Spring Cloud Stream Lab

Setting up a PAS environment - leveraging cloud services and app starters.

(Note: The lab is designed to show what it's like to manually connect apps that are sharing data in a pipeline. This is fully covered in the local mode where you deploy 2 out-of-the-box apps plus one custom processor using the spring-cloud-streams programming model. In a PAS environment this is much easier because the out-of-the-box apps are already registered in the platform and you need only register your custom app. You also need not worry about installing a message bus like RabbitMQ.)

To use the Spring Cloud Data Flow shell interface with Spring Cloud Data Flow for PCF service instances, install the following cf CLI plugins:

- Spring Cloud Data Flow for PCF cf CLI plugin. To install the plugin, run the following command:
- `$ cf install-plugin -r CF-Community "spring-cloud-dataflow-for-pcf"`

    - Java Runtime Environment (JRE). Required to run the Data Flow shell. You can download the JRE from the Java website.
- Service Instance Logging cf CLI plugin. To install the plugin, run the following command:
- `$ cf install-plugin -r CF-Community "Service Instance Logging"`
- Use the cf plugin to now access the dataflow shell.

```
cf dataflow-shell mydataflow
```

- `app list`

app	source	processor	sink	task
	sftp	tcp-client	mqtt	
	jms	scriptable-transform	log	
	ftp	transform	task-launcher-yarn	
	time	header-enricher	throughput	
	load-generator	python-http	task-launcher-local	
	syslog	twitter-sentiment	mongodb	
	s3	splitter	ftp	
	loggregator	image-recognition	jdbc	
	triggertask	bridge	aggregate-counter	
	twitterstream	pmml	router	
	mongodb	python-jython	redis-pubsub	
	gemfire-cq	groovy-transform	file	
	http	httpclient	websocket	
	rabbit	filter	s3	
	tcp	grpc	counter	
	trigger	groovy-filter	rabbit	
	mqtt	tensorflow	pgcopy	
	tcp-client	tasklaunchrequest-transform	sftp	
	mail	object-detection	field-value-counter	
	jdbc		hdfs	
	gemfire		tcp	
	file		gemfire	
			task-launcher-cloudfoundry	

The starter apps available from http://cloud.spring.io/spring-cloud-stream-app-starters/ have already been installed for you. (Note: in

a development environment you generally only install the apps that you're going to be using for your data pipelines).

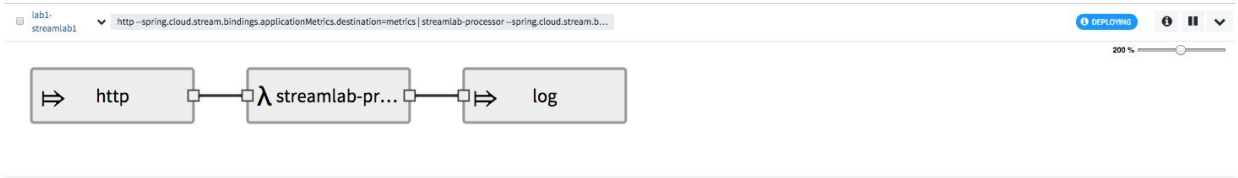
- Access the dashboard from:

```
https://dataflow-b9c6e5f8-fc14-4181-99f8-bd75bfa6ed4f.cfapps.haas-221.pez.pivotal.io/dashboard
```

Now walk through the local development section on "Creating your Processor". When you have it running locally as a spring boot app you are ready to create your first cloud native data pipeline. You do not push an app using cf push when using SCDF. You simply register your app and build a simple data pipeline.

```
app register --type processor --name streamlab-processor --uri https://github.com/wxlund/DNDataflow/raw/master/labs/jars/streamlab-0.0.1-SNAPSHOT.jar
```

Now you can build the app using the dataflow DSL either from the dataflow shell or the dashboard under create streams.



You can also view the logs to see the apps that you post your text message that the custom processor will convert to all upper case:

cf apps
Getting apps in org stubhub / space lab1 as wayne...
OK

```
name                                                requested state
instances   memory   disk   urls
```

```
cf logs aag8JT6-lab1-streamlab1-log-v1
Retrieving logs for app aag8JT6-lab1-streamlab1-log-v1 in org stubhub /
space lab1 as wayne...

   2019-03-06T16:01:15.97-0800 [APP/PROC/WEB/0] OUT 2019-03-07 00:01:15.971
INFO 16 --- [b1-streamlab1-1] aag8JT6-lab1-streamlab1-log-v1           :
THE QUICK BROWN FOX JUMPED OVER THE FENCE.
```

That completes this lab.

(Optional) Setting up a local environment - manual mode

1. Install RabbitMQ following the instructions in Lab0 - Install RabbitMQ. (Note: Wrong versions listed in the lab but instructions will work). Mv
2. Now let's setup the lab environment:
    a. Using git
        i. From a your local terminal or command prompt change directory to a clean working directory.
        ii. Now execute: git clone https://github.c/m/wxlund/DNDatafl/w
        iii. Now cd DNDataflow
    b. Using Thumb drive (Note: we haven't created a zip of the clone yet. Determine if lab will need this.
        i. Copy the DNDataflow directory from the thumb drive to a location on your laptop hard drive
        ii. Now from a terminal or command prompt cd to the DNDataflow directory you just created on your hard drive.

Creating your Processor

- CD into "DNDataflowolabs/lab1" folder
- First let's review the project in your favourite IDE or text-editor
- Start by opening the pom.xml and review the project dependencies
- You'll find the following dependency that brings the binder implementation we would like to use in this lab, which happens to be a RabbitMQ implementation in our case. *(You'll also find few other dependencies that are commented out[a] let's ignore them for now)*

```
<dependency>
        <groupId> org.springframework.cloud </groupId>
        <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

- open “ StreamlabApplication ” and add @EnableBinding(Processoroclass) annotation

```
@EnableBinding(Processor.class)
@SpringBootApplication
public class StreamlabApplication {

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

    @ServiceActivator(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public Object uppercase(Object incomingPayload) {
        String outgoingPayload = incomingPayload.toString().toUpperCase();
        System.out.println("Incoming payload=[" + incomingPayload + "]" + " Outgoing payload=[" + outgoingPayload + "]");
        return outgoingPayload;
    }
}
```

- Uncomment the uppercase () business logic and review the INPUT and OUTPUT channel configurations activated via @ServiceActivator

```
@EnableBinding(Processor.class)
@SpringBootApplication
public class StreamlabApplication {

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

    @ServiceActivator(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public Object uppercase(Object incomingPayload) {
        String outgoingPayload = incomingPayload.toString().toUpperCase();
        System.out.println("Incoming payload=[" + incomingPayload + "]" + " Outgoing payload=[" + outgoingPayload + "]");
        return outgoingPayload;
    }
}
```

- From the terminal or shell command-prompt, let’s build the application

```
mvnw clean package
```

- Run the application and verify it starts normally

```
java  -jar target/streamlab-0.0.1-SNAPSHOT.jar
```

- Uncomment Spring Boot’s Actuator dependency

```
<dependency>
        <groupId> org.springframework.boot</groupId>
        <artifactId> spring-boot-starter-actuator</artifactId>
</dependency>
```

- Rebuild and run the application to review the actuator endpoints; specifically, we would want to confirm whether the channel bindings are setup correctly

```
java  -jar target/streamlab-0-0-1-SNAPSHOT.jar
```

- Launch “http://<HOST>:<PORT>/ configprops ” endpoint and verify the channel bindings loaded from the application.properties

Example:

```
file http://localhost:9002/actuator/configprops
```

Now, let’s build a streaming pipeline

(1) Source

This app is a simple spring boot application and is available from the app starters. For convenience a copy of the jar has been moved into the labs/jars directory. The app-starters are found at https://repo.spring.io/libs-release/org/springframework/cloud/stream/app/spring-cloud-stream-app-descriptor/Einstein.RELEASE/spring-cloud-stream-app-descriptor-Einstein.RELEASE.rabbit-apps-maven-repo-url.properties in our configuration of using rabbit.

```
java -jar labs/jars/http-source-rabbit-2.0.3.RELEASE.jar
-spring.cloud.stream.bindings.output.destination=streamlabdest1
--server.port=9001
```

(2) Processor

This is the app that you built, which is why the jar is found from the labs /target directory

```
java -jar labs/lab1/target/streamlab-0.0.1-SNAPSHOT.jar
--spring.cloud.stream.bindings.input.destination=streamlabdest1
--spring.cloud.stream.bindings.output.destination=streamlabdest2
--server.port=9002
```

(3) Sink

This app is a simple spring boot application and is available from the app starters. See the note from (1) Source.

```
java -jar labs/jars/log-sink-rabbit-2.0.2.RELEASE.jar
--spring.cloud.stream.bindings.input.destination=streamlabdest2
--server.port=9003
```

- Monitor the logs of all the 3 applications in the terminal window where the app was launched.
- Post data to the http://localhost endpoint at port=9001

```
curl  -target http://localhost:9001 -H “Content-Type:text/plain”  -d “hello world”
```

● Review the the logs in the log-sink console (terminal)

Testing your Processor

Both unit and integration tests are included in the `/lab1` project

- Unit Test

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class StreamlabApplicationTests {

    @Test
    public void contextLoads() {
        Assert.assertTrue(true);
    }

    @Test
    public void testUppercase() {
        StreamlabApplication lab = new StreamlabApplication();

        Object payload = lab.uppercase("foo");

        assertTrue(payload != null);
        assertThat(payload.toString(), not(""));
        assertThat(payload.toString(), is("FOO"));
    }
}
```

  - Uncomment “ StreamlabApplicationTests ”
  - Change the expectations and let the test fail
  - Run the tests

    - Verify the assertions fail
    - Revert to original state
    - Run the tests
    - Verify the assertions work correctly
- Integration Test
    - Uncomment “spring-cloud-stream-test-support” from pom.xml

```
<!--<dependency>-->
        <!--<groupId>org.springframework.cloud</groupId>-->
        <!--<artifactId>spring-cloud-stream-test-support</artifactId>-->
<!--</dependency>-->
```

- Re-import/refresh maven dependencies at the project level
- Uncomment “StreamlabIntegrationTests”

```java
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class StreamlabIntegrationTests {

    @Autowired
    protected Processor channels;

    @Autowired
    protected MessageCollector collector;

    @Test
    public void contextLoads() {
        Assert.assertTrue(true);
    }

    /**
     * Checks whether the default properties load successfully.
     * <p>
     * Also, the test verifies whether the channel communication works, too.
     */
    public static class ChannelCommunicationIntegrationTests extends StreamlabIntegrationTests {

        @Test
        public void inAndOutTest() {
            channels.input().send(new GenericMessage<Object>(""));
            channels.input().send(new GenericMessage<>("foo"));
            channels.input().send(new GenericMessage<Object>("bar"));
            channels.input().send(new GenericMessage<Object>("foo meets bar"));
            channels.input().send(new GenericMessage<Object>("nothing but the best test"));
            assertThat(collector.forChannel(channels.output()), receivesPayloadThat(not("")));
            assertThat(collector.forChannel(channels.output()), receivesPayloadThat(is("FOO")));
            assertThat(collector.forChannel(channels.output()), receivesPayloadThat(is("BAR")));
            assertThat(collector.forChannel(channels.output()), receivesPayloadThat(is("FOO MEETS BAR")));
            assertThat(collector.forChannel(channels.output()), receivesPayloadThat(not("nothing but the best test")));
        }
    }
}
```

- Change the expectations and let the test fail
- Run the tests
- Verify the assertions fail
- Revert to original state
- Run the tests
- Verify the assertions work correctly

Appendix

For the curious, who would like to start from Spring Initializr experience, please select the dependencies as listed below and then generate a new project.