Spring Cloud Stream Reference Guide

1.0.0.M3

Table of Contents

I. Reference Guide 1
- 1. Spring Cloud Stream Overview 2
  - 1.1. Introducing Spring Cloud Stream 2
    - Multiple Input or Output Channels 3
    - Inter-module communication 3
    - Advanced binding properties 3
      - Partitioning 4
  - 1.2. Binder selection 5
    - Classpath detection 5
    - Multiple binders on the classpath 5
    - Connecting to multiple systems 6
  - 1.3. Managed vs standalone 6
    - Fat JAR 6

II. Samples 7
- 2. Sample Applications 8

III. Appendices 9
- A. Building 10
  - A.1. Basic Compile and Test 10
  - A.2. Documentation 10
  - A.3. Working with the code 10
    - Importing into eclipse with m2eclipse 10
    - Importing into eclipse without m2eclipse 11
  - A.4. Sign the Contributor License Agreement 11
  - A.5. Code Conventions and Housekeeping 11

Part I. Reference Guide

1. Spring Cloud Stream Overview

1.1 Introducing Spring Cloud Stream

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

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

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

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

}

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

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

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

}
```

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

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

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

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

 @Autowired
 private Source source

 @Test
```

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

}
```

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

Multiple Input or Output Channels

A module can have multiple input or output channels defined as `@Input` and `@Output` methods in an interface. Instead of just one channel named "input" or "output" you can add multiple `MessageChannel` methods annotated `@Input` or `@Output` and their names will be converted to external channel names on the broker. It is common to specify the channel names at runtime in order to have multiple modules communicate over a well known channel names. Channel names can be specified as properties that consist of the channel names prefixed with `spring.cloud.stream.bindings` (e.g. `spring.cloud.stream.bindings.input` or `spring.cloud.stream.bindings.output`). These properties can be specified though environment variables, the application YAML file or the other mechanism supported by Spring Boot.

Channel names can also have a channel type as a colon-separated prefix, and the semantics of the external bus channel changes accordingly. For example, you can have two `MessageChannels` called "output" and "foo" in a module with `spring.cloud.stream.bindings.output=bar` and `spring.cloud.stream.bindings.foo=topic:foo`, and the result is 2 external channels called "bar" and "topic:foo". The queue prefix for point to point semantics is also supported. Note, that in a future release only topic (pub/sub) semantics will be supported.

Inter-module communication

While Spring Cloud Stream makes it easy for individual modules to connect to messaging systems, the typical scenario for Spring Cloud Stream is the creation of multi-module pipelines, where modules are sending data to each other. This can be achieved by correlating the input and output destinations of adjacent modules, as in the following example.

Supposing that the design calls for the `time-source` module to send data to the `log-sink` module, we will use a common destination named `foo` for both modules. `time-source` will set `spring.cloud.stream.bindings.output=foo` and `log-sink` will set `spring.cloud.stream.bindings.input=foo`.

Advanced binding properties

The input and output channel names are the common properties to set in order to have Spring Cloud Stream applications communicate with each other as the channels are bound to an external message broker automatically. However, there are a number of scenarios when it is required to configure other attributes besides the channel name. This is done using the following naming scheme: spring.cloud.stream.bindings.<channelName>.<attributeName>=<attributeValue>. The `destination` attribute can also be used for configuring the external channel, as follows: `spring.cloud.stream.bindings.input.destination=foo`. This is equivalent to `spring.cloud.stream.bindings.input=foo`, but the latter can be used only when there are no other attributes to set on the binding. In other words, `spring.cloud.stream.bindings.input.destination=foo`,`spring.cloud.stream.bindings.input.pa` is a valid setup, whereas `spring.cloud.stream.bindings.input=foo`,`spring.cloud.stream.bindings.input.partitioned=tr` is not valid.

Partitioning

Spring Cloud Stream provides support for partitioning data between multiple instances of a given application. In a partitioned scenario, one or more producer modules will send data to one or more consumer modules, ensuring that data with common characteristics is processed by the same consumer instance. The physical communication medium (i.e. the broker topic or queue) is viewed as structured into multiple partitions. Regardless whether the broker type is naturally partitioned (e.g. Kafka) or not (e.g. Rabbit or Redis), Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion.

Setting up a partitioned processing scenario requires configuring both the data producing and the data consuming end.

Configuring output channels for partitioning

An output channel is configured to send partitioned data, by setting one and only one of its `partitionKeyExpression` or `partitionKeyExtractorClass` properties, as well as its `partitionCount` property. For example seting `spring.cloud.stream.bindings.output.partitionKeyExpression=payload.id`,`spring.cloud.strea` is a valid and typical configuration.

Based on this configuration, the data will be sent to the target partition using the following logic. A partition key's value is calculated for each message sent to a partitioned output channel based on the `partitionKeyExpression`. The `partitionKeyExpression` is a SpEL expression that is evaluated against the outbound message for extracting the partitioning key. If a SpEL expression is not sufficent for your needs, you can instead calculate the partition key value by setting the the property `partitionKeyExtractorClass`. This class must implement the interface `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy`. While, in general, the SpEL expression is enough, more complex cases may use the custom implementation strategy.

Once the message key is calculated, the partition selection process will determine the target partition as a value between `0` and `partitionCount`. The default calculation, applicable in most scenarios is based on the formula `key.hashCode() % partitionCount`. This can be customized on the binding, either by setting a SpEL expression to be evaluated against the key via the `partitionSelectorExpression` property, or by setting a `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` implementation via the `partitionSelectorClass` property.

Additional properties can be configured for more advanced scenarios, as described in the following section.

Configuring input channels for partitioning

An input channel is configured to receive partitioned data by setting its `partitioned` binding property, as well as the instance index and instance count properties on the module, as follows:

`spring.cloud.stream.bindings.input.partitioned=true,spring.cloud.stream.instanceIndex=3,s`
The instance count value represents the total number of similar modules between which the data needs to be partitioned, whereas instance index must be value unique across the multiple instances between `0` and `instanceCount - 1`. The instance index helps each module to identify the unique partition (or in the case of Kafka, the partition set) that they receive data from. It is important that both values are set correctly in order to ensure that all the data is consumed, as well as that the modules receive mutually exclusive datasets.

While setting up multiple instances for partitioned data processing may be complex in the standalone case, Spring Cloud Data Flow can simplify the process significantly, by populating both the input and output values correctly, as well as relying on the runtime infrastructure to provide information about the instance index and instance count.

1.2 Binder selection

Spring Cloud Stream relies on implementations of the Binder SPI to perform the task of connecting channels to message brokers. Each binder implementation typically connects to one type of messaging system. Spring Cloud Stream provides out of the box binders for Redis, Rabbit and Kafka.

Classpath detection

By default, Spring Cloud Stream relies on Spring Boot's auto-configuration configure the binding process. If a single binder implementation is found on the classpath, Spring Cloud Stream will use it automatically. So, for example, a Spring Cloud Stream project that aims to connect to Rabbit MQ can simply add the following dependency to their application:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

Multiple binders on the classpath

When multiple binders are present on the classpath, the application must indicate what binder has to be used for the channel. Each binder configuration contains a `META-INF/spring.binders`, which is in fact a property file:

```
rabbit:\
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration
```

Similar files exist for the other binder implementations (i.e. Kafka and Redis), and it is expected that custom binder implementations will provide them, too. The key represents an identifying name for the binder implementation, whereas the value is a comma-separated list of configuration classes that contain one and only one bean definition of the type `org.springframework.cloud.stream.binder.Binder`.

Selecting the binder can be done globally by either using the `spring.cloud.stream.defaultBinder` property, e.g. `spring.cloud.stream.defaultBinder=redis`, or by individually configuring them on each channel.

For instance, a processor module that reads from Rabbit and writes to Redis can specify the following configuration: `spring.cloud.stream.bindings.input.binder=rabbit,spring.cloud.stream.bindings.output.bin`

Connecting to multiple systems

By default, binders share the Spring Boot autoconfiguration of the application module and create one instance of each binder found on the classpath. In scenarios where a module should connect to more than one broker of the same type, Spring Cloud Stream allows you to specify multiple binder configurations, with different environment settings. Please note that turning on explicit binder configuration will disable the default binder configuration process altogether, so all the binders in use must be included in the configuration.

For example, this is the typical configuration for a processor that connects to two rabbit instances:

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: foo
          binder: rabbit1
        output:
          destination: bar
          binder: rabbit2
      binders:
        rabbit1:
          type: rabbit
          environment:
            spring:
              rabbit:
                host: <host1>
        rabbit2:
          type: rabbit
          environment:
            spring:
              rabbit:
                host: <host2>
```

1.3 Managed vs standalone

Code using the Spring Cloud Stream library can be deployed as a standalone application or be used as a Spring Cloud Data Flow module. In standalone mode your application will run happily as a service or in any PaaS (Cloud Foundry, Lattice, Heroku, Azure, etc.). Spring Cloud Data Flow helps orchestrating the communication between instances, so the aspects of module configuration that deal with module interconnection will be configured transparently.

Fat JAR

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

Part II. Samples

2. Sample Applications

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

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

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

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, Rabbit, and Kafka bindings you should have those servers running before building. See below for more information on running the servers.

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.

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`](.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. [[contributing] == 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.

A.4 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.

A.5 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).