Spring XD Guide

1.0.0

Mark Fisher , Mark Pollack , David Turanski , Gunnar Hillert , Eric Bottard , Patrick Peralta, Gary
Russell , Ilayaperumal Gopinathan , Jennifer Hickey , Michael Minella , Luke Taylor , Thomas Risberg ,
Glenn Renfro , Janne Valkealahti , Thomas Darimont , Dave Syer , Jon Brisbin , Andy Clement

Table of Contents

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

To download the current release, you can download the distribution **spring-xd-1.0.0.RELEASE-dist.zip** and its accompanying documentation.

If you want to try out the latest build of Spring XD, You can download the snapshot distribution 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 stable release.

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.

```
$ cd spring-xd-1.0 (wiki)
```

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.

You can also install Spring XD using homebrew on OSX and yum on RedHat/CentOS.

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 (wiki) | Admin Server Target: http://localhost:9393
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 can then use the `admin config server <url>` to attempt to reconnect to the admin REST endpoint once you've figured out what went wrong:

```
admin config server http://localhost:9393
```

You should now be able to start using Spring XD.

Tip

Spring XD uses ZooKeeper internally which typically runs as an external process. XD singlenode runs with an embedded ZooKeeper server and assigns a random available port. This keeps things very simple. However if you already have a ZooKeeper ensemble set up and want to connect to it, you can edit `xd/config/servers.yml`:

```
#Zookeeper properties
# client connect string: host1:port1,host2:port2,...,hostN:portN
zk:
  client:
     connect: localhost:2181
```

Also, sometimes it is useful in troubleshooting to connect the ZooKeeper CLI to the embedded server. The assigned server port is listed in the console log, but you can also set the port directly by setting the property `zk.embedded.server.port` in `servers.yml` or set JAVA_OPTS before starting xd-singlenode.

```
$export JAVA_OPTS=-Dzk.embedded.server.port=<port>
```

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 definitions are built from a simple DSL. For example, execute:

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

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. Since the `--deploy` flag was provided, this stream will be deployed immediately. In the console 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 possible 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.

2.7 OSX Homebrew installation

If you are on a Mac and using homebrew, all you need to do to install *Spring XD* is:

```
$ brew tap pivotal/tap
$ brew install springxd
```

Homebrew will install `springxd` to `/usr/local/bin`. Now you can jump straight into using **Spring XD**:

```
$ xd-singlenode
```

Brew install also allows you to run *Spring XD* in distributed mode on you OSx. See Running Distributed Mode for details on setting up a distributed runtime.

2.8 RedHat/CentOS Installation

If you are using RHEL or CentOS you can install *Spring XD* using our yum repository.

```
wget -q -O - http://packages.gopivotal.com/pub/rpm/rhel5/app-suite/app-suite-installer |
 sh
yum install spring-xd
```

```
wget -q -O - http://packages.gopivotal.com/pub/rpm/rhel6/app-suite/app-suite-installer |
 sh
yum install spring-xd
```

This installs *Spring XD* and init.d services for managing Admin Server and Container runtimes. Before you can run Admin Server and Container you will need to install and start distributed components. See Running Distributed Mode for details on setting up a distributed runtime. After distributed component are configured, Admin Server and Container can be started as follows:

```
service spring-xd-admin start
service spring-xd-container start
```

You can configure arguments to spring-xd-admin and spring-xd-container scripts by setting them in /etc/sysconfig/spring-xd. For example to run spring-xd-container with transport=RabbitMQ update this property in /etc/sysconfig/spring-xd:

```
TRANSPORT=rabbit
```

To stop *Spring XD*

```
service spring-xd-admin stop
service spring-xd-container stop
```

2.9 Running in Distributed Mode

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.

The XD distributed runtime architecture consists of the following distributed components:

- Admin - Manages Stream and Job deployments and other end user operations and provides REST services to access runtime state, system metrics, and analytics
- Container - Hosts deployed Modules (stream processing tasks) and batch jobs
- ZooKeeper - Provides all runtime information for the XD cluster. Tracks running containers, in which containers modules and jobs are deployed, stream definitions, deployment manifests, and the like, see XD Distributed Runtime for an overview on how XD uses ZooKeeper.
- Spring Batch Job Repository Database - An RDBMS is required for jobs. The XD distribution comes with HSQLDB, but this is not appropriate for a production installation. XD supports any JDBC compliant database.

- A Message Broker - Used for data transport. XD data transport is designed to be pluggable. Currently XD supports Rabbit MQ and Redis for messaging during stream and job processing. A production installation must configure one of these transport options. Rabbit MQ is recommended as it is considered the more reliable of the two. In either case, a separate server must be running to provide the messaging middleware.
- Analytics Repository - XD currently uses Redis to store the counters and gauges provided Analytics

In addition, XD provides a Command Line Interface (CLI), XD Shell as well as a web application, XD-UI to interact with the XD runtime.

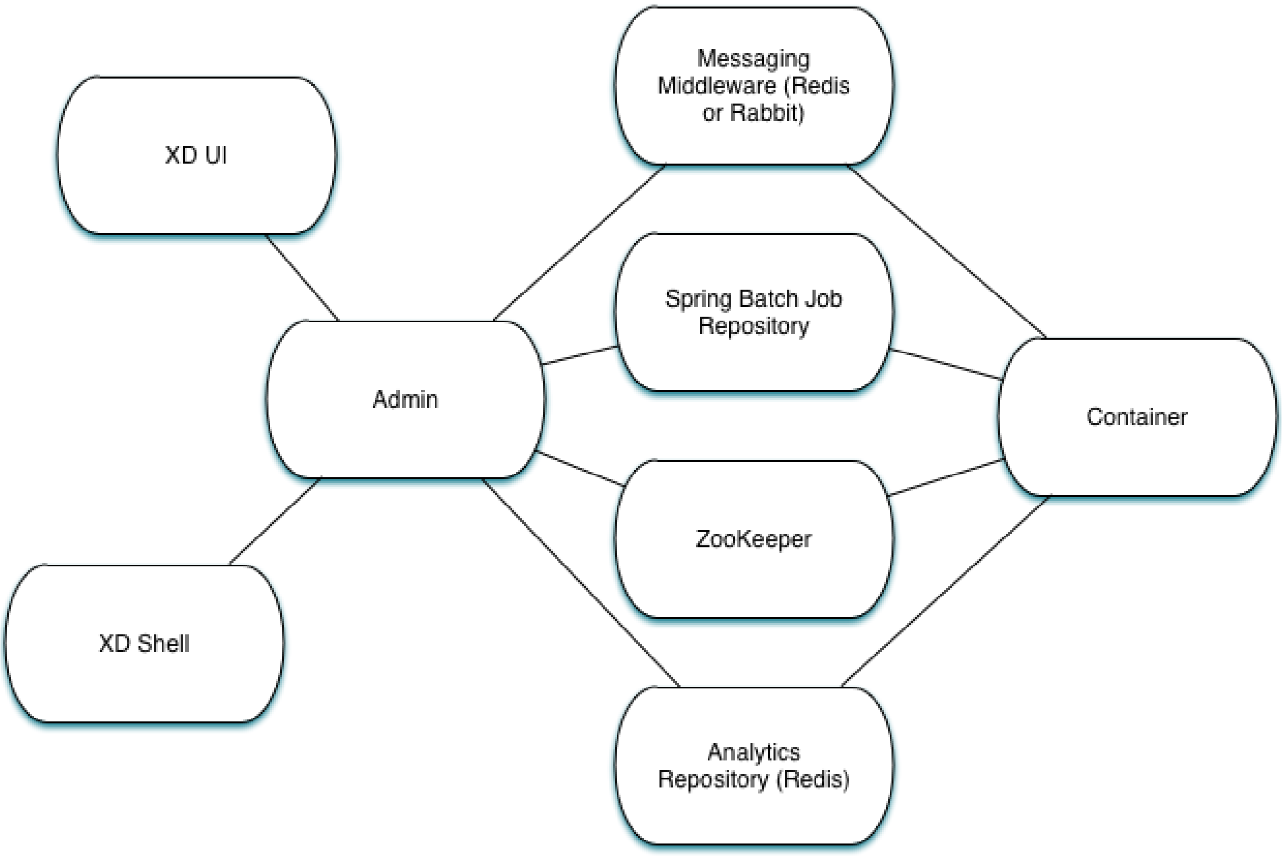

Messaging Middleware (Redis or Rabbit)
XD UI
Spring Batch Job Repository
Admin
Container
ZooKeeper
XD Shell
Analytics Repository (Redis)

Figure 2.1.

XD CommandLine Options

The XD distribution provides shell scripts to start its runtime components under the *xd* directory of the XD installation:

Whether you are running _xd-admin, xd-container or even xd-singlenode you can always get help by typing the command followed by --help. For example:

```
xd/bin/xd-admin --help


 _____                           __   _______
/  ___|          (-)             \ \ / /  _  \
\ `--. _ __  _ __ _ _ __   __ _   \ V /| | | |
 `--. \ '_ \| '__| | '_ \ / _` |  / ^ \| | | |
/\__/ / |_) | |  | | | | | (_| | / / \ \ |/ /
\____/| .__/|_|  |_|_| |_|\__, | \/   \/___/
      | |                  __/ |
      |_|                 |___/
1.0 (wiki)             eXtreme Data



Started : AdminServerApplication
Documentation: https://github.com/spring-projects/spring-xd/wiki

Usage:
 --analytics [redis]   : How to persist analytics such as counters and gauges
 --help (-?, -h)       : Show this help screen
 --httpPort <httpPort> : Http port for the REST API server
 --mgmtPort <mgmtPort> : The port for the management server
```

xd-admin command line args:

- **analytics** - The data store that will be used to store the analytics data. The default is **redis**
- **help** - Displays help for the command args. Help information may be accessed with a -? or -h.
- **httpPort** - The http port for the REST API server. Defaults to 9393.
- **mgmtPort** - The port for the management server. Defaults to the admin server port.

Also, note that it is recommended to use fixed http port for XDAdmin(s). This makes it easy to know the admin server addresses the REST clients (shell, webUI) can point to. If a random port is chosen (with server.port or $PORT set to 0), then one needs to go through the log and find which port admin server's tomcat starts at.

xd-container command line args:

- **analytics** - How to persist analytics such as counters and gauges. The default is **redis**
- **groups** - The assigned group membership for this container as a comma delimited list
- **hadoopDistro** - The Hadoop distribution to be used for HDFS access. HDFS is not available if not set.
- **help** - Displays help for the command args. Help information may be accessed with a -? or -h.
- **mgmtPort** - The port for the management server. Defaults to the container server port.

Setting up a RDBMS

The distributed runtime requires an RDBMS. The XD distrubution comes with an HSQLDB in memory database for testing purposes, but an alternate is expected. To start HSQLDB:

```
$ cd hsqldb/bin
$ ./hsqldb-server
```

To configure XD to connect to a different RDBMS, have a look at `xd/config/servers.yml` in the `spring:datasource` section for details. Note that `spring.batch.initializer.enabled` is set to true by default which will initialize the Spring Batch schema if it is not already set up. However, if those tables have already been created, they will be unaffected.

Setting up ZooKeeper

Currently XD does not ship with ZooKeeper. At the time of this writing, the compliant version is 3.4.6 and you can download it from here. Please refer to the ZooKeeper Getting Started Guide for more information. A ZooKeeper ensemble consisting of at least three members is recommended for production installations, but a single server is all that is needed to have XD up and running.

Setting up Redis

Redis is the default transport when running in distributed mode.

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 `servers.yml` 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) which is included in the .zip download. If you are installing using brew or rpm you should install Redis using those installers or download the source tarball and compile Redis yourself. If you used the .zip download then 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
```

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 5672**. The default account credentials of **guest/guest** are assumed. You can change that in the `servers.yml` 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 3.3.0. Copyright (C) 2007-2013 GoPivotal, 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 10 plugins.
```

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

Spring XD uses data transport for sending data from the output of one module to the input of the next module. In general, this requires remote transport between container nodes. The Admin server also uses the data bus to launch batch jobs by sending a message to the job’s launch channel. Since the same transport must be shared by the Admin and all Containers, the transport configuration is centrally configured in xd/config/servers.yml. The default transport is redis. Open servers.yml with a text editor and you will see the transport configuration near the top. To change the transport, you can uncomment this section and change the transport to `rabbit` or any other supported transport. Any changes to the transport configuration must be replicated to every XD node in the cluster.

Note

XD singlenode also supports a --transport command line argument, useful for testing streams under alternate transports.

```
#xd:
#  transport: redis
```

Note

If you have multiple XD instances running share 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 `spring.rabbitmq.virtual_host` property in `$XD_HOME/config/servers.yml` to point XD at the correct virtual host.

Choosing an Analytics provider

By default, the xd-container will store Analytics data in redis. At the time of writing, this is the only supported option (when running in distributed mode). Use the --analytics option to specify another backing store for Analytics data.

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

Other Options

There are additional configuration options available for these scripts:

To specify the location of the Spring XD install other than the default configured in the script

```
export XD_HOME=<Specific XD install directory>
```

To specify the http port of the XDAdmin server,

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

The XDContainer nodes by default start up with server.port 0 (which means they will scan for an available HTTP port). You can disable the HTTP endpoints for the XDContainer by setting server.port=-1. Note that in this case HTTP source support will not work in a PaaS environment because typically it would require XD to bind to a specific port. Both the XDAdmin and XDContainer processes bind to server.port $PORT (i.e. an environment variable if one is available, as is typical in a PaaS).

Using Hadoop

Spring XD supports the following Hadoop distributions:

- hadoop22 - Apache Hadoop 2.2.0 (default)
- hadoop24 - Apache Hadoop 2.4.1
- phd1 - Pivotal HD 1.1
- phd20 - Pivotal HD 2.0
- cdh5 - Cloudera CDH 5.0.0
- hdp21 - Hortonworks Data Platform 2.1

To specify the distribution libraries to use for Hadoop client connections, use the option `--hadoopDistro` for the `xd-container` and `xd-shell` commands:

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

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

XD-Shell in Distributed Mode

If you wish to use a XD-Shell that is on a different machine than where you deployed your admin server.

1) Open your shell

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

2) From the xd shell use the "admin config server" command i.e.

```
admin config server <yourhost>:9393
```

2.10 Running on YARN

Introduction

The Spring XD distributed runtime (DIRT) supports distribution of processing tasks across multiple nodes. See Running Distributed Mode for information on running Spring XD in distributed mode. One option is to run these nodes on a Hadoop YARN cluster rather than on VMs or physical servers managed by you.

What do you need?

To begin with, you need to have access to a Hadoop cluster running a version based on Apache Hadoop 2.2.0 or later. This includes Apache Hadoop 2.2.0, Pivotal HD 2.0, Hortonworks HDP 2.1 and Cloudera CDH5.

You need a supported transport, see Running Distributed Mode for installation of Redis or Rabbit MQ. Spring XD on YARN currently uses Redis as the default data transport.

You also need Zookeeper running. If your Hadoop cluster doesn’t have Zookeeper installed you need to install and run it specifically for Spring XD. See the Setting up ZooKeeper section of the "Running Distributed Mode" chapter.

Lastly, you need an RDBMs to support batch jobs and JDBC operations.

Download Spring XD on YARN binaries

In addition to the regular `spring-xd-<version>-dist.zip` files we also distribute a zip file that includes all you need to deploy on YARN. The name of this zip file is `spring-xd-<version>-yarn.zip`. You can download zip for the current release or download snapshot build. Unzip the downloaded file and you should see a `spring-xd-<version>-yarn` directory.

Configure your deployment

Configuration options are contained in a `config/servers.yml` file in the Spring XD YARN install directory. You need to configure the hadoop settings, the transport choice plus redis/rabbit settings, the zookeeper settings and the JDBC datasource properties.

Depending on the distribution used you might need to change the `siteYarnAppClasspath` and `siteMapreduceAppClasspath`. We have provided basic settings for the supported distros, you just need to uncomment the ones for the distro you use.

These are the settings used for Hadoop 2.2.0:

```
spring:
    yarn:
        siteYarnAppClasspath: "$HADOOP_CONF_DIR,$HADOOP_COMMON_HOME/share/hadoop/common/
*,$HADOOP_COMMON_HOME/share/hadoop/common/lib/*,$HADOOP_HDFS_HOME/share/hadoop/hdfs/
*,$HADOOP_HDFS_HOME/share/hadoop/hdfs/lib/*,$HADOOP_YARN_HOME/share/hadoop/yarn/*,
$HADOOP_YARN_HOME/share/hadoop/yarn/lib/*"
        siteMapreduceAppClasspath: "$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/*,
$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/lib/*"
```

XD options

For Spring XD you need to define how many admin servers and containers you need. You also need to define the HDFS location (spring.yarn.applicationDir) where the Spring XD binary and config files will be stored.

```
spring:
    xd:
        adminServers: 1
        containers: 3
    yarn:
        applicationDir: /xd/app/
```

Hadoop settings

You need to specify the host where the YARN Resource Manager is running as well as the HDFS URL.

```
# Hadoop properties
spring:
  hadoop:
    fsUri: hdfs://localhost:8020
    resourceManagerHost: localhost
```

Zookeeper settings

You should specify the Zookeeper connection settings

```
---
#Zookeeper properties
# client connect string: host1:port1,host2:port2,...,hostN:portN
zk:
  client:
     connect: localhost:2181
```

Transport options

You should choose either redis (default) or rabbit as the transport and include the host and port in the properties for the choice you made.

```
# Transport used
transport: redis
```

```
---
# Redis properties
spring:
  redis:
   port: 6379
   host: localhost
```

JDBC datasource properties

You should specify the JDBC connection properties based on the RDBMs that you use for the batch jobs and JDBC sink

```
---
#Config for use with MySQL - uncomment and edit with relevant values for your environment
spring:
  datasource:
    url: jdbc:mysql://yourDBhost:3306/yourDB
    username: yourUsername
    password: yourPassword
    driverClassName: com.mysql.jdbc.Driver
```

Customizing module configurations

The configurations for all modules can be customized by modifying the file `modules.yml` in the `config` directory and then adding it to the `modules-config.zip` archive in the same directory.

You can run the following command from the `config` directory to achieve this:

```
jar -uf modules-config.zip modules.yml
```

Adding custom modules

There is an empty archive file named `custom-modules.zip`. You can replace this with your own ZIP archive with the same name and it will be uploaded as part of the deployment to YARN. Place custom module definitions in a `modules` directory inside this new ZIP archive. Module definitions must follow the Spring XD module semantics.

Push and start the jobs

Change current directory to be the directory that was unzipped (spring-xd-<version>-yarn).

Push the Spring XD application binaries and config to HDFS

Run the command

```
./bin/xd-yarn push
```

Submit the Spring XD admin server

Run the command

```
./bin/xd-yarn start admin
```

Submit the Spring XD container

Run the command

```
./bin/xd-yarn start container
```

Check the status of the app

You can use the regular `yarn` command to check the status. Simply run:

```
yarn application -list
```

You should see two applications running named xd-admin and xd-container.

3. Application Configuration

3.1 Introduction

There are two main parts of Spring XD that can be configured, servers and modules.

The servers (`xd-singlenode`, `xd-admin`, `xd-container`) are Spring Boot applications and are configured as described in the Spring Boot Reference documentation. In the most simple case this means editing values in the YAML based configuration file `servers.yml`. The values in this configuration file will overwrite the values in the default application.yml file that is embedded in the XD jar.

Note

The use of YAML is an alternative to using property files. YAML is a superset of JSON, and as such is a very convenient format for specifying hierarchical configuration data.

For modules, each module has its own configuration file located in its own directory, for example `source/http/http.properties`. Shared configuration values for modules can be placed in a common `modules.yml` file.

For both server and module configuration, you can have environment specific settings through the use of application profiles and the ability to override values in files by setting OS environment variables.

In this section we will walk though how to configure servers and modules.

3.2 Server Configuration

The startup scripts for `xd-singlenode`, `xd-admin`, and `xd-container` will by default look for the file `$XD_HOME\config\servers.yml` as a source of externalized configuration information.

The location and name of this resourse can be changed by using the environment variables `XD_CONFIG_LOCATION` and `XD_CONFIG_NAME`. The start up script takes the value of these environment variables to set the Spring Boot properties `spring.config.location` and `spring.config.name`. Note, that for `XD_CONFIG_LOCATION` you can reference any Spring Resource implementation, most commonly denoted using the prefixes `classpath:`, `file:` and `http:`.

It is common to keep your server configuration separate form the installation directory of XD itself. To do this, here is an example environment variable setting

```
export XD_CONFIG_LOCATION=file:/xd/config/
export XD_CONFIG_NAME=region1-servers
```

Note: the file path separator ("/") at the end of XD_CONFIG_LOCATION is **necessary**.

Profile support

Profiles provide a way to segregate parts of your application configuration and change their availability and/or values based on the environment. This lets you have different configuration settings for qa and `prod` environments and to easily switch between them.

To activate a profile, set the OS environment variable `SPRING_PROFILES_ACTIVE` to a comma delimited list of profile names. The server looks to load profile specific variants of the `servers.yml` file based on the naming convention `servers-{profile}.yml`. For example, if `SPRING_PROFILES_ACTIVE=prod` the following files would be searched for in the following order.

1. `XD_CONFIG_LOCATION/servers-prod.yml`

2. `XD_CONFIG_LOCATION/servers.yml`

You may also put multiple profile specific configuration in a single `servers.yml` file by using the key `spring.profiles` in different sections of the configuration file. See Multi-profile YAML documents for more information.

Database Configuration

Spring XD saves the state of the batch job workflows in a relational database. When running `xd-singlenode` a embedded HSQLDB database is run. When running in distributed mode a standalone HSQLDB instance can be used, the startup script `hsqldb-server` is in is provided the installation directory under the folder *hsqldb/bin*. It is recommended to use HSQLDB only for development and learning.

When deploying in a production environment, you will need to select another database. Spring XD is actively tested on MySql (Version: 5.1.23) and Postgres (Version 9.2-1002). All batch workflow tables are automatically created, if they do not exist, for HSQLDB, MySQL and Postgres. The JDBC driver jars for the HSQLDB, MySql, and Postgres are already on the XD classpath.

The provided configuration file `servers.yml` located in `$XD_HOME\config` has commented out configuration for some commonly used databases. You can use these as a basis to support your database environment. XD also utilizes the Tomcat jdbc connection pool and these settings can be configured in the `servers.yml`.

Note

Until full schema support is added for Oracle, Sybase and other database, you will need to put a .jar file in the `xd/lib` directory that contains the equivalent functionality as these DDL scripts.

Note

There was a schema change in version 1.0 RC1. Use or adapt the the sample migration class to update your schema.

HSQLDB

When in distributed mode and you want to use HSQLDB, you need to change the value of `spring.datasource` properties. As an example,

```
hsql:
 server:
   host: localhost
   port: 9102
   dbname: xdjob
spring:
   datasource:
   url: jdbc:hsqldb:hsql://${hsql.server.host:localhost}:${hsql.server.port:9101}/
${hsql.server.dbname:xdjob}
   username: sa
   password:
   driverClassName: org.hsqldb.jdbc.JDBCDriver
```

The properties under hsql.server are substituted in the spring.datasource.url property value. This lets you create short variants of existing Spring Boot properties. Using this style, you can override the value of these configuration variables by setting an OS environment variable, such as xd_server_host. Alternatively, you can not use any placeholders and set spring.datasource.url directly to known values.

MySQL

When in distributed mode and you want to use MySQL, you need to change the value of spring.datasource.* properties. As an example,

```
spring:
 datasource:
 url: jdbc:mysql://yourDBhost:3306/yourDB
 username: yourUsername
 password: yourPassword
 driverClassName: com.mysql.jdbc.Driver
```

To override these settings set an OS environment variable such as spring_datasource_url to the value you require.

PostgreSQL

When in distributed mode and you want to use PostgreSQL, you need to change the value of spring.datasource.* properties. As an example,

```
spring:
  datasource:
    url: jdbc:postgresql://yourDBhost:5432/yourDB
    username: yourUsername
    password: yourPassword
    driverClassName: org.postgresql.Driver
```

To override these settings set an OS environment variable such as spring_datasource_url to the value you require.

Redis

If you want to use Redis for analytics or data transport you should set the host and port of the Redis server.

```
spring:
  redis:
   port: 6379
   host: localhost
```

To override these settings set an OS environment variable such as spring_redis_port to the value you require.

RabbitMQ

If you want to use RabbitMQ as a data transport use the following configuration setting

```
spring:
  rabbitmq:
   addresses: localhost:5672
   username: guest
   password: guest
   virtual_host: /
   useSSL: false
   sslProperties:
```

To override these settings set an OS environment variable such as `spring_rabbitmq_host` to the value you require.

See Message Bus regarding SSL configuration.

In addition, the following default settings for the rabbit message bus can be modified in `servers.yml`…

```
  messagebus:
    rabbit:
      default:
        ackMode:                   AUTO   ❶
        backOffInitialInterval:    1000   ❷
        backOffMaxInterval:        10000  ❸
        backOffMultiplier:         2.0    ❹
        concurrency:               1      ❺
        maxAttempts:               3      ❻
        maxConcurrency:            1      ❼
        prefix:                    xdbus.  ❽
        prefetch:                  1      ❾
        replyHeaderPatterns:       STANDARD_REPLY_HEADERS,*    ❿
        requestHeaderPatterns:     STANDARD_REQUEST_HEADERS,*  ⓫
        requeue:                   true   ⓬
        transacted:                false  ⓭
        txSize:                    1      ⓮
```

❶ AUTO (container acks), NONE (broker acks), MANUAL (consumer acks). Upper case only. Note: MANUAL requires specialized code in the consuming module and is unlikely to be used in an XD application. For more information, see http://docs.spring.io/spring-integration/reference/html/amqp.html#amqp-inbound-ack

❷ The time in milliseconds before retrying a failed message delivery

❸ The maximum time (ms) to wait between retries

❹ The back off multiplier (previous interval x multiplier = next interval)

❺ The minimum number of consumer threads receiving messages for a module

❻ The maximum number of delivery attempts

❼ The maximum number of consumer threads receiving messages for a module

❽ A prefix applied to all queues, exchanges so that policies (HA etc) can be applied

❾ The number of messages to prefetch for each consumer

❿ Determines which request headers will be transported

⓫ Determines which reply headers will be transported

⓬ Whether rejected messages will be requeued by default

⓭ Whether the channel is to be transacted

⓮ The number of messages to process between acks (when ack mode is AUTO).

Admin Server HTTP Port

The default HTTP port of the `xd-admin` server is 9393. To change the value use the following configuration setting

```
server:
  port: 9876
```

Management Port

The XD servers provide general health and JMX exported management endpoints via Jolokia.

By default the management and health endpoints are available on port 9393. To change the value of the port use the following configuration setting.

```
management:
  port: 9876
```

You can also disable http management endpoints by setting the port value to -1.

By default JMX MBeans are exported. You can disable JMX by setting `spring.jmx.enabled=false`.

The section on Monitoring and management over HTTP provides details on how to configure these endpoint.

Local transport

Local transport uses a QueueChannel to pass data between modules. There are a few properties you can configure on the QueueChannel

- `xd.local.transport.named.queueSize` - The capacity of the queue, the default value is `Integer.MAX_VALUE`
- `xd.local.transport.named.polling` - Messages that are buffered in a QueueChannel need to be polled to be consumed. This property controls the fixed rate at which polling occurs. The default value is 1000 ms.

3.3 Module Configuration

Modules are configured by placing property files in a nested directory structure based on their type and name. The root of the nested directory structure is by default `XD_HOME/config/modules`. This location can be customized by setting the OS environment variable `XD_MODULE_CONFIG_LOCATION`, similar to how the environment variable `XD_CONFIG_LOCATION` is used for configuring the server. If `XD_MODULE_CONFIG_LOCATION` is set explicitly, then it is **necessary** to add the file path separator ("/") at the end of the path.

Note

If `XD_MODULE_CONFIG_LOCATION` is set to use explicit location, make sure to copy entire directory structure from the default module config location `xd/config/modules` into the new module config location. The `XD_MODULE_CONFIG_LOCATION` can reference any Spring Resource implementation, most commonly denoted using the prefixes `classpath:`, `file:` and `http:`.

As an example, if you wanted to configure the twittersearch module, you would create a file

```
XD_MODULE_CONFIG_LOCATION\source\twittersearch\twittersearch.properties
```

and the contents of that file would be property names such as `consumerKey` and `consumerSecret`.

Note

You **do not** need to prefix these property names with a `source.twittersearch` prefix.

You can override the values in the module property file in various ways. The following sources of properties are considered in the following order.

1. Properties specified in the stream or job `DSL` definition
2. Java System Properties (e.g. source.http.port=9454)
3. OS environment variables. (e.g. source_http_port=9454)
4. `XD_MODULE_CONFIG_LOCATION\<type>\<name>\<name>.properties` (including profile variants)
5. Default values specified in module metadata (if available).

Values in `XD_MODULE_CONFIG_LOCATION\<type>\<name>\<name>.properties` can be property placeholder references to keys defined in another resource location. By default the resource is the file `XD_MODULE_CONFIG_LOCATION\modules.yml`. You can customize the name of the resource by using setting the OS environment variable `XD_MODULE_CONFIG_NAME` before running a server startup script.

The `modules.yml` file can be used to specify the values of keys that should be shared across different modules. For example, it is common to use the same twitter developer credentials in both the twittersearch and twitterstream modules. To avoid repeating the same credentials in two property files, you can use the following setup.

`modules.yml` contains

```
sharedConsumerKey: alsdjfqwopieur
sharedConsumerSecret: pqwieouralsdjkqwpo
sharedAccessToken: llixzchvpiawued
sharedAccessTokenSecret: ewoqirudhdsldke
```

and `XD_MODULE_CONFIG_LOCATION\source\twitterstream\twitterstream.properties` contains

```
consumerKey=${sharedConsumerKey}
consumerSecret=${sharedConsumerSecret}
accessToken=${sharedAccessToken}
accessTokenSecret=${sharedAccessTokenSecret}
```

and `XD_MODULE_CONFIG_LOCATION\source\twittersearch\twittersearch.properties` contains

```
consumerKey=${sharedConsumerKey}
consumerSecret=${sharedConsumerSecret}
```

Profiles

When resolving property file names, the server will look to load profile specific variants based on the naming convention `<name>-{profile}.properties`. For example, if given the OS environment variable `spring_profiles_active=default,qa` the following configuration file names for the twittersearch module would be searched in this order

1. `XD_MODULE_CONFIG_LOCATION\source\twittersearch\twittersearch.properties`
2. `XD_MODULE_CONFIG_LOCATION\source\twittersearch\twittersearch-default.properties`
3. `XD_MODULE_CONFIG_LOCATION\source\twittersearch\twittersearch-qa.properties`

Also, the shared module configuration file is refernced using profile variants, so given the OS environment variable `spring_profiles_active=default,qa` the following shared module configuration files would be searched for in this order

1. `XD_MODULE_CONFIG_LOCATION\modules.yml`
2. `XD_MODULE_CONFIG_LOCATION\modules-default.yml`
3. `XD_MODULE_CONFIG_LOCATION\modules-qa.yml`

Batch Jobs or modules accessing JDBC

Another common case is access to a relational database from a job or the JDBC Sink module.

As an example, to provide the properties for the batch job `jdbchdfs` the file `XD_MODULE_CONFIG_LOCATION\job\jdbchdfs\jdbchdfs.properites` should contain

```
driverClass=org.hsqldb.jdbc.JDBCDriver
url=jdbc:hsqldb:mem:xd
username=sa
password=
```

A property file with the same keys, but likely different values would be located in `XD_MODULE_CONFIG_LOCATION\sink\jdbc\jdbc.properites`.

4. DSL Guide

4.1 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.

4.2 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 `|`.

4.3 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'''
```

4.4 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 named channel is specified by using a channel type, followed by a `:` followed by a name. The channel types available are:

```
queue - this type of channel has point-to-point (p2p) semantics
```

```
topic - this type of channel has pub/sub semantics
```

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

```
queue:foo > file
```

```
http > queue:foo
```

```
time > queue: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:stream:mystream > file
```

```
tap:stream:mystream > log
```

Once data is received on `mystream`, it will be written to both file and log.

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

4.5 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
```

Labels are especially useful for disambiguating when multiple modules of the same name are used:

```
mystream = http | uppercaser: transform --expression=payload.toUpperCase() | exclaimer:
 transform --expression=payload+'!' | file
```

Refer to this section of the Taps chapter to see how labels facilitate the creation of taps in these cases where a stream contains ambiguous modules.

4.6 Single quotes, Double quotes, Escaping

Spring XD is a complex runtime that involves a lot of systems when you look at the complete picture. There is a **Spring Shell based client** that talks to the admin that is responsible for **parsing**. In turn, modules may themselves rely on embedded languages (like the **Spring Expression Language**) to accomplish their behavior.

Those three components (shell, XD parser and SpEL) have rules about how they handle quotes and how syntax escaping works, and when stacked with each other, confusion may arise. This section explains the rules that apply and provides examples to the most common situations.

It’s not always that complicated

This section focuses on the most complicated cases, when all 3 layers are involved. Of course, if you don’t use the XD shell (for example if you’re using the REST API directly) or if module option values are not SpEL expressions, then escaping rules can be much simpler

Spring Shell

Arguably, the most complex component when it comes to quotes is the shell. The rules can be laid out quite simply, though:

- a shell command is made of keys (`--foo`) and corresponding values. There is a special, key-less mapping though, see below
- a value can not normally contain spaces, as space is the default delimiter for commands
- spaces can be added though, by surrounding the value with quotes (either single [`'`] or double [`"`] quotes)
- if surrounded with quotes, a value can embed a literal quote of the same kind by prefixing it with a backslash (`\`)
- Other escapes are available, such as `\t`, `\n`, `\r`, `\f` and unicode escapes of the form `\uxxxx`
- Lastly, the key-less mapping is handled in a special way in the sense that if does not need quoting to contain spaces

For example, the XD shell supports the `!` command to execute native shell commands. The `!` accepts a single, key-less argument. This is why the following works:

```
xd:>! rm foo
```

The argument here is the whole `rm foo` string, which is passed as is to the underlying shell.

As another example, the following commands are strictly equivalent, and the argument value is `foo` (without the quotes):

```
xd:>stream destroy foo
xd:>stream destroy --name foo
xd:>stream destroy "foo"
xd:>stream destroy --name "foo"
```

XD Syntax

At the XD parser level (that is, inside the body of a stream or job definition) the rules are the following:

- option values are normally parsed until the first space character
- they can be made of literal strings though, surrounded by single or double quotes
- To embed such a quote, use two consecutive quotes of the desired kind

As such, the values of the `--expression` option to the filter module are semantically equivalent in the following examples:

```
filter --expression=payload>5
filter --expression="payload>5"
filter --expression='payload>5'
filter --expression='payload > 5'
```

Arguably, the last one is more readable. It is made possible thanks to the surrounding quotes. The actual expression is `payload > 5` (without quotes).

Now, let's imagine we want to test against string messages. If we'd like to compare the payload to the SpEL literal string, `"foo"`, this is how we could do:

```
filter --expression=payload=='foo'           ❶
filter --expression='payload == ''foo'''     ❷
filter --expression='payload == "foo"'       ❸
```

❶ This works because there are no spaces. Not very legible though

❷ This uses single quotes to protect the whole argument, hence actual single quotes need to be doubled

❸ But SpEL recognizes String literals with either single or double quotes, so this last method is arguably the best

Please note that the examples above are to be considered outside of the Spring XD shell. When entered inside the shell, chances are that the whole stream definition will itself be inside double quotes, which would need escaping. The whole example then becomes:

```
xd:>stream create foo --definition "http | filter --expression=payload='foo' | log"
xd:>stream create foo --definition "htpp | filter --expression='payload == ''foo''' | log"
xd:>stream create foo --definition "http | filter --expression='payload == \"foo\"' | log"
```

SpEL syntax and SpEL literals

The last piece of the puzzle is about SpEL expressions. Many modules accept options that are to be interpreted as SpEL expressions, and as seen above, String literals are handled in a special way there too. Basically,

- literals can be enclosed in either single or double quotes
- quotes need to be doubled to embed a literal quote. Single quotes inside double quotes need no special treatment, and *vice versa*

As a last example, assume you want to use the transform module. That module accepts an `expression` option which is a SpEL expression. It is to be evaluated against the incoming message, with a default of `payload` (which forwards the message payload untouched).

It is important to understand that the following are equivalent:

```
transform --expression=payload
transform --expression='payload'
```

but very different from the following:

```
transform --expression="'payload'"
transform --expression='''payload'''
```

and other variations.

The first series will simply evaluate to the message payload, while the latter examples will evaluate to the actual literal string `payload` (again, without quotes).

Putting it all together

As a last, complete example, let’s review how one could force the transformation of all messages to the string literal `hello world`, by creating a stream in the context of the XD shell:

```
stream create foo --definition "http | transform --expression='''hello world''' | log" ❶
stream create foo --definition "http | transform --expression='\"hello world\"' | log" ❷
stream create foo --definition "http | transform --expression=\"'hello world'\" | log" ❸
```

❶ This uses single quotes around the string (at the XD parser level), but they need to be doubled because we’re inside a string literal (very first single quote after the equals sign)

❷❸ use single and double quotes respectively to encompass the whole string at the XD parser level. Hence, the other kind of quote can be used inside the string. The whole thing is inside the `--definition` argument to the shell though, which uses double quotes. So double quotes are escaped (at the shell level)

5. Interactive Shell

5.1 Introduction

Spring XD includes an interactive shell that you can use create, deploy, destroy and query streams and jobs. There are also commands to help with common tasks such as interacting with HDFS, the UNIX shell, and sending HTTP requests. In this section we will introduce the main commands and features of the shell.

Using the Shell

When you start the shell you can type *help* to show all the commands that are available. Note, that since the XD shell is based on Spring Shell you can contribute you own commands into the shell. The general groups of commands are related to the management of

- Modules
- Streams
- Jobs
- Analytics (Counters, Aggregate Counters, Gauges, etc.)
- HDFS

For example to see what modules are available issue the command

```
xd:>module list
```

Tip

The list of all Spring XD specific commands can be found in the Shell Reference

The shell also provides extensive command completion capabilities. For example, if you type *mod* and hit TAB, you will be presented with all the matching commands.

```
xd:>module
module compose    module delete     module display    module info
module list
```

Note

Tab completion works for module options as well as for the DSL used within the `--definition` option for stream and module commands.

The command `module list` shows all the modules available

```
xd:>module list
      Source              Processor           Sink                     Job
  ------------------  ------------------  -----------------------  ----------------
      gemfire-cq          aggregator          counter                  hdfsjdbc
      post                http-client         log                      jdbchdfs
      twitterstream       splitter            field-value-counter      hdfsmongodb
      http                filter              rich-gauge               filejdbc
      reactor-syslog      json-to-tuple       mqtt                     ftphdfs
      reactor-ip          transform           file                     filepollhdfs
      jms                 bridge              splunk
      tcp-client          object-to-json      mail
      mqtt                script              tcp
      file                                    hdfs
      twittersearch                           gauge
      gemfire                                 jdbc
      mail                                    gemfire-server
      trigger                                 throughput-sampler
      tcp                                     gemfire-json-server
      tail                                    router
      syslog-tcp                              aggregate-counter
      syslog-udp                              rabbit
      rabbit                                  hdfs-dataset
      time
```

Suppose we want to create a stream that uses the `http` source and `file` sink. How do we know what options are available to use? There are two ways to find out. The first is to use the command `module info`. Pressing TAB after typiing `moudle info` will complete the command with the `--name` option and then present all the modules prefixed by their type.

```
xd:>module info --name
job:filejdbc                job:filepollhdfs            job:ftphdfs
job:hdfsjdbc                job:hdfsmongodb             job:jdbchdfs
processor:aggregator        processor:bridge            processor:filter
processor:http-client       processor:json-to-tuple     processor:object-to-json
processor:script            processor:splitter          processor:transform
sink:aggregate-counter      sink:counter                sink:field-value-counter
sink:file                   sink:gauge                  sink:gemfire-json-server
sink:gemfire-server         sink:hdfs                   sink:hdfs-dataset
sink:jdbc                   sink:log                    sink:mail
sink:mqtt                   sink:rabbit                 sink:rich-gauge
sink:router                 sink:splunk                 sink:tcp
sink:throughput-sampler     source:file                 source:gemfire
source:gemfire-cq           source:http                 source:jms
source:mail                 source:mqtt                 source:post
source:rabbit               source:reactor-ip           source:reactor-syslog
source:syslog-tcp           source:syslog-udp           source:tail
source:tcp                  source:tcp-client           source:time
source:trigger              source:twittersearch        source:twitterstream
xd:>module info --name
```

The `module info` command for the http source shows the option names, a brief description, and default values.

```
xd:>module info --name source:http
Information about source module 'http':

  Option Name  Description                                       Default  Type
  -----------  ------------------------------------------------  -------  --------
  port         the port to listen to                             9000     int
  outputType   how this module should emit messages it produces  <none>   MimeType
```

For the `file` sink the options are

```
xd:>module info --name sink:file
Information about sink module 'file':

  Option Name  Description                                                   Default
           Type
  -----------  ----------------------------------------------------------------
 -----------------  --------
  binary       if false, will append a newline character at the end of each line  false
           boolean
  charset      the charset to use when writing a String payload                  UTF-8
           String
  dir          the directory in which files will be created                      /tmp/xd/
output/    String
  mode         what to do if the file already exists                             APPEND
           Mode
  name         filename pattern to use
 ${xd.stream.name}  String
  suffix       filename extension to use                                         <none>
           String
  inputType    how this module should interpret messages it consumes             <none>
           MimeType
```

Note that the default value `${xd.stream.name}` will be resolved to the name of the stream that contains the module.

Tab completion for Job and Stream DSL definitions

When creating a stream defintion tab completion after `--definition` will enable you to see all the options that are available for a given module as well as a list of candidate modules for the subsequent module in the stream. For example, hitting TAB after `http` as shown below

```
xd:>stream create --name test --definition "http
http --outputType=          http --port=               http | aggregate-counter
 http | aggregator
http | bridge               http | counter             http | field-value-counter
 http | file
http | filter               http | gauge               http | gemfire-json-server
 http | gemfire-server
http | hdfs                 http | hdfs-dataset        http | http-client
 http | jdbc
http | json-to-tuple        http | log                 http | mail
 http | mqtt
http | object-to-json       http | rabbit              http | rich-gauge
 http | router
http | script               http | splitter            http | splunk
 http | tcp
http | throughput-sampler   http | transform
```

shows the options `outputType` and `port` in addition to any processors and sinks. Hitting `TAB` after entering `--` after the http module will provide a list of only the http options

```
xd:>stream create --name test --definition "http --
http --outputType=   http --port=
```

Entering the port number and also the pipel `|` symbol and hitting tab will show completions for candidate processor and sink modules. The same process of tab completion for module options applies to each module in the chain.

Executing a script

You can execute a script by either passing in the `--cmdfile` argument when starting the shell or by executing the `script` command inside the shell. When using scripts it is common to add comments using either `//` or `;` characters at the start of the line for one line comments or use `/*` and `*/` for multiline comments

Single quotes, Double quotes, Escaping

There are often three layers of parsing when passing entering commands to the shell. The shell parses the command to recognize `--` options, inside the body of a stream/job definition the values are parsed until the first space character, and inside some command options SpEL is used (e.g. router). Understanding the interaction between these layers can cause some confusion. The DSL Guide section on quotes and escaping will help you if you run into any issues.

6. Admin UI

6.1 Introduction

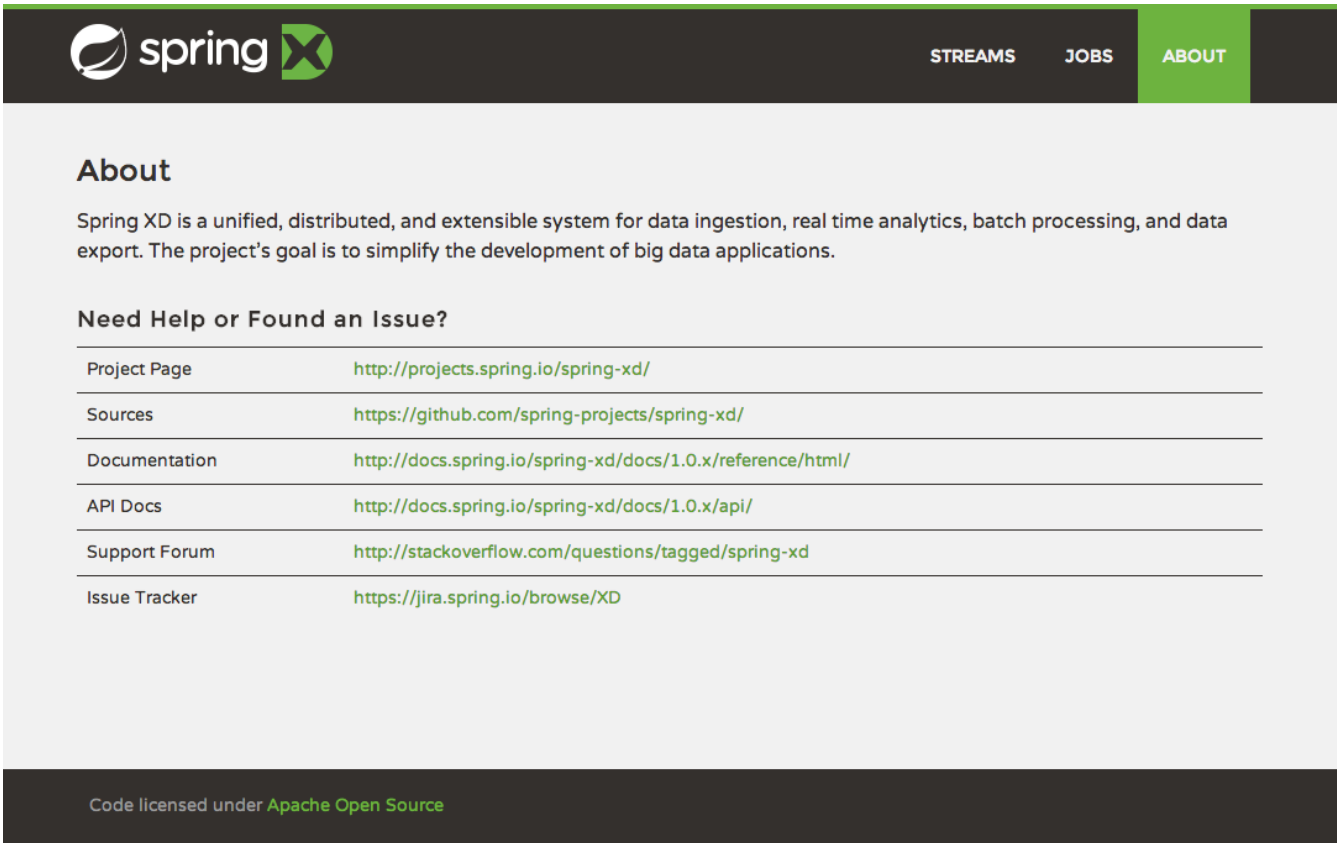
Spring XD provides a browser-based GUI which currently has 2 sections allowing you to

- perform Batch Job related tasks
- deploy/undeploy Stream Definitions

Upon starting Spring XD, the Admin UI is available at:

"http://<adminHost>:<adminPort>/admin-ui" (for example: http://localhost:9393/admin-ui)

Note: Default admin server port is 9393



About

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

Need Help or Found an Issue?

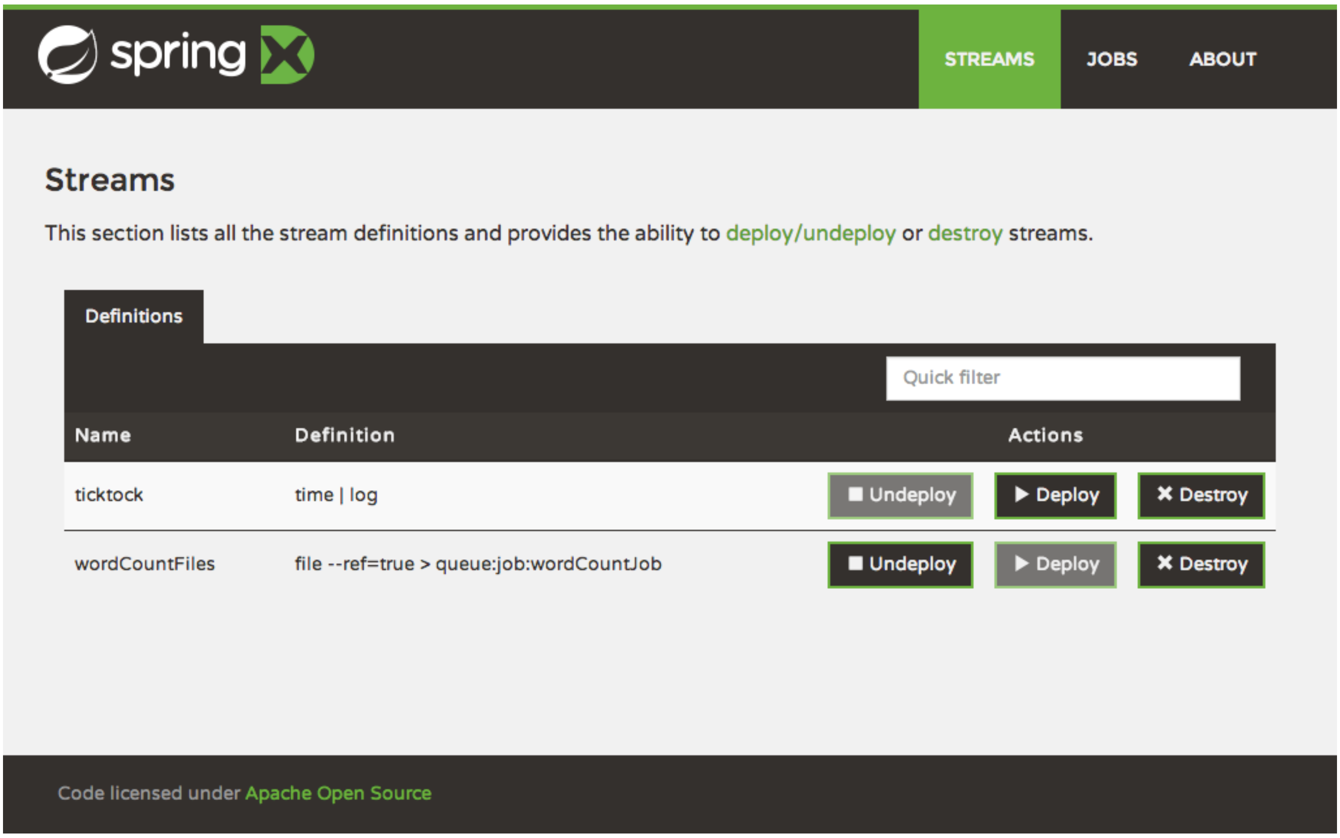
Project Page	http://projects.spring.io/spring-xd/
Sources	https://github.com/spring-projects/spring-xd/
Documentation	http://docs.spring.io/spring-xd/docs/1.0.x/reference/html/
API Docs	http://docs.spring.io/spring-xd/docs/1.0.x/api/
Support Forum	http://stackoverflow.com/questions/tagged/spring-xd
Issue Tracker	https://jira.spring.io/browse/XD

Code licensed under Apache Open Source

Figure 6.1. The Spring XD Admin UI

6.2 Streams

The *Streams* section of the admin UI provides the *Definitions* tab that provides a listing of Stream definitions. There you have the option to **deploy** or **undeploy** those streams. Additionally you can remove the definition by clicking on **destroy**.



Streams

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

Name	Definition	Actions		
ticktock	time | log	■ Undeploy	▶ Deploy	✖ Destroy
wordCountFiles	file --ref=true > queue:job:wordCountJob	■ Undeploy	▶ Deploy	✖ Destroy

Code licensed under Apache Open Source

Figure 6.2. List of Stream Definitions

6.3 Jobs

The *Jobs* section of the admin UI currently has four tabs specific for **Batch Jobs**

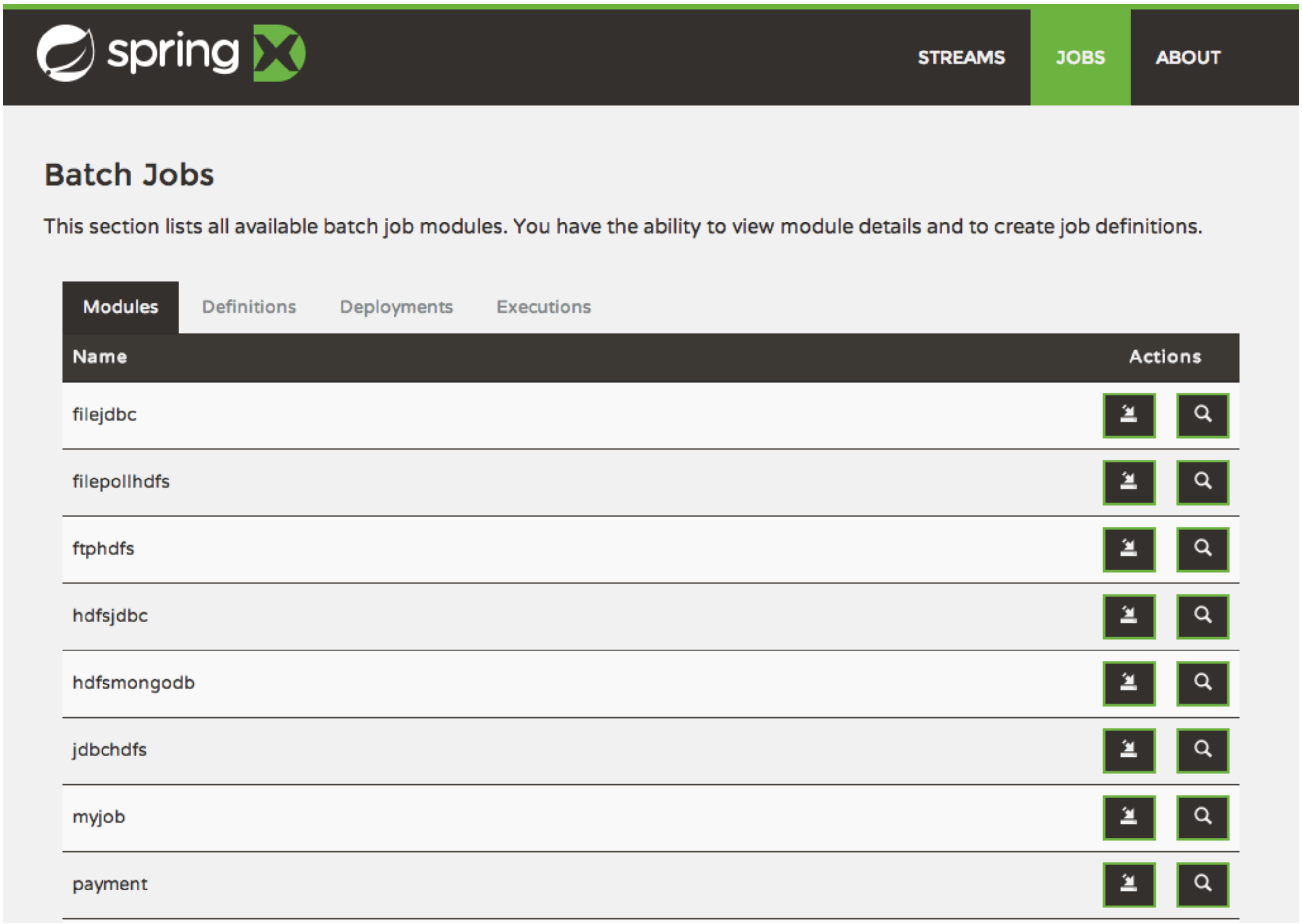
- Modules
- Definitions
- Deployments
- Executions

Modules

Modules encapsulate a unit of work into a reusable component. Within the XD runtime environment Modules allow users to create definitions for *Streams* as well as *Batch Jobs*. Consequently, the *Modules* tab within the *Jobs* section allows users to create *Batch Job* definitions. In order to learn more about *Modules*, please see the chapter on Modules.

List available batch job modules

This page lists the available batch job modules.



Batch Jobs

This section lists all available batch job modules. You have the ability to view module details and to create job definitions.

Name	Actions
filejdbc	⤓ 🔍
filepollhdfs	⤓ 🔍
ftphdfs	⤓ 🔍
hdfsjdbc	⤓ 🔍
hdfsmongodb	⤓ 🔍
jdbchdfs	⤓ 🔍
myjob	⤓ 🔍
payment	⤓ 🔍

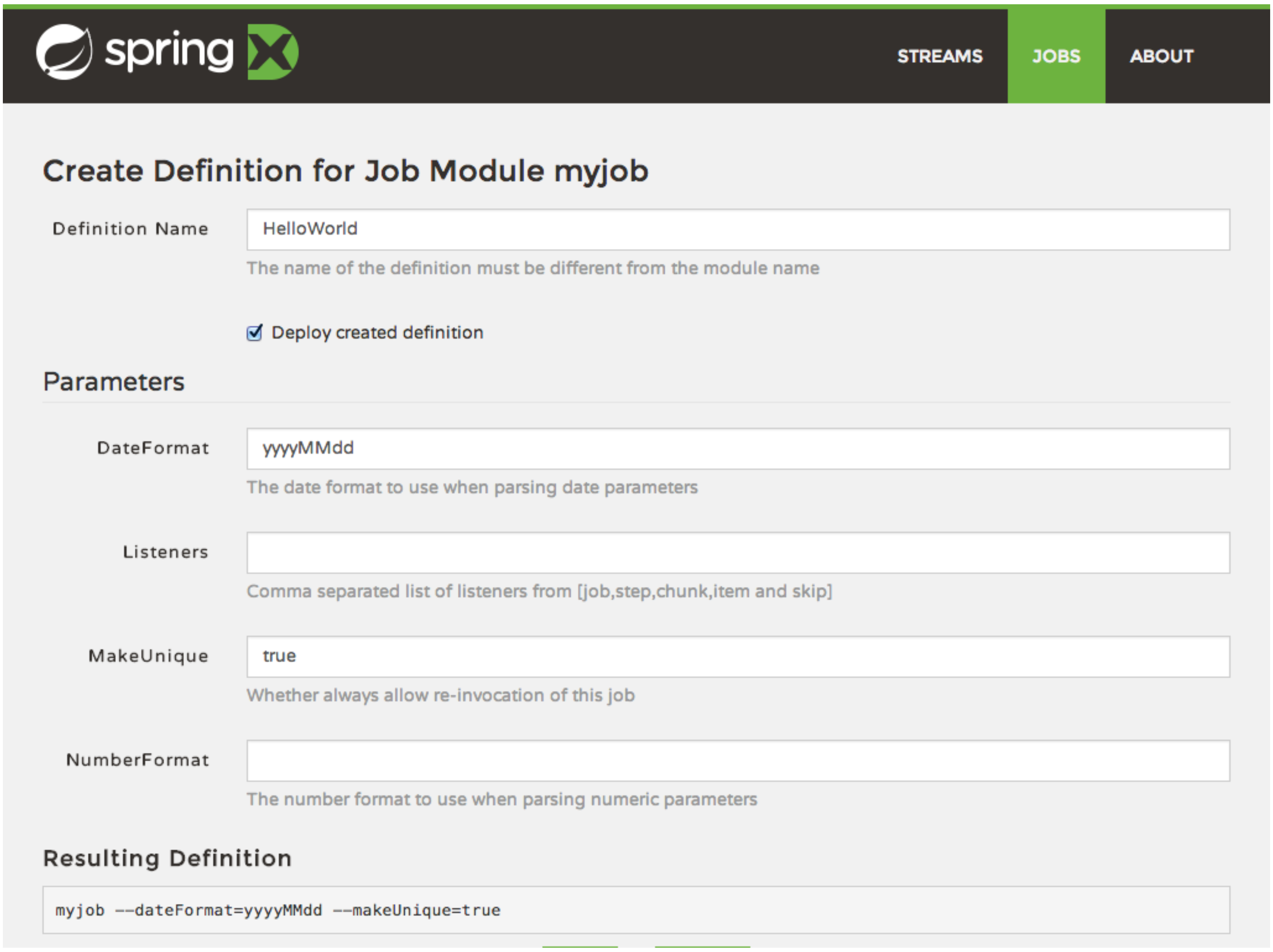
Figure 6.3. List Job Modules

On this screen you can perform the following actions:

🔍	View details such as the job module options and the module XML configuration file.
⤓	Create a Job Definition from the respective Module.

Create a Job Definition from a selected Job Module

On this screen you can create a new Job Definition. As a minimum you must provide a name for the new definition. Optionally you can select wether the new definition shall be automatically deployed. Depending on the selected module, you will also have the option to specify various parameters that are used during the deployment of the definition.



Create Definition for Job Module myjob

Definition Name: HelloWorld

The name of the definition must be different from the module name

☑ Deploy created definition

Parameters

DateFormat: yyyyMMdd

The date format to use when parsing date parameters

Listeners:

Comma separated list of listeners from [job,step,chunk,item and skip]

MakeUnique: true

Whether always allow re-invocation of this job

NumberFormat:

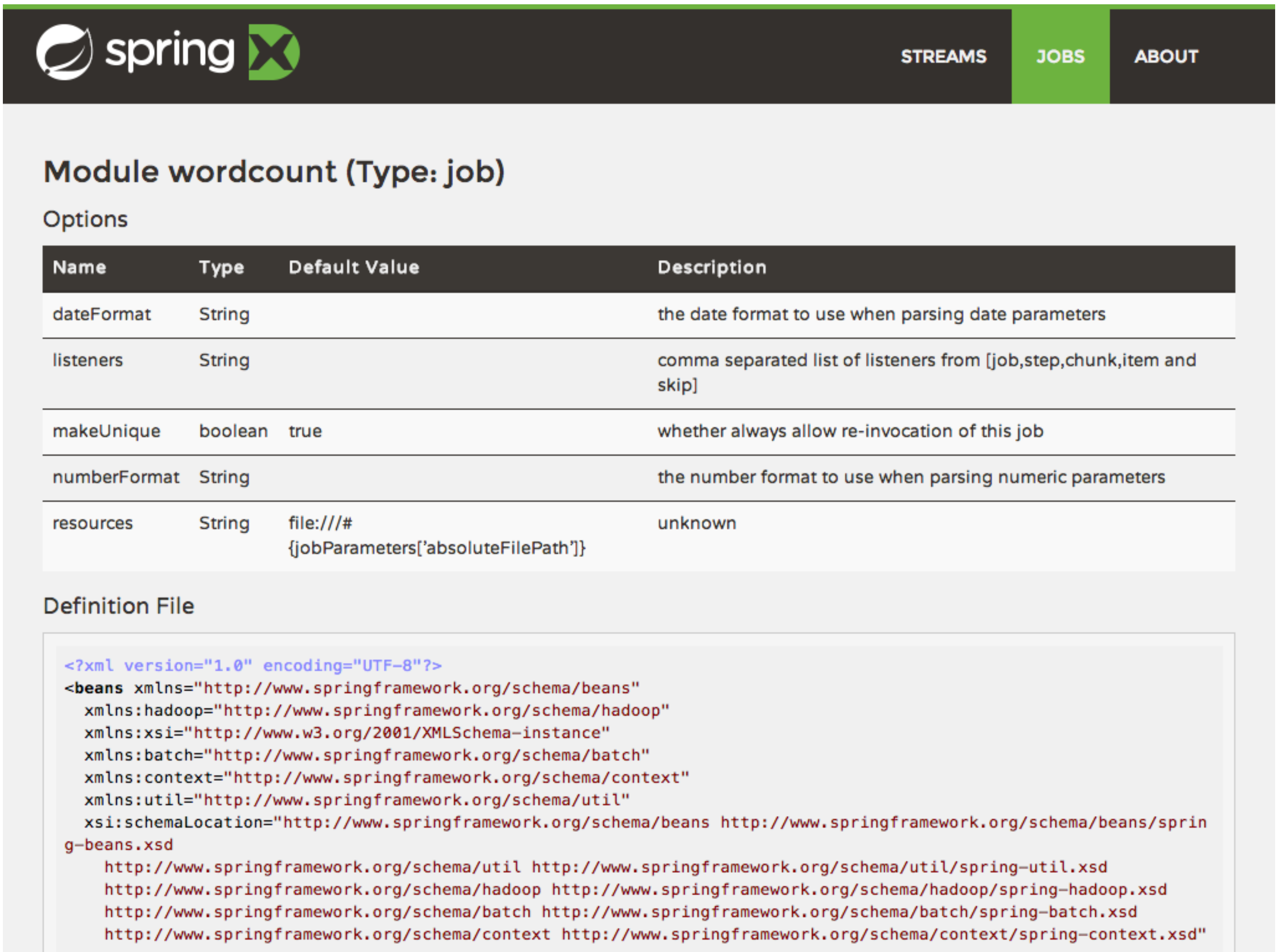
The number format to use when parsing numeric parameters

Resulting Definition

```
myjob --dateFormat=yyyyMMdd --makeUnique=true
```

Figure 6.4. Create a Job Definition

View Job Module Details



Module wordcount (Type: job)

Options

Name	Type	Default Value	Description
dateFormat	String		the date format to use when parsing date parameters
listeners	String		comma separated list of listeners from [job,step,chunk,item and skip]
makeUnique	boolean	true	whether always allow re-invocation of this job
numberFormat	String		the number format to use when parsing numeric parameters
resources	String	file:///#{jobParameters['absoluteFilePath']}	unknown

Definition File

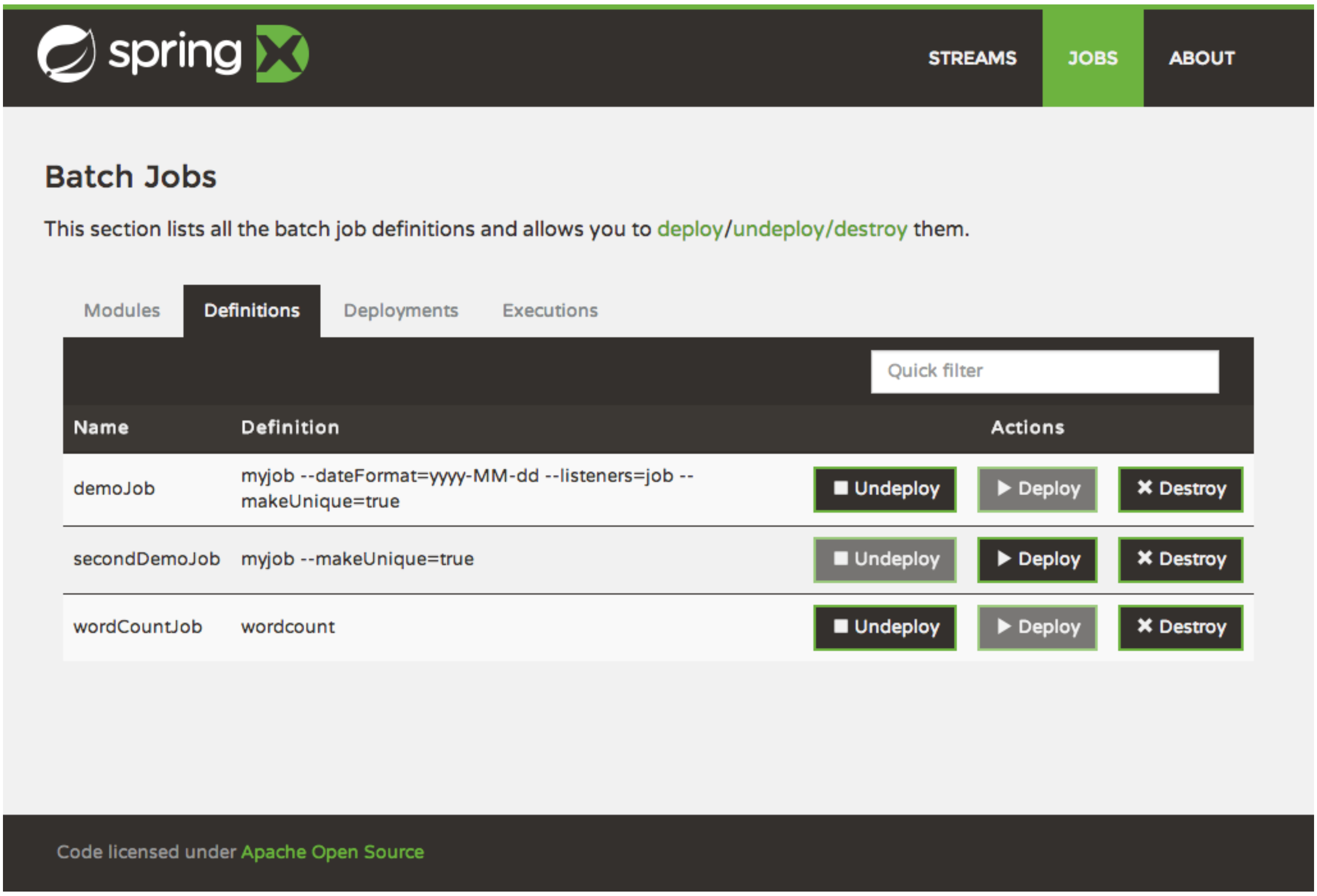
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:hadoop="http://www.springframework.org/schema/hadoop"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:batch="http://www.springframework.org/schema/batch"
  xmlns:context="http://www.springframework.org/schema/context"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

Figure 6.5. View Job Module Details

On this page you can view the details of a selected job module. The pages lists the available options (properties) of the modules as well as the actual XML application context file associated with the module.

List job definitions

This page lists the XD batch job definitions and provides actions to **deploy**, **un-deploy** or **destroy** those jobs.



Batch Jobs

This section lists all the batch job definitions and allows you to deploy/undeploy/destroy them.

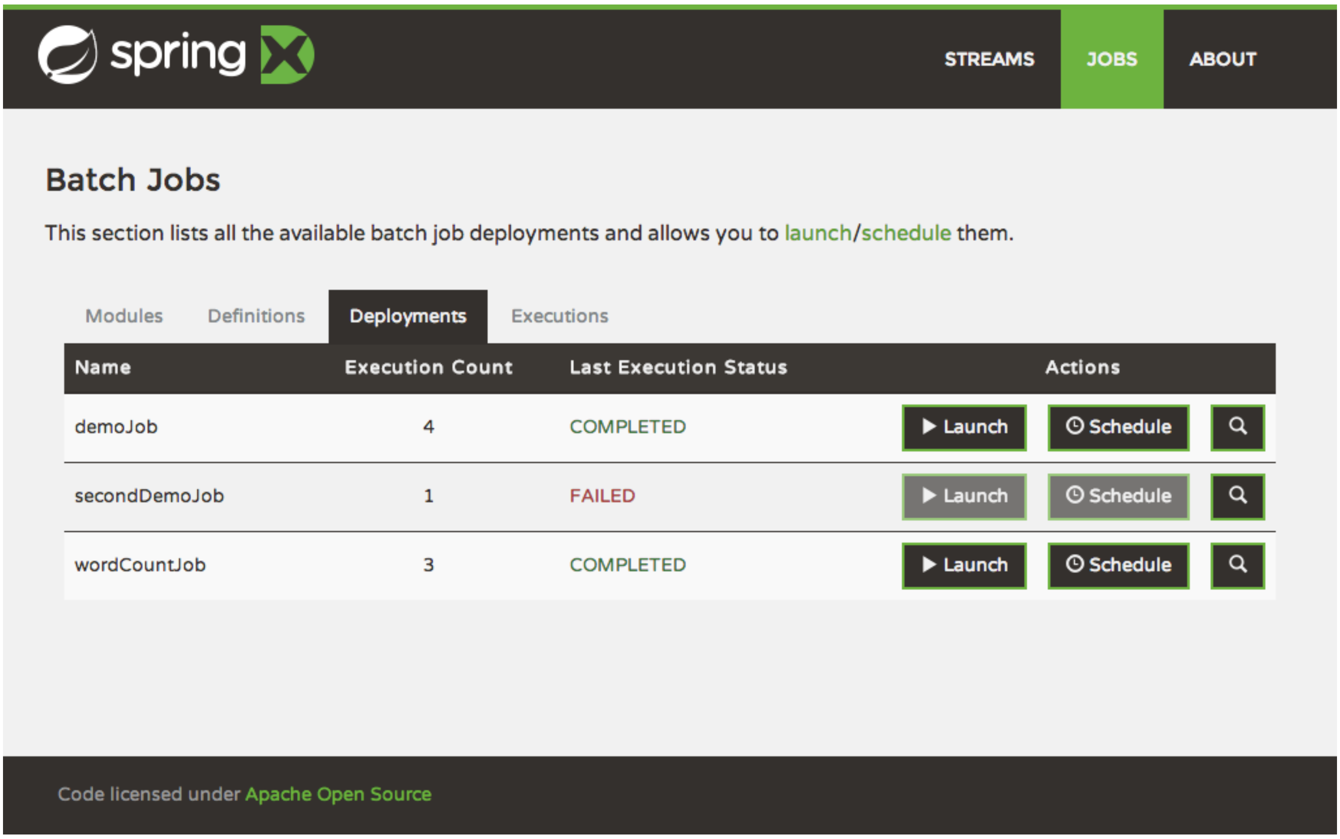
Name	Definition	Actions		
demoJob	myjob --dateFormat=yyyy-MM-dd --listeners=job --makeUnique=true	■ Undeploy	▶ Deploy	✖ Destroy
secondDemoJob	myjob --makeUnique=true	■ Undeploy	▶ Deploy	✖ Destroy
wordCountJob	wordcount	■ Undeploy	▶ Deploy	✖ Destroy

Code licensed under Apache Open Source

Figure 6.6. List Job Definitions

List job deployments

This page lists all the deployed jobs and provides option to **launch** or **schedule** the deployed job.



Batch Jobs

This section lists all the available batch job deployments and allows you to launch/schedule them.

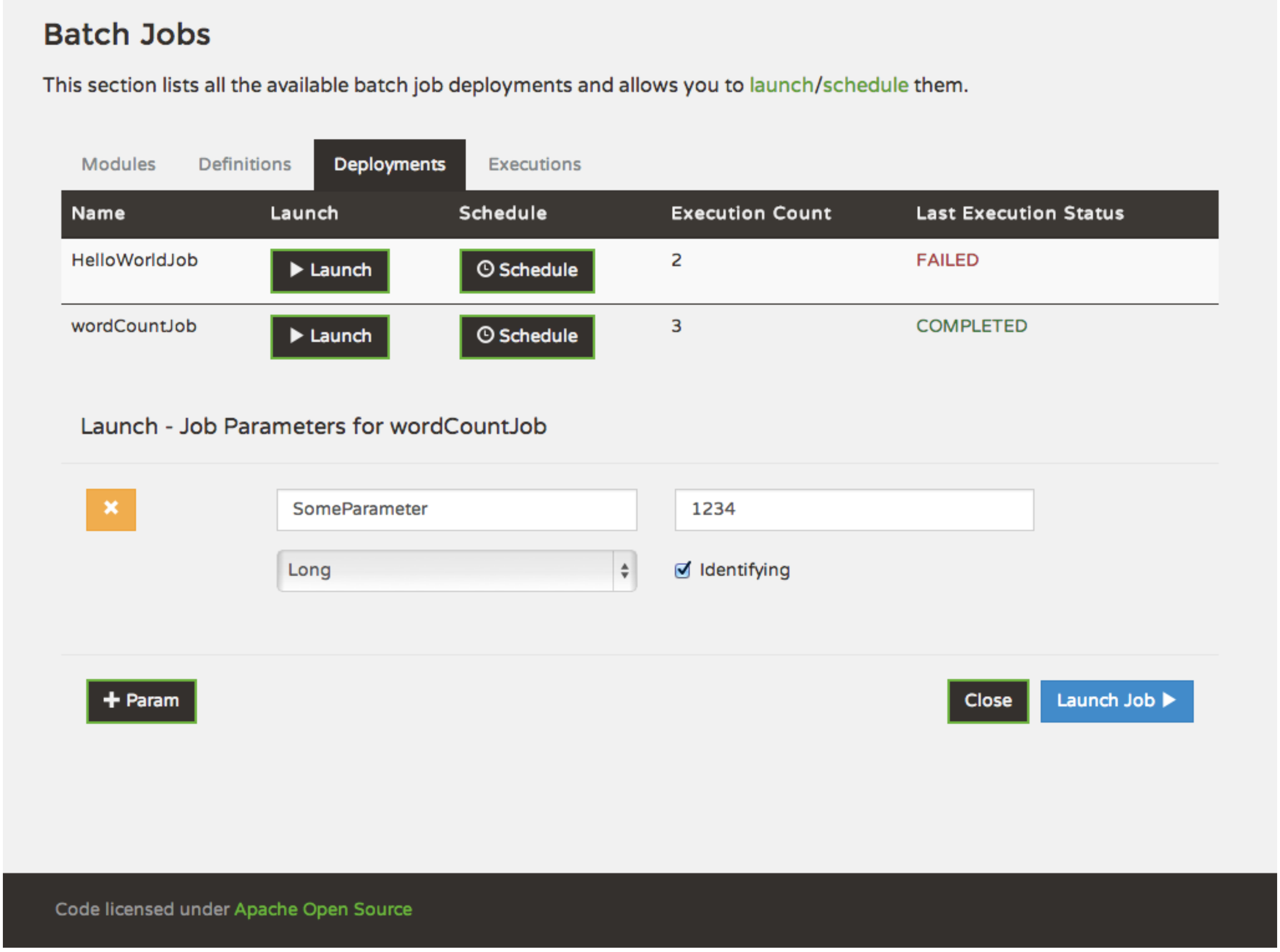
Name	Execution Count	Last Execution Status	Actions
demoJob	4	COMPLETED	▶ Launch Schedule
secondDemoJob	1	FAILED	▶ Launch Schedule
wordCountJob	3	COMPLETED	▶ Launch Schedule

Code licensed under Apache Open Source

Figure 6.7. List Job Deployments

Launching a batch Job

Once the job is deployed, they can be launched through the Admin UI as well. Navigate to the **Deployments** tab. Select the job you want to launch and press `Launch`. The following modal dialog should appear:



Batch Jobs

This section lists all the available batch job deployments and allows you to launch/schedule them.

Name	Launch	Schedule	Execution Count	Last Execution Status
HelloWorldJob	▶ Launch	Schedule	2	FAILED
wordCountJob	▶ Launch	Schedule	3	COMPLETED

Launch - Job Parameters for wordCountJob

SomeParameter 1234

Long ☑ Identifying

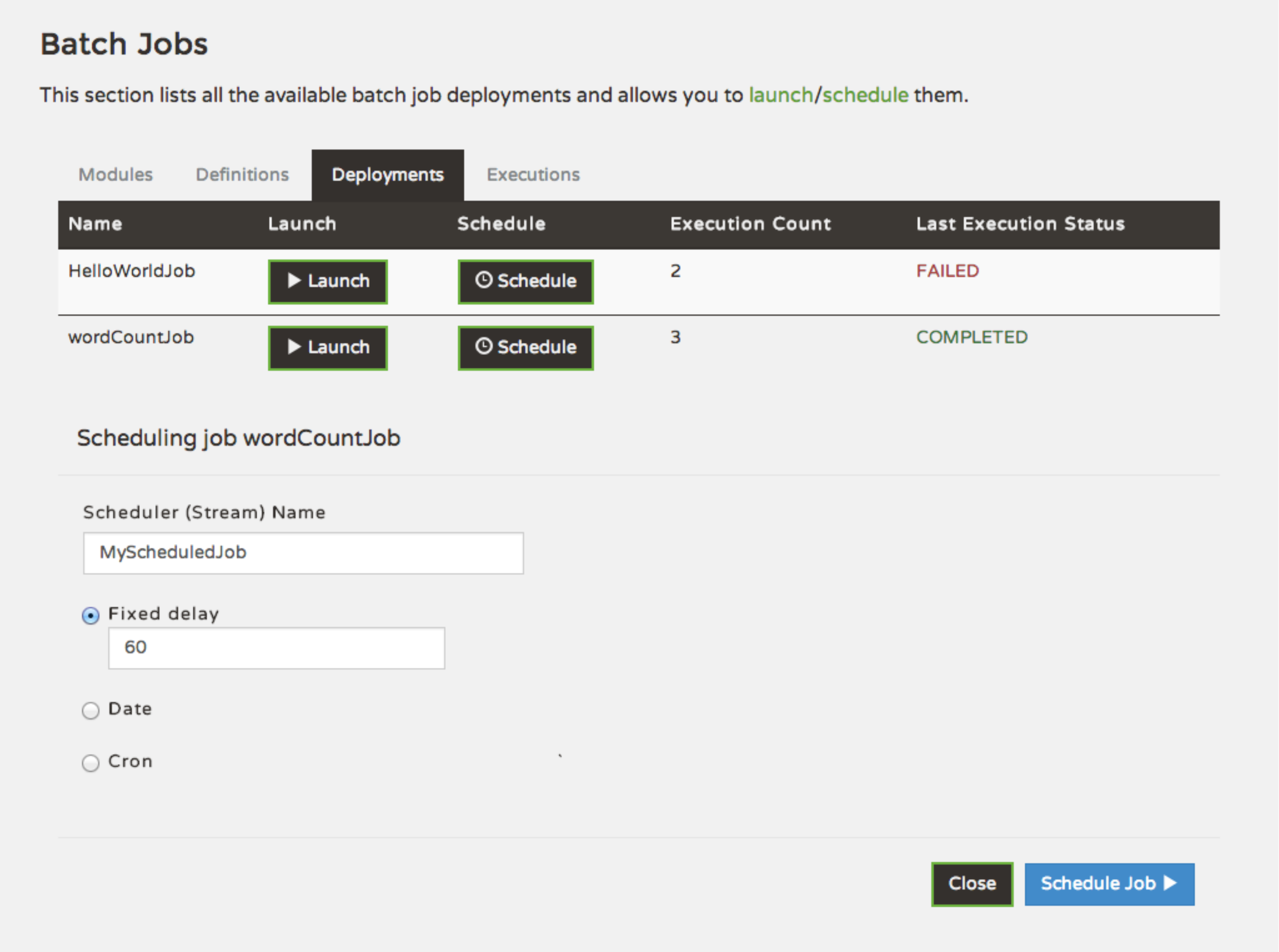
Code licensed under Apache Open Source

Figure 6.8. Launch a Batch Job with parameters

Using this screen, you can define one or more job parameters. Job parameters can be typed and the following data types are available:

- String (The default)
- Date (The default date format is: *yyyy/MM/dd*)
- Long
- Double

Schedule Batch Job Execution



Batch Jobs

This section lists all the available batch job deployments and allows you to launch/schedule them.

Name	Launch	Schedule	Execution Count	Last Execution Status
HelloWorldJob	▶ Launch	Schedule	2	FAILED
wordCountJob	▶ Launch	Schedule	3	COMPLETED

Scheduling job wordCountJob

Scheduler (Stream) Name: MyScheduledJob

● Fixed delay: 60

○ Date

○ Cron

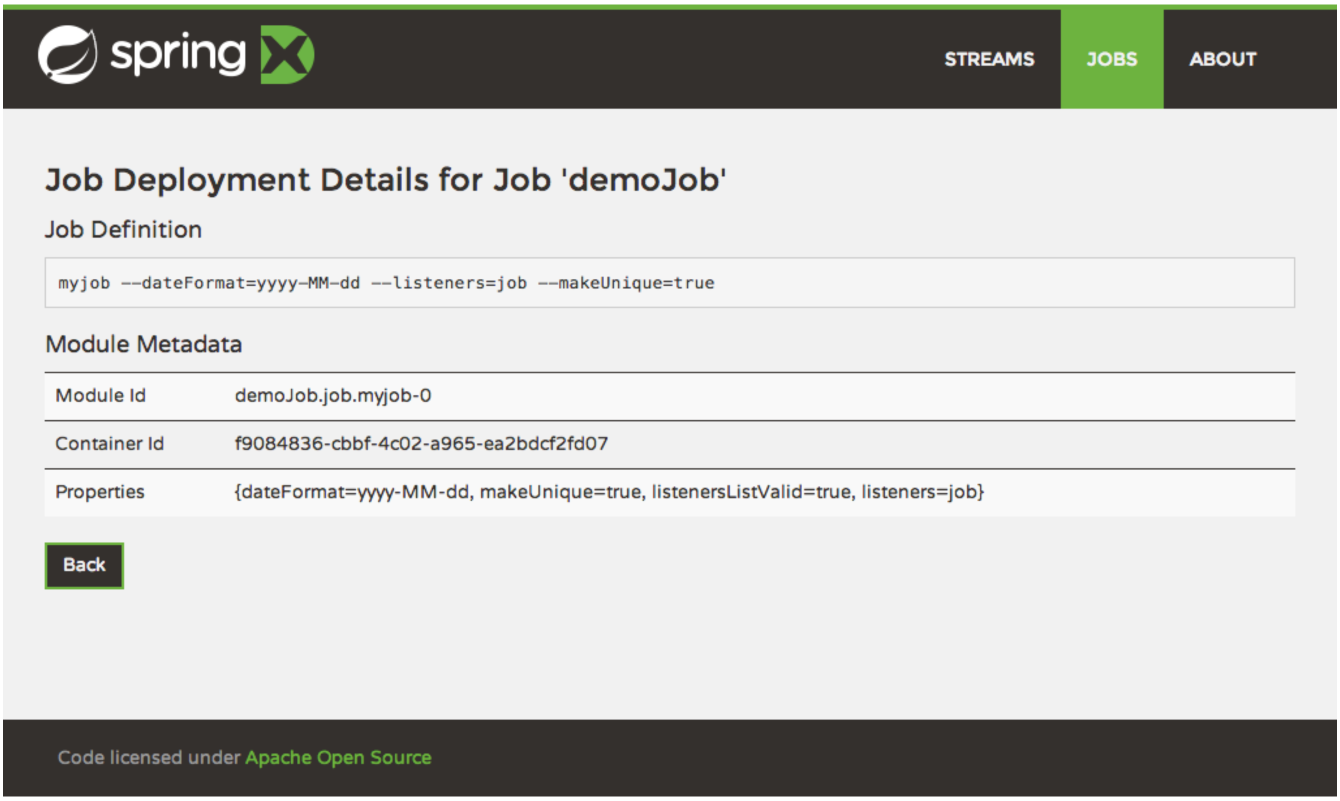
Figure 6.9. Schedule a Batch Job

When clicking on **Schedule**, you have the option to run the job:

- using a fixed delay interval (specified in seconds)
- on a specific data/time
- using a valid CRON expression

Job Deployment Details

On this screen, you can view additional deployment details. Besides viewing the stream definition, the available Module Metadata is shown as well, e.g. on which Container the definition has been deployed to.



Job Deployment Details for Job 'demoJob'

Job Definition

```
myjob --dateFormat=yyyy-MM-dd --listeners=job --makeUnique=true
```

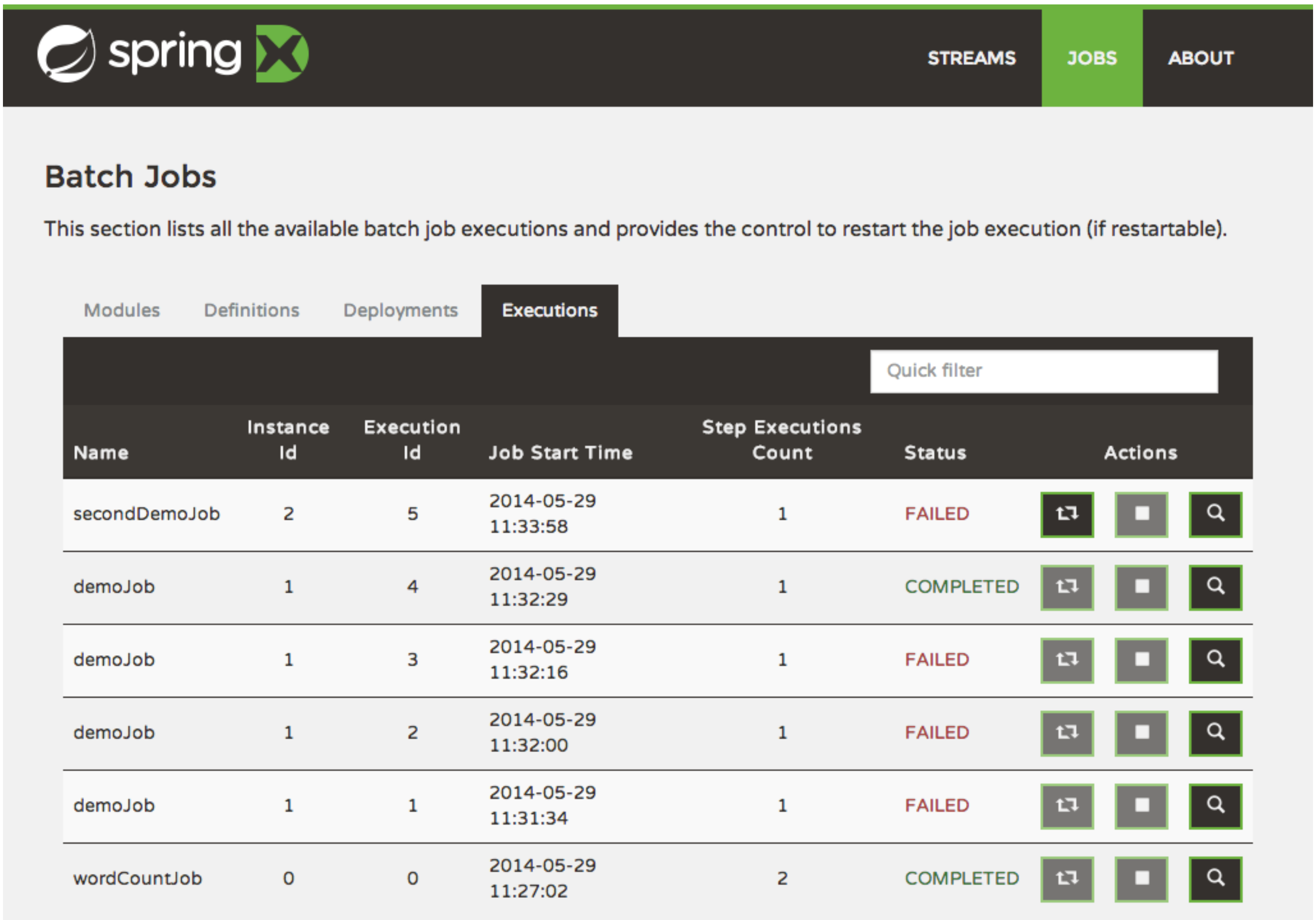
Module Metadata

Module Id	demoJob.job.myjob-0
Container Id	f9084836-cbbf-4c02-a965-ea2bdcf2fd07
Properties	{dateFormat=yyyy-MM-dd, makeUnique=true, listenersListValid=true, listeners=job}

Figure 6.10. Job Deployment Details

List job executions

This page lists the batch job executions and provides option to **restart** specific job executions, provided the batch job is restartable and stopped/failed.



Batch Jobs

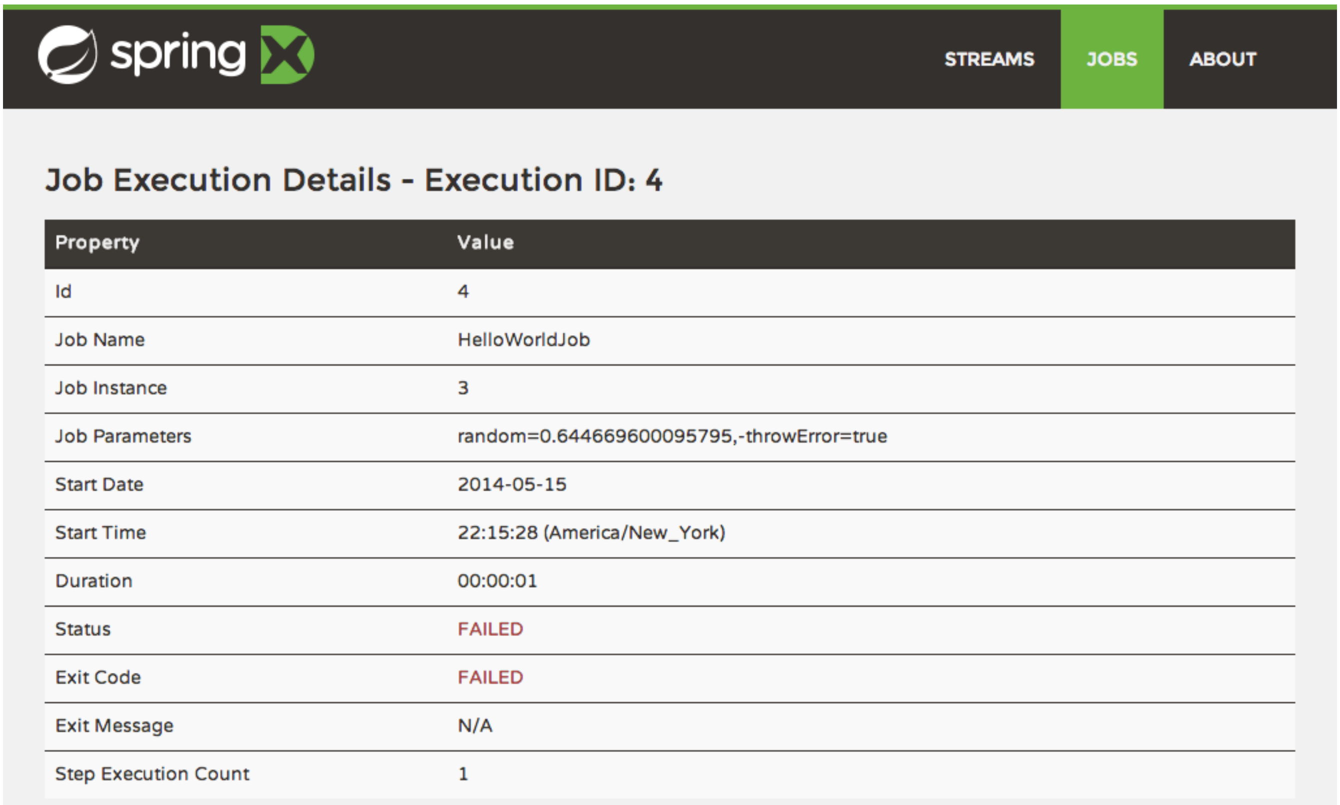
This section lists all the available batch job executions and provides the control to restart the job execution (if restartable).

Name	Instance Id	Execution Id	Job Start Time	Step Executions Count	Status	Actions
secondDemoJob	2	5	2014-05-29 11:33:58	1	FAILED	↻ ■ 🔍
demoJob	1	4	2014-05-29 11:32:29	1	COMPLETED	↻ ■ 🔍
demoJob	1	3	2014-05-29 11:32:16	1	FAILED	↻ ■ 🔍
demoJob	1	2	2014-05-29 11:32:00	1	FAILED	↻ ■ 🔍
demoJob	1	1	2014-05-29 11:31:34	1	FAILED	↻ ■ 🔍
wordCountJob	0	0	2014-05-29 11:27:02	2	COMPLETED	↻ ■ 🔍

Figure 6.11. List Job Executions

Furthermore, you have the option to view the Job execution details.

Job execution details

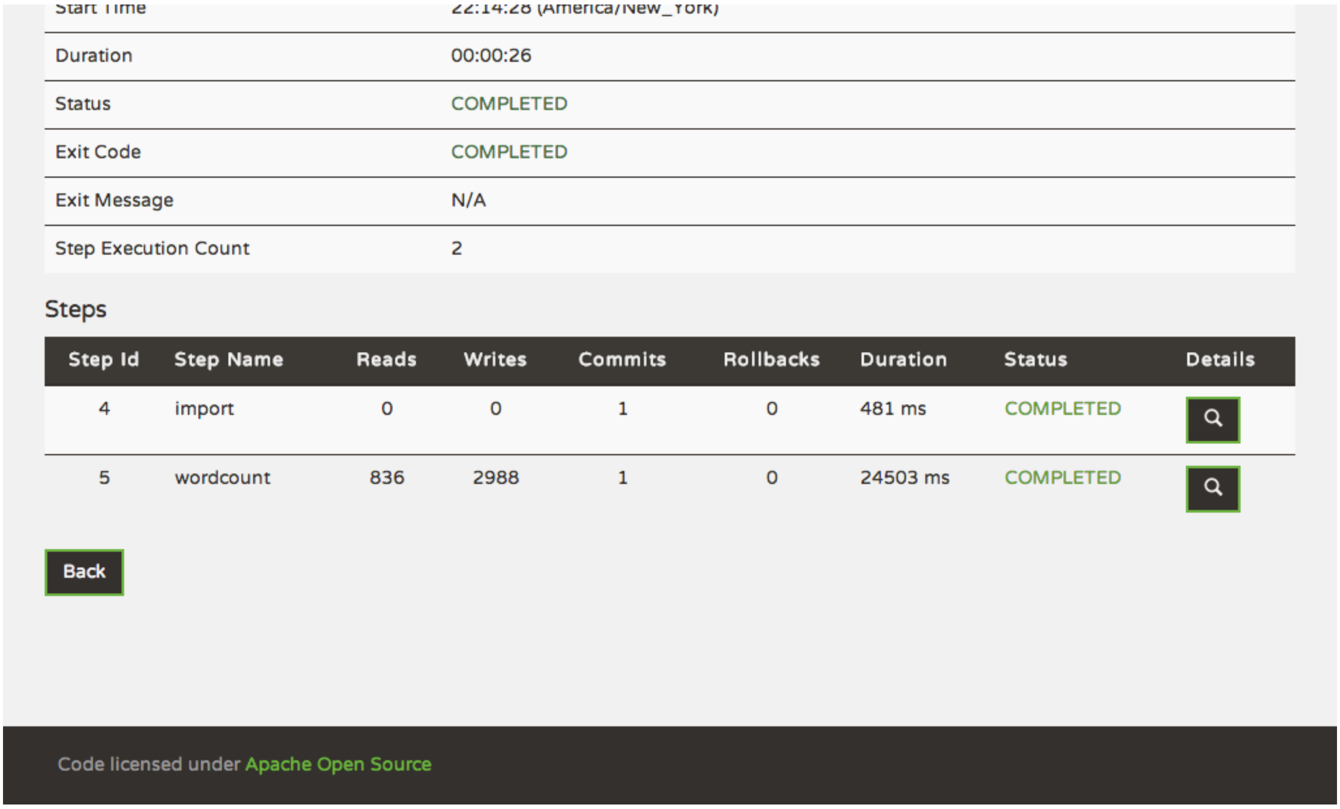


Job Execution Details - Execution ID: 4

Property	Value
Id	4
Job Name	HelloWorldJob
Job Instance	3
Job Parameters	random=0.644669600095795,-throwError=true
Start Date	2014-05-15
Start Time	22:15:28 (America/New_York)
Duration	00:00:01
Status	FAILED
Exit Code	FAILED
Exit Message	N/A
Step Execution Count	1

Figure 6.12. Job Execution Details

The same screen also contains a list of the executed steps:



Start Time	22:14:28 (America/New_York)
Duration	00:00:26
Status	COMPLETED
Exit Code	COMPLETED
Exit Message	N/A
Step Execution Count	2

Steps

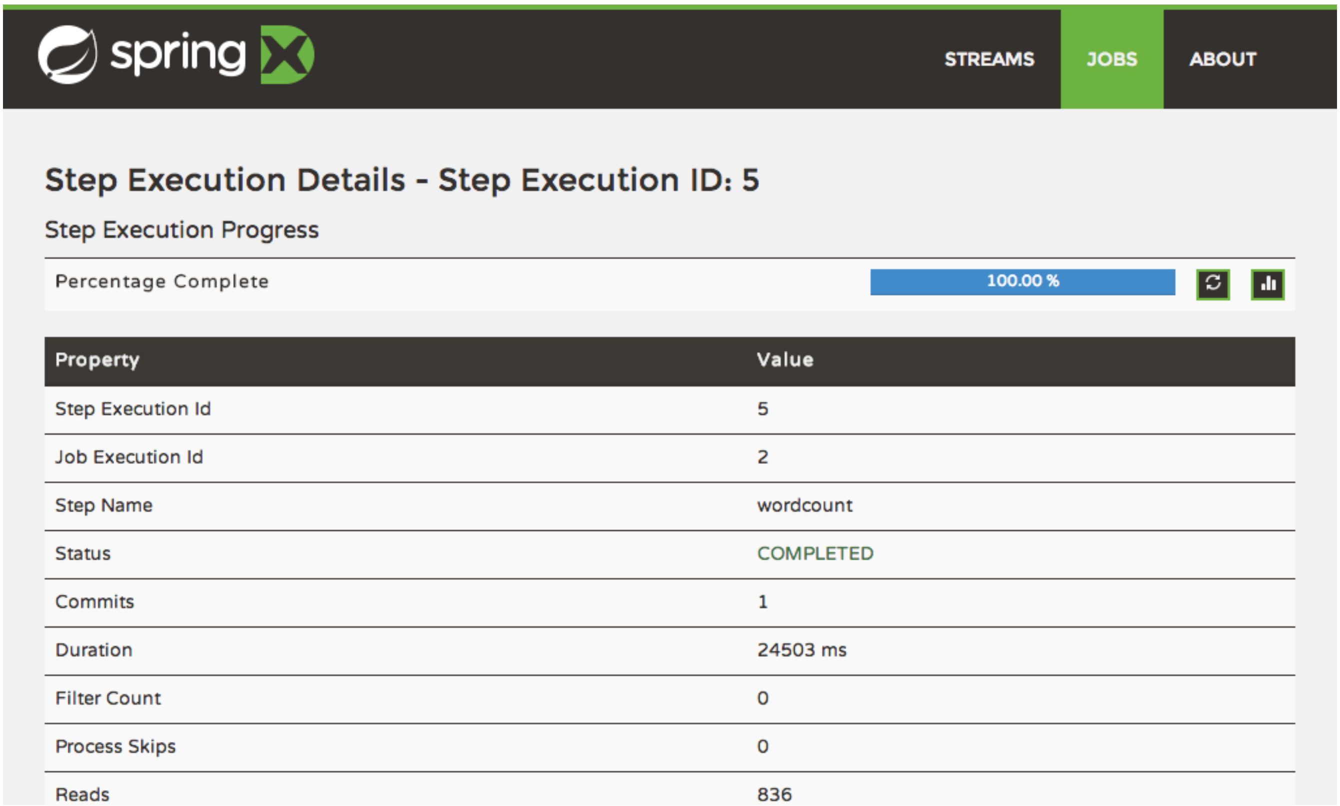
Step Id	Step Name	Reads	Writes	Commits	Rollbacks	Duration	Status	Details
4	import	0	0	1	0	481 ms	COMPLETED	🔍
5	wordcount	836	2988	1	0	24503 ms	COMPLETED	🔍

Figure 6.13. Job Execution Details - Steps

From there you can drill deeper into the *Step Execution Details*.

Step execution details

On the top of the page, you will see progress indicator the respective step, with the option to refresh the indicator. Furthermore, a link is provided to view the *step execution history*.



Step Execution Details - Step Execution ID: 5

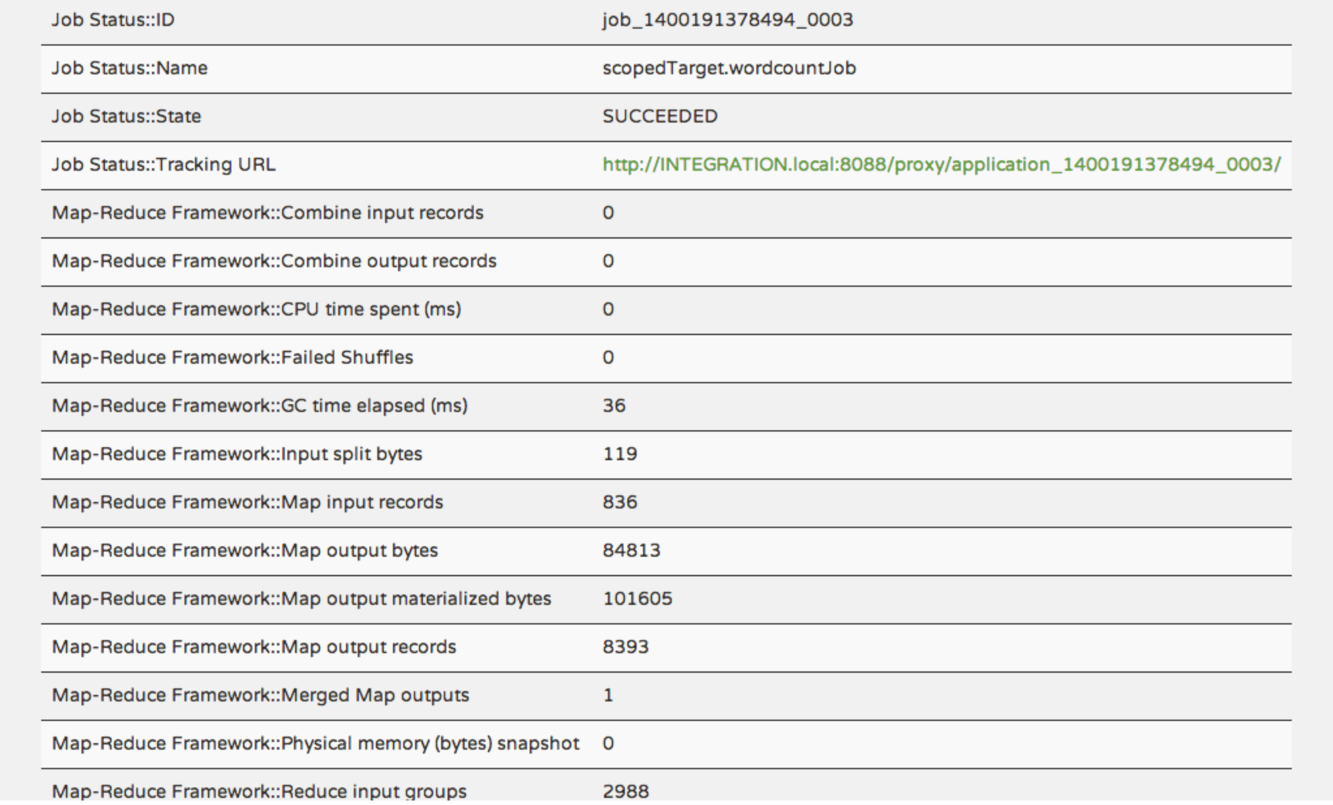
Step Execution Progress

Percentage Complete 100.00 %

Property	Value
Step Execution Id	5
Job Execution Id	2
Step Name	wordcount
Status	COMPLETED
Commits	1
Duration	24503 ms
Filter Count	0
Process Skips	0
Reads	836

Figure 6.14. Step Execution Details

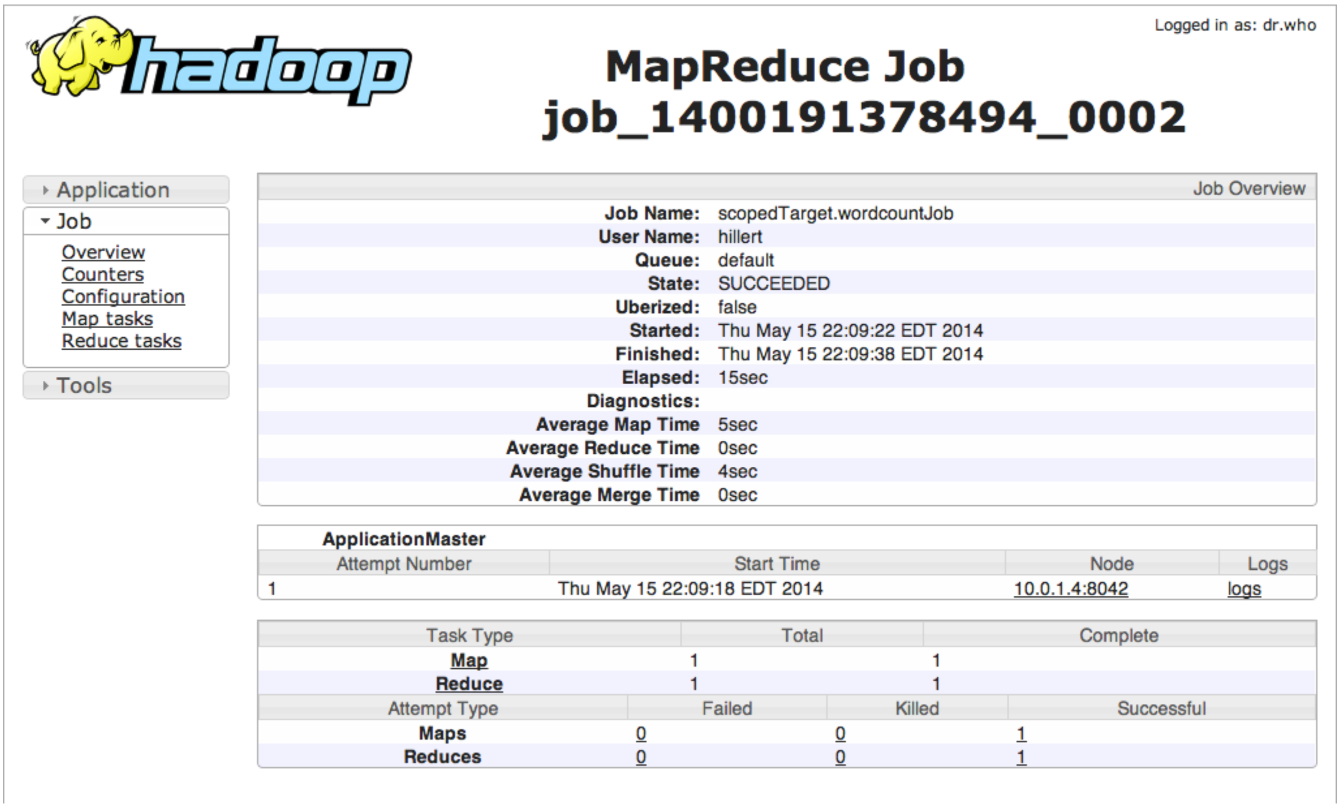
The Step Execution details screen provides a complete list of all Step Execution Context key/value pairs. For example, the *Spring for Apache Hadoop* steps provides exhaustive detail information.



Job Status::ID	job_1400191378494_0003
Job Status::Name	scopedTarget.wordcountJob
Job Status::State	SUCCEEDED
Job Status::Tracking URL	http://INTEGRATION.local:8088/proxy/application_1400191378494_0003/
Map-Reduce Framework::Combine input records	0
Map-Reduce Framework::Combine output records	0
Map-Reduce Framework::CPU time spent (ms)	0
Map-Reduce Framework::Failed Shuffles	0
Map-Reduce Framework::GC time elapsed (ms)	36
Map-Reduce Framework::Input split bytes	119
Map-Reduce Framework::Map input records	836
Map-Reduce Framework::Map output bytes	84813
Map-Reduce Framework::Map output materialized bytes	101605
Map-Reduce Framework::Map output records	8393
Map-Reduce Framework::Merged Map outputs	1
Map-Reduce Framework::Physical memory (bytes) snapshot	0
Map-Reduce Framework::Reduce input groups	2988

Figure 6.15. Step Execution Context

This includes a link back to the *Job History UI* of the Hadoop Cluster.



MapReduce Job job_1400191378494_0002

Job Overview

Job Name:	scopedTarget.wordcountJob
User Name:	hillert
Queue:	default
State:	SUCCEEDED
Uberized:	false
Started:	Thu May 15 22:09:22 EDT 2014
Finished:	Thu May 15 22:09:38 EDT 2014
Elapsed:	15sec
Diagnostics:	
Average Map Time	5sec
Average Reduce Time	0sec
Average Shuffle Time	4sec
Average Merge Time	0sec

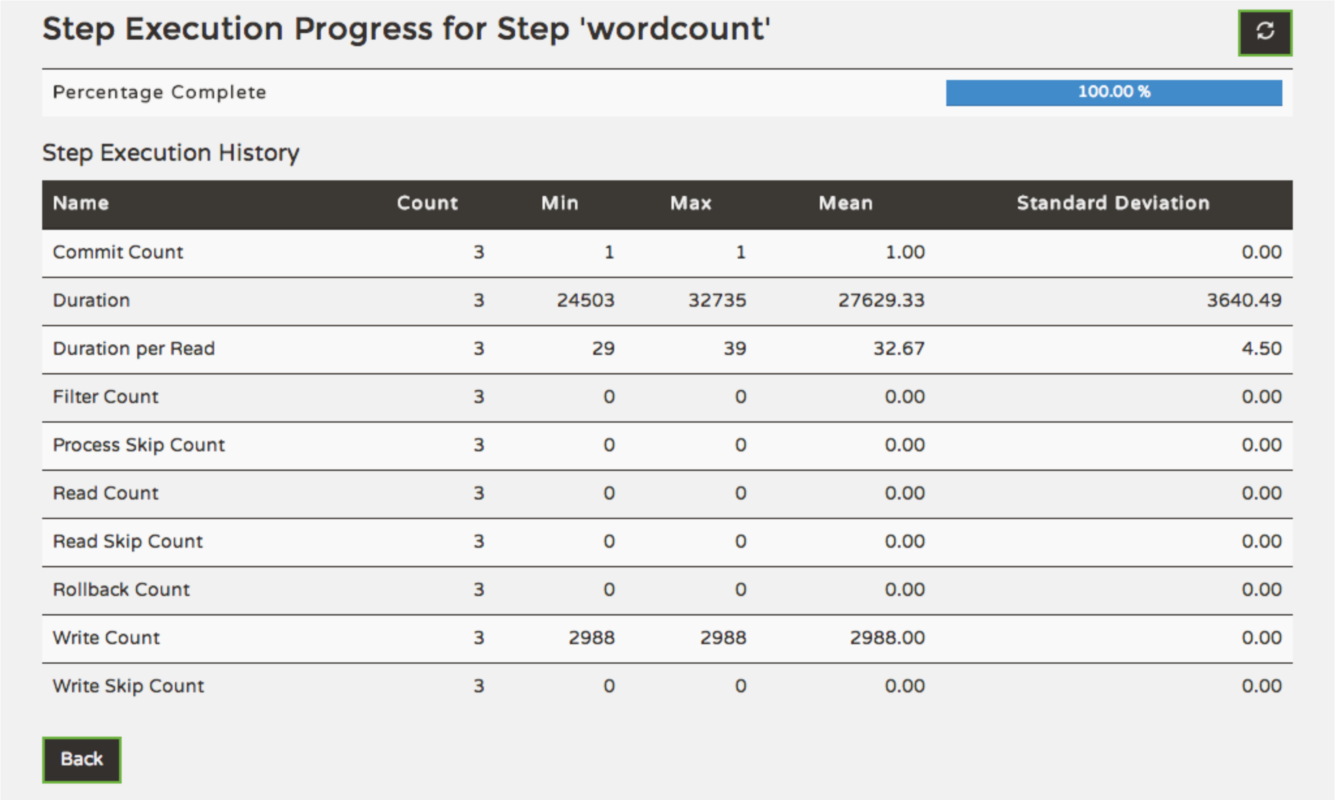
ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Thu May 15 22:09:18 EDT 2014	10.0.1.4:8042	logs

Task Type	Total	Complete	
Map	1	1	
Reduce	1	1	
Attempt Type	Failed	Killed	Successful
Maps	0	0	1
Reduces	0	0	1

Figure 6.16. Job History UI

Step execution history



Step Execution Progress for Step 'wordcount'

Percentage Complete 100.00 %

Step Execution History

Name	Count	Min	Max	Mean	Standard Deviation
Commit Count	3	1	1	1.00	0.00
Duration	3	24503	32735	27629.33	3640.49
Duration per Read	3	29	39	32.67	4.50
Filter Count	3	0	0	0.00	0.00
Process Skip Count	3	0	0	0.00	0.00
Read Count	3	0	0	0.00	0.00
Read Skip Count	3	0	0	0.00	0.00
Rollback Count	3	0	0	0.00	0.00
Write Count	3	2988	2988	2988.00	0.00
Write Skip Count	3	0	0	0.00	0.00

Figure 6.17. Step Execution History

On this screen, you can view various metrics associated with the selected step such as **duration**, **read counts**, **write counts** etc.

7. Architecture

7.1 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 reacts to user commands and runtime events managed within a shared runtime state to coordinate 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 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 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 assigns each module to a container instance using ZooKeeper (see XD Distributed Runtime). Each container listens for module definitions to which it has been assigned and deploys the module, creating a Spring ApplicationContext to run it.

Modules share data by passing messages using a configured messaging middleware (Rabbit, Redis, or Local for single node). 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.

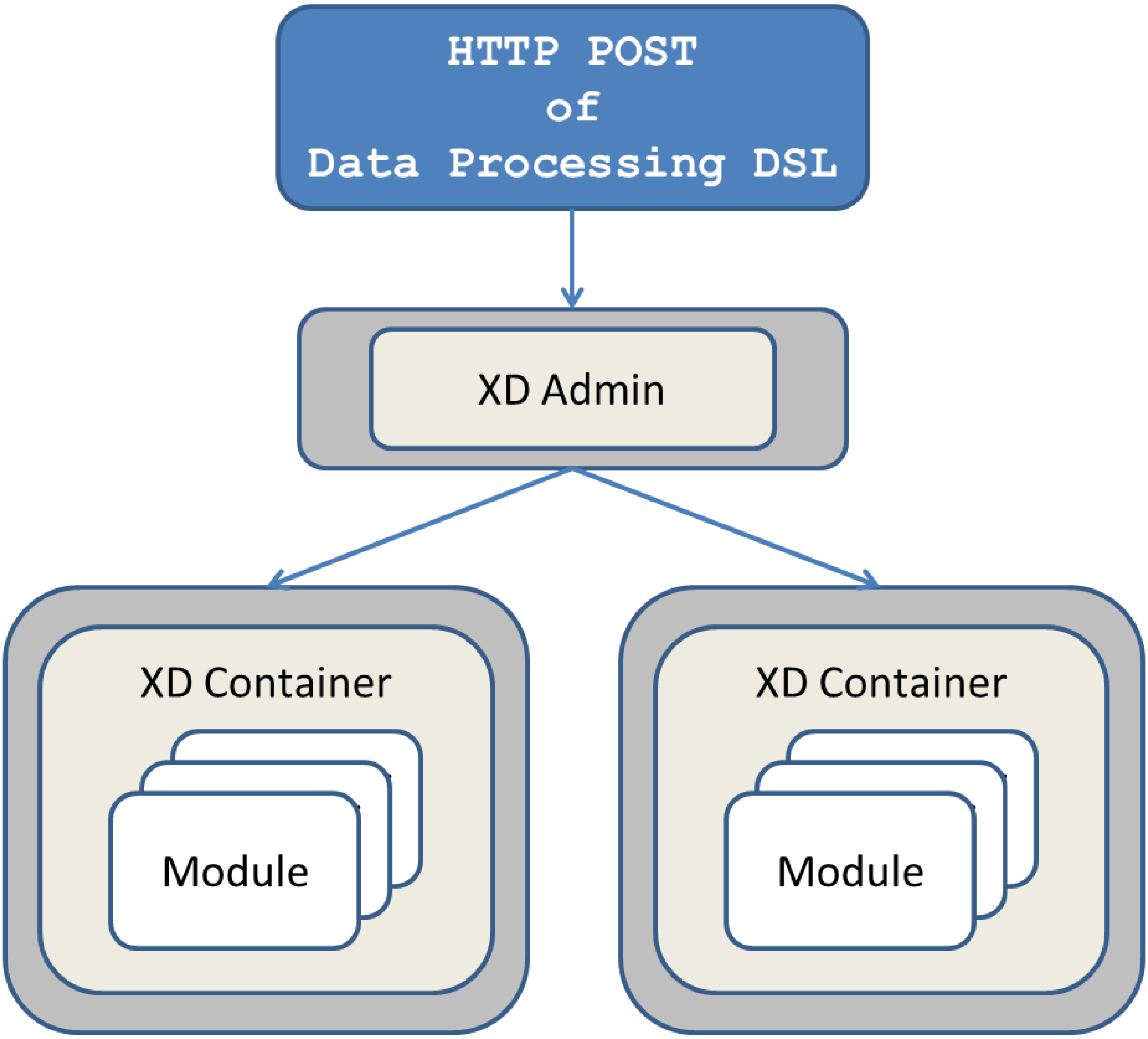


Figure 7.1. The XD Admin Server sending module definitions to each XD Container

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 can run Spring XD natively, in which case you are responsible for starting up the XD Admin and XD Container instances. Alternately you can run Spring XD on Hadoop's YARN, see Running XD on YARN. Pivotal Cloud Foundry support is planned for a future release. If you are feeling a adventurous, you can also take a look at our scripts for deploying Spring XD to EC2. These are used as part of our system integration tests.

Single Node Runtime

A single node runtime is provided that runs the Admin and Container servers, ZooKeeper, and HSQLDB in the same process. the single node runtime is primarily intended for testing and development purposes but it may also appropriate to use in small production use-cases. The communication to the XD Admin server is over HTTP and the XD Admin server communicates to an in-process XD Container using an embedded ZooKeeper server.

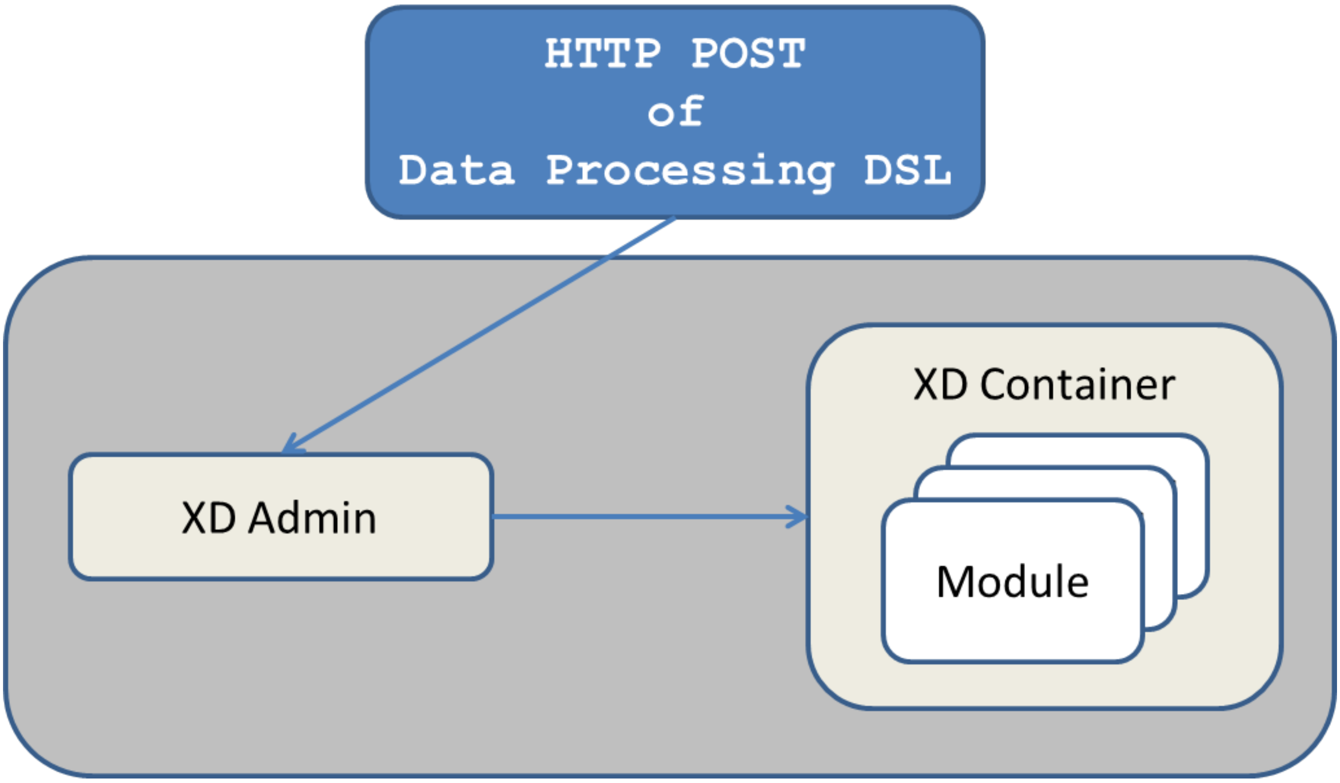


Figure 7.2. Single Node Runtime

Admin Server Architecture

The Admin Server uses an embedded servlet container and exposes REST endpoints for creating, deploying, undeploying, and destroying streams and jobs, querying runtime state, analytics, and the like. 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 and Container Servers monitor and update runtime state using ZooKeeper (see XD Distributed Runtime).

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.

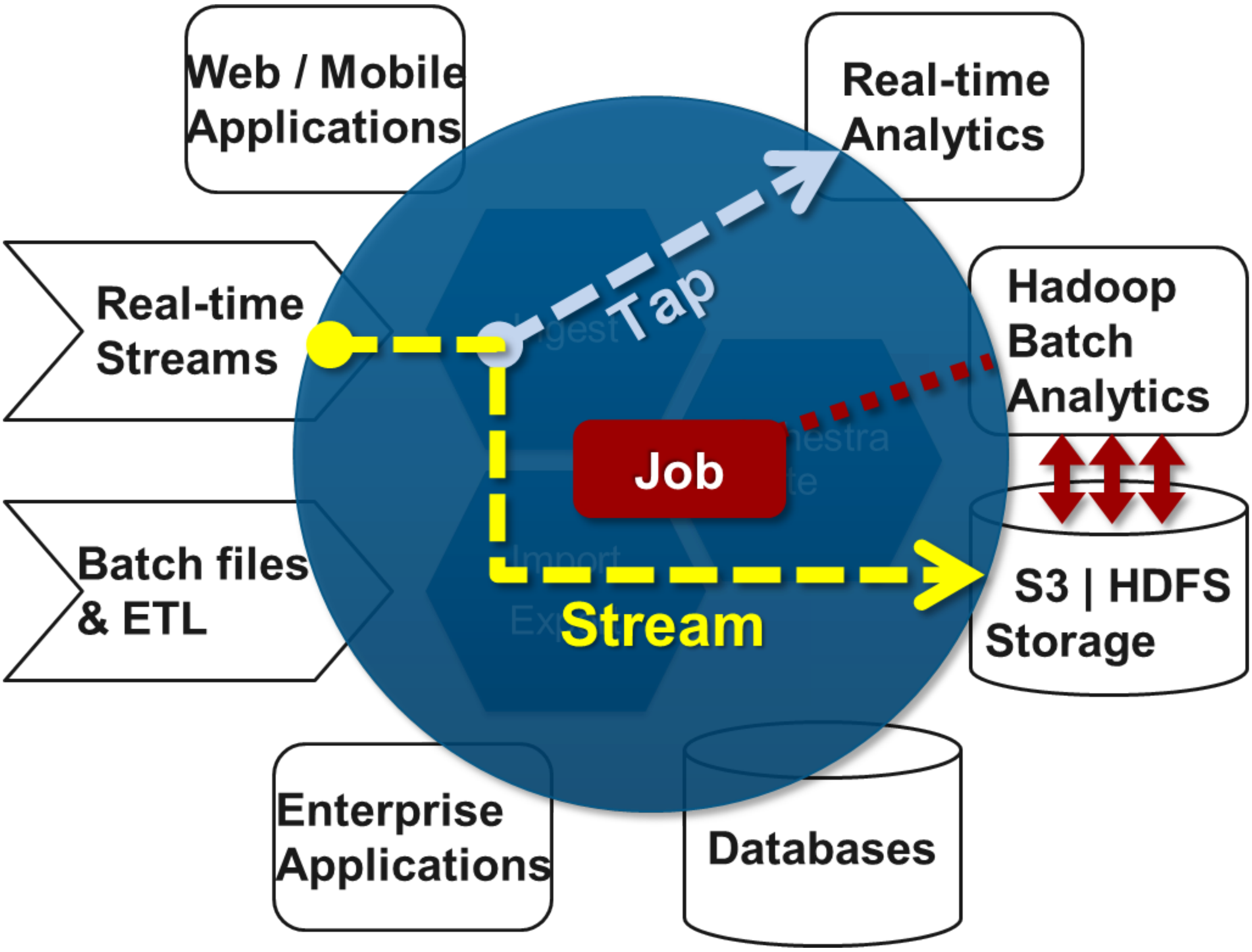


Figure 7.3. Taps, Jobs, and Streams

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 so that it is easy to test components.

A Stream consist of the following types of modules: * An Input source * Processing steps * An Output sink

An Input source produces messages from an external source. XD supports a variety of sources, e.g. syslog, tcp, http. The output from a module is a Spring Message containing 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 delivers the message to an external resource. For example, it is common to write the message to a file system, such as HDFS, but you may also configure the sink to forward the message over tcp, http, or another type of middleware, or route the message to another stream.

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

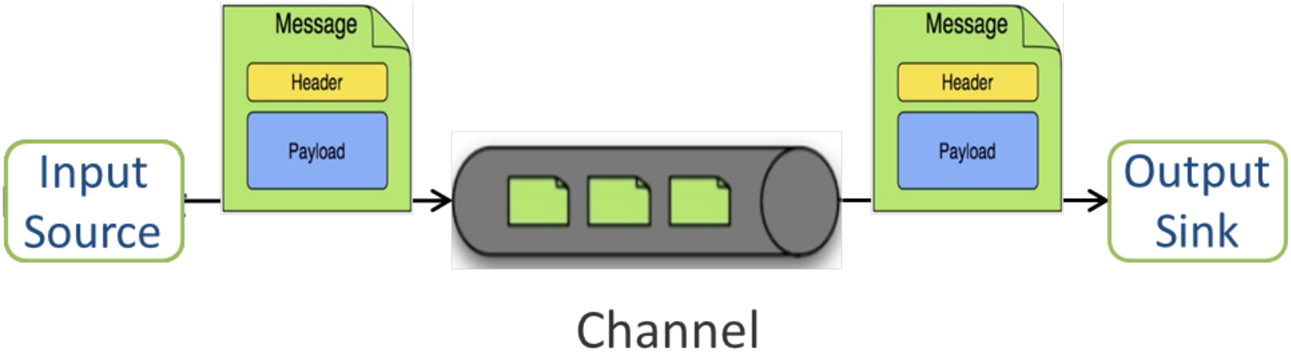


Figure 7.4. Foundational components of the Stream processing model

A stream that incorporates processing steps is shown below

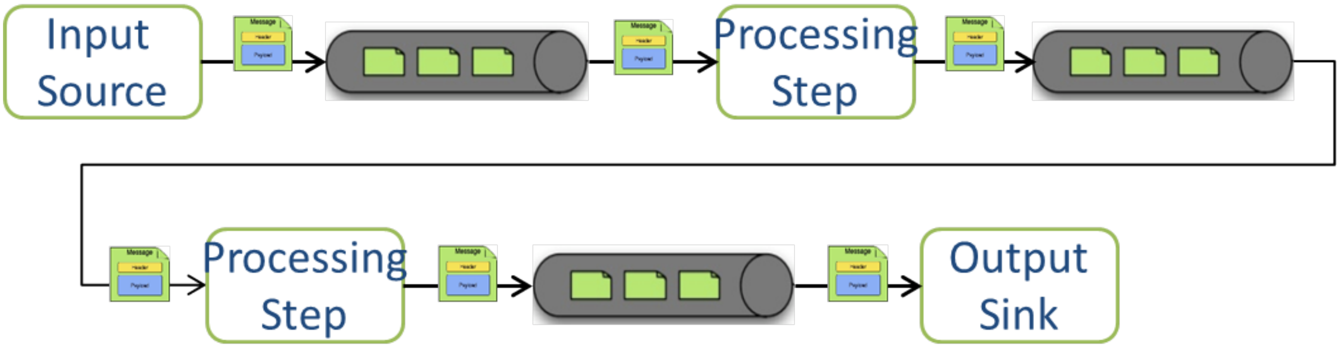


Figure 7.5. Stream processing with multiple steps

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 the modules (source, processing steps, and sink) 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. The message channel abstraction and the XD architecture are designed to support a pluggable data transport. Future releases will support 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. Non-linear processing is partially supported using named channels which can be combined with a router sink to effectively split a single stream into multiple streams (see Dynamic Router Sink). Additional capabilities for non-linear processing are planned for future releases.

The programming model for processing steps in a stream originates from the Spring Integration project and is included in the core Spring Framework as of version 4. 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 events initiated from the Admin Server via ZooKeeper. When the container node handles a module deployment event, 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, the data bus uses in-memory direct channels. In a distributed configuration, the data bus communications are backed by the configured transport middleware. Redis and Rabbit are both provided with the Spring XD distribution, but other transports are envisioned for future releases.

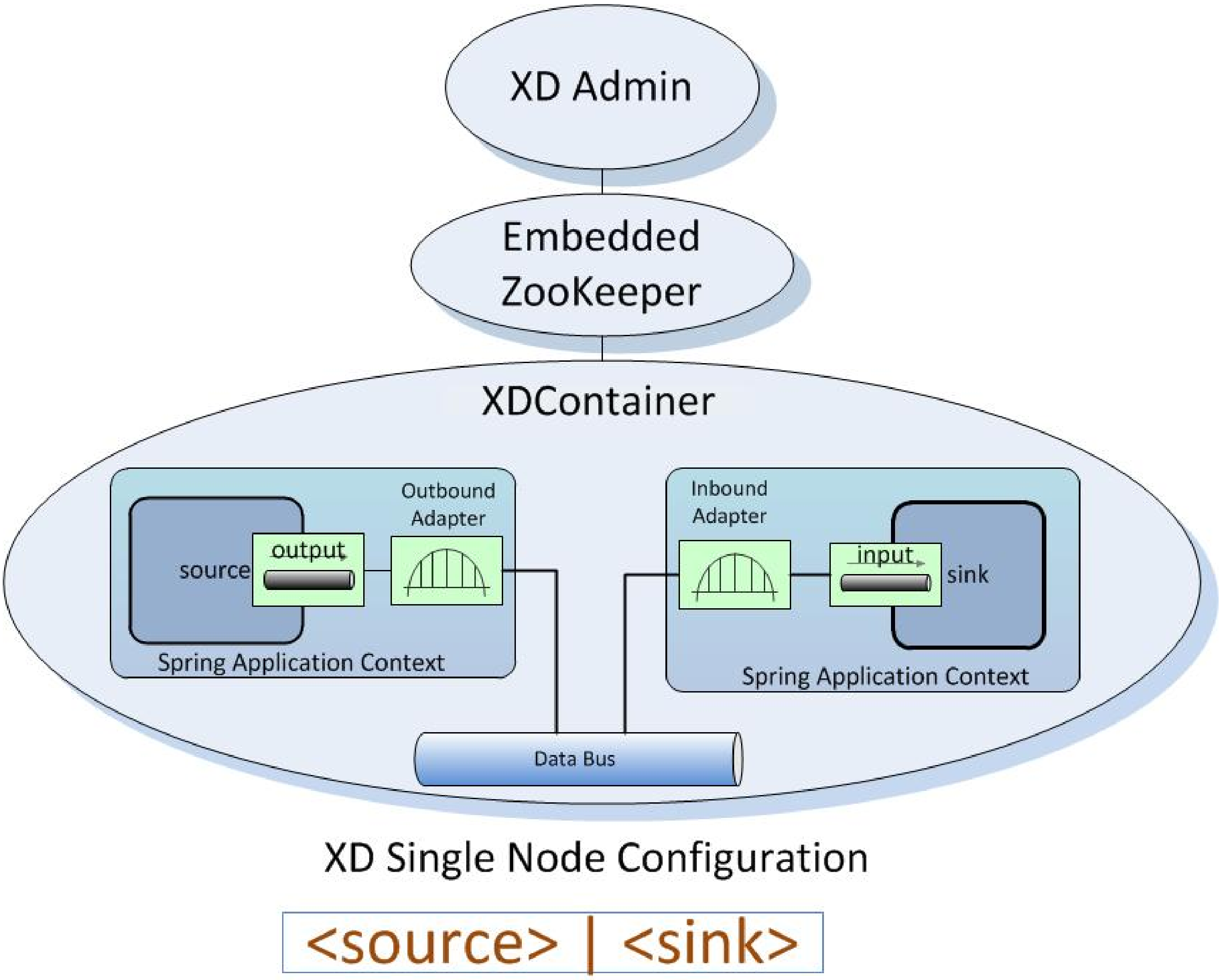


Figure 7.6. A Stream Deployed in a single node server

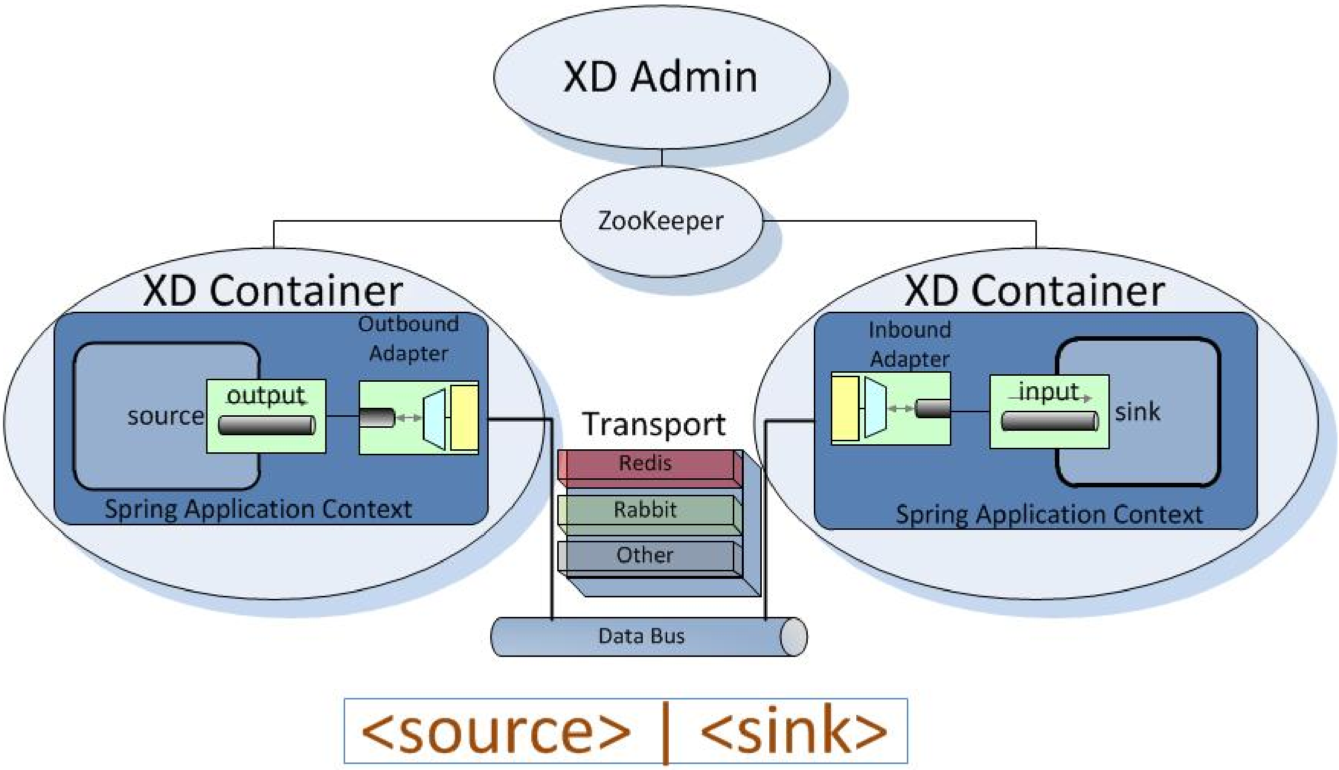


Figure 7.7. A Stream Deployed in a distributed runtime

In the `http | file` example, the Admin assigns each module to a separate Container instance, provided there are at least two Containers available. The `file` module is deployed to one container and the `http` module to another. The definition of a module is stored in a Module Registry. A module definition consists of a Spring XML configuration file, some classes used to validate and handle options defined by the module, and dependent jars. The module definition contains variable placeholders, corresponding to DSL parameters (called *options*) 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 is handled by the Container, the module definition is loaded from the registry and a new Spring ApplicationContext is created in the Container process to run the module. Dependent classes are loaded via the Module Classloader which first looks at jars in the modules /lib directory before delegating to the parent classloader.

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

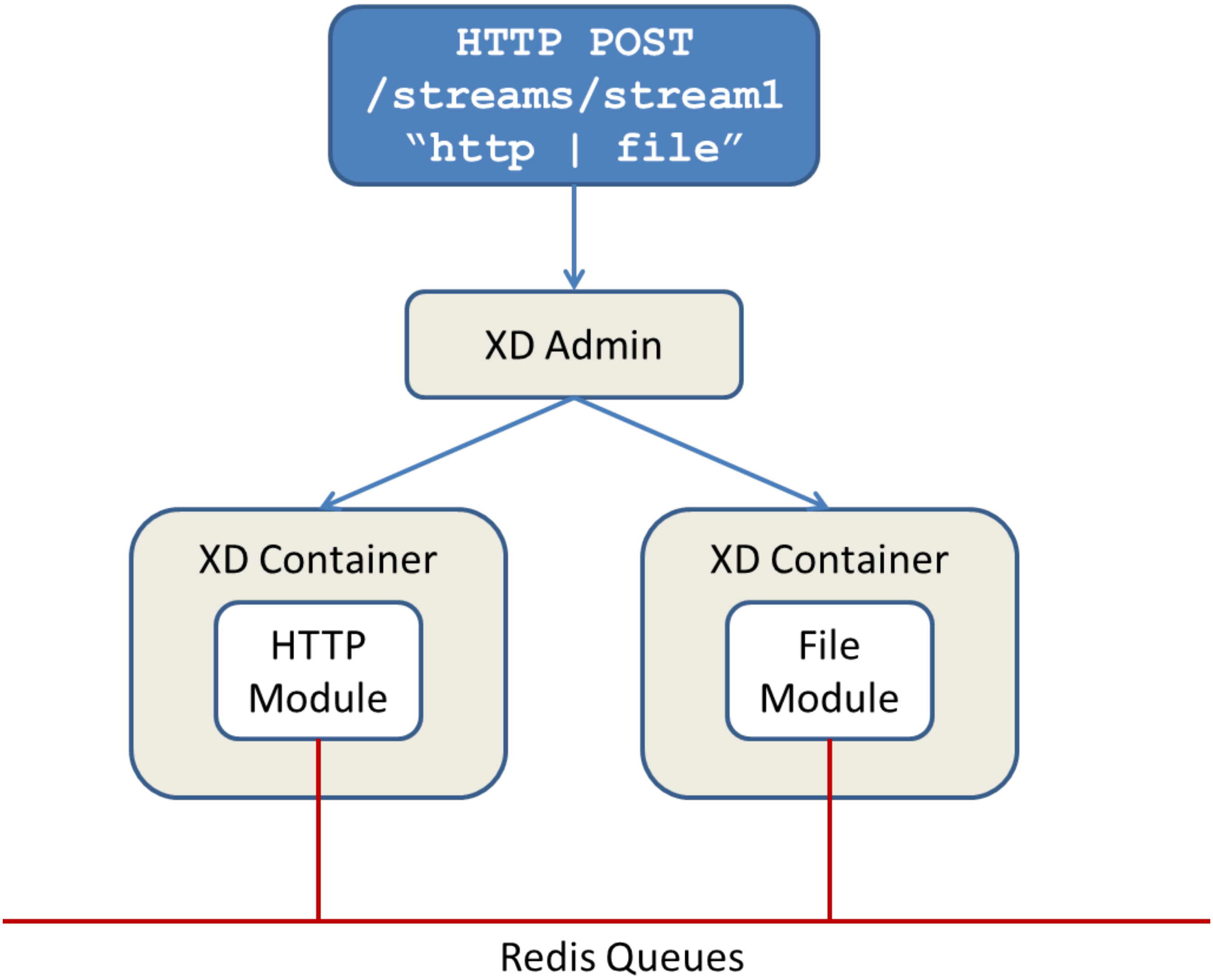


Figure 7.8. Distributed HTTP to File Stream

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.

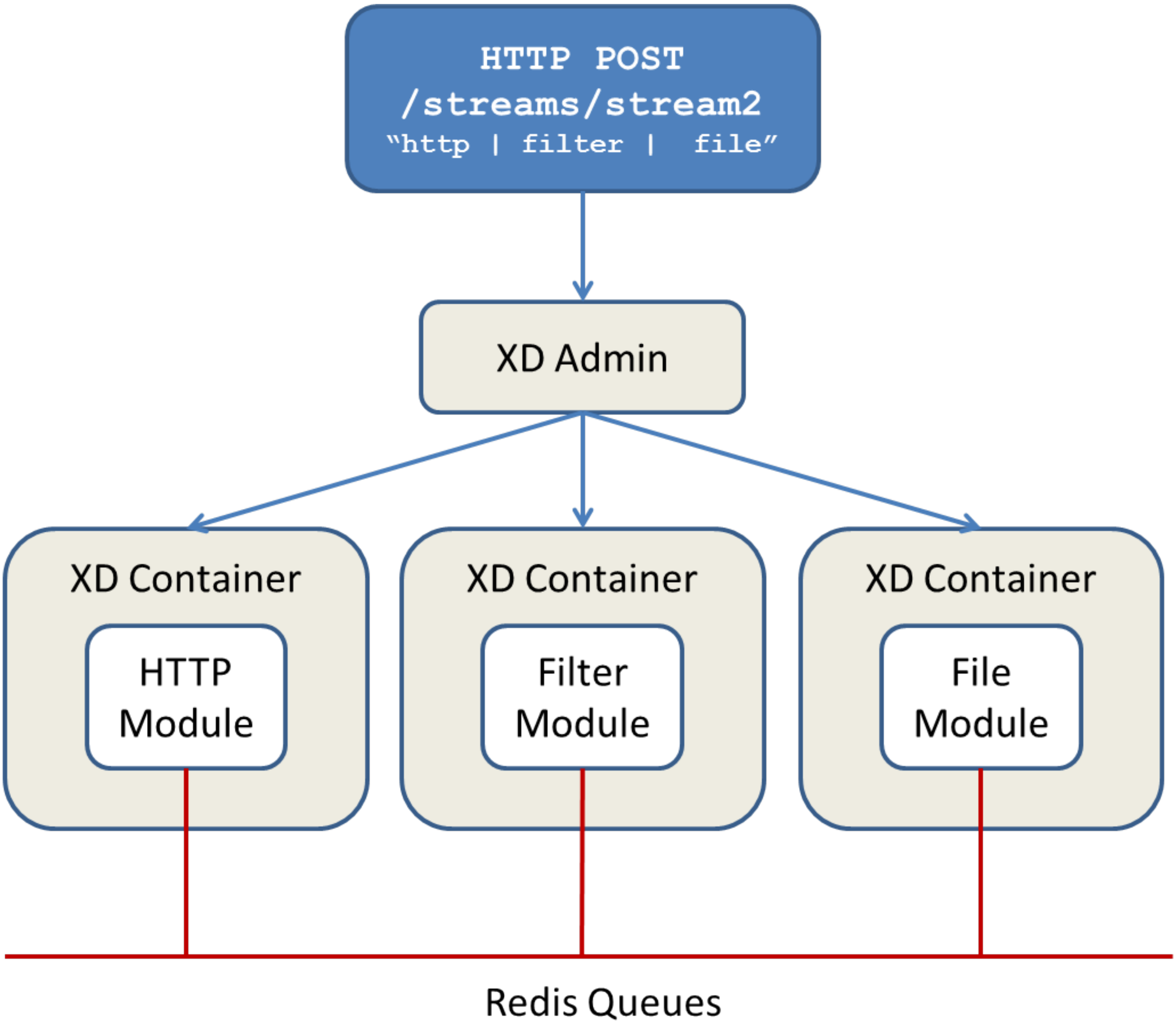


Figure 7.9. Distributed HTTP to Filter to File Stream

7.2 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.

7.3 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.

8. Distributed Runtime

8.1 Introduction

This document describes what’s happening "under the hood" of the Spring XD Distributed Runtime (DIRT) and in particular, how the runtime architecture achieves high availability and failover in a clustered production environment. See Running in Distributed Mode for more information on installing and running Spring XD in distributed mode.

This discussion focuses on Spring XD’s core runtime components and the role of ZooKeeper in managing the state of the Spring XD cluster and enabling automatic recovery from failures.

8.2 Configuring Spring XD for High Availabilty(HA)

A production Spring XD environment is typically distributed among multiple hosts in a clustered environment. Spring XD scales horizontally when you add container instances. In the simplest case, all containers are replicas, that is each instance is running on an identically configured host and modules are deployed to any available container in a round-robin fashion. However, this simplifying assumption does not address real production scenarios in which more control is requred in order to optimize resource utilization. To this end, Spring XD supports a flexible algorithm which allows you to match module deployments to specific container configurations. The container matching algorithm will be covered in more detail later, but for now, let’s assume the simple case. Running multiple containers not only enables horizontal scalability, but enables failure recovery. If a container becomes unavailable due to an unrecoverable connection loss, any modules currently deployed to that container will be deployed automatically to the other available instances.

Spring XD requires that a single active Admin server handle interactions with the containers, such as stream deployment requests, as these types of operations must be processed serially in the order received. Without a backup, the Admin server becomes single point of failure. Therefore, two (or more for the risk averse) Admin servers are recommended for a production environment. Note that every Admin server can handle all requests via REST endpoints but only one instance, the "Leader", will actually perform requests that update the runtime state. If the Leader goes down, another available Admin server will assume the leader role. Leader Election is an example of a common feature for distributed systems provided by the Curator Framework which sits on top of ZooKeeper.

An HA Spring XD installation also requires that external servers - ZooKeeper, messaging middleware, and data stores needed for running Spring XD in distributed mode must be configured for HA as well. Please consult the product documentation for specific recommendations regarding each of these external components. Also see Message Bus Configuration for tips on configuring the MessageBus for HA, error handling, etc.

8.3 ZooKeeper Overview

In the previous section, we claimed that if a container goes down, Spring XD will redeploy any modules deployed on that instance to another available container. We also claimed that if the Admin Leader goes down, another Admin server will assume that role. ZooKeeper is what makes this all possible. ZooKeeper is a widely used Apache project designed primarily for distributed system management and coordination. This section will cover some basic concepts necessary to understand its role in Spring XD. See The ZooKeeper Wiki for a more complete overview.

ZooKeeper is based on a simple hierarchical data structure, formally a tree, and conceptually and semantically similar to a file directory structure. As such, data is stored in *nodes*. A node is referenced via a *path*, for example, */xd/streams/mystream*. Each node can store additional data, serialized as a byte array. In Spring XD, all data is a java.util.Map serialized as JSON. The following figure shows the Spring XD schema:

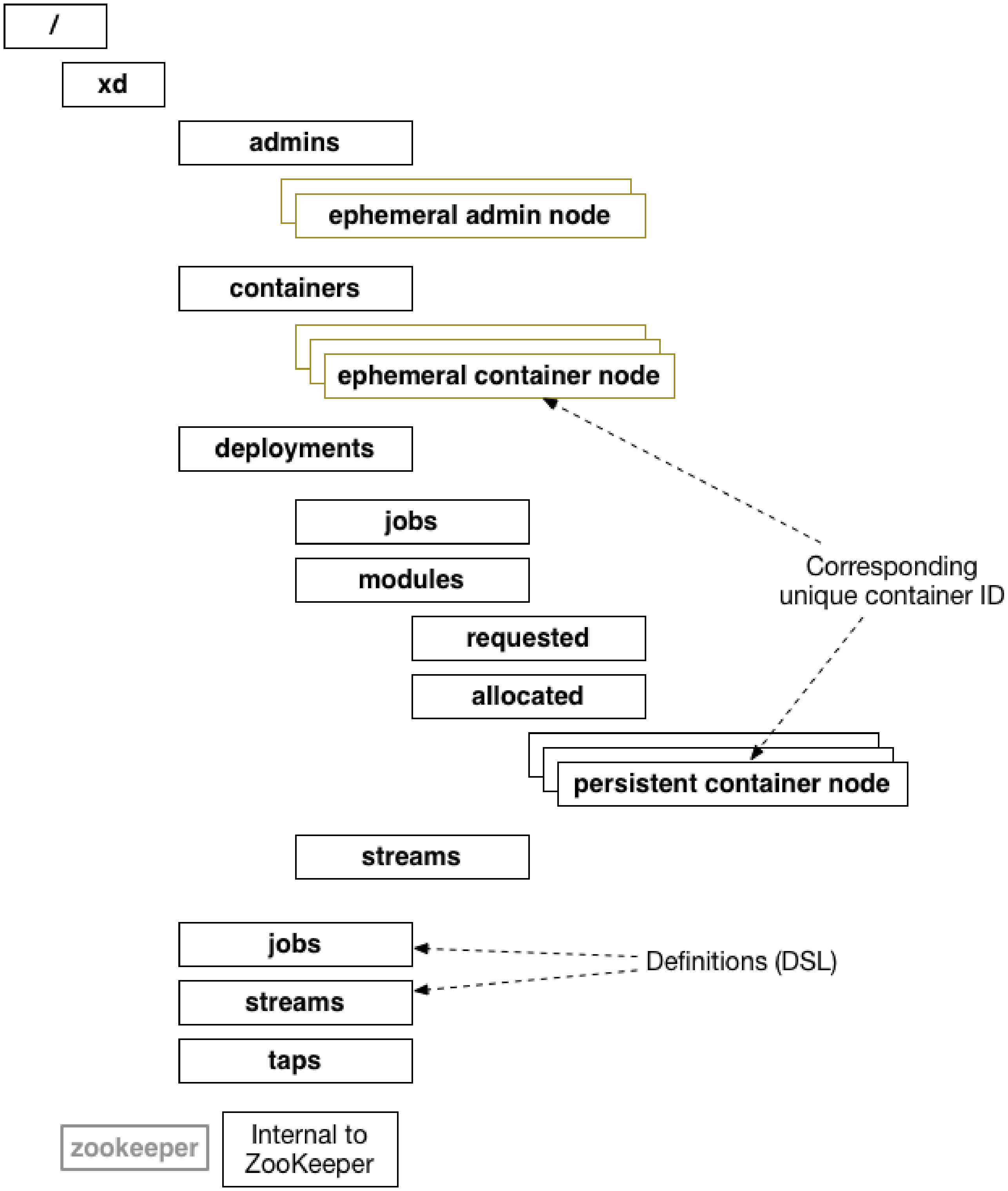


Figure 8.1.

A ZooKeeper node is either *ephemeral* or *persistent*. An ephemeral node exists only as long as the session that created it remains active. A persistent node is, well, persistent. Ephemeral nodes are appropriate for registering Container instances. When an Spring XD container starts up it creates an ephemeral node, */xd/containers/<container-id>*, using an internally generated container id. When the container's session is closed due to a connection loss, for example, the container process terminates, its node is removed. The ephemeral container node also holds metadata such as its hostname and IP address, runtime metrics, and user defined container attributes. Persistent nodes maintain state needed for normal operation and recovery. This includes data such as stream definitions, job definitions, deployment manifests, module deployments, and deployment state for streams and jobs.

Obviously ZooKeeper is a critical piece of the Spring XD runtime and must itself be HA. ZooKeeper itself supports a distributed architecture, called an *ensemble*. The details are beyond the scope of this document but for the sake of this discussion it is worth mentioning that there should be at least three ZooKeeper server instances running (an odd number is always recommended) on dedicated hosts. The Container and Admin nodes are clients to the ZooKeeper ensemble and must connect to ZooKeeper at startup. Spring XD components are configured with a *zk.client.connect* property which may designate a single <host>:<port> or a comma separated list. The ZooKeeper client will attempt to connect to each server in order until it succeeds. If it is unable to connect, it will keep trying. If a connection is lost, the ZooKeeper client will attempt to reconnect to one of the servers. The ZooKeeper cluster guarantees consistent replication of data across the ensemble. Specifically, ZooKeeper guarantees:

- Sequential Consistency - Updates from a client will be applied in the order that they were sent.
- Atomicity - Updates either succeed or fail. No partial results.
- Single System Image - A client will see the same view of the service regardless of the server that it connects to.
- Reliability - Once an update has been applied, it will persist from that time forward until a client overwrites the update.
- Timeliness - The clients view of the system is guaranteed to be up-to-date within a certain time bound.

ZooKeeper maintains data primarily in memory backed by a disk cache. Updates are logged to disk for recoverability, and writes are serialized to disk before they are applied to the in-memory database.

In addition to performing basic CRUD operations on nodes, A ZooKeeper client can register a callback on a node to respond to any events or state changes to that node or any of its children. Such node operations and callbacks are the mechanism that control the Spring XD runtime.

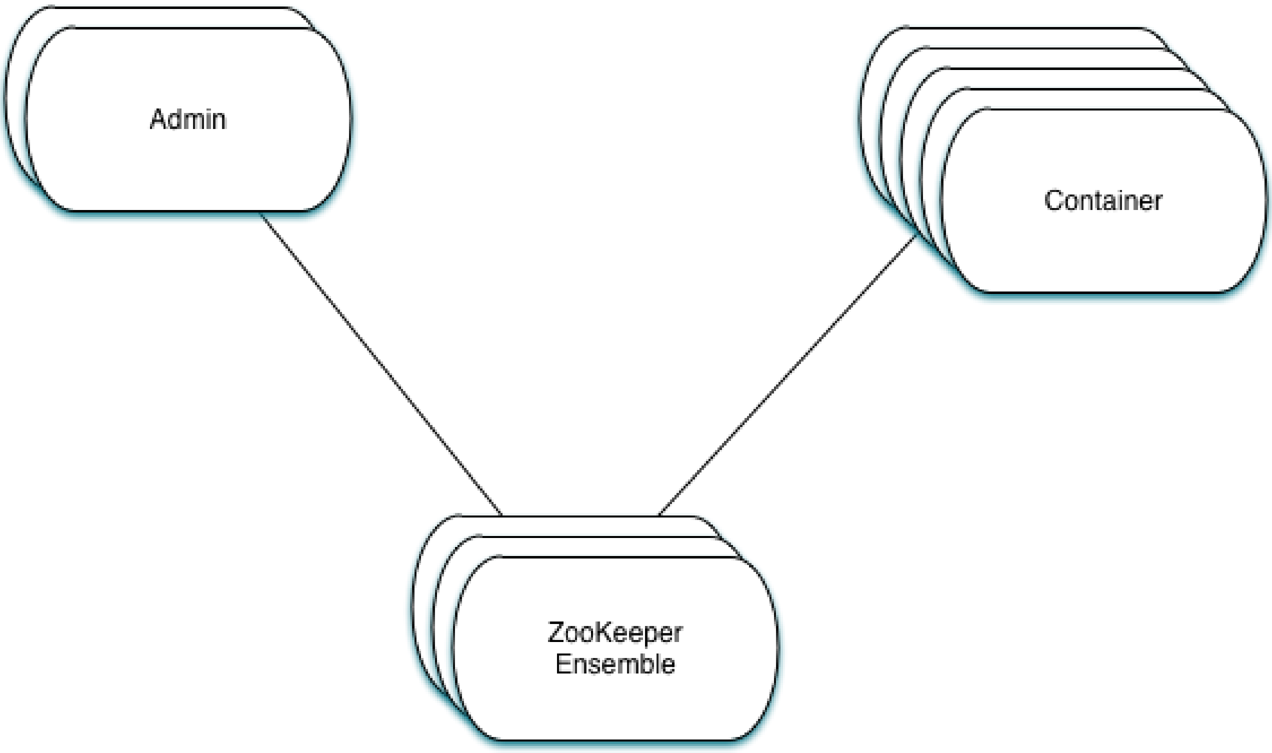


Figure 8.2.

8.4 The Admin Server Internals

Assuming more than one Admin instance is running, Each instance requests leadership at start up. If there is already a designated leader, the instance will watch the *xd/admin* node to be notified if the Leader goes away. The instance designated as the "Leader", using the Leader Selector recipe provided by Curator, a ZooKeeper client library that implements some common patterns. Curator also provides some Listener callback interfaces that the client can register on a node. The AdminServer creates the top level nodes, depicted in the figure above:

- **/xd/admins** - children are ephemeral nodes for each available Admin instance and used for Leader Selector
- **/xd/containers** - children are ephemeral nodes containing runtime attributes including hostname,process id, ip address, and user defined attributes for each container instance.
- **/xd/streams** - children are persistent nodes containing the definition (DSL) for each stream.
- **/xd/jobs** - children are persistent nodes containing the definition (DSL) for each job.
- **/xd/taps** - children are persistent nodes describing each deployed tap.
- **/xd/deployments/streams** - children are nodes containing stream deployment status (leaf nodes are ephemeral).
- **/xd/deployments/jobs** - children are nodes containing job deployment status (leaf nodes are ephemeral).
- **/xd/deployments/modules/requested** - stores module deployment requests including deployment criteria.

- **/xd/deployments/modules/allocated -** stores information describing currently deployed modules.

The admin leader creates a DeploymentSupervisor which registers listeners on */xd/deployments/modules/requested* to handle module deployment requests related to stream and job deployments, and *xd/containers/* to be notified when containers are added and removed from the cluster. Note that any Admin instance can handle user requests. For example, if you enter the following commands via XD shell,

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

This command will invoke a REST service on its connected Admin instance to create a new node /xd/streams/ticktock

```
xd>stream deploy ticktock
```

Assuming the deployment is successful, This will result in the creation of several nodes used to manage deployed resources, for example, */xd/deployments/streams/ticktock*. The details are discussed in the example below.

If the Admin instance connected to the shell is not the Leader, it will perform no further action. The Leader's DeploymentSupervisor will attempt to deploy each module in the stream definition, in accordance with the deployment manifest, to an available container, and update the runtime state.

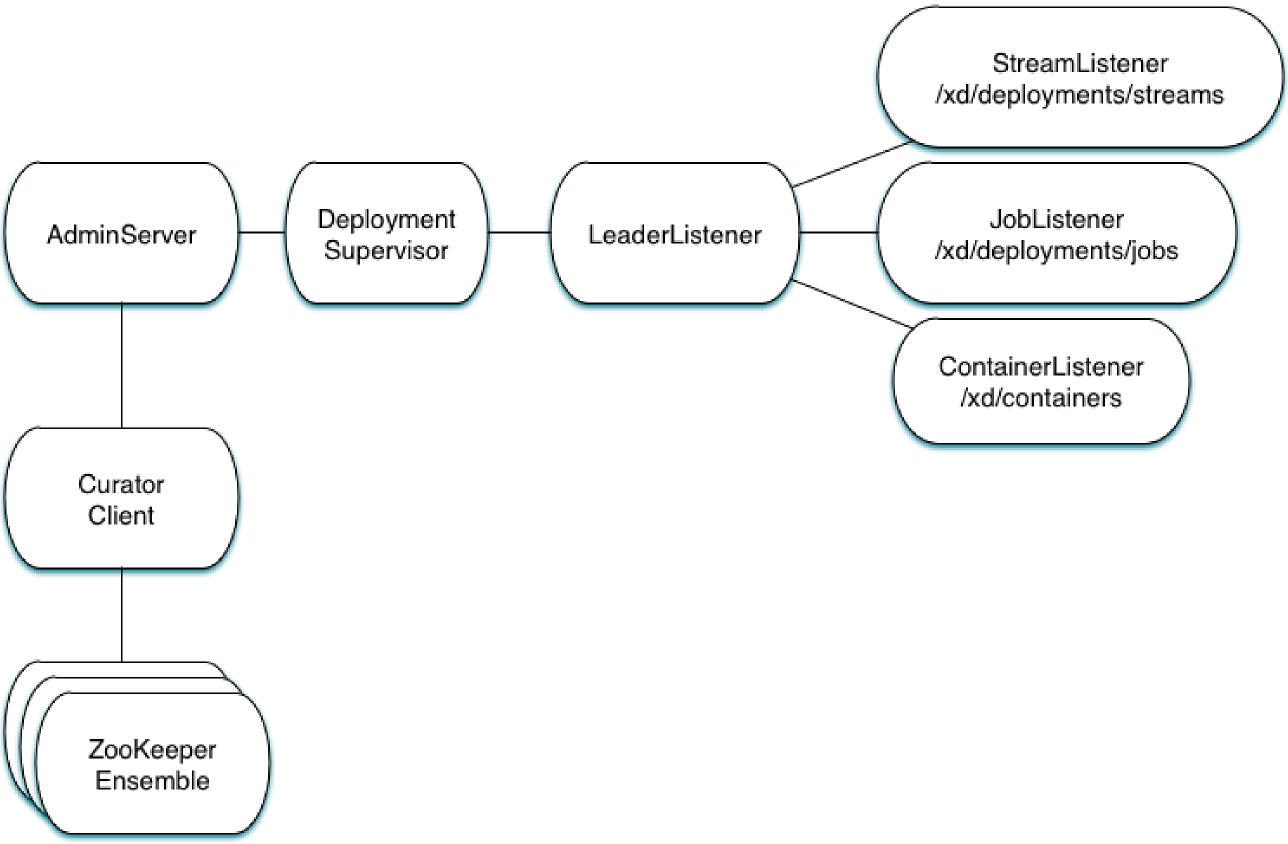


Figure 8.3.

Example

Let’s walk through the simple example above. If you don’t have a Spring XD cluster set up, this example can be easily executed running Spring XD in a single node configuration. The single node application includes an embedded ZooKeeper server by default and allocates a random unused port. The embedded ZooKeeper connect string is reported in the console log for the single node application:

```
...
13:04:27,016  INFO main util.XdConfigLoggingInitializer - Transport: local
13:04:27,016  INFO main util.XdConfigLoggingInitializer - Hadoop Distro: hadoop22
13:04:27,019  INFO main util.XdConfigLoggingInitializer - Hadoop version detected from
 classpath: 2.2.0
13:04:27,019  INFO main util.XdConfigLoggingInitializer - Zookeeper at: localhost:31316
...
```

For our purposes, we will use the ZooKeeper CLI tool to inspect the contents of ZooKeeper nodes reflecting the current state of Spring XD. First, we need to know the port to connect the CLI tool to the embedded server. For convenience, we will assign the ZooKeeper port (5555 in this example) when starting the single node application. From the XD install directory:

```
$export JAVA_OPTS="-Dzk.embedded.server.port=5555"
$xd/bin/xd-singlenode
```

In another terminal session, start the ZooKeeper CLI included with ZooKeeper to connect to the embedded server and inspect the contents of the nodes (NOTE: tab completion works) :

```
$zkCli.sh -server localhost:5555
```

After some console output, you should see a prompt:

```
WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:5555(CONNECTED) 0]
```

navigate using the *ls* command. This will reflect the schema shown in the figure above, the unique container ID will be different for you.

```
[[zk: localhost:5555(CONNECTED) 0] ls /xd
[deployments, containers, admins, taps, streams, jobs]
[zk: localhost:5555(CONNECTED) 1] ls /xd/streams
[]
[zk: localhost:5555(CONNECTED) 2] ls /xd/deployments
[jobs, streams, modules]
[zk: localhost:5555(CONNECTED) 3] ls /xd/deployments/streams
[]
[zk: localhost:5555(CONNECTED) 4] ls /xd/deployments/modules
[requested, allocated]
[zk: localhost:5555(CONNECTED) 5] ls /xd/deployments/modules/allocated
[2ebbbc9b-63ac-4da4-aa32-e39d69eb546b]
[zk: localhost:5555(CONNECTED) 6] ls /xd/deployments/modules/2ebbbc9b-63ac-4da4-aa32-
e39d69eb546b
[]
[zk: localhost:5555(CONNECTED) 7] ls /xd/containers
[2ebbbc9b-63ac-4da4-aa32-e39d69eb546b]
[zk: localhost:5555(CONNECTED) 8]
```

The above reflects the initial state of Spring XD with a running admin and container instance. Nothing is deployed yet and there are no existing stream or job definitions. Note that *xd/deployments/modules/*

allocated has a persistent child corresponding to the id of the container at *xd/containers*. If you are running in a distributed configuration and connected to one of the ZooKeeper servers in the same ensemble that Spring XD is connected to, you might see multiple nodes under */xd/containers*, and *xd/admins*. Because the external ensemble persists the state of the Spring XD cluster, you will also see any deployments that existed when the Spring XD cluster was shut down.

Start the XD Shell in a new terminal session and create a stream:

```
$ shell/bin/xd-shell
 _____                           __   _______
/  ___|          (-)             \ \ / /  _  \
\ `--. _ __  _ __ _ _ __   __ _   \ V /| | | |
 `--. \ '_ \| '__| | '_ \ / _` |  / ^ \| | | |
/\__/ / |_) | |  | | | | | (_| | / / \ \ |/ /
\____/| .__/|_|  |_|_| |_|\__, | \/   \/___/
      | |                  __/ |
      |_|                 |___/
eXtreme Data
1.0 (wiki) | Admin Server Target: http://localhost:9393
Welcome to the Spring XD shell. For assistance hit TAB or type "help".
xd:>stream create ticktock --definition "time | log"
Created new stream 'ticktock'
xd:>
```

Back to the ZK CLI session:

```
[zk: localhost:5555(CONNECTED) 8] ls /xd/streams
[ticktock]
[zk: localhost:5555(CONNECTED) 9] get /xd/streams/ticktock
{"definition":"time | log"}
cZxid = 0x31
ctime = Mon Jul 14 10:32:33 EDT 2014
mZxid = 0x31
mtime = Mon Jul 14 10:32:33 EDT 2014
pZxid = 0x31
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 27
numChildren = 0
[zk: localhost:5555(CONNECTED) 10]
```

using the *get* command on the new stream node, we can see the stream definition represented as JSON, along with some standard ZooKeeper metadata.

Note

ephemeralOwner = 0x0, indicating this is not an ephemeral node. At this point, nothing else should have changed from the initial state.

Now, Using the Spring XD shell, let’s deploy the stream,

```
xd>stream deploy ticktock
Deployed stream 'ticktock'
```

and verify with ZooKeeper:

```
[zk: localhost:5555(CONNECTED) 10] ls /xd/deployments/streams
[ticktock]
[zk: localhost:2181(CONNECTED) 11] ls /xd/streams/deployments/ticktock
[modules, status]
[[zk: localhost:2181(CONNECTED) 12] get /xd/deployments/streams/ticktock/status
{"state":"deployed"}
....
zk: localhost:2181(CONNECTED) 13] ls /xd/deployments/streams/ticktock/modules
[source.time.1.2ebbbc9b-63ac-4da4-aa32-e39d69eb546b, sink.log.1.2ebbbc9b-63ac-4da4-aa32-
e39d69eb546b]
```

Note the deployment state shown for the stream's status node is *deployed*, meaning the deployment request was satisfied. Deployment states are discussed in more detail here.

Spring XD decomposes stream deployment requests to individual module deployment requests. Hence, we see that each module in the stream is associated with a container instance. The container instance in this case is the same since there is only one instance in the single node configuration. In a distributed configuration with more than one instance, the stream source and sink will each be deployed to a separate container. The node name itself is of the form <*module_type*>.<*module_name*>.<*module_sequence_number*>.<*container_id*>, where the sequence number identifies a deployed instance of a module if multiple instances of that module are requested.

```
[zk: localhost:2181(CONNECTED) 14] ls /xd/deployments/modules/
allocated/2ebbbc9b-63ac-4da4-aa32-e39d69eb546b/ticktock.source.time.1
[metadata, status]
```

The *metadata* and *status* nodes are ephemeral nodes which store details about the deployed module. This information is provided to XD shell queries. For example:

```
xd:>runtime modules
  Module                  Container Id                          Options
                       Deployment Properties
  ----------------------  ------------------------------------
 -----------------------------------------------  ---------------------
  ticktock.sink.log.1     2ebbbc9b-63ac-4da4-aa32-e39d69eb546b  {name=ticktock,
 expression=payload, level=INFO}  {count=1, sequence=1}
  ticktock.source.time.1  2ebbbc9b-63ac-4da4-aa32-e39d69eb546b  {fixedDelay=1,
 format=yyyy-MM-dd HH:mm:ss}       {count=1, sequence=1}
```

8.5 Module Deployment

This section describes how the Spring XD runtime manages deployment internally. For more details on how to deploy streams and jobs see Deployment.

To process a stream deployment request, the *StreamDeploymentListener* invokes its *ContainerMatcher* to select a container instance for each module and records the module's deployment properties under */xd/deployments/modules/requested/*. If a match is found, the StreamDeploymentListener creates a node for the module under */xd/deployments/modules/allocated/<container_id>*. The Container includes a *DeploymentListener* that monitors the container node for new modules to deploy. If the deployment is successful, the Container writes the ephemeral nodes *status* and *metadata* under the new module node.

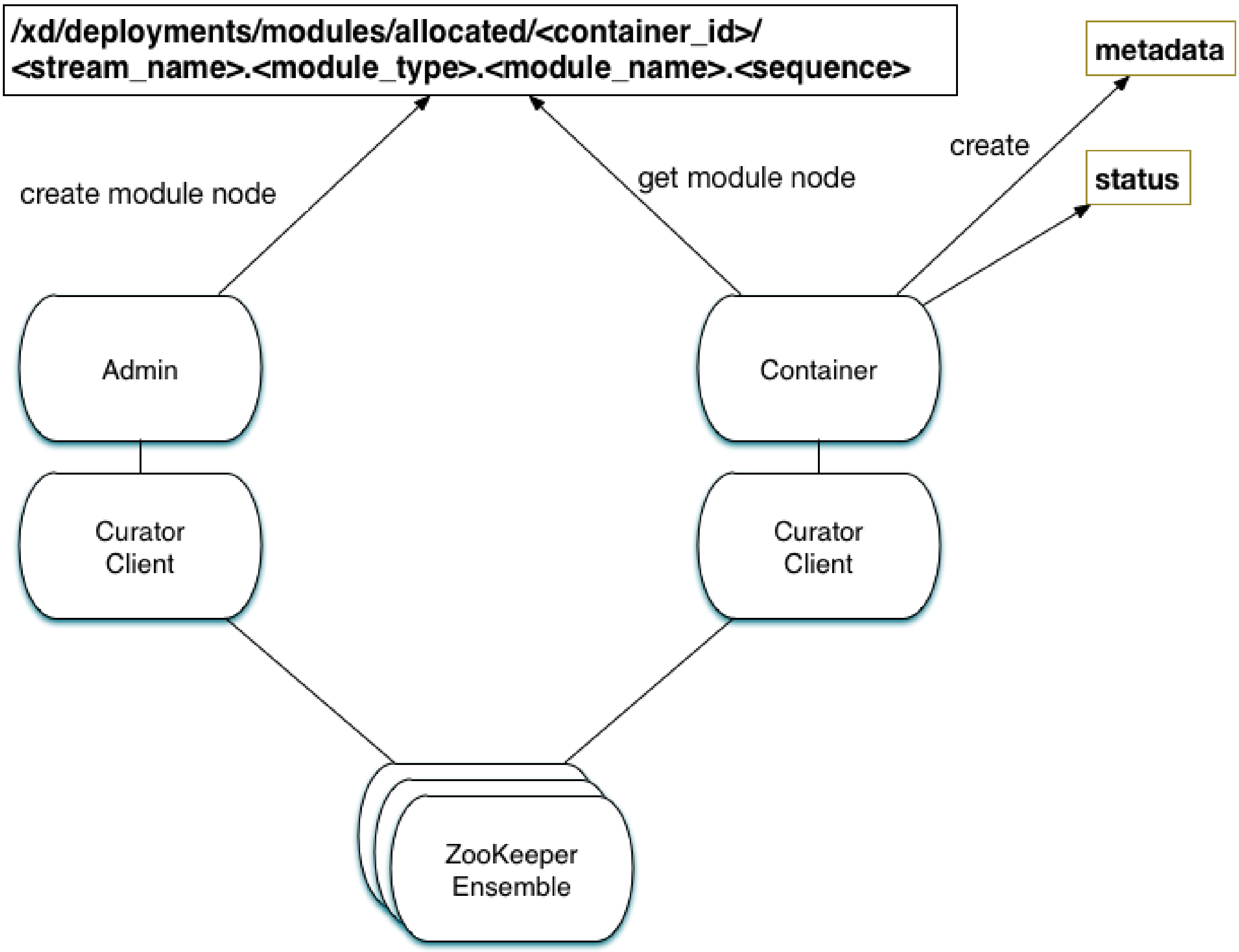


Figure 8.4.

When a container departs, the ephemeral nodes are deleted so its modules are now undeployed. The *ContainerListener* responds to the deleted nodes and attempts to redeploy any affected modules to another instance.

Example: Automatic Redeployment

For this example we start two container instances and deploy and simple stream:

```
xd:>runtime containers
  Container Id                          Host            IP Address   PID    Groups  Custom
 Attributes
  ------------------------------------  --------------  -----------  -----  ------
 -----------------
  0ddf80b9-1e80-44b8-8c12-ecc5c8c32e11  ultrafox.local  192.168.1.6  19222
  6cac85f8-4c52-4861-a225-cdad3675f6c9  ultrafox.local  192.168.1.6  19244

xd:>stream create ticktock --definition "time | log"
Created new stream 'ticktock'
xd:>stream deploy ticktock
Deployed stream 'ticktock'
xd:>runtime modules
  Module                  Container Id                          Options
                       Deployment Properties
  ----------------------  ------------------------------------
 -----------------------------------------------  ---------------------
  ticktock.sink.log.1     0ddf80b9-1e80-44b8-8c12-ecc5c8c32e11  {name=ticktock,
 expression=payload, level=INFO}  {count=1, sequence=1}
  ticktock.source.time.1  6cac85f8-4c52-4861-a225-cdad3675f6c9  {fixedDelay=1,
 format=yyyy-MM-dd HH:mm:ss}       {count=1, sequence=1}
```

Now we will kill one of the container processes and observe that the affect module has been redeployed to the remaining container:

```
xd:>runtime containers
  Container Id                          Host            IP Address   PID    Groups  Custom
 Attributes
  ------------------------------------  --------------  -----------  -----  ------
 -----------------
  6cac85f8-4c52-4861-a225-cdad3675f6c9  ultrafox.local  192.168.1.6  19244

xd:>runtime modules
  Module                  Container Id                          Options
                       Deployment Properties
  ----------------------  ------------------------------------
 -----------------------------------------------  ---------------------
  ticktock.sink.log.1     6cac85f8-4c52-4861-a225-cdad3675f6c9  {name=ticktock,
 expression=payload, level=INFO}  {count=1, sequence=1}
  ticktock.source.time.1  6cac85f8-4c52-4861-a225-cdad3675f6c9  {fixedDelay=1,
 format=yyyy-MM-dd HH:mm:ss}       {count=1, sequence=1}
```

Now if we kill the remaining container, we see warnings in the xd-admin log:

```
14:36:07,593  WARN DeploymentSupervisorCacheListener-0
 server.DepartingContainerModuleRedeployer - No containers available for redeployment of
 log for stream ticktock
14:36:07,599  WARN DeploymentSupervisorCacheListener-0
 server.DepartingContainerModuleRedeployer - No containers available for redeployment of
 time for stream ticktock
```

9. Batch Jobs

9.1 Introduction

One of the features that XD offers is the ability to launch and monitor batch jobs based on Spring Batch. The Spring Batch project was started in 2007 as a collaboration between SpringSource and Accenture to provide a comprehensive framework to support the development of robust batch applications. Batch jobs have their own set of best practices and domain concepts which have been incorporated into Spring Batch building upon Accenture's consulting business. Since then Spring Batch has been used in thousands of enterprise applications and is the basis for the recent JSR standardization of batch processing, JSR-352.

Spring XD builds upon Spring Batch to simplify creating batch workflow solutions that span traditional use-cases such as moving data between flat files and relational databases as well as Hadoop use-cases where analysis logic is broken up into several steps that run on a Hadoop cluster. Steps specific to Hadoop in a workflow can be MapReduce jobs, executing Hive/Pig scripts or HDFS operations.

9.2 Workflow

The concept of a workflow translates to a Job, not to be confused with a MapReduce job. A Job is a directed graph, each node of the graph is a processing Step. Steps can be executed sequentially or in parallel, depending on the configuration. Jobs can be started, stopped, and restarted. Restarting jobs is possible since the progress of executed steps in a Job is persisted in a database via a JobRepository. The following figures shows the basic components of a workflow.

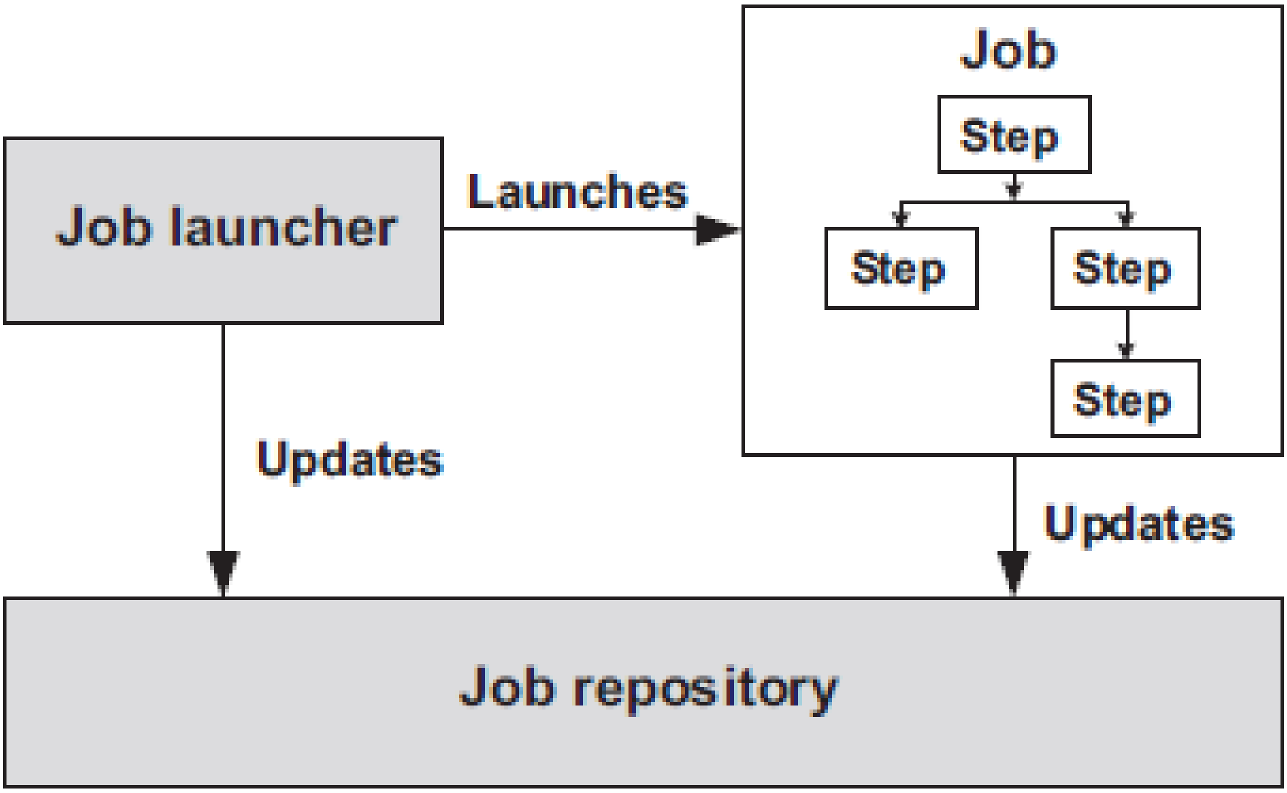


Figure 9.1.

A Job that has steps specific to Hadoop is shown below.

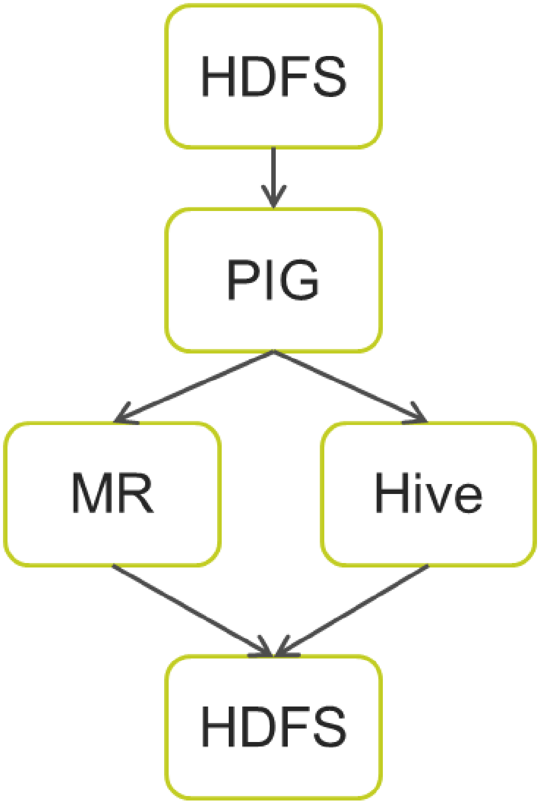


Figure 9.2.

A JobLauncher is responsible for starting a job and is often triggered via a scheduler. Other options to launch a job are through Spring XD's RESTful administration API, the XD web application, or in response to an external event from and XD stream definition, *e.g.* file polling using the file source.

9.3 Features

Spring XD allows you to create and launch jobs. The launching of a job can be triggered using a cron expression or in reaction to data on a stream. When jobs are executing, they are also a souce of event data that can be subscribed to by a stream. There are several type of events sent during a job's execution, the most common being the status of the job and the steps taken within the job. This bi-direction communication between stream processing and batch processing allows for more complex chains of processing to be developed.

As a starting point, jobs for the following cases are provided to use out of the box

- Poll a Directory and import CSV files to HDFS
- Import CSV files to JDBC
- HDFS to JDBC Export
- JDBC to HDFS Import
- HDFS to MongoDB Export

These are described in the section below.

This purpose of this section is to show you how to create, schedule and monitor a job.

9.4 The Lifecycle of a Job in Spring XD

Before we dive deeper into the details of creating batch jobs with Spring XD, we need to understand the typical lifecycle for batch jobs in the context of *Spring XD*:

1. Register a Job Module
2. Create a Job Definition
3. Deploy a Job
4. Launch a Job
5. Job Execution
6. Un-deploy a Job
7. Destroy a Job Definition

Register a Job Module

Register a **Job Module** with the *Module Registry* by putting XML and/or jar files into the `$XD_HOME/modules/jobs` directory.

Create a Job Definition

Create a **Job Definition** from a *Job Module* by providing a definition name as well as properties that apply to all *Job Instances*. At this point the job is not deployed, yet.

Deploy the Job

Deploy the **Job Definition** to one or more *Spring XD* containers. This will initialize the *Job Definitions* on those containers. The jobs are now "live" and a job can be created by sending a message to a job queue that contains optional runtime **Job Parameters**.

Launch a Job

Launch a job by sending a message to the job queue with **Job Parameters**. A **Job Instance** is created, representing a specific run of the job. A **Job Instance** is the **Job Definition** plus the runtime **Job Parameters**. You can query for the **Job Instances** associated with a given job name.

Job Execution

The job is executed creating a **Job Execution** object that captures the success or failure of the job. You can query for **Job Executions** associated with a given job name.

Un-deploy a Job

This removes the job from the *Spring XD* container(s) preventing the launching of any new *Job Instances*. For reporting purposes, you will still be able to view historic *Job Executions* associated with the the job.

Destroy a Job Definition

Destroying a **Job Definition** will not only un-deploy any still deployed *Job Definitions* but will also remove the *Job Definition* itself.

Creating Jobs - Additional Options

When creating jobs, the following options are available to all job definitions:

dateFormat
: The optional date format for job parameters **(default: `yyyy-MM-dd`)**

numberFormat
: Defines the number format when parsing numeric parameters **(default: `NumberFormat.getInstance(Locale.US)`)**

makeUnique
: Shall job parameters be made unique? **(default: `true`)**

Also, similar to the `stream create` command, the `job create` command has an optional `--deploy` option to create the job definition and deploy it. `--deploy` option is false by default.

Below is an example of some of these options combined:

```
job create myjob --definition "fooJob --makeUnique=false"
```

Remember that you can always find out about available options for a job by using the `module info` command.

9.5 Deployment manifest support for job

When deploying batch job you can provide a deployment manifest. Deployment manifest properties for jobs are the same as for streams, you can declare

- The number of job modules to deploy
- The criteria expression to use for matching the job to available containers

For example,

```
job create myjob --definition "fooJob --makeUnique=false"

job deploy myjob --properties
 "module.fooJob.count=3,module.fooJob.criteria=groups.contains('hdfs-containers-group')"
```

The above deployment manifest would deploy 3 number of `fooJob` modules into containers whose group name matches "hdfs-containers-group".

When a batch job is launched/scheduled, the job module that picks up the job launching request message executes the batch job. To support partitioning of the job across multiple containers, the job definition needs to define how the job will be partitioned. The type of partitioning depends on the type of the job, for example a job reading from JDBC would partition the data in a table by dividing up the number of rows and a job reading files form a directory would partition on the number of files available.

The FTP to HDFS and FILE to JDBC jobs support for partitioning. To add partitioning support for your own jobs you should import singlestep-partition-support.xml in your job definition. This provides the infrastructure so that the job module that processes the launch request can communicate as the master with the other job modules that have been deployed. You will also need to provide an implementation of the Partitioner interface.

For more information on the deployment manifest, please refer here

9.6 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.job.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 trigger source.

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

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 "trigger --cron='0/5 * * * * *'  --
payload={\"param1\":\"Kenny\"} > queue:job:myCronJob" --deploy
```

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 "trigger --payload={\"param1\":
\"fixedDelayKenny\"} --fixedDelay=5 > queue:job:myXDJob" --deploy
```

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 an 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" --deploy
```

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

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

9.7 Retrieve job notifications

Spring XD offers the facilities to capture the notifications that are sent from the job as it is executing. When a batch job is deployed, by default it registers the following listeners along with pub/sub channels that these listeners send messages to.

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

Along with the pub/sub channels for each of these listeners, there will also be a pub/sub channel that the aggregated events from all these listeners are published to.

In the following example, we setup a Batch Job called *myHttpJob*. Afterwards we create a stream that will tap into the pub/sub channels that were implicitly generated when the *myHttpJob* job was deployed.

To receive aggregated events

The stream receives aggregated event messages from all the default batch job listeners and sends those messages to the log.

```
xd>job create --name myHttpJob --definition "httpJob" --deploy
xd>stream create --name aggregatedEvents --definition "tap:job:myHttpJob >log" --deploy
xd>job launch myHttpJob
```

Note: The syntax for the tap that receives the aggregated events is: `tap:job:<job-name>`

In the logging output of the container you should see something like the following when the job completes (with the aggregated events

```
09:55:53,532  WARN SimpleAsyncTaskExecutor-1 logger.aggregatedEvents:150
 - JobExecution: id=2, version=1, startTime=Sat Apr 12 09:55:53 PDT 2014,
 endTime=null, lastUpdated=Sat Apr 12 09:55:53 PDT 2014, status=STARTED,
 exitStatus=exitCode=UNKNOWN;exitDescription=, job=[JobInstance: id=2, version=0,
 Job=[myHttpJob]], jobParameters=[{random=0.07002785662707867}]
09:55:53,554  WARN SimpleAsyncTaskExecutor-1 logger.aggregatedEvents:150 - StepExecution:
 id=2, version=1, name=step1, status=STARTED, exitStatus=EXECUTING, readCount=0,
 filterCount=0, writeCount=0 readSkipCount=0, writeSkipCount=0, processSkipCount=0,
 commitCount=0, rollbackCount=0, exitDescription=
09:55:53,561  WARN SimpleAsyncTaskExecutor-1 logger.aggregatedEvents:150 -
 XdChunkContextInfo [complete=false, stepExecution=StepExecution: id=2, version=1,
 name=step1, status=STARTED, exitStatus=EXECUTING, readCount=0, filterCount=0,
 writeCount=0 readSkipCount=0, writeSkipCount=0, processSkipCount=0, commitCount=0,
 rollbackCount=0, exitDescription=, attributes={}]
09:55:53,567  WARN SimpleAsyncTaskExecutor-1 logger.aggregatedEvents:150 -
 XdChunkContextInfo [complete=false, stepExecution=StepExecution: id=2, version=2,
 name=step1, status=STARTED, exitStatus=EXECUTING, readCount=0, filterCount=0,
 writeCount=0 readSkipCount=0, writeSkipCount=0, processSkipCount=0, commitCount=1,
 rollbackCount=0, exitDescription=, attributes={}]
09:55:53,573  WARN SimpleAsyncTaskExecutor-1 logger.aggregatedEvents:150 - StepExecution:
 id=2, version=2, name=step1, status=COMPLETED, exitStatus=COMPLETED, readCount=0,
 filterCount=0, writeCount=0 readSkipCount=0, writeSkipCount=0, processSkipCount=0,
 commitCount=1, rollbackCount=0, exitDescription=
09:55:53,580  WARN SimpleAsyncTaskExecutor-1 logger.aggregatedEvents:150 - JobExecution:
 id=2, version=1, startTime=Sat Apr 12 09:55:53 PDT 2014, endTime=Sat Apr 12
 09:55:53 PDT 2014, lastUpdated=Sat Apr 12 09:55:53 PDT 2014, status=COMPLETED,
 exitStatus=exitCode=COMPLETED;exitDescription=, job=[JobInstance: id=2, version=0,
 Job=[myHttpJob]], jobParameters=[{random=0.07002785662707867}]
```

To receive job execution events

```
xd>job create --name myHttpJob --definition "httpJob" --deploy
xd>stream create --name jobExecutionEvents --definition "tap:job:myHttpJob.job >log" --
deploy
xd>job launch myHttpJob
```

Note: The syntax for the tap that receives the job execution events is: `tap:job:<job-name>.job`

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

```
10:06:41,579  WARN SimpleAsyncTaskExecutor-1 logger.jobExecutionEvents:150
 - JobExecution: id=3, version=1, startTime=Sat Apr 12 10:06:41 PDT 2014,
 endTime=null, lastUpdated=Sat Apr 12 10:06:41 PDT 2014, status=STARTED,
 exitStatus=exitCode=UNKNOWN;exitDescription=, job=[JobInstance: id=3, version=0,
 Job=[myHttpJob]], jobParameters=[{random=0.3774227747555795}]
10:06:41,626  INFO SimpleAsyncTaskExecutor-1 support.SimpleJobLauncher:136
 - Job: [FlowJob: [name=myHttpJob]] completed with the following parameters:
 [{random=0.3774227747555795}] and the following status: [COMPLETED]
10:06:41,626  WARN SimpleAsyncTaskExecutor-1 logger.jobExecutionEvents:150 -
 JobExecution: id=3, version=1, startTime=Sat Apr 12 10:06:41 PDT 2014, endTime=Sat
 Apr 12 10:06:41 PDT 2014, lastUpdated=Sat Apr 12 10:06:41 PDT 2014, status=COMPLETED,
 exitStatus=exitCode=COMPLETED;exitDescription=, job=[JobInstance: id=3, version=0,
 Job=[myHttpJob]], jobParameters=[{random=0.3774227747555795}]
```

To receive step execution events

```
xd>job create --name myHttpJob --definition "httpJob" --deploy
xd>stream create --name stepExecutionEvents --definition "tap:job:myHttpJob.step >log" --
deploy
xd>job launch myHttpJob
```

Note: The syntax for the tap that receives the step execution events is: `tap:job:<job-name>.step`

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

```

10:13:16,072  WARN SimpleAsyncTaskExecutor-1 logger.stepExecutionEvents:150 -
 StepExecution: id=6, version=1, name=step1, status=STARTED, exitStatus=EXECUTING,
 readCount=0, filterCount=0, writeCount=0 readSkipCount=0, writeSkipCount=0,
 processSkipCount=0, commitCount=0, rollbackCount=0, exitDescription=
10:13:16,092  WARN SimpleAsyncTaskExecutor-1 logger.stepExecutionEvents:150 -
 StepExecution: id=6, version=2, name=step1, status=COMPLETED, exitStatus=COMPLETED,
 readCount=0, filterCount=0, writeCount=0 readSkipCount=0, writeSkipCount=0,
 processSkipCount=0, commitCount=1, rollbackCount=0, exitDescription=
```

To receive item, skip and chunk events

```
xd>job create --name myHttpJob --definition "httpJob" --deploy

xd>stream create --name itemEvents --definition "tap:job:myHttpJob.item >log" --deploy
xd>stream create --name skipEvents --definition "tap:job:myHttpJob.skip >log" --deploy
xd>stream create --name chunkEvents --definition "tap:job:myHttpJob.chunk >log" --deploy

xd>job launch myHttpJob
```

Note: The syntax for the tap that receives the item events: `tap:job:<job-name>.item`,for skip events: `tap:job:<job-name>.skip` and for chunk events: `tap:job:<job-name>.chunk`

To disable the default listeners

```
xd>job create --name myHttpJob --definition "httpJob --listeners=disable" --deploy
```

To select specific listeners

To select specific listeners, specify comma separated list in `--listeners` option. Following example illustrates the selection of job and step execution listeners only:

```
xd>job create --name myHttpJob --definition "httpJob --listeners=job,step" --deploy
```

Note: List of options are: job, step, item, chunk and skip The aggregated channel is registered if at least one of these default listeners are enabled.

For a complete example, please see the Batch Notifications Sample which is part of the Spring XD Samples repository.

9.8 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
```

9.9 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.

Note HDFS Configuration

To use the hdfs based jobs below, XD needs to have append enabled for hdfs. Update the hdfs-site.xml with the following settings:

For Hadoop 1.x

```
    <property>
      <name>dfs.support.broken.append</name>
      <value>true</value>
    </property>
```

For Hadoop 2.x

```
    <property>
        <name>dfs.support.append</name>
        <value>true</value>
    </property>
```

Poll a Directory and Import CSV Files to HDFS (`filepollhdfs`)

This module is designed to be driven by a stream polling a directory. 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 "filepollhdfs --names=forename,surname,address" --
deploy
```

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" --deploy
```

The **filepollhdfs** job has the following options:

deleteFiles
: whether to delete files after successful import **(boolean, default: `false`)**

directory
: the directory to write the file(s) to in HDFS **(String, default: `/xd/<job name>`)**

fileExtension
: the file extension to use **(String, default: `csv`)**

fileName
: the filename to use in HDFS **(String, default: `<job name>`)**

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

names
: the field names in the CSV file **(String, no default)**

restartable
: whether the job should be restartable or not in case of failure **(boolean, default: `false`)**

rollover
: the number of bytes to write before creating a new file in HDFS **(int, default: `1000000`)**

Import CSV Files to JDBC (`filejdbc`)

A module which loads CSV files into a JDBC table using a single batch job. By default it uses the internal HSQL DB which is used by Spring Batch. Refer to how module options are resolved for further details on how to change defaults (one can of course always use `--foo=bar` notation in the job definition to achieve the same effect).

The **filejdbc** job has the following options:

abandonWhenPercentageFull
: connections that have timed out wont get closed and reported up unless the number of connections in use are above the percentage **(int, default: `0`)**

alternateUsernameAllowed
: uses an alternate user name if connection fails **(boolean, default: `false`)**

connectionProperties
: connection properties that will be sent to our JDBC driver when establishing new connections **(String, no default)**

deleteFiles
: whether to delete files after successful import **(boolean, default: `false`)**

delimiter
: the delimiter for the delimited file **(String, default: `,`)**

driverClassName
: the JDBC driver to use **(String, no default)**

fairQueue
: set to true if you wish that calls to getConnection should be treated fairly in a true FIFO fashion **(boolean, default: `true`)**

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

initSQL
: custom query to be run when a connection is first created **(String, no default)**

initialSize
: initial number of connections that are created when the pool is started **(int, default: `0`)**

initializeDatabase
: whether the database initialization script should be run **(boolean, default: `false`)**

initializerScript
: the name of the SQL script (in /config) to run if *initializeDatabase* is set **(String, default: `init_batch_import.sql`)**

jdbcInterceptors
: semicolon separated list of classnames extending org.apache.tomcat.jdbc.pool.JdbcInterceptor **(String, no default)**

jmxEnabled
: register the pool with JMX or not **(boolean, default: `true`)**

logAbandoned
: flag to log stack traces for application code which abandoned a Connection **(boolean, default: `false`)**

maxActive
: maximum number of active connections that can be allocated from this pool at the same time **(int, default: `100`)**

maxAge
: time in milliseconds to keep this connection **(int, default: `0`)**

maxIdle
: maximum number of connections that should be kept in the pool at all times **(int, default: `100`)**

maxWait
: maximum number of milliseconds that the pool will wait for a connection **(int, default: `30000`)**

minEvictableIdleTimeMillis
: minimum amount of time an object may sit idle in the pool before it is eligible for eviction **(int, default: `60000`)**

minIdle
: minimum number of established connections that should be kept in the pool at all times **(int, default: `10`)**

names
: the field names in the CSV file **(String, no default)**

password
: the JDBC password **(Password, no default)**

removeAbandoned
: flag to remove abandoned connections if they exceed the removeAbandonedTimout **(boolean, default: `false`)**

removeAbandonedTimeout
: timeout in seconds before an abandoned connection can be removed **(int, default: `60`)**

resources
: the list of paths to import (Spring resources) **(String, no default)**

restartable
: whether the job should be restartable or not in case of failure **(boolean, default: `false`)**

suspectTimeout
: this simply logs the warning after timeout, connection remains **(int, default: `0`)**

tableName
: the database table to which the data will be written **(String, default: `<job name>`)**

testOnBorrow
: indication of whether objects will be validated before being borrowed from the pool **(boolean, default: `false`)**

testOnReturn
: indication of whether objects will be validated before being returned to the pool **(boolean, default: `false`)**

testWhileIdle
: indication of whether objects will be validated by the idle object evictor **(boolean, default: `false`)**

timeBetweenEvictionRunsMillis
: number of milliseconds to sleep between runs of the idle connection validation/cleaner thread **(int, default: `5000`)**

url
: the JDBC URL for the database **(String, no default)**

useEquals
: true if you wish the ProxyConnection class to use String.equals **(boolean, default: `true`)**

username
: the JDBC username **(String, no default)**

validationInterval
: avoid excess validation, only run validation at most at this frequency - time in milliseconds **(long, default: `30000`)**

validationQuery
: sql query that will be used to validate connections from this pool **(String, no default)**

validatorClassName
: name of a class which implements the org.apache.tomcat.jdbc.pool.Validator **(String, no default)**

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 create it for you if you use `--initializeDatabase=true` when the job is created. The table initialization 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 customized by setting the `tableName` parameter. As an example, if you run the command

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

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. As with the `filepollhdfs` job, this module also supports the `deleteFiles` parameter which will remove the files defined by the `resources` parameter on successful completion of the job.

Launch the job using:

```
xd:> job launch myjob
```

Tip

The connection pool settings for xd are located in servers.yml (i.e. `spring.datasource.*`)

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.

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

Launch the job using:

```
xd:> job launch myjob
```

The **hdfsjdbc** job has the following options:

abandonWhenPercentageFull
: connections that have timed out wont get closed and reported up unless the number of connections in use are above the percentage **(int, default: `0`)**

alternateUsernameAllowed
: uses an alternate user name if connection fails **(boolean, default: `false`)**

connectionProperties
: connection properties that will be sent to our JDBC driver when establishing new connections **(String, no default)**

delimiter
: the delimiter for the delimited file **(String, default: `,`)**

driverClassName
: the JDBC driver to use **(String, no default)**

fairQueue
: set to true if you wish that calls to getConnection should be treated fairly in a true FIFO fashion **(boolean, default: `true`)**

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

initSQL
: custom query to be run when a connection is first created **(String, no default)**

initialSize
: initial number of connections that are created when the pool is started **(int, default: `0`)**

initializeDatabase
: whether the database initialization script should be run **(boolean, default: `false`)**

initializerScript
: the name of the SQL script (in /config) to run if *initializeDatabase* is set **(String, default: `init_batch_import.sql`)**

jdbcInterceptors
: semicolon separated list of classnames extending org.apache.tomcat.jdbc.pool.JdbcInterceptor **(String, no default)**

jmxEnabled
: register the pool with JMX or not **(boolean, default: `true`)**

logAbandoned
: flag to log stack traces for application code which abandoned a Connection **(boolean, default: `false`)**

maxActive
: maximum number of active connections that can be allocated from this pool at the same time **(int, default: `100`)**

maxAge
: time in milliseconds to keep this connection **(int, default: `0`)**

maxIdle
: maximum number of connections that should be kept in the pool at all times **(int, default: `100`)**

maxWait
: maximum number of milliseconds that the pool will wait for a connection **(int, default: `30000`)**

minEvictableIdleTimeMillis
: minimum amount of time an object may sit idle in the pool before it is eligible for eviction **(int, default: `60000`)**

minIdle
: minimum number of established connections that should be kept in the pool at all times **(int, default: `10`)**

names
: the field names in the CSV file **(String, no default)**

password
: the JDBC password **(Password, no default)**

removeAbandoned
: flag to remove abandoned connections if they exceed the removeAbandonedTimout **(boolean, default: `false`)**

removeAbandonedTimeout
: timeout in seconds before an abandoned connection can be removed **(int, default: `60`)**

resources
: the list of paths to import (Spring resources) **(String, no default)**

restartable
: whether the job should be restartable or not in case of failure **(boolean, default: `false`)**

suspectTimeout
: this simply logs the warning after timeout, connection remains **(int, default: `0`)**

tableName
: the database table to which the data will be written **(String, default: `<job name>`)**

testOnBorrow
: indication of whether objects will be validated before being borrowed from the pool **(boolean, default: `false`)**

testOnReturn
: indication of whether objects will be validated before being returned to the pool **(boolean, default: `false`)**

testWhileIdle
: indication of whether objects will be validated by the idle object evictor **(boolean, default: `false`)**

timeBetweenEvictionRunsMillis
: number of milliseconds to sleep between runs of the idle connection validation/cleaner thread **(int, default: `5000`)**

url
: the JDBC URL for the database **(String, no default)**

useEquals
: true if you wish the ProxyConnection class to use String.equals **(boolean, default: `true`)**

username
: the JDBC username **(String, no default)**

validationInterval
: avoid excess validation, only run validation at most at this frequency - time in milliseconds **(long, default: `30000`)**

validationQuery
: sql query that will be used to validate connections from this pool **(String, no default)**

validatorClassName
: name of a class which implements the org.apache.tomcat.jdbc.pool.Validator **(String, no default)**

Tip

The connection pool settings for xd are located in servers.yml (i.e. `spring.datasource.*`)

JDBC to HDFS Import (`jdbchdfs`)

Performs the reverse of the previous module. The database configuration is the same as for `filejdbc` but without the initialization options since you need to already have the data to import into HDFS. When creating the job, you must either supply the select statement by setting the `sql` parameter, or you can supply both `tableName` and `columns` options (which will be used to build the SQL statement).

To import data from the database table `some_table`, you could use

```
xd:> job create myjob --definition "jdbchdfs --sql='select col1,col2,col3 from
 some_table'" --deploy
```

You can customize how the data is written to HDFS by supplying the options `directory` (defaults to `/xd/(job name)`), `fileName` (defaults to job name), `rollover` (in bytes, default 1000000) and `fileExtension` (defaults to *csv*).

Launch the job using:

```
xd:> job launch myjob
```

The **jdbchdfs** job has the following options:

abandonWhenPercentageFull

  connections that have timed out wont get closed and reported up unless the number of connections in use are above the percentage **(int, default: `0`)**

alternateUsernameAllowed

  uses an alternate user name if connection fails **(boolean, default: `false`)**

columns

  the column names to read from the supplied table **(String, default: ``)**

connectionProperties

  connection properties that will be sent to our JDBC driver when establishing new connections **(String, no default)**

delimiter

  the delimiter for the delimited file **(String, default: `,`)**

directory

  the directory to write the file(s) to in HDFS **(String, default: `/xd/<job name>`)**

driverClassName

  the JDBC driver to use **(String, no default)**

fairQueue

  set to true if you wish that calls to getConnection should be treated fairly in a true FIFO fashion **(boolean, default: `true`)**

fileExtension

  the file extension to use **(String, default: `csv`)**

fileName
: the filename to use in HDFS **(String, default: `<job name>`)**

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

initSQL
: custom query to be run when a connection is first created **(String, no default)**

initialSize
: initial number of connections that are created when the pool is started **(int, default: `0`)**

jdbcInterceptors
: semicolon separated list of classnames extending org.apache.tomcat.jdbc.pool.JdbcInterceptor **(String, no default)**

jmxEnabled
: register the pool with JMX or not **(boolean, default: `true`)**

logAbandoned
: flag to log stack traces for application code which abandoned a Connection **(boolean, default: `false`)**

maxActive
: maximum number of active connections that can be allocated from this pool at the same time **(int, default: `100`)**

maxAge
: time in milliseconds to keep this connection **(int, default: `0`)**

maxIdle
: maximum number of connections that should be kept in the pool at all times **(int, default: `100`)**

maxWait
: maximum number of milliseconds that the pool will wait for a connection **(int, default: `30000`)**

minEvictableIdleTimeMillis
: minimum amount of time an object may sit idle in the pool before it is eligible for eviction **(int, default: `60000`)**

minIdle
: minimum number of established connections that should be kept in the pool at all times **(int, default: `10`)**

password
: the JDBC password **(Password, no default)**

removeAbandoned
: flag to remove abandoned connections if they exceed the removeAbandonedTimout **(boolean, default: `false`)**

removeAbandonedTimeout
: timeout in seconds before an abandoned connection can be removed **(int, default: `60`)**

restartable
: whether the job should be restartable or not in case of failure **(boolean, default: `false`)**

rollover
: the number of bytes to write before creating a new file in HDFS **(int, default: `1000000`)**

sql
: the SQL to use to extract data **(String, default: ``)**

suspectTimeout
: this simply logs the warning after timeout, connection remains **(int, default: `0`)**

tableName
: the table to read data from **(String, default: ``)**

testOnBorrow
: indication of whether objects will be validated before being borrowed from the pool **(boolean, default: `false`)**

testOnReturn
: indication of whether objects will be validated before being returned to the pool **(boolean, default: `false`)**

testWhileIdle
: indication of whether objects will be validated by the idle object evictor **(boolean, default: `false`)**

timeBetweenEvictionRunsMillis
: number of milliseconds to sleep between runs of the idle connection validation/cleaner thread **(int, default: `5000`)**

url
: the JDBC URL for the database **(String, no default)**

useEquals
: true if you wish the ProxyConnection class to use String.equals **(boolean, default: `true`)**

username
: the JDBC username **(String, no default)**

validationInterval
: avoid excess validation, only run validation at most at this frequency - time in milliseconds **(long, default: `30000`)**

validationQuery
: sql query that will be used to validate connections from this pool **(String, no default)**

validatorClassName
: name of a class which implements the org.apache.tomcat.jdbc.pool.Validator **(String, no default)**

Tip

The connection pool settings for xd are located in servers.yml (i.e. `spring.datasource.*`)

HDFS to MongoDB Export (`hdfsmongodb`)

Exports CSV data from HDFS and stores it in a MongoDB collection which defaults to the job name. This can be overridden with the `collectionName` parameter. 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" --
deploy
```

The **hdfsmongodb** job has the following options:

collectionName
: the MongoDB collection to store **(String, default: `<job name>`)**

databaseName
: the MongoDB database name **(String, default: xd)**

delimiter
: the delimiter for the delimited file **(String, default: `,`)**

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

host
: the MongoDB host **(String, default: `localhost`)**

idField
: the name of the field to use as the identity in MongoDB **(String, no default)**

names
: the field names in the CSV file **(String, no default)**

port
: the MongoDB port **(int, default: `27017`)**

resources
: the list of paths to import (Spring resources) **(String, no default)**

restartable
: whether the job should be restartable or not in case of failure **(boolean, default: `false`)**

FTP to HDFS Export (`ftphdfs`)

Copies files from FTP directory into HDFS. Job is partitioned in a way that each separate file copy is executed on its own partitioned step.

An example which copies files:

```
job create --name ftphdfsjob --definition "ftphdfs --host=ftp.example.com --port=21" --
deploy
job launch --name ftphdfsjob --params {"remoteDirectory":"/pub/files","hdfsDirectory":"/
ftp"}
```

Full path is preserved so that above command would result files in HDFS shown below:

```
/ftp/pub/files
/ftp/pub/files/file1.txt
/ftp/pub/files/file2.txt
```

The **ftphdfs** job has the following options:

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

host
: the host name for the FTP server **(String, no default)**

partitionResultsTimeout
: time (ms) that the partition handler will wait for results **(long, default: `300000`)**

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

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

restartable
: whether the job should be restartable or not in case of failure **(boolean, default: `false`)**

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

10. Streams

10.1 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.

10.2 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 Started 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.

Then to deploy the stream execute the following shell command (or alternatively add the `--deploy` flag when creating the stream so that this step is not needed):

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

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.

[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

```
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
```

If you would like to have multiple instances of a module in the stream, you can include a property with the deploy command:

```
xd:> stream deploy --name ticktock --properties "module.time.count=3"
```

You can also include a SpEL Expression as a `criteria` property for any module. That will be evaluated against the attributes of each currently available Container. Instances of the module will only be deployed to Containers for which the expression evaluates to true.

```
xd:> stream deploy --name ticktock --properties
 "module.time.count=3,module.log.criteria=groups.contains('x')"
```

10.3 Deleting a Stream

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

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

10.4 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
```

10.5 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 --deploy
```

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.

10.6 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 create --definition "http | transform --expression=payload.toUpperCase() |
 log" --name myprocstrem --deploy
```

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.

10.7 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.

10.8 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.

11. Modules

11.1 Introduction

Spring XD 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 Spring XD module types include `source`, `sink`, `processor`, and `job`. The type indicates how the modules may be composed in a stream, or used to deploy a batch job. Specifically, a source polls an external resource, or is triggered by an event and only provides an output. The first module in a stream must be a source. A processor performs some type of transformation or business logic on the message and requires both an input and output. A sink provides only an input and outputs data to an external resource to terminate the stream. A job module implements a Spring Batch job.

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

However, if you are interested in extending Spring 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.

11.2 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 ModuleDefinition 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*

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 Spring 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 options. Options may be required or optional where optional properties define a default value in Module Options Metadata. Module options metadata may be provided in a properties file or a Java class.

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

```
<beans>

  <bean class="org.springframework.integration.x.twitter.TwitterSearchChannelAdapter">
    <constructor-arg ref="twitterTemplate"/>
    <property name="readTimeout" value="${readTimeout}"/>
    <property name="connectTimeout" value="${connectTimeout}"/>
    <property name="autoStartup" value="false"/>
    <property name="outputChannel" ref="output"/>
    <property name="query" value="${query}" />
    <property name="language" value="${language}" />
    <property name="geocode" value="${geocode}" />
    <property name="resultType" value="${resultType}"/>
    <property name="includeEntities" value="${includeEntities}"/>
  </bean>

  <bean id="twitterTemplate" class="org.springframework.social.twitter.api.impl.TwitterTemplate">
    <constructor-arg value="${consumerKey}"/>
    <constructor-arg value="${consumerSecret}"/>
  </bean>

  <int:channel id="output"/>

</beans>
```

Note the property placeholders such as *query*, *language*, *consumerKey* and *consumerSecret*. Spring XD will substitute values for all of these properties as configured for each module instance. The options for this module are defined in TwitterSearchOptionsMetadata.java

For example, we can create two different streams, each using the *twittersearch* source with a different configuration.

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

and

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

In addition to options, modules may reference Spring beans such that each module instance may inject a different implementation of a bean. The ability to deploy the same module definition with different configurations is only possible because each module is created in its own application context. This results in some very useful features, such as the ability to use standard bean ids such as *input* and *output* and simple property names without having to worry about naming collisions. Overall, this allows modules to adhere to the KISS principle.

Stream Modules

Sources, Processors, and Sinks are built using Spring Integration and are typically perform one small task so that they may be easily reused. 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 typically performs some type of transformation on the message, using the message contents to create a new message.

- A *sink* is a valid message flow that contains a direct channel named *input* and an outbound adapter, or service activator used to provide the message to an external resource, HDFS for example.

For example, take a look at the file source which simply polls a directory using a file inbound adapter and file sink which appends an incoming message payload 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 input and output channels are always named *input* and *output*, in keeping with the KISS principle (let us know if you come up with some simpler names). These names are required to bind these channels to the messaging transport.

Also observe 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). A stream definition using the file source may specify the the directory name by providing a value for the *dir* option. If not provided, it will default to the stream name, which is contained in the `xd.stream.name` property provided to the module by the Spring XD runtime, see file source metadata. By convention, XD defined properties are prefixed with *xd*. This can be seen when using the `module info` command:

```
xd:>module info --name source:file
Information about source module 'file':

  Option Name        Description
        Default                          Type
  -----------------
 ---------------------------------------------------------------------------
 -------------------------------  --------
  dir                the absolute path to the directory to monitor for files
        /tmp/xd/input/${xd.stream.name}  String
  pattern            a filter expression (Ant style) to accept only files that match the
 pattern   *                               String
  preventDuplicates  whether to prevent the same file from being processed twice
        true                             boolean
  ref                set to true to output the File object itself
        false                            boolean
  fixedDelay         the fixed delay polling interval specified in seconds
        5                                int
  outputType         how this module should emit messages it produces
        <none>                           MimeType
```

Placeholders available to all modules

Below is the list of all available `${xd.xxx}` keys that module authors may use in their declaration.

Using placeholders in stream definitions

One can also use the `${xd.xxx}` notation directly inside the DSL definition of a stream or a job. For example:

```
xd:>stream create foo --definition "http | filter --expression=
\"'${xd.stream.name}'\" | log"
```

will only let messages that read "foo" pass through.

Placeholder	Context	Meaning
`${xd.stream.name}`	streams	the name of the stream the module lives in
`${xd.job.name}`	jobs	the name of the job the module lives in
`${xd.module.name}`	streams, jobs	the technical name of the module
`${xd.module.type}`	streams, jobs	the type of the module
`${xd.module.index}`	streams	the 0-based position of the module inside the stream
`${xd.container.id}`	streams, jobs	the generated unique id of the container the module is deployed in
`${xd.container.host}`	streams, jobs	the hostname of the container the module is deployed in
`${xd.container.pid}`	streams, jobs	the process id of the container the module is deployed in
`${xd.container.ip}`	streams, jobs	the IP address of the container the module is deployed in
`${xd.container.<foo>}`	streams, jobs	the value of the custom attribute `<foo>` for the container

11.3 Registering a Module

Spring 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 Spring XD runtime. By default the Spring XD is configured with the FileModuleRegistry which looks for modules in `${xd.home:..}/modules`. Where `xd.home` is set by the environment variable `XD_HOME` or passed as a Java System property to the container launcher, . So out of the box, the modules are contained in the xd/modules directory where Spring XD is installed. The modules directory organizes module types in sub-directories. So you will see something like:

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

To register a module simply drop your module files (XML configuration, options metadata, and any dependent jars that are not already on the Spring XD classpath) into the xd/modules directory.

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 support for modules that have specific library dependencies.

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 (installed into xd/lib).

11.4 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 share a common processing chain. 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 these two streams are the same. Composite Modules are a good way to avoid this type of duplication. In addition, Composite modules are co-located in a single container and may communicate directly in memory.

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 modules uses a bridge between in-memory channels. When deploying a stream to a distributed runtime environment, however, the communication between each module typically occurs over messaging middleware, as modules are distributed evenly among the available containers. At times a stream will perform better if adjacent modules are co-located and can avoid middleware "hops". In such cases, you may wrap multiple modules together so that they act as a single "black box." In other words, if "foo | bar" are composed together as a new module named "baz", the input and/or output to "baz" will still go over the middleware, but foo and bar will be co-located and communicate via a local memory.

Let's look at an example. Returning to the two similar streams above, the filter processor and file sink may be combined into a single module using the `module compose` shell command:

```
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
      Source              Processor           Sink                     Job
  ------------------  ------------------  -----------------------  ----------------
      file                aggregator          aggregate-counter        filejdbc
      gemfire             http-client         counter                  ftphdfs

                                   (....)

      trigger                                 splunk
      twittersearch                           tcp
      twitterstream                           throughput-sampler
      time                                (c) foo
```

Notice that the composed module shows up in the list of **sink** modules. That is because logically it acts as 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). Also notice that the module has a small `(c)` prefixed to it, to indicate that it is a composed module.

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

```
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:

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

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 sink:foo
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 sink:foo
Successfully destroyed module 'foo' with type sink
```

When creating a module, if you duplicate the name of an existing module for the same type, you will receive an error. In the example below the user tried to compose a tcp module, however one already exists:

```
xd:>module compose tcp --definition "filter --expression=payload.contains('foo') | file"
14:52:27,781  WARN Spring Shell client.RestTemplate:566 - POST request for "http://
ec2-50-16-24-31.compute-1.amazonaws.com:9393/modules" resulted in 409 (Conflict); invoking
 error handler
Command failed org.springframework.xd.rest.client.impl.SpringXDException: There is already
 a module named 'tcp' with type 'sink'
```

However, you can create a module for a given type even though a module of that name exists but as a different type. For example: I can create a sink module named filter, even though a filter module exists already as a processor.

Finally, it’s worth mentioning that in some cases duplication may be avoided by reusing an actual stream rather than a composed module. This 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.

11.5 Getting Information about Modules

To view the available modules use the the `module list` command. Modules appearing with a `(c)` marker are composed modules. For example:

```
xd:>module list
      Source              Processor           Sink                     Job
  ------------------  ------------------  -----------------------  ----------------
      file                aggregator          aggregate-counter        filejdbc
      gemfire             analytic-pmml       counter                  ftphdfs
      gemfire-cq          http-client         field-value-counter      hdfsjdbc
      http                bridge              file                     hdfsmongodb
      jms                 filter              gauge                    jdbchdfs
      mail                json-to-tuple       gemfire-json-server      filepollhdfs
      mqtt                object-to-json      gemfire-server
      post                script              jdbc
      reactor-syslog      splitter            mail
      reactor-tcp         transform           mqtt
      syslog-tcp      (c) myfilter            rich-gauge
      syslog-udp                              splunk
      tail                                    tcp
      tcp                                     throughput-sampler
      tcp-client                              avro
      trigger                                 hdfs
      twittersearch                           log
      twitterstream                           rabbit
      rabbit                                  router
      time
```

To get information about a particular module (such as what options it accepts), use the `module info --<module type>:<module name>` command. For example:

```
xd:>module info --name source:file
Information about source module 'file':

  Option Name        Description
        Default  Type
  -----------------
 ---------------------------------------------------------------------------  -------
 ---------
  dir                the absolute path to the directory to monitor for files
        <none>   String
  pattern            a filter expression (Ant style) to accept only files that match the
 pattern  *        String
  outputType         how this module should emit messages it produces
        <none>   MimeType
  preventDuplicates  whether to prevent the same file from being processed twice
        true     boolean
  ref                set to true to output the File object itself
        false    boolean
  fixedDelay         the fixed delay polling interval specified in seconds
        5        int
```

11.6 How module options are resolved

As we've seen so far, a module is a re-usable piece of Spring Integration (or Spring Batch) software that can be dynamically configured thru the use of **module options**.

A module option is any value that the module author has deemed worthy of configuration at deployment time. Preferably, the module author will have provided metadata to describe the available options. This section explains how default values are computed for each module option.

In a nutshell, actual values are drawn from the following 3 sources, from most precedent to least precedent:

1. actual values in the stream definition (*e.g.* `--foo=bar`)

2. platform-wide defaults (appearing *e.g.* in .yml and .properties files, see below)

3. defaults the module author chose (see metadata)

Going into more detail, the mid layer above (platform-wide defaults) will resolve like so, assuming option `<optionname>` of module `<modulename>` (which is of type `<moduletype>`):

a. a **system property** named `<moduletype>.<modulename>.<optionname>`

b. an **environment variable** named `<moduletype>.<modulename>.<optionname>` (or `<MODULETYPE>_<MODULENAME>_<OPTIONNAME>`)

c. a key named `<optionname>` in the **properties** file `<root>/<moduletype>/<modulename>/<modulename>.properties`

d. a key named `<moduletype>.<modulename>.<optionname>` in the **YaML** file `<root>/<module-config>.yml`

where

`<root>`
: is the value of the `xd.module.config.location` system property (driven by the `XD_MODULE_CONFIG_LOCATION` env var when using the canonical Spring XD shell scripts). Defaults to `${xd.config.home}/modules/`

`<module-config>`
: is the value of the `xd.module.config.name` system property (driven by the `XD_MODULE_CONFIG_NAME` env var). Defaults to `xd-module-config`

Note that YaML is particularly well suited for hierarchical configuration, so for example, instead of

```
source.file.dir: foo
source.file.pattern: *.txt

source.http.port: 1234
```

one can write

```
source:
  file:
    dir: foo
    pattern: *.txt
  http:
    port: 1234
```

Note that options in the `.properties` files can reference values that appear in the `modules.yml` file (this makes sharing common configuration easy). Also, the values that are used to configure the server runtimes (in `servers.yml`) are visible to `modules.yml` and `.properties` file (but the inverse is *not* true).

12. Sources

12.1 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
- Reactor IP
- JMS
- RabbitMQ
- Time
- MQTT
- Stdout Capture

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.

12.2 HTTP

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

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

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
```

To send binary data, set the `Content-Type` header to `application/octet-string`

```
$ curl --data-binary @foo.zip -H'Content-Type: application-octet-string' http://
localhost:9000
```

HTTP with options

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

https
: true for https:// **(boolean, default: `false`)**

port
: the port to listen to **(int, default: `9000`)**

sslPropertiesLocation
: location (resource) of properties containing the location of the pkcs12 keyStore and pass phrase **(String, default: `classpath:httpSSL.properties`)**

Here is an example

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

Post some data to the new port

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

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

Note

When using `https`, you need to provide a properties file that references a pkcs12 key store (containing the server certificate(s)) and its passphrase. Setting `--https=true` enables https:// and the module looks for the SSL properties in resource `classpath:httpSSL.properties`. This location can be overridden with the `--sslPropertiesLocation` property. For example:

```
xd:> stream create --name https9021 --definition "http --port=9021 --https=true --
sslPropertiesLocation=file:/secret/ssl.properties | file" --deploy
```

```
$ cat /secret/ssl.properties
keyStore=file:/secret/httpSource.p12
keyStore.passPhrase=secret
```

Since this properties file contains sensitive information, it will typically be secured by the operating system with the XD container process having read access.

12.3 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" --deploy
```

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 the following options:

delay
: how often (ms) to poll for new lines (forces use of the Apache Tailer, requires nativeOptions='') **(long, no default)**

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

fromEnd
: whether to tail from the end (true) or from the start (false) of the file (forces use of the Apache Tailer, requires nativeOptions='') **(boolean, no default)**

lines
: the number of lines prior to the end of an existing file to tail; does not apply if *nativeOptions* is provided **(int, default: `0`)**

name
: the absolute path of the file to tail **(String, default: `/tmp/xd/input/<stream name>`)**

nativeOptions
: options for a native tail command; do not set and use *end*, *delay*, and/or *reOpen* to use the Apache Tailer **(String, no default)**

reOpen
: whether to reopen the file each time it is polled (forces use of the Apache Tailer, requires nativeOptions='') **(boolean, no default)**

Here is an example

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

```
$ 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]
```

12.4 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" --deploy
```

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" --
deploy
```

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 the following options:

dir
: the absolute path to the directory to monitor for files **(String, default: `/tmp/xd/input/<stream name>`)**

fixedDelay
: the fixed delay polling interval specified in seconds **(int, default: 5)**

pattern
: a filter expression (Ant style) to accept only files that match the pattern **(String, default: *)**

preventDuplicates
: whether to prevent the same file from being processed twice **(boolean, default: `true`)**

ref
: set to true to output the File object itself **(boolean, default: `false`)**

The `ref` option is useful in some cases in which the file contents are large and it would be more efficient to send the file path.

12.5 Mail

Spring XD provides a source module for receiving emails, named `mail`. Depending on the protocol used, in can work by polling or receive mails as they become available.

Let’s see an example:

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

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` source is below:

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

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

delete
: whether to delete the emails once they’ve been fetched **(boolean, default: `true`)**

expression
: a SpEL expression which filters which mail messages will be processed (non polling imap only) **(String, default: `true`)**

fixedDelay
: the polling interval used for looking up messages (s) **(int, default: `60`)**

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

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

markAsRead
: whether to mark emails as read once they’ve been fetched **(boolean, default: `false`)**

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

port
: the port of the mail server **(int, default: `25`)**

protocol
: the protocol to use to retrieve messages **(MailProtocol, default: `imap`, possible values: `imap,imaps,pop3,pop3s`)**

usePolling
: whether to use polling or not (no polling works with imap(s) only) **(boolean, default: `false`)**

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

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.

12.6 Twitter Search

The twittersearch source runs a continuous query against Twitter.

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

connectTimeout
: the connection timeout for making a connection to Twitter (ms) **(int, default: `5000`)**

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

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

geocode
: geo-location given as latitude,longitude,radius. e.g., *37.781157,-122.398720,1mi* **(String, default: ``)**

includeEntities
: whether to include entities such as urls, media and hashtags **(boolean, default: `true`)**

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

query
: the query string **(String, default: ``)**

readTimeout
: the read timeout for the underlying URLConnection to the twitter stream (ms) **(int, default: `9000`)**

resultType
: result type: recent, popular, or mixed **(ResultType, default: `mixed`, possible values: `mixed,recent,popular`)**

For information on how to construct a query, see the Search API v1.1.

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.

Tip

For both `twittersearch` and `twitterstream` you can put these keys in a module properties file instead of supplying them in the stream definition. If both sources share the same credentials, it is easiest to configure the required credentials in `config/modules/modules.yml`. Alternately, each module has its own properties file. For twittersearch, the file would be `config/modules/source/twittersearch/twittersearch.properties`.

To create and deploy a stream definition in the server using the XD shell:

```
xd:> stream create --name springone2gx --definition "twittersearch --query='#springone2gx'
 | file" --deploy
```

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
```

Note

Both `twittersearch` and `twitterstream` emit JSON in the native Twitter format.

12.7 Twitter Stream

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

You need to supply all keys and secrets (both consumer and accessToken) to authenticate for this source, so it is easiest if you just add these to `XD_HOME/config/modules/modules.yml` or `XD_HOME/config/modules/source/twitterstream/twitterstream.properties` file.

Stream creation is then straightforward:

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

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

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

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

connectTimeout
: the connection timeout for making a connection to Twitter (ms) **(int, default: `5000`)**

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

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

delimited

set to true to get length delimiters in the stream data **(boolean, default: `false`)**

discardDeletes

set to discard *delete* events **(boolean, default: `true`)**

filterLevel

controls which tweets make it through to the stream: none,low,or medium **(FilterLevel, default: none, possible values: `none,low,medium`)**

follow

comma delimited set of user ids whose tweets should be included in the stream **(String, default: ``)**

language

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

locations

comma delimited set of latitude/longitude pairs to include in the stream **(String, default: ``)**

readTimeout

the read timeout for the underlying URLConnection to the twitter stream (ms) **(int, default: `9000`)**

stallWarnings

set to true to enable stall warnings **(boolean, default: `false`)**

track

comma delimited set of terms to include in the stream **(String, default: ``)**

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

Note

Both `twittersearch` and `twitterstream` emit JSON in the native Twitter format.

12.8 GemFire Source

This source configures a client cache and client region, along with the necessary subscriptions enabled, 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** source has the following options:

cacheEventExpression

an optional SpEL expression referencing the event **(String, default: `newValue`)**

host

host name of the cache server or locator (if useLocator=true) **(String, default: `localhost`)**

port

port of the cache server or locator (if useLocator=true) **(int, default: `40404`)**

regionName
: the name of the region for which events are to be monitored **(String, default: `<stream name>`)**

useLocator
: set to true if using a locator **(boolean, default: `false`)**

Example

Use of the gemfire source requires an external process (or a separate stream) that creates or updates entries in a GemFire region configured for a cache server. Such events may feed an XD stream. To enable such a stream, the XD container must join a GemFire distributed client-server system as a client, creating a client region corresponding to an existing region on a cache server. The client region registers a cache listener via the Spring Integration GemFire inbound channel adapter. The client region and pool are configured for a subscription on all keys in the region.

The following example creates two streams: One to write http messages to a Gemfire region named *Stocks*, and another to listen for cache events and record the updates to a file. This works with the Cache Server and sample configuration included with the Spring XD distribution:

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

Now send some messages to the stocks stream.

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

Note

Avoid spaces in the JSON when using the shell to post data

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

```
$ cat /tmp/xd/output/gftest.out

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

Launching the XD GemFire Server

This source requires a cache server to be running in a separate process and its host and port, or a locator host and port must be configured. 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
```

12.9 GemFire Continuous Query

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.

Options

The **gemfire-cq** source has the following options:

host
: the host on which the GemFire server is running **(String, default: `localhost`)**

port
: the port on which the GemFire server is running **(int, default: `40404`)**

query
: the query string in Object Query Language (OQL) **(String, no default)**

useLocator
: set to true if using a locator **(boolean, default: `false`)**

The example is similar to that presented for the gemfire source above, and requires an external cache server as described in the above section. In this case the query provides a finer filter on data events. In the example below, the `cqtest` stream will only receive events matching a single ticker symbol, whereas the `gftest` stream example above will receive updates to every entry in the region.

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

Now send some messages to the stocks stream.

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

The *cqtest* stream is now listening for any stock quote updates for the ticker symbol `FAKE`. As updates are posted to the cache you should see them captured in the output file:

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

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

12.10 Syslog

Three syslog sources are provided: `reactor-syslog`, `syslog-udp`, and `syslog-tcp`. The reactor-syslog adapter uses tcp and builds upon the functionality available in the Reactor project and provides improved throughput over the syslog-tcp adapter.

The **reactor-syslog** source has the following options:

port
: the port on which the system will listen for syslog messages **(int, default: `5140`)**

The **syslog-udp** source has the following options:

port
: the port on which the system will listen for syslog messages **(int, default: `5140`)**

The **syslog-tcp** source has the following options:

port
: the port on which the system will listen for syslog messages **(int, default: `5140`)**

To create a stream definition (using shell command)

```
xd:> stream create --name syslogtest --definition "reactor-syslog --port=5140 | file" --deploy
```

or

```
xd:> stream create --name syslogtest --definition "syslog-udp --port=5140 | file" --deploy
```

or

```
xd:> stream create --name syslogtest --definition "syslog-tcp --port=5140 | file" --deploy
```

(`--port` is not required when using the default `5140`)

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:5140
```

TCP - Ubuntu (rsyslog.conf)

```
$ModLoad omfwd
*.*    @@localhost:5140
```

Restart the syslog daemon after reconfiguring.

12.11 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" --deploy
```

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
```

By default, the TCP module will emit a `byte[]`; to convert to a String, add `--outputType=text/plain` to the module definition.

TCP with options

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

bufferSize
: the size of the buffer (bytes) to use when encoding/decoding **(int, default: `2048`)**

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

decoder
: the decoder to use when receiving messages **(Encoding, default: `CRLF`, possible values: `CRLF,LF,NULL,STXETX,RAW,L1,L2,L4`)**

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

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

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

socketTimeout
: the timeout (ms) before closing the socket when no data is received **(int, default: `120000`)**

useDirectBuffers
: whether or not to use direct buffers **(boolean, default: `false`)**

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" --deploy
```

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" --
deploy
```

```
$ 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" --
deploy
```

```
$ 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"
 --deploy
```

```
$ 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" --
deploy
```

```
$ 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" --
deploy
```

```
$ 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" --
deploy
```

```
$ 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" --
deploy
```

```
$ 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" --deploy
```

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
```

12.12 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 **tcp-client** source has the following options:

bufferSize
: the size of the buffer (bytes) to use when encoding/decoding **(int, default: `2048`)**

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

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

decoder
: the decoder to use when receiving messages **(Encoding, default: `CRLF`, possible values: `CRLF,LF,NULL,STXETX,RAW,L1,L2,L4`)**

encoder
: the encoder to use when sending messages **(Encoding, default: `CRLF`, possible values: `CRLF,LF,NULL,STXETX,RAW,L1,L2,L4`)**

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

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

host
: the remote host to connect to **(String, default: `localhost`)**

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

port
: the port on the remote host to connect to **(int, default: `1234`)**

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

script
: reference to a script used to transform messages **(String, no default)**

socketTimeout
: the timeout (ms) before closing the socket when no data is received **(int, default: `120000`)**

useDirectBuffers
: whether or not to use direct buffers **(boolean, default: `false`)**

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
```

12.13 Reactor IP

The `reactor-ip` source acts as a server and allows a remote party to connect to XD and submit data over a raw TCP or UDP socket. The reactor-ip source differs from the standard tcp source in that it is based on the Reactor Project and can be configured to use the LMAX Disruptor RingBuffer library allowing for extremely high ingestion rates, e.g. ~ 1M/sec.

To create a stream definition use the following XD shell command

```
xd:> stream create --name tcpReactor --definition "reactor-ip | file" --deploy
```

This will create the reactor TCP source and send data read from it to the file named tcpReactor.

The **reactor-ip** source has the following options:

codec
: codec used to transcode data **(String, default: `string`)**

dispatcher
: type of Reactor Dispatcher to use **(String, default: `ringBuffer`)**

framing
: method of framing the data **(String, default: `linefeed`)**

host
: host to bind the server to **(String, default: `0.0.0.0`)**

lengthFieldLength
: byte precision of the number used in the length field **(int, default: `4`)**

port
: port to bind the server to **(int, default: `3000`)**

transport
: whether to use TCP or UDP as a transport **(String, default: `tcp`)**

12.14 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" --deploy
```

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 **rabbit** source has the following options:

ackMode
: the acknowledge mode (AUTO, NONE, MANUAL) **(String, default: `AUTO`)**

addresses
: a comma separated list of *host[:port]* addresses **(String, default: `${spring.rabbitmq.addresses}`)**

concurrency
: the minimum number of consumers **(int, default: `1`)**

converterClass
: the class name of the message converter **(String, default: `org.springframework.amqp.support.converter.SimpleMessageConverter`)**

mappedRequestHeaders
: request message header names to be propagated to/from the adpater/gateway **(String, default: `STANDARD_REQUEST_HEADERS`)**

maxConcurrency
: the maximum number of consumers **(int, default: `1`)**

password
: the password to use to connect to the broker **(String, default: `${spring.rabbitmq.password}`)**

prefetch
: the prefetch size **(int, default: `1`)**

queues
: the queue(s) from which messages will be received **(String, default: `<stream name>`)**

requeue
: whether rejected messages will be requeued by default **(boolean, default: `true`)**

sslPropertiesLocation
: resource containing SSL properties **(String, default: `${spring.rabbitmq.sslProperties}`)**

transacted
: true if the channel is to be transacted **(boolean, default: `false`)**

txSize
: the number of messages to process before acking **(int, default: `1`)**

useSSL
: true if SSL should be used for the connection **(String, default: `${spring.rabbitmq.useSSL}`)**

username
: the username to use to connect to the broker **(String, default: `${spring.rabbitmq.username}`)**

vhost
: the RabbitMQ virtual host to use **(String, default: `${spring.rabbitmq.virtual_host}`)**

See the RabbitMQ MessageBus Documentation for more information about SSL configuration.

12.15 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" --deploy
```

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`.

The out of the box configuration is setup to use ActiveMQ. To use another JMS provider you will need to update a few files in the XD distribution. There are sample files for HornetMQ in the distribution as an example for you to follow. You will also need to add the appropriate libraries for your provider in the JMS module lib directory or in the main XD lib directory.

JMS with Options

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

clientId
: an identifier for the client, to be associated with a durable topic subscription **(String, no default)**

destination
: the destination name from which messages will be received **(String, default: `<stream name>`)**

durableSubscription
: when true, indicates the subscription to a topic is durable **(boolean, default: `false`)**

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

pubSub
: when true, indicates that the destination is a topic **(boolean, default: `false`)**

subscriptionName
: a name that will be assigned to the topic subscription **(String, no default)**

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.

12.16 Time

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

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

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

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

12.17 MQTT

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

Configure a stream:

```
xd:> stream create tcptest --definition "mqtt --url='tcp://localhost:1883' --
topics='xd.mqtt.test' | log" --deploy
```

If you wish to use the MQTT Source defaults you can execute the command as follows:

```
xd:> stream create tcptest --definition "mqtt | log" --deploy
```

Options

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

clientId
: identifies the client **(String, default: `xd.mqtt.client.id.src`)**

password
: the password to use when connecting to the broker **(String, default: `guest`)**

topics
: the topic(s) to which the source will subscribe **(String, default: `xd.mqtt.test`)**

url
: location of the mqtt broker **(String, default: `tcp://localhost:1883`)**

username
: the username to use when connecting to the broker **(String, default: `guest`)**

Note

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

12.18 Stdout Capture

There isn't actually a source named "stdin" but it is easy to capture stdin by redirecting it to a `tcp` source. For example if you wanted to capture the output of a command, you would first create the `tcp` stream, as above, using the appropriate sink for your requirements:

```
xd:> stream create tcpforstdout --definition "tcp --decoder=LF | log" --deploy
```

You can then capture the output from commands using the `netcat` command:

```
$ cat mylog.txt | netcat localhost 1234
```

13. Processors

13.1 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
- Transform
- Script
- Splitter
- Aggregator
- HTTP Client
- JSON to Tuple
- Object to JSON

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

13.2 Filter

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

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

expression
a SpEL expression to evaluate as a predicate **(String, no default)**

script
location of a groovy script to use as a predicate to the filter **(String, no default)**

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" --deploy
```

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

```
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 using jsonPath evaluation

As part of the SpEL expression you can make use of the pre-registered JSON Path function.

This filter example shows to 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 | filter --
expression=#jsonPath(payload,'$.firstName').contains('John') | log" --deploy
```

Note: There is no space between payload JSON and the jsonPath in the expression

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.

Here is another example usage of filter

```
filter --expression=#jsonPath(payload,'$.entities.hashtags[*].text').contains('obama')
```

This is an example that is operating on a JSON payload of tweets as consumed from the twitter search module.

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.

Note that an implicit variable named *payload* is available to give you access to the data contained in a message.

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

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.

13.3 Transform

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

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

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

script
: reference to a script used to transform messages **(String, no default)**

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" --deploy
```

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"
```

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

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"
 --deploy
```

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.

13.4 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"
 --deploy
```

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.

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

location
: the script to use (Spring resource, uses classpath: when no scheme given) **(String, no default)**

13.5 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** processor has the following options:

expression
: a SpEL expression which would typically evaluate to an array or collection **(String, default: `payload`)**

Note

The default value for `expression` is `payload`, which actually does not split, unless the message is already a collection.

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

Extract the value of a specific field

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

```
xd:> stream create --name jsontransformtest --definition "http --port=9005 | splitter --
expression=#jsonPath(payload,'$.firstName') | log" --deploy
```

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"}'
```

13.6 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" --deploy
```

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** processor has the following options:

aggregation

how to construct the aggregated message (SpEL expression against a collection of messages) **(String, default: `#this.![payload]`)**

correlation

how to correlate messages (SpEL expression against each message) **(String, default: `'<stream name>'`)**

count

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

dbkind

which flavor of init scripts to use for the jdbc store (blank to attempt autodetection) **(String, no default)**

driverClassName

the jdbc driver to use when using the jdbc store **(String, no default)**

hostname

hostname of the redis instance to use as a store **(String, default: `localhost`)**

initializeDatabase

whether to auto-create the database tables for the jdbc store **(boolean, default: `false`)**

password

the password to use when using the jdbc or redis store **(String, default: ``)**

port

port of the redis instance to use as a store **(int, default: `6379`)**

release

when to release messages (SpEL expression against a collection of messages accumulated so far) **(String, no default)**

store

the kind of store to use to retain messages **(StoreKind, default: `memory`, possible values: `memory,jdbc,redis`)**

timeout

the delay (ms) after which messages should be released, even if the completion criteria is not met **(int, default: `50000`)**

url

the jdbc url to connect to when using the jdbc store **(String, no default)**

username

the username to use when using the jdbc store **(String, no default)**

Note

- Some of the options are only relevant when using a particular `store`
- The default `correlation` of `'<stream name>'` actually considers all messages to be correlated, since they all belong to the same stream.

- Using the `release` option overrides the `count` option (which is a simpler approach)
- The default for `aggregation` creates a new collection made of the payloads of the accumulated messages
- About the `timeout` option: due to the way it is implemented (see MessageGroupStoreReaper in the Spring Integration documentation), the actual observed delay may vary between `timeout` and `2xtimeout`.

13.7 HTTP Client

The `http-client` processor acts as a client that issues HTTP requests to a remote server, submitting the message payload it receices to that server and in turn emitting the response it receives to the next module down the line.

For example, the following command will result in an immediate fetching of earthquake data and it being logged in the container:

```
xd:>stream create earthquakes --definition "trigger | http-client --url='''http://
earthquake.usgs.gov/earthquakes/feed/geojson/all/day''' --httpMethod=GET | log" --deploy
```

Note

Please be aware that the `url` option above is actually a SpEL expression, hence the triple quotes. If you'd like to learn more about quotes, please read the relevant documentation.

The **http-client** processor has the following options:

charset
: the charset to use when in the Content-Type header when emitting Strings **(String, default: `UTF-8`)**

httpMethod
: the http method to use when performing the request **(HttpMethod, default: `POST`, possible values: `OPTIONS,GET,HEAD,POST,PUT,PATCH,DELETE,TRACE,CONNECT`)**

mappedRequestHeaders
: request message header names to be propagated to/from the adpater/gateway **(String, default: `HTTP_REQUEST_HEADERS`)**

mappedResponseHeaders
: response message header names to be propagated from the adpater/gateway **(String, default: `HTTP_RESPONSE_HEADERS`)**

replyTimeout
: the amount of time to wait (ms) for a response from the remote server **(int, default: `0`)**

url
: the url to perform an http request on **(String, no default)**

13.8 JSON to Tuple

The `json-to-tuple` processor is able to transform a String representation of some JSON map into a Tuple.

Here is a simple example:

```
xd:>stream create tuples --definition "http | json-to-tuple | transform --
expression='payload.firstName + payload.lastName' | log" --deploy

xd:>http post --data '{"firstName": "Spring", "lastName": "XD"}'
```

Note

Transformation to Tuple can be used as an alternative or in addition of Type Conversion, depending on your usecase.

The **json-to-tuple** processor has no particular option (in addition to options shared by all modules)

13.9 Object to JSON

The `object-to-json` processor can be used to convert any java Objec to a JSON String.

In the following example, notice how the collection of three elements is transformed to JSON (in particular, the three Strings are surrounded by quotes):

```
xd:>stream create json --deploy --definition "http | aggregator --count | abject-to-json |
 log"

xd:>http post --data hello
xd:>http post --data world
xd:>http post --data !
```

results in `["hello", "world", "!"]` appearing in the log.

The **object-to-json** processor has no particular option (in addition to options shared by all modules)

14. Sinks

14.1 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
- HDFS Dataset
- JDBC
- TCP
- Mongo
- 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.

14.2 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" --deploy
```

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

```
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 INFO sink.mylogstream: hello
```

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

expression

: the expression to be evaluated for the log content; use *#root* to log the full message **(String, default: `payload`)**

level

: the log level **(String, default: `INFO`)**

name

: the name of the log category to log to (will be prefixed by *xd.sink.*) **(String, default: `<stream name>`)**

Here are some examples explaining the above options:

The logger name is the sink name prefixed with the string `xd.sink.`. 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" --deploy
```

The log level is `INFO` by default; this can be changed with the `--level` property (`FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE`)

```
xd:> stream create --name mylogstream --definition "http --port=8001 | log --level=WARN"
 --deploy
```

By default, the message payload is logged; this can be changed with the `--expression` property (e.g. `payload.foo` to log some property `foo` of the payload, or `#root` to log the entire message)

```
xd:> stream create --name mylogstream --definition "http --port=8001 | log --
expression=#root" --deploy
```

14.3 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" --deploy
```

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` options of the output file. For example

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

File with Options

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

binary
: if false, will append a newline character at the end of each line **(boolean, default: `false`)**

charset
: the charset to use when writing a String payload **(String, default: `UTF-8`)**

dir
: the directory in which files will be created **(String, default: `/tmp/xd/output/`)**

mode
: what to do if the file already exists **(Mode, default: `APPEND`, possible values: `APPEND,REPLACE,FAIL,IGNORE`)**

name
: filename pattern to use **(String, default: `<stream name>`)**

suffix
: filename extension to use **(String, no default)**

14.4 Hadoop (HDFS)

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

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

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

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

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

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

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

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

When you destroy a stream

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Often a stream of data may not have a high enough rate to roll over files frequently, leaving the file in an opened state. This prevents users from reading a consistent set of data when running mapreduce jobs. While one can alleviate this problem by using a small rollover value, a better way is to use the `idleTimeout` option that will automatically close the file if there was no writes during the specified period of time. This feature is also useful in cases where burst of data is written into a stream and you’d like that data to become visible in HDFS.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
xd:>stream destroy --name myhdfsstream8
Destroyed stream 'myhdfsstream8'
xd:>hadoop fs ls --recursive true --dir /xd
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/myhdfsstream8
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/myhdfsstream8/2014
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/10_range
-rw-r--r--   3 jvalkealahti supergroup         16 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/10_range/myhdfsstream8-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/3_range
-rw-r--r--   3 jvalkealahti supergroup         35 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/3_range/myhdfsstream8-0.txt
drwxr-xr-x   - jvalkealahti supergroup          0 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/5_range
-rw-r--r--   3 jvalkealahti supergroup          5 2014-05-28 19:34 /xd/
myhdfsstream8/2014/05/28/19/5_range/myhdfsstream8-0.txt
xd:>hadoop fs cat /xd/myhdfsstream8/2014/05/28/19/3_range/myhdfsstream8-0.txt
APP3
APP3
APP1
APP0
APP1
xd:>hadoop fs cat /xd/myhdfsstream8/2014/05/28/19/5_range/myhdfsstream8-0.txt
APP4
xd:>hadoop fs cat /xd/myhdfsstream8/2014/05/28/19/10_range/myhdfsstream8-0.txt
APP6
APP15
APP7
```

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

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

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

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

HDFS with Options

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

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

directory
: where to output the files in the Hadoop FileSystem **(String, default: `/xd/<stream name>`)**

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

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

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

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

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

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

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

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

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

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

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

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

Partition Path Expression

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

Accessing Properties

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

Custom Methods

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

path

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

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

Parameters

paths
: Any number of path parts

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

dateFormat

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

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

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

Parameters

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

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

date
A `Date` to be formatted.

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

datestring
`Date` as a `String`

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

list

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

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

Lets assume that data is being written and it’s possible to extrace an `appid` either from headers or payload. We can automatically do a list based partition by using a partition method `list(headers.appid,{{'1TO3','APP1','APP2','APP3'},{'4TO6','APP4','APP5','APP6'}})`. This method would create three partitions, `1TO3_list`, `4TO6_list` and `list`. Latter is used if no match is found from partition lists passed to `lists`.

Parameters

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

lists
A definition of list of lists.

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

range

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

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

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

source
An Object to be matched against list.

list
A definition of list.

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

hash

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

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

source
An Object which hashCode will be used.

bucketcount
A number of buckets

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

14.5 HDFS Dataset (Avro/Parquet)

The HDFS Dataset sink is used to store Java classes that are sent as the payload on the stream. It uses the Kite SDK Data Module's Dataset implementation to store the payload data serialized in either Avro or Parquet format. The Avro schema is generated from the Java class that is persisted. For Parquet the

Java object must follow JavaBean conventions with properties for any fields to be persisted. The fields can only be simple scalar values like Strings and numbers.

The HDFS Dataset sink requires that you have a Hadoop installation that is based on Hadoop v2 (Hadoop 2.2.0, Pivotal HD 1.0, Cloudera CDH4 or Hortonworks HDP 2.0), 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-dataset` sink when creating a stream

```
xd:>stream create --name mydataset --definition "time | hdfs-dataset --batchSize=20" --
deploy
```

In the above example, we've scheduled `time` source to automatically send ticks to the `hdfs-dataset` sink once every second. The data will be stored in a directory named `/xd/<streamname>` by default, so in this example it will be `/xd/mydataset`. You can change this by supplying a `--directory` parameter. The Avro format is usd by default and the data files are stored in a sub-directory named after the payload Java class. In this example the stream payload is a String so the name of the data sub-directory is `string`. If you have multiple Java classes as payloads, each class will get its own sub-directory.

Let the stream run for a minute or so. 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. We use the hdfs protocol is to access the hadoop name node.

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

Then list the contents of the stream's data directory.

```
xd:>hadoop fs ls /xd/mydataset/string
Found 3 items
drwxr-xr-x   - trisberg supergroup          0 2013-12-19 12:23 /xd/mydataset/
string/.metadata
-rw-r--r--   3 trisberg supergroup        202 2013-12-19 12:23 /xd/mydataset/
string/1387473825754-63.avro
-rw-r--r--   3 trisberg supergroup        216 2013-12-19 12:24 /xd/mydataset/
string/1387473846708-80.avro
```

You can see that the sink has created two files containing the first two batches of 20 stream payloads each. There is also a `.metadata` directory created that contains the metadata that the Kite SDK Dataset implementation uses as well as the generated Avro schema for the persisted type.

```
xd:>hadoop fs ls /xd/mydataset/string/.metadata
Found 2 items
-rw-r--r--   3 trisberg supergroup        136 2013-12-19 12:23 /xd/mydataset/
string/.metadata/descriptor.properties
-rw-r--r--   3 trisberg supergroup          8 2013-12-19 12:23 /xd/mydataset/
string/.metadata/schema.avsc
```

Now destroy the stream.

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

HDFS Dataset with Options

The HDFS Dataset Sink has the following options:

batchSize
: The number of payload objects that will be stored in each write operation. **(default: `10000`)**

directory
: Where the files will be written in the Hadoop FileSystem **(default: `/xd/<streamname>`)**

idleTimeout
: Idle timeout in milliseconds for when the aggregated batch of payload objects will be written even if the batchSize has not been reached. **(default: `-1`, no timeout)**

allowNullValues
: Whether to allow null values in fields of the Java class to be written to the sink. If this is set to true then each field in the generated schema will use a union of *null* and the data type of the field. You can annotate fields in a POJO using Avro's `org.apache.avro.reflect.Nullable` annotation to create a schema using a union with *null* for that field. **(default: `false`)**

format
: The format to use when writing the dataset data. Options are `avro` and `parquet`. **(default: `avro`)**

partitionPath
: This option lets you specify one or more paths that will be used to partition the files that the data is written to based on the content of the data. You can use any of the FieldPartitioners that are available for the Kite SDK project. We simply pass in what is specified to create the corresponding partition strategy. You can separate multiple paths with a / character. The following partitioning functions are available:

- *year, month, day, hour, minute* creates partitions based on the value of a timestamp and creates directories named like "YEAR=2014" (works well with fields of datatype long)
  - specify function plus field name like: `year('timestamp')`
- *dateformat* creates partitions based on a timestamp and a dateformat expression provided - creates directories based on the name provided (works well with fields of datatype long)
  - specify function plus field name, a name for the partition and the date format like: `dateFormat('timestamp', 'Y-M', 'yyyyMM')`
- *range* creates partitions based on a field value and the upper bounds for each bucket that is specified (works well with fields of datatype int and string)
  - specify function plus field name and the upper bounds for each partition bucket like: `range('age',20,50,80,T(Integer).MAX_VALUE)` (Note that you can use SpEL expressions like we just did for the Integer.MAX_VALUE)
- *identity* creates partitions based on the exact value of a field (works well with fields of datatype string, long and int)
  - specify function plus field name, a name for the partition, the type of the field (String or Integer) and the number of values/buckets for the partition like: `identity('region','R',T(String),10)`
- *hash* creates partitions based on the hash calculated from the value of a field divided into a number of buckets that is specified (works well with all data types)
  - specify function plus field name and number of buckets like: `hash('lastname',10)`

Multiple expressions can be specified by separating them with a / like: `identity('region','R',T(String),10)/year('timestamp')/month('timestamp')`

14.6 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 a table named after the stream name in the HSQLDB database that XD uses to store metadata for batch jobs. To alter this behavior, the jdbc sink accepts several options that you can pass using the `--foo=bar` notation in the stream, or change globally. 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 a table with your specific layout when the sink starts. You should also change the *initializeDatabase* property to *true* to have this script execute when the sink starts up.

The payload data will be inserted as-is if the *names* 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 mydata --definition "time | jdbc --initializeDatabase=true" --
deploy
```

This will insert the time messages into a *payload* column in a table named *mydata*. Since the default is using the XD batch metadata HSQLDB database we can connect to this database instance from an external tool. After we let the stream run for a little while, we can connect to the database and look at the data stored in the database.

You can query the database with your favorite SQL tool using the following database URL: `jdbc:hsqldb:hsql://localhost:9101/xdjob` with `sa` as the user name and a blank password. You can also use the HSQL provided SQL Tool (download from HSQLDB) to run a quick query from the command line:

```
$ java -cp ~/Downloads/hsqldb-2.3.0/hsqldb/lib/sqltool.jar org.hsqldb.cmdline.SqlTool
 --inlineRc url=jdbc:hsqldb:hsql://localhost:9101/xdjob,user=sa,password= --sql "select
 payload from mydata;"
```

This should result in something similar to the following output:

```
2014-01-06 09:33:25
2014-01-06 09:33:26
2014-01-06 09:33:27
2014-01-06 09:33:28
2014-01-06 09:33:29
2014-01-06 09:33:30
2014-01-06 09:33:31
2014-01-06 09:33:32
2014-01-06 09:33:33
2014-01-06 09:33:34
2014-01-06 09:33:35
2014-01-06 09:33:36
2014-01-06 09:33:37
```

Now we can destroy the stream using:

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

JDBC with Options

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

abandonWhenPercentageFull
: connections that have timed out wont get closed and reported up unless the number of connections in use are above the percentage **(int, default: `0`)**

alternateUsernameAllowed
: uses an alternate user name if connection fails **(boolean, default: `false`)**

columns
: the database columns to map the data to **(String, default: `payload`)**

connectionProperties
: connection properties that will be sent to our JDBC driver when establishing new connections **(String, no default)**

driverClassName
: the JDBC driver to use **(String, no default)**

fairQueue
: set to true if you wish that calls to getConnection should be treated fairly in a true FIFO fashion **(boolean, default: `true`)**

initSQL
: custom query to be run when a connection is first created **(String, no default)**

initialSize
: initial number of connections that are created when the pool is started **(int, default: `0`)**

initializeDatabase
: whether the database initialization script should be run **(boolean, default: `false`)**

initializerScript
: the name of the SQL script (in /config) to run if *initializeDatabase* is set **(String, default: `init_db.sql`)**

jdbcInterceptors
: semicolon separated list of classnames extending org.apache.tomcat.jdbc.pool.JdbcInterceptor **(String, no default)**

jmxEnabled
: register the pool with JMX or not **(boolean, default: `true`)**

logAbandoned
: flag to log stack traces for application code which abandoned a Connection **(boolean, default: `false`)**

maxActive
: maximum number of active connections that can be allocated from this pool at the same time **(int, default: `100`)**

maxAge
: time in milliseconds to keep this connection **(int, default: `0`)**

maxIdle
: maximum number of connections that should be kept in the pool at all times **(int, default: `100`)**

maxWait
: maximum number of milliseconds that the pool will wait for a connection **(int, default: `30000`)**

minEvictableIdleTimeMillis
: minimum amount of time an object may sit idle in the pool before it is eligible for eviction **(int, default: `60000`)**

minIdle
: minimum number of established connections that should be kept in the pool at all times **(int, default: `10`)**

password
: the JDBC password **(Password, no default)**

removeAbandoned
: flag to remove abandoned connections if they exceed the removeAbandonedTimout **(boolean, default: `false`)**

removeAbandonedTimeout
: timeout in seconds before an abandoned connection can be removed **(int, default: `60`)**

suspectTimeout
: this simply logs the warning after timeout, connection remains **(int, default: `0`)**

tableName
: the database table to which the data will be written **(String, default: `<stream name>`)**

testOnBorrow
: indication of whether objects will be validated before being borrowed from the pool **(boolean, default: `false`)**

testOnReturn
: indication of whether objects will be validated before being returned to the pool **(boolean, default: `false`)**

testWhileIdle
: indication of whether objects will be validated by the idle object evictor **(boolean, default: `false`)**

timeBetweenEvictionRunsMillis
: number of milliseconds to sleep between runs of the idle connection validation/cleaner thread **(int, default: `5000`)**

url
: the JDBC URL for the database **(String, no default)**

useEquals
: true if you wish the ProxyConnection class to use String.equals **(boolean, default: `true`)**

username
: the JDBC username **(String, no default)**

validationInterval

avoid excess validation, only run validation at most at this frequency - time in milliseconds **(long, default: 30000)**

validationQuery

sql query that will be used to validate connections from this pool **(String, no default)**

validatorClassName

name of a class which implements the org.apache.tomcat.jdbc.pool.Validator **(String, no default)**

Note

To include the whole message into a single column, use `payload` (the default) for the `columns` option

Tip

The connection pool settings for xd are located in servers.yml (i.e. `spring.datasource.*`)

14.7 TCP Sink

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" --deploy
```

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:

bufferSize

the size of the buffer (bytes) to use when encoding/decoding **(int, default: 2048)**

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

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

encoder
: the encoder to use when sending messages **(Encoding, default: `CRLF`, possible values: `CRLF,LF,NULL,STXETX,RAW,L1,L2,L4`)**

host
: the remote host to connect to **(String, default: `localhost`)**

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

port
: the port on the remote host to connect to **(int, default: `1234`)**

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

socketTimeout
: the timeout (ms) before closing the socket when no data is received **(int, default: `120000`)**

useDirectBuffers
: whether or not to use direct buffers **(boolean, default: `false`)**

Note

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" --deploy
```

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.

14.8 Mongo

The Mongo sink writes into a Mongo collection. Here is a simple example

```
xd:>stream create --name attendees --definition "http | mongodb --databaseName=test --
collectionName=names" --deploy
```

Then,

```
xd:>http post --data {"firstName":"mark"}
```

In the mongo console you will see the document stored

```
> use test
switched to db test
> show collections
names
system.indexes
> db.names.find()
{ "_id" : ObjectId("53c93bc324ac76925a77b9df"), "firstName" : "mark" }
```

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

collectionName

the MongoDB collection to store **(String, default: `<stream name>`)**

databaseName
: the MongoDB database name **(String, default: `xd`)**

host
: the MongoDB host **(String, default: `localhost`)**

port
: the MongoDB port **(int, default: `27017`)**

14.9 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'" --deploy
```

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 **mail** sink has the following options:

bcc
: the recipient(s) that should receive a blind carbon copy (SpEL) **(String, default: `null`)**

cc
: the recipient(s) that should receive a carbon copy (SpEL) **(String, default: `null`)**

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

from
: the primary recipient(s) of the email (SpEL) **(String, default: `null`)**

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

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

port
: the port of the mail server **(int, default: `25`)**

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

subject
: the email subject (SpEL) **(String, default: `null`)**

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

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

14.10 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" --deploy
```

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 **rabbit** sink has the following options:

addresses
: a comma separated list of *host[:port]* addresses **(String, default: `${spring.rabbitmq.addresses}`)**

converterClass
: the class name of the message converter **(String, default: `org.springframework.amqp.support.converter.SimpleMessageConverter`)**

deliveryMode
: the delivery mode (PERSISTENT, NON_PERSISTENT) **(String, default: `PERSISTENT`)**

exchange
: the Exchange on the RabbitMQ broker to which messages should be sent **(String, default: ``)**

mappedRequestHeaders
: request message header names to be propagated to/from the adpater/gateway **(String, default: `STANDARD_REQUEST_HEADERS`)**

password
: the password to use to connect to the broker **(String, default: `${spring.rabbitmq.password}`)**

routingKey
: the routing key to be passed with the message, as a SpEL expression **(String, default: `'<stream name>'`)**

sslPropertiesLocation
: resource containing SSL properties **(String, default: `${spring.rabbitmq.sslProperties}`)**

useSSL
: true if SSL should be used for the connection **(String, default: `${spring.rabbitmq.useSSL}`)**

username
: the username to use to connect to the broker **(String, default: `${spring.rabbitmq.username}`)**

vhost
: the RabbitMQ virtual host to use **(String, default: `${spring.rabbitmq.virtual_host}`)**

Note

Please be aware that the `routingKey` option is actually a SpEL expression. Hence if a simple, constant, string literal is to be used, make sure to use something like this:

```
xd:> stream create rabbitSinkStream --definition "http | rabbit --
routingKey='\"myqueue\"'" --deploy
```

See the RabbitMQ MessageBus Documentation for more information about SSL configuration.

14.11 GemFire Server

Currently XD supports GemFire's client-server topology. A sink that writes data to a GemFire cache requires at least one cache server to be running in a separate process and may also be configured to use a Locator. While Gemfire configuration is outside of the scope of this document, details are covered in the GemFire Product documentation. The XD distribution includes a standalone GemFire server executable suitable for development and test purposes and bootstrapped using a Spring configuration file provided as a command line argument. The GemFire jar is distributed freely under GemFire's development license and is subject to the license's terms and conditions. Sink modules provided with the XD distrubution that write data to GemFire create a client cache and client region. No data is cached on the client.

Launching the XD GemFire Server

To start the GemFire cache server GemFire Server included in the Spring XD distribution, go to the XD install directory:

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

The command line argument is the path of a Spring Data Gemfire configuration file with including a configured cache server and one or more regions. A sample cache configuration is provided cq-demo.xml located in the `config` directory. Note that Spring interprets the path as a relative path unless it is explicitly preceded by `file:`. The sample configuration starts a server on port 40404 and creates a region named *Stocks*.

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. Both modules accept the same options.

The **gemfire-server** sink has the following options:

host
: host name of the cache server or locator (if useLocator=true) **(String, default: `localhost`)**

keyExpression
: a SpEL expression which is evaluated to create a cache key **(String, default: `'<stream name>'`)**

port
: port of the cache server or locator (if useLocator=true) **(int, default: `40404`)**

regionName
: name of the region to use when storing data **(String, default: `<stream name>`)**

useLocator
: indicates whether a locator is used to access the cache server **(boolean, default: `false`)**

Tip

The `keyExpression`, as its name suggests, is a SpEL. Typically, the key value is derived from the payload. The default of `'<streamname>'` (mind the quotes), will overwrite the same entry for every message received on the stream.

Note

The locator option is mostly intended for integration with an existing GemFire installation in which the cache servers are configured to use locators in accordance with best practice. While GemFire supports configuration of multiple locators for failover, this is currently not supported in XD. However, using a single virtual IP backed by hardware routers for failover has proven to be an effective and simpler alternative.

Example

Suppose we have a JSON document containing a stock price:

```
{"symbol":"FAKE", "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')" --deploy
```

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

This will write an entry to the GemFire *Stocks* region with the key *FAKE*. 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 FAKE
```

Note

In order to use locators you must add the `use-locator` to the active profile via server.yml or the spring.profiles_active environment variable.

14.12 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 **splunk** sink has the following options:

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

owner
: the owner of the tcpPort **(String, default: `admin`)**

password
: the password associated with the username **(String, default: `password`)**

port
: the TCP port number of the Splunk server **(int, default: `8089`)**

tcpPort
: the TCP port number to where XD will send the data **(int, default: `9500`)**

username
: the login name that has rights to send data to the tcpPort **(String, default: `admin`)**

How To 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" --deploy
```

14.13 MQTT Sink

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

Options

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

clientId
identifies the client **(String, default: `xd.mqtt.client.id.snk`)**

password
the password to use when connecting to the broker **(String, default: `guest`)**

qos
the quality of service to use **(int, default: `1`)**

retained
whether to set the *retained* flag **(boolean, default: `false`)**

topic
the topic to which the sink will publish **(String, default: `xd.mqtt.test`)**

url
location of the mqtt broker **(String, default: `tcp://localhost:1883`)**

username
the username to use when connecting to the broker **(String, default: `guest`)**

Note

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

14.14 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 "queue:foo > transform --expression=payload+'-foo' | log"
 --deploy
Created new stream 'f'

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

xd:>stream create r --definition "http | router --
expression=payload.contains('a')?'queue:foo':'queue:bar'" --deploy
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 sink.f:145 - a-foo
11:54:25,669  WARN ThreadPoolTaskScheduler-1 sink.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" --
deploy
Created new stream 'f'

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

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

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 sink.f:145 - a-foo
Groovy processing payload 'b'
11:34:09,797  WARN ThreadPoolTaskScheduler-1 sink.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

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

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

script
    reference to a script used to transform messages **(String, no default)**

Tip

If the `script` option is set, then the `expression` option will be ignored. As with other modules, 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'`

15. Taps

15.1 Introduction

A Tap allows you to "listen" to data while it is processed in an existing stream and process the data in a separate stream. The original stream is unaffected by the tap and isn't aware of its presence, similar to a phone wiretap. (WireTap is included in the standard catalog of EAI patterns and implemented in the Spring Integration EAI framework used by Spring XD).

Simply put, a Tap is a stream that uses a point in another stream as a source.

Example

The following XD shell commands create a stream `foo1` and a tap named `foo1tap`:

```
xd:> stream create --name foo1 --definition "time | log" --deploy
xd:> stream create --name foo1tap --definition "tap:stream:foo1 > log" --deploy
```

Since a tap is a type of stream, use the `stream create` command to create the tap. The tap source is specified using the named channel syntax and always begins with `tap:`. In this case, we are tapping the stream named `foo1` specified by `:stream:foo1`

Note

`stream:` is required in this case as it is possible to tap alternate XD targets such as jobs. This tap consumes data at the source of the target stream.

A tap can consume data from any point along the target stream's processing pipeline. XD provides a few ways to tap a stream after a given processor has been applied:

Example - tap after a processor has been applied

If the module name is unique in the target stream, use tap:stream:<stream_name>.<module_name>

If you have a stream called `mystream`, defined as

```
http | filter --expression=payload.startsWith('A') | transform --
expression=payload.toLowerCase() | file
```

Create a tap after the filter is applied using

```
tap:stream:mystream.filter > ....
```

Example - using a label

You may also use labels to create an alias for a module and reference the label in the tap

If you have a stream called `mystream`, defined as

```
http | transform --expression=payload.toLowerCase() | flibble: transform --
expression=payload.substring(3) | file
```

Create a tap after the second transformer is applied using

```
tap:stream:mystream.flibble > ....
```

A primary use case for a Tap 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.

Note

In cases where a multiple modules with the same module name, a label must be specified on the module to be tapped. For example if you want to tap the 2nd transform: `http | transform --expression=payload.toLowerCase() | tapMe: transform --expression=payload.substring(3) | file`

15.2 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.

16. Analytics

16.1 Introduction

Spring XD provides support for the real-time evaluation of various machine learning scoring algorithms as well simple real-time data analytics using various types of counters and gauges. The analytics functionality is provided via modules that can be added to a stream. In that sense, real-time analytics is accomplished via the same exact model as data-ingestion. It’s possible that the primary role of a stream is to perform real-time analytics, but it’s quite common to add a tap to initiate a secondary stream where analytics, e.g. a field-value-counter, is applied to the same data being ingested through a primary stream. You will see both approaches in the examples below.

16.2 Predictive analytics

Spring XD’s support for implementing predictive analytics by scoring analytical models that leverage machine learning algorithms begins with an extensible class library foundation upon which implementations can be built, such as the PMML Module that we describe here.

That module integrates with the JPMML-Evaluator library that provides support for a wide range of model types and is interoperable with models exported from R, Rattle, KNIME, and RapidMiner. For counter and gauge analytics, in-memory and Redis implementations are provided.

Incorporating the evaluation of machine learning algorithms into stream processing is as easy as using any other processing module. Here is a simple example

```
http --outputType=application/x-xd-tuple | analytic-pmml
     --location=/models/iris-flower-naive-bayes.pmml.xml
     --inputFieldMapping=
       'sepalLength:Sepal.Length,
        sepalWidth:Sepal.Width,
        petalLength:Petal.Length,
        petalWidth:Petal.Width'
     --outputFieldMapping='Predicted_Species:predictedSpecies' | log"
```

The `http` source converts posted data to a Tuple. The `analytic-pmml` processor loads the model from the specifed file and creates two mappings so that fields from the Tuple can be mapped into the input and output model names. The `log` sink writes the payload of the event message to the log file of the XD container.

Posting the following JSON data to the http source

```
{
  "sepalLength": "6.4",
  "sepalWidth":  "3.2",
  "petalLength": "4.5",
  "petalWidth":  "1.5"
}
```

will produce output in the log file as shown below.

```
{
   "id":"1722ec00-baad-11e3-b988-005056c00008",
   "timestamp":1396473833152,
   "sepalLength":"6.4",
   "sepalWidth":"3.2",
   "petalLength":"4.5",
   "petalWidth":"1.5",
   "predictedSpecies":"versicolor"
}
```

The next section on analytical models goes into more detail on the general infrastructure

16.3 Analytical Models

We provide some core abstractions for implementing analytical models in stream processing applications. The main interface for integrating analytical models is **Analytic**. Some analytical models need to adjust the domain input and the model output in some way, therefore we provide a special base class **MappedAnalytic** which has core abstractions for implementing that mapping via **InputMapper** and **OutputMapper**.

Since **Spring XD 1.0.0.M6** we support the integration of analytical models, also called statistical models or mining models, that are defined via PMML. **PMML** is the abbreviation for **Predictive Model Markup Language** and is a standard XML representation that allows specifications of different mining models, their ensembles, and associated preprocessing.

Note

PMML is maintained by the **Data Mining Group** (**DMG**) and supported by several state-of-the-art statistics and data mining software tools such as InfoSphere Warehouse, R / Rattle, SAS Enterprise Miner, SPSS®, and Weka. The current version of the **PMML** specification is 4.2 at the time of this writing. Applications can produce and consume **PMML** models, thus allowing an analytical model created in one application to be implemented and used for scoring or prediction in another.

PMML is just one of many other technologies that one can integrate to implement analytics with, more will follow in upcoming releases.

Modeling and Evaluation

Analytical models are usually defined by a statistician *aka* data scientist or quant by using some statistical tool to analyze the data and build an appropriate model. In order to implement those models in a business application they are usually transformed and exported in some way (*e.g.* in the form of a **PMML** definition). This model is then loaded into the application which then evaluates it against a given input (event, tuple, example).

Modeling

Analytical models can be defined in various ways. For the sake of brevity we use **R** from the r-project to demonstrate how easy it is to export an analytical model to **PMML** and use it later in stream processing.

For our example we use the iris example dataset in **R** to generate a classifier for iris flower species by applying the Naive Bayes algorithm.

```
library(e1071) # Load library with the naive bayes algorithm support.

library(pmml) # Load library with PMML export support.

data(iris) # Load the IRIS example dataset

#Helper function to split the given dataset into a dataset used for training (trainset)
 and (testset) used for evaulation.
splitDataFrame <- function(dataframe, seed = NULL, n = trainSize) {

   if (!is.null(seed)){
      set.seed(seed)
   }

   index <- 1:nrow(dataframe)
   trainindex <- sample(index, n)
   trainset <- dataframe[trainindex, ]
   testset <- dataframe[-trainindex, ]

   list(trainset = trainset, testset = testset)
}

#We want to use 95% of the IRIS data as training data and 5% as test data for evaluation.
datasets <- splitDataFrame(iris, seed = 1337, n= round(0.95 * nrow(iris)))

#Create a naive Bayes classifier to predict iris flower species (iris[,5]) from [,1:4] =
 Sepal.Length Sepal.Width Petal.Length Petal.Width
model <- naiveBayes(datasets$trainset[,1:4], datasets$trainset[,5])

#The name of the model and it's externalId could be used to uniquely identify this version
 of the model.
modelName = "iris-flower-classifier"
externalId = 42

#Convert the given model into a PMML model definition
pmmlDefinition = pmml.naiveBayes(model,model.name=paste(modelName,externalId,sep = ";"),
 predictedField='Species')

#Print the PMML definition to stdout
cat(toString(pmmlDefinition))
```

The r script above should produce the following **PMML** document that contains the abstract definition of the naive bayes classifier that we derived from the training dataset of the IRIS dataset.

```
<PMML version="4.1" xmlns="http://www.dmg.org/PMML-4_1" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://www.dmg.org/PMML-4_1 http://www.dmg.org/
v4-1/pmml-4-1.xsd">
<Header copyright="Copyright (c) 2014 tom" description="NaiveBayes Model">
 <Extension name="user" value="tom" extender="Rattle/PMML"/>
 <Application name="Rattle/PMML" version="1.4"/>
 <Timestamp>2014-04-02 13:22:15</Timestamp>
</Header>
<DataDictionary numberOfFields="6">
 <DataField name="Species" optype="categorical" dataType="string">
  <Value value="setosa"/>
  <Value value="versicolor"/>
  <Value value="virginica"/>
 </DataField>
 <DataField name="Sepal.Length" optype="continuous" dataType="double"/>
 <DataField name="Sepal.Width"  optype="continuous" dataType="double"/>
 <DataField name="Petal.Length" optype="continuous" dataType="double"/>
 <DataField name="Petal.Width"  optype="continuous" dataType="double"/>
 <DataField name="DiscretePlaceHolder" optype="categorical" dataType="string">
  <Value value="pseudoValue"/>
 </DataField>
</DataDictionary>
<NaiveBayesModel modelName="iris-flower-classifier;42"
                 functionName="classification" threshold="0.001">
 <MiningSchema>
  <MiningField name="Species"      usageType="predicted"/>
  <MiningField name="Sepal.Length" usageType="active"/>
  <MiningField name="Sepal.Width"  usageType="active"/>
  <MiningField name="Petal.Length" usageType="active"/>
  <MiningField name="Petal.Width"  usageType="active"/>
  <MiningField name="DiscretePlaceHolder" usageType="active"
               missingValueReplacement="pseudoValue"/>
 </MiningSchema>
 <Output>
  <OutputField name="Predicted_Species" feature="predictedValue"/>
  <OutputField name="Probability_setosa" optype="continuous"
               dataType="double" feature="probability" value="setosa"/>
  <OutputField name="Probability_versicolor" optype="continuous"
               dataType="double" feature="probability" value="versicolor"/>
  <OutputField name="Probability_virginica" optype="continuous"
               dataType="double" feature="probability" value="virginica"/>
 </Output>
 <BayesInputs>
  <Extension>
   <BayesInput fieldName="Sepal.Length">
    <TargetValueStats>
     <TargetValueStat value="setosa">
      <GaussianDistribution mean="5.006" variance="0.124248979591837"/>
     </TargetValueStat>
     <TargetValueStat value="versicolor">
      <GaussianDistribution mean="5.8953488372093" variance="0.283311184939092"/>
     </TargetValueStat>
     <TargetValueStat value="virginica">
      <GaussianDistribution mean="6.58163265306122" variance="0.410697278911565"/>
     </TargetValueStat>
    </TargetValueStats>
   </BayesInput>
  </Extension>
  <Extension>
   <BayesInput fieldName="Sepal.Width">
    <TargetValueStats>
     <TargetValueStat value="setosa">
...
```

```
...
      <GaussianDistribution mean="3.428" variance="0.143689795918367"/>
     </TargetValueStat>
     <TargetValueStat value="versicolor">
      <GaussianDistribution mean="2.76279069767442" variance="0.0966777408637874"/>
     </TargetValueStat>
     <TargetValueStat value="virginica">
      <GaussianDistribution mean="2.97142857142857" variance="0.105833333333333"/>
     </TargetValueStat>
    </TargetValueStats>
   </BayesInput>
  </Extension>
  <Extension>
   <BayesInput fieldName="Petal.Length">
    <TargetValueStats>
     <TargetValueStat value="setosa">
      <GaussianDistribution mean="1.462" variance="0.0301591836734694"/>
     </TargetValueStat>
     <TargetValueStat value="versicolor">
      <GaussianDistribution mean="4.21627906976744" variance="0.236633444075305"/>
     </TargetValueStat>
     <TargetValueStat value="virginica">
      <GaussianDistribution mean="5.55510204081633" variance="0.310442176870748"/>
     </TargetValueStat>
    </TargetValueStats>
   </BayesInput>
  </Extension>
  <Extension>
   <BayesInput fieldName="Petal.Width">
    <TargetValueStats>
     <TargetValueStat value="setosa">
      <GaussianDistribution mean="0.246" variance="0.0111061224489796"/>
     </TargetValueStat>
     <TargetValueStat value="versicolor">
      <GaussianDistribution mean="1.30697674418605" variance="0.042093023255814"/>
     </TargetValueStat>
     <TargetValueStat value="virginica">
      <GaussianDistribution mean="2.02448979591837" variance="0.0768877551020408"/>
     </TargetValueStat>
    </TargetValueStats>
   </BayesInput>
  </Extension>
  <BayesInput fieldName="DiscretePlaceHolder">
   <PairCounts value="pseudoValue">
    <TargetValueCounts>
     <TargetValueCount value="setosa" count="50"/>
     <TargetValueCount value="versicolor" count="43"/>
     <TargetValueCount value="virginica" count="49"/>
    </TargetValueCounts>
   </PairCounts>
  </BayesInput>
 </BayesInputs>
 <BayesOutput fieldName="Species">
  <TargetValueCounts>
   <TargetValueCount value="setosa" count="50"/>
   <TargetValueCount value="versicolor" count="43"/>
   <TargetValueCount value="virginica" count="49"/>
  </TargetValueCounts>
 </BayesOutput>
</NaiveBayesModel>
</PMML>
```

Evaluation

The above defined **PMML** model can be evaluated in a Spring XD stream definition by using the **analytic-pmml** module as a processor in your stream definition. The actual evaluation of the **PMML** is performed via the **PmmlAnalytic** which uses the **jpmml-evaluator** library.

Model Selection

The PMML standard allows multiple models to be defined within a single PMML document. The model to be used can be configured through the **modelName** option.

NOTE The PMML standard also supports other ways for selection models, *e.g.* based on a predicate. This is currently not supported.

In order to perform the evaluation in Spring XD you need to save the generated PMML document to some folder, typically the with the extension "pmml.xml". For this example we save the PMML document under the name **iris-flower-classification-naive-bayes-1.pmml.xml**.

In the following example we set up a stream definition with an `http` source that produces iris-flower-records that are piped to the `analytic-pmml` module which applies our iris flower classifier to predict the species of a given flower record. The result of that is a new record extended by a new attribute **predictedSpecies** which simply sent to a `log` sink.

The definition of the stream, which we call **iris-flower-classification**, looks as follows:

```
xd:>stream create --name iris-flower-classification
  --definition "http --outputType=application/x-xd-tuple | analytic-pmml
  --location=/models/iris-flower-classification-naive-bayes-1.pmml.xml
  --inputFieldMapping='sepalLength:Sepal.Length,
                       sepalWidth:Sepal.Width,
                       petalLength:Petal.Length,
                       petalWidth:Petal.Width'
  --outputFieldMapping='Predicted_Species:predictedSpecies' | log" --deploy
```

- The **location** parameter can be used to specify the exact location of the pmml document. The value must be a valid spring **resource** location
- The **inputFieldMapping** parameter defines a mapping of domain input fields to model input fields. It is just a list of fields or optional field:alias mappings to control which fields and how they are going to end up in the model-input. If no inputFieldMapping is defined then all domain input fields are used as model input.
- The **outputFieldMapping** parameter defines a mapping of model output fields to domain output fields with semantics analog to the inputFieldMapping.
- The optional **modelName** parameter of the analytic-pmml module can be used to refer to a particular named model within the PMML definition. If modelName is not defined the first model is selected by default.

NOTE Some analytical models like for instance **association rules** require a different typ of mapping. You can implement your own custom mapping strategies by implementing a custom **InputMapper** and **OutputMapper** and defining a new **PmmlAnalytic** or **TuplePmmlAnalytic** bean that uses your custom mappers.

After the stream has been successfully deployed to **Spring XD** we can eventually start to throw some data at it by issuing the following http request via the **XD-Shell** (or `curl`, or any other tool):

Note that our example record contains no information about which species the example belongs to - this will be added by our classifier.

```
xd:>http post --target http://localhost:9000 --contentType application/json --data
 "{ \"sepalLength\": 6.4, \"sepalWidth\": 3.2, \"petalLength\":4.5, \"petalWidth\":1.5 }"
```

After posting the above json document to the stream we should see the following output in the console:

```
   {
     "id":"1722ec00-baad-11e3-b988-005056c00008"
   , "timestamp":1396473833152
   , "sepalLength":"6.4"
   , "sepalWidth":"3.2"
   , "petalLength":"4.5"
   , "petalWidth":"1.5"
   , "predictedSpecies":"versicolor"
   }
```

NOTE the generated field **predictedSpecies** which now identifies our input as belonging to the iris species **versicolor**.

We verify that the generated **PMML** classifier produces the same result as **R** by executing the issuing the following commands in **rproject**:

```
datasets$testset[,1:4][1,]
# This is the first example record that we sent via the http post.
   Sepal.Length Sepal.Width Petal.Length Petal.Width
52          6.4         3.2          4.5         1.5

#Predict the class for the example record by using our naiveBayes model.
> predict(model, datasets$testset[,1:4][1,])
[1] versicolor
```

16.4 Counters and Gauges

Counter and Gauges are analytical data structures collectively referred to as metrics. 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.

The 1.0 release provides the following types of metrics

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

Spring XD supports these metrics and analytical data structures as a general purpose class library that works with several backend storage technologies. The 1.0 release provides in memory and Redis implementations.

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/" --deploy
```

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" --deploy
```

To retrieve the count:

```
xd:>counter display tweetcount
```

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

name
: the name of the metric to contribute to (will be created if necessary) **(String, default: `<stream name>`)**

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
```

The **field-value-counter** sink has the following options:

fieldName
: the name of the field for which values are counted **(String, no default)**

name
: the name of the metric to contribute to (will be created if necessary) **(String, default: `<stream name>`)**

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" --deploy
```

Now create a tap for a field value counter:

```
xd:> stream create --name fromUserCount --definition "tap:stream:springtweets > field-
value-counter --fieldName=user.screen_name" --deploy
```

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 *fromUserCount* in Redis. To view the counts:

```
From xd-shell,
  xd:> field-value-counter display fromUserCount
```

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"
 --deploy
```

you’d simply create a tap which pipes the input to `aggregate-counter`:

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

From the XD shell:

```
xd:> aggregate-counter display tweettap
```

Note: you can also use some criteria to filter out aggregate counter display values. Please refer to Shell documentation for aggregate counter for more details.

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

dateFormat
: a pattern (as in SimpleDateFormat) for parsing/formatting dates and timestamps **(String, default: `yyyy-MM-dd'T'HH:mm:ss.SSS'Z'`)**

name
: the name of the metric to contribute to (will be created if necessary) **(String, default: `<stream name>`)**

timeField
: name of a field in the message that contains the timestamp to contribute to **(String, default: `null`)**

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.

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

name
: the name of the metric to contribute to (will be created if necessary) **(String, default: `<stream name>`)**

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" --deploy
```

Next create the tap:

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

Now Post a message to the ingest stream:

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

Check the gauge:

```
xd:>gauge display --name simplegauge
```

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 *rich-gauge* 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 **rich-gauge** sink has the following options:

alpha

smoothing constant, or -1 to use arithmetic mean **(double, default: `-1.0`)**

name

the name of the metric to contribute to (will be created if necessary) **(String, default: `<stream name>`)**

Note

The smoothing factor behaves as an exponential moving average. The default value does no smoothing.

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" --deploy
```

Next create the tap:

```
xd:> stream create --name testgauge --definition "tap:stream:test > rich-gauge" --deploy
```

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:

```
xd:>rich-gauge display testgauge
```

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 transform module to extract the stock price from the payload:

```
xd:> stream create --name stockprice --definition "tap:stream:stocks > transform --
expression=#jsonPath(payload,'$.price') | rich-gauge"
```

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}
```

Note: JSON fields should be separated by a comma without any spaces. Alternatively, enclose the whole argument to `--data` with quotes and escape inner quotes with a backslash.

Check the gauge:

```
xd:>rich-gauge display stockprice
```

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 tap, using the transform module to extract the stock price from the payload:

```
xd:> stream create --name vmwprice --definition "tap:stream:stocks >
 filter --expression=#jsonPath(payload,'$.symbol')==VMW | transform --
expression=#jsonPath(payload,'$.price') | rich-gauge" --deploy
xd:> stream create --name emcprice --definition "tap:stream:stocks >
 filter --expression=#jsonPath(payload,'$.symbol')==EMC | transform --
expression=#jsonPath(payload,'$.price') | rich-gauge" --deploy
```

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:

```
xd:>rich-gauge display emcprice
xd:>rich-gauge display vmwprice
```

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" --deploy
xd:>stream create --name httptap --definition "tap:stream:httpStream > counter" --deploy
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":"rich-gauges",
      "href":"http://localhost:9393/metrics/rich-gauges"
    }
  ]
}
```

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
}
```

17. Tuples

17.1 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"));
```

Gradle Dependencies

If you wish to use Spring XD Tuples in you project add the following dependencies:

```
//Add this repo to your repositories if it does not already exist.
maven { url "http://repo.spring.io/libs-snapshot"}

//Add this dependency
compile 'org.springframework.xd:spring-xd-tuple:1.0 (wiki)'
```

18. Type Conversion

18.1 Introduction

Spring XD allows you to declaratively configure type conversion in stream definitions using the *inputType* and *outputType* module options. Note that general type conversion may also be accomplished easily within a transformer or a custom module. Currently, Spring XD natively supports the following type conversions commonly used in streams:

- **JSON** to/from **POJO**
- **JSON** to/from org.springframework.xd.tuple.Tuple
- **Object** to/from **byte[]** : Either the raw bytes serialized for remote transport, bytes emitted by a module, or converted to bytes using Java serialization(requires the object to be Serializable)
- **String** to/from **byte[]**
- **Object** to **plain text** (invokes the object's *toString()* method)

Where *JSON* represents either a byte array or String payload containing JSON. Currently, Objects may be converted from a JSON byte array or String. Converting to JSON always produces a String. Registration of custom type converters is covered in this section.

18.2 MIME types

inputType and *outputType* values are parsed as media types, e.g., `application/json` or `text/plain;charset=UTF-8`. MIME types are especially useful for indicating how to convert to String or byte[] content. Spring XD also uses MIME type format to represent Java types, using the general type `application/x-java-object` with a `type` parameter. For example, `application/x-java-object;type=java.util.Map` or `application/x-java-object;type=com.bar.Foo` . For convenience, you can use the class name by itself and Spring XD will translate a valid class name to the corresponding MIME type. In addition, Spring XD provides custom MIME types, notably, `application/x-xd-tuple` to specify a Tuple.

18.3 Stream Definition Examples

18.4 POJO to JSON

Type conversion will likely come up when implementing a custom module which produces or consumes a custom domain object. For example, you want to create a stream that integrates with a legacy system that includes custom domain types in its API. To process custom domain types directly minimally requires these types to be defined in Spring XD's class path. This approach will be cumbersome to maintain when the domain model changes. The recommended approach is to convert such types to JSON at the source, or back to POJO at the sink. You can do this by declaring the required conversions in the stream definition:

```
customPojoSource --outputType=application/json |p1 | p2 | ... | customPojoSink --
inputType=application/x-java-object;type=com.acme.MyDomainType
```

Note that the sink above does require the declared type to be in the module's classpath to perform the JSON to POJO conversion. Generally, POJO to JSON does not require the Java class. Once the payload is converted to JSON, Spring XD provided transformers and filters (p1, p2, etc.) can evaluate the payload contents using JsonPath functions in SpEL expressions. Alternately, you can convert the JSON to a Tuple, as shown in the following example.

JSON to Tuple

Sometimes it is convenient to convert JSON content to a Tuple in order to evaluate and access individual field values.

```
xd:> stream create tuple --definition "http | filter --inputType=application/
x-xd-tuple --expression=payload.hasFieldName('hello') | transform --
expression=payload.getString('hello').toUpperCase() |  log" --deploy
Created and deployed new stream 'tuple'
```

Note *inputType=application/x-xd-tuple* on the filter module will cause the payload to be converted to a Tuple at the filter's input channel. Thus, subsequent expressions are evaluated on a Tuple object. Here we invoke the Tuple methods `hasFieldName('hello')` on the filter and `getString('hello')` on the transformer. The output of the http source is expected to be JSON in this case. We set the *Content-Type* header to tell Spring XD that the payload is JSON.

```
xd:>http post --data {"hello":"world","foo":"bar"} --contentType application/json --target
 http://localhost:9000
> POST (application/json;charset=UTF-8) http://localhost:9000
 {"hello":"world","foo":"bar"}
> 200 OK
```

In the Spring XD console log, you should see something like:

```
13:19:45,054  INFO pool-42-thread-4 sink.tuple - WORLD
```

Java Serialization

The following serializes a java.io.Serializable object to a file. Presumably the *foo* module outputs a Serializable type. If not, this will result in an exception. If remote transport is configured, the output of *foo* will be marshalled using Spring XD's internal serialization. The object will be unmarshalled in the *file* module and then converted to a byte array using Java serialization.

```
foo  | --inputType=application/x-java-serialized-object file
```

18.5 MIME types and Java types

Internally, Spring XD implements type conversion using Spring Integration's data type channels. The data type channel converts payloads to the configured data type using Spring's MessageConverter.

Note

The use of *MessageCoverter* for data type channels was introduced in Spring Integration 4 to pass the Message to the converter method to allow it to access the Message's *content-type* header. This provides greater flexibility. For example, it is now possible to support multiple strategies for converting a String or byte array to a POJO, based on the content-type header.

When Spring XD deploys a module with a declared type conversion, it modifies the module's input and/or output channel definition to set the required Java type and registers MessageConverters associated with the target MIME type and Java type to the channel. The type conversions Spring XD provides out of the box are summarized in the following table:

Source Payload	Target Payload	content-type header	outputType/ inputType	Comments
POJO	JSON String	ignored	application/json	
Tuple	JSON String	ignored	application/json	JSON is tailored for Tuple
POJO	String (toString())	ignored	text/plain, java.lang.String	
POJO	byte[] (java.io serialized)	ignored	application/x-java-serialized-object	
JSON byte[] or String	POJO	application/json (or none)	application/x-java-object	
byte[] or String	Serializable	application/x-java-serialized-object	application/x-java-object	
JSON byte[] or String	Tuple	application/json (or none)	application/x-xd-tuple	
byte[]	String	any	text/plain, java.lang.String	will apply any Charset specified in the content-type header
String	byte[]	any	application/octet-stream	will apply any Charset specified in the content-type header

Caveats

Note that that inputType and outputType parameters only apply to payloads that require type conversion. For example, if a module produces an XML string with outputType=application/json, the payload will not be converted from XML to JSON. This is because the payload at the module's output channel is already a String so no conversion will be applied at runtime.

Part II. Developing Modules and Extensions

19. Creating a Source Module

19.1 Introduction

As outlined in the modules document, XD currently supports 4 types of modules: source, sink, and processor for stream processing and job for batch processing. 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.

19.2 Create the module Application Context file

Create the Inbound Channel Adapter in a file called *feed.xml*:

```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 id="xdFeed" 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  id="xdFeed" 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.

19.3 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.spring.io/libs-snapshot" }
  maven { url "http://repo.spring.io/plugins-release" }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'

ext {
    junitVersion = '4.11'
    springVersion = '4.0.3.RELEASE'
    springIntegrationVersion = '4.0.0.M4'
}

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 extends TestCase {

 @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!

19.4 Installing 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 by copying the following jars from your gradle cache to `${xd.home}/lib`:

```
spring-integration-feed-4.0.0.M4.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.

19.5 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" --deploy
```

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.

19.6 Installing the module in a sub-directory

Modules can also reside in a sub-directory named after the module itself, e.g. `modules/source/feed`. This sub-directory in turn contains `config` and `lib` sub-directories. If you install a module following this convention then the module will run in an isolated classpath, a child of the parent classpath created from the contents of `xd/lib`. See Modules with isolated classpath for more information.

[NOTE] To help you better package up a module for installation in a sub-directory the sample build file form the JPMML module can be used as a starting point.

20. Creating a Processor Module

20.1 Introduction

As outlined in the modules document, XD currently supports 4 types of modules: source, sink, and processor for stream processing and job for batch processing. This document walks through creation of a custom processor module.

One or more processors can be included in a stream definition to modify the data as it passes between the initial 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.

20.2 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);
    }
  }
}
```

20.3 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>
```

20.4 Install the Module

To deploy the module, you need to copy the *tweettransformer.xml* file to the `${xd.home}/modules/processors` directory. We also need to make the custom module code available. Spring XD looks for code in the jars it finds in the `${xd.home}/lib` directory. So create a jar with the `TweetTransformer` class in it (and the correct package structure) and drop it into `lib`.

20.5 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" --deploy
```

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.

20.6 Installing the module in a sub-directory

Modules can also reside in a sub-directory named after the module itself, e.g. `modules/sink/tweettransform`. This sub-directory in turn contains `config` and `lib` sub-directories. If you install a module following this convention then the module will run in an isolated classpath, a child of the parent classpath created from the contents of `xd/lib`. See Modules with isolated classpath for more information.

[NOTE] To help you better package up a module for installation in a sub-directory the sample build file form the JPMML module can be used as a starting point.

21. Creating a Sink Module

21.1 Introduction

As outlined in the modules document, XD currently supports 4 types of modules: source, sink, and processor for stream processing and job for batch procesing. 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.

21.2 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.

21.3 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.

21.4 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.spring.io/libs-snapshot" }
  maven { url "http://repo.spring.io/plugins-release" }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'

ext {
    junitVersion = '4.11'
    lettuceVersion = '2.3.3'
    springVersion = '4.0.3.RELEASE'
    springIntegrationVersion = '4.0.0.M4'
    springDataRedisVersion = '1.1.1.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!

21.5 Install 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.

21.6 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" --deploy
```

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\",...\"}"
```

21.7 Installing the module in a sub-directory

Modules can also reside in a sub-directory named after the module itself, e.g. `modules/sink/redis`. This sub-directory in turn contains `config` and `lib` sub-directories. If you install a module following this convention then the module will run in an isolated classpath, a child of the parent classpath created from the contents of `xd/lib`. See Modules with isolated classpath for more information.

[NOTE] To help you better package up a module for installation in a sub-directory the sample build file form the JPMML module can be used as a starting point.

22. Creating a Job Module

22.1 Introduction

As outlined in the modules document, XD currently supports 4 types of modules: source, sink, and processor for stream processing and job for batch procesing. This document walks through creation of a custom job module.

22.2 Developing your Job

The Job definitions provided as part of the Spring XD distribution as well as those included in the Spring XD Samples repository can be used a basis for building your own custom Jobs. The development of a Job largely follows the development of a Spring Batch job, for which there are several references.

- Spring Batch home page
- Spring Batch In Action - Manning
- Pro Spring Batch - APress

For help developing Job steps specific to Hadoop, e.g. HDFS, Pig, Hive, the Spring XD Samples is useful as well as the following resources

- Spring for Apache Hadoop home page
- Spring Data - O'Reilly - Chapter 13

Once your Jobs have been developed and unit tested, they are integrated into Spring XD by copying the resulting .jar file and Job XML definition to $XD_HOME/lib and $XD_HOME/modules/jobs.

22.3 Creating a Simple Job

To describe the creation of a job we will use the job definition that is part of the batch-simple example.

To create a job in the XD shell, execute the 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.

The job defintion file *myjob.xml* is directory $XD_HOME/modules/jobs/myjob/myjob.xml and looks like:

```
<?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: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/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>
```

A *Tasklet* is the extension point to handle processing in a batch processing step. In this example there is only one step and it simply prints out the job parameters.

```
xd:> job create --name helloSpringXD --definition "myjob" --deploy
xd:> job launch helloSpringXD --params {"myStringParameter":"foobar","-
secondParam(long)":"123456"}
```

Note: by default, deploy is set to *false*. "--deploy" or "--deploy true" will deploy the job along with job creation.

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

```
Hello Spring XD!
The following 3 Job Parameter(s) is/are present:
Parameter name: secondParam; isIdentifying: false; type: LONG; value: 123456
Parameter name: myStringParameter; isIdentifying: true; type: STRING; value: foobar
Parameter name: random; isIdentifying: true; type: STRING; value: 0.06893349621991496
```

22.4 Creating a read-write processing Job

Often a batch job will involve reading batches of data from a source, tranforming or processing that data and then wrting the batch of data to a destination. This flow is encapsulated in the *ChunkOrientedTasklet* that is represented in the job configuration using the `<chunk/>` element that has `reader`, `writer` and optional `processor` elements. Other attributes define the size of the chunck and various policies for handling failure cases.

You will usually be able to reuse existing reader and writer implementations. The filejdbc job provided in the distribution shows an example of this using the standard File reader and JDBC writer.

The processor is based on the ItemProcessor interface. It has a generic signature that lets you operate on a record at at time. The batch of records is handled as a collection in reader and writer implementations. In the `filejdbc` job, the reader converts input records into a Spring XD Tuple. The tuple serves as a generic data structure but you can also use or write another converter to convert the input record to your own custom POJO object.

22.5 Orchestrating Hadoop Jobs

There are several tasklet implementation that will run various types of Hadoop Jobs

- MapReduce Job
- HDFS Scripts
- Hive Scripts
- Pig Scripts

The Spring Hadoop Samples project provides examples of how to create batch jobs that orchestate various hadoop jobs at each step. You can also mix and match steps related to work that is executed on the Hadoop cluster and work that is executed on the Spring XD cluster.

22.6 Installing the module in a sub-directory

Modules can also reside in a sub-directory named after the module itself, e.g. `modules/job/myjob`. This sub-directory in turn contains `config` and `lib` sub-directories. If you install a module following this convention then the module will run in an isolated classpath, a child of the parent classpath created from the contents of `xd/lib`. See Modules with isolated classpath for more information.

[NOTE] To help you better package up a module for installation in a sub-directory the sample build file form the JPMML module can be used as a starting point.

23. Providing Module Options Metadata

23.1 Introduction

Each available module can expose metadata about the options it accepts. This is useful to enhance the user experience, and is the foundation to advanced features like contextual help and code completion. the For example, provided that the file source module has been enriched with options metadata (and it has), one can use the `module info` command in the shell to get information about the module:

```
xd:> module info source:file
Information about source module 'file':

  Option Name        Description
        Default  Type
  -----------------
 ---------------------------------------------------------------------------  -------
 ---------
  dir                the absolute path to the directory to monitor for files
        <none>   String
  pattern            a filter expression (Ant style) to accept only files that match the
 pattern  *        String
  outputType         how this module should emit messages it produces
        <none>   MediaType
  preventDuplicates  whether to prevent the same file from being processed twice
        true     boolean
  ref                set to true to output the File object itself
        false    boolean
  fixedDelay         the fixed delay polling interval specified in seconds
        5        int
```

For this to be available, module authors have to provide a little bit of extra information, known as "Module Options Metadata". That metadata can take two forms, depending on the needs of the module: one can either use the "simple" approach, or the "POJO" approach. If one does not need advanced features like profile activation, validation or options encapsulation, then the "simple" approach is sufficient.

23.2 Using the "Simple" approach

To use the simple approach, simply create a file named `<module>.properties` right next to the `<module>.xml` file for your module.

Declaring and documenting an option

In that file, each option `<option>` is declared by adding a line of the form

```
options.<option>.description = the description
```

The description for the option is the only required part, and is a very important piece of information for the end user, so pay special attention to it (see also Style remarks)

That sole line in the properties file makes a `--<option>=` construct available in the definition of a stream using your module.

About plugin provided options metadata

Some options are automatically added to a module, depending on its type. For example, every source module automatically inherits a `outputType` option, that controls the type conversion feature between modules. You don’t have to do anything for that to happen.

Similarly, every job module benefits from a handful of job specific options.

Here is a recap of those automatically provided options:

Module Type	Options
Source	outputType
Processor	outputType, inputType
Sink	inputType
Job	makeUnique, numberFormat, dateFormat

Advertising default values

In addition to this, one can also provide a default value for the option, using

```
options.<option>.default = SomeDefault
```

Doing this, the default value should **not** be used in the placeholder syntax in the xml file. Assuming this is the contents of `foo.properties`:

```
options.bar.description = a very useful option
options.bar.default = 5
```

then in `foo.xml`:

```
<!-- this is correct -->
<feature the-bar="${bar}"" />

<!-- this is incorrect/not needed -->
<feature the-bar="${bar:5}" />
```

Exposing the option type

Lastly, one can document the option type using a construct like

```
options.<option>.type = fully.qualified.class.Name
```

For simple "primitive" types, one can use short names, like so:

```
options.<option>.type = String
or
options.<option>.type = boolean
or
options.<option>.type = Integer
```

Note that there is support for both wrapper types (*e.g.* Integer) and primitive types (*e.g.* int). Although this is used for documentation purposes only, the primitive type would typically be used to indicate a **required** option (`null` being prohibited).

23.3 Using the "POJO" approach

To use advanced features such as profile activation driven by the values provided by the end user, one would need to leverage the "POJO" approach.

Instead of writing a properties file, you will need to write a custom java class that will hold the values at runtime. That class is also introspected to derive metadata about your module.

Declaring options to the module

For the simplest cases, the class you need to write does not need to implement or inherit from anything. The only thing you need to do is to reference it in a properties file named after your module (the same file location you would have used had you been leveraging the "simple" approach):

```
options_class = fully.qualified.name.of.your.Pojo
```

Note that the key is `options_class`, with an *s* and an underscore (not to be confused with `option.<optionname>` that is used in the "simple" approach)

For each option you want available using the `--<option>=` syntax, you must write a public setter annotated with `@ModuleOption`, providing the option description in the annotation.

The type accepted by that setter will be used as the documented type.

That setter will typically be used to store the value in a private field. How the module application can get ahold of the value is the topic of the next section.

Exposing values to the context

For a provided value to be used in the module definition (using the `${foo}` syntax), your POJO class needs to expose a `getFoo()` getter.

At runtime, an instance of the POJO class will be created (it requires a no-arg constructor, by the way) and values given by the user will be bound (using setters). The POJO class thus acts as an intermediate `PropertySource` to provide values to `${foo}` constructs.

Providing defaults

To provide default values, one would most certainly simply store a default value in the backing field of a getter/setter pair. That value (actually, the result of invoking the matching getter to a setter on a newly instanciated object) is what is advertised as the default.

Encapsulating options

Although one would typically use the combination of a `foo` field and a `getFoo(), setFoo(x)` pair, one does not have to.

In particular, if your module definition requires some "complex" (all things being relative here) value to be computed from "simpler" ones (*e.g.* a *suffix* value would be computed from an *extension* option, that would take care of adding a dot, depending on whether it is blank or not), then you'd simply do the following:

```
public class MyOptions {
 private String extension;

 @ModuleOption("the file extension to use")
 public void setExtension(String extension) {
     this.extension = extension;
 }

 public String getSuffix() {
     return extension == null ? null : "." + extension;
 }
}
```

This would expose a `--extension=` option, being surfaced as a `${suffix}` placeholder construct.

The astute reader will have realized that the default can not be computed then, because there is no `getExtension()` (and there should not be, as this could be mistakenly used in `${extension}`). To provide the default value, you should use the `defaultValue` attribute of the `@ModuleOption` annotation.

Using profiles

The real benefit of using a POJO class for options metadata comes with advanced features though, one of which is dynamic profile activation.

If the set of beans (or xml namespaced elements) you would define in the module definition file depends on the value that the user provided for one or several options, then you can make your POJO class implement `ProfileNamesProvider`. That interface brings one contract method, `profilesToActivate()` that you must implement, returning the names of the profiles you want to use (this method is invoked **after** user option values have been bound, so you can use any logic involving those to compute the list of profile names).

As an example of this feature, see *e.g.* `TriggerSourceOptionsMetadata`.

Using validation

Your POJO class can optionally bear JSR303 annotations. If it does, then validation will occur after values have been successfully bound (understand that injection can fail early due to type incoherence by the way. This comes for free and does not require JSR303 annotations).

This can be used to validate a set of options passed in (some are often mutually exclusive) or to catch misconfiguration earlier than deployment time (*e.g.* a port number cannot be negative).

23.4 Metadata style remarks

To provide a uniform user experience, it is better if your options metadata information adheres to the following style:

- option names should follow the `camelCase` syntax, as this is easier with the POJO approach. If we later decide to switch to a more `unix-style`, this will be taken care of by XD itself, with no change to the metadata artifacts described here

- description sentences should be concise
- descriptions should start with a **lowercase** letter and should **not** end with a dot
- use primitive types for required numbers
- descriptions should mention the unit for numbers (*e.g* ms)
- descriptions should **not** describe the default value, to the best extent possible (this is surfaced thru the actual *default* metadata awareness)
- options metadata should know about the default, rather than relying on the `${foo:default}` construct

24. Extending Spring XD

24.1 Introduction

This document describes how to customize or extend the Spring XD Container. Spring XD is a distributed runtime platform delivered as executable components including XD Admin, XD Container, and XD Shell. The XD Container is a Spring application combining XML resources, Java @Configuration classes, and Spring Boot auto configuration for its internal configuration, initialized via the Spring Boot SpringApplicationBuilder. Since Spring XD is open source, the curious user can see exactly how it is configured. However, all Spring XD's configuration is bundled in jar files and therefore not directly accessible to end users. Most users do not need to customize or extend the XD Container. For those that do, Spring XD provides hooks to:

- Provide additional bean definitions
- Override selected bean definitions with custom implementations

Customization scenarios might include:

- Add a new data transport
- Add a Spring XD plugin to configure modules
- Embed a shared component used by user provided Plugin, such as a GemFire cache or a data source
- Providing additional type converters

This following sections provide an overview of XD Container internals and explain how to extend Spring XD for each of these scenarios. The reader is expected to have a working knowledge of both the Spring Framework and Spring Integration.

24.2 Spring XD Application Contexts

The diagram below shows how Spring XD is organized into several Spring application contexts. Some understanding of the Spring XD application context hierarchy is necessary for extending XD. In the diagram, solid arrows indicate a parent-child relationship. As with any Spring application a child application context may reference beans defined in its parent application context, but the parent context cannot access beans defined in the child context. It is important to keep in mind that a bean definition registered in a child context with the same id as a bean in the parent context will create a separate instance in the child context. Similarly, any bean definition will override an earlier bean definition in the same application context registered with the same id (Sometimes referred to as "last one wins").

Spring XD's primary extension mechanism targets the *Plugin Context* highlighted in the diagram. Using a separate convention, it is also possible to register an alternate MessageBus implementation in the *Shared Server Context*.

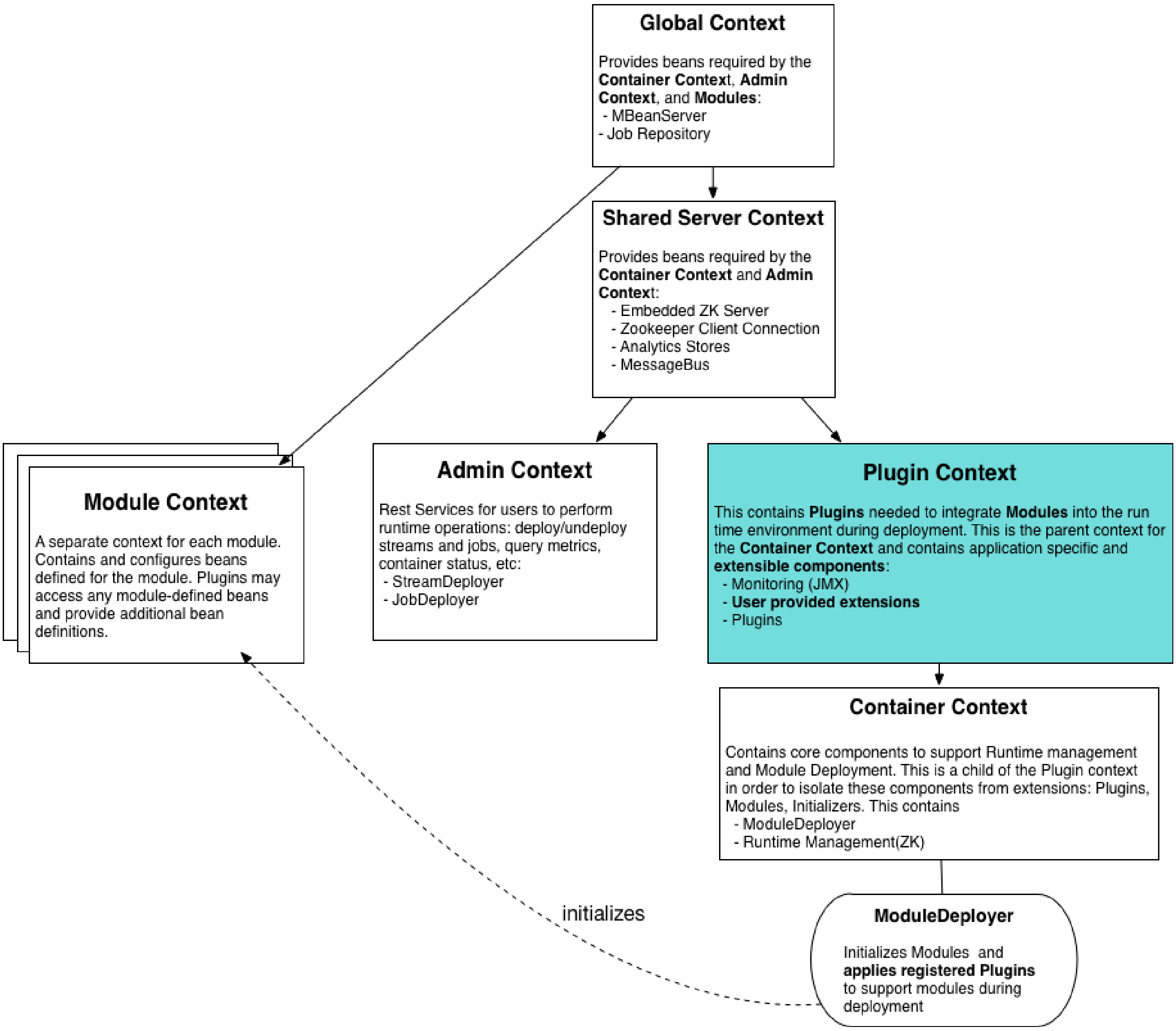

Global Context
Provides beans required by the Container Context, Admin Context, and Modules:
- MBeanServer
- Job Repository
Shared Server Context
Provides beans required by the Container Context and Admin Context:
- Embedded ZK Server
- Zookeeper Client Connection
- Analytics Stores
- MessageBus
Module Context
A separate context for each module. Contains and configures beans defined for the module. Plugins may access any module-defined beans and provide additional bean definitions.
Admin Context
Rest Services for users to perform runtime operations: deploy/undeploy streams and jobs, query metrics, container status, etc:
- StreamDeployer
- JobDeployer
Plugin Context
This contains Plugins needed to integrate Modules into the run time environment during deployment. This is the parent context for the Container Context and contains application specific and extensible components:
- Monitoring (JMX)
- User provided extensions
- Plugins
Container Context
Contains core components to support Runtime management and Module Deployment. This is a child of the Plugin context in order to isolate these components from extensions: Plugins, Modules, Initializers. This contains
- ModuleDeployer
- Runtime Management(ZK)
initializes
ModuleDeployer
Initializes Modules and applies registered Plugins to support modules during deployment

Figure 24.1. The Spring XD Application Context Hierarchy

While this arrangement of application contexts is more complex than the typical Spring application, XD is designed this way for the following reasons:

- **Bean isolation** - Some beans are "global" in that they are shared by all XD runtime components: Admin, Container, and Modules. Those allocated to the *Shared Server Context* are shared only by Admin and Container. Some beans must be available to Plugins ,used to configure *Modules*. However Plugins and Modules should be isolated from critical internal components. While complete isolation has proven difficult to achieve, the intention is to minimize any undesirable side effects when introducing extensions.
- **Bean scoping** - To ensure that single node and distributed configurations of the Spring XD runtime are logically equivalent, the Spring configuration is identical in both cases, avoiding unnecessary duplication of bean definitions.
- **Lifecycle management** - Plugins and other beans used to configure these application contexts are also Spring beans which Spring XD dynamically "discovers" during initialization. Such components must be fully instantiated prior to the creation of the application context to which they are targeted.

To ensure initialization happens in the desired order, such beans may be either defined in an isolated application context (i.e., not part of the hierarchy) or in a parent context which Spring initializes before any of its descendants.

24.3 Plugin Architecture

The XD Container at its core is simply a runtime environment for hosting and managing micro Spring applications called *Modules*. Each module runs in its own application context (*Module Context*). The Module Context is a child of *Global Context*, as modules share some bean definitions, but otherwise is logically isolated from beans defined in the XD Container. The *Module Context* is fundamental to the Spring XD design. In fact, this is what allows each module to define its own *input* and *output* channels, and in general, enables beans be uniquely configured via property placeholders evaluated for each deployed instance of a Module. The http://docs.spring.io/spring-xd/docs/1.0 (wiki)/api/org/springframework/xd/module/core/Module.html[Module] interface and its default implementation provide a thin wrapper around a Spring Application Context for which properties are bound, profiles activated, and beans added or enhanced in order to "plug" the module into the XD Container.

The *ModuleDeployer*, shown in the diagram, is a core component of the Container Context, responsible for initializing modules during deployment, and shutting them down during undeployment. The ModuleDeployer sees the module as a "black box", unaware of its purpose or runtime requirements. Binding a module's channels to XD's data transport, for instance, is the responsibility of the http://docs.spring.io/spring-xd/docs/1.0 (wiki)/api/org/springframework/integration/x/bus/MessageBus.html[MessageBus] implementation configured for the transport. The MessageBus binding methods are actually invoked by the *StreamPlugin* during the initialization of a stream module. To support jobs, XD provides a *JobPlugin* to wire the Spring Batch components defined in the module during deployment. The JobPlugin also invokes the MessageBus to support communications between XD and job modules. These, and other functions critical to Spring XD are performed by classes that implement the http://docs.spring.io/spring-xd/docs/1.0 (wiki)/api/org/springframework/xd/module/core/Plugin.html[Plugin] interface. A Plugin operates on every deployed Module which it is implemented to support. Thus the ModuleDeployer simply invokes the deployment life cycle methods provided by every Plugin registered in the Plugin Context.

The ModuleDeployer discovers registered Plugins by calling `getBeansOfType(Plugin.class)` for the *Plugin Context* (its parent context). This means that adding your own Plugin requires these steps:

- Implement the http://docs.spring.io/spring-xd/docs/1.0 (wiki)/api/org/springframework/xd/module/core/Plugin.html[Plugin] interface
- Add your Plugin implementation and any dependent classes to Spring XD's class path
- Follow conventions that Spring XD uses to register Plugins

The next section covers these steps in more detail.

24.4 How to Add a Spring bean to the XD Container

This section applies to adding a *Plugin*, which is generally useful since a Plugin has access to every module as it is being deployed (see the previous section on Plugin Architecture). Furthermore, this section describes a generic mechanism for adding any bean definition to the *Plugin Context*. Spring XD uses both Spring Framework's class path component scanning and resource resolution to find any components that you add to specified locations in the class path. This means you may provide Java

@Configuration and/or any classes annotated with the @Component stereotype in a configured base package in addition to bean definitions defined in any XML or Groovy resource placed under a configured resource location. These locations are given by the properties `xd.extensions.locations` and `xd.extensions.basepackages`, optionally configured in `servers.yml` down at the bottom:

```
# User Extensions: Where XD scans the class path to discover extended container
 configuration to adds beans to the Plugins context.
# Each property may be a comma delimited string. 'basepackages' refers to package names
 used for
# annotated component (@Configuration or @Component stereotypes) scanning. 'locations' is
 a list of root resource directories containing XML or Groovy configuration.
# XD prepends classpath:* if no prefix included and appends **/*.* to each location
#xd:
#  extensions:
#      basepackages: com.acme.xd.extensions
#      locations: META-INF/spring-xd/ext
```

As the pluralization of these property names suggests, you may represent multiple values as a comma delimited string. Also note that there is no default for `xd.extensions.basepackages`. So if you want to use annotation based configuration, you must first set up one or more base package locations. The resource location(s) define the root locations where any XML or Groovy Spring bean definition file found in the given root or any of its subdirectories will be loaded. The root location defaults to `META-INF/spring-xd/ext`

The Container loads any bean definitions found in these configured locations on the class path and adds them to the Plugin Context. This is the appropriate application context since in order to apply custom logic to modules, you will most likely need to provide a custom Plugin.

Note

The extension mechanism is very flexible. In theory, one can define BeanPostProcessors, BeanFactoryPostProcessors, or ApplicationListeners to manipulate Spring XD application contexts. Do so at your own risk as the Spring XD initialization process is fairly complex, and not all beans are intended to be extensible.

Extensions are packaged in a jar file which must be added to Spring XD's class path. Currently, you must manually copy the jar to `$XD_HOME/lib` for each container instance. To implement a Plugin, you will need to include a compile time dependency on `spring-xd-module` in your build. To access other container classes and to test your code in a container you will also require `spring-xd-dirt`.

24.5 Providing A new Type Converter

Spring XD supports automatic type conversion to convert payloads declaratively. For example, to convert an object to JSON, you provide the module option `--outputType=application/json` to a module used in a stream definition. The conversion is enabled by a *Plugin* that binds a Spring MessageConverter to a media type. The default type converters are currently configured in streams.xml, packaged in `spring-xd-dirt-<version>.jar`. If you look at that file, you can see an empty list registered as `customMessageConverters`.

```
<!-- Users can override this to add converters.-->
 <util:list id="customMessageConverters"/>
```

So registering new type converters is a matter of registering an alternate list as `customMessageConverters` to the application context. Spring XD will replace the default empty list with yours. `xd.messageConverters` and `customMessageConverters` are two lists injected into the *ModuleTypeConversionPlugin* to build an instance of CompositeMessageConverter which delegates to the first converter in list order that is able to perform the necessary conversion. The Plugin injects the CompositeMessageConverter into the module’s *input* or *output* the MessageChannel, corresponding to the `inputType` or `outputType` options declared for any module in the stream definition (or defined as the module’s default `inputType`).

The CompositeMessageConverter is desirable because a module does not generally know what payload type it will get from its predecessor. For example, the converters that Spring XD provides out of the box can convert any Java object, including a http://docs.spring.io/spring-xd/docs/1.0 (wiki)/api/org/springframework/xd/tuple/Tuple.html[Tuple] and a byte array to a JSON String. However the methods for converting a byte array or a Tuple are each optimized for the respective type. The CompositeMessageConverter for --outputType=application/json must provide all three methods and the Data Type channel chooses the first converter that applies to both the incoming payload type and the media type (e.g., application/json). Note that the order that the converters appear in the list is significant. In general, converters for specific payload types precede more general converters for the same media type. The `customMessageConverters` are added after the standard converters in the order defined. So it is generally easier to add converters for new media types than to replace existing converters.

For example, a member of the Spring XD community inquired about Spring XD’s support for Google protocol buffers. This user was interested in integrating Spring XD with an existing messaging system that uses GPB heavily and needed a way to convert incoming and outgoing GPB payloads to interoperate with XD streams. This could be accomplished by providing a `customMessageConverters` bean containing a list of required message converters. Writing a custom converter to work with XD requires extending http://docs.spring.io/spring-xd/docs/1.0 (wiki)/api/org/springframework/integration/x/bus/converter/AbstractFromMessageConverter.html[AbstractFromMessageConverter] provided by `spring-xd-dirt`. It is recommended to review the existing implementations listed in streams.xml to get a feel for how to do this. In addition, you would likely define a custom MimeType such as `application/gpb`.

Note

It is worth mentioning that GPB is commonly used for marshaling objects over the network. In the context of Spring XD marshaling is treated as a separate concern from payload conversion. In Spring XD, marshaling happens at the "pipe" indicated by the | symbol using a different serialization mechanism, described below. In this case, the GPB payloads are produced and consumed by systems external to Spring XD and need to be converted in order that a GPB payload can work with XD streams. In this scenario, if the GPB is represented as a byte array, the bytes are transmitted over the network directly and marshaling is unnecessary.

As an illustration, suppose this user has developed a source module that emits GPB payloads from a legacy service. Spring XD provides transform and filter modules that accept SpEL expressions to perform their respective tasks. These modules are useful in many situations but the SpEL expressions generally require a POJO representing a domain type, or a JSON string. In this case it would be convenient to support stream definitions such as

```
gpb-source --outputType=application/x-java-object | transform --expression=...
```

where *gpb-source* represents a custom module that emits a GPB payload and *expression* references some specific object property. The media type application/x-java-object is a convention used by XD to indicate that the payload should be converted to a Java type embedded in the serialized representation (GPB in this example). Alternately, converting to JSON could be performed if the stream definition were:

```
gpb-source --outputType=application/json | transform --expression=...
```

To convert an XD stream result to GPB to be consumed by an external service might look like:

```
source | P1 ... | Pn | gpb-sink --inputType=application/gpb
```

These examples would require registering custom MessageConverters to handle the indicated conversions. Alternately, this may be accomplished by writing custom processor modules to perform the required conversion. The above examples would then have stream definitions that look more like:

```
gpb-source | gpb-to-pojo | transform --expression=...
source | P1 ... | Pn | json-to-gpb | gpb-sink
```

Tip

While custom processor modules are easier to implement, they add unnecessary complexity to stream definitions that use them. If such conversions are required everywhere, enabling automatic conversion may be worth the effort. Also, note that using a separate module generally requires additional network hops (at each pipe). If a processor module is necessary only to perform a common payload conversion, it is more efficient to install a custom converter.

24.6 Adding a New Data Transport

Spring XD offers Redis and Rabbit MQ for data transport out of the box. Transport is configured simply by setting the property `xd.transport` to `redis` or `rabbit`. In addition xd-singlenode supports a `--transport` command line option that can accept *local*(the single node default) in addition. This simple configuration mechanism is supported internally by an import declaration that binds the transport implementation to a name.

```
<import resource="classpath*:/META-INF/spring-xd/transports/${XD_TRANSPORT}-bus.xml"/>
```

The above snippet is from an internal Spring configuration file loaded into the *Shared Server Context*. Spring XD provides MessageBus implementations in `META-INF/spring-xd/transports/redis-bus.xml` and `META-INF/spring-xd/transports/rabbit-bus.xml`

This makes it relatively simple for Spring XD developers and advanced users to provide alternate MessageBus implementations to enable a new transport and activate that transport by setting the `xd.transport` property. For example, to implement a JMS MessageBus you would add a jar containing `/META-INF/spring-xd/transports/jms-bus.xml` in the class path. This file must register a bean of type *MessageBus* with the ID `messageBus`. A jar providing the above configuration file along with the MessageBus implementation and any dependencies must be installed `$XD_HOME/lib`.

When implementing a MessageBus, it is advisable to review and understand the existing implementations which extend http://docs.spring.io/spring-xd/docs/1.0 (wiki)/api/org/springframework/integration/x/bus/MessageBusSupport.html[MessageBusSupport]. This base class performs some common tasks including payload marshaling. Spring XD uses the term *codec* to connote a component that performs both serialization and deserialization and provides a bean with the same name. In the example above, the JMS MessageBus configuration`/META-INF/spring-xd/transports/jms-bus.xml` might look something like:

```
<bean id="messageBus" class="my.example.JmsMessageBus">
 <constructor-arg ref="jmsConnectionFactory" />
 <constructor-arg ref="codec"/>
</bean>
```

where JmsMessageBus extends MessageBusSupport and the developer is responsible for configuring any dependent JMS resources appropriately.

25. Samples

We have a number of sample projects in the Spring XD Samples GitHub repository. Below are some additional examples for ingesting syslog data to HDFS.

25.1 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)

If you're using syslog over TCP and need the highest throughput, use the Reactor-backed syslog module.

```
xd:> stream create --definition "reactor-syslog --port=<tcp-port> | hdfs" --name <stream-
name>
```

The `reactor-syslog` module doesn't yet support UDP (though it soon will), so if you're using syslog over UDP you'll want to use the standard syslog module.

```
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 syslog-ng configuration (for example: /etc/syslog-ng/syslog-ng.conf):

1) Add destination

```
destination <destinationName> {
      tcp("<host>" port("<tcp-port>"));
};
```

or,

```
destination <destinationName> {
      udp("<host>" port("<udp-port>"));
};
```

where "host" is the container(launcher) host where the syslog module is deployed.

2) Add log rule to log message sources:

```
log {
  source(<message_source>); destination(<destinationName>);
};
```

3) Make sure to restart the service after the change:

```
sudo service syslog-ng restart
```

Now, the syslog messages from the syslog message sources are written into HDFS /xd/<stream-name>/

Part III. Configuration Guidelines

26. Overview

When running a distributed Spring XD runtime, there are a number of considerations related to performance and reliability. In most cases, these involve settings that have tradeoffs, but in this section we provide some background so you know what the options are and how to configure them.

In the Deployment section that follows, we provide detailed information about various properties that can be passed along with the `stream deploy` command. That section also describes a scenario that is common for minimizing network hops, where direct binding can occur between modules rather than having each pipe within a stream correspond to a send and receive over the Message Bus. For more detail see the Direct Binding subsection.

Another relevant topic for minimizing network hops is the ability to *compose* modules. That is a useful technique where a subset of the stream's contiguous modules can be grouped together as if a single module. All of the pipes within the composed module will rely upon a local transport rather than sending and receiving via the Message Bus. For more detail read the Composing Modules section.

For production use, high availability will typically be a requirement for the data transport. Therefore, we recommend using Spring XD's RabbitMQ-based Message Bus. In the Message Bus Configuration section below, we provide details on the relevant HA configuration settings as well as other reliability settings, security settings (including enabling SSL), and error-handling capabilities.

When configuring a RabbitMQ Message Bus, you will also want to consider several performance settings. For example, unless strict sequential ordering is required, the *prefetch* and *concurrency* values should be overridden (the default for each is 1). That can lead to a significant performance improvement. In the less likely case that performance concerns completely outweigh reliability, you can disable acknowledgements and even disable the persistence of messages. For a listing of these settings and more, refer to the RabbitMQ Configuration section. Several performance related configuration settings exist on the broker itself, and those are well-documented in the RabbitMQ Admin Guide. For example, the *vm_memory_high_watermark* and *vm_memory_high_watermark_paging_ratio* are both explained within the Flow Control subsection of the guide.

If you are using the HTTP source module in a stream and want to scale, you can deploy multiple instances by specifying the *module.http.count* property as described in the Deployment Properties section. Keep in mind that each instance will share the same port value. The default is 9000, but that can be overridden, for all instances, by including *--port* as an option for the HTTP module in the stream definition. That means you would want to ensure that each container that may be a candidate for deploying one of the HTTP module instances (taking into account the *criteria* deployment property if provided), is running on a different host, either physically or on separate virtual machines. Of course, in a production environment, you would likely want to add a load balancer in front of those HTTP endpoints.

Also when using the HTTP source module, you may want to consider enabling support for HTTPS. An example is provided in the documentation for that module's options.

27. Deployment

27.1 Introduction

This section covers topics related to deployment, including:

- The Deployment Manifest
- Deployment States
- Container Attributes
- Stream Partitioning
- Direct Binding

When you deploy a Stream or Job, the Spring XD Runtime performs the following steps:

- parse the stream or job definition (DSL Guide) to resolve each Module reference along with its options
- set option values assigned to each component Module
- parse and store the deployment request including the Deployment Manifest
- allocate each Module to an available Container instance in accordance with the Deployment Manifest
- binding Module channel(s), either to the MessageBus or directly using Direct Binding
- track the state of each deployed module
- track the overall stream or job's Deployment State

27.2 Deployment Manifest

A stream is composed of modules. Each module is deployed to one or more Container instance(s). In this way, stream processing is distributed among multiple containers. By default, deploying a stream to a distributed runtime configuration uses simple round robin logic. For example if there are three containers and three modules in a stream definition, `s1= m1 | m2 | m3`, then Spring XD will attempt to distribute the work load evenly among each container. This is a very simplistic strategy and does not take into account things like:

- server load - how many modules are already deployed to a container? What is the current memory and CPU utilization?
- server affinity - some containers may have external software installed and specific modules will benefit from co-location. For example, an hdfs sink might be deployed only to hosts running Hadoop. Or perhaps a file sink should be deployed to hosts configured with extra disk space.
- scalability - Suppose the stream s1, above, can achieve higher throughput with multiple instances of m2 running, so we want to deploy m2 to every available container.
- fault tolerance - the ability to target physical servers on redundant networks, routers, racks, etc.

Generally, more complex deployment strategies are needed to tune and operate XD. Additionally, we must consider various features and constraints when deploying to a PaaS, Yarn or some other cluster manager. Additionally, Spring XD allows supports Stream Partitioning and Direct Binding.

To address such deployment concerns, Spring XD provides a *Deployment Manifest* which is submitted with the deployment request, in the form of in-line deployment properties (or potentially a reference to a separate document containing deployment properties).

Deployment Properties

When you execute the `stream deploy` shell command, you can optionally provide a `properties` parameter which is a comma delimited list of key=value pairs. Examples for the key include **module.[modulename].count** and **module.[modulename].criteria** (for a full list of properties, see below). The value for the count is a positive integer, and the value for criteria is a valid SpEL expression. The Spring XD runtime matches an available container for each module according to the deployment manifest.

The deployment properties allow you to specify deployment instructions for each module. Currently this includes:

- The number of module instances
- A target server or server group
- MessageBus attributes required for a specific module
- Stream Partitioning
- Direct Binding

General Properties

Note

You can apply criteria to all modules in the stream by using the wildcard * for [modulename]

module.[modulename].count
The number of module instances (see above).

module.[modulename].criteria
A boolean SpEL expression using the Container Attributes as an evaluation context.

Example:

```
xd:>stream deploy --name test1 --properties
 "module.transform.count=3,module.log.criteria=groups.contains('group1')"
```

Bus Properties

Common Bus Properties

Note

The following properties are only allowed when using a *RabbitMessageBus* or a *RedisMessageBus*; the *LocalMessageBus* does not support properties.

module.[modulename].consumer.backOffInitialInterval
The number of milliseconds to wait for the first delivery retry **(default 1000)**

module.[modulename].consumer.backOffMaxInterval
: The maximum number of milliseconds to wait between retries **(default 10000)**

module.[modulename].consumer.backOffMultiplier
: The previous retry interval is multiplied by this to determine the current interval (but see *backOffMaxInterval*) **(default 2.0)**

module.[modulename].consumer.concurrency
: The number of concurrent consumers for the module **(default 1)**.

module.[modulename].consumer.maxAttempts
: The maximum number of attempts to make a delivery when a failure occurs **(default 3)**

RabbitMQ Bus Properties

Note

The following properties are only allowed when using a *RabbitMessageBus*.

See the Spring AMQP reference documentation for information about the RabbitMQ-specific attributes.

module.[modulename].consumer.ackMode
: Controls message acknowledgements **(default AUTO)**

module.[modulename].consumer.maxConcurrency
: The maximum number of concurrent consumers for the module **(default 1)**.

module.[modulename].consumer.prefetch
: The number of messages prefetched from the RabbitMQ broker **(default 1)**

module.[modulename].consumer.prefix
: A prefix applied to all queues/exchanges that are declared by the bus - allows policies to be applied **(default xdbus.)**

module.[modulename].consumer.requestHeaderPatterns
: Controls which message headers are passed between modules **(default STANDARD_REQUEST_HEADERS,*)**

module.[modulename].consumer.replyHeaderPatterns
: Controls which message headers are passed between modules (only used in partitioned jobs) **(default STANDARD_REPLY_HEADERS,*)**

module.[modulename].consumer.requeue
: Whether messages will be requeued (and retried) on failure **(default true)**

module.[modulename].consumer.transacted
: Whether consumers use transacted channels **(default false)**

module.[modulename].consumer.txSize
: The number of delivered messages between acknowledgements (when *ackMode=AUTO*) **(default 1)**

module.[modulename].producer.deliveryMode
: THe delivery mode of messages sent to RabbitMQ (*PERSISTENT* or *NON_PERSISTENT*) **(default PERSISTENT)**

module.[modulename].producer.requestHeaderPatterns
: Controls which message headers are passed between modules **(default STANDARD_REQUEST_HEADERS,*)**

module.[modulename].producer.replyHeaderPatterns
: Controls which message headers are passed between modules (only used in partitioned jobs) **(default STANDARD_REPLY_HEADERS,*)**

Stream Partitioning

Note

Partitioning is only allowed when using a *RabbitMessageBus* or a *RedisMessageBus*.

A common pattern in stream processing is to partition the data as it is streamed. This entails deploying multiple instances of a message consuming module and using content-based routing so that messages containing the identical data value(s) are always routed to the same module instance. You can use the Deployment Manifest to declaratively configure a partitioning strategy to route each message to a specific consumer instance.

Partition Properties

See below for examples of deploying partitioned streams.

module.[modulename].producer.partitionKeyExtractorClass
: The class name of a *PartitionKeyExtractorStrategy* **(default null)**

module.[modulename].producer.partitionKeyExpression
: A *SpEL* expression, evaluated against the message, to determine the partition key; only applies if *partitionKeyExtractorClass* is null. If both are null, the module is not partitioned **(default null)**

module.[modulename].producer.partitionSelectorClass
: The class name of a *PartitionSelectorStrategy* **(default null)**

module.[modulename].producer.partitionSelectorExpression
: A *SpEL* expression, evaluated against the partition key, to determine the partition index to which the message will be routed. The final partition index will be the return value (an integer) modulo *[nextModule].count* If both the class and expression are null, the bus's default *PartitionSelectorStrategy* will be applied to the key **(default null)**

In summary, a module is partitioned if its *count* is > 1 and the previous module has a *partitionKeyExtractorClass* or *partitionKeyExpression* (class takes precedence). When a partition key is extracted, the partitioned module instance is determined by invoking the *partitionSelectorClass*, if present, or the *partitionSelectorExpression % count*. If neither is present the result is *key.hashCode() % count*.

Direct Binding

Sometimes it is desirable to allow co-located, contiguous modules to communicate directly, rather than using the configured remote transport, to eliminate network latency. Spring XD creates direct bindings by default only in cases where every "pair" of producer and consumer (modules bound on either side of a pipe) are guaranteed to be co-located.

Currently Spring XD implements no conditional logic to force modules to be co-located. The only way to guarantee that every producer-consumer pair is co-located is to specify that the pair be deployed to every available container instance, in other words, the module counts must be 0. The figure below illustrates this concept. In the first hypothetical case, we deploy one instance (the default)of producer m1, and two instances of the consumer m2. In this case, enabling direct binding would isolate one of the consumer instances. Spring XD will not create direct bindings in this case. The second case guarantees co-location of the pairs and will result in direct binding.

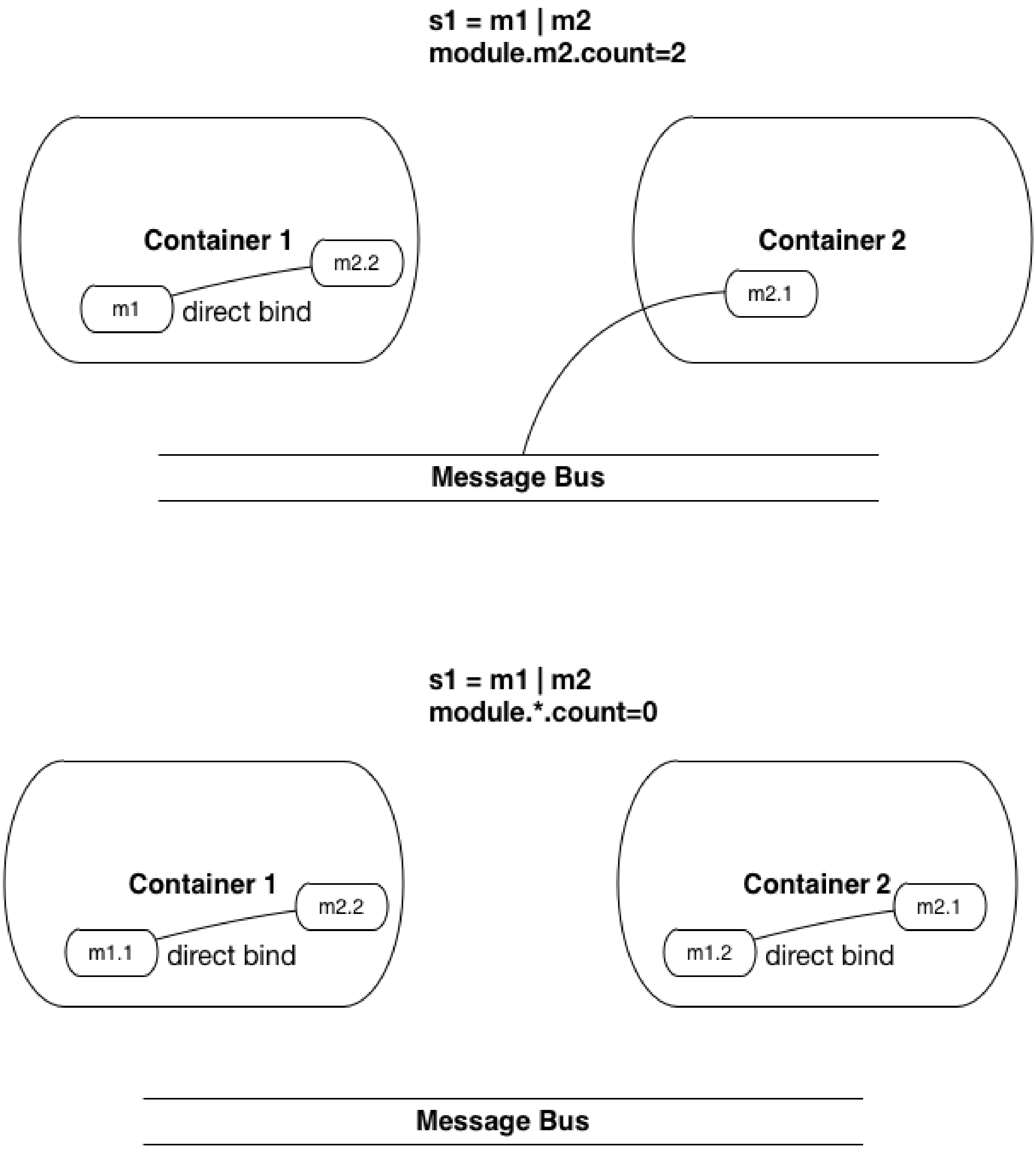


Figure 27.1.

In addition, direct binding requires that the producer is not configured for partitioning since partitioning is **implemented** by the Message Bus.

Using *module.*.count=0* is the most straightforward way to enable direct binding. Direct binding may be disabled for the stream using *module.*.producer.directBindingAllowed=false*. Additional direct binding deployment examples are shown below.

27.3 Deployment States

The ability to specify criteria to match container instances and deploy multiple instances for each module leads to one of several possible deployment states for the stream as a whole. Consider a stream in an initial *undeployed* state.

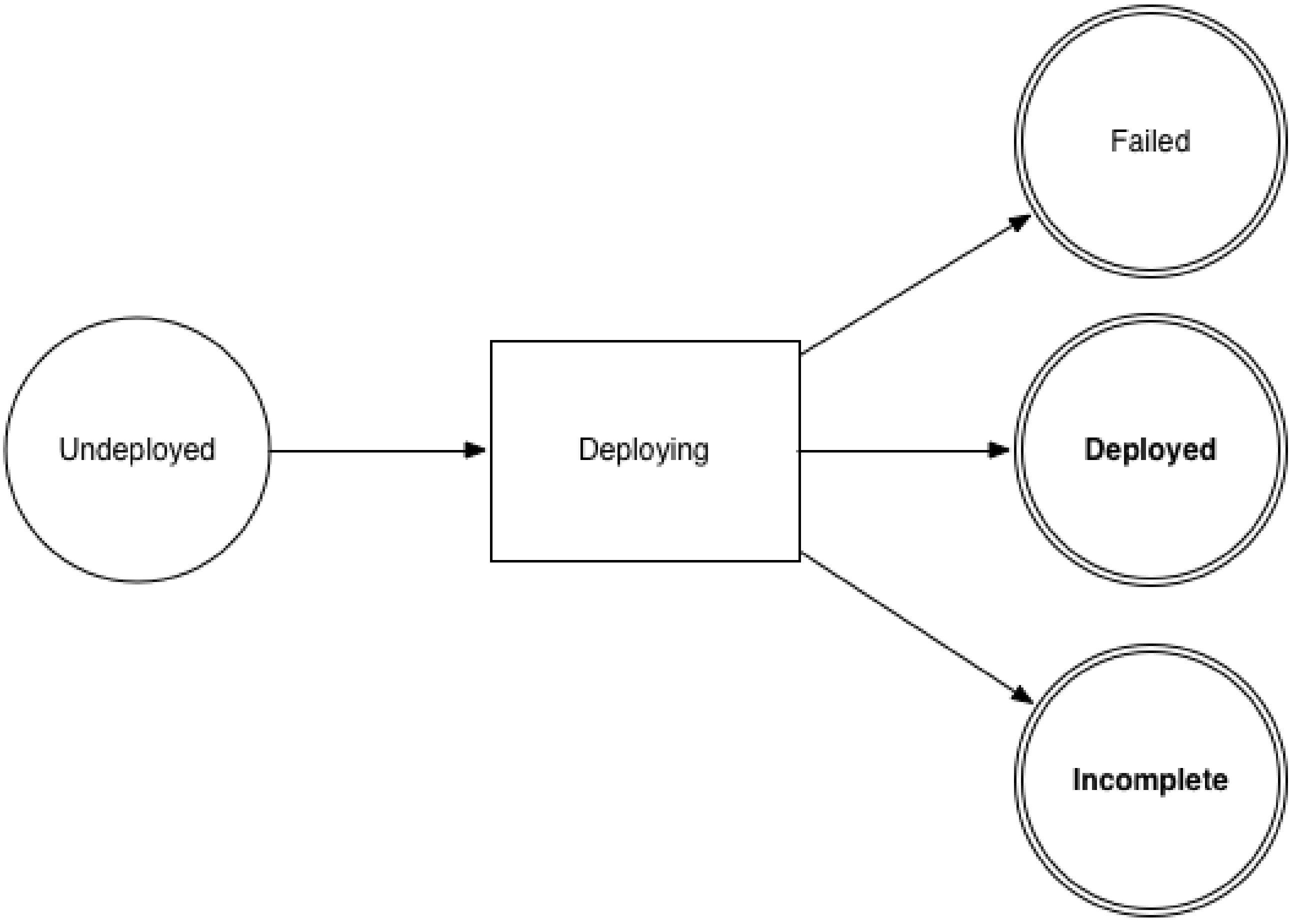


Figure 27.2.

After executing the stream deployment request, the stream will be one of the following states:

- **Deployed** - All modules deployed successfully as specified in the deployment manifest.
- **Incomplete** - One of the requested module instances could not be deployed, but at least one instance of each module definition was successfully deployed. The stream is operational and can process messages end-to-end but the deployment manifest was not completely satisfied.
- **Failed** - At least one of the module definitions was not deployed. The stream is not operational.

Note

The state diagram above represents these states as final. This is an over-simplification since these states are affected by container arrivals and departures that occur during or after the execution of a deployment request. Such transitions have been omitted intentionally but are worth considering. Also, there is an analogous state machine for undeploying a stream, initially in any of these states, which is left as an exercise for the reader.

Example

```
xd:>stream create test1 --definition "http | transform --expression=payload.toUpperCase()
 | log"
Created new stream 'test1'
```

Next, deploy it requesting three transformer instances:

```
xd:>stream deploy --name test1 --properties "module.transform.count=3"
Deployed stream 'test1'

xd:>stream list
  Stream Name  Stream Definition                                          Status
  -----------  ---------------------------------------------------------  ----------
  test1        http | transform --expression=payload.toUpperCase() | log  incomplete
```

If there are only two container instances available, only two instances of *transform* will be deployed. The stream deployment state is *incomplete* and the stream is functional. However the unfulfilled deployment request remains active and the third instance will be deployed if a new container comes on line that matches the criteria.

27.4 Container Attributes

The SpEL context (root object) for module.[modulename].criteria is ContainerAtrtributes, basically a map derivative that contains some standard attributes:

- **id** - the generated container ID
- **pid** - the process ID of the container instance
- **host** - the host name of the machine running the container instance
- **ip** — the IP address of the machine running the container instance

ContainerAttributes also includes any user-defined attribute values configured for the container. These attributes are configured by editing *xd/config/servers.yml* the file included in the XD distribution contains some commented out sections as examples. In this case, the container attributes configuration looks something like:

```
xd:
  container:
      groups: group2
      color: red
```

Note

Groups may also be assigned to a container via the optional command line argument *--groups* or by setting the environment variable *XD_CONTAINER_GROUPS*. As the property name suggests, a container may belong to more than one group, represented as comma-delimited string. The concept of server groups is considered an especially useful convention for targeting groups of servers for deployment to support many common scenarios, so it enjoys special status. Internally, *groups* is simply a user defined attribute.

27.5 Stream Deployment Examples

To Illustrate how to use the Deployment Manifest, We will use a runtime configuration with 3 container instances, as displayed in the XD shell:

```
xd:>runtime containers
  Container Id                          Host              IP Address     PID   Groups
 Custom Attributes
  ------------------------------------  ----------------  -------------  ----  ------
 -----------------
  bc624816-f8a8-4f35-83f6-a125ed147b7c  ip-10-110-18-10   10.110.18.10   1708  group2
 {color=red}
  018b7c8d-6fa9-4759-8471-76899766f892  ip-10-139-36-168  10.139.36.168  1852  group2
 {color=blue}
  afc3741c-217a-415a-9d86-a1f62de03613  ip-10-139-17-116  10.139.17.116  1861  group1
 {color=green}
```

Each of the three containers is running on a different host and has configured Groups and Custom Attributes as shown.

First, create a stream:

```
xd:>stream create test1 --definition "http | transform --expression=payload.toUpperCase()
 | log"
Created new stream 'test1'
```

Next, deploy it using a manifest:

```
xd:>stream deploy --name test1 --properties
 "module.transform.count=3,module.log.criteria=groups.contains('group1')"
Deployed stream 'test1'
```

Verify the deployment:

```
xd:>runtime modules
  Module                       Container Id                          Options
                           Deployment Properties
  ---------------------------  ------------------------------------
  ----------------------------------------------
 ---------------------------------------------------------
  test1.processor.transform.1  bc624816-f8a8-4f35-83f6-a125ed147b7c  {valid=true,
 expression=payload.toUpperCase()}  {count=3, sequence=1}
  test1.processor.transform.2  018b7c8d-6fa9-4759-8471-76899766f892  {valid=true,
 expression=payload.toUpperCase()}  {count=3, sequence=2}
  test1.processor.transform.3  afc3741c-217a-415a-9d86-a1f62de03613  {valid=true,
 expression=payload.toUpperCase()}  {count=3, sequence=3}
  test1.sink.log.1             afc3741c-217a-415a-9d86-a1f62de03613
 {name=test1, expression=payload, level=INFO}    {count=1, sequence=1,
 criteria=groups.contains('group1')}
  test1.source.http.1          bc624816-f8a8-4f35-83f6-a125ed147b7c  {port=9000}
                           {count=1, sequence=1}
```

We can see that three instances of the *transform* processor have been deployed, one to each container instance. Also the log module has been deployed to the container assigned to *group1*. Now we can undeploy and deploy the stream using a different manifest:

```
xd:>stream undeploy test1
Un-deployed stream 'test1'
xd:>runtime modules
  Module  Container Id  Properties
  ------  ------------  ----------

xd:>stream deploy --name test1 --properties "module.log.count=3,module.log.criteria=!
groups.contains('group1')"
Deployed stream 'test1'

xd:>stream list
  Stream Name  Stream Definition                                          Status
  -----------  ---------------------------------------------------------  ----------
  test1        http | transform --expression=payload.toUpperCase() | log  incomplete

xd:>runtime modules
  Module                       Container Id                          Options
                           Deployment Properties
  ---------------------------  ------------------------------------
  ----------------------------------------------
 ----------------------------------------------------------
  test1.processor.transform.1  018b7c8d-6fa9-4759-8471-76899766f892  {valid=true,
 expression=payload.toUpperCase()}  {count=1, sequence=1}
  test1.sink.log.1             bc624816-f8a8-4f35-83f6-a125ed147b7c
 {name=test1, expression=payload, level=INFO}    {count=3, sequence=1, criteria=!
groups.contains('group1')}
  test1.sink.log.2             018b7c8d-6fa9-4759-8471-76899766f892
 {name=test1, expression=payload, level=INFO}    {count=3, sequence=2, criteria=!
groups.contains('group1')}
  test1.source.http.1          afc3741c-217a-415a-9d86-a1f62de03613  {port=9000}
                           {count=1, sequence=1}
```

Now there are only two instances of the *log* module deployed. We asked for three however the deployment criteria specifies only containers not in *group1* are eligible. The *log* module is deployed only to the two containers matching the criteria. The deployment status of stream *test1* is shown as *incomplete*. The stream is functional even though the deployment manifest is not completely satisfied. If we fire up a new container not in *group1*, the DeploymentSupervisor will handle any outstanding deployment requests by comparing *xd/deployments/modules/requested* to *xd/deployments/modules/allocated*, and will deploy the third *log* instance and update the stream state to *deployed*.

27.6 Partitioned Stream Deployment Examples

Using SpEL Expressions

First, create a stream:

```
xd:>stream create --name partitioned --definition "jms | transform --
expression=#expensiveTransformation(payload) | log"

Created new stream 'partitioned'
```

The hypothetical SpEL function *expensiveTransformation* represents a resource intensive processor which we want to load balance by running on multiple containers. In this case, we also want to partition the stream so that payloads containing the same *customerId* are always routed to the same processor instance. Perhaps the processor aggregates data by customerId and this step needs to run using co-located resources.

Next, deploy it using a manifest:

```
xd:>stream deploy --name partitioned --properties
 "module.jms.producer.partitionKeyExpression=payload.customerId,module.transform.count=3"

Deployed stream 'partitioned'
```

In this example three instances of the transformer will be created (with partition index of 0, 1, and 2). When the jms module sends a message it will take the *customerId* property on the message payload, invoke its *hashCode()* method and apply the modulo function with the divisor being the *transform.count* property to determine which instance of the transform will process the message (**payload.getCustomerId().hashCode() % 3**). Messages with the same *customerId* will always be processed by the same instance.

27.7 Direct Binding Deployment Examples

In the simplest case, we enforce direct binding by setting the instance count to 0 for all modules in the stream. A count of 0 means deploy the module to all available containers:

```
xd:>runtime containers
  Container Id                          Host            IP Address    PID    Groups
 Custom Attributes
  ------------------------------------  --------------  ------------  -----  ------
 -----------------
  8e814924-15de-4ca1-82d3-ddfe851668ab  ultrafox.local  192.168.1.18  81532
  a2b89274-2d40-46e4-afc5-4988bea28a16  ultrafox.local  192.168.1.9   4605   group1
```

We start with two container instances. One belongs to the group *group1*.

```
xd:>stream create direct --definition "time | log"
Created new stream 'direct'
xd:>stream deploy direct --properties module.*.count=0
Deployed stream 'direct'
xd:>runtime modules
  Module                Container Id                          Options
                   Deployment Properties
  --------------------  ------------------------------------
 ---------------------------------------------
 ---------------------------------------------------------
  direct.sink.log.0     a2b89274-2d40-46e4-afc5-4988bea28a16  {name=direct,
 expression=payload, level=INFO}  {count=0, sequence=0}
  direct.sink.log.0     8e814924-15de-4ca1-82d3-ddfe851668ab  {name=direct,
 expression=payload, level=INFO}  {count=0, sequence=0}
  direct.source.time.0  a2b89274-2d40-46e4-afc5-4988bea28a16  {fixedDelay=1, format=yyyy-
MM-dd HH:mm:ss}     {producer.directBindingAllowed=true, count=0, sequence=0}
  direct.source.time.0  8e814924-15de-4ca1-82d3-ddfe851668ab  {fixedDelay=1, format=yyyy-
MM-dd HH:mm:ss}     {producer.directBindingAllowed=true, count=0, sequence=0}
```

Note that we have two containers and two instances of each module deployed to each. Spring XD automatically sets the bus properties needed to allow direct binding, *producer.directBindingAllowed=true* on the *time* module.

Suppose we only want one instance of this stream and we want it to use direct binding. Here we can add deployment criteria to restrict the available containers to *group1*.

```
xd:>stream undeploy direct
Un-deployed stream 'direct'
xd:>stream deploy direct --properties "module.*.count=0,
 module.*.criteria=groups.contains('group1')"
Deployed stream 'direct'
xd:>runtime modules
  Module                Container Id                          Options
                   Deployment Properties
  --------------------  ------------------------------------
 ---------------------------------------------
 ---------------------------------------------------------------------------------------------
  direct.sink.log.0     a2b89274-2d40-46e4-afc5-4988bea28a16
 {name=direct, expression=payload, level=INFO}  {count=0, sequence=0,
 criteria=groups.contains('group1')}
  direct.source.time.0  a2b89274-2d40-46e4-afc5-4988bea28a16  {fixedDelay=1, format=yyyy-
MM-dd HH:mm:ss}     {producer.directBindingAllowed=true, count=0, sequence=0,
 criteria=groups.contains('group1')}
```

Direct binding eliminates latency between modules but sacrifices some of the resiliency provided by the messaging middleware. In the scenario above, if we lose one of the containers, we lose messages. To disable direct binding when module counts are set to 0, set *module.*.producer.directBindingAllowed=false*.

```
xd:>stream undeploy direct
Un-deployed stream 'direct'
xd:>stream deploy direct --properties "module.*.count=0,
 module.*.producer.directBindingAllowed=false"
Deployed stream 'direct'
xd:>runtime modules
  Module                Container Id                          Options
                   Deployment Properties
  --------------------  ------------------------------------
 ---------------------------------------------
 ----------------------------------------------------------
  direct.sink.log.0     a2b89274-2d40-46e4-afc5-4988bea28a16  {name=direct,
 expression=payload, level=INFO}  {producer.directBindingAllowed=false, count=0,
 sequence=0}
  direct.sink.log.0     8e814924-15de-4ca1-82d3-ddfe851668ab  {name=direct,
 expression=payload, level=INFO}  {producer.directBindingAllowed=false, count=0,
 sequence=0}
  direct.source.time.0  a2b89274-2d40-46e4-afc5-4988bea28a16  {fixedDelay=1, format=yyyy-
MM-dd HH:mm:ss}     {producer.directBindingAllowed=false, count=0, sequence=0}
  direct.source.time.0  8e814924-15de-4ca1-82d3-ddfe851668ab  {fixedDelay=1, format=yyyy-
MM-dd HH:mm:ss}     {producer.directBindingAllowed=false, count=0, sequence=0}
```

Finally, we can still have the best of both worlds by enabling guaranteed delivery at one point in the stream, usually the source. If the tail of the stream is co-located and the source uses the message bus, the message bus may be configured so that if a container instance goes down, any unacknowledged messages will be retried until the container comes back or its modules are redeployed.

TDB: A realistic example

An alternate scenario with similar characteristics would be if the stream uses a *rabbit* or *jms* source. In this case, guaranteed delivery would be configured in the external messaging system instead of the Spring XD transport.

28. Message Bus Configuration

28.1 Introduction

This section contains additional information about configuring the Message Bus, including High Availability, SSL, and Error handling.

28.2 Rabbit Message Bus High Availability (HA) Configuration

The `RabbitMessageBus` allows for HA configuration using normal RabbitMQ HA Configuration.

First, use the `addresses` property in `servers.yml` to include the host/port for each server in the cluster. See Application Configuration.

By default, queues and exchanges declared by the bus are prefixed with `xdbus.` (this prefix can be changed as described in Application Configuration).

To configure the entire bus for HA, create a policy:

```
rabbitmqctl set_policy ha-xdbus "^xdbus\." '{"ha-mode":"all"}'
```

28.3 Error Handling (Message Delivery Failures)

RabbitMQ Message Bus

Note: the following applies to normally deployed streams. When direct binding between modules is being used, exceptions thrown by the consumer are thrown back to the producer.

When a consuming module (processor, sink) fails to handle a message, the bus will retry delivery based on the module (or default bus) retry configuration. The default configuration will make 3 attempts to deliver the message. The retry configuration can be modified at the bus level (in servers.yml), or for an individual stream/module using the deployment manifest.

When retries are exhausted, by default, messages are discarded. However, using RabbitMQ, you can configure such messages to be routed to a dead-letter exchange/dead letter queue. See the RabbitMQ Documentation for more information.

Consider a stream: `stream create foo --definition "source | processor | sink"`

The first *pipe* (by default) will be backed by a queue named `xdbus.foo.0`, the second by `xdbus.foo.1`. Messages are routed to these queues using the default exchange (with routing keys equal to the queue names).

To enable dead lettering just for this stream, first configure a policy:

```
rabbitmqctl set_policy foo.DLX "^xdbus\.foo\..*" '{"dead-letter-exchange":"foo.dlx"}' --apply-to queues
```

To configure dead-lettering for all streams:

```
rabbitmqctl set_policy DLX "^xdbus\..*" '{"dead-letter-exchange":"dlx"}' --apply-to queues
```

The next step is to declare the dead letter exchange, and bind dead letter queues with the appropriate routing keys.

For example, for the second "pipe" in the stream above we might bind a queue `foo.sink.dlq` to exchange `foo.dlx` with a routing key `xdbus.foo.1` (remember, the original routing key was the queue name).

Now, when the sink fails to handle a message, after the configured retries are exhausted, the failed message will be routed to `foo.sink.dlq`.

There is no automated mechanism provided to move dead lettered messages back to the bus queue.

Redis Message Bus

When Redis is the transport, the failed messages (after retries are exhausted) are `LPUSH+ed to a +LIST ERRORS:<stream>.n` (e.g. `ERRORS:foo.1` in the above example in the *RabbitMQ Message Bus* section).

This is unconditional; the data in the `ERRORS LIST` is in "bus" format; again, some external mechanism would be needed to move the data from the ERRORS LIST back to the bus’s foo.1 LIST.

28.4 Rabbit Message Bus Secure Sockets Layer (SSL)

If you wish to use SSL for communications with the RabbitMQ server, consult the RabbitMQ SSL Support Documentation.

First configure the broker as described there. The message bus is a client of the broker and supports both of the described configurations for connecting clients (SSL *without certificate validation* and *with certficate validation*).

To use SSL without certificate validation, simply set

```
spring:
  rabbitmq:
    useSSL: true
```

In `application.yml` (and set the port(s) in the `addresses` property appropriately).

To use SSL with certificate validation, set

```
spring:
  rabbitmq:
    useSSL: true
    sslProperties: file:path/to/secret/ssl.properties
```

The `sslProperties` property is a Spring resource (`file:`, `classpath:` etc) that points to a properties file, Typically, this file would be secured by the operating system (and readable by the XD container) because it contains security information. Specifically:

```
keyStore=file:/secret/client/keycert.p12
trustStore=file:/secret/trustStore
keyStore.passPhrase=secret
trustStore.passPhrase=secret
```

Where the `pkcs12` keystore contains the client certificate and the truststore contains the server’s certificate as described in the rabbit documentation. The key/trust store properties are Spring resources.

Note

By default, the `rabbit` source and sink modules inherit their default configuration from the container, but it can be overridden, either using `modules.yml` or with specific module definitions.

Part IV. Administration

29. Monitoring and Management

Spring XD uses Spring Boot’s monitoring and management support over HTTP and JMX along with Spring Integration’s MBean Exporters

29.1 Monitoring XD Admin, Container and Single-node servers

Following are available by **default**

JMX is enabled `XD_JMX_ENABLED=true`

The spring boot management endpoints are exposed over `HTTP` and since `JMX` is enabled these endpoints are exposed over `JMX`

Spring integration components are exposed over `JMX` using `IntegrationMBeanExporter`

All the availble MBeans can be accessed over `HTTP` using `Jolokia`

To enable boot provided management endpoints over HTTP

By **default** `management.port` is set to use admin/container server port and all the exposed endpoints can be accessed from the root context of admin and container servers.

When starting admin, container or singlenode server, the command-line option `--mgmtPort` can be specified to use an explicit port for management server. With the given valid management port, the management endpoints can be accessed from that port. Please refer Spring Boot document here for more details on the endpoints.

For instance, once XD admin is started on localhost and the management port set to use the admin port (9393)

```
http://localhost:9393/management/health
http://localhost:9393/management/env
http://localhost:9393/management/beans

etc..
```

29.2 To disable boot endpoints over HTTP

Set `management.port=-1` for both default and container profiles in config/servers.yml

29.3 Management over JMX

All the boot endpoints are exposed over JMX with the domain name `org.springframework.boot` The MBeans that are exposed within XD admin, container server level are available with the domain names `xd.admin` (for XD admin), `xd.container` (for XD container), `xd.shared.server` and `xd.parent` representing the application contexts common to both XD admin and container. Singlenode server will have all these domain names exposed. When the stream/job gets deployed into the XD container, the stream/job MBeans are exposed with specific domain/object naming strategy.

29.4 To disable management over JMX

Set XD_JMX_ENABLED=false in config/servers.yml or set it as an environment variable to disable the management over JMX

29.5 Monitoring deployed modules in XD container

When a module is deployed (with JMX is enabled on the XD container), the **IntegrationMBeanExporter** is injected into module's context via MBeanExportingPlugin and this exposes all the spring integration components inside the module. For the given module, the IntegrationMBeanExporter uses a specific object naming strategy that assigns domain name as xd.<stream/job name> and, object name as <module name>.<module index>.

Streams

For a stream name mystream with DSL http | log will have

MBeans with domain name xd.mystream with two objects http.0 and log.1

Source, processor, and sink modules will generally have the following attributes and operations

Module Type	Attributes and Operations
Source	MessageSourceMetrics
Processor,Sink	MessageHandlerMetrics

In addition, each module has channel attributes and operations defined by MessageChannelMetrics.

Jobs

For a job name myjob with DSL jdbchdfs will have

MBeans with domain name xd.myjob with an object jdbchdfs.0

You can also obtain monitoring information for Jobs using the UI or accessing the Job management REST API. Documentation for the Job Management REST API is forthcoming, but until then please reference the request mappings in BatchJobsController, BatchJobExecutionsController, BatchStepExecutionsController, and BatchJobInstancesController.

29.6 Using Jolokia to access JMX over http

When JMX is enabled (which is **default** via XD_JMX_ENABLED property), Jolokia is auto-configured to expose the XD admin, container and singenode server MBeans.

For example, with XD singlenode running management port 9080

```
http://localhost:9080/management/jolokia/search/xd.*:type=*,*
```

will list all the MBeans exposed in XD admin/container servers and the deployed modules. Apart from this, other available domain and types can be accessed via Jolokia.

30. REST API

30.1 Introduction

The Spring XD Administrator process (Admin) provides a REST API to access various Spring XD resources such as streams, jobs, metrics, modules, Spring batch resources, and container runtime information. The REST API is used internally by the XD Shell and Admin UI and can support any custom client application that requires interaction with XD.

The HTTP port is configurable and may be set as a command line argument when starting the Admin server, or set in $XD_HOME/config/servers.yml. The default port is 9393:

```
> $XD_HOME/bin/xd-admin --httpPort <port>
```

The Admin server also exposes runtime management resources enabled by Spring Boot under the `/management` context path, e.g., http://localhost:9393/management/metrics. These resources are covered in the Spring Boot documentation.

Note

There is also a `mgmtPort` command line argument which assigns a separate port for management services. Normally the same port is used for everything.

30.2 XD Resources

Table 30.1. Table XD REST endpoints

stream definitions	/streams/definitions
stream deployments	/streams/deployments
job definitions	/jobs/definitions
job deployments	/jobs/deployments
batch job configurations	/jobs/configurations
batch job executions	/jobs/executions
batch job instances	/jobs/instances
module definitions	/modules
deployed modules	/runtime/modules
containers	/runtime/containers
counters	/metrics/counters
field value counters	/metrics/field-value-counters
aggregate counters	/metrics/aggregate-counters
gauges	/metrics/gauges

rich-gauges	/metrics/rich-gauges
completions	/completions

30.3 Stream Definitions

Table 30.2. Table Stream Definitions

Resource URL	Request Method	Description
/streams/definitions	GET	list defined streams along with deployment state
/streams/definitions	DELETE	delete all stream definitions, undeploying deployed streams
/streams/definitions/{name}	GET	get a stream definition (currently no deployment information is included)
/streams/definitions	POST	create a new stream, optionally deploying it if `deploy=true`(default). The request body is application/x-www-form-urlencoded and requires two parameters, `name` and `definition` (DSL)
/streams/definitions/{name}	DELETE	delete a stream, undeploying if deployed.

30.4 Stream Deployments

Table 30.3. Table Stream Deployments

Resource URL	Request Method	Description
/streams/deployments/	GET	get detailed deployment state for all streams (TBD)
/streams/deployments	DELETE	undeploy all streams
/streams/deployments/{name}	GET	get detailed deployment state for a stream (TBD)
/streams/deployments/{name}	POST	deploy a stream, where the request body contains the deployment properties application/x-www-form-urlencoded
/streams/deployments/{name}	DELETE	undeploy a stream

30.5 Job Definitions

Table 30.4. Table Job Definitions

Resource URL	Request Method	Description
/jobs/definitions	GET	list defined jobs along with deployment state
/jobs/definitions	DELETE	delete all job definitions, undeploying deployed jobs
/jobs/definitions/{name}	GET	get a job definition
/jobs/definitions	POST	create a new job, where the The request body is application/x-www-form-urlencoded and requires two parameters, `name` and `definition` (DSL)
/jobs/definitions/{name}	DELETE	delete a job, undeploying if deployed

30.6 Job Deployments

Table 30.5. Table Job Deployments

Resource URL	Request Method	Description
/jobs/deployments/	GET	get detailed deployment state for all jobs (TBD)
/jobs/deployments	DELETE	undeploy all jobs
/jobs/deployments/{name}	GET	get detailed deployment state for a job (TBD. Probably not in 1.0)
/jobs/deployments/{name}	POST	deploy a job, where the request body contains the deployment properties
/jobs/deployments/{name}	DELETE	undeploy a job

30.7 Batch Job Configurations

Spring Batch configured jobs stored in the Spring Batch Repository

Table 30.6. Table Batch Jobs

Resource URL	Request Method	Description
/jobs/configurations	GET	get configuration information about all batch jobs

Resource URL	Request Method	Description
/jobs/configurations/{jobName}	GET	get configuration information about a batch job

30.8 Batch Job Executions

Table 30.7. Table Batch Executions

Resource URL	Request Method	Description
/jobs/executions	GET	list all job executions (Only *application/json* mediatype supported)
/jobs/executions?stop=true	PUT	stop all jobs
/jobs/executions?jobname={jobName}	GET	get information about all executions of a job (Only *application/json* accept header is supported)
/jobs/executions?jobname={jobName}	POST	request the launch of a job
/jobs/executions/{jobExecutionId}	GET	get information about a particular execution of a job
/jobs/executions/{jobExecutionId}?restart=true	PUT	restart a job
/jobs/executions/{jobExecutionId}?stop=true	PUT	stop a job
/jobs/executions/{jobExecutionId}/steps	GET	list the steps for a job execution (Only *application/json* accept header is supported)
/jobs/executions/{jobExecutionId}/steps/{stepExecutionId}	GET	get a step execution
/jobs/executions/{jobExecutionId}/steps/{stepExecutionId}/progress	GET	get the step execution progress

30.9 Batch Job Instances

Table 30.8. Table Batch Job Instances

Resource URL	Request Method	Description
/jobs/instances?jobname={jobName}	GET	get information about all instances of a job

Resource URL	Request Method	Description
/jobs/instances/{instanceId}	GET	get information about a batch job instance

For both the GET endpoints **only** *application/json* accept header is supported.

30.10 Module Definitions

Table 30.9. Table Module Definitions

Resource URL	Request Method	Description
/modules	GET	list all registered modules
/modules	POST	create a composite module, where The request body is application/x-www-form-urlencoded and requires two parameters, `name` and `definition` (DSL)
/modules/{type}/{name}	GET	list a module along with options metadata, where type is `source`,`processor`,`sink`, or `job`
/modules/{type}/{name}	DELETE	delete a composed module
/modules/{type}/{name}/definition	GET	display the module bean definition file (NOTE: currently only XML response supported)

30.11 Runtime Modules

Table 30.10. Table Runtime Modules

Resource URL	Request Method	Description
/runtime/modules	GET	display runtime module option values and deployment information for deployed modules, optional parameters are `moduleId` (<stream>.<type>.<moduleName>,`containerI`

30.12 Runtime Containers

Table 30.11. Table Runtime Containers

Resource URL	Request Method	Description
/runtime/containers	GET	display all available containers along with runtime and user-defined container attributes

30.13 Counters

Table 30.12. Table Counters

Resource URL	Request Method	Description
/metrics/counters/	GET	list all the known counters
/metrics/counters/{name}	GET	get the current metric value
/metrics/counters/{name}	DELETE	delete the metric

30.14 Field Value Counters

Table 30.13. Table Field Value Counters

Resource URL	Request Method	Description
/metrics/field-value-counters/	GET	list all the known field value counters
/metrics/field-value-counters/{name}	GET	get the current metric values
/metrics/field-value-counters/{name}	DELETE	delete the metric

30.15 Aggregate Counters

Table 30.14. Table Aggregate Counters

Resource URL	Request Method	Description
/metrics/aggregate-counters/	GET	list all the known aggregate counters
/metrics/aggregate-counters/{name}	GET	get the current metric values
/metrics/aggregate-counters/{name}	DELETE	delete the metric

30.16 Gauges

Table 30.15. Table Gauges

Resource URL	Request Method	Description
/metrics/gauges/	GET	list all the known gauges
/metrics/gauges/{name}	GET	get the current metric values
/metrics/gauges/{name}	DELETE	delete the metric

30.17 Rich Gauges

Table 30.16. Table Rich Gauges

Resource URL	Request Method	Description
/metrics/rich-gauges/	GET	list all the known rich gauges
/metrics/rich-gauges/{name}	GET	get the current metric values
/metrics/rich-gauges/{name}	DELETE	delete the metric

30.18 Tab Completions

Used to support DSL tab completion for the XD Shell. All requests require the `start` parameter which contains the incomplete definition.

Table 30.17. Table Tab Completions

Resource URL	Request Method	Description
/completions/stream?start={start}	GET	retrieve valid choices to complete a stream definition
/completions/job?start={start}	GET	retrieve valid choices to complete a job definition
/completions/module?start={start}	GET	retrieve valid choices to complete a module definition

Part V. 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 (v2.2.0) 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 (hadoop-2.2.0.tar.gz) 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-2.2.0/hadoop-2.2.0.tar.gz
```

Unpack the distribution with:

```
$ tar xzf hadoop-2.2.0.tar.gz
```

Change into the directory and have a look around

```
$ cd hadoop-2.2.0
$ ls
$ bin/hadoop
Usage: hadoop [--config confdir] COMMAND
       where COMMAND is one of:
  fs                   run a generic filesystem user client
  version              print the version
  jar <jar>            run a jar file
  ...
```

The `bin` directory contains the start and stop scripts as well as the `hadoop` and `hdfs` scripts which allow us to interact with *Hadoop* from the command line.

Java Setup

Make sure that you set `JAVA_HOME` in the `etc/hadoop/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* home directory by executing `$ /usr/libexec/java_home -v 1.6`

Tip

When using *Ubuntu* you can determine the *Java* home directory by executing `$ sudo update-java-alternatives -l`

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` and `ssh 0.0.0.0` before trying to start *Hadoop*:

```
$ ssh localhost
Last login: Thu May  1 15:02:32 2014 from localhost
...
$ ssh 0.0.0.0
Last login: Thu May  1 15:06:02 2014 from localhost
```

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 `etc/hadoop/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 can be overridden in `${xd.home}/config/server.yml`, 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 property setting in `etc/hadoop/core-site.xml`:

```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:8020</value>
</property>
```

Further Configuration File Changes

In `etc/hadoop/hdfs-site.xml` add the following properties:

```
<property>
    <name>dfs.replication</name>
    <value>1</value>
</property>
<property>
    <name>dfs.support.append</name>
    <value>true</value>
</property>
<property>
    <name>dfs.webhdfs.enabled</name>
    <value>true</value>
</property>
```

Create `etc/hadoop/mapred-site.xml` and add:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

In `etc/hadoop/yarn-site.xml` add these properties:

```
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>
<property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

A.2 Running Hadoop

First we need to set up the environment settings. It’s convenient to add these to a file that you can source when you want to work with Hadoop. We create a file called `hadoop-env` and add the following content:

```
# The directory of the unpacked distribution
export HADOOP_INSTALL="$HOME/Downloads/hadoop-2.2.0"

# The JAVE_HOME (see above how to determine this)
export JAVA_HOME=/usr/lib/jdk1.6.0_45

# Some HOME settings
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_YARN_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL

# Add Hadoop scripts to the PATH
export PATH=$HADOOP_INSTALL/bin:$HADOOP_INSTALL/sbin:$PATH
```

To use these settings we need to source this script:

```
$ source hadoop-env
```

You should now finally be ready to run *Hadoop*. Run the following commands

```
$ start-dfs.sh
$ start-yarn.sh
$ mr-jobhistory-daemon.sh start historyserver
```

You should see six Hadoop Java processes running:

```
$ jps
  21636 NameNode
  22004 SecondaryNameNode
  22360 NodeManager
  22425 JobHistoryServer
  21808 DataNode
  22159 ResourceManager
  22471 Jps
```

Try a few commands with `hdfs dfs` to make sure the basic system works

```
$ hdfs dfs -ls /
Found 1 items
drwxrwx---   - trisberg supergroup          0 2014-05-01 15:31 /tmp

$ hdfs dfs -mkdir /xd
$ bin/hadoop dfs -ls /
Found 2 items
drwxrwx---   - trisberg supergroup          0 2014-05-01 15:31 /tmp
drwxr-xr-x   - trisberg supergroup          0 2014-05-01 15:34 /xd
```

Lastly, you can also browse the web interface for *NameNode* and *ResourceManager* at:

- NameNode: http://localhost:50070/
- ResourceManager: http://localhost:8088/

At this point you should be good to create a *Spring XD* stream using a *Hadoop* sink.

Appendix B. Building Spring XD

B.1 Instructions

Here are some useful steps to build and run Spring XD.

To build all sub-projects and run tests for Spring XD (please note tests require a running Redis instance):

```
./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.

B.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.

B.3 Running script tests

Apart from the unit and integration tests, the directory `src/test/scripts` contains set of scripts that run end-to-end tests on XD runtime. Please see the instructions to setup and run:

- Once XD is built (with copyInstall), from the distribution directory: `build/dist/spring-xd/xd/bin/xd/bin/xd-singlenode(.bat)`
- Setup XD_HOME environment variable that points to `build/dist/spring-xd/xd`
- From the directory `src/test/scripts`, run `basic_stream_tests`

- For the jdbc_tests, we need to run install_sqlite_jar first that installs sqlite jar into $XD_HOME/lib
- For the hdfs_import_export_tests, make sure you have setup hadoop environment and have the xd-singlenode started with appropriate hadoopDistro option and hadoop lib jars for the version chosen
- For tweet_tests, make sure you have the twitter properties updated before running the tests

Appendix C. Using MQTT Modules

C.1 Introduction

MQTT(MQ for telemetry transport) is a machine to machine connectivity protocol. It is a lightweight pub/sub protocol for devices where bandwidth and battery power are at a premium. This purpose of this document is to show you how to: enable the RabbitMQ MQTT plugin, setup a Spring XD MQTT source and Spring MQTT sink.

Note

This document assumes that you have a RabbitMQ installed and running. If you don't have RabbitMQ available already you can download it from http://www.rabbitmq.com/download.html.

Setting up MQTT on RabbitMQ

If you are using RabbitMQ 3.3.4 or above then the MQTT plugin is already included with your deployment, however it is inactive. To Activate:

1. Shutdown the Rabbit MQ instance
2. $RABBIT_HOME/sbin/rabbitmq-plugins list

```
...
[ ] rabbitmq_federation_management    3.3.4
[E] rabbitmq_management               3.3.4
[e] rabbitmq_management_agent         3.3.4
[ ] rabbitmq_management_visualiser    3.3.4
[ ] rabbitmq_mqtt                     3.3.4
[ ] rabbitmq_shovel                   3.3.4
[ ] rabbitmq_shovel_management        3.3.4
…
```

3. We see that the rabitt_mqtt does not have a [E] denoted next to it. Thus it is not enabled. Note: if you do see the [E] next to the rabbitmq_mqtt then your plugin is enabled and all you need to do is restart your RabbitMQ.
4. Now enable rabbit_mqtt plugin

   a. Run: $RABBIT_HOME/sbin/rabbitmq-plugins enable rabbitmq_mqtt

   b. Run: $RABBIT_HOME/sbin/rabbitmq-plugins list

```
...
[ ] rabbitmq_federation_management    3.3.4
[E] rabbitmq_management               3.3.4
[e] rabbitmq_management_agent         3.3.4
[ ] rabbitmq_management_visualiser    3.3.4
[E] rabbitmq_mqtt                     3.3.4
[ ] rabbitmq_shovel                   3.3.4
[ ] rabbitmq_shovel_management        3.3.4
...
```

   c. Now we see that the rabbitmq_mqtt plugin is now active.

d. Restart your RabbitMQ.

Rabbit MQTT Plugin settings

The MQTT plugin can be can be configured via the rabbitmq.config file and this is covered here: http://www.rabbitmq.com/mqtt.html. The settings for the MQTT plugin that Spring XD are concerned about are as follows:

1. allow_anonymous Determines if the user must supply a user name or password. If true then the plugin will use the default_user and default password enumerated below. If false the Spring XD source or sink must provide the username and password Default: true

2. default_user If allow_anonymous is set to true then this will set the user for anonymous clients. Default: guest

3. default_password If allow_anonymous is set to true then this will set the password for anonymous clients. Default: guest

4. exchange - The name of the exchange that will route all MQTT messages to a the queues. Default: amq.topic

5. tcp_listeners - host and port that rabbit will monitor for MQTT messages. Default: 1883

Out of the box the Spring XD MQTT source and sink currently works with the MQTT plugin defaults without any configuration.

MQTT Source

When Spring XD deploys the MQTT source module, a message queue is created along with the necessary binding on RabbitMQ. The message queue that is created will have the name structure `[mqtt-subscription-][client_id][srcqos1]`. * mqtt-subscription- Queues created for MQTT subscribers will have names starting with mqtt-subscription. * client-id is the client-id specified by the MQTT source module, the default is `xd.mqtt.client.id` * srcqos1 - The QoS level for the queue.

The MQTT source module also generates the binding from the amq.topic to the message queue via the routing key (topic). The default topic for the MQTT source module is `xd.mqtt.test`.

Example 1: Using defaults

To show this in detail let us create the following stream: `stream create mqtt-in --definition “mqtt|log” --deploy`. In this example the stream will retrieve MQTT messages from RabbitMQ and write the content to Spring XD’s log. So on RabbitMQ a message queue named `mqtt-subscription-xd.mqtt.client.id.srcqos1` and a binding for the topic (routing key) `xd.mqtt.test` will be created.

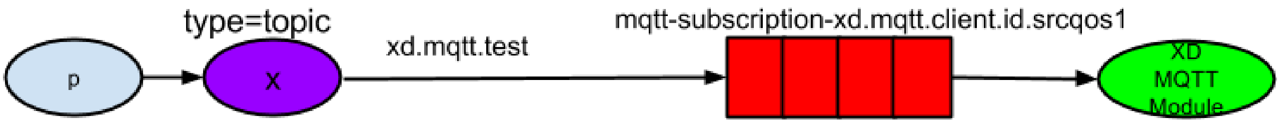


Figure C.1.

Thus any message published with the topic, `xd.mqtt.test` will be sent to the `mqtt-subscription-xd.mqtt.client.id.srcqos1` message queue and thus picked up by the Spring XD MQTT module and then written to log. So to exercise the stream created we can write the following:

```
stream create --name rabbittest --definition "http|rabbit --exchange='amq.topic' --
routingKey='''xd.mqtt.test'''" --deploy
http post --data 'hello world'
```

In the log you should see:

```
09:53:34,487  INFO MQTT Call: xd.mqtt.client.id.src sink.mqtt-in - hello world
```

Example 2: Monitoring different topics.

In this scenario we want to setup a MQTT Source Module to retrieve messages that may come in from different topics. So lets pretend that we want to monitor all the infusion machines at a medical facility. Our monitor wants to log all messages that notify us that a machine has completed its task or if a machine in need of maintenance.

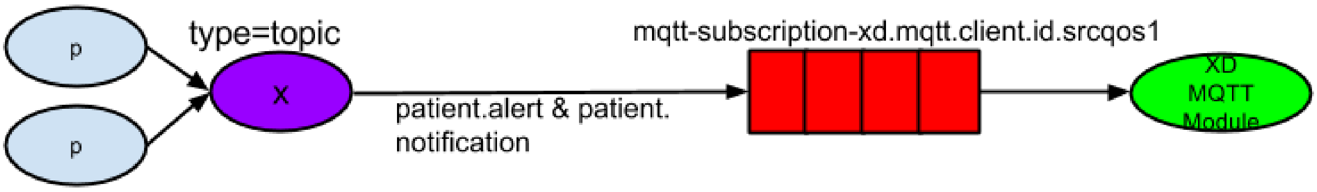


Figure C.2.

In this case it would look like this:

```
#Create a simulated device that will dispatch a patient alert message
stream create --name patientAlert --definition "http|rabbit --exchange='amq.topic' --
routingKey='''patient.alert'''" --deploy
#Create a simulated device that will dispatch a patient notification message
stream create --name patientNotification --definition "http --port=9005|rabbit --
exchange='amq.topic' --routingKey='''patient.notification'''" --deploy
# create our monitor that will capture the mqtt traffic.
stream create --name patientMonitor --definition "mqtt --
topics=patient.alert,patient.notification |log" --deploy
```

Now lets dispatch messages to both topics:

```
http post --target http://localhost:9005  --data 'infusion complete'
http post --data 'pump failure'
```

In the log you should see:

```
10:25:21,403  INFO MQTT Call: xd.mqtt.client.id.src sink.patientMonitor - infusion
 complete
10:25:46,226  INFO MQTT Call: xd.mqtt.client.id.src sink.patientMonitor - pump failure
```

MQTT Sink

The MQTT sink module will publish messages for a topic to the broker for a specific topic.

Example 1: Using defaults

In this example we will create a stream that will publish a message to topic using the defaults:

```
stream create mqtt-out --definition “http|mqtt” --deploy
stream create mqtt-in --definition”mqtt|log”
```

This mqtt-out stream will receive http messages to port 9000 on localhost and then the mqtt will publish the information to a rabbit instance on the localhost. The message will be routed to the queue (mqtt-subscription-xd.mqtt.client.id.srcqos1) that was created by MQTT source module and then, the message will be delivered to the MQTT source module.

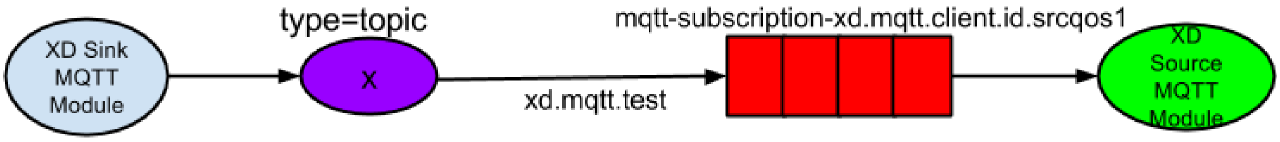


Figure C.3.

So the output will look something like this, if you execute a `http post --data ‘hello world’`

```
14:03:57,340  INFO MQTT Call: xd.mqtt.client.id.src sink.mqtt-in - hello world
```

Appendix D. XD Shell Command Reference

Below is a reference list of all Spring XD specific commands you can use in the XD Shell.

D.1 Configuration 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 timezone set

Set the timezone of the Spring XD Shell (Not persisted).

```
admin config timezone set [--timeZone] <timeZone>
```

timeZone

the id of the timezone, You can obtain a list of timezone ids using *admin config timezone list*, If an invalid timezone id is provided, then *Greenwich Mean Time* is being used. **(required)**

admin config info

Show the XD admin server being used.

```
admin config info
```

admin config timezone list

List all timezones.

```
admin config timezone list
```

D.2 Runtime Commands

runtime containers

List runtime containers.

```
runtime containers
```

runtime modules

List runtime modules.

```
runtime modules [--containerId <containerId>] [--moduleId <moduleId>]
```

containerId
to filter by container id.

moduleId
to filter by module id.

D.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: `false`, or `true` if `--deploy` is specified without a value)**

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> [--properties <properties>]
```

name
the name of the stream to deploy. **(required)**

properties
: the properties for this deployment.

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
```

D.4 Job Commands

job create

Create a job.

```
job create [--name] <name> --definition <definition> [--deploy [<deploy>]]
```

name
: the name to give to the job. **(required)**

definition
: job definition using xd dsl . **(required)**

deploy
: whether to deploy the job immediately. **(default: `false`, or `true` if `--deploy` is specified without a value)**

job list

List all jobs.

```
job list
```

job execution list

List all job executions.

```
job execution list
```

job execution step list

List all step executions for the provided job execution id.

```
job execution step list [--id] <id>
```

id
: the id of the job execution. **(required)**

job execution step progress

Get the progress info for the given step execution.

```
job execution step progress [--id] <id> --jobExecutionId <jobExecutionId>
```

id
: the id of the step execution. **(required)**

jobExecutionId
: the job execution id. **(required)**

job execution step display

Display the details of a Step Execution.

```
job execution step display [--id] <id> --jobExecutionId <jobExecutionId>
```

id
: the id of the step execution. **(required)**

jobExecutionId
: the job execution id. **(required)**

job execution display

Display the details of a Job Execution.

```
job execution display [--id] <id>
```

id
: the id of the job execution. **(required)**

job execution all stop

Stop all the job executions that are running.

```
job execution all stop [--force [<force>]]
```

force
: bypass confirmation prompt. **(default: `false`, or `true` if `--force` is specified without a value)**

job execution stop

Stop a job execution that is running.

```
job execution stop [--id] <id>
```

id
the id of the job execution. **(required)**

job execution restart

Restart a job that failed or interrupted previously.

```
job execution restart [--id] <id>
```

id
the id of the job execution that failed or interrupted. **(required)**

job deploy

Deploy a previously created job.

```
job deploy [--name] <name> [--properties <properties>]
```

name
the name of the job to deploy. **(required)**

properties
the properties for this deployment.

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 instance display

Display information about a given job instance.

```
job instance display [[--id] <id>]
```

id
the id of the job instance to retrieve.

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)**

D.5 Module Commands

module info

Get information about a module.

```
module info [--name] <name> [--hidden [<hidden>]]
```

name
name of the module to query, in the form *type:name*. **(required)**

hidden
whether to show *hidden* options. **(default: `false`, or `true` if `--hidden` is specified without a value)**

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>
```

name
name of the module to delete, in the form *type:name*. **(required)**

module list

List all modules.

```
module list
```

D.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>`)**

field-value-counter list

List all available field-value-counter names.

```
field-value-counter list
```

field-value-counter delete

Delete the field-value-counter with the given name.

```
field-value-counter delete [--name] <name>
```

name
the name of the field-value-counter to delete. **(required)**

field-value-counter display

Display the value of a field-value-counter.

```
field-value-counter 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`)**

aggregate-counter list

List all available aggregate counter names.

```
aggregate-counter list
```

aggregate-counter delete

Delete an aggregate counter.

```
aggregate-counter delete [--name] <name>
```

name
the name of the aggregate counter to delete. **(required)**

aggregate-counter display

Display aggregate counter values by chosen interval and resolution(minute, hour).

```
aggregate-counter 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>`)**

rich-gauge list

List all available richgauge names.

```
rich-gauge list
```

rich-gauge delete

Delete the richgauge.

```
rich-gauge delete [--name] <name>
```

name
: the name of the richgauge to delete. **(required)**

rich-gauge display

Display Rich Gauge value.

```
rich-gauge 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>`)**

D.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`)**

http get

Make GET request to http endpoint.

```
http get [[--target] <target>]
```

target
: the URL to make the request to. **(default: `http://localhost:9393`)**

D.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 URL - can be file:///|hdfs://<namenode>:<port>|webhdfs://<namenode>:<port>. **(required)**

D.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)**