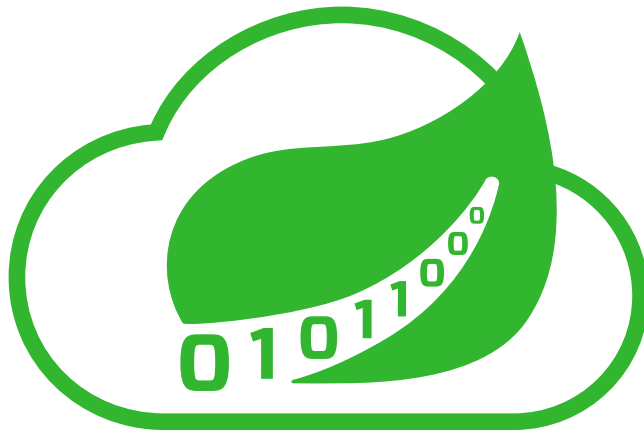




Spring Cloud Stream App Starters Reference Guide

Darwin.RELEASE

Sabby Anandan, Artem Bilan, Marius Bogoevici, Eric Bottard, Mark Fisher, Ilayaperumal Gopinathan, Gunnar Hillert, Mark Pollack, Patrick Peralta, Glenn Renfro, Gary Russell, Thomas Risberg, David Turanski, Janne Valkealahti, Soby Chacko, Christian Tzolov

Copyright © 2013-2016 Pivotal Software, Inc.

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

Table of Contents

I. Reference Guide	1
1. Introduction	2
1.1. Pre-built applications	2
1.2. Classification	2
1.3. Using the Artifacts	3
Maven and Docker access	3
Building the Artifacts	3
1.4. Custom Artifacts	4
Using a different binder	4
New Applications	5
Generic Applications	5
Customize Starter Applications	5
1.5. Patching Pre-built Applications	6
1.6. Creating New Stream Application Starters and Generating Artifacts	7
1.7. General faq on Spring Cloud Stream App Starters	11
II. Starters	15
2. Sources	16
2.1. File Source	16
Input	16
Output	16
mode = contents	16
mode = lines	16
mode = ref	17
Options	17
Build	18
Examples	18
2.2. FTP Source	18
Input	18
Output	18
mode = contents	18
mode = lines	19
mode = ref	19
Options	19
Build	21
Examples	21
2.3. Gemfire Source	21
Input	21
Output	21
Headers	21
Payload	21
Options	21
Build	22
Examples	22
2.4. Gemfire-CQ Source	22
Input	22
Output	22
Headers	22

Payload	22
Options	22
Build	23
Examples	23
2.5. Http Source	23
Input	23
Output	23
Headers:	23
Payload:	24
Options	24
Build	24
Examples	25
2.6. JDBC Source	25
Input	25
Output	25
Headers	25
Payload	25
Options	25
Build	26
Examples	26
2.7. JMS Source	26
Input	26
Output	26
Headers	26
Payload	27
Headers	27
Payload	27
Headers	27
Payload	27
Headers	27
Payload	27
Options	27
Build	28
Examples	28
2.8. Load Generator Source	28
Input	28
Output	28
Headers:	28
Payload:	28
Options	29
Build	29
Examples	29
2.9. Mail Source	29
Input	29
Output	29
Headers	29
Payload	29
Options	29
Build	30
Examples	30

2.10. MongoDB Source	31
Input	31
Output	31
Headers:	31
Payload:	31
Options	31
Build	32
Examples	32
2.11. MQTT Source	32
Input	32
Output	32
Headers:	32
Payload:	32
Options	33
Build	33
Examples	34
2.12. RabbitMQ Source	34
Input	34
Output	34
Headers	34
Payload	34
Headers	34
Payload	34
Headers	34
Payload	34
Options	34
A Note About Retry	36
Build	36
Examples	36
2.13. Amazon S3 Source	36
Input	36
Output	37
mode = contents	37
mode = lines	37
mode = ref	37
Options	37
Amazon AWS common options	39
Build	39
Examples	39
2.14. SFTP Source	39
Input	40
Output	40
mode = contents	40
mode = lines	40
mode = ref	40
Options	41
Build	43
Examples	43
2.15. SYSLOG Source	43
Input	44

Output	44
Headers	44
Payload	44
Options	44
Build	44
Examples	44
2.16. TCP	45
Input	45
Output	45
Headers:	45
Payload:	45
Options	45
Available Decoders	45
Build	46
Examples	46
2.17. TCP Client as a Source which connects to a TCP server and receives data	46
Input	46
Output	46
Headers:	46
Payload:	46
Options	47
Build	47
Examples	47
2.18. Time Source	47
Input	48
Output	48
Headers:	48
Payload:	48
Options	48
Build	48
Examples	48
2.19. Trigger Source	48
Input	49
Output	49
Headers:	49
Payload:	49
Options	49
Build	49
Examples	49
2.20. TriggerTask Source	49
Input	50
Output	50
Headers:	50
Payload:	50
Options	50
Build	51
Examples	51
2.21. Twitter Stream Source	51
Input	51
Output	51

Headers	51
Payload	51
Options	51
Build	52
Examples	52
3. Processors	53
3.1. Aggregator Processor	53
Input	53
Headers	53
Payload	53
Output	53
Headers	53
Payload	53
Options	53
Build	56
Examples	56
Code of Conduct	57
3.2. Bridge Processor	57
Input	57
Headers	57
Payload	57
Output	57
Headers	57
Payload	57
Options	57
Build	57
Examples	57
3.3. Filter Processor	57
Input	57
Headers	57
Payload	57
Output	58
Headers	58
Payload	58
Options	58
Build	58
Examples	58
3.4. Groovy Filter Processor	58
Input	58
Headers	58
Payload	58
Output	58
Headers	58
Payload	58
Options	59
Build	59
Examples	59
3.5. Groovy Transform Processor	59
Input	59
Headers	59

Payload	59
Output	59
Headers	59
Payload	59
Options	59
Build	60
Examples	60
3.6. gRPC Processor	60
Input	60
Headers	60
Payload	60
Output	61
Headers	61
Payload	61
Options	61
3.7. Header Enricher Processor	61
Input	61
Headers	61
Payload	61
Output	62
Headers	62
Payload	62
Options	62
Build	62
Examples	62
Code of Conduct	62
3.8. Http Client Processor	62
Input	62
Headers	62
Payload	63
Output	63
Headers	63
Payload	63
Options	63
Build	64
Examples	64
3.9. PMML Processor	64
Input	64
Headers	64
Payload	64
Output	64
Headers	64
Payload	64
Options	64
Build	65
Examples	65
3.10. Python Http Processor	65
Input	66
Headers	66
Payload	66

Output	66
Headers	66
Payload	66
Options	67
Build	68
Examples	68
3.11. Jython Processor	68
Input	69
Headers	69
Payload	69
Output	69
Headers	69
Payload	69
Options	69
Build	70
Examples	70
3.12. Scribable Transform Processor	70
Input	70
Headers	70
Payload	70
Output	70
Headers	70
Payload	70
Options	71
Build	71
Examples	71
3.13. Splitter Processor	71
Input	71
Headers	71
Payload	71
Output	71
Headers	71
Payload	71
Options	72
JSON Example	72
Build	73
Examples	73
3.14. TCP Client as a processor which connects to a TCP server, sends data to it and also receives data.	73
Input	73
Headers:	73
Payload:	73
Headers:	73
Payload:	73
Output	74
Headers:	74
Payload:	74
Options	74
Build	74
Examples	75

3.15. Transform Processor	75
Input	75
Headers	75
Payload	75
Output	75
Headers	75
Payload	75
Options	75
Build	75
Examples	76
3.16. TensorFlow Processor	76
Input	77
Headers	77
Payload	77
Output	77
Headers	77
Payload	77
Options	77
Build	78
Examples	78
3.17. Twitter Sentiment Analysis Processor	78
Input	79
Headers	79
Payload	79
Output	79
Headers	79
Payload	79
Payload	79
Options	79
Build	80
Examples	80
4. Sinks	81
4.1. Aggregate Counter Sink	81
Input	81
Headers	81
Payload	81
Output	81
Options	81
Build	82
Examples	82
4.2. Cassandra Sink	82
Input	82
Output	82
Options	82
Build	84
Examples	84
4.3. Counter Sink	84
Input	84
Headers	84
Payload	84

Output	84
Options	84
Build	85
Examples	85
4.4. Field Value Counter Sink	85
Input	85
Headers	85
Payload	85
Headers	85
Payload	86
Headers	86
Payload	86
Output	86
Options	86
Build	87
Examples	87
4.5. File Sink	87
Input	87
Headers	87
Payload	87
Output	88
Options	88
Build	88
Examples	88
4.6. FTP Sink	88
Input	89
Headers	89
Payload	89
Output	89
Options	89
Build	90
Examples	90
4.7. Gemfire Sink	90
Input	90
Headers	90
Payload	90
Headers	90
Payload	90
Output	91
Options	91
Build	91
Examples	91
4.8. Gpfdist Sink	91
Input	92
Headers:	92
Payload:	92
Output	92
Options	92
Implementation Notes	94
Detailed Option Descriptions	94

How Data Is Sent Into Segments	96
Example Usage	97
Tuning Transfer Rate	98
Build	98
Examples	98
4.9. HDFS Sink	98
Input	99
Headers	99
Payload	99
Output	99
Options	99
Build	100
Examples	100
4.10. Jdbc Sink	100
Input	100
Headers	100
Payload	100
Output	100
Options	100
Build	101
Examples	102
4.11. Log Sink	102
Input	102
Headers	102
Payload	102
Output	102
Options	102
Build	102
Examples	102
4.12. RabbitMQ Sink	103
Input	103
Headers	103
Payload	103
Headers	103
Payload	103
Headers	103
Payload	103
Output	103
Options	103
Build	105
Examples	105
4.13. MongoDB Sink	105
Input	105
Headers	105
Payload	105
Output	105
Options	105
Build	106
Examples	106
4.14. MQTT Sink	106

Input	106
Headers:	106
Payload:	106
Output	106
Options	106
Build	107
Examples	107
4.15. Pgcopy Sink	108
Input	108
Headers	108
Payload	108
Output	108
Options	108
Build	109
Examples	109
4.16. Redis Sink	109
Input	110
Headers	110
Payload	110
Headers	110
Payload	110
Output	110
Options	110
Build	111
Examples	112
4.17. Router Sink	112
Input	112
Headers	112
Payload	112
Output	112
Options	112
Options	112
SpEL-based Routing	113
Groovy-based Routing	113
Build	113
Examples	114
4.18. Amazon S3 Sink	114
Input	114
Headers	114
Payload	114
Output	114
Options	114
Amazon AWS common options	115
Build	115
Examples	116
4.19. SFTP Sink	116
Input	116
Headers	116
Payload	116
Output	116

Options	116
Build	117
Examples	118
4.20. TCP Sink	118
Input	118
Headers:	118
Payload:	118
Headers:	118
Payload:	118
Output	118
Options	118
Available Encoders	119
Build	119
Examples	119
4.21. Throughput Sink	120
Input	120
Headers	120
Payload	120
Output	120
Options	120
Build	120
Examples	120
4.22. Websocket Sink	120
Input	120
Headers	120
Payload	120
Output	120
Options	120
Build	121
Examples	121
Step 1: Start Rabbitmq	121
Step 2: Deploy a time-source	121
Step 3: Deploy a websocket-sink (the app that contains this starter jar)	121
Actuators	121
III. Appendices	123
A. Building	124
A.1. Basic Compile and Test	124
A.2. Documentation	124
A.3. Working with the code	124
Importing into eclipse with m2eclipse	124
Importing into eclipse without m2eclipse	125
B. App Starter POM Dependencies	126
C. App Starter Naming Conventions	127
5. Contributing	129
5.1. Sign the Contributor License Agreement	129
5.2. Code Conventions and Housekeeping	129

Part I. Reference Guide

This section will provide you with a detailed overview of Spring Cloud Stream Application Starters, their purpose, and how to use them. It assumes familiarity with general Spring Cloud Stream concepts, which can be found in the Spring Cloud Stream [reference documentation](#).

1. Introduction

Spring Cloud Stream Application Starters provide you with out-of-the-box Spring Cloud Stream utility applications that you can run independently or with Spring Cloud Data Flow. You can also use the starters as a basis for creating your own applications.

They include:

- connectors (sources, processors, and sinks) for a variety of middleware technologies including message brokers, storage (relational, non-relational, filesystem);
- adapters for various network protocols;
- generic processors that can be customized via [Spring Expression Language \(SpEL\)](#) or scripting.

You can find a detailed listing of all the starters and as their options in the [corresponding](#) section of this guide.

You can find all available app starter repositories in this [GitHub Organization](#).

1.1 Pre-built applications

As a user of Spring Cloud Stream Application Starters you have access to two types of artifacts.

Starters are libraries that contain the complete configuration of a Spring Cloud Stream application with a specific role (e.g. an *HTTP source* that receives HTTP POST requests and forwards the data on its output channel to downstream Spring Cloud Stream applications). Starters are *not* executable applications, and are intended to be included in other Spring Boot applications, along with an available Binder implementation of your choice.

Out-of-the-box applications are Spring Boot applications that include the starters and a Binder implementation - a fully functional uber-jar. These [uber-jar's](#) include minimal code required to execute standalone. For each starter application, the project provides a prebuilt version for Apache Kafka and Rabbit MQ Binders.



Note

Only starters are present in the source code of the project. Prebuilt applications are generated according to the [stream apps generator maven plugin](#).

1.2 Classification

Based on their target application type, starters can be either:

- a *source* that connects to an external resource to poll and receive data that is published to the default "output" channel;
- a *processor* that receives data from an "input" channel and processes it, sending the result on the default "output" channel;
- a *sink* that connects to an external resource to send the received data to the default "input" channel.

You can easily identify the type and functionality of a starter based on its name. All starters are named following the convention `spring-cloud-starter-stream-<type>-<functionality>`.

For example `spring-cloud-starter-stream-source-file` is a starter for a *file source* that polls a directory and sends file data on the output channel (read [the reference documentation of the source](#) for details). Conversely, `spring-cloud-starter-stream-sink-cassandra` is a starter for a *Cassandra sink* that writes the data that it receives on the input channel to Cassandra (read [the reference documentation of the sink](#) for details).

The prebuilt applications follow a naming convention too: `<functionality>-<type>-<binder>`. For example, `cassandra-sink-kafka-10` is a *Cassandra sink* using the Kafka binder that is running with Kafka version 0.10.

1.3 Using the Artifacts

You either get access to the artifacts produced by Spring Cloud Stream Application Starters via Maven, Docker, or building the artifacts yourself.

Maven and Docker access

Starters are available as Maven artifacts in the [Spring repositories](#). You can add them as dependencies to your application, as follows:

```
<dependency>
  <groupId>org.springframework.cloud.stream.app</groupId>
  <artifactId>spring-cloud-starter-stream-sink-cassandra</artifactId>
  <version>1.0.0.BUILD-SNAPSHOT</version>
</dependency>
```

From this, you can infer the coordinates for other starters found in this guide. While the version may vary, the group will always remain `org.springframework.cloud.stream.app` and the artifact id follows the naming convention `spring-cloud-starter-stream-<type>-<functionality>` described [previously](#).

Prebuilt applications are available as Maven artifacts too. It is not encouraged to use them directly as dependencies, as starters should be used instead. Following the typical Maven `<group>:<artifactId>:<version>` convention, they can be referenced for example as:

```
org.springframework.cloud.stream.app:cassandra-sink-rabbit:1.0.0.BUILD-SNAPSHOT
```

You can download the executable jar artifacts from the Spring Maven repositories. The root directory of the Maven repository that hosts release versions is repo.spring.io/release/org/springframework/cloud/stream/app/. From there you can navigate to the latest release version of a specific app, for example [log-sink-rabbit-1.1.1.RELEASE.jar](#). Use the [Milestone](#) and [Snapshot](#) repository locations for Milestone and Snapshot executable jar artifacts.

The Docker versions of the applications are available in Docker Hub, at hub.docker.com/r/springcloudstream/. Naming and versioning follows the same general conventions as Maven, e.g.

```
docker pull springcloudstream/cassandra-sink-kafka-10
```

will pull the *latest* Docker image of the *Cassandra sink* with the Kafka binder that is running with Kafka version 0.10.

Building the Artifacts

You can build the project and generate the artifacts (including the prebuilt applications) on your own. This is useful if you want to deploy the artifacts locally or add additional features.

First, you need to generate the prebuilt applications. This is done by running the application generation Maven plugin. You can do so by simply invoking the maven build with the generateApps profile and install lifecycle.

```
mvn clean install -PgenerateApps
```

Each of the prebuilt application will contain:

- pom.xml file with the required dependencies (starter and binder)
- a class that contains the main method of the application and imports the predefined configuration
- generated integration test code that validates the component against the configured binder.

For example, spring-cloud-starter-stream-sink-cassandra will generate cassandra-sink-rabbit and cassandra-sink-kafka-10 as completely functional applications.

1.4 Custom Artifacts

Apart from accessing the sources, sinks and processors already provided by the project, in this section we will describe how to:

- Use a different binder than Kafka or Rabbit
- Create your own applications
- Customize dependencies such as Hadoop distributions or JDBC drivers

Using a different binder

Prebuilt applications are provided for both kafka and rabbit binders. But if you want to connect to a different middleware system, and you have a binder for it, you will need to create new artifacts.

```
<dependencies>
  <!-- other dependencies -->
  <dependency>
    <groupId>org.springframework.cloud.stream.app</groupId>
    <artifactId>spring-cloud-starter-stream-sink-cassandra</artifactId>
    <version>1.0.0.BUILD-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-gemfire</artifactId>
    <version>1.0.0.BUILD-SNAPSHOT</version>
  </dependency>
</dependencies>
```

The next step is to create the project's main class and import the configuration provided by the starter.

```
package org.springframework.cloud.stream.app.cassandra.sink.rabbit;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.app.cassandra.sink.CassandraSinkConfiguration;
import org.springframework.context.annotation.Import;

@SpringBootApplication
@Import(CassandraSinkConfiguration.class)
public class CassandraSinkGemfireApplication {
```

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

New Applications

Spring Cloud Stream Application Starters consists of regular Spring Boot applications with some additional conventions that facilitate generating prebuilt applications with the preconfigured binders. Sometimes, your solution may require additional applications that are not in the scope of out-of-the-box Spring Cloud Stream Application Starters, or require additional tweaks and enhancements. In this section we will show you how to create custom applications that can be part of your solution, along with Spring Cloud Stream application starters. You have the following options:

- create new Spring Cloud Stream applications;
- use the starters to create customized versions;

Generic Applications

If you want to add your own custom applications to your solution, you can simply create a new Spring Cloud Stream app project with the binder of your choice and run it the same way as the applications provided by Spring Cloud Stream Application Starters, independently or via Spring Cloud Data Flow. The process is described in the [Getting Started Guide](#) of Spring Cloud Stream.

An alternative way to bootstrap your application is to go to the [Spring Initializr](#) and choose a Spring Cloud Stream Binder of your choice. This way you already have the necessary infrastructure ready to go and mainly focus on the specifics of the application.

The following requirements need to be followed when you go with this option:

- a single inbound channel named `input` for sources - the simplest way to do so is by using the predefined interface `org.springframework.cloud.stream.messaging.Source`;
- a single outbound channel named `output` for sinks - the simplest way to do so is by using the predefined interface `org.springframework.cloud.stream.messaging.Sink`;
- both an inbound channel named `input` and an outbound channel named `output` for processors - the simplest way to do so is by using the predefined interface `org.springframework.cloud.stream.messaging.Processor`.

Customize Starter Applications

You can also reuse the starters provided by Spring Cloud Stream Application Starters to create custom components, enriching the behavior of the application. For example, you can add a Spring Security layer to your *HTTP source*, add additional configurations to the `ObjectMapper` used for JSON transformation wherever that happens, or change the JDBC driver or Hadoop distribution that the application is using. In order to do this, you should set up your project following a process similar to [customizing a binder](#). In fact, customizing the binder is the simplest form of creating a custom component.

As a reminder, this involves:

- adding the starter to your project

- choosing the binder
- adding the main class and importing the starter configuration.

After doing so, you can simply add the additional configuration for the extra features of your application.

1.5 Patching Pre-built Applications

If you're looking to patch the pre-built applications to accommodate the addition of new dependencies, you can use the following example as the reference. Let's review the steps to add mysql driver to jdbc-sink application.

- Go to: start-scs.cfapps.io/
- Select the application and binder dependencies [*JDBC sink* and *Rabbit binder starter*]
- Generate and load the project in an IDE
- Add mysql java-driver dependency

```
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.37</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud.stream.app</groupId>
    <artifactId>spring-cloud-starter-stream-sink-jdbc</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

- Import the respective configuration class to the generated Spring Boot application. In the case of jdbc sink, it is: `@Import(org.springframework.cloud.stream.app.jdbc.sink.JdbcSinkConfiguration.class)`. You can find the configuration class for other applications in their respective [repositories](#).

```
@SpringBootApplication
@Import(org.springframework.cloud.stream.app.jdbc.sink.JdbcSinkConfiguration.class)
public class DemoApplication {

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

- Build and install the application to desired maven repository
- The patched copy of jdbc-sink application now includes mysql driver in it
- This application can be run as a standalone *uberjar*

1.6 Creating New Stream Application Starters and Generating Artifacts

In this section, we will explain how to develop a custom source/sink/processor application and then generate maven and docker artifacts for it with the necessary middleware bindings using the existing tooling provided by the spring cloud stream app starter infrastructure. For explanation purposes, we will assume that we are creating a new source application for a technology named foobar.

- Create a repository called foobar in your local github account
- The root artifact (something like foobar-app-starters-build) must inherit from app-starters-build

Please follow the instructions above for designing a proper Spring Cloud Stream Source. You may also look into the existing starters for how to structure a new one. The default naming for the main `@Configuration` class is `FoobarSourceConfiguration` and the default package for this `@Configuration` is `org.springframework.cloud.stream.app.foobar.source`. If you have a different class/package name, see below for overriding that in the app generator. The technology/functionality name for which you create a starter can be a hyphenated stream of strings such as in `scriptable-transform` which is a processor type in the module `spring-cloud-starter-stream-processor-scriptable-transform`.

The starters in `spring-cloud-stream-app-starters` are slightly different from the other starters in `spring-boot` and `spring-cloud` in that here we don't provide a way to auto configure any configuration through spring factories mechanism. Rather, we delegate this responsibility to the maven plugin that is generating the binder based apps. Therefore, you don't have to provide a `spring.factories` file that lists all your configuration classes.

- The starter module needs to inherit from the parent (foobar-app-starters-build)
- Add the new foobar source module to the root pom of the new repository
- In the pom.xml for the source module, add the following in the build section. This will add the necessary plugin configuration for app generation as well as generating proper documentation metadata. Please ensure that your root pom inherits [app-starters-build](#) as the base configuration for the plugins is specified there.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-app-starter-doc-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>org.springframework.cloud.stream.app.plugin</groupId>
      <artifactId>spring-cloud-stream-app-maven-plugin</artifactId>
      <configuration>
        <generatedProjectHome>${session.executionRootDirectory}/apps</generatedProjectHome>
        <generatedProjectVersion>${project.version}</generatedProjectVersion>
        <bom>
          <name>scs-bom</name>
          <groupId>org.springframework.cloud.stream.app</groupId>
          <artifactId>foobar-app-dependencies</artifactId>
          <version>${project.version}</version>
        </bom>
        <generatedApps>
          <foobar-source/>
        </generatedApps>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```
</plugins>
</build>
```

More information about the maven plugin used above to generate the apps can be found here: github.com/spring-cloud/spring-cloud-stream-app-maven-plugin

If you did not follow the default convention expected by the plugin for where it is looking for the main configuration class, which is `org.springframework.cloud.stream.app.foobar.source.FoobarSourceConfiguration`, you can override that in the configuration for the plugin. For example, if your main configuration class is `foo.bar.SpecialFooBarConfiguration.class`, this is how you can tell the plugin to override the default.

```
<foobar-source>
  <autoConfigClass>foo.bar.SpecialFooBarConfiguration.class</autoConfigClass>
</foobar-source>
```

- Create a new module to manage dependencies for foobar (foobar-app-dependencies). This is the bom (bill of material) for this project. It is advised that this bom is inherited from spring-cloud-dependencies-parent. Please see other starter repositories for guidelines.
- You need to add the new starter dependency to the BOM in the dependency management section. For example,

```
<dependencyManagement>
...
...
  <dependency>
    <groupId>org.springframework.cloud.stream.app</groupId>
    <artifactId>spring-cloud-starter-stream-source-foobar</artifactId>
    <version>1.0.0.BUILD-SNAPSHOT</version>
  </dependency>
...
...
```

- At the root of the repository build, install and generate the apps:

```
./mvnw clean install -PgenerateApps
```

This will generate the binder based foobar source apps in a directory named apps at the root of the repository. If you want to change the location where the apps are generated, for instance ``/tmp/scs-apps`, you can do it in the configuration section of the plugin.

```
<configuration>
...
  <generatedProjectHome>/tmp/scs-apps</generatedProjectHome>
...
</configuration>
```

By default, we generate apps for both Kafka 09/10 and Rabbitmq binders - spring-cloud-stream-binder-kafka and spring-cloud-stream-binder-rabbit. Say, if you have a custom binder you created for some middleware (say JMS), which you need to generate apps for foobar source, you can add that binder to the binders list in the configuration section as in the following.

```
<binders>
  <jms />
</binders>
```

Please note that this would only work, as long as there is a binder with the maven coordinates of `org.springframework.cloud.stream` as group id and `spring-cloud-stream-binder-jms`

as artifact id. This artifact needs to be specified in the BOM above and available through a maven repository as well.

If you have an artifact that is only available through a private internal maven repository (may be an enterprise wide Nexus repo that you use globally across teams), and you need that for your app, you can define that as part of the maven plugin configuration.

For example,

```
<configuration>
...
  <extraRepositories>
    <repository>
      <id>private-internal-nexus</id>
      <url>...</url>
      <name>...</name>
      <snapshotEnabled>...</snapshotEnabled>
    </repository>
  </extraRepositories>
</configuration>
```

Then you can define this as part of your app tag:

```
<foobar-source>
  <extraRepositories>
    <private-internal-nexus />
  </extraRepositories>
</foobar-source>
```

- cd into the directory where you generated the apps (apps at the root of the repository by default, unless you changed it elsewhere as described above).

Here you will see `foobar-source-kafka-09`, `foobar-source-kafka-10` and `foobar-source-rabbit`. If you added more binders as described above, you would see that app as well here - for example `foobar-source-jms`.

You can import these apps directly into your IDE of choice if you further want to do any customizations on them. Each of them is a self contained spring boot application project. For the generated apps, the parent is `spring-boot-starter-parent` as required by the underlying Spring Initializr library.

You can cd into these custom `foobar-source` directories and do the following to build the apps:

```
cd foo-source-kafka-10
```

```
mvn clean install
```

This would install the `foo-source-kafka-10` into your local maven cache (`~/.m2` by default).

The app generation phase adds an integration test to the app project that is making sure that all the spring components and contexts are loaded properly. However, these tests are not run by default when you do a `mvn install`. You can force the running of these tests by doing the following:

```
mvn clean install -DskipTests=false
```

One important note about running these tests in generated apps:

If your application's spring beans need to interact with some real services out there or expect some properties to be present in the context, these tests will fail unless you make those things available. An example would be a Twitter Source, where the underlying spring beans are

trying to create a twitter template and will fail if it can't find the credentials available through properties. One way to solve this and still run the generated context load tests would be to create a mock class that provides these properties or mock beans (for example, a mock twitter template) and tell the maven plugin about its existence. You can use the existing module `app-starters-test-support` for this purpose and add the mock class there. See the class `org.springframework.cloud.stream.app.test.twitter.TwitterTestConfiguration` for reference. You can create a similar class for your foobar source - `FooBarTestConfiguration` and add that to the plugin configuration. You only need to do this if you run into this particular issue of spring beans are not created properly in the integration test in the generated apps.

```
<foobar-source>
  <extraTestClass>org.springframework.cloud.stream.app.test.foobar.FoobarTestConfiguration.class</extraTestClass>
</foobar-source>
```

When you do the above, this test configuration will be automatically imported into the context of your test class.

Also note that, you need to regenerate the apps each time you make a configuration change in the plugin.

- Now that you built the applications, they are available under the `target` directories of the respective apps and also as maven artifacts in your local maven repository. Go to the `target` directory and run the following:

```
java -jar foobar-source-kafa-10.jar [Ensure that you have kafka running locally when you do this]
```

It should start the application up.

- The generated apps also support the creation of docker images. You can `cd` into one of the `foobar-source*` app and do the following:

```
mvn clean package docker:build
```

This creates the docker image under the `target/docker/springcloudstream` directory. Please ensure that the Docker container is up and running and `DOCKER_HOST` environment variable is properly set before you try `docker:build`.

All the generated apps from the various app repositories are uploaded to [Docker Hub](#)

However, for a custom app that you build, this won't be uploaded to docker hub under `springcloudstream` repository. If you think that there is a general need for this app, you should try contributing this starter as a new repository to [Spring Cloud Stream App Starters](#). Upon review, this app then can be eventually available through the above location in docker hub.

If you still need to push this to docker hub under a different repository (may be an enterprise repo that you manage for your organization) you can take the following steps.

Go to the `pom.xml` of the generated app [example - `foo-source-kafka/pom.xml`] Search for `springcloudstream`. Replace with your repository name.

Then do this:

```
mvn clean package docker:build docker:push -Ddocker.username=[provide your username] -Ddocker.password=[provide password]
```

This would upload the docker image to the docker hub in your custom repository.

1.7 General faq on Spring Cloud Stream App Starters

In the following sections, you can find a brief faq on various things that we discussed above and a few other infrastructure related topics.

1. What are Spring Cloud Stream Application Starters?

Spring Cloud Stream Application Starters are Spring Boot based Spring Integration applications that provide integration with external systems. GitHub: github.com/spring-cloud-stream-app-starters
Project page: cloud.spring.io/spring-cloud-stream-app-starters/

2. What is the parent for stream app starters?

The parent for all app starters is `app-starters-build` which is coming from the core project. github.com/spring-cloud-stream-app-starters/core For example:

```
<parent>
  <groupId>org.springframework.cloud.stream.app</groupId>
  <artifactId>app-starters-build</artifactId>
  <version>2.0.0.RELEASE</version>
  <relativePath/>
</parent>
```

3. Why is there a BOM in the core proejct?

Core defines a BOM which contains all the dependency management for common artifacts. This BOM is named as `app-starters-core-dependencies`. We need this bom during app generation to pull down all the core dependencies.

4. What are the contents of the core BOM?

In addition to the common artifacts in core, the `app-starters-core-dependencies` BOM also adds dependency management for `spring-cloud-dependencies` which will include `spring-cloud-stream` transitively.

5. Where is the core BOM used?

There are two places where the core BOM is used. It is used to provide compile time dependency management for all the starters. This is defined in the `app-starters-build` artifact. This same BOM is referenced through the maven plugin configuration for the app generation. The generated apps thus will include this bom also in their `pom.xml` files.

6. What spring cloud stream artifacts does the parent artifact (`app-starters-build`) include?

- `spring-cloud-stream`
- `Spring-cloud-stream-test-support-internal`
- `spring-cloud-stream-test-support`

7. What other artiacts are available through the parent `app-starters-build` and where are they coming from?

In addition to the above artifacts, the artifacts below also included in `app-starters-build` by default.

- `json-path`
- `spring-integration-xml`

- `spring-boot-starter-logging`

- `spring boot-starter-security`

`Spring-cloud-build` is the parent for `app-starters-build`. `Spring-cloud-build` imports `spring-boot-dependencies` and that is from where these artifacts are coming from.

8. I did not see any other Spring Integration components used in the above 2 lists. Where are those dependencies coming from for individual starters?

`Spring-integration bom` is imported in the `spring-boot-dependencies bom` and this is where the default SI dependencies are coming for SCSt app starters.

9. Can you summarize all the BOM's that SCSt app starters depend on?

All SCSt app starters have access to dependencies defined in the following BOM's and other dependencies from any other BOM's these three boms import transitively as in the case of Spring Integration:

- `app-starters-core-dependencies`
- `spring-cloud-dependencies`
- `spring-boot-dependencies`

10 Each app starter has `app-starter-build` as the parent which in turn has `spring-cloud-build` as parent. The above documentation states that the generated apps have `spring-boot-starter` as the parent. Why the mismatch?

There is no mismatch per se, but a slight subtlety. As the question frames, each app starter has access to artifacts managed all the way through `spring-cloud-build` at compile time. However, this is not the case for the generated apps at runtime. Generated apps are managed by boot. Their parent is `spring-boot-starter` that imports `spring-boot-dependencies bom` that includes a majority of the components that these apps need. The additional dependencies that the generated application needs are managed by including a BOM specific to each application starter.

11 Why is there an app starter specific BOM in each app starter repositories? For example, `time-app-dependencies`.

This is an important BOM. At runtime, the generated apps get the versions used in their dependencies through a BOM that is managing the dependencies. Since all the boms that we specified above only for the helper artifacts, we need a place to manage the starters themselves. This is where the app specific BOM comes into play. In addition to this need, as it becomes clear below, there are other uses for this BOM such as dependency overrides etc. But in a nutshell, all the starter dependencies go to this BOM. For instance, take TCP repo as an example. It has a starter for source, sink, client processor etc. All these dependencies are managed through the app specific `tcp-app-dependencies bom`. This bom is provided to the app generator maven plugin in addition to the core bom. This app specific bom has `spring-cloud-dependencies-parent` as parent.

12 How do I create a new app starter project?

If you have a general purpose starter that can be provided as an of of the box app, create an issue for that in [app-starters-release](#). If there is a consensus, then a repository can be created in the `spring-cloud-stream-app-starters` organization where you can start contributing the starters and other components.

13 I created a new starter according to the guidelines above, now how do I generate binder specific apps for the new starters?

By default, the app-starters-build in core is configured with the common configuration needed for the app generator maven plugin. It is configured for generating apps for kafka-09, kafka-10 and rabbitmq binders. In your starter you already have the configuration specified for the plugin from the parent. Modify the configuration for your starter accordingly. Refer to an existing starter for guidelines. Here is an example of modifying such a configuration : github.com/spring-cloud-stream-app-starters/time/blob/master/spring-cloud-starter-stream-source-time/pom.xml Look for spring-cloud-stream-app-maven-plugin in the plugins section under build. You generate binder based apps using the generateApps maven profile. You need the maven install lifecycle to generate the apps.

14How do I override Spring Integration version that is coming from spring-boot-dependencies by default?

The following solution only works if the versions you want to override are available through a new Spring Integration BOM. Go to your app starter specific bom. Override the property as following:

```
<spring-integration.version>VERSION GOES HERE</spring-integration.version>
```

Then add the following in the dependencies management section in the BOM.

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-bom</artifactId>
  <version>${spring-integration.version}</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

15How do I override spring-cloud-stream artifacts coming by default in spring-cloud-dependencies defined in core BOM?

The following solution only works if the versions you want to override are available through a new Spring-Cloud-Dependencies BOM. Go to your app starter specific bom. Override the property as following:

```
<spring-cloud-dependencies.version>VERSION GOES HERE</spring-cloud-dependencies.version>
```

Then add the following in the dependencies management section in the BOM.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-dependencies</artifactId>
  <version>${spring-cloud-dependencies.version}</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

16What if there is no spring-cloud-dependencies BOM available that contains my versions of spring-cloud-stream, but there is a spring-cloud-stream BOM available?

Go to your app starter specific BOM. Override the property as below.

```
<spring-cloud-stream.version>VERSION GOES HERE</spring-cloud-stream.version>
```

Then add the following in the dependencies management section in the BOM.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-dependencies</artifactId>
  <version>${spring-cloud-stream.version}</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

17. What if I want to override a single artifact that is provided through a bom? For example spring-integration-java-dsl?

Go to your app starter BOM and add the following property with the version you want to override:

```
<spring-integration-java-dsl.version>VERSION GOES HERE</spring-integration-java-dsl.version>
```

Then in the dependency management section add the following:

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-java-dsl</artifactId>
  <version>${spring-integration-java-dsl.version}</version>
</dependency>
```

18. How do I override the boot version used in a particular app?

When you generate the app, override the boot version as follows.

```
./mvnw clean install -PgenerateApps -DbootVersion=<boot version to override>
```

For example: `./mvnw clean install -PgenerateApps -DbootVersion=2.0.0.BUILD-SNAPSHOT`

You can also override the boot version more permanently by overriding the following property in your starter pom.

```
<bootVersion>2.0.0.BUILD-SNAPSHOT</bootVersion>
```

Part II. Starters

2. Sources

2.1 File Source

This application polls a directory and sends new files or their contents to the output channel. The file source provides the contents of a File as a byte array by default. However, this can be customized using the `--mode` option:

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

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

Input

N/A (Reads from a file system directory).

Output

mode = contents

Headers:

- Content-Type: `application/octet-stream`
- `file_originalFile`: `<java.io.File>`
- `file_name`: `<file name>`

Payload:

A `byte[]` filled with the file contents.

mode = lines

Headers:

- Content-Type: `text/plain`
- `file_originalFile`: `<java.io.File>`
- `file_name`: `<file name>`
- `correlationId`: `<UUID>` (same for each line)
- `sequenceNumber`: `<n>`
- `sequenceSize`: `0` (number of lines is not known until the file is read)

Payload:

A `String` for each line.

The first line is optionally preceded by a message with a START marker payload. The last line is optionally followed by a message with an END marker payload.

Marker presence and format are determined by the `with-markers` and `markers-json` properties.

mode = ref

Headers:

None.

Payload:

A `java.io.File` object.

Options

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

`file.consumer.markers-json`

When `'fileMarkers == true'`, specify if they should be produced as `FileSplitter.FileMarker` objects or JSON. **(Boolean, default: true)**

`file.consumer.mode`

The `FileReadingMode` to use for file reading sources. Values are `'ref'` - The File object, `'lines'` - a message per line, or `'contents'` - the contents as bytes. **(FileReadingMode, default: <none>, possible values: ref,lines,contents)**

`file.consumer.with-markers`

Set to true to emit start of file/end of file marker messages before/after the data. Only valid with `FileReadingMode 'lines'`. **(Boolean, default: <none>)**

`file.directory`

The directory to poll for new files. **(String, default: <none>)**

`file.filename-pattern`

A simple ant pattern to match files. **(String, default: <none>)**

`file.filename-regex`

A regex pattern to match files. **(Pattern, default: <none>)**

`file.prevent-duplicates`

Set to true to include an `AcceptOnceFileListFilter` which prevents duplicates. **(Boolean, default: true)**

`trigger.cron`

Cron expression value for the Cron Trigger. **(String, default: <none>)**

`trigger.date-format`

Format for the date value. **(String, default: <none>)**

`trigger.fixed-delay`

Fixed delay for periodic triggers. **(Integer, default: 1)**

`trigger.initial-delay`

Initial delay for periodic triggers. **(Integer, default: 0)**

trigger.max-messages

Maximum messages per poll, -1 means infinity. **(Long, default: -1)**

trigger.time-unit

The TimeUnit to apply to delay values. **(TimeUnit, default: SECONDS, possible values: NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS)**

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.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps

You can find the corresponding binder based projects here. You can then cd into one of the folders and
build it:

$ ./mvnw clean package
```

Examples

```
java -jar file_source.jar --file.directory=/tmp/foo --file.consumer.mode=lines --trigger.fixed-delay=60
```

2.2 FTP Source

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

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

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

Input

N/A (Fetches files from an FTP server).

Output

mode = contents

Headers:

- Content-Type: application/octet-stream
- file_originalFile: <java.io.File>
- file_name: <file name>

Payload:

A `byte[]` filled with the file contents.

mode = lines**Headers:**

- Content-Type: `text/plain`
- `file_originalFile`: `<java.io.File>`
- `file_name`: `<file name>`
- `correlationId`: `<UUID>` (same for each line)
- `sequenceNumber`: `<n>`
- `sequenceSize`: `0` (number of lines is not known until the file is read)

Payload:

A `String` for each line.

The first line is optionally preceded by a message with a START marker payload. The last line is optionally followed by a message with an END marker payload.

Marker presence and format are determined by the `with-markers` and `markers-json` properties.

mode = ref**Headers:**

None.

Payload:

A `java.io.File` object.

Options

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

`file.consumer.markers-json`

When `'fileMarkers == true'`, specify if they should be produced as `FileSplitter.FileMarker` objects or JSON. **(Boolean, default: true)**

`file.consumer.mode`

The `FileReadingMode` to use for file reading sources. Values are `'ref'` - The File object, `'lines'` - a message per line, or `'contents'` - the contents as bytes. **(FileReadingMode, default: <none>, possible values: ref,lines,contents)**

`file.consumer.with-markers`

Set to true to emit start of file/end of file marker messages before/after the data. Only valid with `FileReadingMode 'lines'`. **(Boolean, default: <none>)**

`ftp.auto-create-local-dir`

Set to true to create the local directory if it does not exist. **(Boolean, default: true)**

ftp.delete-remote-files

Set to true to delete remote files after successful transfer. (**Boolean, default: false**)

ftp.factory.cache-sessions

<documentation missing> (**Boolean, default: <none>**)

ftp.factory.client-mode

The client mode to use for the FTP session. (**ClientMode, default: <none>, possible values: ACTIVE,PASSIVE**)

ftp.factory.host

<documentation missing> (**String, default: <none>**)

ftp.factory.password

<documentation missing> (**String, default: <none>**)

ftp.factory.port

The port of the server. (**Integer, default: 21**)

ftp.factory.username

<documentation missing> (**String, default: <none>**)

ftp.filename-pattern

A filter pattern to match the names of files to transfer. (**String, default: <none>**)

ftp.filename-regex

A filter regex pattern to match the names of files to transfer. (**Pattern, default: <none>**)

ftp.local-dir

The local directory to use for file transfers. (**File, default: <none>**)

ftp.preserve-timestamp

Set to true to preserve the original timestamp. (**Boolean, default: true**)

ftp.remote-dir

The remote FTP directory. (**String, default: /**)

ftp.remote-file-separator

The remote file separator. (**String, default: /**)

ftp.tmp-file-suffix

The suffix to use while the transfer is in progress. (**String, default: .tmp**)

trigger.cron

Cron expression value for the Cron Trigger. (**String, default: <none>**)

trigger.date-format

Format for the date value. (**String, default: <none>**)

trigger.fixed-delay

Fixed delay for periodic triggers. (**Integer, default: 1**)

trigger.initial-delay

Initial delay for periodic triggers. (**Integer, default: 0**)

trigger.max-messages

Maximum messages per poll, -1 means infinity. **(Long, default: -1)**

trigger.time-unit

The TimeUnit to apply to delay values. **(TimeUnit, default: SECONDS, possible values: NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar ftp_source.jar --ftp.remote-dir=foo --file.mode=lines --trigger.fixed-delay=60 --
ftp.factory.host=ftpsrvr \
    --ftp.factory.username=user --ftp.factory.password=pw --ftp.local-dir=/foo
```

2.3 Gemfire Source

This source allows you to subscribe to any creates or updates to a Gemfire region. The application configures a client cache and client region, along with the necessary subscriptions enabled. 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.

Input

N/A

Output

Headers

- content-type: text/plain

Payload

- String

Options

The **gemfire** source supports the following configuration properties:

gemfire.cache-event-expression

SpEL expression to extract fields from a cache event. **(Expression, default: <none>)**

gemfire.pool.connect-type

Specifies connection type: 'server' or 'locator'. **(ConnectType, default: <none>, possible values: locator, server)**

gemfire.pool.host-addresses

Specifies one or more Gemfire locator or server addresses formatted as [host]:[port].
(**InetSocketAddress[], default: <none>**)

gemfire.pool.subscription-enabled

Set to true to enable subscriptions for the client pool. Required to sync updates to the client cache.
(**Boolean, default: false**)

gemfire.region.region-name

The region name. (**String, default: <none>**)

gemfire.security.password

The cache password. (**String, default: <none>**)

gemfire.security.username

The cache username. (**String, default: <none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar ./gemfire-source.jar --gemfire.region.region-name=MyRegion --
gemfire.cacheEventExpression="newValue"
```

2.4 Gemfire-CQ Source

Continuous query allows client applications to create a GemFire query using Object Query Language (OQL) and to 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.

Input

N/A

Output

Headers

- content-type: text/plain

Payload

- String

Options

The **gemfire-cq** source supports the following configuration properties:

gemfire.cq-event-expression

SpEL expression to use to extract data from a cq event. **(Expression, default: <none>)**

gemfire.pool.connect-type

Specifies connection type: 'server' or 'locator'. **(ConnectType, default: <none>, possible values: locator,server)**

gemfire.pool.host-addresses

Specifies one or more Gemfire locator or server addresses formatted as [host]:[port]. **(InetSocketAddress[], default: <none>)**

gemfire.pool.subscription-enabled

Set to true to enable subscriptions for the client pool. Required to sync updates to the client cache. **(Boolean, default: false)**

gemfire.query

The OQL query **(String, default: <none>)**

gemfire.security.password

The cache password. **(String, default: <none>)**

gemfire.security.username

The cache username. **(String, default: <none>)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar gemfire-cq-source.jar --gemfire.query=
```

2.5 Http Source

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

Input

N/A

Output

Headers:

- Content-Type: Any

Payload:

If content type matches `text/*` or `application/json`

- String

If content type does not match `text/*` or `application/json`

- byte array

Options

The **http** source supports the following configuration properties:

`http.cors.allow-credentials`

Whether the browser should include any cookies associated with the domain of the request being annotated. **(Boolean, default: <none>)**

`http.cors.allowed-headers`

List of request headers that can be used during the actual request. **(String[], default: <none>)**

`http.cors.allowed-origins`

List of allowed origins, e.g. `"http://domain1.com"`. **(String[], default: <none>)**

`http.enable-csrf`

The security CSRF enabling flag. Makes sense only if `'enableSecurity = true'`. **(Boolean, default: false)**

`http.enable-security`

The security enabling flag. **(Boolean, default: false)**

`http.mapped-request-headers`

Headers that will be mapped. **(String[], default: <none>)**

`http.path-pattern`

An Ant-Style pattern to determine which http requests will be captured. **(String, default: /)**

`server.port`

Server HTTP port. **(Integer, default: 8080)**

**Note**

Security is disabled for this application by default. To enable it, you should use the mentioned above `http.enable-security = true` property.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar http_source.jar
```

2.6 JDBC Source

This source polls data from an RDBMS. This source is fully based on the `DataSourceAutoConfiguration`, so refer to the [Spring Boot JDBC Support](#) for more information.

Input

N/A

Output

Headers

- Content-Type: application/x-java-object

Payload

- `List<Map<String, Object>>`

Options

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

`jdbc.max-rows-per-poll`

Max numbers of rows to process for each poll. (**Integer, default: 0**)

`jdbc.query`

The query to use to select data. (**String, default: <none>**)

`jdbc.split`

Whether to split the SQL result as individual messages. (**Boolean, default: true**)

`jdbc.update`

An SQL update statement to execute for marking polled messages as 'seen'. (**String, default: <none>**)

`spring.datasource.data`

Data (DML) script resource references. (**List<String>, default: <none>**)

`spring.datasource.driver-class-name`

Fully qualified name of the JDBC driver. Auto-detected based on the URL by default. (**String, default: <none>**)

`spring.datasource.initialization-mode`

Initialize the datasource using available DDL and DML scripts. (**DataSourceInitializationMode, default: embedded, possible values: ALWAYS,EMBEDDED,NEVER**)

`spring.datasource.password`

Login password of the database. (**String, default: <none>**)

`spring.datasource.schema`

Schema (DDL) script resource references. (**List<String>, default: <none>**)

spring.datasource.url

JDBC url of the database. **(String, default: <none>)**

spring.datasource.username

Login username of the database. **(String, default: <none>)**

trigger.cron

Cron expression value for the Cron Trigger. **(String, default: <none>)**

trigger.date-format

Format for the date value. **(String, default: <none>)**

trigger.fixed-delay

Fixed delay for periodic triggers. **(Integer, default: 1)**

trigger.initial-delay

Initial delay for periodic triggers. **(Integer, default: 0)**

trigger.max-messages

Maximum messages per poll, -1 means infinity. **(Long, default: 1)**

trigger.time-unit

The TimeUnit to apply to delay values. **(TimeUnit, default: <none>, possible values: NANoseconds, MICROseconds, MILLIseconds, SECONDS, MINUTES, HOURS, DAYS)**

Also see the [Spring Boot Documentation](#) for additional DataSource properties and TriggerProperties and MaxMessagesProperties for polling options.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar jdbc_source.jar --query=<QUERY> [datasource credentials]
```

2.7 JMS Source

The "jms" source enables receiving messages from JMS.

Input

N/A

Output

Headers

- content-type: text/plain

Payload

- String

Headers

- content-type: application/octet-stream

Payload

- byte[]

Headers

- content-type: application/x-java-serialized-object

Payload

- java.io.Serializable

Headers

- content-type: application/x-java-object

Payload

- Map

Options

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

jms.client-id

Client id for durable subscriptions. (**String, default: <none>**)

jms.destination

The destination from which to receive messages (queue or topic). (**String, default: <none>**)

jms.message-selector

A selector for messages; (**String, default: <none>**)

jms.session-transacted

True to enable transactions and select a DefaultMessageListenerContainer, false to select a SimpleMessageListenerContainer. (**Boolean, default: true**)

jms.subscription-durable

True for a durable subscription. (**Boolean, default: <none>**)

jms.subscription-name

The name of a durable or shared subscription. (**String, default: <none>**)

jms.subscription-shared

True for a shared subscription. (**Boolean, default: <none>**)

spring.jms.jndi-name

Connection factory JNDI name. When set, takes precedence to others connection factory auto-configurations. (**String, default: <none>**)

spring.jms.listener.acknowledge-mode

Acknowledge mode of the container. By default, the listener is transacted with automatic acknowledgment. (**AcknowledgeMode**, default: **<none>**, possible values: **AUTO,CLIENT,DUPS_OK**)

spring.jms.listener.auto-startup

Start the container automatically on startup. (**Boolean**, default: **true**)

spring.jms.listener.concurrency

Minimum number of concurrent consumers. (**Integer**, default: **<none>**)

spring.jms.listener.max-concurrency

Maximum number of concurrent consumers. (**Integer**, default: **<none>**)

spring.jms.pub-sub-domain

Whether the default destination type is topic. (**Boolean**, default: **false**)

**Note**

Spring boot broker configuration is used; refer to the [Spring Boot Documentation](#) for more information. The `spring.jms.*` properties above are also handled by the boot JMS support.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar jms-source.jar --jms.destination=
```

2.8 Load Generator Source

A source that sends generated data and dispatches it to the stream. This is to provide a method for users to identify the performance of Spring Cloud Data Flow in different environments and deployment types.

Input

N/A

Output**Headers:**

- Content-Type: application/octet-stream

Payload:

- byte[]

Options

The **load-generator** source has the following options:

load-generator.generate-timestamp
<documentation missing> (**Boolean, default: false**)

load-generator.message-count
<documentation missing> (**Integer, default: 1000**)

load-generator.message-size
<documentation missing> (**Integer, default: 1000**)

load-generator.producers
<documentation missing> (**Integer, default: 1**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar load-generator-source.jar
```

2.9 Mail Source

A source module that listens for Emails and emits the message body as a message payload.

Input

N/A

Output

Headers

- content-type: text/plain

Payload

- String

Options

The **mail** source supports the following configuration properties:

mail.charset
The charset for byte[] mail-to-string transformation. (**String, default: UTF-8**)

mail.delete

Set to true to delete email after download. **(Boolean, default: false)**

mail.expression

Configure a SpEL expression to select messages. **(String, default: true)**

mail.idle-imap

Set to true to use IdleImap Configuration. **(Boolean, default: false)**

mail.java-mail-properties

JavaMail properties as a new line delimited string of name-value pairs, e.g. 'foo=bar\n baz=car'.
(Properties, default: <none>)

mail.mark-as-read

Set to true to mark email as read. **(Boolean, default: false)**

mail.url

Mail connection URL for connection to Mail server e.g. 'imaps://username:password@imap.server.com:993/Inbox'. **(URLName, default: <none>)**

mail.user-flag

The flag to mark messages when the server does not support \Recent **(String, default: <none>)**

trigger.cron

Cron expression value for the Cron Trigger. **(String, default: <none>)**

trigger.date-format

Format for the date value. **(String, default: <none>)**

trigger.fixed-delay

Fixed delay for periodic triggers. **(Integer, default: 1)**

trigger.initial-delay

Initial delay for periodic triggers. **(Integer, default: 0)**

trigger.max-messages

Maximum messages per poll, -1 means infinity. **(Long, default: 1)**

trigger.time-unit

The TimeUnit to apply to delay values. **(TimeUnit, default: <none>, possible values: NANoseconds,MICROseconds,MILLIseconds,SECONDS,MINUTES,HOURS,DAYS)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar mail-source.jar --mail.javaMailProperties= --mail.url= --mail.expression= \
--mail.charset= --mail.userFlag=
```

2.10 MongoDB Source

This source polls data from MongoDB. This source is fully based on the `MongoDataAutoConfiguration`, so refer to the [Spring Boot MongoDB Support](#) for more information.

Input

N/A

Output

Headers:

- Content-Type: text/plain

Payload:

- String

Options

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

`mongodb.collection`

The MongoDB collection to query (**String**, default: **<none>**)

`mongodb.query`

The MongoDB query (**String**, default: **{ }**)

`mongodb.query-expression`

The SpEL expression in MongoDB query DSL style (**Expression**, default: **<none>**)

`mongodb.split`

Whether to split the query result as individual messages. (**Boolean**, default: **true**)

`spring.data.mongodb.authentication-database`

Authentication database name. (**String**, default: **<none>**)

`spring.data.mongodb.database`

Database name. (**String**, default: **<none>**)

`spring.data.mongodb.field-naming-strategy`

Fully qualified name of the `FieldNamingStrategy` to use. (**Class<?>**, default: **<none>**)

`spring.data.mongodb.grid-fs-database`

GridFS database name. (**String**, default: **<none>**)

`spring.data.mongodb.host`

Mongo server host. Cannot be set with URI. (**String**, default: **<none>**)

`spring.data.mongodb.password`

Login password of the mongo server. Cannot be set with URI. (**Character[]**, default: **<none>**)

`spring.data.mongodb.port`

Mongo server port. Cannot be set with URI. (**Integer**, default: **<none>**)

spring.data.mongodb.uri

Mongo database URI. Cannot be set with host, port and credentials. **(String, default: <none>)**

spring.data.mongodb.username

Login user of the mongo server. Cannot be set with URI. **(String, default: <none>)**

trigger.cron

Cron expression value for the Cron Trigger. **(String, default: <none>)**

trigger.date-format

Format for the date value. **(String, default: <none>)**

trigger.fixed-delay

Fixed delay for periodic triggers. **(Integer, default: 1)**

trigger.initial-delay

Initial delay for periodic triggers. **(Integer, default: 0)**

trigger.max-messages

Maximum messages per poll, -1 means infinity. **(Long, default: -1)**

trigger.time-unit

The TimeUnit to apply to delay values. **(TimeUnit, default: SECONDS, possible values: NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS)**

Also see the [Spring Boot Documentation](#) for additional MongoProperties properties. See and TriggerProperties for polling options.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar mongodb-source.jar --mongodb.query= --mongodb.collection=
```

2.11 MQTT Source

The "mqtt" source enables receiving messages from MQTT.

Input

N/A

Output

Headers:

Payload:

- String if binary setting is false (default)

- `byte[]` if binary setting is `true`

Options

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

`mqtt.binary`

true to leave the payload as bytes (**Boolean, default: false**)

`mqtt.charset`

the charset used to convert bytes to String (when binary is false) (**String, default: UTF-8**)

`mqtt.clean-session`

whether the client and server should remember state across restarts and reconnects (**Boolean, default: true**)

`mqtt.client-id`

identifies the client (**String, default: stream.client.id.source**)

`mqtt.connection-timeout`

the connection timeout in seconds (**Integer, default: 30**)

`mqtt.keep-alive-interval`

the ping interval in seconds (**Integer, default: 60**)

`mqtt.password`

the password to use when connecting to the broker (**String, default: guest**)

`mqtt.persistence`

'memory' or 'file' (**String, default: memory**)

`mqtt.persistence-directory`

Persistence directory (**String, default: /tmp/paho**)

`mqtt.qos`

the qos; a single value for all topics or a comma-delimited list to match the topics (**int[], default: [0]**)

`mqtt.topics`

the topic(s) (comma-delimited) to which the source will subscribe (**String[], default: [stream.mqtt]**)

`mqtt.url`

location of the mqtt broker(s) (comma-delimited list) (**String[], default: [tcp://localhost:1883]**)

`mqtt.username`

the username to use when connecting to the broker (**String, default: guest**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar mqtt-source.jar --mqtt.clientId=
```

2.12 RabbitMQ Source

The "rabbit" source enables receiving messages from RabbitMQ.

The queue(s) must exist before the stream is deployed; they are not created automatically. You can easily create a Queue using the RabbitMQ web UI.

Input

N/A

Output

Headers

- content-type: text/plain

Payload

- String

Headers

- content-type: application/octet-stream

Payload

- byte[]

Headers

- content-type: application/x-java-serialized-object

Payload

- java.io.Serializable

Options

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

rabbit.enable-retry

true to enable retry. (**Boolean, default: false**)

rabbit.initial-retry-interval

Initial retry interval when retry is enabled. (**Integer, default: 1000**)

rabbit.mapped-request-headers

Headers that will be mapped. (**String[], default: [STANDARD_REQUEST_HEADERS]**)

rabbit.max-attempts

The maximum delivery attempts when retry is enabled. **(Integer, default: 3)**

rabbit.max-retry-interval

Max retry interval when retry is enabled. **(Integer, default: 30000)**

rabbit.own-connection

When true, use a separate connection based on the boot properties. **(Boolean, default: false)**

rabbit.queues

The queues to which the source will listen for messages. **(String[], default: <none>)**

rabbit.queue

Whether rejected messages should be queued. **(Boolean, default: true)**

rabbit.retry-multiplier

Retry backoff multiplier when retry is enabled. **(Double, default: 2)**

rabbit.transacted

Whether the channel is transacted. **(Boolean, default: false)**

spring.rabbitmq.addresses

Comma-separated list of addresses to which the client should connect. **(String, default: <none>)**

spring.rabbitmq.connection-timeout

Connection timeout. Set it to zero to wait forever. **(Duration, default: <none>)**

spring.rabbitmq.host

RabbitMQ host. **(String, default: localhost)**

spring.rabbitmq.password

Login to authenticate against the broker. **(String, default: guest)**

spring.rabbitmq.port

RabbitMQ port. **(Integer, default: 5672)**

spring.rabbitmq.publisher-confirms

Whether to enable publisher confirms. **(Boolean, default: false)**

spring.rabbitmq.publisher-returns

Whether to enable publisher returns. **(Boolean, default: false)**

spring.rabbitmq.requested-heartbeat

Requested heartbeat timeout; zero for none. If a duration suffix is not specified, seconds will be used. **(Duration, default: <none>)**

spring.rabbitmq.username

Login user to authenticate to the broker. **(String, default: guest)**

spring.rabbitmq.virtual-host

Virtual host to use when connecting to the broker. **(String, default: <none>)**

Also see the [Spring Boot Documentation](#) for addition properties for the broker connections and listener properties.

A Note About Retry



Note

With the default *ackMode* (**AUTO**) and *requeue* (**true**) options, failed message deliveries will be retried indefinitely. Since there is not much processing in the rabbit source, the risk of failure in the source itself is small, unless the downstream Binder is not connected for some reason. Setting *requeue* to **false** will cause messages to be rejected on the first attempt (and possibly sent to a Dead Letter Exchange/Queue if the broker is so configured). The *enableRetry* option allows configuration of retry parameters such that a failed message delivery can be retried and eventually discarded (or dead-lettered) when retries are exhausted. The delivery thread is suspended during the retry interval(s). Retry options are *enableRetry*, *maxAttempts*, *initialRetryInterval*, *retryMultiplier*, and *maxRetryInterval*. Message deliveries failing with a *MessageConversionException* are never retried; the assumption being that if a message could not be converted on the first attempt, subsequent attempts will also fail. Such messages are discarded (or dead-lettered).

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar rabbit-source.jar --rabbit.queues=
```

2.13 Amazon S3 Source

This source app supports transfer of files using the Amazon S3 protocol. Files are transferred from the remote directory (S3 bucket) to the local directory where the application is deployed.

Messages emitted by the source are provided as a byte array by default. However, this can be customized using the `--mode` option:

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

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

Input

N/A

Output

mode = contents

Headers:

- Content-Type: application/octet-stream
- file_originalFile: <java.io.File>
- file_name: <file name>

Payload:

A `byte[]` filled with the file contents.

mode = lines

Headers:

- Content-Type: text/plain
- file_originalFile: <java.io.File>
- file_name: <file name>
- correlationId: <UUID> (same for each line)
- sequenceNumber: <n>
- sequenceSize: 0 (number of lines is not know until the file is read)

Payload:

A `String` for each line.

The first line is optionally preceded by a message with a START marker payload. The last line is optionally followed by a message with an END marker payload.

Marker presence and format are determined by the `with-markers` and `markers-json` properties.

mode = ref

Headers:

None.

Payload:

A `java.io.File` object.

Options

The **s3** source has the following options:

`file.consumer.markers-json`

When `'fileMarkers == true'`, specify if they should be produced as `FileSplitter.FileMarker` objects or JSON. (**Boolean, default: true**)

file.consumer.mode

The `FileReadingMode` to use for file reading sources. Values are 'ref' - The File object, 'lines' - a message per line, or 'contents' - the contents as bytes. (**FileReadingMode, default: <none>, possible values: ref,lines,contents**)

file.consumer.with-markers

Set to true to emit start of file/end of file marker messages before/after the data. Only valid with `FileReadingMode 'lines'`. (**Boolean, default: <none>**)

s3.auto-create-local-dir

Create or not the local directory. (**Boolean, default: true**)

s3.delete-remote-files

Delete or not remote files after processing. (**Boolean, default: false**)

s3.filename-pattern

The pattern to filter remote files. (**String, default: <none>**)

s3.filename-regex

The regex to filter remote files. (**Pattern, default: <none>**)

s3.local-dir

The local directory to store files. (**File, default: <none>**)

s3.preserve-timestamp

To transfer or not the timestamp of the remote file to the local one. (**Boolean, default: true**)

s3.remote-dir

AWS S3 bucket resource. (**String, default: bucket**)

s3.remote-file-separator

Remote File separator. (**String, default: /**)

s3.tmp-file-suffix

Temporary file suffix. (**String, default: .tmp**)

trigger.cron

Cron expression value for the Cron Trigger. (**String, default: <none>**)

trigger.date-format

Format for the date value. (**String, default: <none>**)

trigger.fixed-delay

Fixed delay for periodic triggers. (**Integer, default: 1**)

trigger.initial-delay

Initial delay for periodic triggers. (**Integer, default: 0**)

trigger.max-messages

Maximum messages per poll, -1 means infinity. (**Long, default: -1**)

trigger.time-unit

The `TimeUnit` to apply to delay values. (**TimeUnit, default: SECONDS, possible values: NANoseconds,MICROseconds,MILLIseconds,SECONDS,MINUTES,HOURS,DAYS**)

Amazon AWS common options

The Amazon S3 Source (as all other Amazon AWS applications) is based on the [Spring Cloud AWS](#) project as a foundation, and its auto-configuration classes are used automatically by Spring Boot. Consult their documentation regarding required and useful auto-configuration properties.

Some of them are about AWS credentials:

- `cloud.aws.credentials.accessKey`
- `cloud.aws.credentials.secretKey`
- `cloud.aws.credentials.instanceProfile`
- `cloud.aws.credentials.profileName`
- `cloud.aws.credentials.profilePath`

Other are for AWS Region definition:

- `cloud.aws.region.auto`
- `cloud.aws.region.static`

And for AWS Stack:

- `cloud.aws.stack.auto`
- `cloud.aws.stack.name`

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar s3-source.jar --s3.remoteDir=/tmp/foo --file.consumer.mode=lines --trigger.fixed-delay=60
```

2.14 SFTP Source

This source app supports transfer of files using the SFTP protocol. Files are transferred from the `remote` directory to the `local` directory where the application is deployed.

Messages emitted by the source are provided as a byte array by default. However, this can be customized using the `--mode` option:

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

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

When configuring the `sftp.factory.known-hosts-expression` option, the root object of the evaluation is the application context, an example might be `sftp.factory.known-hosts-expression = @systemProperties['user.home'] + '/.ssh/known_hosts'`.

Input

N/A (Fetches files from an SFTP server).

Output

mode = contents

Headers:

- Content-Type: application/octet-stream
- file_originalFile: <java.io.File>
- file_name: <file name>

Payload:

A `byte[]` filled with the file contents.

mode = lines

Headers:

- Content-Type: text/plain
- file_originalFile: <java.io.File>
- file_name: <file name>
- correlationId: <UUID> (same for each line)
- sequenceNumber: <n>
- sequenceSize: 0 (number of lines is not know until the file is read)

Payload:

A `String` for each line.

The first line is optionally preceded by a message with a START marker payload. The last line is optionally followed by a message with an END marker payload.

Marker presence and format are determined by the `with-markers` and `markers-json` properties.

mode = ref

Headers:

None.

Payload:

A `java.io.File` object.

Options

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

file.consumer.markers-json

When 'fileMarkers == true', specify if they should be produced as `FileSplitter.FileMarker` objects or JSON. **(Boolean, default: true)**

file.consumer.mode

The `FileReadingMode` to use for file reading sources. Values are 'ref' - The File object, 'lines' - a message per line, or 'contents' - the contents as bytes. **(FileReadingMode, default: <none>, possible values: ref,lines,contents)**

file.consumer.with-markers

Set to true to emit start of file/end of file marker messages before/after the data. Only valid with `FileReadingMode 'lines'`. **(Boolean, default: <none>)**

sftp.auto-create-local-dir

Set to true to create the local directory if it does not exist. **(Boolean, default: true)**

sftp.batch.batch-resource-uri

The URI of the batch artifact to be applied to the `TaskLaunchRequest`. **(String, default: <empty string>)**

sftp.batch.data-source-password

The datasource password to be applied to the `TaskLaunchRequest`. **(String, default: <none>)**

sftp.batch.data-source-url

The datasource url to be applied to the `TaskLaunchRequest`. Defaults to h2 in-memory JDBC datasource url. **(String, default: jdbc:h2:tcp://localhost:19092/mem:dataflow)**

sftp.batch.data-source-user-name

The datasource user name to be applied to the `TaskLaunchRequest`. Defaults to "sa" **(String, default: sa)**

sftp.batch.deployment-properties

Comma delimited list of deployment properties to be applied to the `TaskLaunchRequest`. **(String, default: <none>)**

sftp.batch.environment-properties

Comma delimited list of environment properties to be applied to the `TaskLaunchRequest`. **(String, default: <none>)**

sftp.batch.job-parameters

Comma separated list of optional job parameters in key=value format. **(List<String>, default: <none>)**

sftp.batch.local-file-path-job-parameter-name

Value to use as the local file job parameter name. Defaults to "localFilePath". **(String, default: localFilePath)**

sftp.batch.local-file-path-job-parameter-value

The file path to use as the local file job parameter value. Defaults to "java.io.tmpdir". (**String, default: <none>**)

sftp.batch.remote-file-path-job-parameter-name

Value to use as the remote file job parameter name. Defaults to "remoteFilePath". (**String, default: remoteFilePath**)

sftp.delete-remote-files

Set to true to delete remote files after successful transfer. (**Boolean, default: false**)

sftp.factory.allow-unknown-keys

True to allow an unknown or changed key. (**Boolean, default: false**)

sftp.factory.cache-sessions

Cache sessions (**Boolean, default: <none>**)

sftp.factory.host

The host name of the server. (**String, default: localhost**)

sftp.factory.known-hosts-expression

A SpEL expression resolving to the location of the known hosts file. (**Expression, default: <none>**)

sftp.factory.pass-phrase

Passphrase for user's private key. (**String, default: <empty string>**)

sftp.factory.password

The password to use to connect to the server. (**String, default: <none>**)

sftp.factory.port

The port of the server. (**Integer, default: 22**)

sftp.factory.private-key

Resource location of user's private key. (**String, default: <empty string>**)

sftp.factory.username

The username to use to connect to the server. (**String, default: <none>**)

sftp.filename-pattern

A filter pattern to match the names of files to transfer. (**String, default: <none>**)

sftp.filename-regex

A filter regex pattern to match the names of files to transfer. (**Pattern, default: <none>**)

sftp.list-only

Set to true to return file metadata without the entire payload. (**Boolean, default: false**)

sftp.local-dir

The local directory to use for file transfers. (**File, default: <none>**)

sftp.metadata.redis.key-name

The key name to use when storing file metadata. Defaults to "sftpSource". (**String, default: sftpSource**)

sftp.preserve-timestamp

Set to true to preserve the original timestamp. (**Boolean, default: true**)

sftp.remote-dir

The remote FTP directory. **(String, default: /)**

sftp.remote-file-separator

The remote file separator. **(String, default: /)**

sftp.stream

Set to true to stream the file rather than copy to a local directory. **(Boolean, default: false)**

sftp.task-launcher-output

Set to true to create output suitable for a task launch request. **(Boolean, default: false)**

sftp.tmp-file-suffix

The suffix to use while the transfer is in progress. **(String, default: .tmp)**

trigger.cron

Cron expression value for the Cron Trigger. **(String, default: <none>)**

trigger.date-format

Format for the date value. **(String, default: <none>)**

trigger.fixed-delay

Fixed delay for periodic triggers. **(Integer, default: 1)**

trigger.initial-delay

Initial delay for periodic triggers. **(Integer, default: 0)**

trigger.max-messages

Maximum messages per poll, -1 means infinity. **(Long, default: -1)**

trigger.time-unit

The TimeUnit to apply to delay values. **(TimeUnit, default: SECONDS, possible values: NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar sftp_source.jar --sftp.remote-dir=foo --file.consumer.mode=lines --trigger.fixed-delay=60 \
--sftp.factory.host=sftpserver --sftp.factory.username=user --sftp.factory.password=pw --
sftp.local-dir=/foo
```

2.15 SYSLOG Source

The syslog source receives SYSLOG packets over UDP, TCP, or both. RFC3164 (BSD) and RFC5424 formats are supported.

Input

N/A

Output

Headers

- `content-type: application/json`

Payload

- Map of field/values

Options

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

`syslog.buffer-size`

the buffer size used when decoding messages; larger messages will be rejected. (**Integer, default: 2048**)

`syslog.nio`

whether or not to use NIO (when supporting a large number of connections). (**Boolean, default: false**)

`syslog.port`

The port to listen on. (**Integer, default: 1514**)

`syslog.protocol`

tcp or udp (**String, default: tcp**)

`syslog.reverse-lookup`

whether or not to perform a reverse lookup on the incoming socket. (**Boolean, default: false**)

`syslog.rfc`

'5424' or '3164' - the syslog format according the the RFC; 3164 is aka 'BSD' format. (**String, default: 3164**)

`syslog.socket-timeout`

the socket timeout. (**Integer, default: 0**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar syslog-source.jar --syslog.rfc=5424 --syslog.protocol=tcp
```

2.16 TCP

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

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.

Messages produced by the TCP source application have a `byte[]` payload.

Input

N/A

Output

Headers:

- Content-Type: `application/octet-stream`

Payload:

- `byte[]`

Options

`tcp.buffer-size`

The buffer size used when decoding messages; larger messages will be rejected. (**Integer, default: 2048**)

`tcp.decoder`

The decoder to use when receiving messages. (**Encoding, default: <none>, possible values: CRLF,LF,NULL,STXETX,RAW,L1,L2,L4**)

`tcp.nio`

Whether or not to use NIO. (**Boolean, default: false**)

`tcp.port`

The port on which to listen; 0 for the OS to choose a port. (**Integer, default: 1234**)

`tcp.reverse-lookup`

Perform a reverse DNS lookup on the remote IP Address; if false, just the IP address is included in the message headers. (**Boolean, default: false**)

`tcp.socket-timeout`

The timeout (ms) before closing the socket when no data is received. (**Integer, default: 120000**)

`tcp.use-direct-buffers`

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)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar tcp_source.jar --tcp.decoder=LF
```

The "decoder" property determines the message format (default is termination with CRLF).

2.17 TCP Client as a Source which connects to a TCP server and receives data

Input

N/A

Output**Headers:**

- Content-Type: application/octet-stream

Payload:

- byte[]

Options

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

tcp.buffer-size

The buffer size used when decoding messages; larger messages will be rejected. **(Integer, default: 2048)**

tcp.charset

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

tcp.decoder

The decoder to use when receiving messages. **(Encoding, default: <none>, possible values: CRLF,LF,NULL,STXETX,RAW,L1,L2,L4)**

tcp.host

The host to which this client will connect. **(String, default: localhost)**

tcp.nio

Whether or not to use NIO. **(Boolean, default: false)**

tcp.port

The port on which to listen; 0 for the OS to choose a port. **(Integer, default: 1234)**

tcp.retry-interval

Retry interval (in milliseconds) to check the connection and reconnect. **(Long, default: 60000)**

tcp.reverse-lookup

Perform a reverse DNS lookup on the remote IP Address; if false, just the IP address is included in the message headers. **(Boolean, default: false)**

tcp.socket-timeout

The timeout (ms) before closing the socket when no data is received. **(Integer, default: 120000)**

tcp.use-direct-buffers

Whether or not to use direct buffers. **(Boolean, default: false)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar tcp_client_source.jar --tcp.decoder=LF
```

2.18 Time Source

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

Input

N/A

Output

Headers:

- Content-Type: text/plain

Payload:

A String with the time output.

Options

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

trigger.cron

Cron expression value for the Cron Trigger. (**String, default: <none>**)

trigger.date-format

Format for the date value. (**String, default: <none>**)

trigger.fixed-delay

Fixed delay for periodic triggers. (**Integer, default: 1**)

trigger.initial-delay

Initial delay for periodic triggers. (**Integer, default: 0**)

trigger.max-messages

Maximum messages per poll, -1 means infinity. (**Long, default: 1**)

trigger.time-unit

The TimeUnit to apply to delay values. (**TimeUnit, default: <none>, possible values: NANoseconds,MICROseconds,MILLIseconds,SECONDS,MINUTES,HOURS,DAYS**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar time-source.jar
```

2.19 Trigger Source

This app sends trigger based on a fixed delay, date or cron expression. A payload which is evaluated using SpEL can also be sent each time the trigger fires.

Input

N/A

Output

Headers:

Payload:

- Any

Options

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

trigger.cron

Cron expression value for the Cron Trigger. (**String**, default: <none>)

trigger.date-format

Format for the date value. (**String**, default: <none>)

trigger.fixed-delay

Fixed delay for periodic triggers. (**Integer**, default: 1)

trigger.initial-delay

Initial delay for periodic triggers. (**Integer**, default: 0)

trigger.max-messages

Maximum messages per poll, -1 means infinity. (**Long**, default: 1)

trigger.source.payload

The expression for the payload of the Source module. (**Expression**, default: <none>)

trigger.time-unit

The TimeUnit to apply to delay values. (**TimeUnit**, default: <none>, possible values: **NANOSECONDS,MICROSECONDS,MILLISECONDS,SECONDS,MINUTES,HOURS,DAYS**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar trigger_source.jar --trigger.source.payload=payload-expression
```

2.20 TriggerTask Source

The TriggerTask app sends a TaskLaunchRequest based on a fixed delay, date or cron expression. The TaskLaunchRequest is used by a tasklauncher-* sink that will deploy and launch a task. The only

required property for the `triggertask` is the `--uri` which specifies the artifact that will be launched by the `tasklauncher-*` that you have selected. The user is also allowed to set the command line arguments as well as the [Spring Boot properties](#) that are used by the task.

Input

N/A

Output

Headers:

- Content-Type: `application/octet-stream`

Payload:

A byte array containing the `TaskLaunchRequest`

Options

The **`triggertask`** source has the following options:

`trigger.cron`

Cron expression value for the Cron Trigger. **(String, default: <none>)**

`trigger.date-format`

Format for the date value. **(String, default: <none>)**

`trigger.fixed-delay`

Fixed delay for periodic triggers. **(Integer, default: 1)**

`trigger.initial-delay`

Initial delay for periodic triggers. **(Integer, default: 0)**

`trigger.max-messages`

Maximum messages per poll, -1 means infinity. **(Long, default: 1)**

`trigger.source.payload`

The expression for the payload of the Source module. **(Expression, default: <none>)**

`trigger.time-unit`

The `TimeUnit` to apply to delay values. **(TimeUnit, default: <none>, possible values: NANSECONDS,MICROSECONDS,MILLISECONDS,SECONDS,MINUTES,HOURS,DAYS)**

`triggertask.application-name`

The name to be applied to the launched task.. **(String, default: <empty string>)**

`triggertask.command-line-args`

Space delimited key=value pairs to be used as commandline variables for the task. **(String, default: <empty string>)**

`triggertask.deployment-properties`

Comma delimited key=value pairs to be used as `deploymentProperties` for the task. **(String, default: <empty string>)**

triggertask.environment-properties

Comma delimited key=value pairs to be used as environmentProperties for the task. (**String**, **default: <empty string>**)

triggertask.uri

The uri to the task artifact. (**String**, **default: <empty string>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar trigger_task.jar --triggertask.uri=maven://org.springframework.cloud.task.app:timestamp-task:1.2.0.RELEASE
```

2.21 Twitter Stream Source

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

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

Input

N/A

Output

Headers

- Content-Type: text/plain

Payload

- String

Options

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

twitter.credentials.access-token

Access token (**String**, **default: <none>**)

twitter.credentials.access-token-secret

Access token secret (**String**, **default: <none>**)

twitter.credentials.consumer-key

Consumer key (**String**, default: <none>)

twitter.credentials.consumer-secret

Consumer secret (**String**, default: <none>)

twitter.stream.follow

A comma separated list of user IDs, indicating the users to return statuses for in the stream. (**String**, default: <none>)

twitter.stream.language

The language of the tweet text. (**String**, default: <none>)

twitter.stream.locations

A set of bounding boxes to track. (**String**, default: <none>)

twitter.stream.stream-type

Twitter stream type (such as sample, firehose). Default is sample. (**TwitterStreamType**, default: <none>, possible values: **SAMPLE,FIREHOSE,FILTER**)

twitter.stream.track

Keywords to track. (**String**, default: <none>)



Note

twitterstream emit JSON in the [native Twitter format](#).

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar twitter_stream_source.jar --twitter.credentials.consumerKey=<CONSUMER_KEY> --
twitter.credentials.consumerSecret=<CONSUMER_SECRET> \
--twitter.credentials.accessToken=<ACCESS_TOKEN> --
twitter.credentials.accessTokenSecret=<ACCESS_TOKEN_SECRET>
```

3. Processors

3.1 Aggregator Processor

Use the aggregator application to combine multiple messages into one, based on some correlation mechanism.

This processor is fully based on the Aggregator component from [Spring Integration](#). So, please, consult there for use-cases and functionality.

Input

Headers

If the aggregation and correlation logic is based on the default strategies, the `correlationId`, `sequenceNumber` and `sequenceSize` headers must be presented in the incoming message.

Payload

Aggregator Processor is fully based on the Spring Integration's `AggregatingMessageHandler` and since correlation and aggregation logic don't require particular types, the input payload can be anything able to be transferred over the network and Spring Cloud Stream Binder. If payload is JSON, the `JsonPropertyAccessor` helps to build straightforward SpEL expressions for correlation, release and aggregation functions.

Output

Headers

Returns all headers of the incoming messages that have no conflicts among the group. An absent header on one or more messages within the group is not considered a conflict.

Payload

By default the `DefaultAggregatingMessageGroupProcessor` is used for aggregation function with meaning return the `java.util.List` of payloads of incoming messages. The custom aggregation SpEL expression may produce any required object to be sent to the output of the processor.

Options

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

`aggregator.aggregation`

SpEL expression for aggregation strategy. Default is collection of payloads (**Expression, default: <none>**)

`aggregator.correlation`

SpEL expression for correlation key. Default to `correlationId` header (**Expression, default: <none>**)

`aggregator.group-timeout`

SpEL expression for timeout to expiring uncompleted groups (**Expression, default: <none>**)

`aggregator.message-store-entity`

Persistence message store entity: table prefix in RDBMS, collection name in MongoDB, etc (**String, default: <none>**)

- aggregator.message-store-type
Message store type (**String**, default: **<none>**)
- aggregator.release
SpEL expression for release strategy. Default is based on the sequenceSize header (**Expression**, default: **<none>**)
- spring.data.mongodb.authentication-database
Authentication database name. (**String**, default: **<none>**)
- spring.data.mongodb.database
Database name. (**String**, default: **<none>**)
- spring.data.mongodb.field-naming-strategy
Fully qualified name of the FieldNamingStrategy to use. (**Class<?>**, default: **<none>**)
- spring.data.mongodb.grid-fs-database
GridFS database name. (**String**, default: **<none>**)
- spring.data.mongodb.host
Mongo server host. Cannot be set with URI. (**String**, default: **<none>**)
- spring.data.mongodb.password
Login password of the mongo server. Cannot be set with URI. (**Character[]**, default: **<none>**)
- spring.data.mongodb.port
Mongo server port. Cannot be set with URI. (**Integer**, default: **<none>**)
- spring.data.mongodb.uri
Mongo database URI. Cannot be set with host, port and credentials. (**String**, default: **mongodb://localhost/test**)
- spring.data.mongodb.username
Login user of the mongo server. Cannot be set with URI. (**String**, default: **<none>**)
- spring.datasource.continue-on-error
Whether to stop if an error occurs while initializing the database. (**Boolean**, default: **false**)
- spring.datasource.data
Data (DML) script resource references. (**List<String>**, default: **<none>**)
- spring.datasource.data-password
Password of the database to execute DML scripts (if different). (**String**, default: **<none>**)
- spring.datasource.data-username
Username of the database to execute DML scripts (if different). (**String**, default: **<none>**)
- spring.datasource.driver-class-name
Fully qualified name of the JDBC driver. Auto-detected based on the URL by default. (**String**, default: **<none>**)
- spring.datasource.generate-unique-name
Whether to generate a random datasource name. (**Boolean**, default: **false**)
- spring.datasource.initialization-mode
Initialize the datasource with available DDL and DML scripts. (**DataSourceInitializationMode**, default: **embedded**, possible values: **ALWAYS,EMBEDDED,NEVER**)

spring.datasource.jndi-name

JNDI location of the datasource. Class, url, username & password are ignored when set. **(String, default: <none>)**

spring.datasource.name

Name of the datasource. Default to "testdb" when using an embedded database. **(String, default: <none>)**

spring.datasource.password

Login password of the database. **(String, default: <none>)**

spring.datasource.platform

Platform to use in the DDL or DML scripts (such as schema-`${platform}.sql` or data-`${platform}.sql`). **(String, default: all)**

spring.datasource.schema

Schema (DDL) script resource references. **(List<String>, default: <none>)**

spring.datasource.schema-password

Password of the database to execute DDL scripts (if different). **(String, default: <none>)**

spring.datasource.schema-username

Username of the database to execute DDL scripts (if different). **(String, default: <none>)**

spring.datasource.separator

Statement separator in SQL initialization scripts. **(String, default: ;)**

spring.datasource.sql-script-encoding

SQL scripts encoding. **(Charset, default: <none>)**

spring.datasource.type

Fully qualified name of the connection pool implementation to use. By default, it is auto-detected from the classpath. **(Class<DataSource>, default: <none>)**

spring.datasource.url

JDBC URL of the database. **(String, default: <none>)**

spring.datasource.username

Login username of the database. **(String, default: <none>)**

spring.mongodb.embedded.features

Comma-separated list of features to enable. **(Set<Feature>, default: [sync_delay])**

spring.mongodb.embedded.version

Version of Mongo to use. **(String, default: 3.2.2)**

spring.redis.database

Database index used by the connection factory. **(Integer, default: 0)**

spring.redis.host

Redis server host. **(String, default: localhost)**

spring.redis.password

Login password of the redis server. **(String, default: <none>)**

spring.redis.port

Redis server port. **(Integer, default: 6379)**

spring.redis.ssl

Whether to enable SSL support. **(Boolean, default: false)**

spring.redis.timeout

Connection timeout. **(Duration, default: <none>)**

spring.redis.url

Connection URL. Overrides host, port, and password. User is ignored. Example: redis://user:password@example.com:6379 **(String, default: <none>)**

By default the aggregator processor uses: - HeaderAttributeCorrelationStrategy(IntegrationMessageHeaderAccessor.CORRELATION_ID) - for correlation; - SequenceSizeReleaseStrategy - for release; - DefaultAggregatingMessageGroupProcessor - for aggregation; - SimpleMessageStore - for messageStoreType.

The aggregator application can be configured for persistent MessageGroupStore [implementations](#). The configuration for target technology is fully based on the Spring Boot auto-configuration. But default JDBC, MongoDB and Redis auto-configurations are excluded. They are @Import ed basing on the aggregator.messageStoreType configuration property. Consult Spring Boot [Reference Manual](#) for auto-configuration for particular technology you use for aggregator.

The JDBC JdbcMessageStore requires particular tables in the target data base. You can find schema scripts for appropriate RDBMS vendors in the org.springframework.integration.jdbc package of the spring-integration-jdbc jar. Those scripts can be used for automatic data base initialization via Spring Boot.

For example:

```
java -jar aggregator-rabbit-1.0.0.RELEASE.jar
  --aggregator.message-store-type=jdbc
  --spring.datasource.url=jdbc:h2:mem:test
  --spring.datasource.schema=org/springframework/integration/jdbc/schema-h2.sql
```

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar aggregator_processor.jar
  --aggregator.message-store-type=jdbc
  --spring.datasource.url=jdbc:h2:mem:test
  --spring.datasource.schema=org/springframework/integration/jdbc/schema-h2.sql

java -jar aggregator_processor.jar
  --spring.data.mongodb.port=0
  --aggregator.correlation=T(Thread).currentThread().id
  --aggregator.release="!#this.[payload == 'bar'].empty"
  --aggregator.aggregation="!#this.[payload == 'foo'].![payload]"
  --aggregator.message-store-type=mongodb
  --aggregator.message-store-entity=aggregatorTest
```

Code of Conduct

This project adheres to the Contributor Covenant [code of conduct](#). By participating, you are expected to uphold this code. Please report unacceptable behavior to spring-code-of-conduct@pivotal.io.

3.2 Bridge Processor

A Processor module that returns messages that is passed by connecting just the input and output channels.

Input

Headers

Payload

Any

Output

Headers

Payload

Any

Options

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar bridge-processor.jar
```

3.3 Filter Processor

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

Input

Headers

N/A

Payload

Any

Output

Headers

N/A

Payload

Any

Options

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

filter.expression

A SpEL expression to be evaluated against each message, to decide whether or not to accept it.
(Expression, default: true)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar filter-processor.jar --expression="payload"
```

3.4 Groovy Filter Processor

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

Input

Headers

N/A

Payload

- Any

Output

Headers

N/A

Payload

- Any

Options

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

groovy-filter.script

The resource location of the groovy script (**Resource, default: <none>**)

groovy-filter.variables

Variable bindings as a new line delimited string of name-value pairs, e.g. 'foo=bar\n baz=car'.
(**Properties, default: <none>**)

groovy-filter.variables-location

The location of a properties file containing custom script variable bindings. (**Resource, default: <none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar groovy-filter-processor.jar --script=script.groovy
```

3.5 Groovy Transform Processor

A Processor module that transforms messages using a Groovy script.

Input

Headers

N/A

Payload

- Any

Output

Headers

N/A

Payload

- Any

Options

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

groovy-transformer.script

Reference to a script used to process messages. **(Resource, default: <none>)**

groovy-transformer.variables

Variable bindings as a new line delimited string of name-value pairs, e.g. 'foo=bar\n baz=car'.
(Properties, default: <none>)

groovy-transformer.variables-location

The location of a properties file containing custom script variable bindings. **(Resource, default: <none>)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar groovy-transform-processor.jar --script=script.groovy
```

3.6 gRPC Processor

This processor uses gRPC to process Messages via a remote process written in any language that supports gRPC. This pattern, allows the Java app to handle the stream processing while the gRPC service handles the business logic. The service must implement the [gRPC](#) service using link: `./grpc-app-protos/src/main/proto/processor.proto` [this protobuf schema].



Note

The gRPC client stub is blocking by default. Asynchronous and streaming stubs are provided. The Asynchronous stub will perform better if the server is multi-threaded however message ordering will not be guaranteed. If the server supports bidirectional streaming, use the streaming stub.



Note

A riff stub is available for interoperability with [riff](#) function containers. This does not interact with the Riff FaaS platform but supports running an existing function container standalone, for example, `docker run -it -p10382:10382 some/riff-function:latest`.

Input

Headers

Headers are available to the sidecar application via the [process](#) schema if `grpc.include-headers` is `true`. The header value contains one or more string values to support multiple values, e.g., the HTTP Accepts header.

Payload

The payload is a byte array as defined by the [schema](#).

Output

Headers

In most cases the return message should simply contain the original headers provided. The sidecar application may modify or add headers however it is recommended to only add headers if necessary.

Payload

It is expected that the payload will normally be a string or byte array. However common primitive types are supported as defined by the [schema](#).

Options

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

grpc.host

The gRPC host name. (**String, default: <none>**)

grpc.idle-timeout

The idle timeout in seconds. (**Long, default: 0**)

grpc.include-headers

Flag to include headers in Messages to the remote process. (**Boolean, default: false**)

grpc.max-message-size

The maximum message size (bytes). (**Integer, default: 0**)

grpc.plain-text

Flag to send messages in plain text. SSL configuration required otherwise. (**Boolean, default: true**)

grpc.port

The gRPC server port. (**Integer, default: 0**)

grpc.stub

RPC communications style (default 'blocking'). (**Stub, default: <none>, possible values: async,blocking,streaming,riff**)

3.7 Header Enricher Processor

Use the header-enricher app to add message headers.

The headers are provided in the form of new line delimited key value pairs, where the keys are the header names and the values are SpEL expressions. For example -- headers='foo=payload.someProperty \n bar=payload.otherProperty'

Input

Headers

N/A

Payload

- Any

Output

Headers

N/A

Payload

- Any

Options

The **header-enricher** processor has the following options:

header.enricher.headers

\n separated properties representing headers in which values are SpEL expressions, e.g foo='bar'
\n baz=payload.baz (**Properties, default: <none>**)

header.enricher.overwrite

set to true to overwrite any existing message headers (**Boolean, default: false**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar header-enricher-processor.jar --headers='foo=payload.someProperty \n
bar=payload.otherProperty'
```

Code of Conduct

This project adheres to the Contributor Covenant [code of conduct](#). By participating, you are expected to uphold this code. Please report unacceptable behavior to spring-code-of-conduct@pivotal.io.

3.8 Http Client Processor

A processor app that makes requests to an HTTP resource and emits the response body as a message payload. This processor can be combined, e.g., with a time source app to periodically poll results from a HTTP resource.

Input

Headers

Any Required HTTP headers must be explicitly set via the headers-expression property. See examples below. Header values may also be used to construct the request body when referenced in the body-expression property.

Payload

You can set the `http-method-expression` property to derive the HTTP method from the inbound Message, or `http-method` to set it statically (defaults to GET method). The Message payload may be any Java type. Generally, standard Java types such as String (e.g., JSON, XML) or byte array payloads are recommended. A Map should work without too much effort. By default, the payload will become HTTP request body (if needed). You may also set the `body-expression` property to construct a value derived from the Message, or `body` to use a static (literal) value.

Internally, the processor uses [RestTemplate.exchange\(...\)](#).

The RestTemplate supports Jackson JSON serialization to support any request and response types if necessary. The `expected-response-type` property, `String.class` by default, may be set to any class in your application class path. (Note user defined payload types will require adding required dependencies to your pom file)

Output

Headers

No HTTP message headers are mapped to the outbound Message.

Payload

The raw output object is [ResponseEntity<?>](#) any of its fields (e.g., `body`, `headers`) or accessor methods (`statusCode`) may be referenced as part of the `reply-expression`. By default the outbound Message payload is the response body.

Options

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

httpClient.body

The (static) request body; if neither this nor `bodyExpression` is provided, the payload will be used.
(Object, default: <none>)

httpClient.body-expression

A SpEL expression to derive the request body from the incoming message. **(Expression, default: <none>)**

httpClient.expected-response-type

The type used to interpret the response. **(Class<?>, default: <none>)**

httpClient.headers-expression

A SpEL expression used to derive the http headers map to use. **(Expression, default: <none>)**

httpClient.http-method

The kind of http method to use. **(HttpMethod, default: <none>, possible values: GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE)**

httpClient.http-method-expression

A SpEL expression to derive the request method from the incoming message. **(Expression, default: <none>)**

httpClient.reply-expression

A SpEL expression used to compute the final result, applied against the whole http response.
(Expression, default: body)

httpClient.url

The URL to issue an http request to, as a static value. **(String, default: <none>)**

httpClient.url-expression

A SpEL expression against incoming message to determine the URL to use. **(Expression, default: <none>)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
$ java -jar httpclient-processor.jar --httpClient.url=http://someurl --httpClient.http-method=POST --
httpClient.headers-expression="{ 'Content-Type': 'application/json' }"

$ java -jar httpclient-processor.jar --httpClient.url=http://someurl --httpClient.reply-
expression="statusCode.name()"
```

3.9 PMML Processor

A processor that evaluates a machine learning model stored in PMML format.

Input

Headers

N/A

Payload

- PMML model data

Output

Headers

N/A

Payload

- Tuple carrying information about the evaluated data

Options

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

pmml.inputs

How to compute model active fields from input message properties as `modelField->SpEL`.
(Map<String, Expression>, default: <none>)

pmml.model-location

The location of the PMML model file. **(Resource, default: <none>)**

pmml.model-name

If the model file contains multiple models, the name of the one to use. **(String, default: <none>)**

pmml.model-name-expression

If the model file contains multiple models, the name of the one to use, as a SpEL expression.
(Expression, default: <none>)

pmml.outputs

How to emit evaluation results in the output message as `msgProperty->SpEL`. **(Map<String, Expression>, default: <none>)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar pmml-processor.jar --pmml.modelLocation= --pmml.modelName="
java -jar pmml-processor.jar --pmml.modelLocation= --pmml.modelNameExpression="
```

3.10 Python Http Processor

Spring Cloud Stream App Starters for integrating with python

This application invokes a REST service, similar to the standard [httpclient processor](#). In fact, this application embeds the [httpclient processor](#). As a convenience for Python developers, this allows you to provide a Jython wrapper script that may execute a function before and after REST call in order to perform any necessary data transformation. If you don't require any custom transformations, just use the [httpclient processor](#).

The diagram shows input and output adapters as conceptual components. These are actually implemented as functions defined in a single script that must conform to a simple convention:

```
def input():
    return "Pre" + payload;

def output():
    return payload + "Post";

result = locals()[channel]()
```

The function names `input` and `output` map to the conventional channel names used by Spring Cloud Stream processors. The last line is a bit of Python reflection magic to invoke a function by its name, given by the bound variable `channel`. Implemented with Spring Integration Scripting, headers and

payload are always bound to the Message headers and payload respectively. The payload on the input side is the object you use to build the REST request. The output side transforms the response. If you don't need any additional processing on one side, implement the function with `pass` as the body:

```
def output():
    pass
```



Note

The last line in the script must be an assignment statement. The variable name doesn't matter. This is required to bind the return value correctly.



Note

The script is evaluated for every message. This tends to create a lot of classes for each execution which puts stress on the JRE Metaspace memory region (or Permgen if using a JRE prior to version 8). In Java 8, Metaspace is unlimited by default, allocated from native memory, and therefore limited by the native OS. If deploying to CloudFoundry, the Java Buildpack Memory Calculator sets `-XXMaxMetaspaceSize`. (see github.com/cloudfoundry/java-buildpack-memory-calculator for details). If using JBP v4.x, you may override the calculated value (and others) by specifying `-XXMaxMetaspaceSize` explicitly in `JAVA_OPTS`. You also need to increase the container memory accordingly. Similar tuning is advised in any containerized environment.

Input

Headers

Headers will be bound automatically to the wrapper script variable `headers`.

Payload

Any type. Payload will be automatically bound to the wrapper script variable `payload`. Jython scripts can effectively access any Java type on the app's classpath.

Output

Headers

Headers may be set by the Jython wrapper script if the `output()` script function returns a Message.

Payload

Whatever the `output()` wrapper script function returns.



Note

The wrapper script is intended to perform some required transformations prior to sending an HTTP request and/or after the response is received. The return value of the input adapter will be the inbound payload of the [httpclient processor](#) and should conform to its requirements. Likewise the HTTP reply-expression will bound to be the payload when the `output()` function is invoked.

Options

The **python-http** processor has the following options:

`git.basedir`

The base directory where the repository should be cloned. If not specified, a temporary directory will be created. **(File, default: <none>)**

`git.clone-on-start`

Flag to indicate that the repository should be cloned on startup (not on demand). Generally leads to slower startup but faster first query. **(Boolean, default: true)**

`git.label`

The label or branch to clone. **(String, default: master)**

`git.passphrase`

The passphrase for the remote repository. **(String, default: <none>)**

`git.password`

The password for the remote repository. **(String, default: <none>)**

`git.timeout`

Timeout (in seconds) for obtaining HTTP or SSH connection (if applicable). Default 5 seconds. **(Integer, default: 5)**

`git.uri`

The URI of the remote repository. **(String, default: <none>)**

`git.username`

The username for the remote repository. **(String, default: <none>)**

`httpclient.body`

The (static) request body; if neither this nor `bodyExpression` is provided, the payload will be used. **(Object, default: <none>)**

`httpclient.body-expression`

A SpEL expression to derive the request body from the incoming message. **(Expression, default: <none>)**

`httpclient.expected-response-type`

The type used to interpret the response. **(Class<?>, default: <none>)**

`httpclient.headers-expression`

A SpEL expression used to derive the http headers map to use. **(Expression, default: <none>)**

`httpclient.http-method`

The kind of http method to use. **(HttpMethod, default: <none>, possible values: GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS,TRACE)**

`httpclient.reply-expression`

A SpEL expression used to compute the final result, applied against the whole http response. **(Expression, default: body)**

`httpclient.url`

The URL to issue an http request to, as a static value. **(String, default: <none>)**

`httpClient.url-expression`

A SpEL expression against incoming message to determine the URL to use. (**Expression, default: <none>**)

`wrapper.content-type`

Sets the Content type header for the outgoing Message. (**MediaType, default: <none>**)

`wrapper.delimiter`

The variable delimiter. (**Delimiter, default: <none>, possible values: COMMA,SPACE,TAB,NEWLINE**)

`wrapper.script`

The Python script file name. (**String, default: <none>**)

`wrapper.variables`

Variable bindings as a delimited string of name-value pairs, e.g. 'foo=bar,baz=car'. (**String, default: <none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

See [httpClient processor](#) for more examples on `httpClient` properties.

```
$java -jar python-http-processor.jar --wrapper.script=/local/directory/build-json.py --
httpClient.url=http://someurl
--httpClient.http-method=POST --httpClient.headers-expression="{ 'Content-Type': 'application/json' }"

$java -jar python-http-processor.jar --git.uri=https://github.com/some-repo --wrapper.script=some-
script.py --wrapper
.variables=foo=0.45,bar=0.55 --httpClient.url=http://someurl
```

3.11 Jython Processor

This application executes a Jython script that binds payload and headers variables to the Message payload and headers respectively. In addition you may provide a `jython.variables` property containing a (comma delimited by default) delimited string, e.g., `var1=val1,var2=val2,...`

This processor uses a JSR-223 compliant embedded ScriptEngine provided by www.jython.org/.



Note

The last line in the script must be an assignment statement. The variable name doesn't matter. This is required to bind the return value correctly.



Note

The script is evaluated for every message which may limit your performance with high message loads. This also tends to create a lot of classes for each execution which puts stress on the JRE Metaspace memory region (or Permgen if using a JRE prior to version 8). In Java

8, Metaspace is unlimited by default, allocated from native memory, and therefore limited by the native OS. If deploying to CloudFoundry, the Java Buildpack Memory Calculator sets `-XXMaxMetaspaceSize`. (see github.com/cloudfoundry/java-buildpack-memory-calculator for details). If using JBP v4.x, you may override the calculated value (and others) by specifying `-XXMaxMetaspaceSize` explicitly in `JAVA_OPTS`. You also need to increase the container memory accordingly. Similar tuning is advised in any containerized environment.

Input

Headers

Headers will be bound automatically to the script variable `headers`.

Payload

Any type. Payload will be automatically bound to the script variable `payload`.

Output

Headers

Headers may be set by the Jython script if the script returns a `Message`.

Payload

Whatever the script returns.

Options

The `jython` processor has the following options:

`git.basedir`

The base directory where the repository should be cloned. If not specified, a temporary directory will be created. **(File, default: <none>)**

`git.clone-on-start`

Flag to indicate that the repository should be cloned on startup (not on demand). Generally leads to slower startup but faster first query. **(Boolean, default: true)**

`git.label`

The label or branch to clone. **(String, default: master)**

`git.passphrase`

The passphrase for the remote repository. **(String, default: <none>)**

`git.password`

The password for the remote repository. **(String, default: <none>)**

`git.timeout`

Timeout (in seconds) for obtaining HTTP or SSH connection (if applicable). Default 5 seconds. **(Integer, default: 5)**

`git.uri`

The URI of the remote repository. **(String, default: <none>)**

git.username

The username for the remote repository. (**String, default: <none>**)

jython.content-type

Sets the Content type header for the outgoing Message. (**MediaType, default: <none>**)

jython.delimiter

The variable delimiter. (**Delimiter, default: <none>, possible values: COMMA,SPACE,TAB,NEWLINE**)

jython.script

The Python script file name. (**String, default: <none>**)

jython.variables

Variable bindings as a delimited string of name-value pairs, e.g. 'foo=bar,baz=car'. (**String, default: <none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
$java -jar python-jython-processor.jar --jython.script=/local/directory/to_uppercase.py

$java -jar python-jython-processor.jar --git.uri=https://github.com/some-repo --jython
.script=map-tweet-sentiments.py --jython.variables=neutral=0.45,positive=0.55
```

3.12 Scribable Transform Processor

A **Spring Cloud Stream** module that transforms messages using a script. The script body is supplied directly as a property value. The language of the script can be specified (groovy/javascript/ruby/python).

Input

Headers

N/A

Payload

- Any

Output

Headers

N/A

Payload

- Any

Options

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

`scriptable-transformer.language`

Language of the text in the script property. Supported: groovy, javascript, ruby, python. (**String**, **default: <none>**)

`scriptable-transformer.script`

Text of the script. (**String**, **default: <none>**)

`scriptable-transformer.variables`

Variable bindings as a new line delimited string of name-value pairs, e.g. 'foo=bar\n baz=car'. (**Properties**, **default: <none>**)

`scriptable-transformer.variables-location`

The location of a properties file containing custom script variable bindings. (**Resource**, **default: <none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar scriptable-transform-processor.jar --language=ruby --script="return '#{payload.upcase}'"
```

3.13 Splitter Processor

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

Input

Headers

N/A

Payload

- Any

Output

Headers

N/A

Payload

- A collection of split messages based on a given expression, delimiter or file marker.

Options

`splitter.apply-sequence`

Add correlation/sequence information in headers to facilitate later aggregation. **(Boolean, default: true)**

`splitter.charset`

The charset to use when converting bytes in text-based files to String. **(String, default: <none>)**

`splitter.delimiters`

When expression is null, delimiters to use when tokenizing {@link String} payloads. **(String, default: <none>)**

`splitter.expression`

A SpEL expression for splitting payloads. **(Expression, default: <none>)**

`splitter.file-markers`

Set to true or false to use a {@code FileSplitter} (to split text-based files by line) that includes (or not) beginning/end of file markers. **(Boolean, default: <none>)**

`splitter.markers-json`

When 'fileMarkers == true', specify if they should be produced as FileSplitter.FileMarker objects or JSON. **(Boolean, default: true)**

When no expression, fileMarkers, or charset is provided, a DefaultMessageSplitter is configured with (optional) delimiters. When fileMarkers or charset is provided, a FileSplitter is configured (you must provide either a fileMarkers or charset to split files, which must be text-based - they are split into lines). Otherwise, an ExpressionEvaluatingMessageSplitter is configured.

When splitting File payloads, the sequenceSize header is zero because the size cannot be determined at the beginning.



Caution

Ambiguous properties are not allowed.

JSON Example

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>')`.

For example, consider the following JSON:

```
{ "store": {
  "book": [
    {
      "category": "reference",
      "author": "Nigel Rees",
      "title": "Sayings of the Century",
      "price": 8.95
    },
    {
      "category": "fiction",
      "author": "Evelyn Waugh",
      "title": "Sword of Honour",
      "price": 12.99
    }
  ]
}
```

```

    {
      "category": "fiction",
      "author": "Herman Melville",
      "title": "Moby Dick",
      "isbn": "0-553-21311-3",
      "price": 8.99
    },
    {
      "category": "fiction",
      "author": "J. R. R. Tolkien",
      "title": "The Lord of the Rings",
      "isbn": "0-395-19395-8",
      "price": 22.99
    }
  ],
  "bicycle": {
    "color": "red",
    "price": 19.95
  }
}

```

and an expression `#jsonPath(payload, '$.store.book')`; the result will be 4 messages, each with a Map payload containing the properties of a single book.

Build

```

$ ./mvnw clean install -PgenerateApps
$ cd apps

```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```

$ ./mvnw clean package

```

Examples

```

java -jar splitter-processor.jar --splitter.expression=expression
java -jar splitter-processor.jar --splitter.delimiters=delimiters

```

3.14 TCP Client as a processor which connects to a TCP server, sends data to it and also receives data.

Input

Headers:

- Content-Type: application/octet-stream

Payload:

- byte[]

Headers:

- Content-Type: text/plain

Payload:

- String

Output

Headers:

- Content-Type: application/octet-stream

Payload:

- byte[]

Options

The **tcp-client** processor has the following options:

tcp.buffer-size

The buffer size used when decoding messages; larger messages will be rejected. **(Integer, default: 2048)**

tcp.charset

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

tcp.decoder

The decoder to use when receiving messages. **(Encoding, default: <none>, possible values: CRLF,LF,NULL,STXETX,RAW,L1,L2,L4)**

tcp.encoder

The encoder to use when sending messages. **(Encoding, default: <none>, possible values: CRLF,LF,NULL,STXETX,RAW,L1,L2,L4)**

tcp.host

The host to which this sink will connect. **(String, default: localhost)**

tcp.nio

Whether or not to use NIO. **(Boolean, default: false)**

tcp.port

The port on which to listen; 0 for the OS to choose a port. **(Integer, default: 1234)**

tcp.retry-interval

Retry interval (in milliseconds) to check the connection and reconnect. **(Long, default: 60000)**

tcp.reverse-lookup

Perform a reverse DNS lookup on the remote IP Address; if false, just the IP address is included in the message headers. **(Boolean, default: false)**

tcp.socket-timeout

The timeout (ms) before closing the socket when no data is received. **(Integer, default: 120000)**

tcp.use-direct-buffers

Whether or not to use direct buffers. **(Boolean, default: false)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar tcp-client-processor.jar --tcp.decoder=LF --tcp.encoder=LF
```

3.15 Transform Processor

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

The transform processor is used by passing a SpEL expression. The expression should return the modified message or payload.

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>')`

Input

Headers

N/A

Payload

- Any

Output

Headers

N/A

Payload

- Any

Options

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

`transformer.expression`
<documentation missing> (**Expression, default: payload**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar transform-processor.jar --expression=payload.toUpperCase()
```

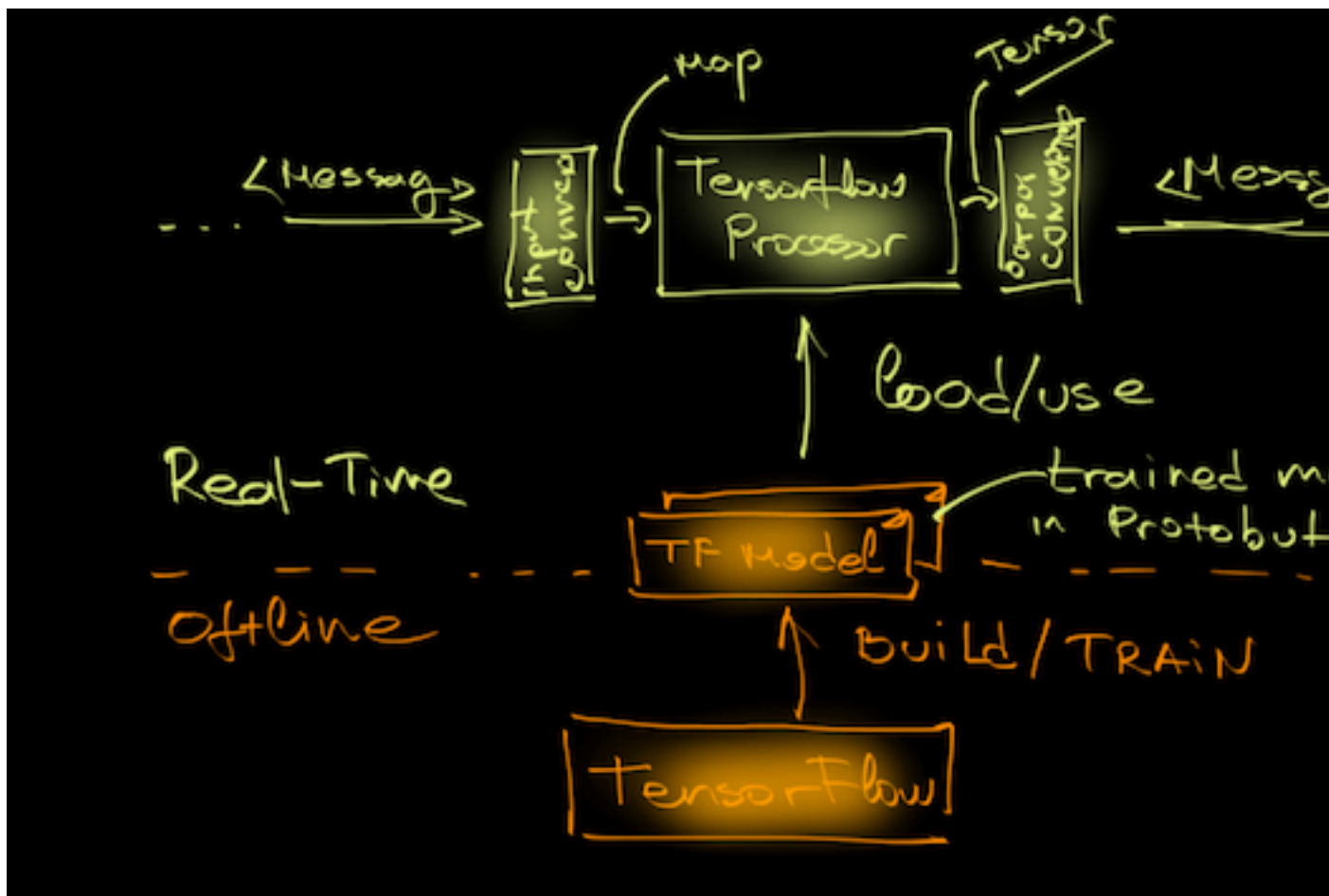
This transform will convert all message payloads to upper case.

3.16 TensorFlow Processor

A processor that evaluates a machine learning model stored in TensorFlow Protobuf format.

Following snippet shows how to export a TensorFlow model into ProtocolBuffer binary format as required by the Processor.

```
from tensorflow.python.framework.graph_util import convert_variables_to_constants
...
SAVE_DIR = os.path.abspath(os.path.curdir)
minimal_graph = convert_variables_to_constants(sess, sess.graph_def, ['<model output>'])
tf.train.write_graph(minimal_graph, SAVE_DIR, 'my_graph.proto', as_text=False)
tf.train.write_graph(minimal_graph, SAVE_DIR, 'my.txt', as_text=True)
```



The `--tensorflow.model` property configures the Processor with the location of the serialized Tensorflow model.

The `TensorflowInputConverter` converts the input data into the format, specific for the given model.

The `TensorflowOutputConverter` converts the computed Tensor's result into a pipeline Message.

The `--tensorflow.modelFetch` property defines the list of TensorFlow graph outputs to fetch the output Tensors from.

The `--tensorflow.mode` property defines whether the computed results are passed in the message payload or in the message header.

Input

Headers

N/A

Payload

The TensorFlow Processor uses a `TensorflowInputConverter` to convert the input data into data format compliant with the TensorFlow Model used. The input converter converts the input Messages into key/value Map, where the Key corresponds to a model input placeholder and the content is `org.tensorflow.DataType` compliant value. The default converter implementation expects either Map payload or flat json message that can be converted into a Map.

The `TensorflowInputConverter` can be extended and customized. See [TwitterSentimentTensorflowInputConverter.java](#) for example.

Output

Headers

N/A

Payload

Processor's output uses `TensorflowOutputConverter` to convert the computed Tensor result into a serializable message. The default implementation uses `Tuple` triple.

Custom `TensorflowOutputConverter` can provide more convenient data representations. See [TwitterSentimentTensorflowOutputConverter.java](#).

Options

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

tensorflow.expression

How to obtain the input data from the input message. If empty it defaults to the input message payload. The `payload.myInTupleName` expression treats the input payload as a `Tuple`, and `myInTupleName` stands for a `Tuple` key. The `headers[myHeaderName]` expression to get input data from message's header using `myHeaderName` as a key. **(Expression, default: <none>)**

tensorflow.mode

Defines how to store the output data and if the input payload is passed through or discarded. `Payload` (Default) stores the output data in the outbound message payload. The input payload is discarded. `Header` stores the output data in `outputName` message's header. The the input payload is passed through. `Tuple` stores the output data in an `Tuple` payload, using the `outputName` key. The input payload is passed through in the same `Tuple` using the `'original.input.data'`. If the input payload is already a `Tuple` that contains a `'original.input.data'` key, then copy the input `Tuple` into the new `Tuple` to be returned. **(OutputMode, default: <none>, possible values: payload,tuple,header)**

tensorflow.model

The location of the TensorFlow model file. (**Resource**, default: <none>)

tensorflow.model-fetch

The TensorFlow graph model outputs. Comma separate list of TensorFlow operation names to fetch the output Tensors from. (**List<String>**, default: <none>)

tensorflow.output-name

The output data key used in the Header or Tuple modes. (**String**, default: result)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

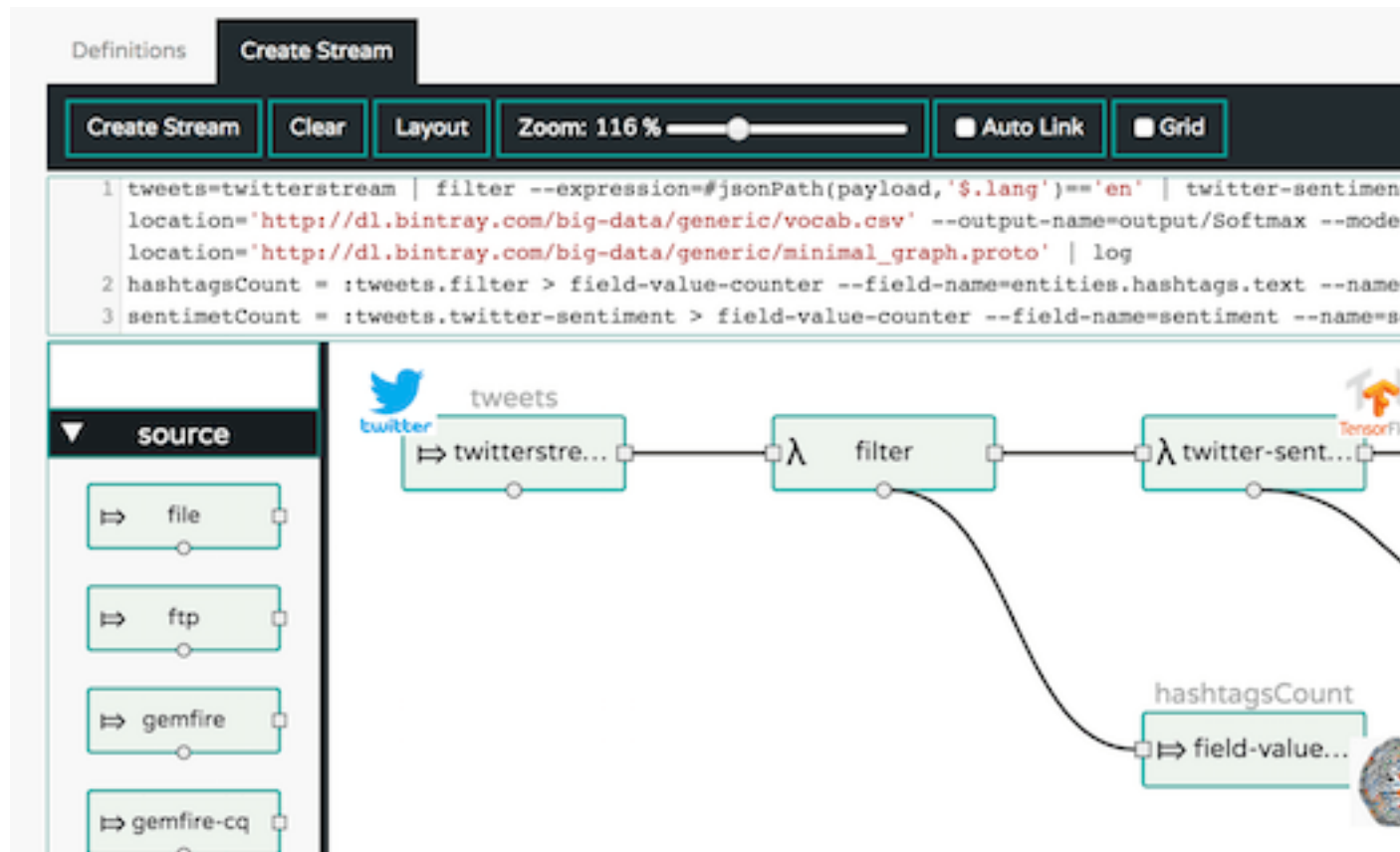
```
$ ./mvnw clean package
```

Examples

```
java -jar tensorflow-processor.jar --model= --modelFetch= --mode=
```

3.17 Twitter Sentiment Analysis Processor

A processor that evaluates a machine learning model stored in TensorFlow Protobuf format. It operationalizes the github.com/danielegrattarola/twitter-sentiment-cnn



[Real-time Twitter Sentiment Analytics with TensorFlow and Spring Cloud Dataflow](#)

Input

Headers

- content-type: application/json

Payload

- JSON tweet message

Output

Headers

- content-type: application/json

Payload

Decodes the evaluated result into POSITIVE, NEGATIVE and NEUTRAL values. Then creates and returns a simple JSON message with this structure:

N/A

Payload

Processor's output uses `TensorflowOutputConverter` to convert the computed Tensor result into a serializable message. The default implementation uses `Tuple` triple.

Custom `TensorflowOutputConverter` can provide more convenient data representations. See [TwitterSentimentTensorflowOutputConverter.java](#).

Options

The **twitter-sentiment** processor has the following options:

tensorflow.expression

How to obtain the input data from the input message. If empty it defaults to the input message payload. The `payload.myInTupleName` expression treats the input payload as a `Tuple`, and `myInTupleName` stands for a `Tuple` key. The `headers[myHeaderName]` expression to get input data from message's header using `myHeaderName` as a key. **(Expression, default: <none>)**

tensorflow.mode

Defines how to store the output data and if the input payload is passed through or discarded. `Payload` (Default) stores the output data in the outbound message payload. The input payload is discarded. `Header` stores the output data in `outputName` message's header. The the input payload is passed through. `Tuple` stores the output data in an `Tuple` payload, using the `outputName` key. The input payload is passed through in the same `Tuple` using the `'original.input.data'`. If the input payload is already a `Tuple` that contains a `'original.input.data'` key, then copy the input `Tuple` into the new `Tuple` to be returned. **(OutputMode, default: <none>, possible values: payload,tuple,header)**

tensorflow.model

The location of the TensorFlow model file. **(Resource, default: <none>)**

tensorflow.model-fetch

The TensorFlow graph model outputs. Comma separate list of TensorFlow operation names to fetch the output Tensors from. (**List<String>**, default: **<none>**)

tensorflow.output-name

The output data key used in the Header or Tuple modes. (**String**, default: **result**)

tensorflow.twitter.vocabulary

The location of the word vocabulary file, used for training the model (**Resource**, default: **<none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar twitter-sentiment-processor.jar --tensorflow.twitter.vocabulary= --tensorflow.model= \
--tensorflow.modelFetch= --tensorflow.mode=
```

And here is a sample pipeline that computes sentiments for json tweets coming from the twitterstream source and using the pre-build minimal_graph.proto and vocab.csv:

```
tweets=twitterstream --access-token-secret=xxx --access-token=xxx --consumer-secret=xxx --consumer-
key=xxx \
| filter --expression=#jsonPath(payload, '$.lang')== 'en' \
| twitter-sentimet --vocabulary='http://dl.bintray.com/big-data/generic/vocab.csv' \
--output-name=output/Softmax --model='http://dl.bintray.com/big-data/generic/minimal_graph.proto' \
--model-fetch=output/Softmax \
| log
```

4. Sinks

4.1 Aggregate Counter Sink

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.

Input

Headers

N/A

Payload

- Any

Output

N/A

Options

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

`aggregate-counter.date-format`

<documentation missing> (**String, default: yyyy-MM-dd'T'HH:mm:ss.SSS'Z'**)

`aggregate-counter.increment-expression`

Increment value for each bucket as a SpEL against the message (**Expression, default: <none>**)

`aggregate-counter.name`

The name of the aggregate counter. (**String, default: <none>**)

`aggregate-counter.name-expression`

A SpEL expression (against the incoming Message) to derive the name of the aggregate counter. (**Expression, default: <none>**)

`aggregate-counter.time-field`

A SpEL expression (against the incoming Message) to derive the timestamp value. (**Expression, default: <none>**)

`spring.redis.database`

Database index used by the connection factory. (**Integer, default: 0**)

`spring.redis.host`

Redis server host. (**String, default: localhost**)

`spring.redis.password`

Login password of the redis server. (**String, default: <none>**)

spring.redis.port

Redis server port. (**Integer, default: 6379**)

spring.redis.ssl

Whether to enable SSL support. (**Boolean, default: false**)

spring.redis.timeout

Connection timeout. (**Duration, default: <none>**)

spring.redis.url

Connection URL. Overrides host, port, and password. User is ignored. Example: redis://user:password@example.com:6379 (**String, default: <none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar aggregate-counter-sink.jar --name=<name of the counter>
java -jar aggregate-counter-sink.jar --nameExpression=<name expression>
```

4.2 Cassandra Sink

This sink application writes the content of each message it receives into Cassandra.

It expects a payload of JSON String and uses its properties to map to table columns.

Input

Headers:

- Content-Type: application/json

Payload:

A JSON String or byte array representing the entity (or a list of entities) to be persisted

Output

N/A

Options

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

cassandra.cluster.compression-type

The compression to use for the transport. (**CompressionType, default: <none>, possible values: NONE,SNAPPY,LZ4**)

`cassandra.cluster.contact-points`

The comma-delimited string of the hosts to connect to Cassandra. (**String**, default: **<none>**)

`cassandra.cluster.create-keyspace`

The flag to create (or not) keyspace on application startup. (**Boolean**, default: **false**)

`cassandra.cluster.entity-base-packages`

The base packages to scan for entities annotated with Table annotations. (**String[]**, default: **[]**)

`cassandra.cluster.init-script`

The resource with CQL scripts (delimited by ';') to initialize keyspace schema. (**Resource**, default: **<none>**)

`cassandra.cluster.keyspace`

The keyspace name to connect to. (**String**, default: **<none>**)

`cassandra.cluster.metrics-enabled`

Enable/disable metrics collection for the created cluster. (**Boolean**, default: **<none>**)

`cassandra.cluster.password`

The password for connection. (**String**, default: **<none>**)

`cassandra.cluster.port`

The port to use to connect to the Cassandra host. (**Integer**, default: **<none>**)

`cassandra.cluster.schema-action`

The schema action to perform. (**SchemaAction**, default: **<none>**, possible values: **NONE,CREATE,CREATE_IF_NOT_EXISTS,RECREATE,RECREATE_DROP_UNUSED**)

`cassandra.cluster.skip-ssl-validation`

The flag to validate the Servers' SSL certs (**Boolean**, default: **false**)

`cassandra.cluster.use-ssl`

The flag to use SSL to connect (**Boolean**, default: **false**)

`cassandra.cluster.username`

The username for connection. (**String**, default: **<none>**)

`cassandra.consistency-level`

The consistencyLevel option of WriteOptions. (**ConsistencyLevel**, default: **<none>**, possible values:

ANY,ONE,TWO,THREE,ALL,LOCAL_ONE,SERIAL,LOCAL_SERIAL,QUORUM,LOCAL_QUORUM,EACH_QUORUM)

`cassandra.ingest-query`

The ingest Cassandra query. (**String**, default: **<none>**)

`cassandra.query-type`

The queryType for Cassandra Sink. (**Type**, default: **<none>**, possible values: **INSERT,UPDATE,DELETE,STATEMENT**)

`cassandra.retry-policy`

The retryPolicy option of WriteOptions. (**CassandraRetryPolicy**, default: **<none>**, possible values: **DEFAULT,DOWNGRADING_CONSISTENCY,FALLTHROUGH,LOGGING**)

cassandra.statement-expression

The expression in Cassandra query DSL style. **(Expression, default: <none>)**

cassandra.ttl

The time-to-live option of WriteOptions. **(Integer, default: 0)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

The following example assumes a JSON payload is sent to a default destination called `input`, the sink parses some of its properties (`id,time,customer_id,value`) and persists them into a table called `orders`.

```
java -jar cassandra_sink.jar --cassandra.cluster.keyspace=test
--cassandra.ingest-query="insert into orders(id,time,customer_id, value) values (?, ?, ?, ?)"
```

4.3 Counter Sink

The counter sink simply counts the number of messages it receives, optionally storing counts in a separate store such as redis.

Input

Headers

N/A

Payload

- Any

Output

N/A

Options

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

counter.name

The name of the counter to increment. **(String, default: <none>)**

counter.name-expression

A SpEL expression (against the incoming Message) to derive the name of the counter to increment. **(Expression, default: <none>)**

spring.redis.database

Database index used by the connection factory. **(Integer, default: 0)**

spring.redis.host

Redis server host. (**String, default: localhost**)

spring.redis.password

Login password of the redis server. (**String, default: <none>**)

spring.redis.port

Redis server port. (**Integer, default: 6379**)

spring.redis.ssl

Whether to enable SSL support. (**Boolean, default: false**)

spring.redis.timeout

Connection timeout. (**Duration, default: <none>**)

spring.redis.url

Connection URL. Overrides host, port, and password. User is ignored. Example: redis://user:password@example.com:6379 (**String, default: <none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar counter-sink.jar --name=<name of the counter>
java -jar counter-sink.jar --nameExpression=<name expression>
```

4.4 Field Value Counter Sink

A field value counter is a Metric used for counting occurrences of unique values for a named field in a message payload.

Input

Headers

content-type: application/octet-stream

Payload

- POJO (Java bean)
- Map

Headers

content-type: text/plain

Payload

- JSON String

Headers

content-type: application/x-spring-tuple

Payload

- Tuple

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
```

Output

N/A

Options

The **field-value-counter** sink has the following options:

field-value-counter.field-name

The field name to extract for the counter. **(String, default: <none>)**

field-value-counter.name

The name of the counter to increment. **(String, default: <none>)**

field-value-counter.name-expression

A SpEL expression (against the incoming Message) to derive the name of the counter to increment.
(Expression, default: <none>)

spring.redis.database

Database index used by the connection factory. **(Integer, default: 0)**

spring.redis.host

Redis server host. **(String, default: localhost)**

spring.redis.password

Login password of the redis server. **(String, default: <none>)**

spring.redis.port

Redis server port. **(Integer, default: 6379)**

spring.redis.ssl

Whether to enable SSL support. **(Boolean, default: false)**

spring.redis.timeout

Connection timeout. **(Duration, default: <none>)**

spring.redis.url

Connection URL. Overrides host, port, and password. User is ignored. Example: redis://user:password@example.com:6379 **(String, default: <none>)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar field-value-counter-sink.jar --fieldName=<field-name> --name=<name of the counter>
```

4.5 File Sink

This module writes each message it receives to a file.

Input

Headers

N/A

Payload

- java.io.File
- java.io.InputStream
- byte[]
- String

Output

N/A (writes to the file system).

Options

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

file.binary

A flag to indicate whether adding a newline after the write should be suppressed. (**Boolean, default: false**)

file.charset

The charset to use when writing text content. (**String, default: UTF-8**)

file.directory

The parent directory of the target file. (**String, default: <none>**)

file.directory-expression

The expression to evaluate for the parent directory of the target file. (**Expression, default: <none>**)

file.mode

The FileExistsMode to use if the target file already exists. (**FileExistsMode, default: <none>, possible values: APPEND, APPEND_NO_FLUSH, FAIL, IGNORE, REPLACE, REPLACE_IF_MODIFIED**)

file.name

The name of the target file. (**String, default: file-sink**)

file.name-expression

The expression to evaluate for the name of the target file. (**String, default: <none>**)

file.suffix

The suffix to append to file name. (**String, default: <empty string>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar file_sink.jar --file.directory=/tmp/bar
```

4.6 FTP Sink

FTP sink is a simple option to push files to an FTP server from incoming messages.

It uses an `ftp-outbound-adapter`, therefore incoming messages can be either a `java.io.File` object, a `String` (content of the file) or an array of bytes (file content as well).

To use this sink, you need a username and a password to login.

**Note**

By default Spring Integration will use `o.s.i.file.DefaultFileNameGenerator` if none is specified. `DefaultFileNameGenerator` will determine the file name based on the value of the `file_name` header (if it exists) in the `MessageHeaders`, or if the payload of the `Message` is already a `java.io.File`, then it will use the original name of that file.

Input**Headers**

- `file_name` (See note above)

Payload

- `java.io.File`
- `java.io.InputStream`
- `byte[]`
- `String`

Output

N/A (writes to the FTP server).

Options

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

`ftp.auto-create-dir`

Whether or not to create the remote directory. (**Boolean, default: true**)

`ftp.factory.cache-sessions`

<documentation missing> (**Boolean, default: <none>**)

`ftp.factory.client-mode`

The client mode to use for the FTP session. (**ClientMode, default: <none>, possible values: ACTIVE,PASSIVE**)

`ftp.factory.host`

<documentation missing> (**String, default: <none>**)

`ftp.factory.password`

<documentation missing> (**String, default: <none>**)

`ftp.factory.port`

The port of the server. (**Integer, default: 21**)

`ftp.factory.username`

<documentation missing> (**String, default: <none>**)

`ftp.filename-expression`

A SpEL expression to generate the remote file name. (**Expression, default: <none>**)

ftp.mode

Action to take if the remote file already exists. (**FileExistsMode**, default: <none>, possible values: **APPEND,APPEND_NO_FLUSH,FAIL,IGNORE,REPLACE,REPLACE_IF_MODIFIED**)

ftp.remote-dir

The remote FTP directory. (**String**, default: /)

ftp.remote-file-separator

The remote file separator. (**String**, default: /)

ftp.tmp-dir

A temporary directory where the file will be written if '#isUseTemporaryFilename()' is true. (**String**, default: /)

ftp.tmp-file-suffix

The suffix to use while the transfer is in progress. (**String**, default: .tmp)

ftp.use-temporary-filename

Whether or not to write to a temporary file and rename. (**Boolean**, default: true)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar ftp_sink.jar --ftp.remote-dir=bar --ftp.factory.host=ftpsrvr \
  --ftp.factory.username=user --ftp.factory.password=pw
```

4.7 Gemfire Sink

The Gemfire sink allows one to write message payloads to a Gemfire server.

Input

Headers

- content-type: text/plain

Payload

- String

Headers

- content-type: application/x-java-serialized-object

Payload

- java.io.Serializable

Output

N/A

Options

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

gemfire.json

Indicates if the Gemfire region stores json objects as native Gemfire PdxInstance (**Boolean, default: false**)

gemfire.key-expression

SpEL expression to use as a cache key (**String, default: <none>**)

gemfire.pool.connect-type

Specifies connection type: 'server' or 'locator'. (**ConnectType, default: <none>, possible values: locator,server**)

gemfire.pool.host-addresses

Specifies one or more Gemfire locator or server addresses formatted as [host]:[port]. (**InetSocketAddress[], default: <none>**)

gemfire.pool.subscription-enabled

Set to true to enable subscriptions for the client pool. Required to sync updates to the client cache. (**Boolean, default: false**)

gemfire.region.region-name

The region name. (**String, default: <none>**)

gemfire.security.password

The cache password. (**String, default: <none>**)

gemfire.security.username

The cache username. (**String, default: <none>**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar gemfire-sink.jar --gemfire.keyExpression=
```

4.8 Gpfdist Sink

A sink module that route messages into GPDB/HAWQ segments via *gpfdist* protocol. Internally, this sink creates a custom http listener that supports the *gpfdist* protocol and schedules a task that orchestrates a *gpload* session in the same way it is done natively in Greenplum.

No data is written into temporary files and all data is kept in stream buffers waiting to get inserted into Greenplum DB or HAWQ. If there are no existing load sessions from Greenplum, the sink will block until such sessions are established.

Input

Headers:

- Content-Type: text/plain

Payload:

- String

Output

N/A

Options

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

gpfdist.batch-count

Number of windowed batch each segment takest (int, default: 100) (**Integer, default: 100**)

gpfdist.batch-period

Time in seconds for each load operation to sleep in between operations (int, default: 10) (**Integer, default: 10**)

gpfdist.batch-timeout

Timeout in seconds for segment inactivity. (Integer, default: 4) (**Integer, default: 4**)

gpfdist.column-delimiter

Data record column delimiter. *(Character, default: no default) (**Character, default: <none>**)

gpfdist.control-file

Path to yaml control file (String, no default) (**Resource, default: <none>**)

gpfdist.db-host

Database host (String, default: localhost) (**String, default: localhost**)

gpfdist.db-name

Database name (String, default: gpadmin) (**String, default: gpadmin**)

gpfdist.db-password

Database password (String, default: gpadmin) (**String, default: gpadmin**)

gpfdist.db-port

Database port (int, default: 5432) (**Integer, default: 5432**)

gpfdist.db-user

Database user (String, default: gpadmin) (**String, default: gpadmin**)

gpfdist.delimiter

Data line delimiter (String, default: newline character) (**String, default:**)

gpfdist.flush-count

Flush item count (int, default: 100) **(Integer, default: 100)**

gpfdist.flush-time

Flush item time (int, default: 2) **(Integer, default: 2)**

gpfdist.gpfdist-port

Port of gpfdist server. Default port `0` indicates that a random port is chosen. (Integer, default: 0)
(Integer, default: 0)

gpfdist.log-errors

Enable log errors. (Boolean, default: false) **(Boolean, default: false)**

gpfdist.match-columns

Match columns with update (String, no default) **(String, default: <none>)**

gpfdist.mode

Mode, either insert or update (String, no default) **(String, default: <none>)**

gpfdist.null-string

Null string definition. (String, default: ``) **(String, default: <none>)**

gpfdist.rate-interval

Enable transfer rate interval (int, default: 0) **(Integer, default: 0)**

gpfdist.segment-reject-limit

Error reject limit. (String, default: ``) **(String, default: <none>)**

gpfdist.segment-reject-type

Error reject type, either `rows` or `percent`. (String, default: `rows`) **(SegmentRejectType, default: <none>, possible values: ROWS,PERCENT)**

gpfdist.sql-after

Sql to run after load (String, no default) **(String, default: <none>)**

gpfdist.sql-before

Sql to run before load (String, no default) **(String, default: <none>)**

gpfdist.table

Target database table (String, no default) **(String, default: <none>)**

gpfdist.update-columns

Update columns with update (String, no default) **(String, default: <none>)**

spring.net.hostdiscovery.loopback

The new loopback flag. Default value is FALSE **(Boolean, default: false)**

spring.net.hostdiscovery.match-interface

The new match interface regex pattern. Default value is empty **(String, default: <none>)**

spring.net.hostdiscovery.match-ipv4

Used to match ip address from a network using a cidr notation **(String, default: <none>)**

spring.net.hostdiscovery.point-to-point

The new point to point flag. Default value is FALSE **(Boolean, default: false)**

spring.net.hostdiscovery.prefer-interface

The new preferred interface list (**List<String>**, default: <none>)

Implementation Notes

Within a `gpfdist` sink we have a Reactor based stream where data is published from the incoming SI channel. This channel receives data from the Message Bus. The Reactor stream is then connected to Netty based http channel adapters so that when a new http connection is established, the Reactor stream is flushed and balanced among existing http clients. When Greenplum does a load from an external table, each segment will initiate a http connection and start loading data. The net effect is that incoming data is automatically spread among the Greenplum segments.

Detailed Option Descriptions

The **gpfdist** sink supports the following configuration properties:

table

Database table to work with. (**String**, default: ````, required)

This option denotes a table where data will be inserted or updated. Also external table structure will be derived from structure of this table.

Currently `table` is only way to define a structure of an external table. Effectively it will replace `other_table` in below clause segment.

```
CREATE READABLE EXTERNAL TABLE table_name LIKE other_table
```

mode

Gpfdist mode, either `insert` or `update`. (**String**, default: `insert`)

Currently only `insert` and `update` gpfdist mode is supported. Mode `merge` familiar from a native gpfdist loader is not yet supported.

For mode `update` options `matchColumns` and `updateColumns` are required.

columnDelimiter

Data record column delimiter. (**Character**, default: ````)

Defines used `delimiter` character in below clause segment which would be part of a `FORMAT 'TEXT'` or `FORMAT 'CSV'` sections.

```
[DELIMITER AS 'delimiter']
```

segmentRejectLimit

Error reject limit. (**String**, default: ````)

Defines a count value in a below clause segment.

```
[ [LOG ERRORS] SEGMENT REJECT LIMIT count  
[ROWS | PERCENT] ]
```

As a convenience this reject limit also recognizes a percentage format `2%` and if used, `segmentRejectType` is automatically set to `percent`.

segmentRejectType

Error reject type, either `rows` or `percent`. (**String**, default: ````)

Defines ROWS or PERCENT in below clause segment.

```
[ [LOG ERRORS] SEGMENT REJECT LIMIT count
[ROWS | PERCENT] ]
```

logErrors

Enable or disable log errors. **(Boolean, default: false)**

As error logging is optional with SEGMENT REJECT LIMIT, it's only used if both segmentRejectLimit and segmentRejectType are set. Enables the error log in below clause segment.

```
[ [LOG ERRORS] SEGMENT REJECT LIMIT count
[ROWS | PERCENT] ]
```

nullString

Null string definition. **(String, default: ``)**

Defines used null string in below clause segment which would be part of a FORMAT 'TEXT' or FORMAT 'CSV' sections.

```
[NULL AS 'null string']
```

delimiter

Data record delimiter for incoming messages. **(String, default: \n)**

On default a delimiter in this option will be added as a postfix to every message sent into this sink. Currently *NEWLINE* is not a supported config option and line termination for data is coming from a default functionality.

If not specified, a Greenplum Database segment will detect the newline type by looking at the first row of data it receives and using the first newline type encountered.

— External Table Docs

matchColumns

Comma delimited list of columns to match. **(String, default: ``)**



Note

See more from examples below.

updateColumns

Comma delimited list of columns to update. **(String, default: ``)**



Note

See more from examples below.

sqlBefore

Sql clause to run before each load operation. **(String, default: ``)**

sqlAfter

Sql clause to run after each load operation. **(String, default: ``)**

rateInterval

Debug rate of data transfer. **(Integer, default: 0)**

If set to non zero, sink will log a rate of messages passing through a sink after number of messages denoted by this setting has been processed. Value 0 means that this rate calculation and logging is disabled.

`flushCount`

Max collected size per windowed data. **(Integer, default: 100)**



Note

For more info on flush and batch settings, see above.

How Data Is Sent Into Segments

There are few important concepts involving how data passes into a sink, through it and finally lands into a database.

- Sink has its normal message handler for incoming data from a source module, gpfdist protocol listener based on netty where segments connect to and in between those two a reactor based streams controlling load balancing into different segment connections.
- Incoming data is first sent into a reactor which first constructs a window. This window is then released into a downstream when it gets full(`flushTime`) or timeouts(`flushTime`) if window doesn't get full. One window is then ready to get sent into a segment.
- Segments which connects to this stream are now able to see a stream of window data, not stream of individual messages. We can also call this as a stream of batches.
- When segment makes a connection to a protocol listener it subscribes itself into this stream and takes count of batches denoted by `batchCount` and completes a stream if it got enough batches or if `batchTimeout` occurred due to inactivity.
- It doesn't matter how many simultaneous connections there are from a database cluster at any given time as reactor will load balance batches with all subscribers.
- Database cluster will initiate this loading session when select is done from an external table which will point to this sink. These loading operations are run in a background in a loop one after another. Option `batchPeriod` is then used as a sleep time in between these load sessions.

Lets take a closer look how options `flushCount`, `flushTime`, `batchCount`, `batchTimeout` and `batchPeriod` work.

As in a highest level where incoming data into a sink is windowed, `flushCount` and `flushTime` controls when a batch of messages are sent into a downstream. If there are a lot of simultaneous segment connections, flushing less will keep more segments inactive as there is more demand for batches than what flushing will produce.

When existing segment connection is active and it has subscribed itself with a stream of batches, data will keep flowing until either `batchCount` is met or `batchTimeout` occurs due to inactivity of data from an upstream. Higher a `batchCount` is more data each segment will read. Higher a `batchTimeout` is more time segment will wait in case there is more data to come.

As gpfdist load operations are done in a loop, `batchPeriod` simply controls not to run things in a busy loop. Busy loop would be ok if there is a constant stream of data coming in but if incoming data is more like bursts then busy loop would be unnecessary.

**Note**

Data loaded via `gpfdist` will not become visible in a database until whole distributed loading session have finished successfully.

Reactor is also handling backpressure meaning if existing load operations will not produce enough demand for data, eventually message passing into a sink will block. This happens when Reactor's internal ring buffer (size of 32 items) gets full. Flow of data through sink really happens when data is pulled from it by segments.

Example Usage

In this first example we're just creating a simple stream which inserts data from a `time` source. Let's create a table with two `text` columns.

```
gpadmin=# create table ticktock (date text, time text);
```

Create a simple stream `gpstream`.

```
dataflow:>stream create --name gpstream1 --definition "time | gpfdist
--dbHost=mdw --table=ticktock --batchTime=1 --batchPeriod=1
--flushCount=2 --flushTime=2 --columnDelimiter=' ' --deploy
```

Let it run and see results from a database.

```
gpadmin=# select count(*) from ticktock;
count
-----
14
(1 row)
```

In previous example we did a simple inserts into a table. Let's see how we can update data in a table. Create a simple table `httpdata` with three text columns and insert some data.

```
gpadmin=# create table httpdata (col1 text, col2 text, col3 text);
gpadmin=# insert into httpdata values ('DATA1', 'DATA', 'DATA');
gpadmin=# insert into httpdata values ('DATA2', 'DATA', 'DATA');
gpadmin=# insert into httpdata values ('DATA3', 'DATA', 'DATA');
```

Now table looks like this.

```
gpadmin=# select * from httpdata;
col1 | col2 | col3
-----+-----+-----
DATA3 | DATA | DATA
DATA2 | DATA | DATA
DATA1 | DATA | DATA
(3 rows)
```

Let's create a stream which will update table `httpdata` by matching a column `col1` and updates columns `col2` and `col3`.

```
dataflow:>stream create --name gpfdiststream2 --definition "http
--server.port=8081|gpfdist --mode=update --table=httpdata
--dbHost=mdw --columnDelimiter=',' --matchColumns=col1
--updateColumns=col2,col3" --deploy
```

Post some data into a stream which will be passed into a `gpfdist` sink via `http` source.

```
curl --data "DATA1,DATA1,DATA1" -H "Content-Type:text/plain" http://localhost:8081/
```

If you query table again, you'll see that row for *DATA1* has been updated.

```
gpadmin=# select * from httpdata;
 col1  | col2  | col3
-----+-----
 DATA3 | DATA | DATA
 DATA2 | DATA | DATA
 DATA1 | DATA1 | DATA1
(3 rows)
```

Tuning Transfer Rate

Default values for options `flushCount`, `flushTime`, `batchCount`, `batchTimeout` and `batchPeriod` are relatively conservative and needs to be *tuned* for every use case for optimal performance. Order to make a decision on how to tune sink behaviour to suit your needs few things needs to be considered.

- What is an average size of messages ingested by a sink.
- How fast you want data to become visible in a database.
- Is incoming data a constant flow or a bursts of data.

Everything what flows through a sink is kept in-memory and because sink is handling backpressure, memory consumption is relatively low. However because sink cannot predict what is an average size of an incoming data and this data is anyway windowed later in a downstream you should not allow window size to become too large if average data size is large as every batch of data is kept in memory.

Generally speaking if you have a lot of segments in a load operation, it's advised to keep flushed window size relatively small which allows more segments to stay active. This however also depends on how much data is flowing in into a sink itself.

Longer a load session for each segment is active higher the overall transfer rate is going to be. Option `batchCount` naturally controls this. However option `batchTimeout` then really controls how fast each segment will complete a stream due to inactivity from upstream and to step away from a loading session to allow distributes session to finish and data become visible in a database.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

See above.

4.9 HDFS Sink

This module writes each message it receives to HDFS.

Input

Headers

Payload

Any

Output

N/A

Options

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

hdfs.close-timeout

Timeout in ms, regardless of activity, after which file will be automatically closed. **(Long, default: 0)**

hdfs.codec

Compression codec alias name (gzip, snappy, bzip2, lzo, or slzo). **(String, default: <none>)**

hdfs.directory

Base path to write files to. **(String, default: <none>)**

hdfs.enable-sync

Whether writer will sync to datanode when flush is called, setting this to 'true' could impact throughput. **(Boolean, default: false)**

hdfs.file-extension

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

hdfs.file-name

The base filename to use for the created files. **(String, default: <none>)**

hdfs.file-open-attempts

Maximum number of file open attempts to find a path. **(Integer, default: 10)**

hdfs.file-uuid

Whether file name should contain uuid. **(Boolean, default: false)**

hdfs.flush-timeout

Timeout in ms, regardless of activity, after which data written to file will be flushed. **(Long, default: 0)**

hdfs.fs-uri

URL for HDFS Namenode. **(String, default: <none>)**

hdfs.idle-timeout

Inactivity timeout in ms after which file will be automatically closed. **(Long, default: 0)**

hdfs.in-use-prefix

Prefix for files currently being written. **(String, default: <none>)**

hdfs.in-use-suffix

Suffix for files currently being written. **(String, default: <none>)**

hdfs.overwrite

Whether writer is allowed to overwrite files in Hadoop FileSystem. **(Boolean, default: false)**

hdfs.partition-path

A SpEL expression defining the partition path. **(String, default: <none>)**

hdfs.rollover

Threshold in bytes when file will be automatically rolled over. **(Integer, default: 1000000000)**



Note

This module can have its runtime dependencies provided during startup if you would like to use a Hadoop distribution other than the default one.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar hdfs-sink.jar --fsUri
```

4.10 Jdbc Sink

A module that writes its incoming payload to an RDBMS using JDBC.

Input

Headers

Payload

- Any

Column expression will be evaluated against the message and the expression will usually be compatible with only one type (such as a Map or bean etc.)

Output

N/A

Options

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

jdbc.columns

The comma separated colon-based pairs of column names and SpEL expressions for values to insert/update. Names are used at initialization time to issue the DDL. **(String, default: payload:payload.toString())**

`jdbc.initialize`

'true', 'false' or the location of a custom initialization script for the table. **(String, default: false)**

`jdbc.table-name`

The name of the table to write into. **(String, default: messages)**

`spring.datasource.data`

Data (DML) script resource references. **(List<String>, default: <none>)**

`spring.datasource.driver-class-name`

Fully qualified name of the JDBC driver. Auto-detected based on the URL by default. **(String, default: <none>)**

`spring.datasource.initialization-mode`

Initialize the datasource using available DDL and DML scripts. **(DataSourceInitializationMode, default: embedded, possible values: ALWAYS, EMBEDDED, NEVER)**

`spring.datasource.password`

Login password of the database. **(String, default: <none>)**

`spring.datasource.schema`

Schema (DDL) script resource references. **(List<String>, default: <none>)**

`spring.datasource.url`

JDBC url of the database. **(String, default: <none>)**

`spring.datasource.username`

Login username of the database. **(String, default: <none>)**

The `jdbc.columns` property represents pairs of `COLUMN_NAME [ : EXPRESSION_FOR_VALUE ]` where `EXPRESSION_FOR_VALUE` (together with the colon) is optional. In this case the value is evaluated via generated expression like `payload.COLUMN_NAME`, so this way we have a direct mapping from object properties to the table column. For example we have a JSON payload like:

```
{
  "name": "My Name"
  "address": {
    "city": "Big City",
    "street": "Narrow Alley"
  }
}
```

So, we can insert it into the table with `name`, `city` and `street` structure using the configuration:

```
--jdbc.columns=name,city:address.city,street:address.street
```



Note

The module also uses Spring Boot's [DataSource support](#) for configuring the database connection, so properties like `spring.datasource.url` etc. apply.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar jdbc-sink.jar --jdbc.tableName=names --jdbc.columns=name --spring.datasource.driver-class-name=org.mariadb.jdbc.Driver \
--spring.datasource.url='jdbc:mysql://localhost:3306/test'
```

4.11 Log Sink

The log sink uses the application logger to output the data for inspection.

Please understand that log sink uses type-less handler, which affects how the actual logging will be performed. This means that if the content-type is textual, then raw payload bytes will be converted to String, otherwise raw bytes will be logged. Please see more info in the [user-guide](#).

Input

Headers

Payload

Any

Output

N/A

Options

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

log.expression

A SpEL expression (against the incoming message) to evaluate as the logged message. **(String, default: payload)**

log.level

The level at which to log messages. **(Level, default: <none>, possible values: FATAL,ERROR,WARN,INFO,DEBUG,TRACE)**

log.name

The name of the logger to use. **(String, default: <none>)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar log-sink.jar
```

4.12 RabbitMQ Sink

This module sends messages to RabbitMQ.

Input

Headers

- content-type: text/plain

Payload

- String

Headers

- content-type: application/octet-stream

Payload

- byte[]

Headers

- content-type: application/x-java-serialized-object

Payload

- java.io.Serializable

Note: With `converterBeanName = jsonConverter` any object that can be converted to JSON by Jackson (content type sent to rabbit will be `application/json` with type information in other headers).

With `converterBeanName` set to something else, payload will be any object that the converter can handle.

Output

N/A

Options

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

(See the Spring Boot documentation for RabbitMQ connection properties)

`rabbit.converter-bean-name`

The bean name for a custom message converter; if omitted, a `SimpleMessageConverter` is used. If 'jsonConverter', a `Jackson2JsonMessageConverter` bean will be created for you. **(String, default: <none>)**

`rabbit.exchange`

Exchange name - overridden by `exchangeNameExpression`, if supplied. **(String, default: <empty string>)**

rabbit.exchange-expression

A SpEL expression that evaluates to an exchange name. **(Expression, default: <none>)**

rabbit.mapped-request-headers

Headers that will be mapped. **(String[], default: [*])**

rabbit.own-connection

When true, use a separate connection based on the boot properties. **(Boolean, default: false)**

rabbit.persistent-delivery-mode

Default delivery mode when 'amqp_deliveryMode' header is not present, true for PERSISTENT. **(Boolean, default: false)**

rabbit.routing-key

Routing key - overridden by routingKeyExpression, if supplied. **(String, default: <none>)**

rabbit.routing-key-expression

A SpEL expression that evaluates to a routing key. **(Expression, default: <none>)**

spring.rabbitmq.addresses

Comma-separated list of addresses to which the client should connect. **(String, default: <none>)**

spring.rabbitmq.connection-timeout

Connection timeout. Set it to zero to wait forever. **(Duration, default: <none>)**

spring.rabbitmq.host

RabbitMQ host. **(String, default: localhost)**

spring.rabbitmq.password

Login to authenticate against the broker. **(String, default: guest)**

spring.rabbitmq.port

RabbitMQ port. **(Integer, default: 5672)**

spring.rabbitmq.publisher-confirms

Whether to enable publisher confirms. **(Boolean, default: false)**

spring.rabbitmq.publisher-returns

Whether to enable publisher returns. **(Boolean, default: false)**

spring.rabbitmq.requested-heartbeat

Requested heartbeat timeout; zero for none. If a duration suffix is not specified, seconds will be used. **(Duration, default: <none>)**

spring.rabbitmq.username

Login user to authenticate to the broker. **(String, default: guest)**

spring.rabbitmq.virtual-host

Virtual host to use when connecting to the broker. **(String, default: <none>)**



Note

By default, the message converter is a `SimpleMessageConverter` which handles `byte[]`, `String` and `java.io.Serializable`. A well-known bean name `jsonConverter` will

configure a `Jackson2JsonMessageConverter` instead. In addition, a custom converter bean can be added to the context and referenced by the `converterBeanName` property.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar rabbit-sink.jar --rabbit.routingKey=
java -jar rabbit-sink.jar --rabbit.routingKeyExpression=
```

4.13 MongoDB Sink

This sink application ingest incoming data into MongoDB. This application is fully based on the `MongoDataAutoConfiguration`, so refer to the [Spring Boot MongoDB Support](#) for more information.

Input

Headers

Payload

- Any POJO

Output

N/A

Options

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

`mongodb.collection`

The MongoDB collection to store data (**String**, default: **<none>**)

`mongodb.collection-expression`

The SpEL expression to evaluate MongoDB collection (**Expression**, default: **<none>**)

`spring.data.mongodb.authentication-database`

Authentication database name. (**String**, default: **<none>**)

`spring.data.mongodb.database`

Database name. (**String**, default: **<none>**)

`spring.data.mongodb.field-naming-strategy`

Fully qualified name of the `FieldNamingStrategy` to use. (**Class<?>**, default: **<none>**)

`spring.data.mongodb.grid-fs-database`

GridFS database name. (**String**, default: **<none>**)

spring.data.mongodb.host

Mongo server host. Cannot be set with URI. (**String**, default: <none>)

spring.data.mongodb.password

Login password of the mongo server. Cannot be set with URI. (**Character[]**, default: <none>)

spring.data.mongodb.port

Mongo server port. Cannot be set with URI. (**Integer**, default: <none>)

spring.data.mongodb.uri

Mongo database URI. Cannot be set with host, port and credentials. (**String**, default: <none>)

spring.data.mongodb.username

Login user of the mongo server. Cannot be set with URI. (**String**, default: <none>)

Also see the [Spring Boot Documentation](#) for additional `MongoProperties` properties.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar mongodb-sink.jar --mongodb.collection=
java -jar mongodb-sink.jar --mongodb.collectionExpression=
```

4.14 MQTT Sink

This module sends messages to MQTT.

Input

Headers:

Payload:

- `byte[]`
- `String`

Output

N/A

Options

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

mqtt.async

whether or not to use async sends (**Boolean**, default: **false**)

mqtt.charset

the charset used to convert a String payload to byte[] (**String, default: UTF-8**)

mqtt.clean-session

whether the client and server should remember state across restarts and reconnects (**Boolean, default: true**)

mqtt.client-id

identifies the client (**String, default: stream.client.id.sink**)

mqtt.connection-timeout

the connection timeout in seconds (**Integer, default: 30**)

mqtt.keep-alive-interval

the ping interval in seconds (**Integer, default: 60**)

mqtt.password

the password to use when connecting to the broker (**String, default: guest**)

mqtt.persistence

'memory' or 'file' (**String, default: memory**)

mqtt.persistence-directory

Persistence directory (**String, default: /tmp/paho**)

mqtt.qos

the quality of service to use (**Integer, default: 1**)

mqtt.retained

whether to set the 'retained' flag (**Boolean, default: false**)

mqtt.topic

the topic to which the sink will publish (**String, default: stream.mqtt**)

mqtt.url

location of the mqtt broker(s) (comma-delimited list) (**String[], default: [tcp://localhost:1883]**)

mqtt.username

the username to use when connecting to the broker (**String, default: guest**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar mqtt-sink.jar --mqtt.clientid= --mqtt.topic=
```

4.15 Pgcopy Sink

A module that writes its incoming payload to an RDBMS using the PostgreSQL COPY command.

Input

Headers

Payload

- Any

Column expression will be evaluated against the message and the expression will usually be compatible with only one type (such as a Map or bean etc.)

Output

N/A

Options

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

pgcopy.batch-size

Threshold in number of messages when data will be flushed to database table. (**Integer, default: 10000**)

pgcopy.columns

The names of the columns that shall receive data. Also used at initialization time to issue the DDL. (**List<String>, default: payload**)

pgcopy.delimiter

Specifies the character that separates columns within each row (line) of the file. The default is a tab character in text format, a comma in CSV format. This must be a single one-byte character. Using an escaped value like '\t' is allowed. (**String, default: <none>**)

pgcopy.error-table

The name of the error table used for writing rows causing errors. The error table should have three columns named "table_name", "error_message" and "payload" large enough to hold potential data values. You can use the following DDL to create this table: 'CREATE TABLE ERRORS (TABLE_NAME VARCHAR(255), ERROR_MESSAGE TEXT,PAYLOAD TEXT)' (**String, default: <none>**)

pgcopy.escape

Specifies the character that should appear before a data character that matches the QUOTE value. The default is the same as the QUOTE value (so that the quoting character is doubled if it appears in the data). This must be a single one-byte character. This option is allowed only when using CSV format. (**Character, default: <none>**)

pgcopy.format

Format to use for the copy command. (**Format, default: <none>, possible values: TEXT,CSV**)

pgcopy.idle-timeout

Idle timeout in milliseconds when data is automatically flushed to database table. (**Long, default: -1**)

pgcopy.initialize

'true', 'false' or the location of a custom initialization script for the table. **(String, default: false)**

pgcopy.null-string

Specifies the string that represents a null value. The default is \N (backslash-N) in text format, and an unquoted empty string in CSV format. **(String, default: <none>)**

pgcopy.quote

Specifies the quoting character to be used when a data value is quoted. The default is double-quote. This must be a single one-byte character. This option is allowed only when using CSV format. **(Character, default: <none>)**

pgcopy.table-name

The name of the table to write into. **(String, default: <none>)**

spring.datasource.driver-class-name

Fully qualified name of the JDBC driver. Auto-detected based on the URL by default. **(String, default: <none>)**

spring.datasource.password

Login password of the database. **(String, default: <none>)**

spring.datasource.url

JDBC url of the database. **(String, default: <none>)**

spring.datasource.username

Login username of the database. **(String, default: <none>)**

**Note**

The module also uses Spring Boot's [DataSource support](#) for configuring the database connection, so properties like `spring.datasource.url` etc. apply.

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

For integration tests to run, start a PostgreSQL database on localhost:

```
docker run -e POSTGRES_PASSWORD=spring -e POSTGRES_DB=test -p 5432:5432 -d postgres:latest
```

Examples

```
java -jar pgcopy-sink.jar --tableName=names --columns=name --spring.datasource.driver-class-name=org.mariadb.jdbc.Driver \
--spring.datasource.url='jdbc:mysql://localhost:3306/test'
```

4.16 Redis Sink

This module sends messages to Redis store.

Input

Headers

- content-type: text/plain

Payload

- String

Headers

- content-type: application/octet-stream

Payload

- byte[]

Output

N/A

Options

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

redis.key

A literal key name to use when storing to a key. (**String, default: <none>**)

redis.key-expression

A SpEL expression to use for storing to a key. (**Expression, default: <none>**)

redis.queue

A literal queue name to use when storing in a queue. (**String, default: <none>**)

redis.queue-expression

A SpEL expression to use for queue. (**Expression, default: <none>**)

redis.topic

A literal topic name to use when publishing to a topic. (**String, default: <none>**)

redis.topic-expression

A SpEL expression to use for topic. (**Expression, default: <none>**)

spring.redis.database

Database index used by the connection factory. (**Integer, default: 0**)

spring.redis.host

Redis server host. (**String, default: localhost**)

spring.redis.jedis.pool.max-active

Maximum number of connections that can be allocated by the pool at a given time. Use a negative value for no limit. (**Integer, default: 8**)

spring.redis.jedis.pool.max-idle

Maximum number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections. (**Integer, default: 8**)

`spring.redis.jedis.pool.max-wait`

Maximum amount of time a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely. **(Duration, default: -1ms)**

`spring.redis.jedis.pool.min-idle`

Target for the minimum number of idle connections to maintain in the pool. This setting only has an effect if it is positive. **(Integer, default: 0)**

`spring.redis.lettuce.pool.max-active`

Maximum number of connections that can be allocated by the pool at a given time. Use a negative value for no limit. **(Integer, default: 8)**

`spring.redis.lettuce.pool.max-idle`

Maximum number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections. **(Integer, default: 8)**

`spring.redis.lettuce.pool.max-wait`

Maximum amount of time a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely. **(Duration, default: -1ms)**

`spring.redis.lettuce.pool.min-idle`

Target for the minimum number of idle connections to maintain in the pool. This setting only has an effect if it is positive. **(Integer, default: 0)**

`spring.redis.password`

Login password of the redis server. **(String, default: <none>)**

`spring.redis.port`

Redis server port. **(Integer, default: 6379)**

`spring.redis.sentinel.master`

Name of the Redis server. **(String, default: <none>)**

`spring.redis.sentinel.nodes`

Comma-separated list of "host:port" pairs. **(List<String>, default: <none>)**

`spring.redis.ssl`

Whether to enable SSL support. **(Boolean, default: false)**

`spring.redis.timeout`

Connection timeout. **(Duration, default: <none>)**

`spring.redis.url`

Connection URL. Overrides host, port, and password. User is ignored. Example: `redis://user:password@example.com:6379` **(String, default: <none>)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar redis-pubsub-sink.jar --redis.queue=
java -jar redis-pubsub-sink.jar --redis.queueExpression=
java -jar redis-pubsub-sink.jar --redis.key=
java -jar redis-pubsub-sink.jar --redis.keyExpression=
java -jar redis-pubsub-sink.jar --redis.topic=
java -jar redis-pubsub-sink.jar --redis.topicExpression=
```

4.17 Router Sink

This application routes messages to named channels.

Input

Headers

Payload

Any

Output

N/A

Options

Options

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

`router.default-output-channel`

Where to send unroutable messages. (**String**, default: `nullChannel`)

`router.destination-mappings`

Destination mappings as a new line delimited string of name-value pairs, e.g. 'foo=bar\n baz=car'. (**Properties**, default: `<none>`)

`router.expression`

The expression to be applied to the message to determine the channel(s) to route to. (**Expression**, default: `<none>`)

`router.refresh-delay`

How often to check for script changes in ms (if present); < 0 means don't refresh. (**Integer**, default: `60000`)

`router.resolution-required`

Whether or not channel resolution is required. (**Boolean**, default: `false`)

`router.script`

The location of a groovy script that returns channels or channel mapping resolution keys. (**Resource**, default: `<none>`)

`router.variables`

Variable bindings as a new line delimited string of name-value pairs, e.g. 'foo=bar\n baz=car'. (**Properties**, default: `<none>`)

router.variables-location

The location of a properties file containing custom script variable bindings. (**Resource, default:** `<none>`)



Note

Since this is a dynamic router, destinations are created as needed; therefore, by default the `defaultOutputChannel` and `resolutionRequired` will only be used if the Binder has some problem binding to the destination.

You can restrict the creation of dynamic bindings using the `spring.cloud.stream.dynamicDestinations` property. By default, all resolved destinations will be bound dynamically; if this property has a comma-delimited list of destination names, only those will be bound. Messages that resolve to a destination that is not in this list will be routed to the `defaultOutputChannel`, which must also appear in the list.

`destinationMappings` are used to map the evaluation results to an actual destination name.

SpEL-based Routing

The expression evaluates against the message and returns either a channel name, or the key to a map of channel names.

For more information, please see the "Routers and the Spring Expression Language (SpEL)" subsection in the Spring Integration Reference manual [Configuring \(Generic\) Router section](#).

Groovy-based Routing

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

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

If you want to pass variable values to your script, you can statically bind values using the *variables* option or optionally pass the path to a properties file containing the bindings using the *propertiesLocation* option. All properties in the file will be made available to the script as variables. You may specify both *variables* and *propertiesLocation*, in which case any duplicate values provided as *variables* override values provided in *propertiesLocation*. Note that *payload* and *headers* are implicitly bound to give you access to the data contained in a message.

For more information, see the Spring Integration Reference manual [Groovy Support](#).

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar router-sink.jar --expression=" "
java -jar router-sink.jar --script=" "
```

4.18 Amazon S3 Sink

This sink app supports transfer files to the Amazon S3 bucket. Files payloads (and directories recursively) are transferred to the remote directory (S3 bucket) to the local directory where the application is deployed.

Messages accepted by this sink must contain payload as:

- File, including directories for recursive upload;
- InputStream;
- byte[]

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

Input

Headers

N/A

Payload

- java.io.File
- java.io.InputStream
- byte[]
- String

Output

N/A

Options

The **s3** sink has the following options:

s3.acl

S3 Object access control list. (**CannedAccessControlList**, default: **<none>**, possible values: **private,public-read,public-read-write,authenticated-read,log-delivery-write,bucket-owner-read,bucket-owner-full-control,aws-exec-read**)

s3.acl-expression

Expression to evaluate S3 Object access control list. **(Expression, default: <none>)**

s3.bucket

AWS bucket for target file(s) to store. **(String, default: <none>)**

s3.bucket-expression

Expression to evaluate AWS bucket name. **(Expression, default: <none>)**

s3.key-expression

Expression to evaluate S3 Object key. **(Expression, default: <none>)**

The target generated application based on the `AmazonS3SinkConfiguration` can be enhanced with the `S3MessageHandler.UploadMetadataProvider` and/or `S3ProgressListener`, which are injected into `S3MessageHandler` bean.

Amazon AWS common options

The Amazon S3 Sink (as all other Amazon AWS applications) is based on the [Spring Cloud AWS](#) project as a foundation, and its auto-configuration classes are used automatically by Spring Boot. Consult their documentation regarding required and useful auto-configuration properties.

Some of them are about AWS credentials:

- `cloud.aws.credentials.accessKey`
- `cloud.aws.credentials.secretKey`
- `cloud.aws.credentials.instanceProfile`
- `cloud.aws.credentials.profileName`
- `cloud.aws.credentials.profilePath`

Other are for AWS Region definition:

- `cloud.aws.region.auto`
- `cloud.aws.region.static`

And for AWS Stack:

- `cloud.aws.stack.auto`
- `cloud.aws.stack.name`

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar s3-sink.jar --s3.bucket=tmp/bar
```

4.19 SFTP Sink

SFTP sink is a simple option to push files to an SFTP server from incoming messages.

It uses an `sftp-outbound-adapter`, therefore incoming messages can be either a `java.io.File` object, a `String` (content of the file) or an array of bytes (file content as well).

To use this sink, you need a username and a password to login.



Note

By default Spring Integration will use `o.s.i.file.DefaultFileNameGenerator` if none is specified. `DefaultFileNameGenerator` will determine the file name based on the value of the `file_name` header (if it exists) in the `MessageHeaders`, or if the payload of the `Message` is already a `java.io.File`, then it will use the original name of that file.

When configuring the `sftp.factory.known-hosts-expression` option, the root object of the evaluation is the application context, an example might be `sftp.factory.known-hosts-expression = @systemProperties['user.home'] + '/.ssh/known_hosts'`.

Input

Headers

- `file_name` (See note above)

Payload

- `java.io.File`
- `java.io.InputStream`
- `byte[]`
- `String`

Output

N/A (writes to the SFTP server).

Options

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

`sftp.auto-create-dir`

Whether or not to create the remote directory. (**Boolean, default: true**)

`sftp.factory.allow-unknown-keys`

True to allow an unknown or changed key. (**Boolean, default: false**)

sftp.factory.cache-sessions

Cache sessions **(Boolean, default: <none>)**

sftp.factory.host

The host name of the server. **(String, default: localhost)**

sftp.factory.known-hosts-expression

A SpEL expression resolving to the location of the known hosts file. **(Expression, default: <none>)**

sftp.factory.pass-phrase

Passphrase for user's private key. **(String, default: <empty string>)**

sftp.factory.password

The password to use to connect to the server. **(String, default: <none>)**

sftp.factory.port

The port of the server. **(Integer, default: 22)**

sftp.factory.private-key

Resource location of user's private key. **(String, default: <empty string>)**

sftp.factory.username

The username to use to connect to the server. **(String, default: <none>)**

sftp.filename-expression

A SpEL expression to generate the remote file name. **(Expression, default: <none>)**

sftp.mode

Action to take if the remote file already exists. **(FileExistsMode, default: <none>, possible values: APPEND,APPEND_NO_FLUSH,FAIL,IGNORE,REPLACE,REPLACE_IF_MODIFIED)**

sftp.remote-dir

The remote FTP directory. **(String, default: /)**

sftp.remote-file-separator

The remote file separator. **(String, default: /)**

sftp.tmporary-remote-dir

A temporary directory where the file will be written if 'isUseTemporaryFilename()' is true. **(String, default: /)**

sftp.tmp-file-suffix

The suffix to use while the transfer is in progress. **(String, default: .tmp)**

sftp.use-temporary-filename

Whether or not to write to a temporary file and rename. **(Boolean, default: true)**

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar sftp_sink.jar --sftp.remote-dir=bar --sftp.factory.host=sftpserver \  
--sftp.factory.username=user --sftp.factory.password=pw
```

4.20 TCP Sink

This module writes messages to TCP using an Encoder.

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'.

Input

Headers:

- Content-Type: application/octet-stream

Payload:

- byte[]

Headers:

- Content-Type: text/plain

Payload:

- String

Output

N/A

Options

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

tcp.charset

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

tcp.close

Whether to close the socket after each message. (**Boolean, default: false**)

tcp.encoder

The encoder to use when sending messages. (**Encoding, default: <none>, possible values: CRLF,LF,NULL,STXETX,RAW,L1,L2,L4**)

tcp.host

The host to which this sink will connect. (**String, default: <none>**)

tcp.nio

Whether or not to use NIO. (**Boolean, default: false**)

tcp.port

The port on which to listen; 0 for the OS to choose a port. **(Integer, default: 1234)**

tcp.reverse-lookup

Perform a reverse DNS lookup on the remote IP Address; if false, just the IP address is included in the message headers. **(Boolean, default: false)**

tcp.socket-timeout

The timeout (ms) before closing the socket when no data is received. **(Integer, default: 120000)**

tcp.use-direct-buffers

Whether or not to use direct buffers. **(Boolean, default: false)**

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)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar tcp_sink.jar --tcp.encoder=LF
```

4.21 Throughput Sink

A simple handler that will count messages and log witnessed throughput at a selected interval.

Input

Headers

Payload

Any

Output

N/A

Options

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects [here](#). You can then cd into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

```
java -jar throughput-sink.jar
```

4.22 Websocket Sink

A simple Websocket Sink implementation.

Input

Headers

Payload

Any

Output

N/A

Options

The following command line arguments are supported:

websocket.log-level

the logLevel for netty channels. Default is `WARN` (**String, default: `<none>`**)

`websocket.path`

the path on which a `WebsocketSink` consumer needs to connect. Default is `<tt>/websocket</tt>` (**String, default: `/websocket`**)

`websocket.port`

the port on which the Netty server listens. Default is `<tt>9292</tt>` (**Integer, default: `9292`**)

`websocket.ssl`

whether or not to create a `{@link io.netty.handler.ssl.SslContext}` (**Boolean, default: `false`**)

`websocket.threads`

the number of threads for the Netty `{@link io.netty.channel.EventLoopGroup}`. Default is `<tt>1</tt>` (**Integer, default: `1`**)

Build

```
$ ./mvnw clean install -PgenerateApps
$ cd apps
```

You can find the corresponding binder based projects here. You can then `cd` into one of the folders and build it:

```
$ ./mvnw clean package
```

Examples

To verify that the `websocket-sink` receives messages from other `spring-cloud-stream` apps, you can use the following simple end-to-end setup.

Step 1: Start Rabbitmq

Step 2: Deploy a `time-source`

Step 3: Deploy a `websocket-sink` (the app that contains this starter jar)

Finally start a `websocket-sink` in `trace` mode so that you see the messages produced by the `time-source` in the log:

```
java -jar <spring boot application for websocket-sink> --spring.cloud.stream.bindings.input=ticktock --
server.port=9393 \
--logging.level.org.springframework.cloud.stream.module.websocket=TRACE
```

You should start seeing log messages in the console where you started the `WebsocketSink` like this:

```
Handling message: GenericMessage [payload=2015-10-21 12:52:53, headers={id=09ae31e0-a04e-b811-d211-
b4d4e75b6f29, timestamp=1445424778065}]
Handling message: GenericMessage [payload=2015-10-21 12:52:54, headers={id=75eaf30-e5c6-494f-
b007-9d5b5b920001, timestamp=1445424778065}]
Handling message: GenericMessage [payload=2015-10-21 12:52:55, headers={id=18b887db-81fc-
c634-7a9a-16b1c72de291, timestamp=1445424778066}]
```

Actuators

There is an Endpoint that you can use to access the last `n` messages sent and received. You have to enable it by providing `--endpoints.websocketSinkTrace.enabled=true`. By default it shows the last 100 messages via the [host:port/websocketSinkTrace](#). Here is a sample output:

```
[
  {
    "timestamp": 1445453703508,
    "info": {
      "type": "text",
      "direction": "out",
      "id": "2ff9be50-c9b2-724b-5404-1a6305c033e4",
      "payload": "2015-10-21 20:54:33"
    }
  },
  ...
  {
    "timestamp": 1445453703506,
    "info": {
      "type": "text",
      "direction": "out",
      "id": "2b9dbcaf-c808-084d-a51b-50f617ae6a75",
      "payload": "2015-10-21 20:54:32"
    }
  }
]
```

There is also a simple HTML page where you see forwarded messages in a text area. You can access it directly via [host:port](#) in your browser



Note

For SSL mode (`--ssl=true`) a self signed certificate is used that might cause troubles with some Websocket clients. In a future release, there will be a `--certificate=mycert.cer` switch to pass a valid (not self-signed) certificate.

Part III. Appendices

Appendix A. Building

A.1 Basic Compile and Test

To build the source you will need to install JDK 1.7.

The build uses the Maven wrapper so you don't have to install a specific version of Maven. To enable the tests for Redis you should run the server before building. See below for more information on how to run Redis.

The main build command is

```
$ ./mvnw clean install
```

You can also add '-DskipTests' if you like, to avoid running the tests.



Note

You can also install Maven ($\geq 3.3.3$) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.



Note

Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

The projects that require middleware generally include a `docker-compose.yml`, so consider using [Docker Compose](#) to run the middleware servers in Docker containers. See the README in the [scripts demo repository](#) for specific instructions about the common cases of mongo, rabbit and redis.

A.2 Documentation

There is a "full" profile that will generate documentation. You can build just the documentation by executing

```
$ ./mvnw package -DskipTests=true -P full -pl spring-cloud-stream-app-starters-docs -am
```

A.3 Working with the code

If you don't have an IDE preference we would recommend that you use [Spring Tools Suite](#) or [Eclipse](#) when working with the code. We use the [m2eclipse](#) eclipse plugin for maven support. Other IDEs and tools should also work without issue.

Importing into eclipse with m2eclipse

We recommend the [m2eclipse](#) eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".

Unfortunately m2e does not yet support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the `.settings.xml` file for the projects. If you do not do this

you may see many different errors related to the POMs in the projects. Open your Eclipse preferences, expand the Maven preferences, and select User Settings. In the User Settings field click Browse and navigate to the Spring Cloud project you imported selecting the `.settings.xml` file in that project. Click Apply and then OK to save the preference changes.

**Note**

Alternatively you can copy the repository settings from [.settings.xml](#) into your own `~/m2/settings.xml`.

Importing into eclipse without m2eclipse

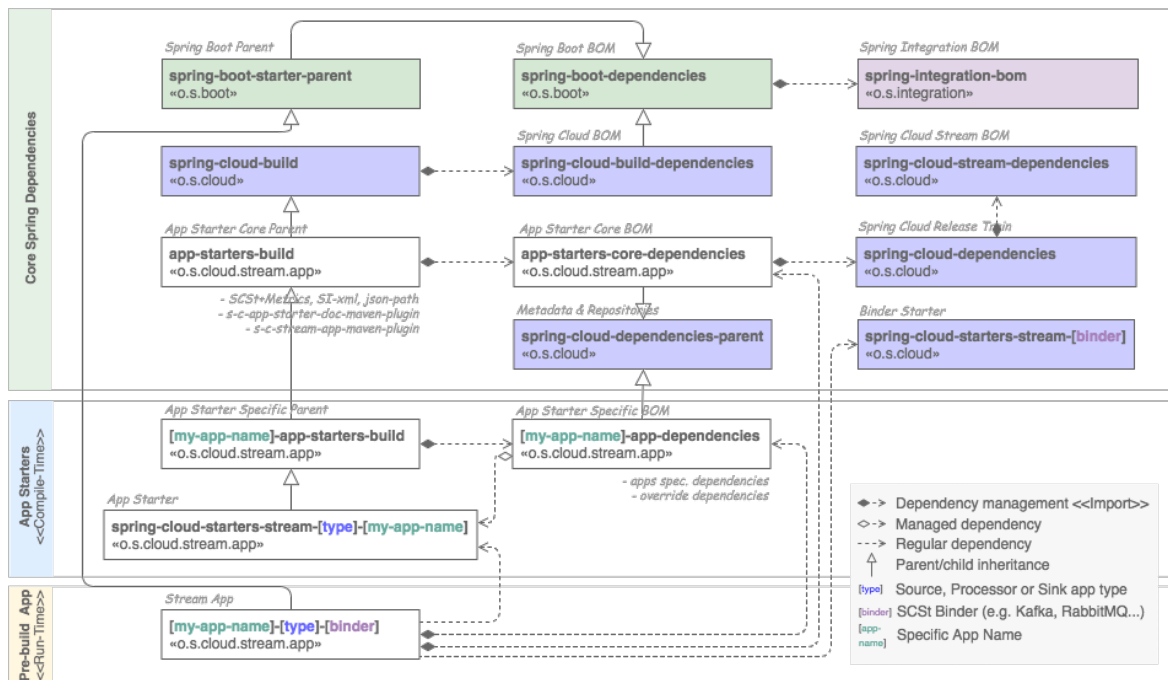
If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.

Appendix B. App Starter POM Dependencies

Following diagram highlights some of the important Stream App and Stream App Starter POM dependencies.



The dependencies are grouped in three categories:

- **Core Spring libraries** - represent the core framework libraries such as Spring Boot, Spring Integration, Spring Cloud. The "Bill Of Materials" (BOM) pattern is used throughout the stack to decouple the dependency management from the lifecycle configurations. The app-starters-build parent POM and the app-starters-core-dependencies BOM use inherit by all app starters.
- **App Starters** - libraries that contain the complete configuration of a Spring Cloud Stream application with a specific role Starters are not executable applications, and are intended to be included in the Pre-build Apps, along with a Binder implementation. The App Starter root pom ([my-app-name]-app-starters-build) inherit all compile-time configuration for its parent the core app-starters-build. Starter's BOM [my-app-name]-app-dependencies is used to manage starter's own dependencies.
- **Pre-build App** - pre-build Spring Boot applications that include the app starters and a Binder implementation.

Appendix C. App Starter Naming Conventions

The `spring-cloud-stream-app-maven-plugin` (used to generate the Pre-build Apps) asserts a naming convention over certain starter's resources.

Following diagram describes which resources are involved and how the convention is applied to them.



The `[type]` placeholder represents the application type and must be either `Source`, `Processor` or `Sink` values. The `[my-app-name]` placeholder represents the name of your app starter project. For multi-word app names, the hyphens (-) is used as word delimiter (e.g. `my-app-name`). Mind that for package names the hyphen delimiter is replaced by (`.`) character (e.g. `o.s.c.s.a.my.app.name`). Class name convention expects CamelCase names in place of any delimiters (e.g. `MyAppNameSourceConfiguration`).

The capital letters in the placeholders are relevant. For example the `[type]` refers to lower case names such as `source`, `processor` or `sink` types, while capitalized placeholder `[Type]` refers to names like `Source`, `Processor` and `Sink`.

The `Configuration` and `Properties` class suffixes are expected as well.

With the help of the maven plugin configuration all default conventions can be customized or replaced. More information about the maven plugin can be found here: github.com/spring-cloud/spring-cloud-stream-app-maven-plugin

5. Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

5.1 Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the [contributor's agreement](#). Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

5.2 Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the [Spring Cloud Build](#) project. If using IntelliJ, you can use the [Eclipse Code Formatter Plugin](#) to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the `.java` files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow [these conventions](#), if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).