Spring Cloud Stream App Starters Reference Guide

1.0.1.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

Table of Contents

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 predefined Spring Cloud Stream 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 and sinks) for middleware 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.

1.1 Starters and 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 a Binder implementation.

Prebuilt applications are Spring Boot applications that include the starters and a Binder implementation. Prebuilt applications are uberjars and include minimal code required to execute standalone. For each starter, the project provides a prebuilt version including the Kafka Binder and a prebuilt version including the Rabbit MQ Binder.

Note

Only starters are present in the source code of the project. Prebuilt applications are generated according to the Maven plugin configuration.

1.2 Classification

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

- a *source* that connects to an external resource to receive data that is sent on its sole output channel;
- a *processor* that receives data from a single input channel and processes it, sending the result on its single output channel;
- a *sink* that connects to an external resource to send data that is received on its sole 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` is a *Cassandra sink* using the Kafka binder.

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>
  <group>org.springframework.cloud.stream.app</group>
  <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
```

Just as with the starters, you can infer the coordinates for other prebuilt applications found in the guide. The group will be always `org.springframework.cloud.stream.app`. The version may vary. The artifact id follows the format `<functionality>-<type>-<binder>` previously described.

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
```

will pull the *latest* Docker image of the *Cassandra sink* with the Kafka binder.

Building the artifacts

You can also 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, for example for adding a new starter, or if you want to build the entire set of artifacts with a new binder.

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 corresponding script in the root of the project.

```
./generate.sh
```

For the each of the prebuilt applications, the script will generate the following items:

- `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 exercises the component against the configured binder.

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

1.4 Creating 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

If you want to use one of the applications found in Spring Cloud Stream Application Starters and you want to use one of the predefined binders (i.e. Kafka or Rabbit), you can just use the prebuilt versions of the artifacts. But if you want to connect to a different middleware system, and you have a binder for it, you will 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. For example, in the same case of the Cassandra sink it can look like the following:

```
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);
 }
}
```

Creating your own applications

Spring Cloud Stream Application Starters consists of regular Spring Cloud Stream 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 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;

Using generic Spring Cloud Stream applications

If you want to add your own custom applications to your solution, you can simply create a new Spring Cloud Stream 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. One restriction is that the applications must have:

- a single inbound channel named `input` for sources - the simplest way to do so is by using the predefined interface `org.spring.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.spring.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.spring.cloud.stream.messaging.Processor`.

The other restriction is to use the same kind of binder as the rest of your solution.

Using the starters to create custom components

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. For doing so 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 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 appliation 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 packages.

```
@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 standalone *uberjars*

1.6 Contributing 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 new top level module named spring-cloud-starter-stream-source-foobar

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.springfamework.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.

- Add the new foobar source module to the root pom of the repository
- At the root of the repository build and install the new module into your local maven cache:

```
./mvnw clean install -pl :spring-cloud-starter-stream-source-foobar
```

- You need to add the new starter dependency to the `spring-cloud-stream-app-dependencies` bill of material (BOM) in the dependecy 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>
...
...
```

- Build and install the newly updated bom:

```
./mvnw clean install -pl :spring-cloud-stream-app-dependencies
```

- At this point, you are ready to generate the binder based spring boot apps for foobar-source. Go to the `spring-cloud-stream-app-generator` module and start editing as below.

The minimal configuration needed to generate the app is to add to plugin configuration in spring-cloud-stream-app-generator/pom.xml. There are other plugin options that customize the generated applications which are described in the plugin documentation (github.com/spring-cloud/spring-cloud-stream-app-maven-plugin). A few plugin features are described below.

```
<generatedApps>
....
    <foobar-source />
....
</generatedApps>
```

More information about the maven plugin used above 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 of where it is looking for the main configuration class, which is `org.springfamework.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>
```

- Go to the root of the repository and execute the script: `./generateApps.sh`

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 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>
    <kafka />
    <rabbit />
    <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 both `foobar-source-kafka` and `foobar-source-rabbit` along with all the other out of the box apps that is generated. If you added more binders as described above, you would see that app as well here - for example foobar-source-jms.

If you only care about the foobar-source apps and nothing else, you can cd into those particular foo-bar-[binder] directories and import them directly into your IDE of choice. Each of them is a self contained spring boot application project. For all 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`

`mvn clean install`

This would install the foo-source-kafka 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 be present in the context, these tests would 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 would 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>

  <extraTestConfigClass>org.springframework.cloud.stream.app.test.foobar.FoobarTestConfiguration.class</
extraTestConfigClass>
</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 rerun the script for generating 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.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 repository 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 to the main repository and upon review, this app then can be uploaded to the above location in docker hub.

If you still need to push this to docker hub under a different repository 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.

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.

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.

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.

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

ftp.delete-remote-files
: <documentation missing> **(Boolean, default: `<none>`)**

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

ftp.filename-regex
: <documentation missing> **(Pattern, default: `<none>`)**

ftp.local-dir
: <documentation missing> **(File, default: `<none>`)**

ftp.preserve-timestamp
: <documentation missing> **(Boolean, default: `<none>`)**

ftp.remote-dir
: <documentation missing> **(String, default: `<none>`)**

ftp.remote-file-separator
: <documentation missing> **(String, default: `<none>`)**

ftp.tmp-file-suffix
: <documentation missing> **(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`)**

2.3 Http Source

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

Options

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

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

server.port
: Server HTTP port. **(Integer, default: <none>)**

2.4 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.

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.driver-class-name
: <documentation missing> **(String, default: <none>)**

spring.datasource.init-sql
: <documentation missing> **(String, default: <none>)**

spring.datasource.initialize
: Populate the database using 'data.sql'. **(Boolean, default: `true`)**

spring.datasource.password
: <documentation missing> **(String, default: <none>)**

spring.datasource.url
: <documentation missing> **(String, default: <none>)**

spring.datasource.username
: <documentation missing> **(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 addition `DataSource` properties and `TriggerProperties` and `MaxMessagesProperties` for polling options.

2.5 JMS Source

The "jms" source enables receiving messages from JMS.

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
: Specify if 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.

2.6 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.

Options

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

Unresolved directive in sources.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-stream-app-starters/master/testing/spring-cloud-starter-stream-source-mail/README.adoc[tags=ref-doc]

Unresolved directive in sources.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-stream-app-starters/master/testing/spring-cloud-starter-stream-source-mongodb/README.adoc[tags=ref-doc]

2.7 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.

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

rabbit.requeue
: Whether rejected messages should be requeued. **(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 to. **(String, default: `<none>`)**

spring.rabbitmq.host
: RabbitMQ host. **(String, default: `localhost`)**

spring.rabbitmq.password
: Login to authenticate against the broker. **(String, default: `<none>`)**

spring.rabbitmq.port
: RabbitMQ port. **(Integer, default: `5672`)**

spring.rabbitmq.requested-heartbeat
: Requested heartbeat timeout, in seconds; zero for none. **(Integer, default: `<none>`)**

spring.rabbitmq.username
: Login user to authenticate to the broker. **(String, default: `<none>`)**

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).

Unresolved directive in sources.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-stream-app-starters/master/s3/spring-cloud-starter-stream-source-s3/README.adoc[tags=ref-doc]

2.8 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.

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

sftp.delete-remote-files
: <documentation missing> **(Boolean, default: <none>)**

sftp.factory.allow-unknown-keys
: True to allow an unknown or changed key. **(Boolean, default: `false`)**

sftp.factory.cache-sessions
: <documentation missing> **(Boolean, default: <none>)**

sftp.factory.host
: <documentation missing> **(String, default: <none>)**

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

sftp.factory.pass-phrase
: Passphrase for user's private key. **(String, default: `<empty string>`)**

sftp.factory.password
: <documentation missing> **(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
: <documentation missing> **(String, default: <none>)**

sftp.filename-pattern
: <documentation missing> **(String, default: <none>)**

sftp.filename-regex
: <documentation missing> **(Pattern, default: <none>)**

sftp.local-dir
: <documentation missing> **(File, default: <none>)**

sftp.preserve-timestamp
: <documentation missing> **(Boolean, default: <none>)**

sftp.remote-dir
: <documentation missing> **(String, default: <none>)**

sftp.remote-file-separator
: <documentation missing> **(String, default: <none>)**

sftp.tmp-file-suffix
: <documentation missing> **(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`)**

2.9 SYSLOG Source

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

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`)**

2.10 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.

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

tcp.port
: <documentation missing> **(Integer, default: <none>)**

tcp.reverse-lookup
: <documentation missing> **(Boolean, default: <none>)**

tcp.socket-timeout
: <documentation missing> **(Integer, default: <none>)**

tcp.use-direct-buffers
: <documentation missing> **(Boolean, default: <none>)**

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)

2.11 Time Source

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

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`)**

2.12 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.

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`)**

2.13 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.

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

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

Note

`twitterstream` emit JSON in the native Twitter format.

3. Processors

3.1 Bridge Processor

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

3.2 Filter Processor

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

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`)**

3.3 Groovy Filter Processor

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

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>)**

3.4 Groovy Transform Processor

A Processor module that transforms messages using a Groovy script.

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>)**

3.5 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.

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

3.6 PMML Processor

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

Options

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

pmml.inputs
: How to compute model active fields from input message properties as modelField->SpEL. **(java.util.Map<java.lang.String,org.springframework.expression.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. **(java.util.Map<java.lang.String,org.springframework.expression.Expression>, default: <none>)**

3.7 Scripable 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).

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>)**

3.8 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.

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.

3.9 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. For example, `--expression=payload.toUpperCase()`.

This transform will convert all message payloads to upper case.

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>')

Options

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

transformer.expression
    <documentation missing> **(Expression, default: `payload`)**

4. Sinks

4.1 Cassandra Sink

This sink application writes the content of each message it receives into Cassandra.

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`)**

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`,`RECREATE`,`RECREATE_DROP_UNUSED`)**

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`,`QUOROM`,`LOCAL_QUOROM`,`EACH_QUOROM`,`ALL`,`LOCAL_ONE`,`SERIAL`,`LOCAL_SERIAL`)**

cassandra.ingest-query
: The ingest Cassandra query. **(String, default: <none>)**

cassandra.query-type
: The queryType for Cassandra Sink. **(org.springframework.integration.cassandra.outbound.CassandraMessageHandler<T> $Type, default: <none>)**

cassandra.retry-policy
: The retryPolicy option of WriteOptions. **(RetryPolicy, 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`)**

4.2 Counter Sink

The counter sink simply counts the number of messages it receives, optionally storing counts in a separate store such as redis.

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.timeout
: Connection timeout in milliseconds. **(Integer, default: `0`)**

4.3 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. This sinks supports the following payload types out of the box:

- POJO (Java bean)

- Tuple
- JSON String

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

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

If the stream source produces messages with the following objects:

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

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

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

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

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

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

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

Options

The **field-value-counter** sink has the following options:

field-value-counter.field-name
: <documentation missing> **(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.timeout
: Connection timeout in milliseconds. **(Integer, default: `0`)**

4.4 File Sink

This module writes each message it receives to a file.

Options

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

file.binary
: A flag to indicate whether content should be written as bytes. **(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`,`FAIL`,`IGNORE`,`REPLACE`)**

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>`)**

4.5 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 could 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.

Options

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

ftp.auto-create-dir
: <documentation missing> **(Boolean, default: <none>)**

ftp.filename-expression
: <documentation missing> **(Expression, default: <none>)**

ftp.mode
: <documentation missing> **(FileExistsMode, default: <none>, possible values: `APPEND`,`FAIL`,`IGNORE`,`REPLACE`)**

ftp.remote-dir
: <documentation missing> **(String, default: <none>)**

ftp.remote-file-separator
: <documentation missing> **(String, default: <none>)**

ftp.temporary-remote-dir
: <documentation missing> **(String, default: <none>)**

ftp.tmp-file-suffix
: <documentation missing> **(String, default: <none>)**

ftp.use-temporary-filename
: <documentation missing> **(Boolean, default: <none>)**

4.6 Gemfire Sink

The Gemfire sink allows one to write message payloads to a Gemfire server.

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>)**

4.7 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` protcol and schedules a task that orchestrates a `gploadd` 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.

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.error-table
: Tablename to log errors. (String, default: ``) **(String, default: <none>)**

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.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: \`\`) **(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 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 **(java.util.List<java.lang.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 INTO error_table] SEGMENT REJECT LIMIT count
  [ROWS | PERCENT] ]
```

  As a conveniance 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 INTO error_table] SEGMENT REJECT LIMIT count
  [ROWS | PERCENT] ]
```

errorTable
: Tablename to log errors. **(String, default: ``)**

As error table is optional with `SEGMENT REJECT LIMIT`, it's only used if both `segmentRejectLimit` and `segmentRejectType` are set. Sets `error_table` in below clause segment.

```
[ [LOG ERRORS INTO error_table] 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 throught 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 windows. 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 send 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 buzy loop. Buzy loop would be ok if there is a constant stream of data coming in but if incoming data is more like bursts then buzy 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 enought 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 throught 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 adviced 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.

4.8 HDFS Sink

This module writes each message it receives to HDFS.

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 it's runtime dependencies provided during startup if you would like to use a Hadoop distribution other than the default one.

4.9 Jdbc Sink

A module that writes its incoming payload to an RDBMS using JDBC.

Options

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

jdbc.columns
: The names of the columns that shall receive data, as a set of column[:SpEL] mappings. Also used at initialization time to issue the DDL. **(java.util.Map<java.lang.String,java.lang.String>, default: <none>)**

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: `<none>`)**

spring.datasource.driver-class-name
: <documentation missing> **(String, default: `<none>`)**

spring.datasource.init-sql
: <documentation missing> **(String, default: `<none>`)**

spring.datasource.initialize
: Populate the database using 'data.sql'. **(Boolean, default: `true`)**

spring.datasource.password
: <documentation missing> **(String, default: `<none>`)**

spring.datasource.url
: <documentation missing> **(String, default: `<none>`)**

spring.datasource.username
: <documentation missing> **(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.

4.10 Log Sink

The `log` sink uses the application logger to output the data for inspection.

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>`)**

4.11 RabbitMQ Sink

This module sends messages to RabbitMQ.

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

spring.rabbitmq.host
: RabbitMQ host. **(String, default: `localhost`)**

spring.rabbitmq.password
: Login to authenticate against the broker. **(String, default: <none>)**

spring.rabbitmq.port
: RabbitMQ port. **(Integer, default: `5672`)**

spring.rabbitmq.requested-heartbeat
: Requested heartbeat timeout, in seconds; zero for none. **(Integer, default: <none>)**

spring.rabbitmq.username
: Login user to authenticate to the broker. **(String, default: <none>)**

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.

4.12 Redis Sink

This module sends messages to Redis store.

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

spring.redis.pool.max-active
: Max 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.pool.max-idle
: Max number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections. **(Integer, default: `8`)**

spring.redis.pool.max-wait
: Maximum amount of time (in milliseconds) a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely. **(Integer, default: `-1`)**

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

spring.redis.sentinel.master
: Name of Redis server. **(String, default: <none>)**

spring.redis.sentinel.nodes
: Comma-separated list of host:port pairs. **(String, default: `<none>`)**

spring.redis.timeout
: Connection timeout in milliseconds. **(Integer, default: `0`)**

4.13 Router Sink

This module routes messages to named channels.

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.

Unresolved directive in sinks.adoc - include::https://raw.githubusercontent.com/spring-cloud/spring-cloud-stream-app-starters/master/s3/spring-cloud-starter-stream-sink-s3/README.adoc[tags=ref-doc]

4.14 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'.

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

tcp.port
: <documentation missing> **(Integer, default: <none>)**

tcp.reverse-lookup
: <documentation missing> **(Boolean, default: <none>)**

tcp.socket-timeout
: <documentation missing> **(Integer, default: <none>)**

tcp.use-direct-buffers
: <documentation missing> **(Boolean, default: <none>)**

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)

4.15 Throughput Sink

A simple handler that will count messages and log witnessed throughput at a selected interval.

4.16 Websocket Sink

A simple Websocket Sink implementation.

Options

The following commmand line arguments are supported:

websocket.log-level
: the logLevel for netty channels. Default is <tt>WARN</tt> **(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`)**

Example

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 Redis

The default broker that is used is Redis. Normally can start Redis via `redis-server`.

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=75eaaf30-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 bulding. See below for more information on how 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 (>=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 middeware 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 m2eclipe eclipse plugin for maven support. Other IDEs and tools should also work without issue.

Importing into eclipse with m2eclipse

We recommend the m2eclipe 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.

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).