Spring XD Guide

1.0.0

MarkFisher, MarkPollack, DavidTuranski, GunnarHillert, EricBottard, GaryRussell, IlayaperumalGopinathan, JenniferHickey, MichaelMinella, LukeTaylor, ThomasRisberg, WinstonKoh

Table of Contents

I. Reference Guide 1
1. Introduction 2
1.1. Overview 2
2. Getting Started 3
2.1. Requirements 3
2.2. Download Spring XD 3
2.3. Install Spring XD 3
2.4. Start the Runtime and the XD Shell 3
2.5. Create a Stream 4
2.6. Explore Spring XD 5
3. Running in Distributed Mode 6
3.1. Introduction 6
3.2. Using Redis 6
Installing Redis 6
Troubleshooting 7
Redis on Windows 7
Redis is not running 7
Starting Redis 7
3.3. Using RabbitMQ 7
Installing RabbitMQ 7
Launching RabbitMQ 8
3.4. Starting Spring XD in Distributed Mode 8
Choosing a Transport 8
Choosing a Store 9
Choosing an Analytics provider 9
Other Options 9
3.5. Using Hadoop 9
4. Architecture 10
4.1. Introduction 10
Runtime Architecture 10
DIRT Runtime 10
Support for other distributed runtimes 11
Single Node Runtime 11
Admin Server Architecture 12
Container Server Architecture 12
Streams 13
Stream Deployment 15
4.2. Jobs 19
4.3. Taps 19
5. Streams 20
5.1. Introduction 20
5.2. Creating a Simple Stream 20
5.3. Deleting a Stream 21
5.4. Deploying and Undeploying Streams 21
5.5. Other Source and Sink Types 21
5.6. Simple Stream Processing 21
5.7. DSL Syntax 22

5.8. Advanced Features 22
6. Modules 23
6.1. Introduction 23
6.2. Creating a Module 23
Modules and Spring 23
Integration Modules 24
6.3. Registering a Module 25
Modules with isolated classpath 25
6.4. Composing Modules 26
7. Sources 29
7.1. Introduction 29
7.2. HTTP 29
HTTP with options 30
7.3. Tail 30
Tail with options 30
Tail Status Events 31
7.4. File 31
File with options 31
7.5. Mail sources 32
7.6. Twitter Search 33
7.7. Twitter Stream 34
7.8. GemFire 34
Options 34
Example 34
7.9. GemFire Continuous Query (CQ) 35
Launching the XD GemFire Server 35
Options 35
7.10. Syslog 36
7.11. TCP 36
TCP with options 37
Available Decoders 37
Examples 38
Binary Data Example 39
7.12. TCP Client 39
TCP Client options 40
Implementing a simple conversation 40
7.13. RabbitMQ 41
RabbitMQ with Options 41
7.14. JMS 42
JMS with Options 42
7.15. Time 43
7.16. MQTT 43
Options 43
8. Processors 44
8.1. Introduction 44
8.2. Filter 44
Filter with SpEL expression 44
Filter with Groovy Script 44
8.3. JSON Field Value Filter 45
8.4. Transform 45

Transform with SpEL expression 45
Transform with Groovy Script 45
8.5. JSON Field Extractor 46
8.6. Script 46
8.7. Splitter 46
8.8. Aggregator 46
9. Sinks 48
9.1. Introduction 48
9.2. Log 48
9.3. File Sink 49
File with Options 49
9.4. Hadoop (HDFS) 49
HDFS with Options 50
9.5. JDBC 51
JDBC with Options 52
9.6. TCP 52
TCP with Options 53
Available Encoders 53
An Additional Example 54
9.7. Mail 54
9.8. RabbitMQ 55
RabbitMQ with Options 56
9.9. GemFire Server 56
Launching the XD GemFire Server 56
Gemfire sinks 57
Example 57
9.10. Splunk Server 57
Splunk sinks 58
Setup Splunk for TCP Input 58
Example 58
9.11. MQTT 58
Options 58
9.12. Dynamic Router 59
SpEL-based Routing 59
Groovy-based Routing 60
Options 61
10. Taps 62
10.1. Introduction 62
10.2. Tap Lifecycle 62
11. Batch Jobs 63
11.1. Introduction 63
11.2. Setting up a simple Batch Job 63
Creating the Tasklet 63
Setting Up the Application Context 63
11.3. Creating your Job 64
11.4. Launching a job 65
Ad-hoc 65
Launch the Batch using Cron-Trigger 65
Launch the Batch using a Fixed-Delay-Trigger 66
Launch job as a part of event flow 66

11.5. Retrieve job notifications 66
11.6. Removing Batch Jobs 67
11.7. Pre-Packaged Batch Jobs 67
Import Files to HDFS (`filehdfs`) 67
Import Files to JDBC (`filejdbc`) 67
HDFS to JDBC Export (`hdfsjdbc`) 68
HDFS to MongoDB Export (`hdfsmongodb`) 68
12. Analytics 69
12.1. Introduction 69
12.2. Counter 69
12.3. Field Value Counter 70
12.4. Aggregate Counter 71
12.5. Gauge 71
Simple Tap Example 72
12.6. Rich Gauge 72
Simple Tap Example 72
Stock Price Example 73
Improved Stock Price Example 73
12.7. Accessing Analytics Data over the RESTful API 74
13. DSL Reference 77
13.1. Introduction 77
13.2. Pipes and filters 77
13.3. Module parameters 77
13.4. Named channels 77
13.5. Labels 78
14. Tuples 79
14.1. Introduction 79
Creating a Tuple 79
Getting Tuple values 80
Using SpEL expressions to filter a tuple 81
15. Samples 83
15.1. Syslog ingestion into HDFS 83
A sample configuration using syslog-ng 83
II. Appendices 84
A. Installing Hadoop 85
A.1. Installing Hadoop 85
Download 85
Java Setup 85
Setup SSH 86
Setting the Namenode Port 86
Further Configuration File Changes 87
A.2. Running Hadoop 87
B. Creating a Source Module 89
B.1. Introduction 89
B.2. Create the module Application Context file 89
Make the module configurable 90
B.3. Test the module locally 90
Create a project 90
Create the Spring integration test 91
B.4. Deploy the module 92

B.5. Test the deployed module 92
C. Creating a Processor Module 94
C.1. Introduction 94
C.2. Write the Transformer Code 94
C.3. Create the module Application Context File 94
C.4. Deploy the Module 95
C.5. Test the deployed module 95
D. Creating a Sink Module 96
D.1. Introduction 96
D.2. Create the module Application Context file 96
D.3. Make the module configurable 97
D.4. Test the module locally 97
Create a project 97
Create the Spring integration test 98
Run the test 99
D.5. Deploy the module 100
D.6. Test the deployed module 100
E. Building Spring XD 101
E.1. Instructions 101
E.2. IDE support 101
E.3. Running JavaScript UI Tests 101
F. XD Shell Command Reference 103
F.1. Base Commands 103
admin config server 103
admin config info 103
F.2. Runtime Commands 103
runtime containers 103
runtime modules 103
F.3. Stream Commands 103
stream create 103
stream destroy 104
stream all destroy 104
stream deploy 104
stream all deploy 104
stream undeploy 104
stream all undeploy 104
stream list 105
F.4. Job Commands 105
job create 105
job list 105
job deploy 105
job all deploy 105
job launch 106
job undeploy 106
job all undeploy 106
job destroy 106
job all destroy 106
F.5. Module Commands 107
module compose 107
module delete 107

module display 107
module list 107
F.6. Metrics Commands 107
counter list 107
counter delete 108
counter display 108
fieldvaluecounter list 108
fieldvaluecounter delete 108
fieldvaluecounter display 108
aggregatecounter list 109
aggregatecounter delete 109
aggregatecounter display 109
gauge list 109
gauge delete 109
gauge display 110
richgauge list 110
richgauge delete 110
richgauge display 110
F.7. Http Commands 110
http post 110
F.8. Hadoop Configuration Commands 111
hadoop config props set 111
hadoop config props get 111
hadoop config info 111
hadoop config load 111
hadoop config props list 111
hadoop config fs 112
hadoop config jt 112
F.9. Hadoop FileSystem Commands 112
hadoop fs get 112
hadoop fs put 112
hadoop fs count 112
hadoop fs mkdir 113
hadoop fs tail 113
hadoop fs ls 113
hadoop fs cat 113
hadoop fs chgrp 113
hadoop fs chown 114
hadoop fs chmod 114
hadoop fs copyFromLocal 114
hadoop fs moveFromLocal 114
hadoop fs copyToLocal 115
hadoop fs copyMergeToLocal 115
hadoop fs cp 115
hadoop fs mv 116
hadoop fs du 116
hadoop fs expunge 116
hadoop fs rm 116
hadoop fs setrep 116
hadoop fs text 117

hadoop fs touchz 117

Part I. Reference Guide

1. Introduction

1.1 Overview

Spring XD is a unified, distributed, and extensible service for data ingestion, real time analytics, batch processing, and data export. The Spring XD project is an open source Apache 2 License licenced project whose goal is to tackle big data complexity. Much of the complexity in building real-world big data applications is related to integrating many disparate systems into one cohesive solution across a range of use-cases. Common use-cases encountered in creating a comprehensive big data solution are

- High throughput distributed data ingestion from a variety of input sources into big data store such as HDFS or Splunk
- Real-time analytics at ingestion time, e.g. gathering metrics and counting values.
- Workflow management via batch jobs. The jobs combine interactions with standard enterprise systems (e.g. RDBMS) as well as Hadoop operations (e.g. MapReduce, HDFS, Pig, Hive or Cascading).
- High throughput data export, e.g. from HDFS to a RDBMS or NoSQL database.

The Spring XD project aims to provide a one stop shop solution for these use-cases.

2. Getting Started

2.1 Requirements

To get started, make sure your system has as a minimum **Java JDK 6** or newer installed. **Java JDK 7** is recommended.

2.2 Download Spring XD

If you want to try out Spring XD, we'd recommend downloading a snapshot build, since things are changing quite fast. A snapshot distribution can be downloaded from the spring snapshots repository. You can also build the project from source if you wish. The wiki content should also be kept up to date with the current snapshot so if you are reading this on the github website, things may have changed since the last milestone.

Unzip the distribution which will unpack to a single installation directory. All the commands below are executed from this directory, so change into it before proceeding.

If you are sure you want the previous milestone release, you can also download the distribution **spring-xd-1.0.0.M4-dist.zip** and its accompanying documentation.

```
$ cd spring-xd-1.0.0.M4
```

Set the environment variable `XD_HOME` to the installation directory `<root-install-dir>\spring-xd\xd`

2.3 Install Spring XD

Spring XD can be run in two different modes. There's a single-node runtime option for testing and development, and there's a distributed runtime which supports distribution of processing tasks across multiple nodes. This document will get you up and running quickly with a single-node runtime. See Running Distributed Mode for details on setting up a distributed runtime.

2.4 Start the Runtime and the XD Shell

The single node option is the easiest to get started with. It runs everything you need in a single process. To start it, you just need to `cd` to the `xd` directory and run the following command

```
xd/bin>$ ./xd-singlenode
```

In a separate terminal, `cd` into the `shell` directory and start the XD shell, which you can use to issue commands.

```
shell/bin>$ ./xd-shell
  _____                           __   _______
 /  ___|          (-)             \ \ / /  _  \
 \ `--. _ __  _ __ _ _ __   __ _   \ V /| | | |
  `--. \ '_ \| '__| | '_ \ / _` |  / ^ \| | | |
 /\__/ / |_) | |  | | | | | (_| | / / \ \ |/ /
 \____/| .__/|_|  |_|_| |_|\__, | \/   \/___/
       | |                   __/ |
       |_|                  |___/
eXtreme Data
1.0.0.M4 | Admin Server Target: http://localhost:8080
Welcome to the Spring XD shell. For assistance hit TAB or type "help".
xd:>
```

The shell is a more user-friendly front end to the REST API which Spring XD exposes to clients. The URL of the currently targeted Spring XD server is shown at startup.

Note

If the server could not be reached, the prompt will read

```
server-unknown:>
```

You should now be able to start using Spring XD.

2.5 Create a Stream

In Spring XD, a basic stream defines the ingestion of event driven data from a source to a sink that passes through any number of processors. You can create a new stream by issuing a `stream create` command from the XD shell. Stream defintions are built from a simple DSL. For example, execute:

```
xd:> stream create --definition "time | log" --name ticktock
```

This defines a stream named `ticktock` based off the DSL expression `time | log`. The DSL uses the "pipe" symbol `|`, to connect a source to a sink. The stream server finds the `time` and `log` definitions in the modules directory and uses them to setup the stream. In this simple example, the time source simply sends the current time as a message each second, and the log sink outputs it using the logging framework at the WARN logging level. In the shell where you started the server, you will see log output similar to that listed below

```
13:09:53,812  INFO http-bio-8080-exec-1 module.SimpleModule:109 - started module: Module
 [name=log, type=sink]
13:09:53,813  INFO http-bio-8080-exec-1 module.ModuleDeployer:111 - launched sink module:
 ticktock:log:1
13:09:53,911  INFO http-bio-8080-exec-1 module.SimpleModule:109 - started module: Module
 [name=time, type=source]
13:09:53,912  INFO http-bio-8080-exec-1 module.ModuleDeployer:111 - launched source
 module: ticktock:time:0
13:09:53,945  WARN task-scheduler-1 logger.ticktock:141 - 2013-06-11 13:09:53
13:09:54,948  WARN task-scheduler-1 logger.ticktock:141 - 2013-06-11 13:09:54
13:09:55,949  WARN task-scheduler-2 logger.ticktock:141 - 2013-06-11 13:09:55
```

To stop the stream, and remove the definition completely, you can use the `stream destroy` command:

```
xd:>stream destroy --name ticktock
```

It is also possibly to stop and restart the stream instead, using the `undeploy` and `deploy` commands. The shell supports command completion so you can hit the `tab` key to see which commands and options are available.

2.6 Explore Spring XD

Learn about the modules available in Spring XD in the Sources, Processors, and Sinks sections of the documentation.

Don't see what you're looking for? Create a custom module: source, processor or sink (and then consider contributing it back to Spring XD).

Want to add some analytics to your stream? Check out the Taps and Analytics sections.

3. Running in Distributed Mode

3.1 Introduction

The Spring XD distributed runtime (DIRT) supports distribution of processing tasks across multiple nodes. See Getting Started for information on running Spring XD as a single node.

Spring XD can use several middlewares when running in distributed mode. At the time of writing, **Redis** and **RabbitMQ** are available options.

Let's see how to install those first, before diving into the specifics of running Spring XD. Again, those are alternatives when it comes to transport middleware used, so you need only one (although practically, Redis may be required for other purposes, for example storage of definitions or Analytics).

Redis is actually the default when it comes to running in distributed mode, so let's start with that.

3.2 Using Redis

Installing Redis

If you already have a running instance of **Redis** it can be used for Spring XD. By default Spring XD will try to use a *Redis* instance running on **localhost** using **port 6379**. You can change that in the `redis.properties` file residing in the `config/` directory.

If you don't have a pre-existing installation of *Redis*, you can use the *Spring XD* provided instance (For Linux and Mac). Inside the *Spring XD* installation directory (spring-xd) do:

```
$ cd redis/bin
$ ./install-redis
```

This will compile the *Redis* source tar and add the *Redis* executables under redis/bin:

- redis-check-dump
- redis-sentinel
- redis-benchmark
- redis-cli
- redis-server

You are now ready to start *Redis* by executing

```
$ ./redis-server
```

Tip

For further information on installing *Redis* in general, please checkout the Redis Quick Start guide. If you are using *Mac OS*, you can also install *Redis* via Homebrew

Troubleshooting

Redis on Windows

Presently, *Spring XD* does not ship *Windows* binaries for *Redis* (See XD-151). However, *Microsoft* is actively working on supporting *Redis* on *Windows*. You can download *Windows Redis* binaries from:

https://github.com/MSOpenTech/redis/tree/2.6/bin/release

Redis is not running

If you try to run *Spring XD* and Redis is NOT running, you will see the following exception:

```
11:26:37,830 ERROR main launcher.RedisContainerLauncher:85 - Unable to connect to Redis
 on localhost:6379; nested exception is com.lambdaworks.redis.RedisException: Unable to
 connect
Redis does not seem to be running. Did you install and start Redis? Please see the Getting
 Started section of the guide for instructions.
```

Starting Redis

```
$ redis-server
```

You should see something like this:

```
[35142] 01 May 14:36:28.939 # Warning: no config file specified, using the default config.
 In order to specify a config file use redis-server /path/to/redis.conf
[35142] 01 May 14:36:28.940 * Max number of open files set to 10032
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 2.6.12 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in stand alone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 35142
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           http://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

[35142] 01 May 14:36:28.941 # Server started, Redis version 2.6.12
[35142] 01 May 14:36:28.941 * The server is now ready to accept connections on port 6379
```

3.3 Using RabbitMQ

Installing RabbitMQ

If you already have a running instance of *RabbitMQ* it can be used for Spring XD. By default Spring XD will try to use a *Rabbit* instance running on **localhost** using **port 5674**. The default account credentials of **guest/guest** are assumed. You can change that in the `rabbit.properties` file residing in the `config/` directory.

If you don't have a *RabbitMQ* installation already, head over to http://www.rabbitmq.com and follow the instructions. Packages are provided for Windows, Mac and various flavor of unix/linux.

Launching RabbitMQ

Start the **RabbitMQ** broker by running the rabbitmq-server script:

```
$ rabbitmq-server
```

You should see something similar to this:

```
              RabbitMQ 3.1.1. Copyright (C) 2007-2013 VMware, Inc.
  ##  ##      Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########  Logs: /usr/local/var/log/rabbitmq/rabbit@localhost.log
  ######  ##        /usr/local/var/log/rabbitmq/rabbit@localhost-sasl.log
  ##########
              Starting broker... completed with 7 plugins.
```

3.4 Starting Spring XD in Distributed Mode

Spring XD consists of two servers

- XDAdmin - controls deployment of modules into containers
- XDContainer - executes modules

You can start the `xd-container` and `xd-admin` servers individually as follows:

```
xd/bin>$ ./xd-admin
xd/bin>$ ./xd-container
```

Choosing a Transport

The --transport option drives the choice of middleware to use. As stated previously, **Redis** is currently the default, so the above example is equivalent to

```
xd/bin>$ ./xd-admin --transport redis
xd/bin>$ ./xd-container --transport redis
```

To run using **RabbitMQ**, simply issue the following commands:

```
xd/bin>$ ./xd-admin --transport rabbit
xd/bin>$ ./xd-container --transport rabbit
```

Note

If you have multiple XD systems (i.e. an xd-admin server and 0+ containers) using different Redis instances for storage but sharing a single RabbitMQ server for transport, you may encounter issues if each system contains streams of the same name. We recommend using a different RabbitMQ virtual host for each system. Update the `rabbit.vhost` property in `rabbit.properties` to point XD at the correct virtual host.

Choosing a Store

By default, the xd-admin server stores stream definitions and other information in Redis, using the connection parameters specified in `redis.properties`. Use the --store option to specify another storage type. Currently, only "redis" and "memory" are available.

```
xd/bin>$ ./xd-admin --store memory
```

Choosing an Analytics provider

By default, the xd-container will store Analytics data in redis. Use the --analytics option to specify another backing store for Analytics data. currently only "redis" and "memory are available.

```
xd/bin>$ ./xd-container --analytics memory
```

Other Options

There are additional configuration options available for these scripts:

To specify the location of the Spring XD install,

```
xd/bin>$ ./xd-admin --xdHomeDir <xd-install-directory>
xd/bin>$ ./xd-container --xdHomeDir <xd-install-directory>
```

To specify the http port of the XDAdmin server,

```
xd/bin>$ ./xd-admin --httpPort <httpPort>
```

3.5 Using Hadoop

Spring XD supports the following Hadoop distributions:

- hadoop12 - Apache Hadoop 1.2.1 (default)
- hadoop20 - Apache Hadoop 2.0.6-alpha
- phd1 - Pivotal HD 1.0
- cdh4 - Cloudera CDH 4.3.1
- hdp13 - Hortonworks Data Platform 1.3

To specify the distribution to use for Hadoop client connections,

```
xd/bin>$ ./xd-admin --hadoopDistro <distribution>
xd/bin>$ ./xd-container --hadoopDistro <distribution>
```

Pass in the `--help` option to see other configuration properties.

4. Architecture

4.1 Introduction

Spring XD is a unified, distributed, and extensible service for data ingestion, real time analytics, batch processing, and data export. The foundations of XD's architecture are based on the over 100+ man years of work that have gone into the Spring Batch, Integration and Data projects. Building upon these projects, Spring XD provides servers and a configuration DSL that you can immediately use to start processing data. You do not need to build an application yourself from a collection of jars to start using Spring XD.

Spring XD has two modes of operation - single and multi-node. The first is a single process that is responsible for all processing and administration. This mode helps you get started easily and simplifies the development and testing of your application. The second is a distributed mode, where processing tasks can be spread across a cluster of machines and an administrative server sends commands to control processing tasks executing on the cluster.

Runtime Architecture

The key components in Spring XD are the XD Admin and XD Container Servers. Using a high-level DSL, you post the description of the required processing tasks to the Admin server over HTTP. The Admin server then maps the processing tasks into processing modules. A module is a unit of execution and is implemented as a Spring ApplicationContext. A simple distributed runtime is provided that will assign modules to execute across multiple XD Container servers. A single XD Container server can run multiple modules. When using the single node runtime, all modules are run in a single XD Container and the XD Admin server is run in the same process.

DIRT Runtime

A simple distributed runtime, called Distributed Integration Runtime, aka DIRT, will distribute the processing tasks across multiple XD Container instances. The XD Admin server breaks up a processing task into individual module definitions and publishes them to a shared queue (backed by Redis or RabbitMQ depending upon the provided *transport* option). Each container picks up a module definition off the queue, in a round-robin like manner, and creates a Spring ApplicationContext to run that module.

To reduce the number of hops across messaging middleware between them, multiple modules may be composed into larger deployment units that act as a single module. To learn more about that feature, refer to the Composing Modules section.

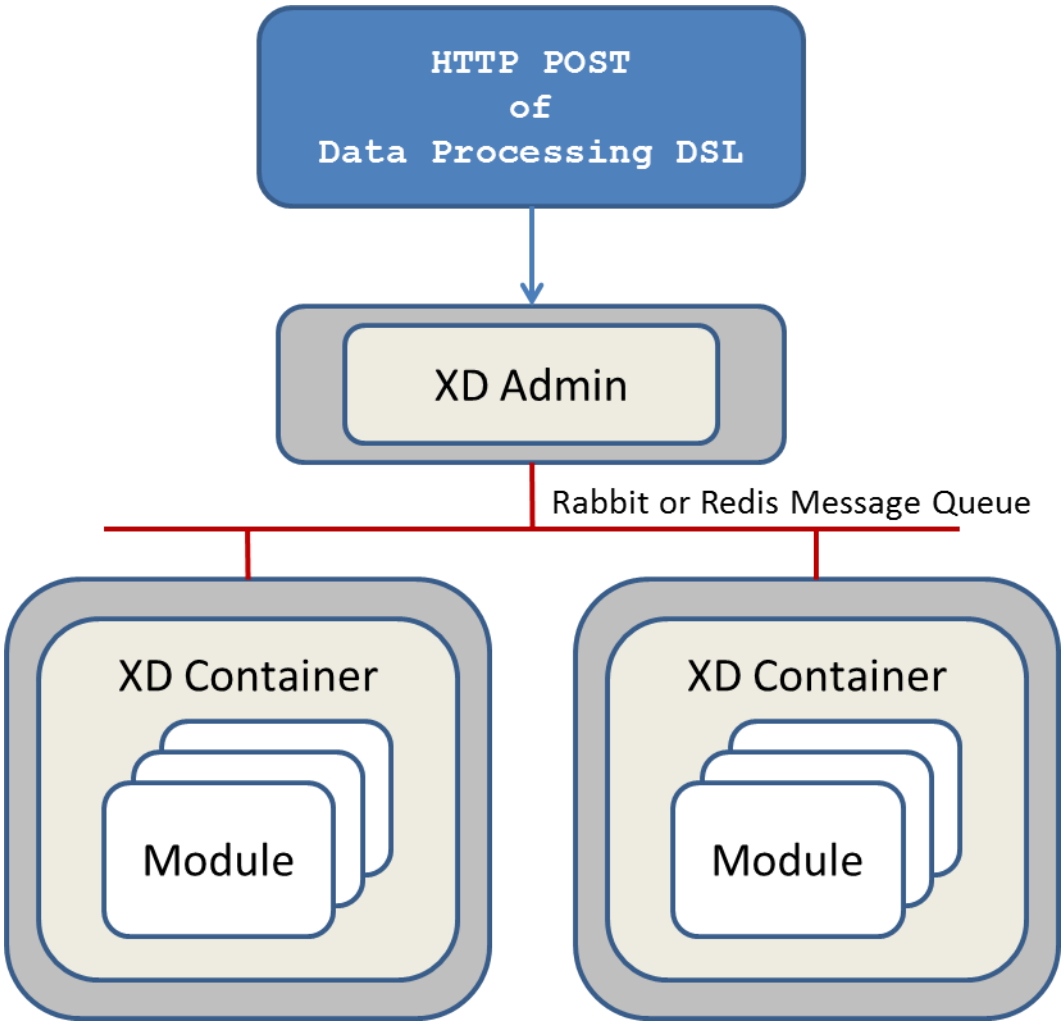

Figure 4.1. The XD Admin Server sending module definitions to each XD Container

How the processing task is broken down into modules is discussed in the section Container Server Architecture.

Support for other distributed runtimes

In the 1.0 release, you are responsible for starting up a single XD Admin server and one or more XD Containers. The 1.1 release will support running XD on top of other distributed runtime environments such as Hadoop's YARN architecture and CloudFoundry.

Single Node Runtime

For testing and development purposes, a single node runtime is provided that runs the Admin and Container servers in the same process. The communication to the XD Admin server is over HTTP and the XD Admin server communicates to an in-process XD Container using an in-memory queue.

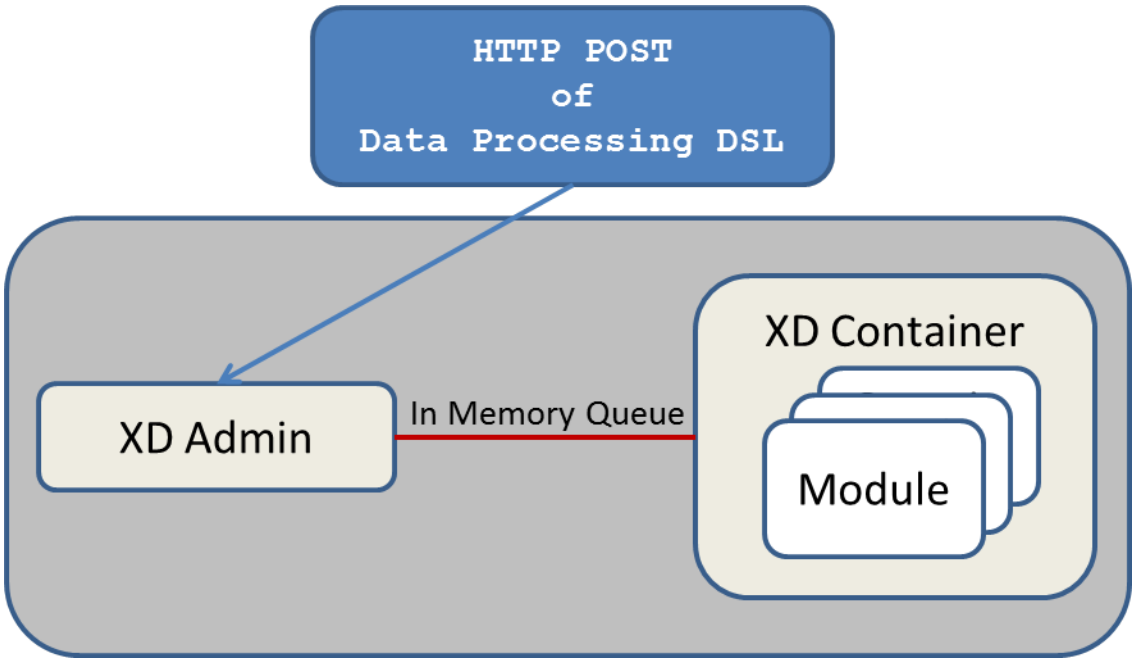

Figure 4.2. Single Node Runtime

Admin Server Architecture

The Admin Server uses an embedded servlet container and exposes two endpoints for creating and deleting the modules required to perform data processing tasks as declared in the DSL. The Admin Server is implemented using Spring's MVC framework and the Spring HATEOAS library to create REST representations that follow the HATEOAS principle. The Admin Server communicates with the Container Servers using a pluggable transport based, the default uses Redis queues.

Container Server Architecture

The key components of data processing in Spring XD are

- Streams
- Jobs
- Taps

Streams define how event driven data is collected, processed, and stored or forwarded. For example, a stream might collect syslog data, filter, and store it in HDFS.

Jobs define how coarse grained and time consuming batch processing steps are orchestrated, for example a job could be be defined to coordinate performing HDFS operations and the subsequent execution of multiple MapReduce processing tasks.

Taps are used to process data in a non-invasive way as data is being processed by a Stream or a Job. Much like wiretaps used on telephones, a Tap on a Stream lets you consume data at any point along the Stream's processing pipeline. The behavior of the original stream is unaffected by the presence of the Tap.

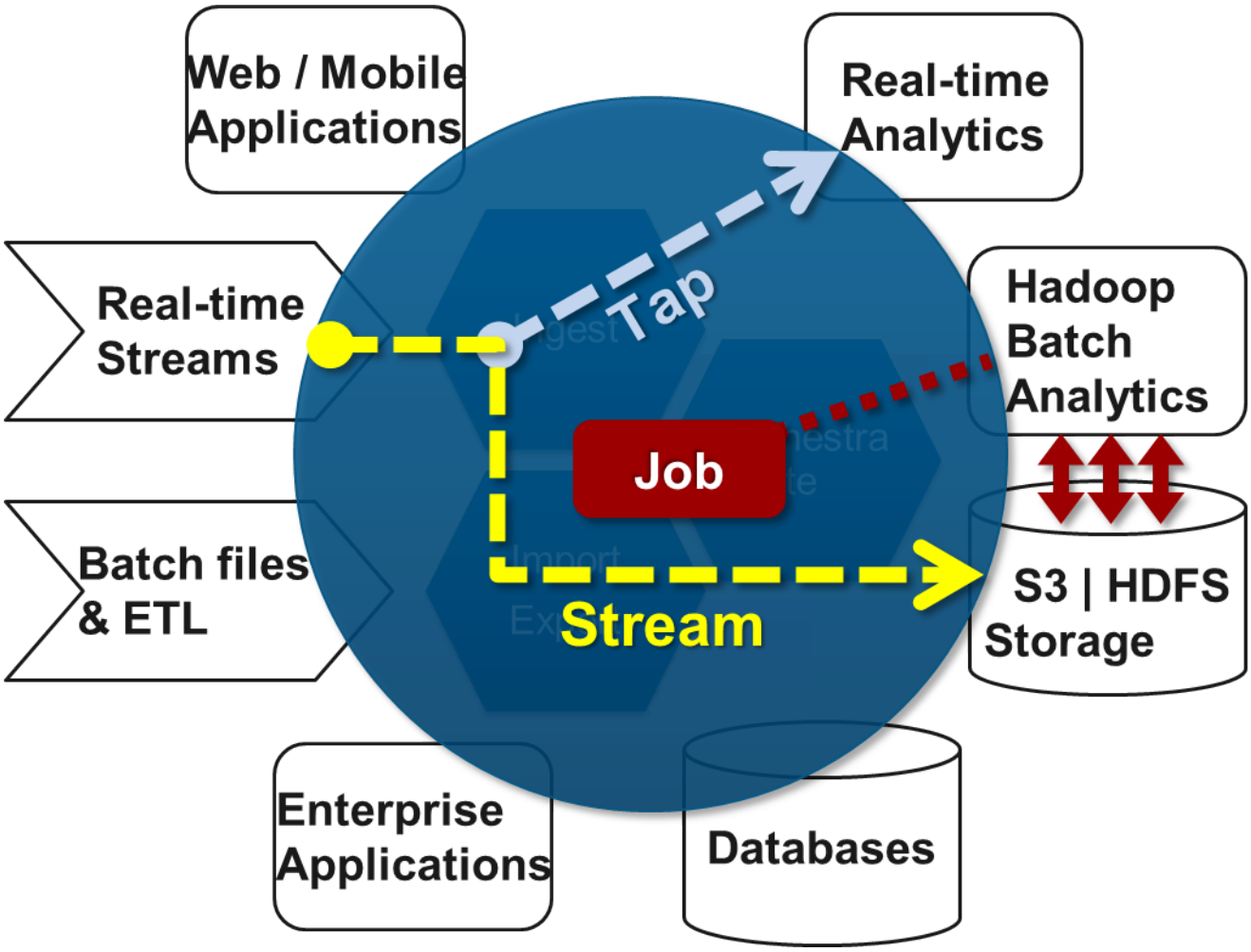

Figure 4.3. Taps, Jobs, and Streams

Streams

The programming model for processing event streams in Spring XD is based on the well known Enterprise Integration Patterns as implemented by components in the Spring Integration project. The programming model was designed to be easy to test components.

Streams consist of the following types of modules: * Input sources * Processing steps * Output sinks

Input sources produce messages from a variety of sources, e.g. syslog, tcp, http. A message contains a payload of data and a collection of key-value headers. Messages flow through message channels from the source, through optional processing steps, to the output sink. The output sink will often write the message to a file system, such as HDFS, but may also forward the message over tcp, http, or another type of middleware.

A stream that consists of a input source and a output sink is shown below

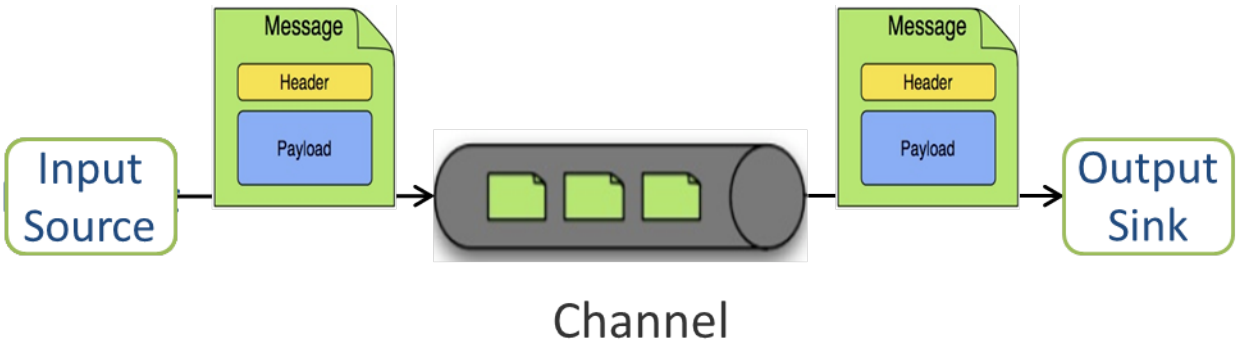

Figure 4.4. Foundational components of the Stream processing model

A stream that incorporates processing steps is shown below

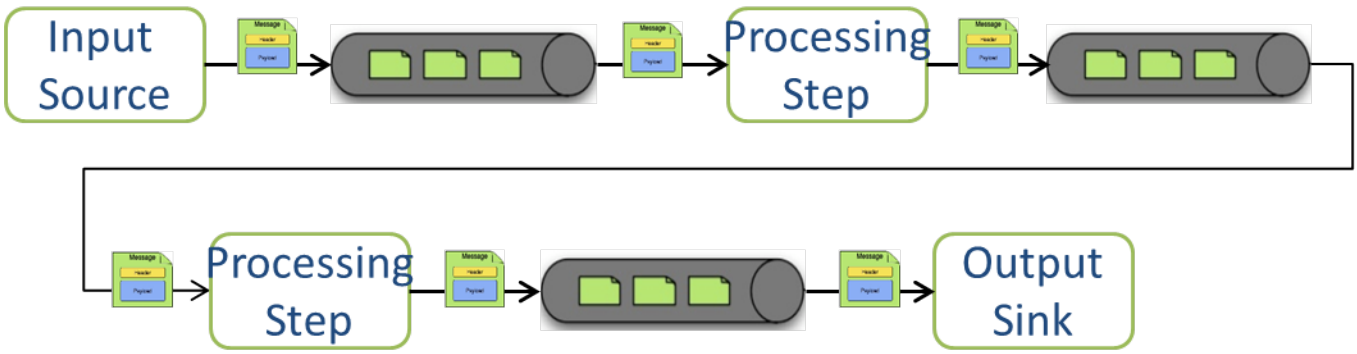

Figure 4.5. Stream processing with multiple steps

For simple linear processing streams, an analogy can be made with the UNIX pipes and filters model. Filters represent any component that produces, processes or consumes events. This corresponds to sources, processing steps, and sinks in a stream. Pipes represent the way data is transported between the Filters. This corresponds to the Message Channel that moves data through a stream.

A simple stream definition using UNIX pipes and filters syntax that takes data sent via a HTTP post and writes it to a file (with no processing done in between) can be expressed as

```
http | file
```

The pipe symbol represents a message channel that passes data from the HTTP source to the File sink. The message channel implementation can either be backed with a local in-memory transport, Redis queues, or RabbitMQ. Future releases will support backing the message channel with other transports such as JMS.

Note that the UNIX pipes and filter syntax is the basis for the DSL that Spring XD uses to describe simple linear flows, but we will significantly extend the syntax to cover non-linear flow in a subsequent release.

The programming model for processing steps in a stream comes from the Spring Integration project. The central concept is one of a Message Handler class, which relies on simple coding conventions to Map incoming messages to processing methods. For example, using an http source you can process the body of an HTTP POST request using the following class

```
public class SimpleProcessor {

  public String process(String payload) {
    return payload.toUpperCase();
  }

}
```

The payload of the incoming Message is passed as a string to the method process. The contents of the payload is the body of the http request as we are using a http source. The non-void return value is used as the payload of the Message passed to the next step. These programming conventions make it very easy to test your Processor component in isolation. There are several processing components provided in Spring XD that do not require you to write any code, such as a filter and transformer that use the Spring Expression Language or Groovy. For example, adding a processing step, such as a transformer, in a stream processing definition can be as simple as

```
http | transformer --expression=payload.toUpperCase() | file
```

For more information on processing modules, refer to the Processors section.

Stream Deployment

The Container Server listens for module deployment requests sent from the Admin Server via the control bus. When the container node receives a module deployment request, it connects the module's input and output channels to the data bus used to transport messages during stream processing. In a single node configuration, both the control bus and data bus use in-memory direct channels. In a distributed configuration, control bus and data bus communications are backed by the configured transport middleware. Redis and Rabbit are both provided with the Spring XD distrubution, but other transports are envisioned for future releases. Currently, the control bus and the data bus use the same transport middleware. Future releases will allow separate configuration of the control bus and data bus (e.g., Redis for control and Rabbit for data).

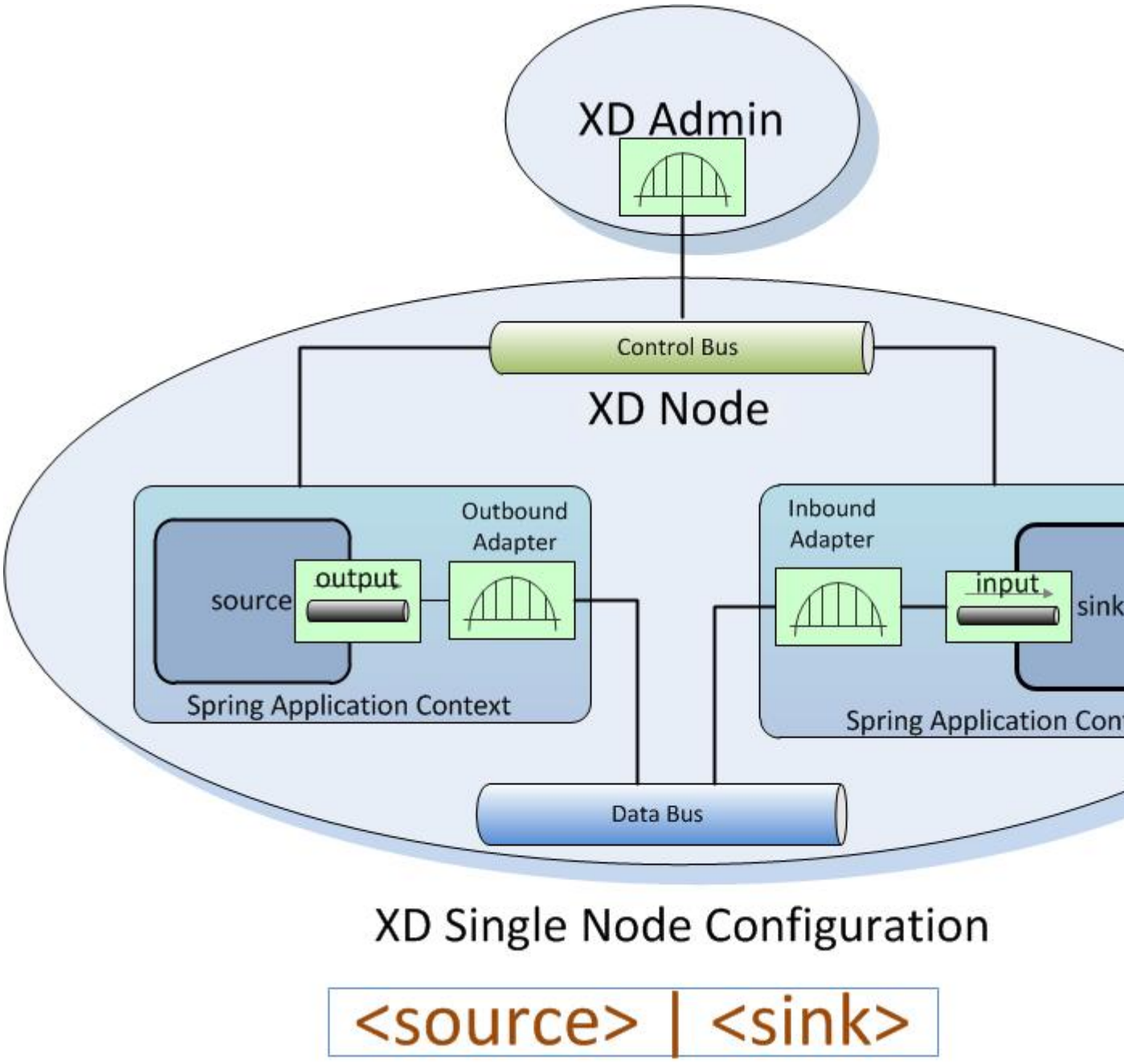

Figure 4.6. A Stream Deployed in a single node server

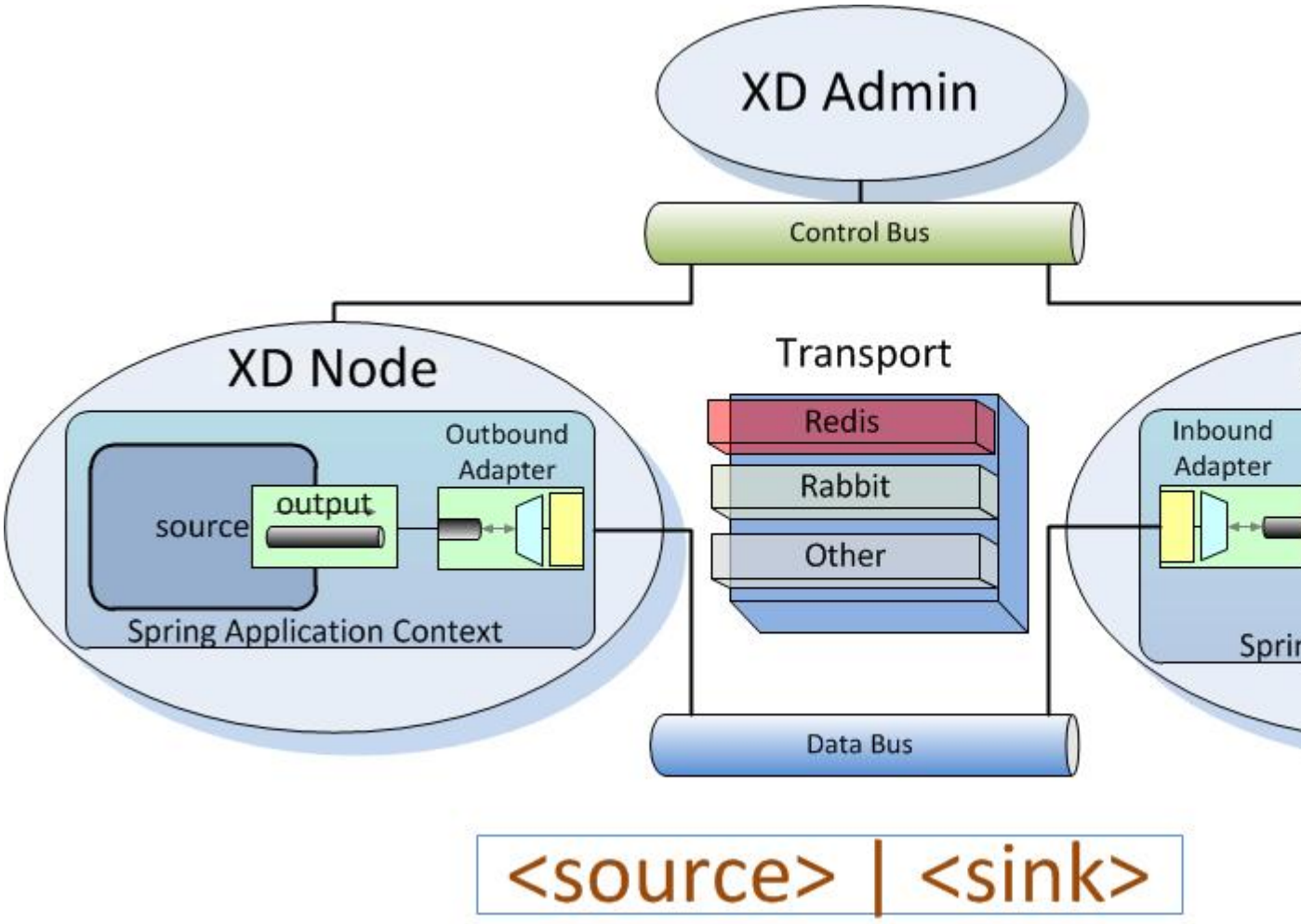

Figure 4.7. A Stream Deployed in a distributed runtime

In the `http | file` example, a module deployment request sent for the http module and another request is sent for the file module. The definition of a module is stored in a Module Registry, which is a Spring XML configuration file. The module definition contains variable placeholders that allow you to customize the behavior of the module. For example, setting the http listening port would be done by passing in the option `--port`, e.g. `http --port=8090 | file`, which is in turn used to substitute a placeholder value in the module definition.

The Module Registry is backed by the filesystem and corresponds to the directory `<xd-install-directory>/modules`. When a module deployment request is processed by the Container, the module definition is loaded from the registry and a Spring ApplicationContext is created.

Using the DIRT runtime, the http | file example would map onto the following runtime architecture

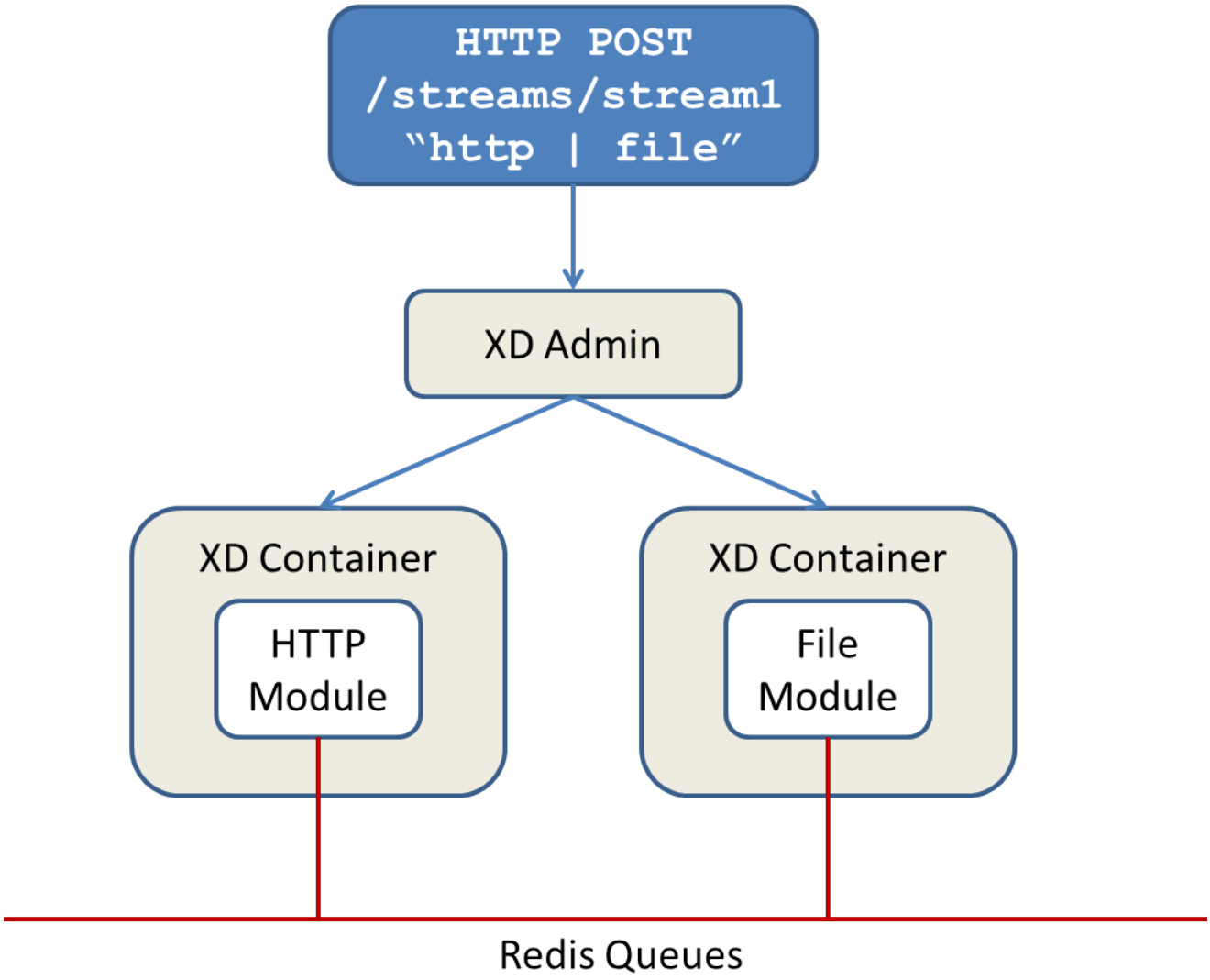

Figure 4.8. Distributed HTTP to File Stream

Data produced by the HTTP module is sent over a Redis Queue and is consumed by the File module. If there was a filter processing module in the steam definition, e.g `http | filter | file` that would map onto the following DIRT runtime architecture.

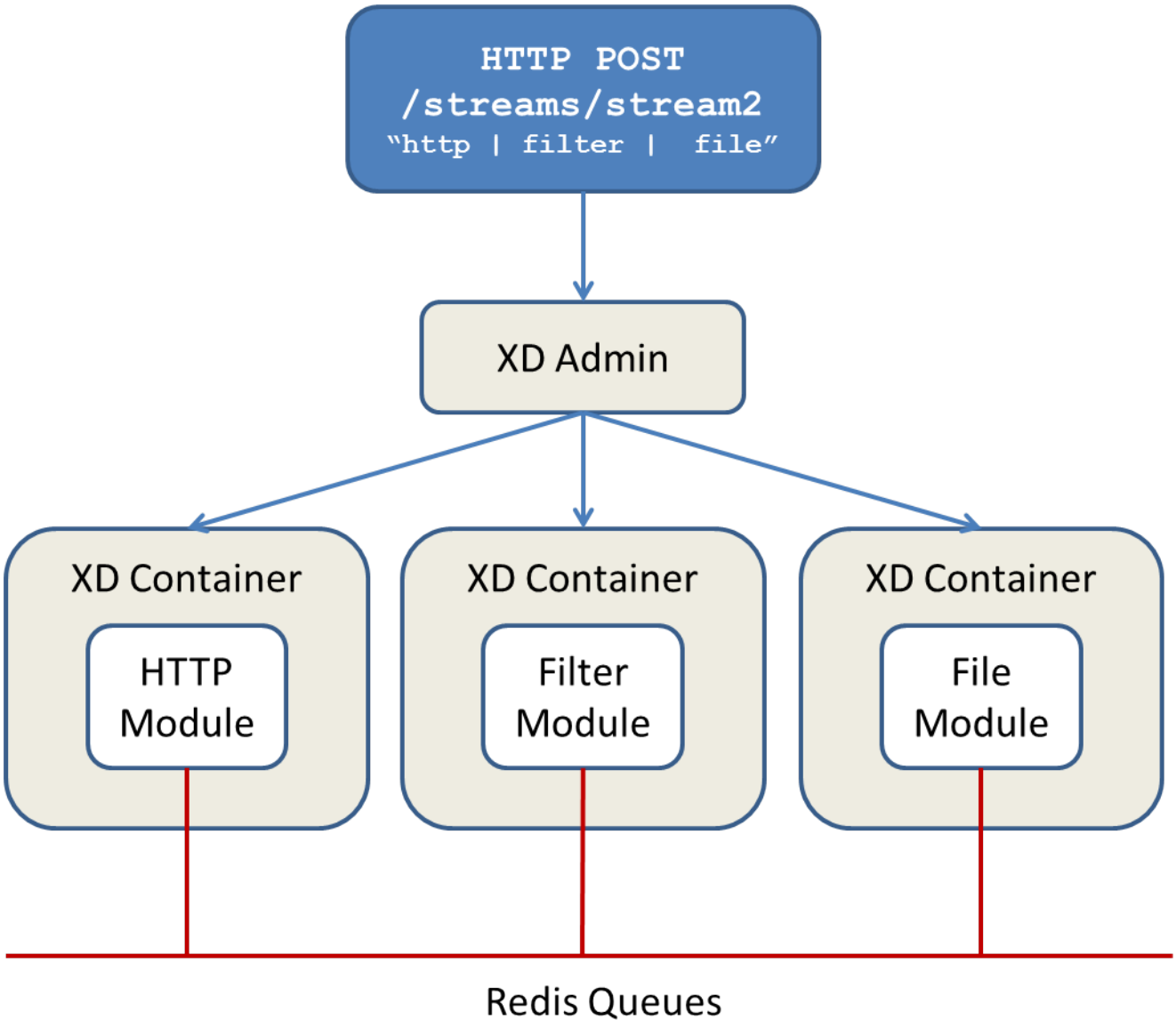

Figure 4.9. Distributed HTTP to Filter to File Stream

4.2 Jobs

The creation and execution of Batch jobs builds upon the functionality available in the Spring Batch and Spring for Apache Hadoop projects. See the Batch Jobs section for more information.

4.3 Taps

Taps provide a non-invasive way to consume the data that is being processed by either a Stream or a Job, much like a real time telephone wire tap lets you eavesdrop on telephone conversations. Taps are recommended as way to collect metrics and perform analytics on a Stream of data. See the section Taps for more information.

5. Streams

5.1 Introduction

In Spring XD, a basic stream defines the ingestion of event driven data from a *source* to a *sink* that passes through any number of *processors*. Stream processing is performed inside the XD Containers and the deployment of stream definitions to containers is done via the XD Admin Server. The Getting Started section shows you how to start these servers and how to start and use the Spring XD shell

Sources, sinks and processors are predefined configurations of a *module*. Module definitions are found in the `xd/modules` directory. [1]. Modules definitions are standard Spring configuration files that use existing Spring classes, such as Input/Output adapters and Transformers from Spring Integration that support general Enterprise Integration Patterns.

A high level DSL is used to create stream definitions. The DSL to define a stream that has an http source and a file sink (with no processors) is shown below

```
http | file
```

The DSL mimics a UNIX pipes and filters syntax. Default values for ports and filenames are used in this example but can be overriden using `--` options, such as

```
http --port=8091 | file --dir=/tmp/httpdata/
```

To create these stream definitions you make an HTTP POST request to the XD Admin Server. More details can be found in the sections below.

5.2 Creating a Simple Stream

The XD Admin server [5] exposes a full RESTful API for managing the lifecycle of stream definitions, but the easiest way to use the XD shell. Start the shell as described in the Getting Start ed section

New streams are created by posting stream definitions. The definitions are built from a simple DSL. For example, let's walk through what happens if we execute the following shell command

```
xd:> stream create --definition "time | log" --name ticktock
```

This defines a stream named `ticktock` based off the DSL expression `time | log`. The DSL uses the "pipe" symbol `|`, to connect a source to a sink. The stream server finds the `time` and `log` definitions in the modules directory and uses them to setup the stream. In this simple example, the time source simply sends the current time as a message each second, and the log sink outputs it using the logging framework.

```
processing module 'Module [name=log, type=sink]' from group 'ticktock' with index: 1
processing module 'Module [name=time, type=source]' from group 'ticktock' with index: 0
17:26:18,774  WARN ThreadPoolTaskScheduler-1 logger.ticktock:141 - Thu May 23 17:26:18 EDT
 2013
```

[1] Using the filesystem is just one possible way of storing module defintions. Other backends will be supported in the future, e.g. Redis.

[5] The server is implemented by the `AdminMain` class in the `spring-xd-dirt` subproject

5.3 Deleting a Stream

You can delete a stream by issuing the `stream destroy` command from the shell:

```
xd:> stream destroy --name ticktock
```

5.4 Deploying and Undeploying Streams

Often you will want to stop a stream, but retain the name and definition for future use. In that case you can `undeploy` the stream by name and issue the `deploy` command at a later time to restart it.

```
xd:> stream undeploy --name ticktock
xd:> stream deploy --name ticktock
```

5.5 Other Source and Sink Types

Let's try something a bit more complicated and swap out the `time` source for something else. Another supported source type is `http`, which accepts data for ingestion over HTTP POSTs. Note that the `http` source accepts data on a different port (default 9000) from the Admin Server (default 8080).

To create a stream using an `http` source, but still using the same `log` sink, we would change the original command above to

```
xd:> stream create --definition "http | log" --name myhttpstream
```

which will produce the following output from the server

```
processing module 'Module [name=log, type=sink]' from group 'myhttpstream' with index: 1
processing module 'Module [name=http, type=source]' from group 'myhttpstream' with index:
 0
```

Note that we don't see any other output this time until we actually post some data (using shell command)

```
xd:> http post --target http://localhost:9000 --data "hello"
xd:> http post --target http://localhost:9000 --data "goodbye"
```

and the stream will then funnel the data from the http source to the output log implemented by the log sink

```
15:08:01,676  WARN ThreadPoolTaskScheduler-1 logger.myhttpstream:141 - hello
15:08:12,520  WARN ThreadPoolTaskScheduler-1 logger.myhttpstream:141 - goodbye
```

Of course, we could also change the sink implementation. You could pipe the output to a file (`file`), to hadoop (`hdfs`) or to any of the other sink modules which are provided. You can also define your own modules.

5.6 Simple Stream Processing

As an example of a simple processing step, we can transform the payload of the HTTP posted data to upper case using the stream definitions

```
http | transform --expression=payload.toUpperCase() | log
```

To create this stream enter the following command in the shell

```
xd:> stream deploy --definition "http | transform --expression=payload.toUpperCase() |
 log" --name myprocstrem
```

Posting some data (using shell command)

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

Will result in an uppercased *hello* in the log

```
15:18:21,345  WARN ThreadPoolTaskScheduler-1 logger.myprocstream:141 - HELLO
```

See the Processors section for more information.

5.7 DSL Syntax

In the examples above, we connected a source to a sink using the pipe symbol `|`. You can also pass parameters to the source and sink configurations. The parameter names will depend on the individual module implementations, but as an example, the `http` source module exposes a `port` setting which allows you to change the data ingestion port from the default value. To create the stream using port 8000, we would use

```
xd:> stream create --definition "http --port=8000 | log" --name myhttpstream
```

If you know a bit about Spring configuration files, you can inspect the module definition to see which properties it exposes. Alternatively, you can read more in the source and sink documentation.

5.8 Advanced Features

In the examples above, simple module definitions are used to construct each stream. However, modules may be grouped together in order to avoid duplication and/or reduce the amount of chattiness over the messaging middleware. To learn more about that feature, refer to the Composing Modules section.

If directed graphs are needed instead of the simple linear streams described above, two features are relevant. First, named channels may be used as a way to combine multiple flows upstream and/or downstream from the channel. The behavior of that channel may either be queue-based or topic-based depending on what prefix is used ("queue:myqueue" or "topic:mytopic", respectively). To learn more, refer to the Named Channels section. Second, you may need to determine the output channel of a stream based on some information that is only known at runtime. To learn about such content-based routing, refer to the Dynamic Router section.

6. Modules

6.1 Introduction

The XD runtime environment supports data ingestion by allowing users to define streams. Streams are composed of *modules* which encapsulate a unit of work into a reusable component.

Modules are categorized by type, typically representing the role or function of the module. Current XD module types include *source*, *sink*, and *processor* which indicate how they modules may be composed in a stream. Specifically, a source polls an external resource, or is triggered by an event and only provides an output. The first module in a stream is always a source. A processor performs some type of transformation or business logic and provides an input and one or more outputs. A sink provides only an input and outputs data to an external resource to terminate the stream.

XD comes with a number of modules used for assembling streams which perform common input and/or output operations with files, HDFS, http, twitter, syslog, GemFire, and more. Users can easily assemble these into streams to build complex big data applications without having to know the underlying Spring products on which XD is built.

However, if you are interested in extending XD with your own modules, some knowledge of Spring, Spring Integration, and Spring Batch is essential. The remainder of this document assumes the reader has some familiarity with these topics.

6.2 Creating a Module

This section provides details on how to write and register custom modules. For a quick start, dive into the examples of creating source, processor, and sink modules.

A Module has the following required attributes:

- name - the name of the component, normally a single word representing the purpose of the module. Examples are *file*, *http*, *syslog*.
- type - the module type, current XD module types include *source*, *sink*, and *processor*
- instance id - This represents a named instance of a module with a given name and type, with a specific configuration.

Modules and Spring

At the core, a module is any component that may be implemented using a Spring application context. In this respect, the concept may be extended for purposes other than data ingestion. The types mentioned above (source, processor,sink) are specific to XD and constructing streams. But other module types are envisioned.

A module is typically configured using property placeholders which are bound to the module's attributes. Attributes may be required or optional and this coincides with whether a default value is provided for the placeholder.

For example, here is part of the Spring configuration for a *twittersearch* source that runs a query against Twitter:

```
<beans>
  ...
  <int:inbound-channel-adapter id="results" auto-startup="false"
 ref="twitterSearchMessageSource" method="getTweets">
 <int:poller fixed-delay="${fixedDelay:5000}"/>
  </int:inbound-channel-adapter>

  <bean id="twitterSearchMessageSource" class="org.springframework.integration.x.twitter.TwitterSearchMessag
 <constructor-arg ref="oauth2Template"/>
 <constructor-arg value="${query}"/>
  </bean>

  <bean id="oauth2Template" class="org.springframework.social.oauth2.OAuth2Template">
 <constructor-arg index="0" value="${consumerKey:${twitter.oauth.consumerKey}}" />
 <constructor-arg index="1" value="${consumerSecret:${twitter.oauth.consumerSecret}}" />
 <constructor-arg index="2" value="http://notused" />
 <constructor-arg index="3" value="http://notused" />
 <constructor-arg index="4" value="https://api.twitter.com/oauth2/token" />
  </bean>
</beans>
```

Note the property placeholders for *query*, *fixedDelay*, *consumerKey* and *consumerSecret*. The *query* property defines no default value, so it is a required attribute for this module. *fixedDelay* defaults to 5000, so it is an optional attribute. Note the defaults for *consumerKey* and *consumerSecret*. The property names prefixed by *twitter* are globally defined for the entire XD system in *config/twitter.properties*. So if the user does not specify a *consumerKey* or *consumerSecret* when creating the stream, XD's twitter configuration will be used instead.

The XD server will substitute values for all of these properties as configured for each module instance. For example, we can create two streams each creating an instance of the *twittersearch* module with a different configuration.

```
xd:> stream create --name tweettest --definition "twittersearch --query='java' | file"
```

or

```
xd:> stream create --name tweettest2 --definition "twittersearch --query-'java' --
consumerKey='mykey' --consumerSecret='mysecret' | file"
```

In addition to properties, modules may reference Spring beans which are defined externally such that each module instance may inject a different implementation of a bean. The ability to configure each module instance differently is only possible if each module is created in its own application context. The module may be configured with a parent context, but this should be done with care. In the simplest case, the module context is completely separate. This results in some very useful features, such as being able to create multiple bean instances with the same id, possibly with different configurations. More generally, this allows modules to adhere to the KISS principle.

Integration Modules

In Spring Integration terms,

- A *source* is a valid message flow that contains a direct channel named *output* which is fed by an inbound adapter, either configured with a poller, or triggered by an event.

- A *processor* is a valid message flow that contains a direct channel named *input* and a subscribable channel named *output* (direct or publish subscribe). It should perform some type of transformation on the message. (TBD: Describe multiple outputs, routing, etc.)

- A *sink* is a valid message flow that contains a direct channel named *input* and an outbound adapter, or service activator used to consume a message payload.

Modules of type source, processor, and sink are built with Spring Integration and are typically very fine-grained.

For example, take a look at the file source which simply polls a directory using a file inbound adapter and file sink which appends incoming message payloads to a file using a file outbound adapter. On the surface, there is nothing special about these components. They are plain old Spring XML bean definition files.

Upon closer inspection, you will notice that modules adhere to some important conventions. For one thing, the file name is the module name. Also note the channels named *input* and *output*, in keeping with the KISS principle (let us know if you come up with some simpler names). These names are by convention what XD uses to discover a module's input and/or output channels which it wires together to compose streams. Another thing you will observe is the use of property placeholders with sensible defaults where possible. For example, the file source requires a directory. An appropriate strategy is to define a common root path for XD input files (At the time of this writing it is /tmp/xd/input/. This is subject to change, but illustrates the point). An instance of this module may specify the directory by providing *name* property. If not provided, it will default to the stream name, which is contained in the *xd.stream.name* property defined by the XD runtime. By convention, XD defined properties are prefixed with *xd*

```
directory="/tmp/xd/input/${name:${xd.stream.name}}"
```

6.3 Registering a Module

XD provides a strategy interface ModuleRegistry which it uses to find a module of a given name and type. Currently XD provides RedisModuleRegistry and FileModuleRegistry, The ModuleRegistry is a required component for the XD Server. By default the XD Server is configured with the FileModuleRegistry which looks for modules in `${xd.home:..}/modules`. Where `xd.home` is a Java System Property or may be passed as a command line argument to the container launcher. So out of the box, the modules are contained in the XD modules directory. The modules directory organizes module types in sub-directories. So you will see something like:

```
modules/processor
modules/sink
modules/source
```

Using the default server configuration, you simply drop your module file into the modules directory and deploy a stream to the server.

Modules with isolated classpath

In addition to the simple format described above, where you would have a `foo` source module implemented as a `modules/source/foo.xml` file, there is also preliminary support for modules that wish to bring their own library dependencies, in an isolated fashion.

This is accomplished by creating a *folder* named after your module name and moving the xml file to a `config` subdirectory. As an example, the `foo.xml` file would then reside in

```
modules/source/foo/config/foo.xml
```

Additional jar files can then be added to a sibling `lib` directory, like so:

```
modules/source/foo/
                   config/
                          foo.xml
                   lib/
                       commons-foo.jar
                       foo-ext.jar
```

Classes will first be loaded from any of the aforementioned jar files and, only if they're not found will they be loaded from the parent, global ClassLoader that Spring XD normally uses. Still, there are a couple of caveats that one should be aware of:

1. refrain from putting into the `lib/` folder jar files that are also part of Spring XD, or you'll likely end up with ClassCastExceptions

2. any class that is directly or indirectly referenced from the payload type of your messages (*i.e.* the types that transit from module to module) must not belong to a particular module `lib/` folder but should rather be loaded by the global Spring XD classloader

6.4 Composing Modules

As described above, a stream is defined as a sequence of modules, minimally a source module followed by a sink module. One or more processor modules may be added in between the source and sink, but they are not mandatory. Sometimes streams are similar for a subset of their modules. For example, consider the following two streams:

```
stream1 = http | filter --expression=payload.contains('foo') | file
stream2 = file | filter --expression=payload.contains('foo') | file
```

Other than the source module, the definitions of those two streams are the same. It would be better to avoid this degree of duplication. This is the first problem that composed modules address.

Each module within a stream represents a unit of deployment. Therefore, in each of the streams defined above, there would be 3 such units (the source, the processor, and the sink). In a singlenode runtime, it doesn't make much of a difference since the communication between each module would be a bridge between in-memory channels. When deploying a stream to a distributed runtime environment, however, the communication between each module occurs over messaging middleware. That decoupling between modules is useful in that it promotes loose-coupling and thus enables load-balancing and buffering of messages when the consuming module(s) are temporarily busy or down. Nevertheless, at times the individual module boundaries are more fine-grained than necessary for these middleware "hops". Overhead may be avoided by reducing the overall number of deployment units and therefore the number of hops. In such cases, it's convenient to be able to wrap multiple modules together so that they act as a single "black box" unit for deployment. In other words, if "foo | bar" are composed together as a new module named "baz", the input and/or output to "baz" would still occur as a hop over the middleware, but the communication from foo to bar would occur directly, in-process. This is the second problem that composed modules address.

Now let's look at an example. Returning to the two similar streams above, the filter processor and file sink could be combined into a single module. In the shell, the following command would take care of that:

```
xd:> module compose foo --definition "filter --expression=payload.contains('foo') | file"
```

Then, to verify the new module composition was successful, check if it exists:

```
xd:> module list --type sink
Module Name          Module Type
-------------------  -----------
...
foo                  sink
```

Notice that the composed module shows up in the list of **sink** modules. That is because logically, it has the structure of a sink: it provides an input channel (which is bridged to the filter processor's input channel), but it provides no output channel (since the file sink has no output).

If a module were composed of two processors, it would be classified as a processor itself:

```
xd:> module compose myprocessor --definition "splitter | filter --
expression=payload.contains('foo')"
```

If a module were composed of a source and a processor, it would be classified as a source itself:

```
xd:> module compose mysource --definition "http | filter --
expression=payload.contains('foo')"
```

Based on the logical type of the composed module, it may be used in a stream as if it were a simple module instance. For example, to redefine the two streams from the first problem case above, now that the "foo" sink module has been composed, you would issue the following shell commands:

```
xd:> stream create httpfoo --definition "http | foo"
xd:> stream create filefoo --definition "file --outputType=text/plain | foo"
```

To test the "httpfoo" stream, try the following:

```
xd:> http post --data hi
xd:> http post --data hifoo
```

The first message should have been ignored due to the filter, but the second one should exist in the file:

```
xd:> ! cat /tmp/xd/output/httpfoo.out
command is:cat /tmp/xd/output/httpfoo.out
hifoo
```

To test the "filefoo" stream, echo "foo" to a file in the /tmp/xd/input/filefoo directory, then verify:

```
xd:> ! cat /tmp/xd/output/filefoo.out
command is:cat /tmp/xd/output/filefoo.out
foo
```

When you no longer need a composed module, you may delete it with the "module delete" command in the shell. However, if that composed module is currently being used by one or more streams, the deletion will fail as shown below:

```
xd:> module delete --name foo --type sink
16:51:37,349  WARN Spring Shell client.RestTemplate:566 - DELETE request for "http://
localhost:9393/modules/sink/foo" resulted in 500 (Internal Server Error); invoking error
 handler
Command failed org.springframework.xd.rest.client.impl.SpringXDException: Cannot delete
 module sink:foo because it is used by [stream:filefoo, stream:httpfoo]
```

As you can see, the failure message shows which stream(s) depend upon the composed module you are trying to delete.

If you destroy both of those streams and try again, it will work:

```
xd:> stream destroy --name filefoo
Destroyed stream 'filefoo'
xd:> stream destroy --name httpfoo
Destroyed stream 'httpfoo'
xd:> module delete --name foo --type sink
Successfully destroyed module 'foo' with type sink
```

Finally, it's worth mentioning that in some cases duplication may be avoided by reusing an actual stream rather than a composed module. That is possible when named channels are used in the source and/or sink position of a stream definition. For example, the same overall functionality as provided by the two streams above could also be achieved as follows:

```
xd:> stream create foofilteredfile --definition "queue:foo > filter --
expression=payload.contains('foo') | file"
xd:> stream create httpfoo --definition "http > queue:foo"
xd:> stream create filefoo --definition "file > queue:foo"
```

This approach is more appropriate for use-cases where individual streams on either side of the named channel may need to be deployed or undeployed independently. Whereas the queue typed channel will load-balance across multiple downstream consumers, the "topic:" prefix may be used if broadcast behavior is needed instead. For more information about named channels, refer to the Named Channels section.

7. Sources

7.1 Introduction

In this section we will show some variations on input sources. As a prerequisite start the XD Container as instructed in the Getting Started page.

The Sources covered are

- HTTP
- Tail
- File
- Mail
- Twitter Search
- Twitter Stream
- Gemfire
- Gemfire CQ
- Syslog
- TCP
- TCP Client
- JMS
- RabbitMQ
- Time
- MQTT

Future releases will provide support for other currently available Spring Integration Adapters. For information on how to adapt an existing Spring Integration Adapter for use in Spring XD see the section Creating a Source Module.

The following sections show a mix of Spring XD shell and plain Unix shell commands, so if you are trying them out, you should open two separate terminal prompts, one running the XD shell and one to enter the standard commands for sending HTTP data, creating directories, reading files and so on.

7.2 HTTP

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

```
xd:> stream create --name httptest --definition "http | file"
```

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

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

See if the data ended up in the file

```
$ cat /tmp/xd/output/httptest
```

HTTP with options

The http source has one option

port
: The http port where data will be posted **(default: `9000`)**

Here is an example

```
xd:> stream create --name httptest9020 --definition "http --port=9020 | file"
```

Post some data to the new port

```
xd:> http post --target http://localhost:9020 --data "hello world"
```

```
$ cat /tmp/xd/output/httptest9020
```

7.3 Tail

Make sure the default input directory exists

```
$ mkdir -p /tmp/xd/input
```

Create an empty file to tail (this is not needed on some platforms such as Linux)

```
touch /tmp/xd/input/tailtest
```

To create a stream definition using the XD shell

```
xd:> stream create --name tailtest --definition "tail | file"
```

Send some text into the file being monitored

```
$ echo blah >> /tmp/xd/input/tailtest
```

See if the data ended up in the file

```
$ cat /tmp/xd/output/tailtest
```

Tail with options

The tail source has 3 options:

name
: the absolute path to the file to tail **(default: `/tmp/xd/input/<streamName>`)**

lines
: the number of lines from the end of an existing file to tail **(default: `0`)**

fixedDelay
: on platforms that don't wait for a missing file to appear, how often (ms) to look for the file **(default: `5000`)**

Here is an example

```
xd:> stream create --name tailtest --definition "tail --name=/tmp/foo | file --name=bar"
```

```
$ echo blah >> /tmp/foo
```

```
$ cat /tmp/xd/output/bar
```

Tail Status Events

Some platforms, such as linux, send status messages to `stderr`. The tail module sends these events to a logging adapter, at WARN level; for example…

```
[message=tail: cannot open `/tmp/xd/input/tailtest' for reading: No such file or
 directory, file=/tmp/xd/input/tailtest]
[message=tail: `/tmp/xd/input/tailtest' has become accessible, file=/tmp/xd/input/
tailtest]
```

7.4 File

The file source provides the contents of a File as a byte array by default but may be configured to provide the file reference itself.

To log the contents of a file create a stream definition using the XD shell

```
xd:> stream create --name filetest --definition "file | log"
```

The file source by default will look into a directory named after the stream, in this case /tmp/xd/input/filetest

Note the above will log the raw bytes. For text files, it is normally desirable to output the contents as plain text. To do this, set the *outputType* parameter:

```
xd:> stream create --name filetest --definition "file --outputType=text/plain | log"
```

For more details on the use of the *outputType* parameter see Type Conversion

Copy a file into the directory `/tmp/xd/input/filetest` and observe its contents being logged in the XD Container.

File with options

The file source has 5 options

dir
: The absolute path to the directory to monitor for files **(default: `/tmp/xd/input/<streamName>`)**

preventDuplicates
: Default value is `true` to prevent the same file from being processed twice.

pattern
: A filter expression (Ant style) that accepts only files that match the pattern.

fixedDelay
: The fixed delay polling interval specified in seconds **(default: 5)**

ref
: Set to true to output the File object itself. This is useful in some cases in which the file contents are large and it would be more efficient to send the file path across the network than the contents. This option requires that the file be in a shared file system.

7.5 Mail sources

Spring XD provides two modules for receiving emails. Both have very similar options so they'll be described together here. The first one is named `imap` and only supports the imap protocol, using the `IDLE` command. As such, it does not use polling. Instead messages are pushed as soon as they arrive. The other module is named `mail` and supports all protocols (pop & imap), but it uses polling.

Let's see an example:

```
xd:> stream create --name mailstream --definition "mail --host=imap.gmail.com --
username=your.user@gmail.com --password=secret | file"
```

Then send an email to yourself and you should see it appear inside a file at `/tmp/xd/output/mailstream`

The full list of options for the `mail` and `imap` sources is below (most of them can be configured once and for all in the `mail.properties` file):

protocol
: the protocol to use amongst pop3, pop3s, imap, imaps (only imap variants for the `imap` module). **(default: `imaps`)**

username
: the username to use to connect to the mail server **(no default)**

password
: the password to use to connect to the mail server **(no default)**

host
: the hostname of the mail server **(default: `localhost`)**

port
: the port of the mail server **(default: none, use the default port according to the protocol used)**

folder
: the folder to take emails from **(default: `INBOX`)**

markAsRead
: whether to mark emails as read once they've been fetched by the module **(default: `false`)**

delete
: whether to delete the emails once they've been fetched by the module **(default: `true`)**

fixedDelay
: **Does not apply to the `imap` source**, the polling interval used for looking up messages, expressed in seconds. **(default: `60`)**

charset
: the charset used to transform the body of the incoming emails to Strings. **(default: `UTF-8`)**

Warning

Of special attention are the `markAsRead` and `delete` options, which by default will **delete** the emails once they are consumed. It is hard to come up with a sensible default option for this (please refer to the Spring Integration documentation section on mail handling for a discussion about this), so just be aware that the default for XD is to delete incoming messages.

7.6 Twitter Search

The twittersearch source has four parameters

query
: The query that will be run against Twitter **(required)**

consumerKey
: An application consumer key issued by twitter

consumerSecret
: The secret corresponding to the `consumerKey`

fixedDelay
: The fixed delay polling interval specified in miliseconds **(default: 5000)**

To get a `consumerKey` and `consumerSecret` you need to register a twitter application. If you don't already have one set up, you can create an app at the Twitter Developers site to get these credentials.

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

```
xd:> stream create --name springone2gx --definition "twittersearch --
consumerKey=<your_key> --consumerSecret=<your_secret> --query='#springone2gx' | file"
```

Make sure the default output directory for the `file` sink exists

```
$ mkdir -p /tmp/xd/output/
```

Let the twittersearch run for a little while and then check to see if some data ended up in the file

```
$ cat /tmp/xd/output/springone2gx
```

Tip

For both `twittersearch` and `twitterstream` you can fill in in the `conf/twitter.properties` file instead of using the DSL parameters to supply keys and secrets.

7.7 Twitter Stream

This source ingests data from Twitter's streaming API. 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 to the `conf/twitter.properties` file. Stream creation is then straightforward:

```
xd:> stream create --name tweets --definition "twitterstream | file"
```

The parameters available are pretty much the same as those listed in the API docs and unless otherwise stated, the accepted formats are the same.

delimited
: set to `true` to get length delimiters in the stream data (defaults to `false`)

stallWarnings
: set to `true` to enable stall warnings (defaults to `false`)

filterLevel, language, follow, track, locations

7.8 GemFire

This source configures a cache and replicated region in the XD container process along with a Spring Integration GemFire inbound channel adapter, backed by a CacheListener that outputs messages triggered by an external entry event on the region. By default the payload contains the updated entry value, but may be controlled by passing in a SpEL expression that uses the EntryEvent as the evaluation context.

Options

The Gemfire CacheListener source has the following options

regionName
: The name of the region for which events are to be monitored **(required, String)**

cacheEventExpression
: An optional SpEL expression referencing the event. **(default: `newValue`)**

Example

Use of the gemfire source requires an external process that creates or updates entries in a GemFire region. Such events may trigger an XD process. For example, suppose a sales application creating and updating orders in a replicated GemFire region named `orders` . To trigger an XD stream, the XD container must join the GemFire distributed system and create a replica of the region, to which a cache listener is bound via the GemFire inbound channel adapter.

```
xd:>stream create --name orderStream --definition "gemfire --regionName=orders | file --
inputType=application/json"
```

In the above example, it is presumed the cache entries are Order POJOs. In this case, it may be convenient to convert to JSON before writing to the file.

7.9 GemFire Continuous Query (CQ)

Continuous query allows client applications to create a GemFire query using Object Query Language(OQL) and register a CQ listener which subscribes to the query and is notified every time the query 's result set changes. The *gemfire_cq* source registers a CQ which will post CQEvent messages to the stream.

Launching the XD GemFire Server

This source requires a cache server to be running in a separate process and its host and port must be known (NOTE: GemFire locators are not supported yet). The XD distribution includes a GemFire server executable suitable for development and test purposes. This is a Java main class that runs with a Spring configured cache server. The configuration is passed as a command line argument to the server's main method. The configuration includes a cache server port and one or more configured region. XD includes a sample cache configuration called cq-demo. This starts a server on port 40404 and creates a region named *Stocks*. A Logging cache listener is configured for the region to log region events.

Run Gemfire cache server by changing to the gemfire/bin directory and execute

```
$ ./gemfire-server ../config/cq-demo.xml
```

Options

The qemfire-cq source has the following options

query
: The query string in Object Query Language(OQL) **(required, String)**

gemfireHost
: The host on which the GemFire server is running. **(default: `localhost`)**

gemfirePort
: The port on which the GemFire server is running. **(default: `40404`)**

Here is an example. Create two streams: One to write http messages to a Gemfire region named *Stocks*, and another to execute the CQ.

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

Now send some messages to the stocks stream.

```
xd:> http post --target http://localhost:9090 --data "{"symbol":"VMW","price":73}"
xd:> http post --target http://localhost:9090 --data "{"symbol":"VMW","price":78}"
xd:> http post --target http://localhost:9090 --data "{"symbol":"VMW","price":80}"
```

Please do not put spaces when separating the JSON key-value pairs, only a comma.

The *cqtest* stream is now listening for any stock quote updates for VMW. Presumably, another process is updating the cache. You may create a separate stream to test this (see GemfireServer for instructions).

As updates are posted to the cache you should see them captured in the output file:

```
$cat /tmp/xd/output/cqtest
```

```
{"symbol":"VMW","price":73}
{"symbol":"VMW","price":78}
{"symbol":"VMW","price":80}
```

7.10 Syslog

Two syslog sources are provided: `syslog-udp` and `syslog-tcp`. They both support the following options:

port
: the port on which the system will listen for syslog messages **(default: 11111)**

To create a stream definition (using shell command)

```
xd:> stream create --name syslogtest --definition "syslog-udp --port=1514 | file"
```

or

```
xd:> stream create --name syslogtest --definition "syslog-tcp --port=1514 | file"
```

Send a test message to the syslog

```
logger -p local3.info -t TESTING "Test Syslog Message"
```

See if the data ended up in the file

```
$ cat /tmp/xd/output/syslogtest
```

Refer to your syslog documentation to configure the syslog daemon to forward syslog messages to the stream; some examples are:

UDP - Mac OSX (syslog.conf) and Ubuntu (rsyslog.conf)

```
*.*    @localhost:11111
```

TCP - Ubuntu (rsyslog.conf)

```
$ModLoad omfwd
*.*    @@localhost:11111
```

Restart the syslog daemon after reconfiguring.

7.11 TCP

The `tcp` source acts as a server and allows a remote party to connect to XD and submit data over a raw tcp socket.

To create a stream definition in the server, use the following XD shell command

```
xd:> stream create --name tcptest --definition "tcp | file"
```

This will create the default TCP source and send data read from it to the `tcptest` file.

TCP is a streaming protocol and some mechanism is needed to frame messages on the wire. A number of decoders are available, the default being *CRLF* which is compatible with Telnet.

```
$ telnet localhost 1234
Trying ::1...
Connected to localhost.
Escape character is '^]'.
foo
^]

telnet> quit
Connection closed.
```

See if the data ended up in the file

```
$ cat /tmp/xd/output/tcptest
```

TCP with options

The TCP source has the following options

port
: the port on which to listen **(default: `1234`)**

reverse-lookup
: perform a reverse DNS lookup on the remote IP Address **(default: `false`)**

socket-timeout
: the timeout (ms) before closing the socket when no data received **(default: `120000`)**

nio
: whether or not to use NIO. NIO is more efficient when there are many connections. **(default: `false`)**

decoder
: how to decode the stream - see below. **(default: `CRLF`)**

binary
: whether the data is binary (true) or text (false). **(default: `false`)**

charset
: the charset used when converting text to `String`. **(default: `UTF-8`)**

Available Decoders

Text Data

CRLF (default)
: text terminated by carriage return (0x0d) followed by line feed (0x0a)

LF
: text terminated by line feed (0x0a)

NULL
: text terminated by a null byte (0x00)

STXETX
: text preceded by an STX (0x02) and terminated by an ETX (0x03)

Text and Binary Data

RAW
: no structure - the client indicates a complete message by closing the socket

L1
: data preceded by a one byte (unsigned) length field (supports up to 255 bytes)

L2
: data preceded by a two byte (unsigned) length field (up to 2^{16}-1 bytes)

L4
: data preceded by a four byte (signed) length field (up to 2^{31}-1 bytes)

Examples

The following examples all use `echo` to send data to `netcat` which sends the data to the source.

The echo options `-en` allows echo to interpret escape sequences and not send a newline.

CRLF Decoder.

```
xd:> stream create --name tcptest --definition "tcp | file"
```

This uses the default (CRLF) decoder and port 1234; send some data

```
$ echo -en 'foobar\r\n' | netcat localhost 1234
```

See if the data ended up in the file

```
$ cat /tmp/xd/output/tcptest
```

LF Decoder.

```
xd:> stream create --name tcptest2 --definition "tcp --decoder=LF --port=1235 | file"
```

```
$ echo -en 'foobar\n' | netcat localhost 1235
```

```
$ cat /tmp/xd/output/tcptest2
```

NULL Decoder.

```
xd:> stream create --name tcptest3 --definition "tcp --decoder=NULL --port=1236 | file"
```

```
$ echo -en 'foobar\x00' | netcat localhost 1236
```

```
$ cat /tmp/xd/output/tcptest3
```

STXETX Decoder.

```
xd:> stream create --name tcptest4 --definition "tcp --decoder=STXETX --port=1237 | file"
```

```
$ echo -en '\x02foobar\x03' | netcat localhost 1237
```

```
$ cat /tmp/xd/output/tcptest4
```

RAW Decoder.

```
xd:> stream create --name tcptest5 --definition "tcp --decoder=RAW --port=1238 | file"
```

```
$ echo -n 'foobar' | netcat localhost 1238
```

```
$ cat /tmp/xd/output/tcptest5
```

L1 Decoder.

```
xd:> stream create --name tcptest6 --definition "tcp --decoder=L1 --port=1239 | file"
```

```
$ echo -en '\x06foobar' | netcat localhost 1239
```

```
$ cat /tmp/xd/output/tcptest6
```

L2 Decoder.

```
xd:> stream create --name tcptest7 --definition "tcp --decoder=L2 --port=1240 | file"
```

```
$ echo -en '\x00\x06foobar' | netcat localhost 1240
```

```
$ cat /tmp/xd/output/tcptest7
```

L4 Decoder.

```
xd:> stream create --name tcptest8 --definition "tcp --decoder=L4 --port=1241 | file"
```

```
$ echo -en '\x00\x00\x00\x06foobar' | netcat localhost 1241
```

```
$ cat /tmp/xd/output/tcptest8
```

Binary Data Example

```
xd:> stream create --name tcptest9 --definition "tcp --decoder=L1 --port=1242 | file --
binary=true"
```

Note that we configure the `file` sink with `binary=true` so that a newline is not appended.

```
$ echo -en '\x08foo\x00bar\x0b' | netcat localhost 1242
```

```
$ hexdump -C /tmp/xd/output/tcptest9
00000000  66 6f 6f 00 62 61 72 0b                           |foo.bar.|
00000008
```

7.12 TCP Client

The `tcp-client` source module uses raw tcp sockets, as does the `tcp` module but contrary to the `tcp` module, acts as a client. Whereas the `tcp` module will open a listening socket and wait for connections from a remote party, the `tcp-client` will initiate the connection to a remote server and emit as

messages what that remote server sends over the wire. As an optional feature, the `tcp-client` can itself emit messages to the remote server, so that a simple conversation can take place.

TCP Client options

The following options are supported:

host
: the host to connect to **(default: `localhost`)**

port
: the port to connect to **(default: `1234`)**

reverse-lookup
: whether to attempt to resolve the host address **(default: `false`)**

nio
: whether to use NIO **(default: `false`)**

encoder
: the encoder to use when sending messages **(default: `LF`, see TCP module)**

decoder
: the decoder to use when receiving messages **(default: `LF`, see TCP module)**

charset
: the charset to use when converting bytes to String **(default: `UTF-8`)**

bufferSize
: the size of the emitting/receiving buffers **(default: `2048`, i.e. 2KB)**

fixedDelay
: the rate at which *stimulus* messages will be emitted **(default: 5 seconds)**

script
: reference to a script that should transform the counter stimulus to messages to send **(default: use `expression`)**

expression
: a SpEL expression to convert the counter stimulus to a message **(default: `payload.toString()`, i.e. emit "1", "2", "3", etc.)**

Implementing a simple conversation

That "stimulus" counter concept bears some explanation. By default, the module will emit (at interval set by `fixedDelay`) an incrementing number, starting at `1`. Given that the default is to use an `expression` of `payload.toString()`, this results in the module sending `1, 2, 3, ...` to the remote server.

By using another expression, or more certainly a `script`, one can implement a simple conversation, assuming it is time based. As an example, let's assume we want to join some kind of chat server where one first needs to authenticate, then specify which rooms to join. Lastly, all clients are supposed to send some keepalive commands to make sure that the connection is open.

The following groovy script could be used to that effect:

```
def commands = ['', // index 0 is not used
'LOGIN user=johndoe', // first command sent
'JOIN weather',
'JOIN news',
'JOIN gossip'
]


// payload will contain an incrementing counter, starting at 1
if (commands.size > payload)
  return commands[payload] + "\n"
else
  return "PING\n"  // send keep alive after 4th 'real' command
```

7.13 RabbitMQ

The "rabbit" source enables receiving messages from RabbitMQ.

The following example shows the default settings.

Configure a stream:

```
xd:> stream create --name rabbittest --definition "rabbit | file --binary=true"
```

This receives messages from a queue named `rabbittest` and writes them to the default file sink (`/tmp/xd/output/rabbittest.out`). It uses the default RabbitMQ broker running on localhost, port 5672.

The queue(s) must exist before the stream is deployed. We do not create the queue(s) automatically. However, you can easily create a Queue using the RabbitMQ web UI. Then, using that same UI, you can navigate to the "rabbittest" Queue and publish test messages to it.

Notice that the `file` sink has `--binary=true`; this is because, by default, the data emitted by the source will be bytes. This can be modified by setting the `content_type` property on messages to `text/plain`. In that case, the source will convert the message to a `String`; you can then omit the `--binary=true` and the file sink will then append a newline after each message.

To destroy the stream, enter the following at the shell prompt:

```
xd:> stream destroy --name rabbittest
```

RabbitMQ with Options

The RabbitMQ Source has the following options

host
: the host (or IP Address) to connect to **(default: `localhost` unless `rabbit.hostname` has been overridden in `rabbit.properties`)**

port
: the port on the `host` **(default: `5672` unless `rabbit.port` has been overridden in `rabbit.properties`)**

vhost
: the virtual host **(default: `/` unless `rabbit.vhost` has been overridden in `rabbit.properties`)**

queues
: the queue(s) from which messages will be received; use a comma-delimited list to receive messages from multiple queues **(default: the stream name)**

Note: the `rabbit.properties` file referred to above is located within the `XD_HOME/config` directory.

7.14 JMS

The "jms" source enables receiving messages from JMS.

The following example shows the default settings.

Configure a stream:

```
xd:> stream create --name jmstest --definition "jms | file"
```

This receives messages from a queue named `jmstest` and writes them to the default file sink (`/tmp/xd/output/jmstest`). It uses the default ActiveMQ broker running on localhost, port 61616.

To destroy the stream, enter the following at the shell prompt:

```
xd:> stream destroy --name jmstest
```

To test the above stream, you can use something like the following…

```
public class Broker {

 public static void main(String[] args) throws Exception {
  BrokerService broker = new BrokerService();
  broker.setBrokerName("broker");
  String brokerURL = "tcp://localhost:61616";
  broker.addConnector(brokerURL);
  broker.start();
  ConnectionFactory cf = new ActiveMQConnectionFactory(brokerURL);
  JmsTemplate template = new JmsTemplate(cf);
  while (System.in.read() >= 0) {
   template.convertAndSend("jmstest", "testFoo");
  }
 }
}
```

and `tail -f /tmp/xd/output/jmstest`

Run this as a Java application; each time you hit <enter> in the console, it will send a message to queue `jmstest`.

JMS with Options

The JMS Source has the following options

provider
: the JMS provider **(default: `activemq`)**

queue
: the queue from which messages will be received; use a comma-delimited list to receive messages from multiple queues

Note: the selected broker requires an infrastructure configuration file `jms-<provider>-infrastructure-context.xml` in `modules/common`. This is used to declare any infrastructure beans needed by the provider. See the default (`jms-activemq-infrastructure-context.xml`) for an example. Typically, all that is required is a `ConnectionFactory`. The activemq provider uses a properties file `jms-activemq.properties` which can be found in the `config` directory. This contains the broker URL.

7.15 Time

The time source will simply emit a String with the current time every so often. It supports the following options:

fixedDelay
: how often to emit a message, expressed in seconds **(default: 1 second)**

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

7.16 MQTT

The mqtt source connects to an mqtt server and receives telemetry messages.

Options

The folllowing options are configured in mqtt.properties in XD_HOME/config

```
mqtt.url=tcp://localhost:1883
mqtt.default.client.id=xd.mqtt.client.id
mqtt.username=guest
mqtt.password=guest
mqtt.default.topic=xd.mqtt.test
```

The defaults are set up to connect to the RabbitMQ MQTT adapter on localhost.

Note that the client id must be no more than 19 characters; this is because `.src` is added and the id must be no more than 23 characters.

clientId
: Identifies the client - overrides the default above.

topics
: The topics to which the source will subscribe - overrides the default above.

8. Processors

8.1 Introduction

This section will cover the processors available out-of-the-box with Spring XD. As a prerequisite, start the XD Container as instructed in the Getting Started page.

The Processors covered are

- Filter
- JSON Field Value Filter
- Transform
- JSON Field Extractor
- Script
- Splitter
- Aggregator

See the section Creating a Processor Module for information on how to create custom processor modules.

8.2 Filter

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

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:

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

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:

```
xd:> 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.

Filter with Groovy Script

For more complex filtering, you can pass the location of a Groovy script using the *script* attribute. If you want to pass variable values to your script, you can optionally pass the path to a properties file using the *properties-location* attribute. All properties in the file will be made available to the script as variables.

```
xd:> stream create --name groovyfiltertest --definition "http --port=9001 | filter --
script=custom-filter.groovy --properties-location=custom-filter.properties | log"
```

By default, Spring XD will search the classpath for *custom-filter.groovy* and *custom-filter.properties*. You can place the script in *${xd.home}/modules/processor/scripts* and the properties file in *${xd.home}/config* to make them available on the classpath. Alternatively, you can prefix the *script* and *properties-location* values with *file:* to load from the file system.

8.3 JSON Field Value Filter

Use this filter to only pass messages to the output channel if they contain a specific JSON field matching a specific value.

```
xd:> stream create --name jsonfiltertest --definition "http --port=9002 | json-field-
value-filter --fieldName=firstName --fieldValue=John | log"
```

This filter will only pass Messages to the log sink if the JSON payload contains the *firstName* "John". Try sending this payload to the HTTP endpoint and you should see it in the XD log:

```
xd:> http post --target http://localhost:9002 --data "{\"firstName\":\"John\", \"lastName
\":\"Smith\"}"
```

Alternatively, if you send a different *firstName*, you shouldn't see the log entry.

8.4 Transform

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

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:

```
xd:> stream create --name transformtest --definition "http --port=9003 | transform --
expression='FOO' | log"
```

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

```
xd:> http post --target http://localhost:9003 --data "some message"
```

Transform with Groovy Script

For more complex transformations, you can pass the location of a Groovy script using the *script* attribute. If you want to pass variable values to your script, you can optionally pass the path to a properties file using the *properties-location* attribute. All properties in the file will be made available to the script as variables.

```
xd:> stream create --name groovytransformtest --definition "http --port=9004 | transform
 --script=custom-transform.groovy --properties-location=custom-transform.properties | log"
```

By default, Spring XD will search the classpath for *custom-transform.groovy* and *custom-transform.properties*. You can place the script in *${xd.home}/modules/processor/scripts* and the properties file in *${xd.home}/config* to make them available on the classpath. Alternatively, you can prefix the *script* and *properties-location* values with *file:* to load from the file system.

8.5 JSON Field Extractor

This processor converts a JSON message payload to the value of a specific JSON field.

```
xd:> stream create --name jsontransformtest --definition "http --port=9005 | json-field-
extractor --fieldName=firstName | log"
```

Try sending this payload to the HTTP endpoint and you should see just the value "John" in the XD log:

```
xd:> http post --target http://localhost:9005 --data "{\"firstName\":\"John\", \"lastName
\":\"Smith\"}"
```

8.6 Script

The script processor contains a *Service Activator* that invokes a specified Groovy script. This is a slightly more generic way to accomplish processing logic, as the provided script may simply terminate the stream as well as transform or filter Messages.

To use the module, pass the location of a Groovy script using the *location* attribute. If you want to pass variable values to your script, you can optionally pass the path to a properties file using the *properties-location* attribute. All properties in the file will be made available to the script as variables.

```
xd:> stream create --name groovyprocessortest --definition "http --port=9006 | script --
location=custom-processor.groovy --properties-location=custom-processor.properties | log"
```

By default, Spring XD will search the classpath for *custom-processor.groovy* and *custom-processor.properties*. You can place the script in *${xd.home}/modules/processor/scripts* and the properties file in *${xd.home}/config* to make them available on the classpath. Alternatively, you can prefix the *location* and *properties-location* values with *file:* to load from the file system.

8.7 Splitter

The splitter module builds upon the concept of the same name in Spring Integration and allows the splitting of a single message into several distinct messages.

The splitter module accepts the following options:

expression
: a SpEL expression which should evaluate to an array or collection. Each element will then be emitted as a separate message **(default: `payload`, which actually does not split, unless the message is already a collection)**

8.8 Aggregator

The aggregator module does the opposite of the splitter, and builds upon the concept of the same name found in Spring Integration. By default, it will consider all incoming messages from a stream to belong to the same group:

```
xd:> stream create --name aggregates --definition "http | aggregator --count=3 --
aggregation=T(org.springframework.util.StringUtils).collectionToDelimitedString(#this.!
[payload],' ') | log"
```

This uses a SpEL expression that will basically concatenate all payloads together, inserting a space character in between. As such,

```
xd:> http post --data Hello
xd:> http post --data World
xd:> http post --data !
```

would emit a single message whose contents is "Hello World !". This is because we set the aggregator release strategy to accumulate 3 messages.

The aggregator modules comes with many more options, as shown below:

correlation
: a SpEL expression to be evaluated against all incoming message and that should evaluate to the "key" used to group messages together **(default: `<streamname>`, which means that all messages from the same stream are actually considered correlated)**

release
: a SpEL expression to be evaluated against a group of messages accumulated so far (a collection) and that should return true when such a group is ready to be released. Using this overrides the *count* option. **(default: use the `'count'` approach)**

count
: the number of messages to group together before emitting a group **(default: `50`)**

aggregation
: a SpEL expression, to be evaluated against the list of accumulated messages. This should return what the new message will be made of. **(default: `#this.![payload]`, which uses the list of message payloads to form the new message)**

timeout
: the delay (in milliseconds) after which messages should be released and aggregated, even though the completion criteria was not met. Due to the way this is implemented (see MessageGroupStoreReaper in the Spring Integration documentation), the actual observed delay may vary between `timeout` and `2xtimeout`. **(default: `60000`, i.e. one minute)**

9. Sinks

9.1 Introduction

In this section we will show some variations on output sinks. As a prerequisite start the XD Container as instructed in the Getting Started page.

The Sinks covered are

- Log
- File
- HDFS
- JDBC
- TCP
- Mail
- RabbitMQ
- GemFire Server
- Splunk Server
- MQTT
- Dynamic Router

See the section Creating a Sink Module for information on how to create sink modules using other Spring Integration Adapters.

9.2 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

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

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

```
xd:> 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 WARN logger.mylogstream: hello
```

The logger name is the sink name prefixed with the string "logger.". The sink name is the same as the stream name by default, but you can set it by passing the `--name` parameter

```
xd:> stream create --name myotherlogstream --definition "http --port=8001 | log --
name=mylogger"
```

9.3 File Sink

Another simple option is to stream data to a file on the host OS. This can be done using the `file` sink module to create a stream.

```
xd:> stream create --name myfilestream --definition "http --port=8000 | file"
```

We've used the `http` source again, so run the following command to send a message

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

The `file` sink uses the stream name as the default name for the file it creates, and places the file in the `/tmp/xd/output/` directory.

```
$ less /tmp/xd/output/myfilestream
hello
```

You can cutomize the behavior and specify the `name` and `dir` properties of the output file. For example

```
xd:> stream create --name otherfilestream --definition "http --port=8000 | file --
name=myfile --dir=/some/custom/directory"
```

File with Options

The file sink, by default, will add a newline at the end of each line; the actual newline will depend on the operating system.

This can be disabled by using `--binary=true`.

9.4 Hadoop (HDFS)

If you do not have Hadoop installed, you can install Hadoop 1.1.2 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

```
xd:> stream create --name myhdfsstream --definition "http --port=8000 | hdfs --
rollover=10"
```

Note that we've set the `rollover` parameter to a small value for this exercise. This is just to avoid buffering, so that we can actually see the data has made it into HDFS.

As in the above examples, we've used the `http` source on port 8000, so we can post some data using the shell's built int http post command

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

Which is the equivalent to using curl

```
$ curl -d "hello" http://localhost:8000
```

Repeat the command a few times.

You can then list the contents of the hadoop filesystem using the shell's built in hadoop fs commands. You will first need to configure the shell to point to your name node using the hadoop config command

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

By default the hdfs protocol is used to access hadoop. then list the contents of the root directory

```
xd:>hadoop fs ls /
Found 1 items
drwxr-xr-x   - mpollack supergroup          0 2013-07-30 02:34 /xd
```

You should see that an `xd` directory has appeared in the root with a sub-directory named after our stream. This is equivalent to using the hadoop command line utility

```
$ hadoop dfs -ls /xd
Found 1 items
drwxr-xr-x   - mpollack supergroup          0 2013-07-30 02:34 /xd
```

And there will be one or more log files in there depending how many times you ran the command to post the data

```
xd:>hadoop fs ls /xd/myhdfsstream
Found 3 items
-rw-r--r--   3 mpollack supergroup         12 2013-07-30 02:34 /xd/myhdfsstream/
myhdfsstream-0.log
-rw-r--r--   3 mpollack supergroup         12 2013-07-30 02:39 /xd/myhdfsstream/
myhdfsstream-1.log
-rw-r--r--   3 mpollack supergroup          0 2013-07-30 02:39 /xd/myhdfsstream/
myhdfsstream-2.log
```

You can examine the file contents using the shell's `hadoop fs cat` command

```
xd:>hadoop fs cat /xd/myhdfsstream/myhdfsstream-0.log
hello
hello
```

HDFS with Options

The HDFS Sink has the following options:

newline
: whether to append a newline to the message payload **(default: `true`)**

directory
: where to output the files in the Hadoop FileSystem **(default: `/xd/<streamname>`)**

filename
: the base filename to use for the created files (a counter will be appended before the file extension). **(default: `<streamname>`)**

suffix
: the file extension to use **(default: `log`)**

rollover
: when to roll files over, expressed in bytes **(default: `1000000`, roughly 1MB)**

9.5 JDBC

The JDBC sink can be used to insert message payload data into a relational database table. By default it inserts the entire payload into an in-memory HSQLDB database table named after the stream name. To alter this behavior you should modify the *config/jdbc.properties* file with the connection parameters you want to use. There is also a *config/init_db.sql* file that contains the SQL statements used to initialize the database table. You can modify this file if you'd like to create the table when the sink starts or change the *initializeDatabase* property to *false* if the table already exists.

The payload data will be inserted as-is if the *columns* option is set to *payload*. This is the default behavior. If you specify any other column names the payload data will be assumed to be a JSON document that will be converted to a hash map. This hash map will be used to populate the data values for the SQL insert statement. A matching of column names with underscores like *user_name* will match onto camel case style keys like *userName* in the hash map. There will be one insert statement executed for each message.

To create a stream using a `jdbc` sink relying on all defaults you would use a command like

```
xd:> stream create --name myjdbc --definition "time | jdbc"
```

This will insert the time messages into a *payload* column in a table named *myjdbc*. Since the default is using an in-memory HSQLDB database we can't connect to this database instance from an external tool. In order to do that we need to alter the connection properties. We can either modify the *config/jdbc.properties* file or provide the *url* property when we create the stream. Here is an example of the latter:

```
xd:> stream create --name mydata --definition "time | jdbc --url='jdbc:hsqldb:file:/tmp/
xd/test'"
```

We let the stream run for a little while and then destroy it so we can look at the data stored in the database.

```
xd:> stream destroy --name mydata
```

You can use the above database URL from your favorite SQL tool or we can use the HSQL provided SQL Tool to run a quick query from the command line:

```
$ java -cp $XD_HOME/lib/hsqldb-1.8.0.10.jar org.hsqldb.util.SqlTool --inlineRc
 url=jdbc:hsqldb:file:/tmp/xd/test,user=sa,password= --sql "select payload from mydata;"
```

This should result in something similar to the following output:

```
2013-07-29 12:05:48
2013-07-29 12:05:49
2013-07-29 12:05:50
2013-07-29 12:05:51
2013-07-29 12:05:52
2013-07-29 12:05:53
2013-07-29 12:05:54
2013-07-29 12:05:55
2013-07-29 12:05:56
2013-07-29 12:05:57

Fetched 10 rows.
```

JDBC with Options

The JDBC Sink has the following options:

configProperties
: base name of properties file (in the config directory) containing configuration options for the sink. This file should contain the usual JDBC properties - driverClass, url, username, password **(default: `jdbc`)**

initializeDatabase
: whether to initialize the database using the initializer script (the default property file jdbc.properties has this set to true) **(default: `false`)**

initializerScript
: the file name for the script containing SQL statements used to initialize the database when the sink starts (will search config directory for this file) **(default: `init_db.sql`)**

tablename
: the name of the table to insert payload data into **(default: `<streamname>`)**

columns
: comma separated list of column names to include in the insert statement. Use *payload* to include the entire message payload into a payload column. **(default: `payload`)**

9.6 TCP

The TCP Sink provides for outbound messaging over TCP.

The following examples use `netcat` (linux) to receive the data; the equivalent on Mac OSX is `nc`.

First, start a netcat to receive the data, and background it

```
$ netcat -l 1234 &
```

Now, configure a stream

```
xd:> stream create --name tcptest --definition "time --interval=3 | tcp"
```

This sends the time, every 3 seconds to the default tcp Sink, which connects to port `1234` on `localhost`.

```
$ Thu May 30 10:28:21 EDT 2013
Thu May 30 10:28:24 EDT 2013
Thu May 30 10:28:27 EDT 2013
Thu May 30 10:28:30 EDT 2013
Thu May 30 10:28:33 EDT 2013
```

TCP is a streaming protocol and some mechanism is needed to frame messages on the wire. A number of encoders are available, the default being *CRLF*.

Destroy the stream; netcat will terminate when the TCP Sink disconnects.

```
http://localhost:8080> stream destroy --name tcptest
```

TCP with Options

The TCP Sink has the following options

host
: the host (or IP Address) to connect to **(default: `localhost`)**

port
: the port on the `host` **(default `1234`)**

reverse-lookup
: perform a reverse DNS lookup on IP Addresses **(default: `false`)**

nio
: whether or not to use NIO **(default: `false`)**

encoder
: how to encode the stream - see below **(default: `CRLF`)**

close
: whether to close the socket after each message **(default: `false`)**

charset
: the charset used when converting text from `String` to bytes **(default: `UTF-8`)**

Retry Options

retry-max-attempts
: the maximum number of attempts to send the data **(default: 5 - original request and 4 retries)**

retry-initial-interval
: the time (ms) to wait for the first retry **(default: `2000`)**

retry-multiplier
: the multiplier for exponential back off of retries **(default: 2)**

With the default retry configuration, the attempts will be made after 0, 2, 4, 8, and 16 seconds.

Available Encoders

Text Data

CRLF (default)
: text terminated by carriage return (0x0d) followed by line feed (0x0a)

LF
: text terminated by line feed (0x0a)

NULL
: text terminated by a null byte (0x00)

STXETX
: text preceded by an STX (0x02) and terminated by an ETX (0x03)

Text and Binary Data

RAW
: no structure - the client indicates a complete message by closing the socket

L1
: data preceded by a one byte (unsigned) length field (supports up to 255 bytes)

L2
: data preceded by a two byte (unsigned) length field (up to 2^{16}-1 bytes)

L4
: data preceded by a four byte (signed) length field (up to 2^{31}-1 bytes)

An Additional Example

Start netcat in the background and redirect the output to a file `foo`

```
$ netcat -l 1235 > foo &
```

Create the stream, using the `L4` encoder

```
xd:> stream create --name tcptest --definition "time --interval=3 | tcp --encoder=L4 --
port=1235"
```

Destroy the stream

```
http://localhost:8080> stream destroy --name tcptest
```

Check the output

```
$ hexdump -C foo
00000000  00 00 00 1c 54 68 75 20  4d 61 79 20 33 30 20 31  |....Thu May 30 1|
00000010  30 3a 34 37 3a 30 33 20  45 44 54 20 32 30 31 33  |0:47:03 EDT 2013|
00000020  00 00 00 1c 54 68 75 20  4d 61 79 20 33 30 20 31  |....Thu May 30 1|
00000030  30 3a 34 37 3a 30 36 20  45 44 54 20 32 30 31 33  |0:47:06 EDT 2013|
00000040  00 00 00 1c 54 68 75 20  4d 61 79 20 33 30 20 31  |....Thu May 30 1|
00000050  30 3a 34 37 3a 30 39 20  45 44 54 20 32 30 31 33  |0:47:09 EDT 2013|
```

Note the 4 byte length field preceding the data generated by the `L4` encoder.

9.7 Mail

The "mail" sink allows sending of messages as emails, leveraging Spring Integration mail-sending channel adapter. Please refer to Spring Integration documentation for the details, but in a nutshell, the sink is able to handle String, byte[] and MimeMessage messages out of the box.

Here is a simple example of how the mail module is used:

```
xd:> stream create mystream --definition "http | mail --to='"your.email@gmail.com"' --
host=your.imap.server --subject=payload+' world'"
```

Then,

```
xd:> http post --data Hello
```

You would then receive an email whose body contains "Hello" and whose subject is "Hellow world". Of special attention here is the way you need to escape strings for most of the parameters, because they're actually SpEL expressions (so here for example, we used a String literal for the `to` parameter).

The full list of options available to the mail module is below (note that most of these options can be set once and for all in the `mail.properties` file):

to
: The primary recipient(s) of the email. **(default: `null`, SpEL Expression)**

from
: The sender address of the email. **(default: `null`, SpEL Expression)**

subject
: The email subject. **(default: `null`, SpEL Expression)**

cc
: The recipient(s) that should receive a carbon copy. **(default: `null`, SpEL Expression)**

bcc
: The recipient(s) that should receive a blind carbon copy. **(default: `null`, SpEL Expression)**

replyTo
: The address that will become the recipient if the original recipient decides to "reply to" the email. **(default: `null`, SpEL Expression)**

contentType
: The content type to use when sending the email. **(default: `null`, SpEL Expression)**

host
: The hostname of the sending server to use. **(default: `localhost`)**

port
: The port of the sending server. **(default: 25)**

username
: The username to use for authentication against the sending server. **(default: none)**

password
: The password to use for authentication against the sending server. **(default: none)**

9.8 RabbitMQ

The "rabbit" sink enables outbound messaging over RabbitMQ.

The following example shows the default settings.

Configure a stream:

```
xd:> stream create --name rabbittest --definition "time --interval=3 | rabbit"
```

This sends the time, every 3 seconds to the default (no-name) Exchange for a RabbitMQ broker running on localhost, port 5672.

The routing key will be the name of the stream by default; in this case: "rabbittest". Since the default Exchange is a direct-exchange to which all Queues are bound with the Queue name as the binding key, all messages sent via this sink will be passed to a Queue named "rabbittest", if one exists. We do not create that Queue automatically. However, you can easily create a Queue using the RabbitMQ web UI.

Then, using that same UI, you can navigate to the "rabbittest" Queue and click the "Get Message(s)" button to pop messages off of that Queue (you can choose whether to requeue those messages).

To destroy the stream, enter the following at the shell prompt:

```
xd:> stream destroy --name rabbittest
```

RabbitMQ with Options

The RabbitMQ Sink has the following options

host
: the host (or IP Address) to connect to **(default: `localhost` unless `rabbit.hostname` has been overridden in `rabbit.properties`)**

port
: the port on the `host` **(default: `5672` unless `rabbit.port` has been overridden in `rabbit.properties`)**

vhost
: the virtual host **(default: `/` unless `rabbit.vhost` has been overridden in `rabbit.properties`)**

exchange
: the Exchange on the RabbitMQ broker to which messages should be sent **(default: `` (empty: therefore, the default no-name Exchange))**

routingKey
: the routing key to be passed with the message **(default: <streamname>)**

Note: the `rabbit.properties` file referred to above is located within the `XD_HOME/config` directory.

9.9 GemFire Server

Currently XD supports GemFire's client-server topology. A sink that writes data to a GemFire cache requires a cache server to be running in a separate process and its host and port must be known (NOTE: GemFire locators are not supported yet). The XD distribution includes a GemFire server executable suitable for development and test purposes. It is made available under GemFire's development license and is limited to 3 nodes. Modules that write to GemFire create a client cache and client region. No data is cached on the client.

Launching the XD GemFire Server

A GemFire Server is included in the Spring XD distribution. To start the server. Go to the XD install directory:

```
$cd gemfire/bin
$./gemfire-server ../config/cq-demo.xml
```

The command line argument is the location of a Spring file with a configured cache server. A sample cache configuration is provided cq-demo.xml located in the `config` directory. This starts a server on port 40404 and creates a region named *Stocks*. A Logging cache listener is configured for the region to log region events.

Gemfire sinks

There are 2 implementation of the gemfire sink: *gemfire-server* and *gemfire-json-server*. They are identical except the latter converts JSON string payloads to a JSON document format proprietary to GemFire and provides JSON field access and query capabilities. If you are not using JSON, the gemfire-server module will write the payload using java serialization to the configured region. Either of these modules accepts the following attributes:

regionName
: the name of the GemFire region. This must be the name of a region configured for the cache server. This module creates the corresponding client region. **(default: `<streamname>`)**

keyExpression
: A SpEL expression which is evaluated to create a cache key. Typically, the key value is derived from the payload. **(default: `<streamname>`**, which will overwrite the same entry for every message received on the stream)

gemfireHost
: The host name or IP address of the cache server **(default: `localhost`)**

gemfirePort
: The TCP port number of the cache server **(default: `40404`)**

Example

Suppose we have a JSON document containing a stock price:

```
{"symbol":"VMW", "price":73}
```

We want this to be cached using the stock symbol as the key. The stream definition is:

```
http | gemfire-json-server --regionName=Stocks --keyExpression=payload.getField('symbol')
```

The keyExpression is a SpEL expression that depends on the payload type. In this case, *com.gemstone.org.json.JSONObject. JSONObject* which provides the *getField* method. To run this example:

```
xd:> stream create --name stocks --definition "http --port=9090 | gemfire-json-server --
regionName=Stocks --keyExpression=payload.getField('symbol')"
```

```
xd:> http post --target http://localhost:9090 --data "{"symbol":"VMW","price":73}"
```

This will write an entry to the GemFire *Stocks* region with the key *VMW*. Please do not put spaces when separating the JSON key-value pairs, only a comma.

You should see a message on STDOUT for the process running the GemFire server like:

```
INFO [LoggingCacheListener] - updated entry VMW
```

9.10 Splunk Server

A Splunk sink that writes data to a TCP Data Input type for Splunk.

Splunk sinks

The Splunk sink converts an object payload to a string using the object's toString method and then converts this to a SplunkEvent that is sent via TCP to Splunk. The module accepts the following attributes:

host
: The host name or IP address of the Splunk server *(default: `localhost`)

port
: The TCP port number of the Splunk Server **(default: `8089`)**

username
: The login name that has rights to send data to the tcp-port **(default: `admin`)**

password
: The password associated with the username **(default: `password`)**

owner
: The owner of the tcp-port **(default: `admin1`)**

tcp-port
: The TCP port number to where XD will send the data **(default: `9500`)**

Setup Splunk for TCP Input

1. From the Manager page select `Manage Inputs` link
2. Click the `Add data` Button
3. Click the `From a TCP port` link
4. `TCP Port` enter the port you want Splunk to monitor
5. `Set Source Type` select `Manual`
6. `Source Type` enter `tcp-raw`
7. Click `Save`

Example

An example stream would be to take data from a twitter search and push it through to a splunk instance.

```
xd:> stream create --name springone2gx --definition "twittersearch --consumerKey= --
consumerSecret= --query='#LOTR' | splunk"
```

9.11 MQTT

The mqtt sink connects to an mqtt server and publishes telemetry messages.

Options

The folllowing options are configured in mqtt.properties in XD_HOME/config

```
mqtt.url=tcp://localhost:1883
mqtt.default.client.id=xd.mqtt.client.id
mqtt.username=guest
mqtt.password=guest
mqtt.default.topic=xd.mqtt.test
```

The defaults are set up to connect to the RabbitMQ MQTT adapter on localhost.

Note that the client id must be no more than 19 characters; this is because `.snk` is added and the id must be no more than 23 characters.

clientId
: Identifies the client - overrides the default above.

topic
: The topic to which the sink will publish - overrides the default above.

qos
: The Quality of Service (default: 1)

retained
: Whether the retained flag is set (default: false)

9.12 Dynamic Router

The Dynamic Router support allows for routing Spring XD messages to **named channels** based on the evaluation of SpEL expressions or Groovy Scripts.

SpEL-based Routing

In the following example, 2 streams are created that listen for message on the **foo** and the **bar** channel. Furthermore, we create a stream that receives messages via HTTP and then delegates the received messages to a router:

```
xd:>stream create f --definition ":foo > transform --expression=payload+'-foo' | log"
Created new stream 'f'

xd:>stream create b --definition ":bar > transform --expression=payload+'-bar' | log"
Created new stream 'b'

xd:>stream create r --definition "http | router --
expression=payload.contains('a')?':foo':':bar'"
Created new stream 'r'
```

Now we make 2 requests to the HTTP source:

```
xd:>http post --data "a"
> POST (text/plain;Charset=UTF-8) http://localhost:9000 a
> 200 OK

xd:>http post --data "b"
> POST (text/plain;Charset=UTF-8) http://localhost:9000 b
> 200 OK
```

In the server log you should see the following output:

```
11:54:19,868  WARN ThreadPoolTaskScheduler-1 logger.f:145 - a-foo
11:54:25,669  WARN ThreadPoolTaskScheduler-1 logger.b:145 - b-bar
```

For more information, please also consult the Spring Integration Reference manual: http://static.springsource.org/spring-integration/reference/html/messaging-routing-chapter.html#router-namespace particularly the section "Routers and the Spring Expression Language (SpEL)".

Groovy-based Routing

Instead of SpEL expressions, Groovy scripts can also be used. Let's create a Groovy script in the file system at "/my/path/router.groovy"

```
println("Groovy processing payload '" + payload +"'");
if (payload.contains('a')) {
 return ":foo"
}
else {
 return ":bar"
}
```

Now we create the following streams:

```
xd:>stream create f --definition ":foo > transform --expression=payload+'-foo' | log"
Created new stream 'f'

xd:>stream create b --definition ":bar > transform --expression=payload+'-bar' | log"
Created new stream 'b'

xd:>stream create g --definition "http | router --script='file:/my/path/router.groovy'"
```

Now post some data to the HTTP source:

```
xd:>http post --data "a"
> POST (text/plain;Charset=UTF-8) http://localhost:9000 a
> 200 OK

xd:>http post --data "b"
> POST (text/plain;Charset=UTF-8) http://localhost:9000 b
> 200 OK
```

In the server log you should see the following output:

```
Groovy processing payload 'a'
11:29:27,274  WARN ThreadPoolTaskScheduler-1 logger.f:145 - a-foo
Groovy processing payload 'b'
11:34:09,797  WARN ThreadPoolTaskScheduler-1 logger.b:145 - b-bar
```

Note

You can also use Groovy scripts located on your classpath by specifying:

```
--script='org/my/package/router.groovy'
```

For more information, please also consult the Spring Integration Reference manual: "Groovy support" http://static.springsource.org/spring-integration/reference/html/messaging-endpoints-chapter.html#groovy

Options

expression
: The SpEL expression to use for routing

script
: Indicates that Groovy Script based routing is used. If this property is set, then the "Expression" attribute will be ignored. The groovy script is checked for updates every 60 seconds. The script can be loaded from the classpath or from the file system e.g. "--script=*org/springframework/springxd/samples/batch/router.groovy*" or "--script=*file:/my/path/router.groovy*"

properties-location
: Will be made available as script variables for Groovy Script based routing. Will only be evaluated once at initialization time. By default the following script variables will be made available: "payload" and "headers".

10. Taps

10.1 Introduction

A Tap allows you to "listen in" to data from another stream and process the data separately from the original stream definition. The original stream is unaffected by the tap and isn't aware of its presence, similar to a phone wiretap (WireTaps are part of the standard catalog of EAI patterns and are part of the Spring Integration EAI framework used by Spring XD).

A tap acts like a source in that it occurs as the first module within a stream and can pipe its output to a sink (and/or one or more processors added to a chain before the ultimate sink), but for a tap the messages are actually those being processed by some other stream.

Taps are specified as named channels in a stream definition, where the channel name always begins with `tap:`.

To create a tap using the shell, use the following command (assuming you want to tap into the "foo1" stream, which we'll create first):

```
xd:> stream create --name foo1 --definition "time | log"
xd:> stream create --name tapname --definition "tap:stream:foo1 > log"
```

A tap can consume data from any point along the target stream's processing pipeline. For example, if you have a stream called `mystream`, defined as

```
source | filter | transformer | sink
```

Then creating a tap using

```
tap:mystream.<filter module name> > sink2
```

would tap into the stream's data after the filter has been applied but before the transformer. So the untransformed data would be sent to `sink2`.

A primary use case is to perform realtime analytics at the same time as data is being ingested via its primary stream. For example, consider a Stream of data that is consuming Twitter search results and writing them to HDFS. A tap can be created before the data is written to HDFS, and the data piped from the tap to a counter that correspond to the number of times specific hashtags were mentioned in the tweets.

Creating a tap on a named channel, a stream whose source is a named channel, or a label is not yet supported. This is planned for a future release.

You'll find specific examples of creating taps on existing streams in the Analytics section.

10.2 Tap Lifecycle

A side effect of a stream being unaware of any taps on its pipeline is that deleting the stream will not automatically delete the taps. The taps have to be deleted separately. However if the tapped stream is re-created, the existing tap will continue to function.

11. Batch Jobs

11.1 Introduction

One of the features that XD offers is the ability to launch and monitor batch jobs using Spring Batch. This purpose of this section is to show you how to create a sample tasklet as well as create, schedule and monitor a job.

11.2 Setting up a simple Batch Job

Creating the Tasklet

We will create a very simple Tasklet. The sole purpose of this Tasklet is to print out "Hello Spring XD!". Note, you can find the the source code and the maven build files for this example in the Spring XD Samples repository.

```
package org.springframework.springxd.samples.batch;

import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;

public class HelloSpringXDTasklet implements Tasklet {

 public RepeatStatus execute(StepContribution contribution,
   ChunkContext chunkContext) throws Exception {

  System.out.println("Hello Spring XD!");

  return RepeatStatus.FINISHED;
 }
}
```

Please ensure that you deploy this class as part of a Jar file to the Spring XD **${xd.home}/lib** folder. Once you restart the Spring XD container the class will be automatically added to the classpath and thus made available. If you are building from the sample repository do the following in the directory `spring-xd-samples/batch-simple`

- `mvn package`
- `cp ./target/springxd-batch-simple-1.0.0.BUILD-SNAPSHOT.jar $XD_HOME/lib`

Setting Up the Application Context

Under **modules/job**, in the Spring XD home directory, please create the following XML application context file named **myjob.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:int="http://www.springframework.org/schema/integration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:batch="http://www.springframework.org/schema/batch"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd
  http://www.springframework.org/schema/batch
  http://www.springframework.org/schema/batch/spring-batch.xsd">

 <batch:job id="job">
  <batch:step id="helloSpringXDStep">
   <batch:tasklet ref="helloSpringXDTasklet" />
  </batch:step>
 </batch:job>

 <bean id="helloSpringXDTasklet"
  class="org.springframework.springxd.samples.batch.HelloSpringXDTasklet" />

</beans>
```

This context file must contain single batch job whose id is `job`.

If you are building from the sample repository do the following in the directory `spring-xd-samples/batch-simple`

- `cp ./src/main/resources/myjob.xml $XD_HOME/modules/job/`

11.3 Creating your Job

Now from the XD-Shell let's create a job. This is done by executing a job create command composed of:

- name - the "name" that will be associated with the Job
- definition - the name of the context file that describes the tasklet.

So using our example above where we declared a myjob.xml that contains the context for our task, so to create the job will look like:

```
xd:> job create --name helloSpringXD --definition "myjob"
```

In the logging output of the XDContainer you should see the following:

```
14:17:46,793  INFO http-bio-8080-exec-5 job.JobPlugin:87 - Configuring module
 with the following properties: {numberFormat=, dateFormat=, makeUnique=true,
 xd.stream.name=helloSpringXD}
14:17:46,837  INFO http-bio-8080-exec-5 module.SimpleModule:140 - initialized module:
 SimpleModule [name=myjob, type=job, group=helloSpringXD, index=0]
14:17:46,840  INFO http-bio-8080-exec-5 module.SimpleModule:154 - started module:
 SimpleModule [name=job, type=job, group=helloSpringXD, index=0]
14:17:46,840  INFO http-bio-8080-exec-5 module.ModuleDeployer:152 - launched job module:
 helloSpringXD:myjob:0
```

11.4 Launching a job

XD uses triggers as well as regular event flow to launch the batch jobs. So in this section we will cover how to:

- Launch the Batch Job Ad-hoc
- Launch the Batch Job using a named Cron-Trigger
- Launch the Batch Job as sink.

Ad-hoc

To launch a job one time, use the launch option of the job command. So going back to our example above, we've created a job module instance named helloSpringXD. Launching that Job Module Instance would look like:

```
xd:> job launch helloSpringXD
```

In the logging output of the XDContainer you should see the following

```
16:45:40,127  INFO http-bio-9393-exec-1 job.JobPlugin:98 - Configuring module with the
 following properties: {numberFormat=, dateFormat=, makeUnique=true, xd.stream.name=myjob}
16:45:40,185  INFO http-bio-9393-exec-1 module.SimpleModule:140 - initialized module:
 SimpleModule [name=job, type=job, group=myjob, index=0 @3a9ecb9d]
16:45:40,198  INFO http-bio-9393-exec-1 module.SimpleModule:161 - started module:
 SimpleModule [name=job, type=job, group=myjob, index=0 @3a9ecb9d]
16:45:40,199  INFO http-bio-9393-exec-1 module.ModuleDeployer:161 - deployed SimpleModule
 [name=job, type=job, group=myjob, index=0 @3a9ecb9d]
Hello Spring XD!
```

To re-launch the job just execute the launch command. For example:

```
xd:> job launch helloSpringXD
```

Launch the Batch using Cron-Trigger

To launch a batch job based on a cron scheduler is done by creating a stream using the cron-trigger source.

```
xd:> stream create --name cronStream --definition "cron-trigger --cron='0/5 * * * * *'  >
 queue:job:myCronJob"
```

A batch job can receive parameters from a source (in this case a trigger) or process. A trigger uses the --payload expression to declare its payload.

```
xd:> stream create --name cronStream --definition "cron-trigger --cron='0/5 * * * * *'  --
payload='{"param1":"Kenny"}' > queue:job:myCronJob"
```

Note

The payload content must be in a JSON-based map representation.

To pause/stop future scheduled jobs from running for this stream, the stream must be undeployed for example:

```
xd:> stream undeploy --name cronStream
```

Launch the Batch using a Fixed-Delay-Trigger

A fixed-delay-trigger is used to launch a Job on a regular interval. Using the --fixedDelay parameter you can set up the number of seconds between executions. In the example below we are running myXDJob every 10 seconds and passing it a payload containing a single attribute.

```
xd:> stream create --name fdStream --definition "fixed-delay-trigger --
payload='{"param1":"fixedDelayKenny"}' --fixedDelay=10 > queue:job:myXDJob"
```

To pause/stop future scheduled jobs from running for this stream, you must undeploy the stream for example:

```
xd:> stream undeploy --name cronStream
```

Launch job as a part of event flow

A batch job is always used as a sink, with that being said it can receive messages from sources (other than triggers) and processors. In the case below we see that the user has created a http source (http source receives http posts and passes the payload of the http message to the next module in the stream) that will pass the http payload to the "myHttpJob".

```
 stream create --name jobStream --definition "http > queue:job:myHttpJob"
```

To test the stream you can execute a http post, like the following:

```
xd:> http post --target http://localhost:9000 --data "{"param1":"fixedDelayKenny"}"
```

11.5 Retrieve job notifications

XD offers the facilities to capture the notifications that are sent from the job as it is executing.

Notifications include:

- Job Execution Listener
- Chunk Listener
- Item Listener
- Step Execution Listener
- Skip Listener

In this example, the job will send notifications to the log.

```
stream create --name jobNotifications --definition ":myHttpJob-notifications >log"
```

In the logging output of the container you should see something like the following when the job completes:

```
15:26:30,029  WARN task-scheduler-5 logger.jobNotifications:145 - JobExecution:
 id=1, version=2, startTime=Wed Aug 28 15:26:30 EDT 2013, endTime=Wed Aug 28
 15:26:30 EDT 2013, lastUpdated=Wed Aug 28 15:26:30 EDT 2013, status=COMPLETED,
 exitStatus=exitCode=COMPLETED;exitDescription=, job=[JobInstance: id=1, version=0,
 Job=[myHttpJob.job]], jobParameters=[{random=0.49881213192780494}]
```

11.6 Removing Batch Jobs

Batch Jobs can be deleted by executing:

```
xd:> job destroy helloSpringXD
```

Alternatively, one can just undeploy the job, keeping its definition for a future redeployment:

```
xd:> job undeploy helloSpringXD
```

11.7 Pre-Packaged Batch Jobs

Spring XD comes with several batch import and export modules. You can run them out of the box or use them as a basis for building your own custom modules.

Import Files to HDFS (`filehdfs`)

This module is designed to be driven by a stream. It imports data from CSV files and requires that you supply a list of named columns for the data using the `names` parameter. For example:

```
xd:> job create myjob --definition "filehdfs --names=forename,surname,address"
```

You would then use a stream with a file source to scan a directory for files and drive the job. A separate file will be started for each job found:

```
xd:> stream create csvStream --definition "file --ref=true --dir=/mycsvdir --pattern=*.csv
 > queue:job:myjob"
```

Import Files to JDBC (`filejdbc`)

A module which loads CSV files from a directory into a JDBC table using a single batch job. By default it uses the file `config/batch-jdbc-import.properties` to configure the module and stores data in the internal HSQL DB which is used by Spring Batch. The job should be defined with the `resources` parameter defining the files which should be loaded. It also requires a `names` parameter (for the CSV field names) and these should match the database column names into which the data should be stored. You can either pre-create the database table or the module will try to create it for you when it is loaded. The table intitialization is configured in a similar way to the JDBC sink and uses the same parameters. The default table name is the job name and can be cumstomized by setting the `tableName` parameter. As an example, if you run the command

```
xd:> job create myjob --definition "filejdbc --resources=/mycsvdir/*.csv --
names=forename,surname,address --tableName=people"
```

it will create the table "people" in the database with three varchar columns called "forename", "surname" and "address". When you launch the job it will load the files matching the resources pattern and write the data to this table.

HDFS to JDBC Export (`hdfsjdbc`)

This module functions very similarly to the `filejdbc` one except that the resources you specify should actually be in HDFS, rather than the OS filesystem. Other than that the syntax is the same

```
xd:> job create myjob --definition "hdfsjdbc --resources=/data/*.log --
names=forename,surname,address --tableName=people"
```

there is also a limitation in that the database table must be created manually. This is due to a bug in Spring Hadoop and will be fixed in the future.

HDFS to MongoDB Export (`hdfsmongodb`)

Exports CSV data from HDFS and stores it in a MongoDB collection which defaults to the stream name. This can be overridden with the `collectionName` parameter. The job is configured using the file `config/batch-mongo-import.properties`. Once again, the field names should be defined by supplying the `names` parameter. The data is converted internally to a Spring XD `Tuple` and the collection items will have an `id` matching the tuple's UUID. You can override this by setting the `idField` parameter to one of the field names if desired.

An example:

```
xd:> job create myjob --definition "hdfsmongodb --resources=/data/*.log --
names=employeeId,forename,surname,address --idField=employeeId --collectionName=people"
```

12. Analytics

12.1 Introduction

Spring XD Analytics provides support for real-time analysis of data using metrics such as counters and gauges. Spring XD intends to support a wide range of these metrics and analytical data structures as a general purpose class library that works with several backend storage technologies.

We'll look at the following metrics

- Counter
- Field Value Counter
- Aggregate Counter
- Gauge
- Rich Gauge

An in memory implementation and a Redis implementation are provided in **Spring XD 1.0.0.M3**. Other metrics that will be provided in a future release are Rate Counters and Histograms.

Metrics can be used directly in place of a sink just as if you were creating any other stream, but you can also analyze data from an existing stream using a tap. We'll look at some examples of using metrics with taps in the following sections. As a prerequisite start the XD Container as instructed in the Getting Started page.

12.2 Counter

A counter is a Metric that associates a unique name with a long value. It is primarily used for counting events triggered by incoming messages on a target stream. You create a counter with a unique name and optionally an initial value then set its value in response to incoming messages. The most straightforward use for counter is simply to count messages coming into the target stream. That is, its value is incremented on every message. This is exactly what the *counter* module provided by Spring XD does.

Here's an example:

Start by creating a data ingestion stream. Something like:

```
xd:> stream create --name springtweets --definition "twittersearch --
consumerKey=<your_key> --consumerSecret=<your_secret> --query=spring | file --dir=/
tweets/"
```

Next, create a tap on the *springtweets* stream that sets a message counter named *tweetcount*

```
xd:> stream create --name tweettap --definition "tap:stream:springtweets > counter --
name=tweetcount"
```

```
$ redis-cli
redis 127.0.0.1:6379> get counters.tweetcount
```

12.3 Field Value Counter

A field value counter is a Metric used for counting occurrences of unique values for a named field in a message payload. XD Supports the following payload types out of the box:

- POJO (Java bean)
- Tuple
- JSON String

For example suppose a message source produces a payload with a field named *user* :

```
class Foo {
   String user;
   public Foo(String user) {
       this.user = user;
   }
}
```

If the stream source produces messages with the following objects:

```
   new Foo("fred")
   new Foo("sue")
   new Foo("dave")
   new Foo("sue")
```

The field value counter on the field *user* will contain:

```
fred:1, sue:2, dave:1
```

Multi-value fields are also supported. For example, if a field contains a list, each value will be counted once:

```
users:["dave","fred","sue"]
users:["sue","jon"]
```

The field value counter on the field *users* will contain:

```
dave:1, fred:1, sue:2, jon:1
```

field_value_counter has the following options:

fieldName
: The name of the field for which values are counted **(required)**

counterName
: A key used to access the counter values. **(default: ${fieldName})**

To try this out, create a stream to ingest twitter feeds containing the word *spring* and output to a file:

```
xd:> stream create --name springtweets --definition "twittersearch --
consumerKey=<your_key> --consumerSecret=<your_secret> --query=spring | file"
```

Now create a tap for a field value counter:

```
xd:> stream create --name tweettap --definition "tap:stream:springtweets > field-value-
counter --fieldName=fromUser"
```

The *twittersearch* source produces JSON strings which contain the user id of the tweeter in the *fromUser* field. The *field_value_counter* sink parses the tweet and updates a field value counter named *fromUser* in Redis. To view the counts:

```
$ redis-cli
redis 127.0.0.1:6379>zrange fieldvaluecounters.fromUser 0 -1 withscores
```

12.4 Aggregate Counter

The aggregate counter differs from a simple counter in that it not only keeps a total value for the count, but also retains the total count values for each minute, hour day and month of the period for which it is run. The data can then be queried by supplying a start and end date and the resolution at which the data should be returned.

Creating an aggregate counter is very similar to a simple counter. For example, to obtain an aggregate count for our spring tweets stream:

```
xd:> stream create --name springtweets --definition "twittersearch --query=spring | file"
```

you'd simply create a tap which pipes the input to `aggregatecounter`:

```
xd:> stream create --name tweettap --definition "tap:stream:springtweets >
 aggregatecounter --name=tweetcount"
```

The Redis back-end stores the aggregate counts in buckets prefixed with `aggregatecounters.${name}`. The rest of the string contains the date information. So for our `tweetcount` counter you might see something like the following keys appearing in Redis:

```
redis 127.0.0.1:6379> keys aggregatecounters.tweetcount*
1) "aggregatecounters.tweetcount"
2) "aggregatecounters.tweetcount.years"
3) "aggregatecounters.tweetcount.2013"
4) "aggregatecounters.tweetcount.201307"
5) "aggregatecounters.tweetcount.20130719"
6) "aggregatecounters.tweetcount.2013071914"
```

The general format is

1. One total value
2. One years hash with a field per year eg. { 2010: value, 2011: value }
3. One hash per year with a field per month { 01: value, …}
4. One hash per month with a field per day
5. One hash per day with a field per hour
6. One hash per hour with a field per minute

12.5 Gauge

A gauge is a Metric, similar to a counter in that it holds a single long value associated with a unique name. In this case the value can represent any numeric value defined by the application.

The *gauge* sink provided with XD stores expects a numeric value as a payload, typically this would be a decimal formatted string, and stores its values in Redis. The gauge includes the following attributes:

name
: The name for the gauge **(default: `<streamname>`)**

Here is an example of creating a tap for a gauge:

Simple Tap Example

Create an ingest stream

```
xd:> stream create --name test --definition "http --port=9090 | file"
```

Next create the tap:

```
xd:> stream create --name simplegauge --definition "tap:stream:test > gauge"
```

Now Post a message to the ingest stream:

```
xd:> http post --target http://localhost:9090 --data "10"
```

Check the gauge:

```
$ redis-cli
redis 127.0.0.1:6379> get gauges.simplegauge
"10"
```

12.6 Rich Gauge

A rich gauge is a Metric that holds a double value associated with a unique name. In addition to the value, the rich gauge keeps a running average, along with the minimum and maximum values and the sample count.

The *richgauge* sink provided with XD expects a numeric value as a payload, typically this would be a decimal formatted string, and keeps its value in a store. The richgauge includes the following attributes:

name
: The name for the gauge **(default: `<streamname>`)**

alpha
: A smoothing factor between 0 and 1, that if set will compute an exponential moving average **(default: `-1, simple average`)**

When stored in Redis, the values are kept as a space delimited string, formatted as *value alpha mean max min count*

Here are some examples of creating a tap for a rich gauge:

Simple Tap Example

Create an ingest stream

```
xd:> stream create --name test --definition "http --port=9090 | file"
```

Next create the tap:

```
xd:> stream create --name testgauge --definition "tap:stream:test > richgauge"
```

Now Post some messages to the ingest stream:

```
xd:> http post --target http://localhost:9090 --data "10"
xd:> http post --target http://localhost:9090 --data "13"
xd:> http post --target http://localhost:9090 --data "16"
```

Check the gauge:

```
$ redis-cli
redis 127.0.0.1:6379> get richgauges.testgauge
"16.0 -1 13.0 16.0 10.0 3"
```

Stock Price Example

In this example, we will track stock prices, which is a more practical example. The data is ingested as JSON strings like

```
{"symbol":"VMW","price":72.04}
```

Create an ingest stream

```
xd:> stream create --name stocks --definition "http --port=9090 | file"
```

Next create the tap, using the json-field-extractor to extract the stock price from the payload:

```
xd:> stream create --name stockprice --definition "tap:stream:stocks > json-field-
extractor --fieldName=price | richgauge"
```

Now Post some messages to the ingest stream:

```
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"VMW\",\"price
\":72.04}"
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"VMW\",\"price
\":72.06}"
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"VMW\",\"price
\":72.08}"
```

Check the gauge:

```
$ redis-cli
redis 127.0.0.1:6379> get richgauges.stockprice
"72.08 -1 72.04 72.08 72.02 3"
```

Improved Stock Price Example

In this example, we will track stock prices for selected stocks. The data is ingested as JSON strings like

```
{"symbol":"VMW","price":72.04}
{"symbol":"EMC","price":24.92}
```

The previous example would feed these prices to a single gauge. What we really want is to create a separate tap for each ticker symbol in which we are interested:

Create an ingest stream

```
xd:> stream create --name stocks --definition "http --port=9090 | file"
```

Next create the taps, using the json-field-extractor to extract the stock price from the payload:

```
xd:> stream create --name vmwprice --definition "tap:stream:stocks > json-field-value-
filter --fieldName=symbol --fieldValue=VMW | json-field-extractor --fieldName=price |
 richgauge"
xd:> stream create --name emcprice --definition "tap:stream:stocks > json-field-value-
filter --fieldName=symbol --fieldValue=EMC | json-field-extractor --fieldName=price |
 richgauge"
```

Now Post some messages to the ingest stream:

```
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"VMW\",\"price
\":72.04}"
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"VMW\",\"price
\":72.06}"
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"VMW\",\"price
\":72.08}"
```

```
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"EMC\",\"price
\":24.92}"
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"EMC\",\"price
\":24.90}"
xd:> http post --target http://localhost:9090 --data "{\"symbol\":\"EMC\",\"price
\":24.96}"
```

Check the gauge:

```
$ redis-cli
redis 127.0.0.1:6379> get richgauges.emcprice
"24.96 -1 24.926666666666666 24.96 24.9 3"
```

```
redis 127.0.0.1:6379> get richgauges.vmwprice
"72.08 -1 72.04 72.08 72.02 3"
```

12.7 Accessing Analytics Data over the RESTful API

Spring XD has a discoverable RESTful API based on the Spring HATEAOS library. You can discover the resources available by making a GET request on the root resource of the Admin server. Here is an example where navigate down to find the data for a counter named *httptap* that was created by these commands

```
xd:>stream create --name httpStream --definition "http | file"
xd:>stream create --name httptap --definition "tap:stream:httpStream > counter"
xd:>http post --target http://localhost:9000 --data "helloworld"
```

The root resource returns

```
xd:>! wget  -q -S -O - http://localhost:9393/
{
  "links":[
    {},
    {
      "rel":"jobs",
      "href":"http://localhost:9393/jobs"
    },
    {
      "rel":"modules",
      "href":"http://localhost:9393/modules"
    },
    {
      "rel":"runtime/modules",
      "href":"http://localhost:9393/runtime/modules"
    },
    {
      "rel":"runtime/containers",
      "href":"http://localhost:9393/runtime/containers"
    },
    {
      "rel":"counters",
      "href":"http://localhost:9393/metrics/counters"
    },
    {
      "rel":"field-value-counters",
      "href":"http://localhost:9393/metrics/field-value-counters"
    },
    {
      "rel":"aggregate-counters",
      "href":"http://localhost:9393/metrics/aggregate-counters"
    },
    {
      "rel":"gauges",
      "href":"http://localhost:9393/metrics/gauges"
    },
    {
      "rel":"richgauges",
      "href":"http://localhost:9393/metrics/richgauges"
    }
  ]
}
```

Following the resource location for the counter

```
xd:>! wget  -q -S -O - http://localhost:9393/metrics/counters
{
  "links":[

  ],
  "content":[
    {
      "links":[
        {
          "rel":"self",
          "href":"http://localhost:9393/metrics/counters/httptap"
        }
      ],
      "name":"httptap"
    }
  ],
  "page":{
    "size":0,
    "totalElements":1,
    "totalPages":1,
    "number":0
  }
}
```

And then the data for the counter itself

```
xd:>! wget  -q -S -O - http://localhost:9393/metrics/counters/httptap
{
  "links":[
    {
      "rel":"self",
      "href":"http://localhost:9393/metrics/counters/httptap"
    }
  ],
  "name":"httptap",
  "value":2
}
```

13. DSL Reference

13.1 Introduction

Spring XD provides a DSL for defining a stream. Over time the DSL is likely to evolve significantly as it gains the ability to define more and more sophisticated streams as well as the steps of a batch job.

13.2 Pipes and filters

A simple linear stream consists of a sequence of modules. Typically an Input Source, (optional) Processing Steps, and an Output Sink. As a simple example consider the collection of data from an HTTP Source writing to a File Sink. Using the DSL the stream description is:

```
http | file
```

A stream that involves some processing:

```
http | filter | transform | file
```

The modules in a stream definition are connected together using the pipe symbol `|`.

13.3 Module parameters

Each module may take parameters. The parameters supported by a module are defined by the module implementation. As an example the `http` source module exposes `port` setting which allows the data ingestion port to be changed from the default value.

```
http --port=1337
```

It is only necessary to quote parameter values if they contain spaces or the `|` character. Here the transform processor module is being passed a SpEL expression that will be applied to any data it encounters:

```
transform --expression='new StringBuilder(payload).reverse()'
```

If the parameter value needs to embed a single quote, use two single quotes:

```
// Query is: Select * from /Customers where name='Smith'
scan --query='Select * from /Customers where name=''Smith'''
```

13.4 Named channels

Instead of a source or sink it is possible to use a named channel. Normally the modules in a stream are connected by anonymous internal channels (represented by the pipes), but by using explicitly named channels it becomes possible to construct more sophisticated flows. In keeping with the unix theme, sourcing/sinking data from/to a particular channel uses the > character. A channel name is prefixed with a `:`

Here is an example that shows how you can use a named channel to share a data pipeline driven by different input sources.

```
:foo > file
```

```
http > :foo
```

```
time > :foo
```

Now if you post data to the http source, you will see that data intermingled with the time value in the file.

The opposite case, the fanout of a message to multiple streams, is planned for a future release. However, taps are a specialization of named channels that do allow publishing data to multiple sinks. For example:

```
tap:mystream > file
```

```
tap:mystream > log
```

Once data is received on `mystream`, it will be written to both file and log. Note that, unlike other named channels, references to a tap do not contain a leading `:`.

Support for routing messages to different streams based on message content is also planned for a future release.

13.5 Labels

Labels provide a means to alias or group modules. Labels are simply a name followed by a `:` When used as an alias a label can provide a more descriptive name for a particular configuration of a module and possibly something easier to refer to in other streams.

```
mystream = http | obfuscator: transform --expression=payload.replaceAll('password','*') |
 file
```

A module may have multiple labels:

```
mystream = http | foo: bar: transform --expression=payload.replaceAll('password','*') |
 file
```

When used for grouping a series of modules might share the same label:

```
mystream = http | group1: filter | group1: transform | file
```

Referring to the label `group1` then effectively refers to all the labeled modules. This is not yet exploited in XD but in future may be used for something like configuring deployment options:

```
// Ensure all modules in group1 are on the same machine
group1.colocation = true
```

14. Tuples

14.1 Introduction

The Tuple class is a central data structure in Spring XD. It is an ordered list of values that can be retrieved by name or by index. Tuples are created by a TupleBuilder and are immutable. The values that are stored can be of any type and null values are allowed.

The underlying Message class that moves data from one processing step to the next can have an arbitrary data type as its payload. Instead of creating a custom Java class that encapsulates the properties of what is read or set in each processing step, the Tuple class can be used instead. Processing steps can be developed that read data from specific named values and write data to specific named values.

There are accessor methods that perform type conversion to the basic primitive types as well as BigDecimal and Date. This avoids you from having to cast the values to specific types. Insteam you can rely on the Tuple's type conversion infastructure to perform the conversion.

The Tuple's types conversion is performed by Spring's Type Conversion Infrastructure which supports commonly encountered type conversions and is extensible.

There are several overloads for getters that let you provide default values for primitive types should the field you are looking for not be found. Date format patterns and Locale aware NumberFormat conversion are also supported. A best effort has been made to preserve the functionality available in Spring Batch's `FieldSet` class that has been extensively used for parsing String based data in files.

Creating a Tuple

The `TupleBuilder` class is how you create new `Tuple` instances. The most basic case is

```
Tuple tuple = TupleBuilder.tuple().of("foo", "bar");
```

This creates a Tuple with a single entry, a key of *foo* with a value of *bar*. You can also use a static import to shorten the syntax.

```
import static org.springframework.xd.tuple.TupleBuilder.tuple;

Tuple tuple = tuple().of("foo", "bar");
```

You can use the `of` method to create a Tuple with up to 4 key-value pairs.

```
Tuple tuple2 = tuple().of("up", 1, "down", 2);
Tuple tuple3 = tuple().of("up", 1, "down", 2, "charm", 3 );
Tuple tuple4 = tuple().of("up", 1, "down", 2, "charm", 3, "strange", 4);
```

To create a Tuple with more then 4 entries use the fluent API that strings together the `put` method and terminates with the `build` method

```
Tuple tuple6 = tuple().put("up", 1)
                      .put("down", 2)
        .put("charm", 3)
        .put("strange", 4)
        .put("bottom", 5)
        .put("top", 6)
        .build();
```

To customize the underlying type conversion system you can specify the `DateFormat` to use for converting `String` to `Date` as well as the `NumberFormat` to use based on a `Locale`. For more advanced customization of the type conversion system you can register an instance of a `FormattingConversionService`. Use the appropriate setter methods on `TupleBuilder` to make these customizations.

You can also create a Tuple from a list of `String` field names and a List of `Object` values.

```
Object[] tokens = new String[]
 { "TestString", "true", "C", "10", "-472", "354224", "543", "124.3", "424.3", "1,3245",
    null, "2007-10-12", "12-10-2007", "" };
String[] nameArray = new String[]
 { "String", "Boolean", "Char", "Byte", "Short", "Integer", "Long", "Float", "Double",
    "BigDecimal", "Null", "Date", "DatePattern", "BlankInput" };

List<String> names = Arrays.asList(nameArray);
List<Object> values = Arrays.asList(tokens);
tuple = tuple().ofNamesAndValues(names, values);
```

Getting Tuple values

There are getters for all the primitive types and also for BigDecimal and Date. The primitive types are

- `Boolean`
- `Byte`
- `Char`
- `Double`
- `Float`
- `Int`
- `Long`
- `Short`
- `String`

Each getter has an overload for providing a default value. You can access the values either by field name or by index.

The overloaded methods for asking for a value to be converted into an integer are

- `int getInt(int index)`
- `int getInt(String name)`
- `int getInt(int index, int defaultValue)`
- `int getInt(String name, int defaultValue)`

There are similar methods for other primitive types. For `Boolean` there is a special case of providing the `String` value that represents a `trueValue`.

- `boolean getBoolean(int index, String trueValue)`
- `boolean getBoolean(String name, String trueValue)`

If the value that is stored for a given field or index is null and you ask for a primitive type, the standard Java defalt value for that type is returned.

The `getString` method will remove and leading and trailing whitespace. If you want to get the String and preserve whitespace use the methods `getRawString`

There is extra functionality for getting `Date`s. The are overloaded getters that take a String based date format

- `Date getDateWithPattern(int index, String pattern)`
- `Date getDateWithPattern(int index, String pattern, Date defaultValue)`
- `Date getDateWithPattern(String name, String pattern)`
- `Date getDateWithPattern(String name, String pattern, Date defaultValue)`

There are a few other more generic methods available. Their functionality should be obvious from their names

- `size()`
- `getFieldCount()`
- `getFieldNames()`
- `getFieldTypes()`
- `getTimestamp()` - the time the tuple was created - milliseconds since epoch
- `getId()` - the UUID of the tuple
- `Object getValue(int index)`
- `Object getValue(String name)`
- `T getValue(int index, Class<T> valueClass)`
- `T getValue(String name, Class<T> valueClass)`
- `List<Object> getValues()`
- `List<String> getFieldNames()`
- `boolean hasFieldName(String name)`

Using SpEL expressions to filter a tuple

SpEL provides support to transform a source collection into another by selecting from its entries. We make use of this functionalty to select a elements of a the tuple into a new one.

```
Tuple tuple = tuple().put("red", "rot")
                     .put("brown", "braun")
       .put("blue", "blau")
       .put("yellow", "gelb")
       .put("beige", "beige")
       .build();

Tuple selectedTuple = tuple.select("?[key.startsWith('b')]");
assertThat(selectedTuple.size(), equalTo(3));
```

To select the first match use the ^ operator

```
selectedTuple = tuple.select("^[key.startsWith('b')]");

assertThat(selectedTuple.size(), equalTo(1));
assertThat(selectedTuple.getFieldNames().get(0), equalTo("brown"));
assertThat(selectedTuple.getString(0), equalTo("braun"));
```

15. Samples

15.1 Syslog ingestion into HDFS

In this section we will show a simple example on how to setup syslog ingestion from multiple hosts into HDFS.

Create the streams with syslog as source and HDFS as sink (Please refer to source and sink)

```
xd:> stream create --definition "syslog-udp --port=<udp-port> | hdfs" --name <stream-name>
```

```
xd:> stream create --definition "syslog-tcp --port=<tcp-port> | hdfs" --name <stream-name>
```

Please note for hdfs sink, set `rollover` parameter to a smaller value to avoid buffering and to see the data has made to HDFS (incase of smaller volume of log).

Configure the external hosts' syslog daemons forward their messages to the xd-container host's UDP/TCP port (where the syslog-udp/syslog-tcp source module is deployed).

A sample configuration using syslog-ng

Edit /etc/syslog-ng/syslog-ng.conf :

1) Add destination

```
Add destination <destinationName> {
      tcp("<xd-container-host>" port("<tcp-port>"));
};
```

or,

```
Add destination <destinationName> {
      udp("<xd-container-host>" port("<udp-port>"));
};
```

2) Add log rule to log message sources:

```
log {
  source(<message_source>); destination(<destinationName>);
};
```

We can use "s_all" as message source to try this example.

3) Make sure to restart the service after the change:

```
sudo service syslog-ng restart
```

Now, the syslog messages are written into HDFS /xd/<stream-name>/

Part II. Appendices

Appendix A. Installing Hadoop

A.1 Installing Hadoop

If you don't have a local *Hadoop* cluster available already, you can do a local single node installation (v1.2.1) and use that to try out *Hadoop* with *Spring XD*.

Tip

This guide is intended to serve as a quick guide to get you started in the context of *Spring XD*. For more complete documentation please refer back to the documentation provided by your respective *Hadoop* distribution.

Download

First, download an installation archive and unpack it locally. Linux users can also install *Hadoop* through the system package manager and on Mac OS X, you can use Homebrew. However, the manual installation is self-contained and it's easier to see what's going on if you just unpack it to a known location.

If you have `wget` available on your system, you can also execute:

```
$ wget http://archive.apache.org/dist/hadoop/common/hadoop-1.2.1/hadoop-1.2.1.tar.gz
```

Unpack the distribution with:

```
$ tar xzf hadoop-1.2.1.tar.gz
```

Change into the directory and have a look around

```
$ cd hadoop-1.2.1
$ ls
$ bin/hadoop
Usage: hadoop [--config confdir] COMMAND
where COMMAND is one of:
  namenode -format     format the DFS filesystem
  secondarynamenode    run the DFS secondary namenode
  namenode             run the DFS namenode
  ...
```

The `bin` directory contains the start and stop scripts as well as the `hadoop` script which allows us to interact with *Hadoop* from the command line. The next place to look at is the `conf` directory.

Java Setup

Make sure that you set `JAVA_HOME` in the `conf/hadoop-env.sh` script, or you will get an error when you start *Hadoop*. For example:

```
# The java implementation to use.  Required.
# export JAVA_HOME=/usr/lib/j2sdk1.5-sun
export JAVA_HOME=/usr/lib/jdk1.6.0_45
```

Tip

When using *Mac OS X* you can determine the *Java 6* home directory by executing `$ /usr/libexec/java_home -v 1.6`

Important

When using *MAC OS X* (Other systems possible also) you may still encounter `Unable to load realm info from SCDynamicStore` (For details see Hadoop Jira HADOOP-7489). In that case, please also add to `conf/hadoop-env.sh` the following line: `export HADOOP_OPTS="-Djava.security.krb5.realm= -Djava.security.krb5.kdc="`.

Setup SSH

As described in the installation guide, you also need to set up SSH login to `localhost` without a passphrase. On Linux, you may need to install the `ssh` package and ensure the `sshd` daemon is running. On Mac OS X, ssh is already installed but the `sshd` daemon isn't usually running. To start it, you need to enable "Remote Login" in the "Sharing" section of the control panel. Then you can carry on and setup SSH keys as described in the installation guide:

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Make sure you can log in at the command line using `ssh localhost` before trying to start *Hadoop*:

```
$ ssh localhost
Last login: Thu May 30 12:52:47 2013
```

You also need to decide where in your local filesystem you want *Hadoop* to store its data. Let's say you decide to use `/data`.

First create the directory and make sure it is writeable:

```
$ mkdir /data
$ chmod 777 /data
```

Now edit `conf/core-site.xml` and add the following property:

```
<property>
    <name>hadoop.tmp.dir</name>
    <value>/data</value>
</property>
```

You're then ready to format the filesystem for use by HDFS

```
$ bin/hadoop namenode -format
```

Setting the Namenode Port

By default Spring XD will use a *Namenode* setting of `hdfs://localhost:8020` which is defined in `${xd.home}/config/hadoop.properties`, depending on the used *Hadoop* distribution and version the by-default-defined port `8020` may be different, e.g. port `9000`. Therefore, please ensure you have the following setting in `conf/core-site.xml`:

```
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:8020</value>
    </property>
</configuration>
```

Further Configuration File Changes

In `conf/hdfs-site.xml` add:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

In `conf/mapred-site.xml` add:

```
<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>localhost:9001</value>
    </property>
</configuration>
```

A.2 Running Hadoop

You should now finally be ready to run *Hadoop*. Run the `start-all.sh` script

```
$ bin/start-all.sh
```

You should see five Hadoop Java processes running:

```
$ jps
4039 TaskTracker
3713 NameNode
3802 DataNode
3954 JobTracker
3889 SecondaryNameNode
4061 Jps
```

Try a few commands with `hadoop dfs` to make sure the basic system works

```
$ bin/hadoop dfs -ls /
Found 1 items
drwxr-xr-x   - luke supergroup          0 2013-05-30 17:28 /data
```

```
$ bin/hadoop dfs -mkdir /test
$ bin/hadoop dfs -ls /
Found 2 items
drwxr-xr-x   - luke supergroup          0 2013-05-30 17:28 /data
drwxr-xr-x   - luke supergroup          0 2013-05-30 17:31 /test
```

```
$ bin/hadoop dfs -rmr /test
Deleted hdfs://localhost:8020/test
```

Lastly, you can also browse the web interface for *NameNode* and *JobTracker* at:

- NameNode: http://localhost:50070/
- JobTracker: http://localhost:50030/

At this point you should be good to create a *Spring XD* stream using a *Hadoop* sink.

Appendix B. Creating a Source Module

B.1 Introduction

As outlined in the modules document, XD currently supports 3 types of modules: source, sink, and processor. This document walks through creation of a custom source module.

The first module in a stream is always a source. Source modules are built with Spring Integration and are typically very fine-grained. A module of type *source* is responsible for placing a message on a channel named *output*. This message can then be consumed by the other processor and sink modules in the stream. A source module is typically fed data by an inbound channel adapter, configured with a poller.

Spring Integration provides a number of adapters out of the box to support various transports, such as JMS, File, HTTP, Web Services, Mail, and more. You can typically create a source module that uses these inbound channel adapters by writing just a single Spring application context file.

These steps will demonstrate how to create and deploy a source module using the Spring Integration Feed Inbound Channel Adapter.

B.2 Create the module Application Context file

Create the Inbound Channel Adapter in a file called *feed.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:int="http://www.springframework.org/schema/integration"
 xmlns:int-feed="http://www.springframework.org/schema/integration/feed"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd
  http://www.springframework.org/schema/integration/feed
  http://www.springframework.org/schema/integration/feed/spring-integration-feed.xsd">

 <int-feed:inbound-channel-adapter  channel="output" url="http://feeds.bbci.co.uk/news/
rss.xml">
   <int:poller fixed-rate="5000" max-messages-per-poll="100" />
 </int-feed:inbound-channel-adapter>

 <int:channel id="output"/>
</beans>
```

The adapter is configured to poll the BBC News Feed every 5 seconds. Once an item is found, it will create a message with a SyndEntryImpl domain object payload and write it to a message channel called *output*. The name *output* should be used by convention so that your source module can easily be combined with any processor and sink module in a stream.

Make the module configurable

Users may want to pull data from feeds other than BBC News. Spring XD will automatically make a PropertyPlaceholderConfigurer available to your application context. You can simply reference property names and users can then pass in values when creating a stream using the DSL.

```
<int-feed:inbound-channel-adapter  channel="output" url="${url:http://feeds.bbci.co.uk/
news/rss.xml}">
  <int:poller fixed-rate="5000" max-messages-per-poll="100" />
</int-feed:inbound-channel-adapter>
```

Now users can optionally pass a *url* property value on stream creation. If not present, the specified default will be used.

B.3 Test the module locally

This section covers setup of a local project containing some code for testing outside of an XD container. This step can be skipped if you prefer to test the module by deploying to Spring XD.

Create a project

The module can be tested by writing a Spring integration test to load the context file and validate that news items are received. In order to write the test, you will need to create a project in an IDE such as STS, Eclipse, or IDEA. Eclipse will be used for this example.

Create a *feed* directory and add *feed.xml* to *src/main/resources*. Add the following *build.gradle* (or an equivalent *pom.xml*) to the root directory:

```
description = 'Feed Source Module'
group = 'org.springframework.xd.samples'

repositories {
  maven { url "http://repo.springsource.org/libs-snapshot" }
  maven { url "http://repo.springsource.org/plugins-release" }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'

ext {
    junitVersion = '4.11'
    springVersion = '3.2.2.RELEASE'
    springIntegrationVersion = '3.0.0.M2'
}

dependencies {
    compile("org.springframework:spring-core:$springVersion")
    compile "org.springframework:spring-context-support:$springVersion"
    compile "org.springframework.integration:spring-integration-feed:
$springIntegrationVersion"

    // Testing
    testCompile "junit:junit:$junitVersion"
    testCompile "org.springframework:spring-test:$springVersion"
}

defaultTasks 'build'
```

Run *gradle eclipse* to generate the Eclipse project. Import the project into Eclipse.

Create the Spring integration test

The main objective of the test is to ensure that news items are received once the module's Application Context is loaded. This can be tested by adding an Outbound Channel Adapter that will direct items to a POJO that can store them for validation.

Add the following *src/test/resources/org/springframework/xd/samples/test-context.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:int="http://www.springframework.org/schema/integration"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd">

 <context:property-placeholder/>

 <int:outbound-channel-adapter channel="output" ref="target" method="add" />

 <bean id="target" class="org.springframework.xd.samples.FeedCache" />

</beans>
```

This context creates an Outbound Channel Adapter that will subscribe to all messages on the *output* channel and pass the message payload to the *add* method of a *FeedCache* object. The context also creates the PropertyPlaceholderConfigurer that is ordinarily provided by the XD container.

Create the *src/test/java/org/springframework/xd/samples/FeedCache* class:

```
package org.springframework.xd.samples;
import ...

public class FeedCache {

 final BlockingDeque<SyndEntry> entries = new LinkedBlockingDeque<SyndEntry>(99);

 public void add(SyndEntry entry) {
  entries.add(entry);
 }
}
```

The *FeedCache* places all received SyndEntry objects on a *BlockingDeque* that our test can use to validate successful routing of messages.

Lastly, create and run the *src/test/java/org/springframework/xd/samples/FeedSourceModuleTest*:

```
package org.springframework.xd.samples;
import ...

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:feed.xml", "test-context.xml"})
public class FeedSourceModuleTest {

 @Autowired
 FeedCache feedCache;

 @Test
 public void testFeedPolling() throws Exception {
  assertNotNull(feedCache.entries.poll(5, TimeUnit.SECONDS));
 }

}
```

The test will load an Application Context using our feed and test context files. It will fail if a item is not placed into the FeedCache within 5 seconds.

You now have a way to build and test your new module independently. Time to deploy to Spring XD!

B.4 Deploy the module

Spring XD looks for modules in the ${xd.home}/modules directory. The modules directory organizes module types in sub-directories. So you will see something like:

```
modules/processor
modules/sink
modules/source
```

Simply drop *feed.xml* into the *modules/source* directory and add the dependencies to the lib directory. For now, all module dependencies need to be added to ${xd.home}/lib. Future versions of Spring XD will provide a more elegant module packaging approach. Copy the following jars from your gradle cache to ${xd.home}/lib:

```
spring-integration-feed-3.0.0.M2.jar
jdom-1.0.jar
rome-1.0.0.jar
rome-fetcher-1.0.0.jar
```

Now fire up the server. See Getting Started to learn how to start the Spring XD server.

B.5 Test the deployed module

Once the XD server is running, create a stream to test it out. This stream will write SyndEntry objects to the XD log:

```
xd:> stream create --name feedtest --definition "feed | log"
```

You should start seeing messages like the following in the container console window:

```
   WARN logger.feedtest: SyndEntryImpl.contributors=[]
SyndEntryImpl.contents=[]
SyndEntryImpl.updatedDate=null
SyndEntryImpl.link=http://www.bbc.co.uk/news/uk-22850006#sa-
ns_mchannel=rss&ns_source=PublicRSS20-sa
SyndEntryImpl.titleEx.value=VIDEO: Queen visits Prince Philip in hospital
...
```

As you can see, the *SyndEntryImpl* toString is fairly verbose. To make the output more concise, create a processor module to further transform the SyndEntry or consider converting the entry to JSON and using the JSON Field Extractor to send a single attribute value to the output channel.

Appendix C. Creating a Processor Module

C.1 Introduction

As outlined in the modules document, XD currently supports 3 types of modules: source, sink, and processor. This document walks through creation of a custom processor module.

One or more processors can be included in a stream definition to modifythe data as it passes between the inital source and the destination sink. The architecture section covers the basics of processors modules provided out of the box are covered in the processors section.

Here we'll look at how to create and deploy a custom processor module to transform the input from an incoming `twittersearch`. The steps are essentially the same for any source though. Rather than using built-in functionality, we'll write a custom processor implementation class and wire it up using Spring Integration.

C.2 Write the Transformer Code

The tweet messages from `twittersearch` contain quite a lot of data (id, author, time and so on). The transformer we'll write will discard everything but the text content and output this as a string. The output messages from the `twittersearch` source are also strings, containing the tweet data as JSON. We first parse this into a map using Jackson library code, then extract the "text" field from the map.

```
package custom;

import java.io.IOException;
import java.util.Map;

import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.type.TypeReference;
import org.springframework.integration.transformer.MessageTransformationException;

public class TweetTransformer {
  private ObjectMapper mapper = new ObjectMapper();

  public String transform(String payload) {
    try {
      Map<String, Object> tweet = mapper.readValue(payload, new TypeReference<Map<String,
 Object>>() {});
      return tweet.get("text").toString();
    } catch (IOException e) {
      throw new MessageTransformationException("Unable to transform tweet: " +
 e.getMessage(), e);
    }
  }
}
```

C.3 Create the module Application Context File

Create the following file as *tweettransformer.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns="http://www.springframework.org/schema/integration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd">
  <channel id="input"/>

  <transformer input-channel="input" output-channel="output">
    <beans:bean class="custom.TweetTransformer" />
  </transformer>

  <channel id="output"/>
</beans:beans>
```

C.4 Deploy the Module

To deploy the module, you need to copy the *tweettransformer.xml* file to the `${xd.home}/modules/processors` directory. We also need to make the custom module code available. Currently Spring XD looks for code in the jars it finds in the `${xd.home}/lib` directory. So create a jar with the `TweetTransformer` class in it (and the correct package structure) and drop it into `lib`.

C.5 Test the deployed module

Start the XD server and try creating a stream to test your processor:

```
xd:> stream create --name javatweets --definition "twittersearch --query=java --
consumerKey=<your_key> --consumerSecret=<your_secret> | tweettransformer | file"
```

If you haven't already used `twittersearch`, read the sources section for more details. This command should stream tweets to the file `/tmp/xd/output/javatweets` but, unlike the normal `twittersearch` output, you should just see the plain tweet text there, rather than the full JSON data.

Appendix D. Creating a Sink Module

D.1 Introduction

As outlined in the modules document, XD currently supports 3 types of modules: source, sink, and processor. This document walks through creation of a custom sink module.

The last module in a stream is always a sink. Sink modules are built with Spring Integration and are typically very fine-grained. A module of type *sink* listens on a channel named *input* and is responsible for outputting received messages to an external resource to terminate the stream.

Spring Integration provides a number of adapters out of the box to support various transports, such as JMS, File, HTTP, Web Services, Mail, and more. You can typically create a sink module that uses these outbound channel adapters by writing just a single Spring application context file.

These steps will demonstrate how to create and deploy a sink module using the Spring Integration RedisStore Outbound Channel Adapter.

D.2 Create the module Application Context file

Create the Outbound Channel Adapter in a file called *redis-store.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:int="http://
www.springframework.org/schema/integration"
 xmlns:int-redis="http://www.springframework.org/schema/integration/redis"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd
  http://www.springframework.org/schema/integration/redis
  http://www.springframework.org/schema/integration/redis/spring-integration-redis.xsd">

 <int:channel id="input" />

 <int-redis:store-outbound-channel-adapter
  id="redisListAdapter" collection-type="LIST" channel="input" key="myCollection" />

 <bean id="redisConnectionFactory"
  class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory">
  <constructor-arg index="0" value="${localhost}" />
  <constructor-arg index="1" value="${6379}" />
 </bean>

</beans>
```

The adapter is configured to listen on a channel named *input*. The name *input* should be used by convention so that your sink module will receive all messages sent in the stream. Once a message is received, it will write the payload to a Redis list with key *myCollection*. By default, the RedisStore Outbound Channel Adapter uses a bean named *redisConnectionFactory* to connect to the Redis server.

Note

By default, the adapter uses a *StringRedisTemplate*. Therefore, this module will store all payloads directly as Strings. Create a custom *RedisTemplate* with different value Serializers to serialize other forms of data like Java objects to the Redis collection.

D.3 Make the module configurable

Users may want to specify a different Redis server or key to use for storing data. Spring XD will automatically make a PropertyPlaceholderConfigurer available to your application context. You can simply reference property names and users can then pass in values when creating a stream using the DSL

```
        <int-redis:store-outbound-channel-adapter
  id="redisListAdapter" collection-type="LIST" channel="input" key="${key:myCollection}" /
>

 <bean id="redisConnectionFactory"
  class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory">
  <constructor-arg index="0" value="${hostname:localhost}" />
  <constructor-arg index="1" value="${port:6379}" />
 </bean>
```

Now users can optionally pass *key*, *hostname*, and *port* property values on stream creation. If not present, the specified defaults will be used.

D.4 Test the module locally

This section covers setup of a local project containing some code for testing outside of an XD container. This step can be skipped if you prefer to test the module by deploying to Spring XD.

Create a project

The module can be tested by writing a Spring integration test to load the context file and validate that messages are stored in Redis. In order to write the test, you will need to create a project in an IDE such as STS, Eclipse, or IDEA. Eclipse will be used for this example.

Create a *redis-store* directory and add *redis-store.xml* to *src/main/resources*. Add the following *build.gradle* (or an equivalent *pom.xml*) to the root directory:

```
description = 'Redis Store Sink Module'
group = 'org.springframework.xd.samples'

repositories {
  maven { url "http://repo.springsource.org/libs-snapshot" }
  maven { url "http://repo.springsource.org/plugins-release" }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'

ext {
    junitVersion = '4.11'
    lettuceVersion = '2.3.2'
    springVersion = '3.2.2.RELEASE'
    springIntegrationVersion = '3.0.0.M2'
    springSocialVersion = '1.0.1.RELEASE'
    springDataRedisVersion = '1.0.4.RELEASE'
}

dependencies {
    compile("org.springframework:spring-core:$springVersion")
    compile "org.springframework:spring-context-support:$springVersion"
    compile "org.springframework.integration:spring-integration-core:
$springIntegrationVersion"
    compile "org.springframework.integration:spring-integration-redis:
$springIntegrationVersion"
    compile "org.springframework.data:spring-data-redis:$springDataRedisVersion"

    // Testing
    testCompile "junit:junit:$junitVersion"
    testCompile "org.springframework:spring-test:$springVersion"
    testCompile "com.lambdaworks:lettuce:$lettuceVersion"
}

defaultTasks 'build'
```

Run *gradle eclipse* to generate the Eclipse project. Import the project into Eclipse.

Create the Spring integration test

The main objective of the test is to ensure that messages are stored in a Redis list once the module's Application Context is loaded. This can be tested by adding an Inbound Channel Adapter that will direct test messages to the *input* channel.

Add the following *src/test/resources/org/springframework/xd/samples/test-context.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:int="http://
www.springframework.org/schema/integration"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd">

 <context:property-placeholder />

 <int:inbound-channel-adapter channel="input" expression="'TESTING'">
  <int:poller fixed-rate="1000" />
 </int:inbound-channel-adapter>

 <bean id="redisTemplate" class="org.springframework.data.redis.core.StringRedisTemplate">
  <property name="connectionFactory" ref="redisConnectionFactory" />
 </bean>

</beans>
```

This context creates an Inbound Channel Adapter that will generate messages with the payload "TESTING". The context also creates the PropertyPlaceholderConfigurer that is ordinarily provided by the XD container. The *redisTemplate* is configured for use by the test to verify that data is placed in Redis.

Lastly, create and run the *src/test/java/org/springframework/xd/samples/RedisStoreSinkModuleTest*:

```
package org.springframework.xd.samples;
import ...

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:redis-store.xml", "test-context.xml"})
public class RedisStoreSinkModuleTest {

 @Autowired
 RedisTemplate<String,String> redisTemplate;

 @Test
 public void testTweetSearch() throws Exception {
      assertNotNull(redisTemplate.boundListOps("myCollection").leftPop(5,
 TimeUnit.SECONDS));
 }
}
```

The test will load an Application Context using our redis-store and test context files. It will fail if an item is not placed in the Redis list within 5 seconds.

Run the test

The test requires a running Redis server. See Getting Started for information on installing and starting Redis.

You now have a way to build and test your new module independently. Time to deploy to Spring XD!

D.5 Deploy the module

Spring XD looks for modules in the ${xd.home}/modules directory. The modules directory organizes module types in sub-directories. So you will see something like:

```
modules/processor
modules/sink
modules/source
```

Simply drop *redis-store.xml* into the *modules/sink* directory and fire up the server. See Getting Started to learn how to start the Spring XD server.

D.6 Test the deployed module

Once the XD server is running, create a stream to test it out. This stream will write tweets containing the word "java" to Redis as a JSON string:

```
xd:> stream create --name javasearch --definition "twittersearch --consumerKey=<your_key>
 --consumerSecret=<your_secret> --query=java | redis-store --key=javatweets"
```

Note that you need to have a consumer key and secret to use the `twittersearch` module. See the description in the streams section for more information.

Fire up the redis-cli and verify that tweets are being stored:

```
$ redis-cli
redis 127.0.0.1:6379> lrange javatweets 0 -1
1) {\"id\":342386150738120704,\"text\":\"Now Hiring: Senior Java Developer\",\"createdAt
\":1370466194000,\"fromUser\":\"jencompgeek\",...\"}"
```

Appendix E. Building Spring XD

E.1 Instructions

Here are some useful steps to build and run Spring XD.

To build all sub-projects and run tests for Spring XD:

```
./gradlew build
```

To build and bundle the distribution of Spring XD

```
./gradlew dist
```

The above gradle task creates spring-xd-<version>.zip binary distribution archive and spring-xd-<version>-docs.zip documentation archive files under *build/distributions*. This will also create a *build/dist/spring-xd* directory which is the expanded version of the binary distribution archive.

To just create the Spring XD expanded binary distribution directory

```
./gradlew copyInstall
```

The above gradle task creates the distribution directory under *build/dist/spring-xd*.

Once the binary distribution directory is created, please refer to Getting Started on how to run Spring XD.

E.2 IDE support

If you would like to work with the Spring XD code in your IDE, please use the following project generation depending on the IDE you use:

For Eclipse/Spring Tool Suite

```
./gradlew eclipse
```

For IntelliJ IDEA

```
./gradlew idea
```

Then just import the project as an existing project.

E.3 Running JavaScript UI Tests

In order to run the UI tests, `phantomjs` must be installed on your system. On a Mac using brew execute:

```
brew install phantomjs
```

Alternatively, you can also install `phantomjs` using the Node Package Manager:

```
npm install -g phantomjs
```

Afterwards, execute:

```
./gradlew jsTest
```

As a result you should see console output similar to the following:

```
:spring-xd-ui:jsTest
Opening: http://localhost:9393/admin-ui/test/SpecRunnerPhantomJS.html
Starting...

Finished
-----------------
8 specs, 0 failures in 0.073s.

ConsoleReporter finished: success
Stopped the SingleNode server.

BUILD SUCCESSFUL
```

Appendix F. XD Shell Command Reference

F.1 Base Commands

admin config server

Configure the XD admin server to use.

```
admin config server [[--uri] <uri>]
```

uri
the location of the XD Admin REST endpoint. **(default: `http://localhost:9393/`)**

admin config info

Show the XD admin server being used.

```
admin config info
```

F.2 Runtime Commands

runtime containers

List runtime containers.

```
runtime containers
```

runtime modules

List runtime modules.

```
runtime modules [[--containerId] <containerId>]
```

containerId
to filter by container id.

F.3 Stream Commands

stream create

Create a new stream definition.

```
stream create [--name] <name> --definition <definition> [--deploy <deploy>]
```

name
the name to give to the stream. **(required)**

definition
a stream definition, using XD DSL (e.g. "http --port=9000 | hdfs"). **(required)**

deploy
whether to deploy the stream immediately. **(default: `true`)**

stream destroy

Destroy an existing stream.

```
stream destroy [--name] <name>
```

name
the name of the stream to destroy. **(required)**

stream all destroy

Destroy all existing streams.

```
stream all destroy [--force [<force>]]
```

force
bypass confirmation prompt. **(default: `false`, or `true` if `--force` is specified without a value)**

stream deploy

Deploy a previously created stream.

```
stream deploy [--name] <name>
```

name
the name of the stream to deploy. **(required)**

stream all deploy

Deploy all previously created stream.

```
stream all deploy [--force [<force>]]
```

force
bypass confirmation prompt. **(default: `false`, or `true` if `--force` is specified without a value)**

stream undeploy

Un-deploy a previously deployed stream.

```
stream undeploy [--name] <name>
```

name
the name of the stream to un-deploy. **(required)**

stream all undeploy

Un-deploy all previously deployed stream.

```
stream all undeploy [--force [<force>]]
```

force
bypass confirmation prompt. **(default: `false`, or `true` if `--force` is specified without a value)**

stream list

List created streams.

```
stream list
```

F.4 Job Commands

job create

Create a job.

```
job create [--name] <name> --definition <definition> [--deploy <deploy>] [--dateFormat
 <dateFormat>] [--numberFormat <numberFormat>] [--makeUnique <makeUnique>]
```

name
the name to give to the job. **(required)**

definition
job definition using xd dsl . **(required)**

deploy
whether to deploy the stream immediately. **(default: `true`)**

dateFormat
the optional date format for job parameters.

numberFormat
the optional number format for job parameters.

makeUnique
shall job parameters be made unique?. **(default: `false`)**

job list

List all jobs.

```
job list
```

job deploy

Deploy a previously created job.

```
job deploy [--name] <name>
```

name
the name of the job to deploy. **(required)**

job all deploy

Deploy previously created job(s).

```
job all deploy [--force [<force>]]
```

force
bypass confirmation prompt. **(default: `false`, or `true` if `--force` is specified without a value)**

job launch

Launch previously deployed job.

```
job launch [[--name] <name>] [--params <params>]
```

name
the name of the job to deploy.

params
the parameters for the job. **(default: ``)**

job undeploy

Un-deploy an existing job.

```
job undeploy [--name] <name>
```

name
the name of the job to un-deploy. **(required)**

job all undeploy

Un-deploy all existing jobs.

```
job all undeploy [--force [<force>]]
```

force
bypass confirmation prompt. **(default: `false`, or `true` if `--force` is specified without a value)**

job destroy

Destroy an existing job.

```
job destroy [--name] <name>
```

name
the name of the job to destroy. **(required)**

job all destroy

Destroy all existing jobs.

```
job all destroy [--force [<force>]]
```

force
bypass confirmation prompt. **(default: `false`, or `true` if `--force` is specified without a value)**

F.5 Module Commands

module compose

Create a virtual module.

```
module compose [--name] <name> --definition <definition>
```

name
the name to give to the module. **(required)**

definition
module definition using xd dsl. **(required)**

module delete

Delete a virtual module.

```
module delete [--name] <name> --type <type>
```

name
the name of the the module. **(required)**

type
the type of the module. **(required)**

module display

Display the configuration file of a module.

```
module display [--name] <name> --type <type>
```

name
the name of the the module. **(required)**

type
the type of the module. **(required)**

module list

List all modules.

```
module list [--type <type>]
```

type
retrieve a specific type of module.

F.6 Metrics Commands

counter list

List all available counter names.

```
counter list
```

counter delete

Delete the counter with the given name.

```
counter delete [--name] <name>
```

name
the name of the counter to delete. **(required)**

counter display

Display the value of a counter.

```
counter display [--name] <name> [--pattern <pattern>]
```

name
the name of the counter to display. **(required)**

pattern
the pattern used to format the value (see DecimalFormat). **(default: `<use platform locale>`)**

fieldvaluecounter list

List all available field-value-counter names.

```
fieldvaluecounter list
```

fieldvaluecounter delete

Delete the field-value-counter with the given name.

```
fieldvaluecounter delete [--name] <name>
```

name
the name of the field-value-counter to delete. **(required)**

fieldvaluecounter display

Display the value of a field-value-counter.

```
fieldvaluecounter display [--name] <name> [--pattern <pattern>] [--size <size>]
```

name
the name of the field-value-counter to display. **(required)**

pattern
the pattern used to format the field-value-counter's field count (see DecimalFormat). **(default: `<use platform locale>`)**

size
the number of values to display. **(default: 25)**

aggregatecounter list

List all available aggregate counter names.

```
aggregatecounter list
```

aggregatecounter delete

Delete an aggregate counter.

```
aggregatecounter delete [--name] <name>
```

name
the name of the aggregate counter to delete. **(required)**

aggregatecounter display

Display aggregate counter values by chosen interval and resolution(minute, hour).

```
aggregatecounter display [--name] <name> [--from <from>] [--to <to>] [--lastHours
 <lastHours>] [--lastDays <lastDays>] [--resolution <resolution>] [--pattern <pattern>]
```

name
the name of the aggregate counter to display. **(required)**

from
start-time for the interval. format: *yyyy-MM-dd HH:mm:ss*.

to
end-time for the interval. format: *yyyy-MM-dd HH:mm:ss*. defaults to now.

lastHours
set the interval to last *n* hours.

lastDays
set the interval to last *n* days.

resolution
the size of the bucket to aggregate (minute, hour, day, month). **(default: hour)**

pattern
the pattern used to format the count values (see DecimalFormat). **(default: `<use platform locale>`)**

gauge list

List all available gauge names.

```
gauge list
```

gauge delete

Delete a gauge.

```
gauge delete [--name] <name>
```

name
the name of the gauge to delete. **(required)**

gauge display

Display the value of a gauge.

```
gauge display [--name] <name> [--pattern <pattern>]
```

name
the name of the gauge to display. **(required)**

pattern
the pattern used to format the value (see DecimalFormat). **(default: `<use platform locale>`)**

richgauge list

List all available richgauge names.

```
richgauge list
```

richgauge delete

Delete the richgauge.

```
richgauge delete [--name] <name>
```

name
the name of the richgauge to delete. **(required)**

richgauge display

Display Rich Gauge value.

```
richgauge display [--name] <name> [--pattern <pattern>]
```

name
the name of the richgauge to display value. **(required)**

pattern
the pattern used to format the richgauge value (see DecimalFormat). **(default: `<use platform locale>`)**

F.7 Http Commands

http post

POST data to http endpoint.

```
http post [[--target] <target>] [--data <data>] [--file <file>] [--contentType
 <contentType>]
```

target
the location to post to. **(default: `http://localhost:9000`)**

data
the text payload to post. exclusive with file. embedded double quotes are not supported if next to a space character.

file
filename to read data from. exclusive with data.

contentType
the content-type to use. file is also read using the specified charset. **(default: `text/plain; Charset=UTF-8`)**

F.8 Hadoop Configuration Commands

hadoop config props set

Sets the value for the given Hadoop property.

```
hadoop config props set [--property] <property>
```

property
what to set, in the form <name=value>. **(required)**

hadoop config props get

Returns the value of the given Hadoop property.

```
hadoop config props get [--key] <key>
```

key
property name. **(required)**

hadoop config info

Returns basic info about the Hadoop configuration.

```
hadoop config info
```

hadoop config load

Loads the Hadoop configuration from the given resource.

```
hadoop config load [--location] <location>
```

location
configuration location (can be a URL). **(required)**

hadoop config props list

Returns (all) the Hadoop properties.

```
hadoop config props list
```

hadoop config fs

Sets the Hadoop namenode.

```
hadoop config fs [--namenode] <namenode>
```

namenode
namenode address - can be local|<namenode:port>. **(required)**

hadoop config jt

Sets the Hadoop job tracker.

```
hadoop config jt [--jobtracker] <jobtracker>
```

jobtracker
job tracker address - can be local|<jobtracker:port>. **(required)**

F.9 Hadoop FileSystem Commands

hadoop fs get

Copy files to the local file system.

```
hadoop fs get --from <from> --to <to> [--ignoreCrc [<ignoreCrc>]] [--crc [<crc>]]
```

from
source file names. **(required)**

to
destination path name. **(required)**

ignoreCrc
whether ignore CRC. **(default: `false`, or `true` if `--ignoreCrc` is specified without a value)**

crc
whether copy CRC. **(default: `false`, or `true` if `--crc` is specified without a value)**

hadoop fs put

Copy single src, or multiple srcs from local file system to the destination file system.

```
hadoop fs put --from <from> --to <to>
```

from
source file names. **(required)**

to
destination path name. **(required)**

hadoop fs count

Count the number of directories, files, bytes, quota, and remaining quota.

```
hadoop fs count [--quota [<quota>]] --path <path>
```

quota
whether with quta information. **(default: `false`, or `true` if `--quota` is specified without a value)**

path
path name. **(required)**

hadoop fs mkdir

Create a new directory.

```
hadoop fs mkdir [--dir] <dir>
```

dir
directory name. **(required)**

hadoop fs tail

Display last kilobyte of the file to stdout.

```
hadoop fs tail [--file] <file> [--follow [<follow>]]
```

file
file to be tailed. **(required)**

follow
whether show content while file grow. **(default: `false`, or `true` if `--follow` is specified without a value)**

hadoop fs ls

List files in the directory.

```
hadoop fs ls [[--dir] <dir>] [--recursive [<recursive>]]
```

dir
directory to be listed. **(default: `.`)**

recursive
whether with recursion. **(default: `false`, or `true` if `--recursive` is specified without a value)**

hadoop fs cat

Copy source paths to stdout.

```
hadoop fs cat [--path] <path>
```

path
file name to be shown. **(required)**

hadoop fs chgrp

Change group association of files.

```
hadoop fs chgrp [--recursive [<recursive>]] --group <group> [--path] <path>
```

recursive
whether with recursion. **(default: `false`, or `true` if `--recursive` is specified without a value)**

group
group name. **(required)**

path
path of the file whose group will be changed. **(required)**

hadoop fs chown

Change the owner of files.

```
hadoop fs chown [--recursive [<recursive>]] --owner <owner> [--path] <path>
```

recursive
whether with recursion. **(default: `false`, or `true` if `--recursive` is specified without a value)**

owner
owner name. **(required)**

path
path of the file whose ownership will be changed. **(required)**

hadoop fs chmod

Change the permissions of files.

```
hadoop fs chmod [--recursive [<recursive>]] --mode <mode> [--path] <path>
```

recursive
whether with recursion. **(default: `false`, or `true` if `--recursive` is specified without a value)**

mode
permission mode. **(required)**

path
path of the file whose permissions will be changed. **(required)**

hadoop fs copyFromLocal

Copy single src, or multiple srcs from local file system to the destination file system. Same as put.

```
hadoop fs copyFromLocal --from <from> --to <to>
```

from
source file names. **(required)**

to
destination path name. **(required)**

hadoop fs moveFromLocal

Similar to put command, except that the source localsrc is deleted after it's copied.

```
hadoop fs moveFromLocal --from <from> --to <to>
```

from
source file names. **(required)**

to
destination path name. **(required)**

hadoop fs copyToLocal

Copy files to the local file system. Same as get.

```
hadoop fs copyToLocal --from <from> --to <to> [--ignoreCrc [<ignoreCrc>]] [--crc [<crc>]]
```

from
source file names. **(required)**

to
destination path name. **(required)**

ignoreCrc
whether ignore CRC. **(default: `false`, or `true` if `--ignoreCrc` is specified without a value)**

crc
whether copy CRC. **(default: `false`, or `true` if `--crc` is specified without a value)**

hadoop fs copyMergeToLocal

Takes a source directory and a destination file as input and concatenates files in src into the destination local file.

```
hadoop fs copyMergeToLocal --from <from> --to <to> [--endline [<endline>]]
```

from
source file names. **(required)**

to
destination path name. **(required)**

endline
whether add a newline character at the end of each file. **(default: `false`, or `true` if `--endline` is specified without a value)**

hadoop fs cp

Copy files from source to destination. This command allows multiple sources as well in which case the destination must be a directory.

```
hadoop fs cp --from <from> --to <to>
```

from
source file names. **(required)**

to
destination path name. **(required)**

hadoop fs mv

Move source files to destination in the HDFS.

```
hadoop fs mv --from <from> --to <to>
```

from
source file names. **(required)**

to
destination path name. **(required)**

hadoop fs du

Displays sizes of files and directories contained in the given directory or the length of a file in case its just a file.

```
hadoop fs du [[--dir] <dir>] [--summary [<summary>]]
```

dir
directory to be listed. **(default: `.`)**

summary
whether with summary. **(default: `false`, or `true` if `--summary` is specified without a value)**

hadoop fs expunge

Empty the trash.

```
hadoop fs expunge
```

hadoop fs rm

Remove files in the HDFS.

```
hadoop fs rm [[--path] <path>] [--skipTrash [<skipTrash>]] [--recursive [<recursive>]]
```

path
path to be deleted. **(default: `.`)**

skipTrash
whether to skip trash. **(default: `false`, or `true` if `--skipTrash` is specified without a value)**

recursive
whether to recurse. **(default: `false`, or `true` if `--recursive` is specified without a value)**

hadoop fs setrep

Change the replication factor of a file.

```
hadoop fs setrep --path <path> --replica <replica> [--recursive [<recursive>]] [--waiting
 [<waiting>]]
```

path
path name. **(required)**

replica
source file names. **(required)**

recursive
whether with recursion. **(default: `false`, or `true` if `--recursive` is specified without a value)**

waiting
whether wait for the replic number is eqal to the number. **(default: `false`, or `true` if `--waiting` is specified without a value)**

hadoop fs text

Take a source file and output the file in text format.

```
hadoop fs text [--file] <file>
```

file
file to be shown. **(required)**

hadoop fs touchz

Create a file of zero length.

```
hadoop fs touchz [--file] <file>
```

file
file to be touched. **(required)**