Spring Cloud Data Flow Reference Guide

1.0.0.M1

Copyright © 2013-2015Pivotal Software, Inc.

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

Table of Contents

Part I. Spring Cloud Data Flow Overview

This section provides a brief overview of the Spring Cloud Data Flow reference documentation. Think of it as map for the rest of the document. You can read this reference guide in a linear fashion, or you can skip sections if something doesn’t interest you.

1. About the documentation

The Spring Cloud Data Flow reference guide is available as html, pdf and epub documents. The latest copy is available at docs.spring.io/spring-cloud-dataflow/docs/current/reference.

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

2. Getting help

Having trouble with Spring Boot, We'd like to help!

- Try the How-to's — they provide solutions to the most common questions.
- Ask a question - we monitor stackoverflow.com for questions tagged with `spring-cloud`.
- Report bugs with Spring Boot at github.com/spring-cloud/spring-cloud-dataflow/issues.

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

3. Introducing Spring Cloud Data Flow

Spring Cloud Data Flow is a cloud native data framework that unifies stream and batch processing for data microservices, across the cloud or on-prem. It allows developers to create, orchestrate and refactor data pipelines with a single programming model for common use cases like data ingest, real time analytics, and data import/export.

Spring Cloud Data Flow defines the best practices for distributed stream and batch data processing.

Spring Cloud Data Flow is the cloud native redesign of Spring XD - a project that aimed to simplify Big Data application development. This redesign allows running stream and batch applications as data microservices and they can independently evolve in isolation.

3.1 Features

- Orchestrate applications across a variety of distributed runtime platforms including: Cloud Foundry, Lattice, and Apache YARN
- Separate runtime dependencies backed by ‘spring profiles’
- Consume stream and batch microservices as maven dependency and push it to production
- Develop using: DSL, Shell, REST-APIs, Admin-UI, and Flo
- Take advantage of metrics, health checks and remote management functionalities
- Scale stream and batch pipelines without interrupting data flows

4. Spring Cloud Data Flow Architecture

The architecture for Spring Cloud Data Flow is separated into a number of distinct components.

4.1 Components

The Core domain module includes the concept of a **stream** that is a composition of spring-cloud-stream modules in a linear pipeline from a **source** to a **sink**, optionally including **processor** modules in between. The domain also includes the concept of a **task**, which may be any process that does not run indefinitely, including Spring Batch jobs.

The Module Registry maintains the set of available modules, and their mappings to Maven coordinates.

The Module Deployer SPI provides the abstraction layer for deploying the modules of a given stream across a variety of runtime environments, including:

- Local
- Lattice
- Cloud Foundry
- Yarn

The Admin provides a REST API and UI. It is an executable Spring Boot application that is profile aware, so that the proper implementation of the Module Deployer SPI will be instantiated based on the environment within which the Admin application itself is running.

The Shell connects to the Admin's REST API and supports a DSL that simplifies the process of defining a stream and managing its lifecycle.

Part II. Getting started

If you’re just getting started with Spring Cloud Data Flow, this is the section for you! Here we answer the basic “what?”, “how?” and “why?” questions. You’ll find a gentle introduction to Spring Cloud Data Flow along with installation instructions. We’ll then build our first Spring Cloud Data Flow application, discussing some core principles as we go.

5. System Requirements

You need Java installed (Java 7 or better, we recommend Java 8) and to build you need to have Maven installed as well.

You also need to have Redis installed and running if you plan on running a local system.

6. Building Spring Cloud Data Flow

Clone the GitHub repository:

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

Switch to the project directory:

```
cd spring-cloud-dataflow
```

Build the project:

```
mvn clean install -s .settings.xml
```

7. Deploying Spring Cloud Data Flow

7.1 Deploying 'local'

1. start Redis locally via `redis-server`
2. download the Spring Cloud Data Flow Admin and Shell apps:

```
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-admin/1.0.0.M1/
spring-cloud-dataflow-admin-1.0.0.M1.jar
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-shell/1.0.0.M1/
spring-cloud-dataflow-shell-1.0.0.M1.jar
```

3. launch the admin:

```
$ java -jar spring-cloud-dataflow-admin-1.0.0.M1.jar
```

4. launch the shell:

```
$ java -jar spring-cloud-dataflow-shell-1.0.0.M1.jar
```

thus far, only the following commands are supported in the shell when running singlenode:

- `stream list`
- `stream create`
- `stream deploy`

7.2 Deploying on Lattice

1. start Redis on Lattice (running as root):

```
ltc create redis redis -r
```

2. launch the admin, with a mapping for port 9393 and extra memory (the default is 128MB):

```
ltc create admin springcloud/dataflow-admin -p 9393 -m 512
```

3. launching the shell is the same as above, but once running must be configured to point to the admin that is running on Lattice:

```
server-unknown:>admin config server http://admin.192.168.11.11.xip.io
Successfully targeted http://admin.192.168.11.11.xip.io
dataflow:>
```

all stream commands are supported in the shell when running on Lattice:

- `stream list`
- `stream create`
- `stream deploy`
- `stream undeploy`
- `stream all undeploy`

- `stream destroy`
- `stream all destroy`

7.3 Deploying on Cloud Foundry

Spring Cloud Data Flow can be used to deploy modules in a Cloud Foundry environment. When doing so, the Admin application can either run itself on Cloud Foundry, or on another installation (*e.g.* a simple laptop).

The required configuration amounts to the same, and is merely related to providing credentials to the Cloud Foundry instance, so that the admin can spawn applications itself. Any Spring Boot compatible configuration mechanism can be used (passing program arguments, editing configuration files before building the application, using Spring Cloud Config, using environment variables, *etc.*), although although some may prove more adequate than others when running *on* Cloud Foundry.

1. provision a redis service instance on Cloud Foundry. Your mileage may vary depending on your Cloud Foundry installation. Use `cf marketplace` to discover which plans are available to you. For example when using Pivotal Web Services:

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

2. download the Spring Cloud Data Flow Admin and Shell apps:

```
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-admin/1.0.0.M1/
spring-cloud-dataflow-admin-1.0.0.M1.jar
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-shell/1.0.0.M1/
spring-cloud-dataflow-shell-1.0.0.M1.jar
```

3a. push the admin application on Cloud Foundry, configure it (see below) and start it

> **Note**
>
> You must use a unique name for your app that's not already used by someone else or your deployment will fail

```
cf push s-c-dataflow-admin --no-start -p spring-cloud-dataflow-admin-1.0.0.M1.jar
cf bind-service s-c-dataflow-admin redis
```

Now we can configure the app. This configuration is for Pivotal Web Services. You need to fill in {org}, {space}, {email} and {password} before running these commands.

```
cf set-env s-c-dataflow-admin CLOUDFOUNDRY_API_ENDPOINT https://api.run.pivotal.io
cf set-env s-c-dataflow-admin CLOUDFOUNDRY_ORGANIZATION {org}
cf set-env s-c-dataflow-admin CLOUDFOUNDRY_SPACE {space}
cf set-env s-c-dataflow-admin CLOUDFOUNDRY_DOMAIN cfapps.io
cf set-env s-c-dataflow-admin CLOUDFOUNDRY_SERVICES redis
cf set-env s-c-dataflow-admin SECURITY_OAUTH2_CLIENT_USERNAME {email}
cf set-env s-c-dataflow-admin SECURITY_OAUTH2_CLIENT_PASSWORD {password}
cf set-env s-c-dataflow-admin SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI https://login.run.pivotal.io/
oauth/token
cf set-env s-c-dataflow-admin SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI https://
login.run.pivotal.io/oauth/authorize
```

We are now ready to start the app.

```
cf start s-c-dataflow-admin
```

alternatively,

3b. run the admin application locally, targeting your Cloud Foundry installation (see below for configuration)

```
java -jar spring-cloud-dataflow-admin-1.0.0.M1.jar [--option1=value1] [--option2=value2] [etc.]
```

4. run the shell and optionally target the Admin application if not running on the same host (will typically be the case if deployed on Cloud Foundry as **3a.**)

```
$ java -jar spring-cloud-dataflow-shell-1.0.0.M1.jar
```

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

At step **3.**, either running *on* Cloud Foundry or *targeting* Cloud Foundry, the following pieces of configuration must be provided, for example using `cf env s-c-dataflow-admin CLOUDFOUNDRY_DOMAIN mydomain.cfapps.io` (note the use of underscores) when running *in* Cloud Foundry

```
# Default values cited after the equal sign.
# Example values, typical for Pivotal Web Services, cited as a comment

# url of the CF API (used when using cf login -a for example), e.g. https://api.run.pivotal.io
# (for setting env var use CLOUDFOUNDRY_API_ENDPOINT)
cloudfoundry.apiEndpoint=

# name of the organization that owns the space above, e.g. youruser-org
# (for setting env var use CLOUDFOUNDRY_ORGANIZATION)
cloudfoundry.organization=

# name of the space into which modules will be deployed
# (for setting env var use CLOUDFOUNDRY_SPACE)
cloudfoundry.space=<same as admin when running on CF or 'development'>

# the root domain to use when mapping routes, e.g. cfapps.io
# (for setting env var use CLOUDFOUNDRY_DOMAIN)
cloudfoundry.domain=

# Comma separated set of service instance names to bind to the module.
# Amongst other things, this should include a service that will be used
# for Spring Cloud Stream binding
# (for setting env var use CLOUDFOUNDRY_SERVICES)
cloudfoundry.services=redis

# url used for obtaining an OAuth2 token, e.g. https://uaa.run.pivotal.io/oauth/token
# (for setting env var use SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI)
security.oauth2.client.access-token-uri=

# url used to grant user authorizations, e.g. https://login.run.pivotal.io/oauth/authorize
# (for setting env var use SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI)
security.oauth2.client.user-authorization-uri=

# username and password of the user to use to create apps (modules)
# (for setting env var use SECURITY_OAUTH2_CLIENT_USERNAME and SECURITY_OAUTH2_CLIENT_PASSWORD)
security.oauth2.client.username=
security.oauth2.client.password=
```

7.4 Deploying on YARN

Currently the YARN configuration is set to use `localhost`, meaning this can only be run against a local cluster. Also, all commands shown here need to be run from the project root.

1. download the Spring Cloud Data Flow YARN and Shell apps:

```
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-yarn-
appmaster/1.0.0.M1/spring-cloud-dataflow-yarn-appmaster-1.0.0.M1.jar
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-yarn-
container/1.0.0.M1/spring-cloud-dataflow-yarn-container-1.0.0.M1.jar
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-yarn-
client/1.0.0.M1/spring-cloud-dataflow-yarn-client-1.0.0.M1.jar
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-admin/1.0.0.M1/
spring-cloud-dataflow-admin-1.0.0.M1.jar
wget http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-shell/1.0.0.M1/
spring-cloud-dataflow-shell-1.0.0.M1.jar
```

2. start Redis locally via `redis-server`

3. optionally wipe existing data on `hdfs`

```
$ hdfs dfs -rm -R /app/app
```

4. start `spring-cloud-dataflow-admin` with `yarn` profile

```
$ java -Dspring.profiles.active=yarn -jar spring-cloud-dataflow-admin-1.0.0.M1.jar
```

5. start `spring-cloud-dataflow-shell`

```
$ java -jar spring-cloud-dataflow-shell-1.0.0.M1.jar

dataflow:>stream create --name "ticktock" --definition "time --fixedDelay=5|log" --deploy

dataflow:>stream list
  Stream Name  Stream Definition        Status
  -----------  ----------------------  --------
  ticktock     time --fixedDelay=5|log  deployed

dataflow:>stream destroy --name "ticktock"
Destroyed stream 'ticktock'
```

YARN application is pushed and started automatically during a stream deployment process. This application instance is not automatically closed which can be done from CLI:

```
$ java -jar spring-cloud-dataflow-yarn-client-1.0.0.M1.jar shell
Spring YARN Cli (v2.3.0.M2)
Hit TAB to complete. Type 'help' and hit RETURN for help, and 'exit' to quit.

$ submitted
  APPLICATION ID                  USER          NAME                              QUEUE    TYPE
 STARTTIME       FINISHTIME  STATE    FINALSTATUS  ORIGINAL TRACKING URL
  ------------------------------  ------------  --------------------------------  -------  --------
 --------------  ----------  -------  -----------  -------------------------
  application_1439803106751_0088  jvalkealahti  spring-cloud-dataflow-yarn-app_app  default  DATAFLOW
 01/09/15 09:02  N/A         RUNNING  UNDEFINED    http://192.168.122.1:48913

$ shutdown -a application_1439803106751_0088
shutdown requested
```

Properties `dataflow.yarn.app.appmaster.path` and `dataflow.yarn.app.container.path` can be used with both `spring-cloud-dataflow-admin` and and `spring-cloud-dataflow-yarn-client` to define directory for `appmaster` and `container` jars. Values for those default to `.` which then assumes all needed jars are in a same working directory.

Part III. Spring Cloud Stream Overview

This section goes into more detail about how you can work with Spring Cloud Stream. It covers topics such as creating and running stream modules.

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

8. Introducing Spring Cloud Stream

The Spring Cloud Stream project allows a user to develop and run messaging microservices using Spring Integration and run them locally, or in the cloud, or even on Spring XD. Just add `@EnableBinding` and run your app as a Spring Boot app (single application context). You just need to connect to the physical broker for the bindings, which is automatic if the relevant binder implementation is available on the classpath. The sample uses Redis.

Here's a sample source module (output channel only):

```
@SpringBootApplication
@ComponentScan(basePackageClasses=TimerSource.class)
public class ModuleApplication {

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

}

@Configuration
@EnableBinding(Source.class)
public class TimerSource {

  @Value("${format}")
  private String format;

  @Bean
  @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "${fixedDelay}",
 maxMessagesPerPoll = "1"))
  public MessageSource<String> timerMessageSource() {
    return () -> new GenericMessage<>(new SimpleDateFormat(format).format(new Date()));
  }

}
```

`@EnableBinding` is parameterized by an interface (in this case `Source`) which declares input and output channels. `Source`, `Sink` and `Processor` are provided off the shelf, but you can define others. Here's the definition of `Source`:

```
public interface Source {
  @Output("output")
  MessageChannel output();
}
```

The `@Output` annotation is used to identify output channels (messages leaving the module) and `@Input` is used to identify input channels (messages entering the module). It is optionally parameterized by a channel name - if the name is not provided the method name is used instead. An implementation of the interface is created for you and can be used in the application context by autowiring it, e.g. into a test case:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = ModuleApplication.class)
@WebAppConfiguration
@DirtiesContext
public class ModuleApplicationTests {

 @Autowired
 private Source source

 @Test
 public void contextLoads() {
  assertNotNull(this.source.output());
```

```
 }

}
```

Note

In this case there is only one `Source` in the application context so there is no need to qualify it when it is autowired. If there is ambiguity, e.g. if you are composing one module from some others, you can use `@Bindings` qualifier to inject a specific channel set. The `@Bindings` qualifier takes a parameter which is the class that carries the `@EnableBinding` annotation (in this case the `TimerSource`).

8.1 Multiple Input or Output Channels

A module can have multiple input or output channels all defined either as `@Input` and `@Output` methods in an interface (preferrable) or as bean definitions. Instead of just one channel named "input" or "output" you can add multiple `MessageChannel` methods annotated `@Input` or `@Output` and the names are converted to external channel names on the broker. The external channel names can be specified as properties that consist of the channel names prefixed with `spring.cloud.stream.bindings` (e.g. `spring.cloud.stream.bindings.input` or `spring.cloud.stream.bindings.output`). External channel names can have a channel type as a colon-separated prefix, and the semantics of the external bus channel changes accordingly. For example, you can have two `MessageChannels` called "output" and "foo" in a module with `spring.cloud.stream.bindings.output=bar` and `spring.cloud.stream.bindings.foo=topic:foo`, and the result is 2 external channels called "bar" and "topic:foo".

8.2 Samples

There are several samples, all running on the redis transport (so you need redis running locally to test them).

Note

The main set of samples are "vanilla" in the sense that they are not deployable as XD modules by the current generation (1.x) of XD. You can still interact with an XD system using the appropriate naming convention for input and output channel names (`<stream>.<index>` format).

- `source` is a Java config version of the classic "timer" module from Spring XD. It has a "fixedDelay" option (in milliseconds) for the period between emitting messages.
- `sink` is a Java config version of the classic "log" module from Spring XD. It has no options (but some could easily be added), and just logs incoming messages at INFO level.
- `transform` is a simple pass through logging transformer (just logs the incoming message and passes it on).
- `double` is a combination of 2 modules defined locally (a source and a sink, so the whole app is self contained).
- `extended` is a multi-module mashup of `source | transform | transform | sink`, where the modules are defined in the other samples and referred to in this app just as dependencies.

If you run the source and the sink and point them at the same redis instance (e.g. do nothing to get the one on localhost, or the one they are both bound to as a service on Cloud Foundry) then they will form a "stream" and start talking to each other. All the samples have friendly JMX and Actuator endpoints for inspecting what is going on in the system.

8.3 Module or App

Code using this library can be deployed as a standalone app or as an XD module. In standalone mode you app will run happily as a service or in any PaaS (Cloud Foundry, Lattice, Heroku, Azure, etc.). Depending on whether your main aim is to develop an XD module and you just want to test it locally using the standalone mode, or if the ultimate goal is a standalone app, there are some things that you might do differently.

Fat JAR

You can run in standalone mode from your IDE for testing. To run in production you can create an executable (or "fat") JAR using the standard Spring Boot tooling.

8.4 Making Standalone Modules Talk to Each Other

The `[input,output]ChannelName` are used to create physical endpoints in the external broker (e.g. `queue.<channelName>` in Redis).

For an XD module the channel names are `<group>.<index>` and a source (output only) has `index=0` (the default) and downstream modules have the same group but incremented index, with a sink module (input only) having the highest index. To listen to the output from a running XD module, just use the same "group" name and an index 1 larger than the app before it in the chain.

> Note: since the same naming conventions are used in XD, you can steal messages from or send messages to an existing XD stream by copying the stream name (to `spring.cloud.streams.group`) and knowing the index of the XD module you want to interact with.

Part IV. Using Spring Cloud Stream Modules

This section dives into the details of using the modules from Spring Cloud Stream Modules with Spring Cloud Data Flow.

9. Sources

9.1 FTP (`ftp`)

This source module supports transfer of files using the FTP protocol. Files are transferred from the `remote` directory to the `local` directory where the module is deployed. Messages emitted by the source are provided as a byte array by default. However, this can be customized using the `--mode` option:

- **ref** Provides a `java.io.File` reference
- **lines** Will split files line-by-line and emit a new message for each line
- **contents** The default. Provides the contents of a file as a byte array

When using `--mode=lines`, you can also provide the additional option `--withMarkers=true`. If set to `true`, the underlying `FileSplitter` will emit additional *start-of-file* and *end-of-file* marker messages before and after the actual data. The payload of these 2 additional marker messages is of type `FileSplitter.FileMarker`. The option `withMarkers` defaults to `false` if not explicitly set.

Options

The **ftp** source has the following options:

autoCreateLocalDir
: local directory must be auto created if it does not exist **(boolean, default: `true`)**

clientMode
: client mode to use : 2 for passive mode and 0 for active mode **(int, default: `0`)**

deleteRemoteFiles
: delete remote files after transfer **(boolean, default: `false`)**

filenamePattern
: simple filename pattern to apply to the filter **(String, default: *)**

fixedDelay
: the rate at which to poll the remote directory **(int, default: `1`)**

host
: the host name for the FTP server **(String, default: `localhost`)**

initialDelay
: an initial delay when using a fixed delay trigger, expressed in TimeUnits (seconds by default) **(int, default: `0`)**

localDir
: set the local directory the remote files are transferred to **(String, default: `/tmp/xd/ftp`)**

maxMessages
: the maximum messages per poll; -1 for unlimited **(long, default: `-1`)**

mode
: specifies how the file is being read. By default the content of a file is provided as byte array **(FileReadingMode, default: `contents`, possible values: `ref`,`lines`,`contents`)**

password
: the password for the FTP connection **(Password, no default)**

port
: the port for the FTP server **(int, default: `21`)**

preserveTimestamp
: whether to preserve the timestamp of files retrieved **(boolean, default: `true`)**

remoteDir
: the remote directory to transfer the files from **(String, default: `/`)**

remoteFileSeparator
: file separator to use on the remote side **(String, default: `/`)**

timeUnit
: the time unit for the fixed and initial delays **(String, default: `SECONDS`)**

tmpFileSuffix
: extension to use when downloading files **(String, default: `.tmp`)**

username
: the username for the FTP connection **(String, no default)**

withMarkers
: if true emits start of file/end of file marker messages before/after the data. Only valid with FileReadingMode 'lines' **(Boolean, no default)**

9.2 HTTP (`http`)

A source module that listens for HTTP requests and emits the body as a message payload. If the Content-Type matches 'text/*' or 'application/json', the payload will be a String, otherwise the payload will be a byte array.

To create a stream definition in the server using the XD shell

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

Post some data to the http server on the default port of 9000

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

See if the data ended up in the log.

9.3 Time (`time`)

The time source will simply emit a String with the current time every so often.

The **time** source has the following options:

fixedDelay
: time delay between messages, expressed in TimeUnits (seconds by default) **(int, default: `1`)**

format
: how to render the current time, using SimpleDateFormat **(String, default: `yyyy-MM-dd HH:mm:ss`)**

initialDelay
: an initial delay when using a fixed delay trigger, expressed in TimeUnits (seconds by default) **(int, default: `0`)**

timeUnit
: the time unit for the fixed and initial delays **(String, default: `SECONDS`)**

9.4 Twitter Stream (`twitterstream`)

This source ingests data from Twitter's streaming API v1.1. It uses the sample and filter stream endpoints rather than the full "firehose" which needs special access. The endpoint used will depend on the parameters you supply in the stream definition (some are specific to the filter endpoint).

You need to supply all keys and secrets (both consumer and accessToken) to authenticate for this source, so it is easiest if you just add these as the following environment variables: CONSUMER_KEY, CONSUMER_SECRET, ACCESS_TOKEN and ACCESS_TOKEN_SECRET.

Stream creation is then straightforward:

```
dataflow:> stream create --name tweets --definition "twitterstream | log" --deploy
```

The **twitterstream** source has the following options:

accessToken
: a valid OAuth access token **(String, no default)**

accessTokenSecret
: an OAuth secret corresponding to the access token **(String, no default)**

consumerKey
: a consumer key issued by twitter **(String, no default)**

consumerSecret
: consumer secret corresponding to the consumer key **(String, no default)**

language
: language code e.g. 'en' **(String, default: ``)**

> **Note**
>
> `twittersearch` emit JSON in the native Twitter format.

10. Processors

10.1 Filter (`filter`)

Use the filter module in a stream to determine whether a Message should be passed to the output channel.

The **filter** processor has the following options:

expression
: a SpEL expression used to transform messages **(String, default: `payload.toString()`)**

Filter with SpEL expression

The simplest way to use the filter processor is to pass a SpEL expression when creating the stream. The expression should evaluate the message and return true or false. For example:

```
dataflow:> stream create --name filtertest --definition "http | filter --expression=payload=='good' |
 log" --deploy
```

This filter will only pass Messages to the log sink if the payload is the word "good". Try sending "good" to the HTTP endpoint and you should see it in the XD log:

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

Alternatively, if you send the word "bad" (or anything else), you shouldn't see the log entry.

10.2 `groovy-filter`

A Processor module that retains or discards messages according to a predicate, expressed as a Groovy script.

The **groovy-filter** processor has the following options:

script
: The script resource location **(String, default: ``)**

variables
: Variable bindings as a comma delimited string of name-value pairs, e.g. 'foo=bar,baz=car' **(String, default: ``)**

variablesLocation
: The location of a properties file containing custom script variable bindings **(String, default: ``)**

10.3 `groovy-transform`

A Processor module that transforms messages using a Groovy script.

The **groovy-filter** processor has the following options:

script
: The script resource location **(String, default: ``)**

variables
: Variable bindings as a comma delimited string of name-value pairs, e.g. 'foo=bar,baz=car' **(String, default: ``)**

variablesLocation
: The location of a properties file containing custom script variable bindings **(String, default: ``)**

10.4 Transform (`transform`)

Use the transform module in a stream to convert a Message's content or structure.

The **transform** processor has the following options:

expression
: a SpEL expression used to transform messages **(String, default: `payload.toString()`)**

Transform with SpEL expression

The simplest way to use the transform processor is to pass a SpEL expression when creating the stream. The expression should return the modified message or payload. For example:

```
dataflow:> stream create --name transformtest --definition "http --port=9003 | transform --
expression=payload.toUpperCase() | log" --deploy
```

This transform will convert all message payloads to upper case. If sending the word "foo" to the HTTP endpoint and you should see "FOO" in the XD log:

```
dataflow:> http post --target http://localhost:9003 --data "foo"
```

As part of the SpEL expression you can make use of the pre-registered JSON Path function. The syntax is #jsonPath(payload,'<json path expression>')

11. Sinks

11.1 Counter (`counter`)

A simple module that counts messages received, using Spring Boot metrics abstraction.

The **counter** sink has the following options:

name
: The name of the counter to increment. **(String, default: `counts`)**

nameExpression
: A SpEL expression (against the incoming Message) to derive the name of the counter to increment. **(String, default: ``)**

store
: The name of a store used to store the counter. **(String, default: `memory`, possible values: `memory`, `redis`)**

11.2 Hadoop (HDFS) (`hdfs`)

If you do not have Hadoop installed, you can install Hadoop as described in our separate guide. Spring XD supports 4 Hadoop distributions, see using Hadoop for more information on how to start Spring XD to target a specific distribution.

Once Hadoop is up and running, you can then use the `hdfs` sink when creating a stream

```
dataflow:> stream create --name myhdfsstream1 --definition "time | hdfs" --deploy
```

In the above example, we've scheduled `time` source to automatically send ticks to `hdfs` once in every second. If you wait a little while for data to accumuluate you can then list can then list the files in the hadoop filesystem using the shell's built in hadoop fs commands. Before making any access to HDFS in the shell you first need to configure the shell to point to your name node. This is done using the `hadoop config` command.

```
dataflow:>hadoop config fs --namenode hdfs://localhost:8020
```

In this example the hdfs protocol is used but you may also use the webhdfs protocol. Listing the contents in the output directory (named by default after the stream name) is done by issuing the following command.

```
dataflow:>hadoop fs ls /xd/myhdfsstream1
Found 1 items
-rw-r--r--   3 jvalkealahti supergroup          0 2013-12-18 18:10 /xd/myhdfsstream1/
myhdfsstream1-0.txt.tmp
```

While the file is being written to it will have the `tmp` suffix. When the data written exceeds the rollover size (default 1GB) it will be renamed to remove the `tmp` suffix. There are several options to control the in use file file naming options. These are `--inUsePrefix` and `--inUseSuffix` set the file name prefix and suffix respectfully.

When you destroy a stream

```
dataflow:>stream destroy --name myhdfsstream1
```

and list the stream directory again, in use file suffix doesn't exist anymore.

```
dataflow:>hadoop fs ls /xd/myhdfsstream1
Found 1 items
-rw-r--r--   3 jvalkealahti supergroup        380 2013-12-18 18:10 /xd/myhdfsstream1/myhdfsstream1-0.txt
```

To list the list the contents of a file directly from a shell execute the hadoop cat command.

```
dataflow:> hadoop fs cat /xd/myhdfsstream1/myhdfsstream1-0.txt
2013-12-18 18:10:07
2013-12-18 18:10:08
2013-12-18 18:10:09
...
```

In the above examples we didn't yet go through why the file was written in a specific directory and why it was named in this specific way. Default location of a file is defined as `/xd/<stream name>/<stream name>-<rolling part>.txt`. These can be changed using options `--directory` and `--fileName` respectively. Example is shown below.

```
dataflow:>stream create --name myhdfsstream2 --definition "time | hdfs --directory=/xd/tmp --
fileName=data" --deploy
dataflow:>stream destroy --name myhdfsstream2
dataflow:>hadoop fs ls /xd/tmp
Found 1 items
-rw-r--r--   3 jvalkealahti supergroup        120 2013-12-18 18:31 /xd/tmp/data-0.txt
```

It is also possible to control the size of a files written into HDFS. The `--rollover` option can be used to control when file currently being written is rolled over and a new file opened by providing the rollover size in bytes, kilobytes, megatypes, gigabytes, and terabytes.

```
dataflow:>stream create --name myhdfsstream3 --definition "time | hdfs --rollover=100" --deploy
dataflow:>stream destroy --name myhdfsstream3
dataflow:>hadoop fs ls /xd/myhdfsstream3
Found 3 items
-rw-r--r--   3 jvalkealahti supergroup        100 2013-12-18 18:41 /xd/myhdfsstream3/myhdfsstream3-0.txt
-rw-r--r--   3 jvalkealahti supergroup        100 2013-12-18 18:41 /xd/myhdfsstream3/myhdfsstream3-1.txt
-rw-r--r--   3 jvalkealahti supergroup        100 2013-12-18 18:41 /xd/myhdfsstream3/myhdfsstream3-2.txt
```

Shortcuts to specify sizes other than bytes are written as `--rollover=64M`, `--rollover=512G` or `--rollover=1T`.

The stream can also be compressed during the write operation. Example of this is shown below.

```
dataflow:>stream create --name myhdfsstream4 --definition "time | hdfs --codec=gzip" --deploy
dataflow:>stream destroy --name myhdfsstream4
dataflow:>hadoop fs ls /xd/myhdfsstream4
Found 1 items
-rw-r--r--   3 jvalkealahti supergroup         80 2013-12-18 18:48 /xd/myhdfsstream4/
myhdfsstream4-0.txt.gzip
```

From a native os shell we can use hadoop's fs commands and pipe data into gunzip.

```
# bin/hadoop fs -cat /xd/myhdfsstream4/myhdfsstream4-0.txt.gzip | gunzip
2013-12-18 18:48:10
2013-12-18 18:48:11
...
```

Often a stream of data may not have a high enough rate to roll over files frequently, leaving the file in an opened state. This prevents users from reading a consistent set of data when running mapreduce

jobs. While one can alleviate this problem by using a small rollover value, a better way is to use the `idleTimeout` option that will automatically close the file if there was no writes during the specified period of time. This feature is also useful in cases where burst of data is written into a stream and you'd like that data to become visible in HDFS.

> **Note**
>
> The `idleTimeout` value should not exceed the timeout values set on the Hadoop cluster. These are typically configured using the `dfs.socket.timeout` and/or `dfs.datanode.socket.write.timeout` properties in the `hdfs-site.xml` configuration file.

```
dataflow:> stream create --name myhdfsstream5 --definition "http --port=8000 | hdfs --rollover=20 --
idleTimeout=10000" --deploy
```

In the above example we changed a source to `http` order to control what we write into a `hdfs` sink. We defined a small rollover size and a timeout of 10 seconds. Now we can simply post data into this stream via source end point using a below command.

```
dataflow:> http post --target http://localhost:8000 --data "hello"
```

If we repeat the command very quickly and then wait for the timeout we should be able to see that some files are closed before rollover size was met and some were simply rolled because of a rollover size.

```
dataflow:>hadoop fs ls /xd/myhdfsstream5
Found 4 items
-rw-r--r--   3 jvalkealahti supergroup         12 2013-12-18 19:02 /xd/myhdfsstream5/myhdfsstream5-0.txt
-rw-r--r--   3 jvalkealahti supergroup         24 2013-12-18 19:03 /xd/myhdfsstream5/myhdfsstream5-1.txt
-rw-r--r--   3 jvalkealahti supergroup         24 2013-12-18 19:03 /xd/myhdfsstream5/myhdfsstream5-2.txt
-rw-r--r--   3 jvalkealahti supergroup         18 2013-12-18 19:03 /xd/myhdfsstream5/myhdfsstream5-3.txt
```

Files can be automatically partitioned using a `partitionPath` expression. If we create a stream with `idleTimeout` and `partitionPath` with simple format `yyyy/MM/dd/HH/mm` we should see writes ending into its own files within every minute boundary.

```
dataflow:>stream create --name myhdfsstream6 --definition "time|hdfs --idleTimeout=10000 --
partitionPath=dateFormat('yyyy/MM/dd/HH/mm')" --deploy
```

Let a stream run for a short period of time and list files.

```
dataflow:>hadoop fs ls --recursive true --dir /xd/myhdfsstream6
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 09:42 /xd/myhdfsstream6/2014
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 09:42 /xd/myhdfsstream6/2014/05
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 09:42 /xd/myhdfsstream6/2014/05/28
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 09:45 /xd/myhdfsstream6/2014/05/28/09
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 09:43 /xd/myhdfsstream6/2014/05/28/09/42
-rw-r--r--   3 jvalkealahti supergroup        140 2014-05-28 09:43 /xd/myhdfsstream6/2014/05/28/09/42/
myhdfsstream6-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 09:44 /xd/myhdfsstream6/2014/05/28/09/43
-rw-r--r--   3 jvalkealahti supergroup       1200 2014-05-28 09:44 /xd/myhdfsstream6/2014/05/28/09/43/
myhdfsstream6-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 09:45 /xd/myhdfsstream6/2014/05/28/09/44
-rw-r--r--   3 jvalkealahti supergroup       1200 2014-05-28 09:45 /xd/myhdfsstream6/2014/05/28/09/44/
myhdfsstream6-0.txt
```

Partitioning can also be based on defined lists. In a below example we simulate feeding data by using a `time` and a `transform` elements. Data passed to `hdfs` sink has a content `APP0:foobar`, `APP1:foobar`, `APP2:foobar` or `APP3:foobar`.

```
dataflow:>stream create --name myhdfsstream7 --definition "time | transform --expression=
\"'APP'+T(Math).round(T(Math).random()*3)+':foobar'\" | hdfs --idleTimeout=10000 --
partitionPath=path(dateFormat('yyyy/MM/dd/HH'),list(payload.split(':')[0],{{'0TO1','APP0','APP1'},
{'2TO3','APP2','APP3'}}))" --deploy
```

Let the stream run few seconds, destroy it and check what got written in those partitioned files.

```
dataflow:>stream destroy --name myhdfsstream7
Destroyed stream 'myhdfsstream7'
dataflow:>hadoop fs ls --recursive true --dir /xd
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:24 /xd/myhdfsstream7
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:24 /xd/myhdfsstream7/2014
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:24 /xd/myhdfsstream7/2014/05
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:24 /xd/myhdfsstream7/2014/05/28
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:24 /xd/myhdfsstream7/2014/05/28/19
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:24 /xd/
myhdfsstream7/2014/05/28/19/0TO1_list
-rw-r--r--   3 jvalkealahti supergroup        108 2014-05-28 19:24 /xd/
myhdfsstream7/2014/05/28/19/0TO1_list/myhdfsstream7-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:24 /xd/
myhdfsstream7/2014/05/28/19/2TO3_list
-rw-r--r--   3 jvalkealahti supergroup        180 2014-05-28 19:24 /xd/
myhdfsstream7/2014/05/28/19/2TO3_list/myhdfsstream7-0.txt
dataflow:>hadoop fs cat /xd/myhdfsstream7/2014/05/28/19/0TO1_list/myhdfsstream7-0.txt
APP1:foobar
APP1:foobar
APP0:foobar
APP0:foobar
APP1:foobar
```

Partitioning can also be based on defined ranges. In a below example we simulate feeding data by using a `time` and a `transform` elements. Data passed to `hdfs` sink has a content ranging from `APP0` to `APP15`. We simple parse the number part and use it to do a partition with ranges `{3,5,10}`.

```
dataflow:>stream create --name myhdfsstream8 --definition "time | transform --
expression=\"'APP'+T(Math).round(T(Math).random()*15)\" | hdfs --idleTimeout=10000 --
partitionPath=path(dateFormat('yyyy/MM/dd/HH'),range(T(Integer).parseInt(payload.substring(3)),
{3,5,10}))" --deploy
```

Let the stream run few seconds, destroy it and check what got written in those partitioned files.

```
dataflow:>stream destroy --name myhdfsstream8
Destroyed stream 'myhdfsstream8'
dataflow:>hadoop fs ls --recursive true --dir /xd
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/myhdfsstream8
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/myhdfsstream8/2014
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/myhdfsstream8/2014/05
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/myhdfsstream8/2014/05/28
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/myhdfsstream8/2014/05/28/19
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/10_range
-rw-r--r--   3 jvalkealahti supergroup         16 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/10_range/myhdfsstream8-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/3_range
-rw-r--r--   3 jvalkealahti supergroup         35 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/3_range/myhdfsstream8-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/5_range
-rw-r--r--   3 jvalkealahti supergroup          5 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/5_range/myhdfsstream8-0.txt
dataflow:>hadoop fs cat /xd/myhdfsstream8/2014/05/28/19/3_range/myhdfsstream8-0.txt
APP3
APP3
APP1
APP0
APP1
dataflow:>hadoop fs cat /xd/myhdfsstream8/2014/05/28/19/5_range/myhdfsstream8-0.txt
```

```
APP4
dataflow:>hadoop fs cat /xd/myhdfsstream8/2014/05/28/19/10_range/myhdfsstream8-0.txt
APP6
APP15
APP7
```

Partition using a `dateFormat` can be based on content itself. This is a good use case if old log files needs to be processed where partitioning should happen based on timestamp of a log entry. We create a fake log data with a simple date string ranging from `1970-01-10` to `1970-01-13`.

```
dataflow:>stream create --name myhdfsstream9 --definition "time | transform --expression=
\"'1970-01-'+1+T(Math).round(T(Math).random()*3)\" | hdfs --idleTimeout=10000 --
partitionPath=path(dateFormat('yyyy/MM/dd/HH',payload,'yyyy-MM-DD'))" --deploy
```

Let the stream run few seconds, destroy it and check what got written in those partitioned files. If you see the partition paths, those are based on year 1970, not present year.

```
dataflow:>stream destroy --name myhdfsstream9
Destroyed stream 'myhdfsstream9'
dataflow:>hadoop fs ls --recursive true --dir /xd
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:56 /xd/myhdfsstream9
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:56 /xd/myhdfsstream9/1970
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:56 /xd/myhdfsstream9/1970/01
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:56 /xd/myhdfsstream9/1970/01/10
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:57 /xd/myhdfsstream9/1970/01/10/00
-rw-r--r--   3 jvalkealahti supergroup         44 2014-05-28 19:57 /xd/myhdfsstream9/1970/01/10/00/
myhdfsstream9-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:56 /xd/myhdfsstream9/1970/01/11
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:57 /xd/myhdfsstream9/1970/01/11/00
-rw-r--r--   3 jvalkealahti supergroup         99 2014-05-28 19:57 /xd/myhdfsstream9/1970/01/11/00/
myhdfsstream9-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:56 /xd/myhdfsstream9/1970/01/12
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:57 /xd/myhdfsstream9/1970/01/12/00
-rw-r--r--   3 jvalkealahti supergroup         44 2014-05-28 19:57 /xd/myhdfsstream9/1970/01/12/00/
myhdfsstream9-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:56 /xd/myhdfsstream9/1970/01/13
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:57 /xd/myhdfsstream9/1970/01/13/00
-rw-r--r--   3 jvalkealahti supergroup         55 2014-05-28 19:57 /xd/myhdfsstream9/1970/01/13/00/
myhdfsstream9-0.txt
dataflow:>hadoop fs cat /xd/myhdfsstream9/1970/01/10/00/myhdfsstream9-0.txt
1970-01-10
1970-01-10
1970-01-10
1970-01-10
```

Options

The **hdfs** sink has the following options:

closeTimeout
: timeout in ms, regardless of activity, after which file will be automatically closed **(long, default: `0`)**

codec
: compression codec alias name (gzip, snappy, bzip2, lzo, or slzo) **(String, default: ``)**

directory
: where to output the files in the Hadoop FileSystem **(String, default: `/tmp/hdfs-sink`)**

fileExtension
: the base filename extension to use for the created files **(String, default: `txt`)**

fileName
: the base filename to use for the created files **(String, default: `<stream name>`)**

fileOpenAttempts
: maximum number of file open attempts to find a path **(int, default: `10`)**

fileUuid
: whether file name should contain uuid **(boolean, default: `false`)**

fsUri
: the URI to use to access the Hadoop FileSystem **(String, default: `${spring.hadoop.fsUri}`)**

idleTimeout
: inactivity timeout in ms after which file will be automatically closed **(long, default: `0`)**

inUsePrefix
: prefix for files currently being written **(String, default: ``)**

inUseSuffix
: suffix for files currently being written **(String, default: `.tmp`)**

overwrite
: whether writer is allowed to overwrite files in Hadoop FileSystem **(boolean, default: `false`)**

partitionPath
: a SpEL expression defining the partition path **(String, default: ``)**

rollover
: threshold in bytes when file will be automatically rolled over **(String, default: `1G`)**

> **Note**
>
> In the context of the `fileOpenAttempts` option, attempt is either one rollover request or failed stream open request for a path (if another writer came up with a same path and already opened it).

Partition Path Expression

SpEL expression is evaluated against a Spring Messaging `Message` passed internally into a HDFS writer. This allows expression to use `headers` and `payload` from that message. While you could do a custom processing within a stream and add custom headers, `timestamp` is always going to be there. Data to be written is then available in a `payload`.

Accessing Properties

Using a `payload` simply returns whatever is currently being written. Access to headers is via `headers` property. Any other property is automatically resolved from headers if found. For example `headers.timestamp` is equivalent to `timestamp`.

Custom Methods

Addition to a normal SpEL functionality, few custom methods has been added to make it easier to build partition paths. These custom methods can be used to work with a normal partition concepts like `date formatting`, `lists`, `ranges` and `hashes`.

path

```
path(String... paths)
```

Concatenates paths together with a delimiter `/`. This method can be used to make the expression less verbose than using a native SpEL functionality to combine path parts together. To create a path `part1/part2`, expression `'part1' + '/' + 'part2'` is equivalent to `path('part1','part2')`.
Parameters

paths
: Any number of path parts

Return Value. Concatenated value of paths delimited with `/`.

dateFormat

```
dateFormat(String pattern)
dateFormat(String pattern, Long epoch)
dateFormat(String pattern, Date date)
dateFormat(String pattern, String datestring)
dateFormat(String pattern, String datestring, String dateformat)
```

Creates a path using date formatting. Internally this method delegates into `SimpleDateFormat` and needs a `Date` and a `pattern`. On default if no parameter used for conversion is given, `timestamp` is expected. Effectively `dateFormat('yyyy')` equals to `dateFormat('yyyy', timestamp)` or `dateFormat('yyyy', headers.timestamp)`.

Method signature with three parameters can be used to create a custom `Date` object which is then passed to `SimpleDateFormat` conversion using a `dateformat` pattern. This is useful in use cases where partition should be based on a date or time string found from a payload content itself. Default `dateformat` pattern if omitted is `yyyy-MM-dd`.
Parameters

pattern
: Pattern compatible with `SimpleDateFormat` to produce a final output.

epoch
: Timestamp as `Long` which is converted into a `Date`.

date
: A `Date` to be formatted.

dateformat
: Secondary pattern to convert `datestring` into a `Date`.

datestring
: `Date` as a `String`

Return Value. A path part representation which can be a simple file or directory name or a directory structure.

list

```
list(Object source, List<List<Object>> lists)
```

Creates a partition path part by matching a `source` against a lists denoted by `lists`.

Lets assume that data is being written and it's possible to extrace an `appid` either from headers or payload. We can automatically do a list based partition by using a partition method `list(headers.appid,{{'1TO3','APP1','APP2','APP3'},`

`{'4TO6','APP4','APP5','APP6'}})`. This method would create three partitions, `1TO3_list`, `4TO6_list` and `list`. Latter is used if no match is found from partition lists passed to `lists`.
Parameters

source
: An `Object` to be matched against `lists`.

lists
: A definition of list of lists.

Return Value. A path part prefixed with a matched key i.e. `XXX_list` or `list` if no match.

range

```
range(Object source, List<Object> list)
```

Creates a partition path part by matching a `source` against a list denoted by `list` using a simple binary search.

The partition method takes a `source` as first argument and `list` as a second argument. Behind the scenes this is using jvm's `binarySearch` which works on an `Object` level so we can pass in anything. Remember that meaningful range match only works if passed in `Object` and types in list are of same type like `Integer`. Range is defined by a binarySearch itself so mostly it is to match against an upper bound except the last range in a list. Having a list of `{1000,3000,5000}` means that everything above 3000 will be matched with 5000. If that is an issue then simply adding `Integer.MAX_VALUE` as last range would overflow everything above 5000 into a new partition. Created partitions would then be `1000_range`, `3000_range` and `5000_range`.
Parameters

source
: An `Object` to be matched against `list`.

list
: A definition of list.

Return Value. A path part prefixed with a matched key i.e. `XXX_range`.

hash

```
hash(Object source, int bucketcount)
```

Creates a partition path part by calculating hashkey using `source`s `hashCode` and `bucketcount`. Using a partition method `hash(timestamp,2)` would then create partitions named `0_hash`, `1_hash` and `2_hash`. Number suffixed with `_hash` is simply calculated using `Object.hashCode() % bucketcount`.
Parameters

source
: An `Object` which `hashCode` will be used.

bucketcount
: A number of buckets

Return Value. A path part prefixed with a hash key i.e. `XXX_hash`.

11.3 Log (`log`)

Probably the simplest option for a sink is just to log the data. The `log` sink uses the application logger to output the data for inspection. The log level is set to `WARN` and the logger name is created from the stream name. To create a stream using a `log` sink you would use a command like

```
dataflow:> stream create --name mylogstream --definition "http --port=8000 | log" --deploy
```

You can then try adding some data. We've used the `http` source on port 8000 here, so run the following command to send a message

```
dataflow:> http post --target http://localhost:8000 --data "hello"
```

and you should see the following output in the XD container console.

```
13/06/07 16:12:18 INFO Received: hello
```

11.4 Redis (`redis`)

Redis sink can be used to ingest data into redis store. You can choose `queue`, `topic` or `key` with selcted collection type to point to a specific data store.

For example,

```
dataflow:>stream create store-into-redis --definition "http | redis --queue=myList" --deploy
dataflow:>Created and deployed new stream 'store-into-redis'
```

Options

The **redis** sink has the following options:

topicExpression
: a SpEL expression to use for topic **(String, no default)**

queueExpression
: a SpEL expression to use for queue **(String, no default)**

keyExpression
: a SpEL expression to use for keyExpression **(String, no default)**

key
: name for the key **(String, no default)**

queue
: name for the queue **(String, no default)**

topic
: name for the topic **(String, no default)**

12. Tasks

12.1 Timestamp (`timestamp`)

Executes a batch job that logs a timestamp.

The **timestamp** task has the following options:

format
: The timestamp format **(String, default: `yyyy-MM-dd HH:mm:ss.SSS`)**