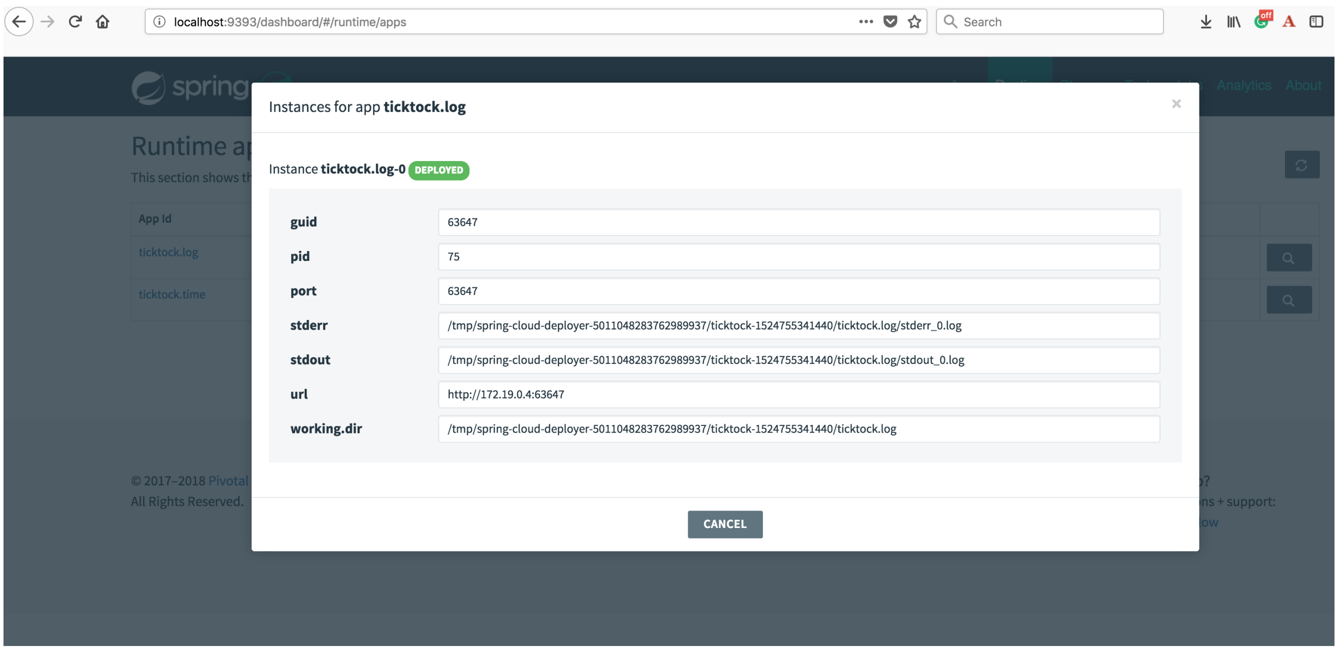
Project
Project Page
Issue Tracker

Documentation
Docs
Sources
Api Docs

Need Help?
For questions + support:
Stackoverflow

Figure 4. Step 4.4

Then I clicked on `ticktock.log` to determine the location of the stdout log file.



Instances for app **ticktock.log**

Instance **ticktock.log-0** DEPLOYED

guid	63647
pid	75
port	63647
stderr	/tmp/spring-cloud-deployer-5011048283762989937/ticktock-1524755341440/ticktock.log/stderr_0.log
stdout	/tmp/spring-cloud-deployer-5011048283762989937/ticktock-1524755341440/ticktock.log/stdout_0.log
url	http://172.19.0.4:63647
working.dir	/tmp/spring-cloud-deployer-5011048283762989937/ticktock-1524755341440/ticktock.log

Figure 5. Step 4.5

Step 5 - Verify that events are being written to the ticktock log every second

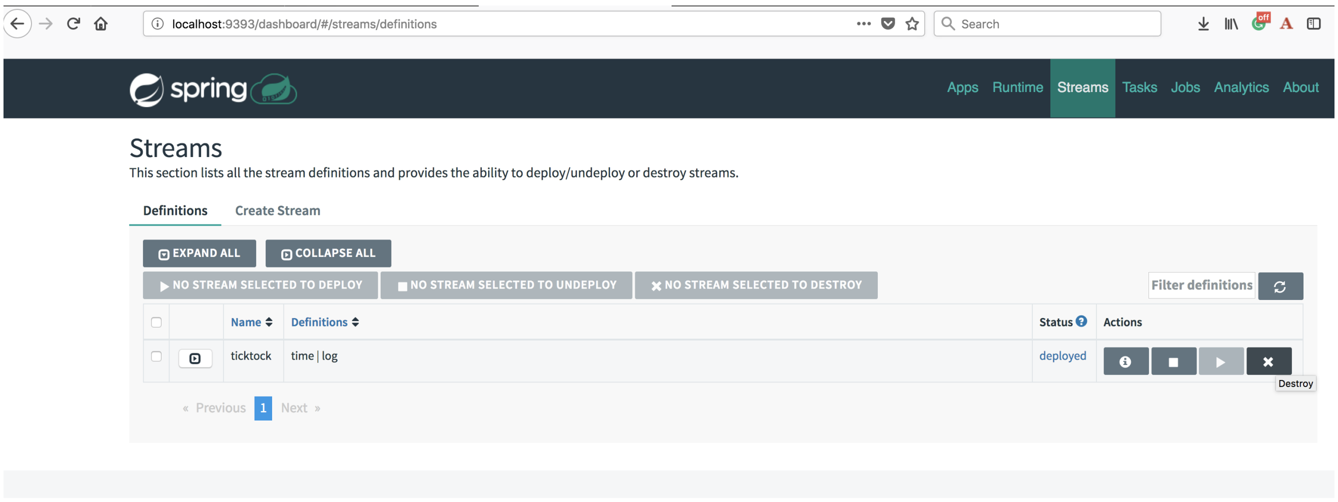
```
$ log=/tmp/spring-cloud-deployer-5011048283762989937/ticktock-
1524755341440/ticktock.log/stdout_0.log
$ tail -f $log
```

This last step did not work 💣. Obvious (I thinked) … the above file is inside the container! So I started to investigate how to read this file by using Docker reading the Spring Cloud Data Flow Reference Guide and yes, in that, I found the solution to type the correct command:

```
$ docker exec -it dataflow-server tail -f $log
```

My conclusion was: the Quick Start page needs to be fixed (on this step) to present the above command correctly.

Step 6 - Delete a Stream



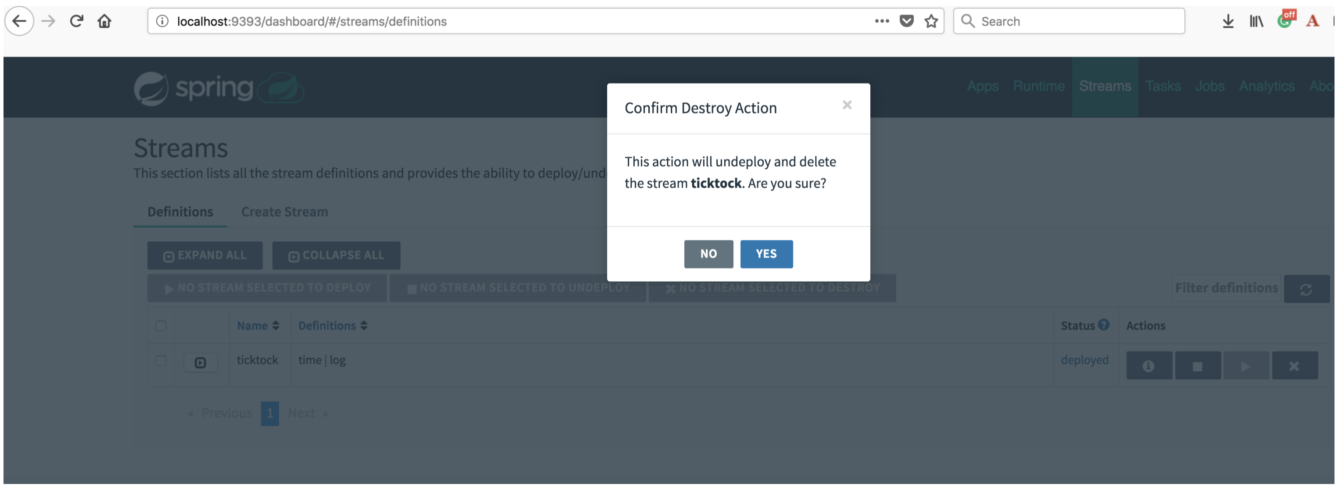
localhost:9393/dashboard/#/streams/definitions
Apps
Runtime
Streams
Tasks
Jobs
Analytics
About
Definitions
Create Stream
EXPAND ALL
COLLAPSE ALL
NO STREAM SELECTED TO DEPLOY
NO STREAM SELECTED TO UNDEPLOY
NO STREAM SELECTED TO DESTROY
Filter definitions
Destroy

Streams

This section lists all the stream definitions and provides the ability to deploy/undeploy or destroy streams.

Name	Definitions	Status	Actions
ticktock	time | log	deployed	

Figure 6. Step 6.1



Confirm Destroy Action

This action will undeploy and delete the stream **ticktock**. Are you sure?

Figure 7. Step 6.2

Step 7 - Destroy the Quick Start environment

I opened another shell and typed:

```
docker-compose down
```

Step 8 - Remove all finished containers

To remove all finished containers, I did:

```
docker ps -a | grep Exit | cut -d ' ' -f 1 | xargs sudo docker rm
```

Step 9 - Change docker-compose.yml to use RabbitMQ, MySQL and Redis and test it

In order to change the contents of `docker-compose.yml` I cloned the repository of this tutorial:

```
$ git clone https://github.com/paulojeronimo/spring-cloud-dataflow-tutorial tutorial
```

After download this repository, I applied a patch on `docker-compose.yml`:

```
$ patch docker-compose.yml < tutorial/files/docker-compose.yml.patch
```

This is the contents of the file `docker-compose.yml.patch`:

```
--- docker-compose.yml.original 2018-05-01 00:35:11.000000000 +0100
+++ docker-compose.yml  2018-05-01 00:44:49.000000000 +0100
@@ -1,31 +1,38 @@
 version: '3'

 services:
-  kafka:
-    image: wurstmeister/kafka:0.10.1.0
+  rabbitmq:
+    image: rabbitmq:3.7
     expose:
-      - "9092"
+      - "5672"
+  mysql:
+    image: mariadb:10.2
     environment:
-      - KAFKA_ADVERTISED_PORT=9092
-      - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
-    depends_on:
-      - zookeeper
-  zookeeper:
-    image: wurstmeister/zookeeper
+      MYSQL_DATABASE: dataflow
+      MYSQL_USER: root
+      MYSQL_ROOT_PASSWORD: rootpw
     expose:
-      - "2181"
-    environment:
-      - KAFKA_ADVERTISED_HOST_NAME=zookeeper
+      - 3306
+  redis:
+    image: redis:2.8
+    expose:
+      - "6379"
   dataflow-server:
     image: springcloud/spring-cloud-dataflow-server-local:1.4.0.RELEASE
     container_name: dataflow-server
     ports:
       - "9393:9393"
     environment:
-      -
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.br
okers=kafka:9092
-      -
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.kafka.binder.zk
Nodes=zookeeper:2181
+      - spring_datasource_url=jdbc:mysql://mysql:3306/dataflow
+      - spring_datasource_username=root
+      - spring_datasource_password=rootpw
+      - spring_datasource_driver-class-name=org.mariadb.jdbc.Driver
+      -
spring.cloud.dataflow.applicationProperties.stream.spring.rabbitmq.host=rabbitmq
+      - spring.cloud.dataflow.applicationProperties.stream.spring.redis.host=redis
+      - spring_redis_host=redis
     depends_on:
-      - kafka
+      - rabbitmq
+      - mysql
   app-import:
     image: alpine:3.7
     depends_on:
@@ -36,7 +43,7 @@
         do
           sleep 1;
         done;
-        wget -qO- 'http://dataflow-server:9393/apps' --post
-data='uri=http://bit.ly/Celsius-SR1-stream-applications-kafka-10-maven&force=true';
+        wget -qO- 'http://dataflow-server:9393/apps' --post
-data='uri=http://bit.ly/Celsius-SR1-stream-applications-rabbit-maven&force=true';
        echo 'Stream apps imported'
        wget -qO- 'http://dataflow-server:9393/apps'  --post
-data='uri=http://bit.ly/Clark-GA-task-applications-maven&force=true';
        echo 'Task apps imported'"
```

Now I could repeat the steps from this tutorial beggining from step 2.

More details about the configuration that I made in `docker-compose.yml` throught the use of of my patch can be found in "Docker Compose Customization" on Spring Cloud Data Flow Reference Guide.

Conclusions

Positive points:

- The Docker configuration to run SCDF is well done and absolutely finished. I did not have to do anything to quickly get things running.
- Running SCDF on Docker on my machine was extremely fast.
- Using Docker Compose, the switching from Kafka to RabbitMQ was quite simple (after I resolve a problem with a non documented step).
- The UI interface (provided by the dashboard) is pretty simple. All sources, sinks, and processors are elements that could be configured through the UI. The stream creation using the UI is very basic (for me this is nice) but can also be configured.
    - Even though I have not done configurations on elements of UI, a good video that shows how to do this is Spring Flo for Spring Cloud Data Flow1.0.0M3.

Negative points:

- The Quick Start page that I used to follow steps has a bug like I said.
- Even the Spring Cloud Data Flow Reference Guide has some missed points that I had to discover myself.

References

- Quick Start page
- Spring Cloud Data Flow Reference Guide

Lab 2: Run SCDF in PCF Dev

By doing this lab, my intention was to discover if it was easy to create an infrastructure prepared to run SCDF in Pivotal Cloud Foundry. Also, one of my goals was to switch from Kafka to RabbitMQ.

To start this lab, in order to have sufficient memory for executing the following steps, I had to quit Docker.

Prerequisites

1. Lab 1 executed.
2. VirtualBox installed (PCF Dev uses it).
3. An account on Pivotal Web Services (do download PCF Dev binary).

Step 1 - Download and install PCF Dev

First of all, after I downloaded the PCF Dev binary, I typed the following commands to install it:

```
$ unzip pcfdev-v0.30.0+PCF1.11.0-osx.zip
Archive:  pcfdev-v0.30.0+PCF1.11.0-osx.zip
  inflating: pcfdev-v0.30.0+PCF1.11.0-osx

$ ./pcfdev-v0.30.0+PCF1.11.0-osx
Plugin successfully upgraded. Current version: 0.30.0. For more info run: cf dev help
```

To check the current version, I typed:

```
$ cf dev version
PCF Dev version 0.30.0 (CLI: 850ae45, OVA: 0.549.0)
```

Step 2 - Start PCF Dev

I started PCF Dev with more memory than default (4096 MB makes the services very hard to work):

```
$ cf dev start -m 8192
Downloading VM...
Progress: |====================>| 100%
VM downloaded.
Allocating 8192 MB out of 16384 MB total system memory (9217 MB free).
Importing VM...
Starting VM...
Provisioning VM...
Waiting for services to start...
7 out of 58 running
7 out of 58 running
7 out of 58 running
7 out of 58 running
40 out of 58 running
56 out of 58 running
58 out of 58 running
 _______  _______  _______    ______   _______  __   __
|       ||       ||       |  |      | |       ||  | |  |
|    _  ||       ||    ___|  |  _    ||    ___||  |_|  |
|   |_| ||       ||   |___   | | |   ||   |___ |       |
|    ___||      _||    ___|  | |_|   ||    ___||       |
|   |    |     |_ |   |      |       ||   |___  |     |
|___|    |_______||___|      |______| |_______|  |___|
is now running.
To begin using PCF Dev, please run:
   cf login -a https://api.local.pcfdev.io --skip-ssl-validation
Apps Manager URL: https://apps.local.pcfdev.io
Admin user => Email: admin / Password: admin
Regular user => Email: user / Password: pass
```

The command above takes some time (around 15 minutes on my MacBook Pro) because it has to download the VM (in OVA format) and starts a bunch of services (the most time-consuming part of the procedure). After it finishes I verified the directory structure where the machine was installed:

```
$ tree ~/.pcfdev/
/Users/pj/.pcfdev/
|-- ova
|   `-- pcfdev-v0.549.0.ova
|-- token
`-- vms
    |-- key.pem
    |-- pcfdev-v0.549.0
    |   |-- Logs
    |   |   |-- VBox.log
    |   |   `-- VBox.log.1
    |   |-- pcfdev-v0.549.0-disk1.vmdk
    |   |-- pcfdev-v0.549.0.vbox
    |   `-- pcfdev-v0.549.0.vbox-prev
    `-- vm_config

4 directories, 9 files
```

Step 3 - Login into PCF Dev

When PCF Dev was started, I made my login:

```
$ cf login -a https://api.local.pcfdev.io --skip-ssl-validation -u admin -p admin -o
pcfdev-org
API endpoint: https://api.local.pcfdev.io
Authenticating...
OK

Targeted org pcfdev-org

Targeted space pcfdev-space



API endpoint:   https://api.local.pcfdev.io (API version: 2.82.0)
User:           admin
Org:            pcfdev-org
Space:          pcfdev-space
```

Step 4 - Create services

After that, I created three services:

MySQL service:

```
$ cf create-service p-mysql 512mb df-mysql
Creating service instance df-mysql in org pcfdev-org / space pcfdev-space as admin...
OK
```

RabbitMQ service:

```
$ cf create-service p-rabbitmq standard df-rabbitmq
Creating service instance df-rabbitmq in org pcfdev-org / space pcfdev-space as
admin...
OK
```

Redis service:

```
$ cf create-service p-redis shared-vm df-redis
Creating service instance df-redis in org pcfdev-org / space pcfdev-space as admin...
OK
```

Step 5 - Download some jars

```
$ cd ~/labs/spring-cloud-dataflow
$ wget -c http://repo.spring.io/release/org/springframework/cloud/spring-cloud-
dataflow-server-cloudfoundry/1.4.0.RELEASE/spring-cloud-dataflow-server-cloudfoundry-
1.4.0.RELEASE.jar
```

Step 6 - Create a manifest.yml file

I copied `tutorial/file/manifest.yml` file to the current dir:

```
$ cp ../tutorial/files/manifest.yml .
```

The `tutorial` dir was created on lab 1, step 9.

The content of this file is presented below:

```
---
applications:
- name: dataflow-server
  memory: 2g
  disk_quota: 2g
  path: spring-cloud-dataflow-server-cloudfoundry-1.4.0.RELEASE.jar
  buildpack: java_buildpack
  services:
    - df-mysql
    - df-redis
  env:
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL: https://api.local.pcfdev.io
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG: pcfdev-org
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE: pcfdev-space
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN: local.pcfdev.io
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES: df-rabbitmq
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME: admin
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD: admin
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION: true
    MAVEN_REMOTE_REPOSITORIES_REPO1_URL: https://repo.spring.io/libs-snapshot
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_MEMORY: 512
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_DISK: 512
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_INSTANCES: 1
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_BUILDPACK: java_buildpack
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_ENABLE_RANDOM_APP_NAME_PREFIX: false
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES: df-mysql
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_MEMORY: 512
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_DISK: 512
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_INSTANCES: 1
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_BUILDPACK: java_buildpack
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_ENABLE_RANDOM_APP_NAME_PREFIX: false
    SPRING_CLOUD_DATAFLOW_FEATURES_EXPERIMENTAL_TASKSENABLED: true
```

The session "Deploying using a Manifest" of the document "Spring Cloud Data Flow Server for Cloud Foundry" details this file.

Step 7 - Push spring-cloud-dataflow-server-cloudfoundry to PCF Dev

```
$ cf push
Pushing from manifest to org pcfdev-org / space pcfdev-space as admin...
Using manifest file /Users/pj/labs/spring-cloud-dataflow/manifest.yml
Getting app info...
Creating app with these attributes...
+ name:        dataflow-server
  path:        /Users/pj/labs/spring-cloud-dataflow/spring-cloud-dataflow-server-
cloudfoundry-1.4.0.RELEASE.jar
```

```
+ buildpack:    java_buildpack
+ disk quota:   2G
+ memory:       2G
  services:
+   df-mysql
+   df-redis
  env:
+   MAVEN_REMOTE_REPOSITORIES_REPO1_URL
+   SPRING_CLOUD_DATAFLOW_FEATURES_EXPERIMENTAL_TASKSENABLED
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_BUILDPACK
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_DISK
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_ENABLE_RANDOM_APP_NAME_PREFIX
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_INSTANCES
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_MEMORY
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_BUILDPACK
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_DISK
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_ENABLE_RANDOM_APP_NAME_PREFIX
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_INSTANCES
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_MEMORY
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL
+   SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME
  routes:
+   dataflow-server.local.pcfdev.io

Creating app dataflow-server...
Mapping routes...
Binding services...
Comparing local files to remote cache...
Packaging files to upload...
Uploading files...
 71.59 MiB / 71.59 MiB
[======================================================================================
=====================================] 100.00% 2s

Waiting for API to complete processing files...

Staging app and tracing logs...
   Downloading java_buildpack...
   Downloaded java_buildpack (244.5M)
   Creating container
   Successfully created container
   Downloading app package...
   Downloaded app package (71.6M)
   Staging...
   -----> Java Buildpack Version: v3.13 (offline) |
https://github.com/cloudfoundry/java-buildpack.git#03b493f
   -----> Downloading Open Jdk JRE 1.8.0_121 from https://java-
buildpack.cloudfoundry.org/openjdk/trusty/x86_64/openjdk-1.8.0_121.tar.gz (found in
cache)
          Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.0s)
   -----> Downloading Open JDK Like Memory Calculator 2.0.2_RELEASE from https://java-
buildpack.cloudfoundry.org/memory-calculator/trusty/x86_64/memory-calculator-
2.0.2_RELEASE.tar.gz (found in cache)
          Memory Settings: -Xmx1363148K -XX:MaxMetaspaceSize=209715K -Xms1363148K
-XX:MetaspaceSize=209715K -Xss699K
   -----> Downloading Container Certificate Trust Store 2.0.0_RELEASE from
https://java-buildpack.cloudfoundry.org/container-certificate-trust-store/container-
certificate-trust-store-2.0.0_RELEASE.jar (found in cache)
          Adding certificates to .java-
buildpack/container_certificate_trust_store/truststore.jks (0.4s)
   -----> Downloading Spring Auto Reconfiguration 1.10.0_RELEASE from https://java-
buildpack.cloudfoundry.org/auto-reconfiguration/auto-reconfiguration-
1.10.0_RELEASE.jar (found in cache)
   Exit status 0
   Staging complete
   Uploading droplet, build artifacts cache...
   Uploading build artifacts cache...
   Uploading droplet...
   Uploaded build artifacts cache (108B)
   Uploaded droplet (116.9M)
   Uploading complete
   Destroying container
   Successfully destroyed container

Waiting for app to start...

name:              dataflow-server
requested state:   started
instances:         1/1
usage:             2G x 1 instances
routes:            dataflow-server.local.pcfdev.io
last uploaded:     Tue 01 May 19:48:22 WEST 2018
stack:             cflinuxfs2
buildpack:         java_buildpack
start command:     CALCULATED_MEMORY=$($PWD/.java-buildpack/open_jdk_jre/bin/java-
buildpack-memory-calculator-2.0.2_RELEASE
                   -memorySizes=metaspace:64m..,stack:228k..
-memoryWeights=heap:65,metaspace:10,native:15,stack:10
-memoryInitials=heap:100%,metaspace:100%
                   -stackThreads=300 -totMemory=$MEMORY_LIMIT) && JAVA_OPTS="-
Djava.io.tmpdir=$TMPDIR
                   -XX:OnOutOfMemoryError=$PWD/.java
-buildpack/open_jdk_jre/bin/killjava.sh $CALCULATED_MEMORY
                   -Djavax.net.ssl.trustStore=$PWD/.java
-buildpack/container_certificate_trust_store/truststore.jks
```

```
                   -Djavax.net.ssl.trustStorePassword=java-buildpack-trust-store
-password" && SERVER_PORT=$PORT eval exec
                   $PWD/.java-buildpack/open_jdk_jre/bin/java $JAVA_OPTS -cp $PWD/.
org.springframework.boot.loader.JarLauncher

     state     since                    cpu      memory         disk          details
#0   running   2018-05-01T18:49:23Z   220.0%   782.6M of 2G   206M of 2G
```

Step 8 - Access the dashboard

After deployed, I pointed my browser to to following URL: http://dataflow-server.local.pcfdev.io/dashboard/.

I noticed that the dashboard gives me a information that it has no applications installed.

Step 9 - Use the shell

In my previous lab, I used the UI interface (through the browser) to create a stream. This time, however, I created the stream by using the command line. These were my steps:

```
$ wget -c http://repo.spring.io/release/org/springframework/cloud/spring-cloud-
dataflow-shell/1.4.0.RELEASE/spring-cloud-dataflow-shell-1.4.0.RELEASE.jar
```

```
$ java -jar spring-cloud-dataflow-shell-1.4.0.RELEASE.jar
  ____                              ____ _                __
 / ___| _ __  _ __(_)_ __   __ _   / ___| | ___  _   _  __| |
 \___ \| '_ \| '__| | '_ \ / _` | | |   | |/ _ \| | | |/ _` |
  ___) | |_) | |  | | | | | (_| | | |___| | (_) | |_| | (_| |
 |____/| .__/|_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____ |_|    _          __|___/                 __________
 |  _ \  __ _| |_ __ _  |  ___| | _____      __  \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /   \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/

1.4.0.RELEASE

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

I configured the shell to access my local PCF Dev instance:

```
server-unknown:>dataflow config server http://dataflow-server.local.pcfdev.io
Shell mode: classic, Server mode: classic
dataflow:>
```

I imported the applications:

```
dataflow:>app import --uri http://bit.ly/Celsius-SR1-stream-applications-rabbit-maven
Successfully registered 65 applications from [source.sftp, source.mqtt.metadata,
sink.mqtt.metadata, source.file.metadata, processor.tcp-client, source.s3.metadata,
source.jms, source.ftp, processor.transform.metadata, source.time, sink.mqtt,
sink.s3.metadata, processor.scriptable-transform, sink.log, source.load-generator,
processor.transform, source.syslog, sink.websocket.metadata, sink.task-launcher-
local.metadata, source.loggregator.metadata, source.s3, source.load-
generator.metadata, processor.pmml.metadata, source.loggregator, source.tcp.metadata,
processor.httpclient.metadata, sink.file.metadata, source.triggertask,
source.twitterstream, source.gemfire-cq.metadata, processor.aggregator.metadata,
source.mongodb, source.time.metadata, source.gemfire-cq, sink.counter.metadata,
source.http, sink.tcp.metadata, sink.pgcopy.metadata, source.rabbit, sink.task-
launcher-yarn, source.jms.metadata, sink.gemfire.metadata, sink.cassandra.metadata,
processor.tcp-client.metadata, processor.header-enricher, sink.throughput, sink.task-
launcher-local, processor.python-http, sink.aggregate-counter.metadata, sink.mongodb,
processor.twitter-sentiment, sink.log.metadata, processor.splitter, sink.hdfs-dataset,
source.tcp, processor.python-jython.metadata, source.trigger, source.mongodb.metadata,
processor.bridge, source.http.metadata, source.rabbit.metadata, sink.ftp, sink.jdbc,
source.jdbc.metadata, source.mqtt, processor.pmml, sink.aggregate-counter,
sink.rabbit.metadata, processor.python-jython, sink.router.metadata, sink.cassandra,
processor.filter.metadata, source.tcp-client.metadata, processor.header-
enricher.metadata, processor.groovy-transform, source.ftp.metadata, sink.router,
sink.redis-pubsub, source.tcp-client, processor.httpclient, sink.file, sink.websocket,
source.syslog.metadata, sink.s3, sink.counter, sink.rabbit, processor.filter,
source.trigger.metadata, source.mail.metadata, sink.gpfdist.metadata, sink.pgcopy,
processor.python-http.metadata, sink.jdbc.metadata, sink.gpfdist, sink.ftp.metadata,
processor.splitter.metadata, sink.sftp, sink.field-value-counter, processor.groovy-
filter.metadata, processor.twitter-sentiment.metadata, source.triggertask.metadata,
sink.hdfs, processor.groovy-filter, sink.redis-pubsub.metadata, source.sftp.metadata,
processor.bridge.metadata, sink.field-value-counter.metadata, processor.groovy-
transform.metadata, processor.aggregator, sink.sftp.metadata,
processor.tensorflow.metadata, sink.throughput.metadata, sink.hdfs-dataset.metadata,
sink.tcp, source.mail, sink.task-launcher-cloudfoundry.metadata,
source.gemfire.metadata, processor.tensorflow, source.jdbc, sink.task-launcher-
yarn.metadata, sink.gemfire, source.gemfire, source.twitterstream.metadata,
sink.hdfs.metadata, processor.tasklaunchrequest-transform, sink.task-launcher-
cloudfoundry, source.file, sink.mongodb.metadata, processor.tasklaunchrequest-
transform.metadata, processor.scriptable-transform.metadata]
```

I refresh my browser and noticed that, now, I had many applications installed (65 according to the last log).

I created the stream:

```
dataflow:>stream create --name httptest --definition "http | log" --deploy
Created new stream 'httptest'
Deployment request has been sent
```

I noticed that the stream was deployed:

```
dataflow:>stream list
╔═══════════╤═════════════════╤════════════════════════════════════════════════
════════════════════════════════════════════════════════════════╗
║Stream Name│Stream Definition│                 Status                 ║
╠═══════════╪═════════════════╪════════════════════════════════════════════════
════════════════════════════════════════════════════════════════╣
║httptest   │http | log       │The stream has been successfully deployed║
╚═══════════╧═════════════════╧════════════════════════════════════════════════
════════════════════════════════════════════════════════════════╝
```

I started another shell (leaving the current shell opened) and then I typed the following command:

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

name                            requested state   instances   memory   disk   urls
dataflow-server                 started           1/1         2G       2G
dataflow-server.local.pcfdev.io
dataflow-server-httptest-log    started           1/1         512M     512M
dataflow-server-httptest-log.local.pcfdev.io
dataflow-server-httptest-http   started           1/1         512M     512M
dataflow-server-httptest-http.local.pcfdev.io
```

In order to view the log output for `dataflow-server-httptest-log`, I typed:

```
$ cf logs dataflow-server-httptest-log
Retrieving logs for app dataflow-httptest-log in org pcfdev-org / space pcfdev-space
as admin...
```

Back to the first shell (running `spring-cloud-dataflow-shell`), I typed:

```
dataflow:>http post --target http://dataflow-server-httptest-http.local.pcfdev.io
--data "hello world"
> POST (text/plain;Charset=UTF-8) http://dataflow-httptest-http.local.pcfdev.io hello
world
> 202 ACCEPTED

dataflow:>http post --target http://dataflow-server-httptest-http.local.pcfdev.io
--data "paulo jeronimo"
> POST (text/plain) http://dataflow-server-httptest-http.local.pcfdev.io paulo
jeronimo
> 202 ACCEPTED
```

So, this lines appears in the output of the command `cf logs dataflow-server-httptest-log`:

```
   2018-04-29T14:15:04.21+0100 [APP/PROC/WEB/0] OUT 2018-04-29 13:15:04.215  INFO 6
--- [http.httptest-1] log.sink                                 : hello world
   2018-04-29T14:17:22.83+0100 [APP/PROC/WEB/0] OUT 2018-04-29 13:17:22.832  INFO 6
--- [http.httptest-1] log.sink                                 : paulo jeronimo
```

My last command was delete the created stream:

```
dataflow:>stream destroy --name httptest
Destroyed stream 'httptest'
dataflow:>exit
```

Step 10 - Stop PCF Dev

```
cf dev stop
```

Step 11 - (Optional) Destroy the PCF Dev environment

```
cf dev destroy
```

Conclusions

Positive points:

- The PCF Dev also makes the deployment of SCDF components very simple. The `cf` command makes easy, in a development environment, to push new sources, sinks, or processors to the servers.
- The shell interface provided by [spring-cloud-dataflow-shell] is extremely useful to automate things. This kind of tool is totally required in a DevOps world.

- Based on this lab, I can deduce that in a large corporation with Pivotal Cloud Foundry installed on its servers, SCDF is definitely prepared and integrated with this kind of infrastructure.

Negative points:

- The time to start PCF Dev on my machine was very slow compared to the time to up Docker containers with Docker compose. So, I would recommend the use of Docker in a developer environment instead of PCF Dev.

References

- "Getting Started with Spring Cloud Data Flow on PCF Dev" (oA)
- Spring Cloud Data Flow Samples Documentation
- Install PCF Dev CLI.

Lab 3: Creating a custom processor application

By doing this lab I wanted to discover the effort related to a creation of a new processor in a development environment. So, if a new processor was easy to create, I could assume that sources and sinks also could be.

Prerequisites

1. Lab 1 executed.
2. SDKMAN installed.
3. Maven installed (in my case, this was done via SDKMAN).

Step 1 - Create a rabbitmq Docker container

I looked for and created a rabbitmq container:

```
$ docker search rabbitmq
$ docker run -d -p 5672:5672 -p 15672:15672  --name rabbitmq rabbitmq
```

Step 2 - Download and start, manually, sources and sinks

I changed my work directory to the directory created on "Lab 1":

```
$ cd ~/labs/spring-cloud-dataflow
```

I downloaded some pre-existing sources and sinks:

```
$ wget -c https://repo.spring.io/libs-
release/org/springframework/cloud/stream/app/http-source-rabbit/1.3.1.RELEASE//http-
source-rabbit-1.3.1.RELEASE.jar
$ wget -c https://repo.spring.io/release/org/springframework/cloud/stream/app/log-
sink-rabbit/1.3.1.RELEASE/log-sink-rabbit-1.3.1.RELEASE.jar
```

I started them:

```
# shell 2
$ java -Dserver.port=8123 \
-Dhttp.path-pattern=/data \
-Dspring.cloud.stream.bindings.output.destination=sensorData \
-jar http-source-rabbit-1.3.1.RELEASE.jar

# shell 3
$ java -Dlog.level=WARN \
-Dspring.cloud.stream.bindings.input.destination=sensorData \
-jar log-sink-rabbit-1.3.1.RELEASE.jar
```

Note the use of the the parameters passed to source and sync.

I sent a post via curl:

```
# shell 1
$ curl -H "Content-Type: application/json" -X POST -d '{"id":"1","temperature":"100"}'
http://localhost:8123/data
```

I noticed the log that apperars on `shell 3`:

```
2018-05-01 20:23:16.955  WARN 23145 --- [Li0KHY-B-ej_w-1] log-sink
: {"id":"1","temperature":"100"}
```

I stopped all running micro services (on shells 2 and 3) and, also, the rabbitmq container:

```
$ docker stop rabbitmq
```

I closed all the other shells (2 and 3) and went back to shell 1.

Step 3 - Create and run a new processor manually

To use Spring Initializr via command line, I installed Spring Boot CLI via SDKMAN:

```
$ sdk install springboot
```

I generated the initial `transformer` code using `spring init`:

```
$ mkdir transformer && cd $_
$ spring init -d=cloud-stream -g=io.spring.stream.sample -a=transformer -n=transformer
--package-name=io.spring.stream.sample ../transformer.zip
$ unzip ../transformer.zip
```

I saw the created directory structure:

```
$ tree
.
|-- mvnw
|-- mvnw.cmd
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- io
    |   |       `-- spring
    |   |           `-- stream
    |   |               `-- sample
    |   |                   `-- TransformerApplication.java
    |   `-- resources
    |       `-- application.properties
    `-- test
        `-- java
            `-- io
                `-- spring
                    `-- stream
                        `-- sample
                            `-- TransformerApplicationTests.java

14 directories, 6 files
```

I added the code `Transformer.java` to the `transformer` project:

```
$ cp ../tutorial/files/Transformer.java src/main/java/io/spring/stream/sample/
```

This is the code that was added:

```
package io.spring.stream.sample;

import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Processor;

import java.util.HashMap;
import java.util.Map;

@EnableBinding(Processor.class)
public class Transformer {

    @StreamListener(Processor.INPUT)
    @Output(Processor.OUTPUT)
    public Map<String, Object> transform(Map<String, Object> doc) {
        Map<String, Object> map = new HashMap<>();
        map.put("sensor_id", doc.getOrDefault("id", "-1"));
        map.put("temp_val", doc.getOrDefault("temperature", "-999"));
        return map;
    }
}
```

I patched the `pom.xml`:

```
$ patch pom.xml < ../tutorial/files/pom.xml.patch
patching file pom.xml
```

This is the patch that was applied:

```
--- pom.xml.original    2018-04-30 23:07:31.000000000 +0100
+++ pom.xml 2018-04-30 23:07:38.000000000 +0100
@@ -28,10 +28,14 @@
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
-           <artifactId>spring-cloud-stream</artifactId>
+           <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
        </dependency>

        <dependency>
+           <groupId>org.springframework</groupId>
+           <artifactId>spring-web</artifactId>
+       </dependency>
+       <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
```

I changed the code of `TransformerApplicationTests.java` by applying another patch:

```
$ patch src/test/java/io/spring/stream/sample/TransformerApplicationTests.java <
../tutorial/files/TransformerApplicationTests.java.patch
patching file src/test/java/io/spring/stream/sample/TransformerApplicationTests.java
```

This is the source code of the applied patch:

```
--- TransformerApplicationTests.java.original   2018-04-30 21:40:01.000000000 +0100
+++ TransformerApplicationTests.java    2018-04-30 21:40:35.000000000 +0100
@@ -2,15 +2,35 @@

 import org.junit.Test;
 import org.junit.runner.RunWith;
+import org.springframework.beans.factory.annotation.Autowired;
 import org.springframework.boot.test.context.SpringBootTest;
 import org.springframework.test.context.junit4.SpringRunner;

+import java.util.HashMap;
+import java.util.Map;
+
+import static org.assertj.core.api.Assertions.assertThat;
+import static org.assertj.core.api.Assertions.entry;
+
 @RunWith(SpringRunner.class)
-@SpringBootTest
+@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
 public class TransformerApplicationTests {

-   @Test
-   public void contextLoads() {
-   }
+    @Autowired
+    private Transformer transformer;
+
+    @Test
+    public void simpleTest() {
+        Map<String, Object> resultMap = transformer.transform(createInputData());
+        assertThat(resultMap).hasSize(2)
+                .contains(entry("sensor_id", "1"))
+                .contains(entry("temp_val", "100"));
+    }

+    private Map<String, Object> createInputData() {
+        HashMap<String, Object> inputData = new HashMap<>();
+        inputData.put("id", "1");
+        inputData.put("temperature", "100");
+        return inputData;
+    }
 }
```

Finally, I built the package using Maven:

```
$ ./mvnw clean package
$ cp target/transformer-0.0.1-SNAPSHOT.jar ..
$ cd ..
```

I started rabbitmq (it it was stopped before):

```
$ docker start rabbitmq
```

I started the services:

```
# shell 2
$ java  -Dserver.port=8123 \
        -Dhttp.path-pattern=/data \
        -Dspring.cloud.stream.bindings.output.destination=sensorData \
        -jar http-source-rabbit-1.3.1.RELEASE.jar

# shell 3
$ java  -Dserver.port=8090 \
        -Dspring.cloud.stream.bindings.input.destination=sensorData \
        -Dspring.cloud.stream.bindings.output.destination=normalizedSensorData \
        -jar transformer-0.0.1-SNAPSHOT.jar

# shell 4
$ java  -Dlog.level=WARN \
        -Dspring.cloud.stream.bindings.input.destination=normalizedSensorData \
        -jar log-sink-rabbit-1.3.1.RELEASE.jar
```

I tested the services using curl:

```
# shell 1
$ curl -H "Content-Type: application/json" -X POST -d '{"id":"1","temperature":"100"}'
http://localhost:8123/data
```

I noticed the following output on `shell 4`:

```
2018-05-01 20:33:30.715  WARN 23584 --- [k2tWV9V6IhDPg-1] log-sink
: {"sensor_id":"24e8778c-c275-237b-5e0a-f9f96eea8d62","temp_val":"-999"}
```

I stopped all micro services and left the rabbitmq container running to execute my next steps.

Step 4 - Run the new processor using spring-cloud-dataflow-server-local

I downloaded the local spring-cloud-dataflow server:

```
$ wget -c https://repo.spring.io/release/org/springframework/cloud/spring-cloud-
dataflow-server-local/1.4.0.RELEASE/spring-cloud-dataflow-server-local-
1.4.0.RELEASE.jar
```

I started it on current shell (1):

```
$ java -jar spring-cloud-dataflow-server-local-1.4.0.RELEASE.jar
```

Using the spring-cloud-dataflow shell, I manually register the components and created the stream:

```
# shell 2
$ java -jar spring-cloud-dataflow-shell-1.4.0.RELEASE.jar
dataflow:>app list
No registered apps.
You can register new apps with the 'app register' and 'app import' commands.

app register --type source --name http --uri file:///Users/pj/labs/spring-cloud-
dataflow/http-source-rabbit-1.3.1.RELEASE.jardataflow:>app register --type source
--name http --uri file:///Users/pj/labs/spring-cloud-dataflow/http-source-rabbit-
1.3.1.RELEASE.jar
Successfully registered application 'source:http'

dataflow:>app register --type sink --name log --uri file:///Users/pj/labs/spring-
cloud-dataflow/log-sink-rabbit-1.3.1.RELEASE.jar
Successfully registered application 'sink:log'

dataflow:>app register --type processor --name transformer --uri
file:///Users/pj/labs/spring-cloud-dataflow/transformer-0.0.1-SNAPSHOT.jar
Successfully registered application 'processor:transformer'

dataflow:>stream create --name httpIngest --definition "http --server.port=8123 --path
-pattern=/data | transformer --server.port=8090 | log --level=WARN" --deploy
Created new stream 'httpIngest'
Deployment request has been sent
```

When the stream was created, I saw in the server log:

```
# shell 1
2018-05-01 01:47:48.121  INFO 15385 --- [nio-9393-exec-2]
o.s.c.d.spi.local.LocalAppDeployer       : Deploying app with deploymentId
httpIngest.log instance 0.
   Logs will be in /var/folders/tx/xxk5qr1j5txfm5yvs3p9k9bc0000gn/T/spring-cloud-
deployer-4223749192815736035/httpIngest-1525135668085/httpIngest.log
```

I checked if the stream was deployed and exited the `spring-cloud-dataflow-shell`:

```
# shell 2
dataflow:>stream list
...
dataflow:>exit
```

I started `tail` to monitoring the `stdout_0.log` file in background:

```
$ tail -f /var/folders/tx/xxk5qr1j5txfm5yvs3p9k9bc0000gn/T/spring-cloud-deployer-
4223749192815736035/httpIngest-1525135668085/httpIngest.log/stdout_0.log &
```

I sent a POST to source via `curl` and saw an output presented by then running instance of `tail`:

```
$ curl -H "Content-Type: application/json" -X POST -d '{"id":"1","temperature":"100"}'
http://localhost:8123/data
2018-05-01 01:52:34.997  WARN 15445 --- [er.httpIngest-1] log-sink
: {"sensor_id":"43738bcb-2af5-917e-61ff-4c2837c5de83","temp_val":"-999"}
```

I stopped the spring-cloud-dataflow server:

```
# shell 1
<Ctrl+C>
```

I killed the `tail` process on `shell 2` and closed it:

```
$ pkill tail
$ exit
```

I went back to `shell 1`.

Step 5 - Stop and destroy rabbitmq container

```
$ docker stop rabbitmq
$ docker ps -a | grep 'Exited.*rabbitmq$' | cut -d ' ' -f 1 | xargs docker rm
```

Conclusions

Positive points:

- Create a project is very quick and simple. Spring Initializr provides these features.
- Create a new processor is also very easy, like to do any other Spring Boot application.
    - Any team with a development background on creating Spring Boot applications has the necessary knowledge. Only a few new annotations/APIs must be studied to getting

started.

- The use of Docker in development environment make things extremally easier.
- The source code for sources (like `http`) and sinks (like `log`) are quite simple and available on GitHub (see the references below). So you can see them to guide yourself in the development of your own components.

Negative points:

- The Spring documentation, one more time, needs to be more accurate to help the developers better.
  - Some bugs I have to solve myself:
    - The generated code for `transformer` came with an incorrect `pom.xml`, so I had to patch it.
  - Some points were omitted or could be better explained in the Spring documentation:
    - How to quickly install and use RabbitMQ in a development environment in order to test Spring Cloud Data Flow? I found a solution to this question in this post.
    - I needed to modify the generated `pom.xml` in order to make the `Transformer` test run successfully. For this, I had to search the Internet until I found a solution on Stack Overflow for my problem. In my point of view, Spring Initializr failed to generate the correct `pom.xml` or I passed wrong arguments to it.

References

- Stream Developer Guide
- Installing the CLI
- Spring Cloud Stream App Starters
- Http Source (GitHub)
- Log Sink (GitHub)

Lab 4: Setting up a Windows environment to develop with SCDF

In this lab, my intention was to discover if I can easily setup a **Windows 10** development environment (using a non-administrative account (`user1`)) to run SCDF manually (without Docker or PCF Dev).

A Window admin account is required in many corporations that obligates their employees to use such environment. So, in this lab, I tried to install the maximum possible quantity of software that would not require me admin rights.

I didn't touch any Java code and in this lab (and in the next three as well). That was because I was only reviewing (or fixing or maybe growing) some concepts that I already presented in previous labs. But anyway, by creating theses labs I reviewed the fundamentals of both technologies from a perspective of a Windows developer.

So, to do the next labs I needed:

1. A Windows 10 environment.
2. A Bash shell installed on it (Git Bash).
3. A Java Development Kit (JDK) (>= 1.8).
4. A RabbitMQ instance started (it will be used in this state during the next windows labs).
5. A clone (or zip) of the repository that contains this tutorial.

Prerequisites

A Windows 10 environment. I started one from a VMware snapshot.

Step 1 - Creating of user1

I followed the Windows steps to create a new user called `user1` with no admin rights. After that, I used this account to do the next steps.

Step 2 - GitBash installation

I downloaded and extracted Git for Windows Portable on `C:\Portable\Git`.

Step 3 - Java Development Kit installation

I downloaded and extracted jdkPortable on `C:\Portable\JDK`.

I have setup up the variables `JAVA_HOME` and `PATH` (appending `%JAVA_HOME%\bin` to it).

Step 4 - RabbitMQ installation

A simple alternative to starting RabbitMQ without needing an administrator account would be download and executing RabbitMQ portable. I tried to do this. But this didn't work in Windows 10 like worked for Windows 7.

In Windows 10 the most rapidly form to leave problems behind was by following the installation steps for RabbitMQ. This required admin rights. But, to move on without wasting time on other alternatives, this was the simplest solution. When all installation procedures was done, I checked if the RabbitMQ service was running by searching it on windows `services`.

Finally, If I could use Docker on Windows, obviously I would use a RabbitMQ container like I did in lab 3.

Step 5 - Download this repository

I just typed the following commands by openning the Git Bash shell:

```
$ cd /c
$ git clone https://github.com/paulojeronimo/spring-cloud-dataflow-tutorial scdf-
tutorial
```

This created the repository's clone for me (on `C:\scdf-tutorial`).

Lab 5: Connecting a source to a sink using Spring Cloud Stream

Prerequisites

Lab 4 executed.

Step 1 - Download some Spring Cloud Stream Apps

To start this lab my first step was to associate all files with `.sh` extension with `git-bash.exe`.

After that, I clicked on `C:\scdf-tutorial\files\get-lab5-jars.sh`.

> `get-lab5-jars.sh` is a script that I created to download two jars for me with the following commands:
>
> ```
> curl -L -O -C -
> http://central.maven.org/maven2/org/springframework/cloud/stream/app/ht
> tp-source-rabbit/1.3.1.RELEASE/http-source-rabbit-1.3.1.RELEASE.jar
>
> curl -L -O -C -
> http://central.maven.org/maven2/org/springframework/cloud/stream/app/lo
> g-sink-rabbit/1.3.1.RELEASE/log-sink-rabbit-1.3.1.RELEASE.jar
> ```

After the download finishes I noticed that the presence of two files (jars) on `C:\scdf-tutorial\files`:

1. `http-source-rabbit-1.3.1.RELEASE.jar`
2. `log-sink-rabbit-1.3.1.RELEASE.jar`

Step 2 - Start http-source-rabbit

To open a window that runs the following Java command, I clicked `http-source-rabbit.bat`.

```
java -Dserver.port=8123 -Dhttp.path-pattern=/data
-Dspring.cloud.stream.bindings.output.destination=sensorData -jar http-source-rabbit-
1.3.1.RELEASE.jar
```

Step 3 - Start log-sink-rabbit

I also clicked `log-sink-rabbit.bat`. This script opened another window running the following command:

```
java -Dlog.level=WARN -Dspring.cloud.stream.bindings.input.destination=sensorData -jar
log-sink-rabbit-1.3.1.RELEASE.jar
```

Step 4 - Send a POST request and get a result

I opened the Git Bash shell and typed the following command:

```
curl -s -H "Content-Type: application/json" -X POST -d
'{"id":"1","temperature":"100"}' http://localhost:8123/data
```

I noticed some log output on `http-source-rabbit` window and, after that, this output on `log-sink-rabbit` window:

```
2018-05-... WARN 4222 --- [tp.httpIngest-1] log-sink                                 :
{"id":"1","temperature":"100"}
```

I finished the `log-sink-rabbit` execution by pressing `Ctrl+C` on its window. Also, to terminate this lab, I did the same in `http-source-rabbit` window.

Notes/conclusions

In this lab, I basically review things that were done behind the scenes from the previous labs. Two microservices where executed independently talking to each other through a message broker (RabbitMQ). The "channel" used in this conversation (`sensorData`) was passed to the following properties:

1. On source (`http-source`): `spring.cloud.stream.bindings.output.destination`.
2. On sink (`log-sink`): `spring.cloud.stream.bindings.input.destination`.

The name of the "channel" could be anything I want. But it needs to be the same for both properties.

The `http-source` is a kind of microservice that produces an output. This output is produced when the source receives a POST request in the path `/data`. The default implementation os this microservice only pass along the received data to the message broker. So the destination of its output is specified by this property.

The `log-sink` is a kind of microservice that consumes an input. In this case, when some input is available on the message broker, it will be consumed (removed from the "channel") and presented in the log output of the microservice. So, I must specify to the `log-sink` where the input will be available by configuring this property.

Both `http-source` and `log-sink` were Spring Boot applications. Spring Boot applications can be scaled up in Real-Time in many ways. So, `http-source` and `log-sink` can be easily scaled up and, also, independently!

Lab 6: Manually using SCDF (server local/shell)

Prerequisites

Lab 5 executed.

Step 1 - Download Spring Cloud Data Flow jars (server-local and shell)

I clicked on `C:\scdf-tutorial\files\get-lab6-jars.sh`. This script only executes the following commands to download the two jars needed by this lab:

```
curl -L -O -C - http://central.maven.org/maven2/org/springframework/cloud/spring-cloud-dataflow-server-local/1.4.0.RELEASE/spring-cloud-dataflow-server-local-1.4.0.RELEASE.jar

curl -L -O -C - http://central.maven.org/maven2/org/springframework/cloud/spring-cloud-dataflow-shell/1.4.0.RELEASE/spring-cloud-dataflow-shell-1.4.0.RELEASE.jar
```

Step 2 - Start Spring Cloud Data Flow (server-local and shell)

I clicked `spring-cloud-dataflow-server-local.bat` to start SCDF local with the following command:

```
java -jar spring-cloud-dataflow-server-local-1.4.0.RELEASE.jar
```

After that, I also clicked on `spring-cloud-dataflow-shell.bat`. This script started SCDF shell through the following command:

```
java -jar spring-cloud-dataflow-shell-1.4.0.RELEASE.jar
```

Step 3 - Register applications, create and deploy a stream

I typed the following commands on shell:

```
app register --type source --name http --uri file://C:/scdf-tutorial/files/http-
source-rabbit-1.3.1.RELEASE.jar
app register --type sink --name log --uri file://C:/scdf-tutorial/files/log-sink-
rabbit-1.3.1.RELEASE.jar
stream create --name httpIngest --definition "http --server.port=8123 --path
-pattern=/data | log --level=WARN"
stream deploy --name httpIngest
stream list
```

Step 4 - Monitor the stream log

I noticed the following log output on `spring-cloud-dataflow-server-local` window:

```
2018-05... INFO 7432 --- [nio-9393-exec-5] o.s.c.d.spi.local.LocalAppDeployer       :
Command to be executed: C:\Portable\JDK\jre\bin\java.exe -jar C:\scdf-
tutorial\files\log-sink-rabbit-1.3.1.RELEASE.jar
2018-05... INFO 7432 --- [nio-9393-exec-5] o.s.c.d.spi.local.LocalAppDeployer       :
Deploying app with deploymentId httpIngest.log instance 0.
   Logs will be in C:\Users\user1\AppData\Local\Temp\spring-cloud-deployer-
7284336212701312273\httpIngest-1526842618079\httpIngest.log
```

So, I opened Git Bash to run the `tail` command in background and monitor the log output with the following command:

```
tail -f "C:\Users\user1\AppData\Local\Temp\spring-cloud-deployer-
7284336212701312273\httpIngest-1526842618079\httpIngest.log\stdout_0.log" &
```

I repeat the `curl` command that I did last lab:

```
curl -s -H "Content-Type: application/json" -X POST -d
'{"id":"1","temperature":"100"}' http://localhost:8123/data
```

I noticed an output like this:

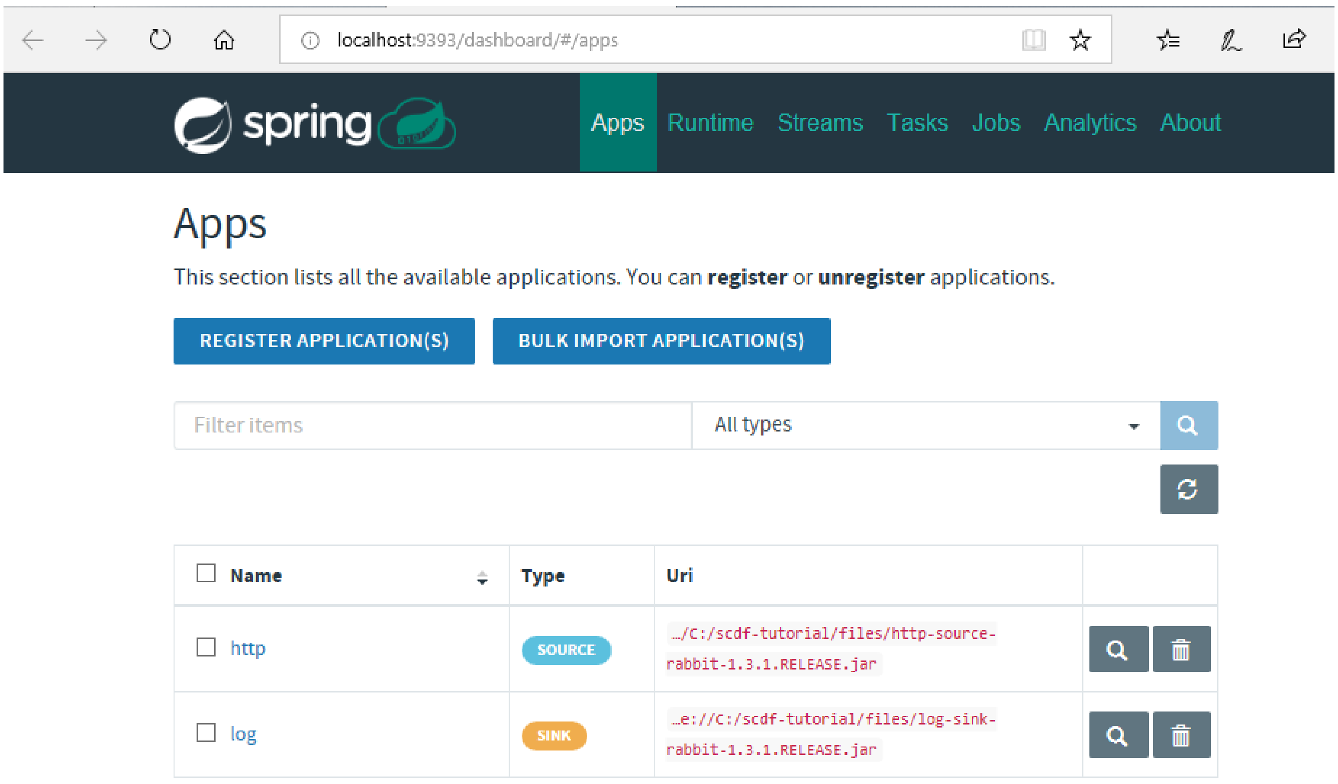
```
2018-05-... WARN 4222 --- [tp.httpIngest-1] log-sink                                 :
{"id":"1","temperature":"100"}
```

Step 5 - Accessing the Graphical UI Dashboard

After many steps did by using a command line, I wanted to see some graphical user interface. I pointed my brower to the dashboard: http://localhost:9393/dashboard.

The Microsoft Edge browser had some CSS problems when I opened the dashboard with it. So, I switched to Firefox.

By opening the dashboard I saw the tab `Apps` where I all the apps were registered. This tab provides the ability to register or unregister applications.



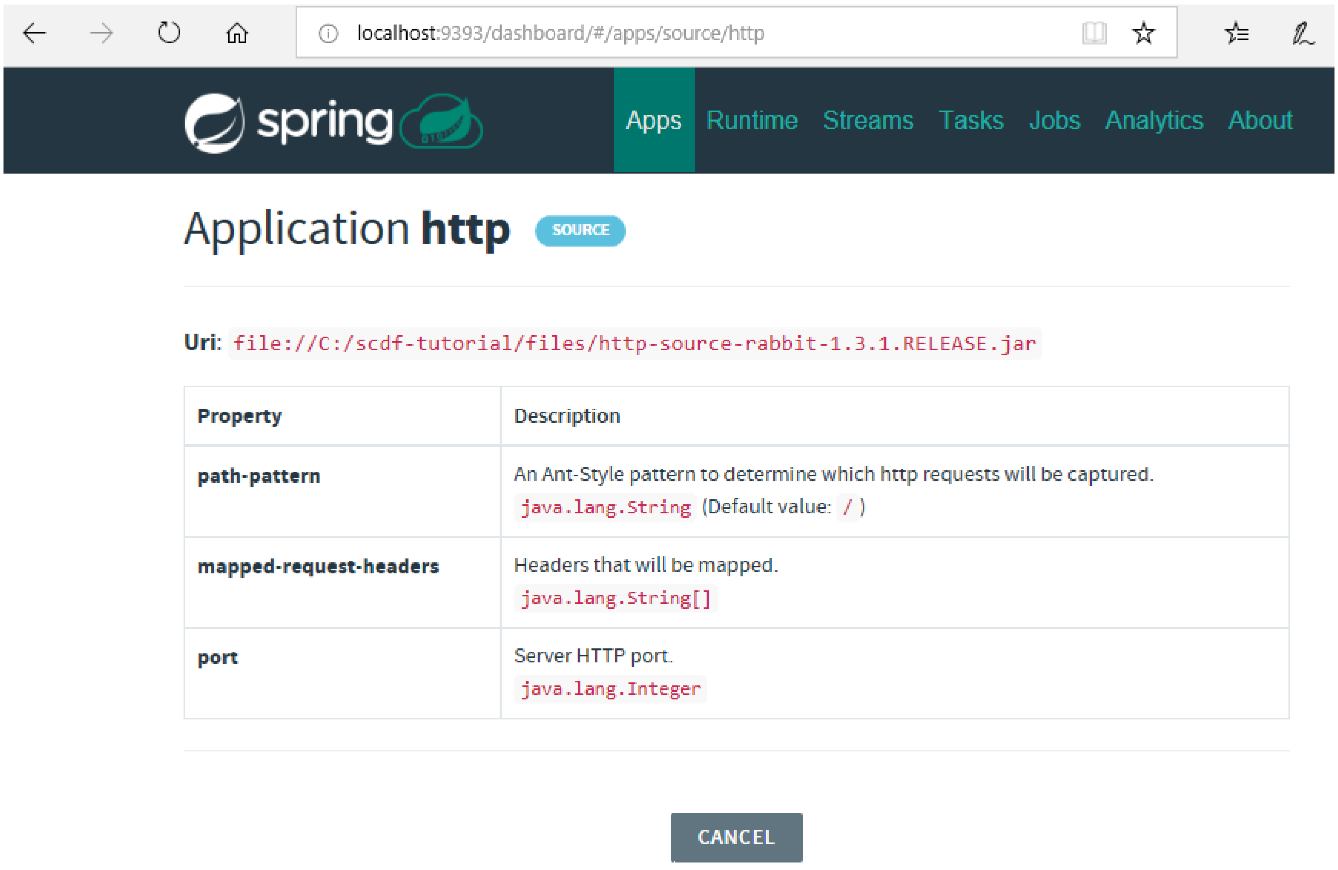
Apps

This section lists all the available applications. You can **register** or **unregister** applications.

Name	Type	Uri
http	SOURCE	…/C:/scdf-tutorial/files/http-source-rabbit-1.3.1.RELEASE.jar
log	SINK	…e://C:/scdf-tutorial/files/log-sink-rabbit-1.3.1.RELEASE.jar

Step 6 - Inspecting the tabs

By clicking on a application (like `http`) I could see its properties:

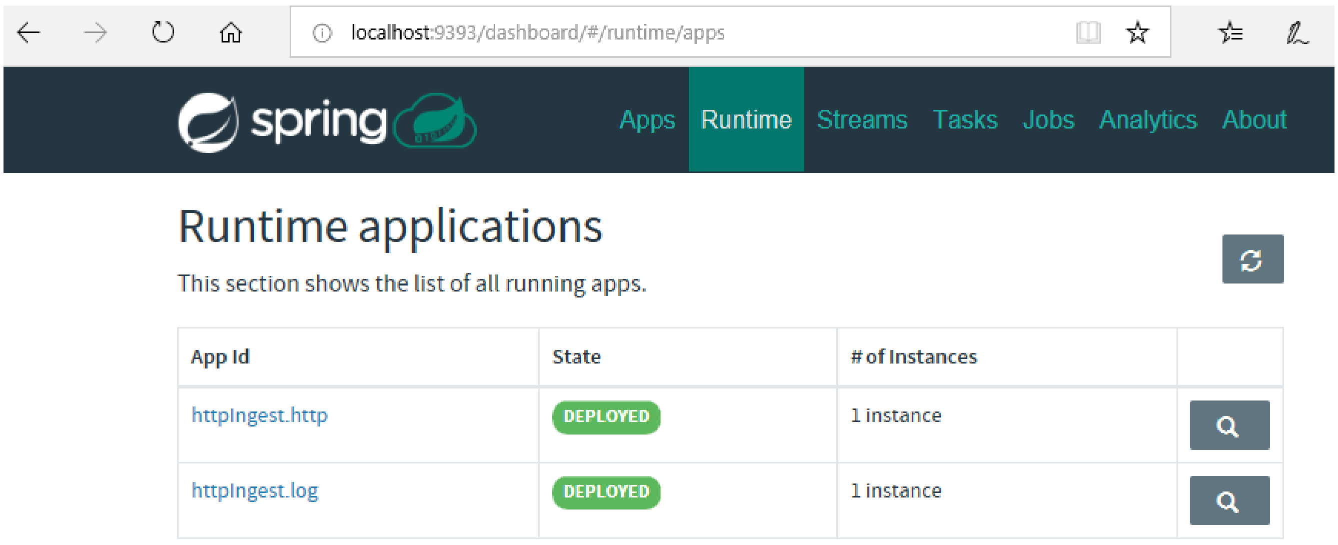


Application **http** SOURCE

Uri: `file://C:/scdf-tutorial/files/http-source-rabbit-1.3.1.RELEASE.jar`

Property	Description
path-pattern	An Ant-Style pattern to determine which http requests will be captured. `java.lang.String` (Default value: `/`)
mapped-request-headers	Headers that will be mapped. `java.lang.String[]`
port	Server HTTP port. `java.lang.Integer`

The `Runtime` tab shows the list of all running apps:

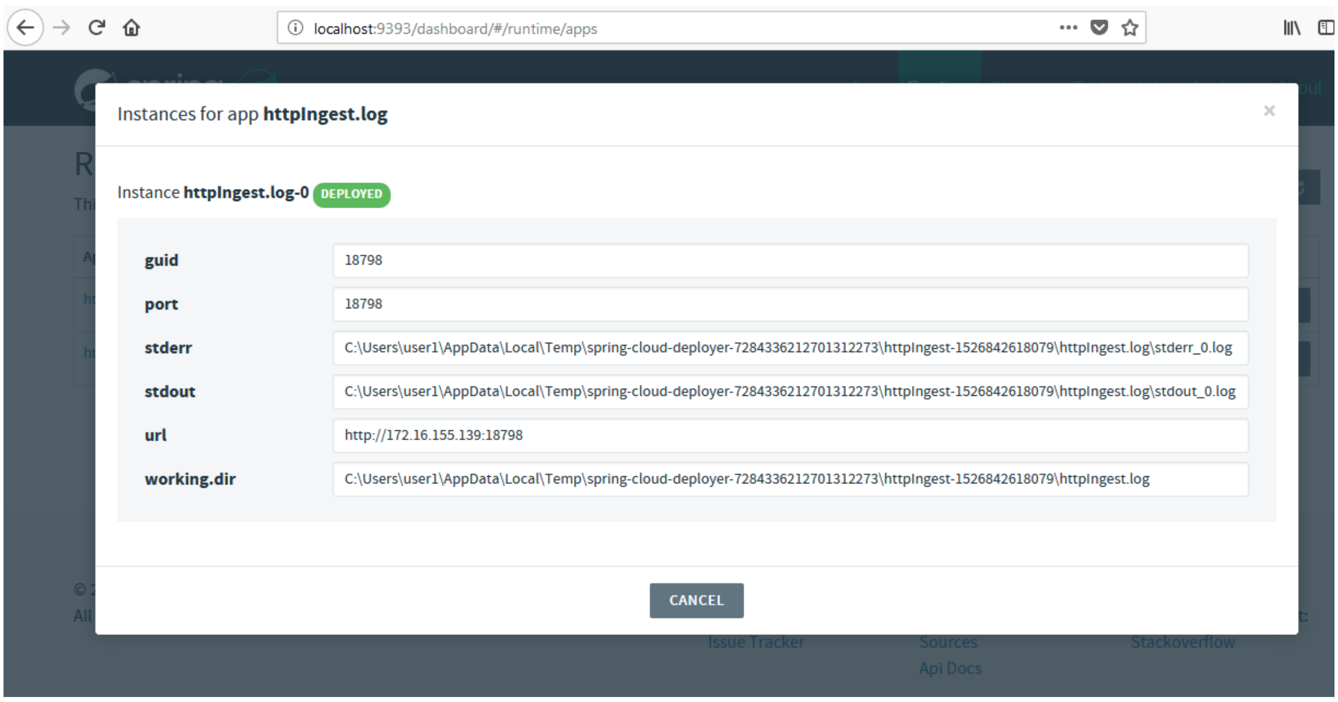


Runtime applications

This section shows the list of all running apps.

App Id	State	# of Instances	
httpIngest.http	DEPLOYED	1 instance	
httpIngest.log	DEPLOYED	1 instance	

By clicking on the app `httpIngest.log` I also saw the `stdout` path.

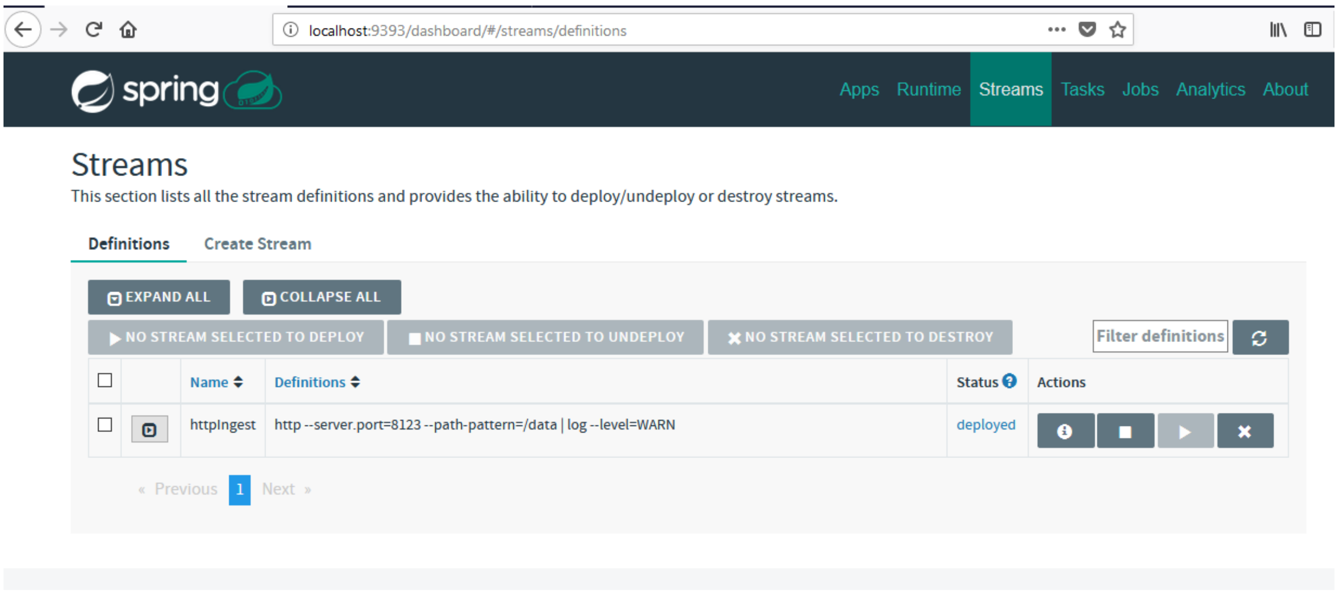

localhost:9393/dashboard/#/runtime/apps

Instances for app **httpIngest.log**

Instance **httpIngest.log-0** DEPLOYED

guid	18798
port	18798
stderr	C:\Users\user1\AppData\Local\Temp\spring-cloud-deployer-7284336212701312273\httpIngest-1526842618079\httpIngest.log\stderr_0.log
stdout	C:\Users\user1\AppData\Local\Temp\spring-cloud-deployer-7284336212701312273\httpIngest-1526842618079\httpIngest.log\stdout_0.log
url	http://172.16.155.139:18798
working.dir	C:\Users\user1\AppData\Local\Temp\spring-cloud-deployer-7284336212701312273\httpIngest-1526842618079\httpIngest.log

The `Streams` tab list all the stream definitions and provides the ability to deploy/deploy or destroy streams:


localhost:9393/dashboard/#/streams/definitions

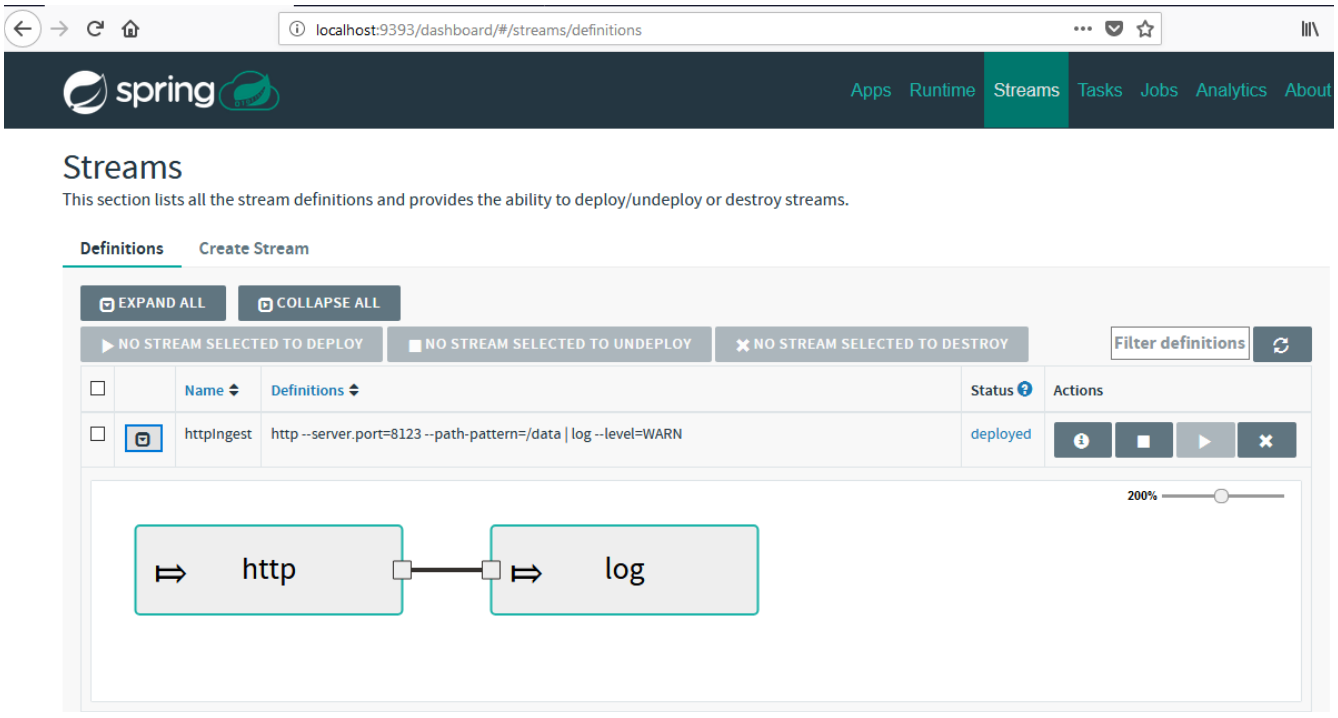
Streams

This section lists all the stream definitions and provides the ability to deploy/undeploy or destroy streams.

Definitions
Create Stream
EXPAND ALL
COLLAPSE ALL
NO STREAM SELECTED TO DEPLOY
NO STREAM SELECTED TO UNDEPLOY
NO STREAM SELECTED TO DESTROY
Filter definitions

		Name	Definitions	Status	Actions
☐		httpIngest	http --server.port=8123 --path-pattern=/data | log --level=WARN	deployed	

I could expand the stream definition:


localhost:9393/dashboard/#/streams/definitions

Streams

This section lists all the stream definitions and provides the ability to deploy/undeploy or destroy streams.

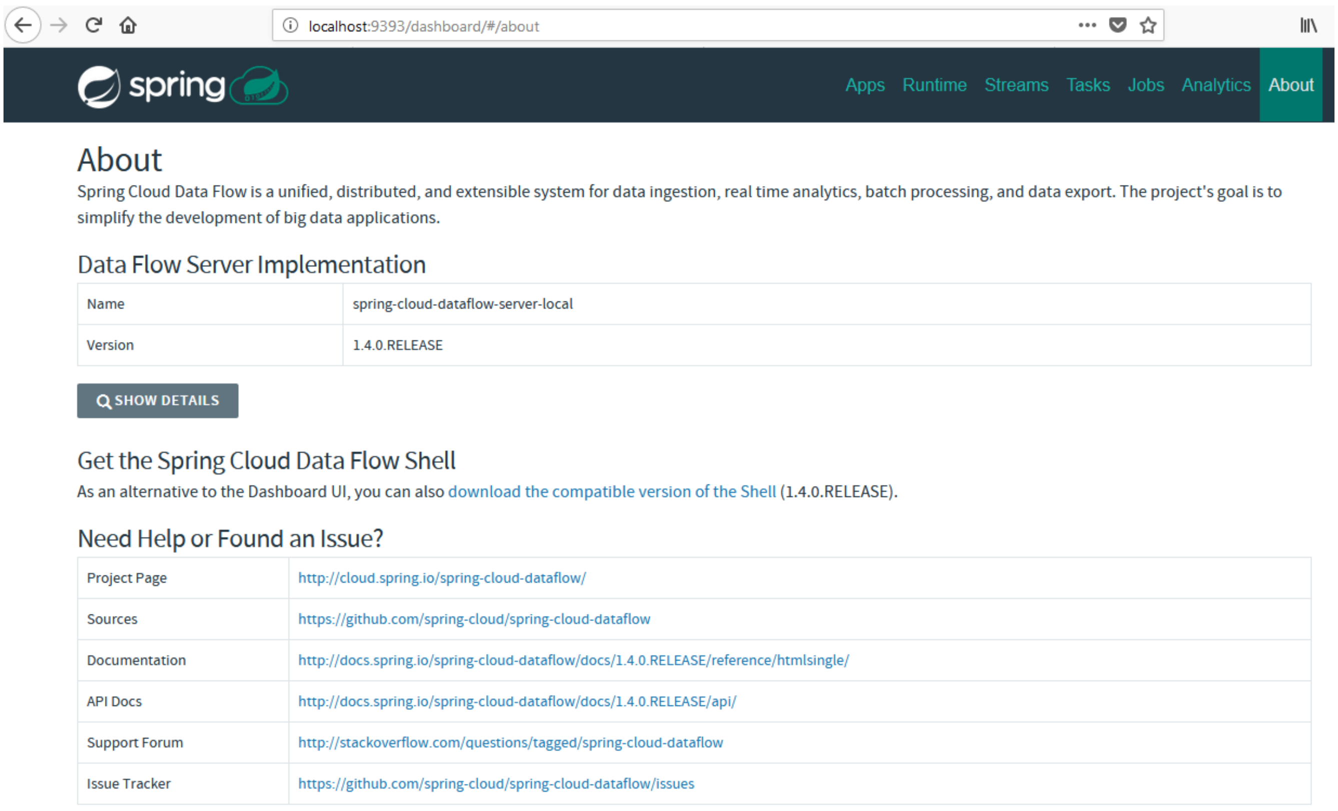
Definitions
Create Stream
EXPAND ALL
COLLAPSE ALL
NO STREAM SELECTED TO DEPLOY
NO STREAM SELECTED TO UNDEPLOY
NO STREAM SELECTED TO DESTROY
Filter definitions

		Name	Definitions	Status	Actions
☐		httpIngest	http --server.port=8123 --path-pattern=/data | log --level=WARN	deployed	

The Tasks tab will be discussed on a next lab.

The Jobs tab will be discussed on a next lab.

The most left tab (About) is where I could get information about SCDF:


localhost:9393/dashboard/#/about

About

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

Data Flow Server Implementation

Name	spring-cloud-dataflow-server-local
Version	1.4.0.RELEASE

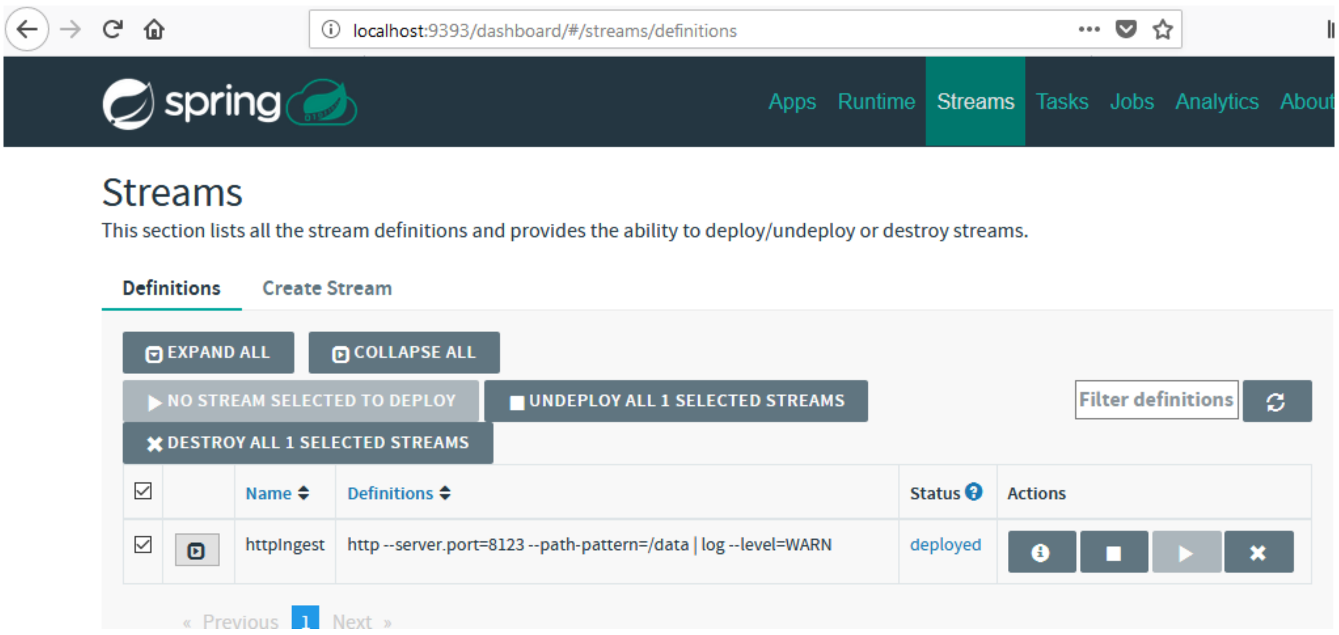
Get the Spring Cloud Data Flow Shell

As an alternative to the Dashboard UI, you can also download the compatible version of the Shell (1.4.0.RELEASE).

Need Help or Found an Issue?

Project Page	http://cloud.spring.io/spring-cloud-dataflow/
Sources	https://github.com/spring-cloud/spring-cloud-dataflow
Documentation	http://docs.spring.io/spring-cloud-dataflow/docs/1.4.0.RELEASE/reference/htmlsingle/
API Docs	http://docs.spring.io/spring-cloud-dataflow/docs/1.4.0.RELEASE/api/
Support Forum	http://stackoverflow.com/questions/tagged/spring-cloud-dataflow
Issue Tracker	https://github.com/spring-cloud/spring-cloud-dataflow/issues

I destroyed the created stream (httpIngest) using the UI:


localhost:9393/dashboard/#/streams/definitions
spring
Apps
Runtime
Streams
Tasks
Jobs
Analytics
About

Streams

This section lists all the stream definitions and provides the ability to deploy/undeploy or destroy streams.

Definitions
Create Stream
EXPAND ALL
COLLAPSE ALL
NO STREAM SELECTED TO DEPLOY
UNDEPLOY ALL 1 SELECTED STREAMS
DESTROY ALL 1 SELECTED STREAMS
Filter definitions

☑		Name	Definitions	Status	Actions
☑		httpIngest	http --server.port=8123 --path-pattern=/data | log --level=WARN	deployed	

Step 7 - Backing to the shell

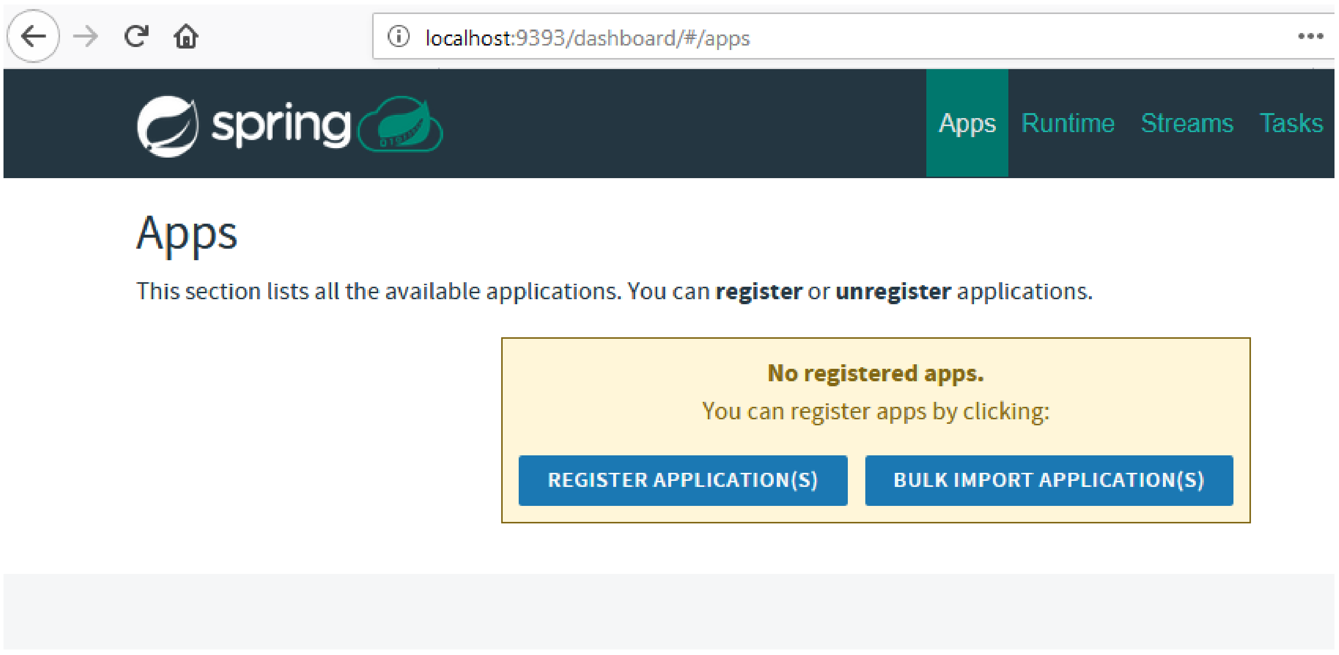
I checked if the stream was really destroyed by using the shell with the following command:

```
stream list
```

I typed the followed commands to unregister the applications:

```
app list
app unregister --type source --name http
app unregister --type sink --name log
```

Backing to the UI, I could check that I had no registered apps anymore:



Apps

This section lists all the available applications. You can **register** or **unregister** applications.

No registered apps.

You can register apps by clicking:

I left the windows that I opened (`spring-cloud-dataflow-server-local.bat` and `spring-cloud-dataflow-shell.bat`) running in order to continue the next lab.

Lab 7: Importing existing sources, sinks, processors and tasks to SCDF

Prerequisites

Windows `spring-cloud-dataflow-server-local` and `spring-cloud-dataflow-shell` started and in the last status of the previous lab.

Steps

Inside `spring-cloud-dataflow-shell.bat` window, I typed the following command:

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

The output of this command was the showed bellow. I saw that a lot of applications (65) were registered:

```
Successfully registered 65 applications from [source.sftp, source.mqtt.metadata,
sink.mqtt.metadata, source.file.metadata, processor.tcp-client, source.s3.metadata,
source.jms, source.ftp, processor.transform.metadata, source.time, sink.mqtt,
sink.s3.metadata, processor.scriptable-transform, sink.log, source.load-generator,
processor.transform, source.syslog, sink.websocket.metadata, sink.task-launcher-
local.metadata, source.loggregator.metadata, source.s3, source.load-
generator.metadata, processor.pmml.metadata, source.loggregator, source.tcp.metadata,
processor.httpclient.metadata, sink.file.metadata, source.triggertask,
source.twitterstream, source.gemfire-cq.metadata, processor.aggregator.metadata,
source.mongodb, source.time.metadata, source.gemfire-cq, sink.counter.metadata,
source.http, sink.tcp.metadata, sink.pgcopy.metadata, source.rabbit, sink.task-
launcher-yarn, source.jms.metadata, sink.gemfire.metadata, sink.cassandra.metadata,
processor.tcp-client.metadata, processor.header-enricher, sink.throughput, sink.task-
launcher-local, processor.python-http, sink.aggregate-counter.metadata, sink.mongodb,
processor.twitter-sentiment, sink.log.metadata, processor.splitter, sink.hdfs-dataset,
source.tcp, processor.python-jython.metadata, source.trigger, source.mongodb.metadata,
processor.bridge, source.http.metadata, source.rabbit.metadata, sink.ftp, sink.jdbc,
source.jdbc.metadata, source.mqtt, processor.pmml, sink.aggregate-counter,
sink.rabbit.metadata, processor.python-jython, sink.router.metadata, sink.cassandra,
processor.filter.metadata, source.tcp-client.metadata, processor.header-
enricher.metadata, processor.groovy-transform, source.ftp.metadata, sink.router,
sink.redis-pubsub, source.tcp-client, processor.httpclient, sink.file, sink.websocket,
source.syslog.metadata, sink.s3, sink.counter, sink.rabbit, processor.filter,
source.trigger.metadata, source.mail.metadata, sink.gpfdist.metadata, sink.pgcopy,
processor.python-http.metadata, sink.jdbc.metadata, sink.gpfdist, sink.ftp.metadata,
processor.splitter.metadata, sink.sftp, sink.field-value-counter, processor.groovy-
filter.metadata, processor.twitter-sentiment.metadata, source.triggertask.metadata,
sink.hdfs, processor.groovy-filter, sink.redis-pubsub.metadata, source.sftp.metadata,
processor.bridge.metadata, sink.field-value-counter.metadata, processor.groovy-
transform.metadata, processor.aggregator, sink.sftp.metadata,
processor.tensorflow.metadata, sink.throughput.metadata, sink.hdfs-dataset.metadata,
sink.tcp, source.mail, sink.task-launcher-cloudfoundry.metadata,
source.gemfire.metadata, processor.tensorflow, source.jdbc, sink.task-launcher-
yarn.metadata, sink.gemfire, source.gemfire, source.twitterstream.metadata,
sink.hdfs.metadata, processor.tasklaunchrequest-transform, sink.task-launcher-
cloudfoundry, source.file, sink.mongodb.metadata, processor.tasklaunchrequest-
transform.metadata, processor.scriptable-transform.metadata]
```

All these aplications could also be viewed in the dashboard (http://localhost:9393/dashboard).

Notes/conclusions

After the registration, the applications were stored in my local maven repository (the `~/.m2` directory).

I could check this information by typing the following command:

```
$ find ~/.m2/repository/org/springframework/cloud/stream -type f -name
'*.jar' | wc -l
      65
```

I could customize the applications that I want to install by creating a properties file.

For example, I could type the following commands to save and edit a `apps.properties` file only with the contents I would like to have:

```
curl -L http://bit.ly/Celsius-SR1-stream-applications-rabbit-maven > apps.properties
vim apps.properties
```

By doing this I could specify which sources, sinks, processors, and tasks, that I want to deploy.

I could also customize the maven repository used to download these applications. To do this I only need to specify a property like in following command:

```
java -jar spring-cloud-dataflow-server-local-1.4.0.RELEASE.jar --maven.remote
-repositories.repo1.url=http://nexus.example.com/repository/maven-central/
```

All references

Spring

GitHub

- spring-cloud-dataflow
- spring-cloud-dataflow-server-local
- spring-cloud-dataflow-docs
- spring-cloud-dataflow-samples
- spring-cloud-dataflow-shell

Documentation

- Quick Start page
- Spring Cloud Data Flow Reference Guide
- Spring Cloud Data Flow Samples

Pivotal

Blog

- Building Flexible Data Pipelines with Spring Cloud Data Flow for PCF

Documentation

- Spring Cloud® Data Flow for PCF

YouTube videos

SpringDeveloper channel

- Webinar Data Microservices with Spring Cloud Data Flow
- Orchestrating Data Microservices with Spring Cloud Data Flow - Mark Pollack
- Spring Tips: Spring Cloud Data Flow
- Demo: Spring Cloud Data Flow Shell Tips and Tricks
- 8 min demo: Composed Tasks in Spring Cloud Data Flow 1.2
- Spring Flo for Spring Cloud Data Flow1.0.0M3

Corby Page channel

- SCDF

Other articles/ examples

- oA: Getting Started with Spring Cloud Data Flow on PCF Dev
- Streaming with Spring Cloud Data Flow
- Getting Started with Stream Processing with Spring Cloud Data Flow
- Setting up a Spring Cloud Data Flow sandbox using Docker with Kubernetes
- Spring Cloud Data Flow – Making Custom Apps and Using Shell

- Building Data Pipelines With Spring Cloud Data Flow