



Solace PubSub+ Connector for AWS S3 Storage Service *User Guide*

Solace Corporation

Version 1.5.1

solace.

Table of Contents

Preface	1
Getting Started	2
Prerequisites	2
Quick Start common steps	2
Quick Start: Running the connector via command line	2
Quick Start: Running the connector via <code>start.sh</code> script	3
Enabling Workflows	7
Configuring Connection Details	8
Solace PubSub+ Connection Details	8
Preventing Message Loss when Publishing to Topic-to-Queue Mappings	8
Connecting to Multiple Systems	8
Message Transforms	10
Example Transformation Use Case	10
Content Type Interpretation	12
Writing Transformation Expressions	13
Content Propagation	14
Saving Intermediate Values	16
Registered Transformation Functions	16
Message Headers	24
Solace Headers	24
Reserved Message Headers	24
Dynamic Producer Destinations	25
Asynchronous Publishing	26
Management and Monitoring Connector	27
Monitoring Connector's States	27
Exposed HTTP/HTTPS Endpoints	27
Health	29
Workflow Health	29
Solace Binder Health	30
Leader Election	31
Leader Election Modes: Standalone / Active-Active	31
Leader Election Mode: Active-Standby	31
Leader Election Management Endpoint	32
Workflow Management	33
Workflow Management Endpoint	33
Workflow States	34
Metrics	35
Connector Meters	35

Add a Monitoring System	36
Security	38
Securing Endpoints	38
Exposed Management Web Endpoints	38
Authentication & Authorization	38
TLS	39
Consuming Object Messages	40
Adding External Libraries	41
Configuration	42
Providing Configuration	42
Converting Canonical Spring Property Names to Environment Variables	42
Spring Profiles	42
Configure Locations to Find Spring Property Files	42
Obtaining Build Information	43
Spring Configuration Options	43
Connector Configuration Options	44
Workflow Configuration Options	45
Logging	48
Configuring Logback	48
License	49
Support	49

Preface

Solace PubSub+ Connector for AWS S3 Storage Service bridges data between the Solace PubSub+ Event Broker and AWS S3 Storage Service servers, enabling seamless and efficient integration of file-based data with your Solace-backed, event-driven architecture and Event Mesh. The connector supports bidirectional data flow, allowing you to read files from SFTP servers, split and process their contents, and publish data to Solace, as well as write Solace events to SFTP servers. The connector can be deployed in standalone or active-standby modes. Each instance supports up to 20 workflows (file-to-message or message-to-file pipelines), reducing the number of connector instances required. The use of various Spring Framework technologies allows for easy configuration of the connector, advanced logging capabilities, and export of live metrics data to external monitoring solutions.

Getting Started

Presuming you're using the default `application.yml` provided with the download, follow one of the quick starts below to connect to a PubSub+ event broker with the AWS S3 Storage Service. The quick starts use default credentials as examples to get started with two workflows enabled, workflow 0 and workflow 1.

Where:

- Workflow 0 consumes messages from the Solace PubSub+ queue, `Solace/Queue/0`, and publishes them to the AWS S3 Storage Service producer destination, `producer-destination`.
- Workflow 1 consumes messages from the AWS S3 Storage Service consumer destination, `consumer-destination`, and publishes them to the Solace PubSub+ topic, `Solace/Topic/1`.

A workflow is the configuration of a flow of messages from a source to a target. The connector supports up to 20 concurrent workflows per instance.



The connector does not provision queues that do not exist.

Prerequisites

- [Solace PubSub+ Event Broker](#)
- AWS S3 Storage Service

Quick Start common steps

These are the steps that are required to run all quick-start examples:

1. Update the provided `samples/config/application.yml` with the values for your deployment.

Quick Start: Running the connector via command line

Run:

```
java -jar pubsubplus-connector-aws-s3-1.5.1.jar --spring.config.additional-location  
=file:samples/config/
```



By default, this command detects any Spring Boot configuration files as per the [Spring Boot's default locations](#).

For more information, see [Configure Locations to Find Spring Property Files](#).

Quick Start: Running the connector via **start.sh** script

For convenience, you can start the connector through the shell script using the following command:

```
chmod 744 ./bin/start.sh
./bin/start.sh [-n NAME] [-l FOLDER] [-p PROFILE] [-c FOLDER] [-ch HOST] [-cp PORT] [-j FILE] [-cm] [-cmh HOST] [-cmp PORT] [-mh HOST] [-mp PORT] [-o OPTIONS] [-b]
```

The script shows you all errors at the same time:

```
./bin/start.sh -l dummy_folder -c dummy_folder -j dummy_file.jar
```

The script shows you all errors at the same time:

Solace PubSub+ Connector for AWS S3 Storage Service

Connector startup failed:

```
Following folder doesn't exists on your filesystem: 'dummy_folder'
Following folder doesn't exists on your filesystem: 'dummy_folder'
Following file doesn't exists on your filesystem: 'dummy_file.jar'
```

In situations where you don't provide a parameter, the script runs with the predefined values as follows:

Parameter	Default Value	Description
-n, --name	application	The name of the connector instance, that is configured in [spring.application.name]. This name impacts on grouping connectors only.
-l, --libs	./libs	The directory that contains the required and optional dependency JAR files, such as Micrometer metrics export dependencies (if configured). If this option is not specified, it will use the current ./libs/ directory.

Parameter	Default Value	Description
<code>-p, --profile</code>	empty, no profile is used	The profile to be used with the connector's configuration. The configuration file named 'application-<profile>.yaml' is used. If this option is not specified, no profile is used.
<code>-c, --config</code>	<code>./</code> or current folder	The path to the folder containing the configuration files to be applied when the connector starts up the chosen profile. If not specified, the current directory is used.
<code>-H, --host</code>	127.0.0.1	Specifies the host where the connector runs.
<code>-P, --port</code>	8090	Specifies the port where connector runs.
<code>-mp, --mgmt_port</code>	9009	Specifies the management port for back calls of current connector from PubSub+ Connector Manager. This parameter is ignored if the <code>-cm</code> parameter is not provided.
<code>-j, --jar</code>	pubsubplus-connector-aws-s3-1.5.1.jar	The path to the specified JAR file to start the connector. If the option is not specified, the default JAR file is used from the current directory.

Parameter	Default Value	Description
<code>-cm, --manager</code>	<code>application</code>	Specifies PubSub+ Connector Manager to use the configuration storage and allows you to enable the cloud configuration for the connector. When this parameter is enabled, you can specify the <code>-mp</code> or <code>--mgmt_port</code> , <code>-H</code> or <code>--host</code> , and <code>-cmh</code> with the <code>-cmp</code> parameters, unless you want to use default values for those parameters. Be aware, this option disable listed parameters to be read from configuration file. In this case, the operator must explicitly specify the parameters for the script, otherwise defaultdefault values are used.
<code>-cmh, --cm_host</code>	<code>127.0.0.1</code>	Specifies the host where Connector Manager is running. This parameter is ignored if the <code>-cm</code> parameter is not provided.
<code>-cmp, --cm_port</code>	<code>9500</code>	Specifies the port where Connector Manager is running. This parameter is ignored if <code>-cm</code> parameter is not provided.
<code>-o, --options</code>	<code>no default values</code>	Specifies the JVM options used on when the connector starts. For example, <code>-Xms64M -Xmx1G</code> .
<code>-tls</code>	<code>N/A</code>	Specifies to use HTTPS instead of HTTP. . When this parameter is used, the configuration file must contain an additional section with the preconfigured paths for the key store and trust store files.
<code>-s, --show</code>	<code>N/A</code>	Performs a dry run (does nothing). The output prints the start CLI command and its raw output and exits. This parameter is useful to check your parameters without running the connector.

Parameter	Default Value	Description
<code>-b, --background</code>	N/A	Runs the connector in the background. No logs are shown and the connector continues running in detached mode.
<code>-h, --help</code>	N/A	Prints the help information and exits.

Script also provides that help information from command line using parameter `-h`.

More configuration example of starting Connector together with Connector Manager are provided by the Connector Manager samples.

Enabling Workflows

The provided `application.yml` enables workflow 0 and 1. To enable additional workflows, define the following properties in the `application.yml`, where `<workflow-id>` is a value between `[0-19]`:

```
spring:
  cloud:
    stream:
      bindings: # Workflow bindings
        input-<workflow-id>:
          destination: <input-destination> # Queue name
          binder: (solace|camel) # Input system
        output-<workflow-id>:
          destination: <output-destination> # Topic name
          binder: (solace|camel) # Output system

solace:
  connector:
    workflows:
      <workflow-id>:
        enabled: true
```



The connector only supports workflows in the directions of:

- `solace` → `AWS S3 Storage Service`
- `AWS S3 Storage Service` → `solace`

For more information about Spring Cloud Stream and the Solace PubSub+ binder, see:

- [Spring Cloud Stream Reference Guide](#)
- [Spring Cloud Stream Binder for Solace PubSub+](#)

Configuring Connection Details

Solace PubSub+ Connection Details

The Spring Cloud Stream Binder for PubSub+ uses [Spring Boot Auto-Configuration for the Solace Java API](#) to configure its session.

In the `application.yml`, this typically is configured as follows:

```
solace:
  java:
    host: tcp://localhost:55555
    msg-vpn: default
    client-username: default
    client-password: default
```

For more information and options to configure the PubSub+ session, see [Spring Boot Auto-Configuration for the Solace Java API](#).

Preventing Message Loss when Publishing to Topic-to-Queue Mappings

If the connector is publishing to a topic that is subscribed to by a queue, messages may be lost if they are rejected. For example, if queue ingress is shutdown.

To prevent message loss, configure `reject-msg-to-sender-on-discard` with the `including-when-shutdown` flag.

Connecting to Multiple Systems

To connect to multiple systems of a same type, use the [multiple binder syntax](#).

For example:

```
spring:
  cloud:
    stream:
      binders:

        # 1st solace binder in this example
        solace1:
          type: solace
          environment:
            solace:
              java:
                host: tcp://localhost:55555

        # 2nd solace binder in this example
```

```

solace2:
  type: solace
  environment:
    solace:
      java:
        host: tcp://other-host:55555

# The only camel binder
camel1:
  type: camel
  # Add `environment` property map here if you need to customize this binder.
  # But for this example, we'll assume that defaults are used.

# Required for internal use
undefined:
  type: undefined
bindings:
  input-0:
    destination: <input-destination>
    binder: camel1
  output-0:
    destination: <output-destination>
    binder: solace1 # Reference 1st solace binder
  input-1:
    destination: <input-destination>
    binder: camel1
  output-1:
    destination: <output-destination>
    binder: solace2 # Reference 2nd solace binder

```

The configuration above defines two binders of type `solace` and one binder of type `camel`, which are then referenced within bindings.

Each binder above is configured independently under `spring.cloud.stream.binders.<binder-name>.environment`.



- When connecting to multiple systems, all binder configuration must be specified using the multiple binder syntax for all binders. For example, under the `spring.cloud.stream.binders.<binder-name>.environment`.
- Do not use single-binder configuration (for example, `solace.java.*` at the root of your `application.yml`) while using the multiple binder syntax.

Message Transforms

The connector allows you to transform the consumed message before it is published to the output destination, by writing Spring Expression Language (SpEL) expressions to map elements from the source message to the target message.



For more information about Spring Expression Language (SpEL) expressions, see [the Spring documentation](#).

To transform messages, the following properties are available for you to use under the workflow configuration prefix, `solace.connector.workflows.<workflow-id>.transform`:

enabled

Must be set to `true` to enable the message transformation capabilities.

source-payload.content-type

The content type to interpret the source payload as.

target-payload.content-type

The content type to interpret the target payload as.

expressions

An ordered list of transformation expressions to apply to the message.

expressions[<index>].transform

A single Spring Expression Language (SpEL) expression to transform the message.

See [Workflow Configuration Options](#) for more information about these properties.

Example Transformation Use Case

Now let's look at a transformation example. Consider a JSON message payload with the following structure:

```
{
  "airline": "ExampleAirline",
  "region": "Ontario",
  "requestId": 44334,
  "flight": {
    "flightModel": "boeing737",
    "flightRoute": "international"
  },
  "origin": "yow",
  "destination": "ewr",
  "status": "boarding",
  "passengers": "300,250,10",
  "lastUpdated": "2024-01-05T14:30:00"
```

}

To process this message, we'll create a transformation that:

1. Builds a dynamic routing header using the flight details
2. Converts the comma-separated `passenger` field into a structured format

Here's the transformation configuration:

```
solace:
  connector:
    workflows:
      0:
        transform:
          enabled: true
          source-payload:
            content-type: application/json
          target-payload:
            content-type: application/json
          expressions:
            # Set the dynamic destination header to the value of
            '#{airline}/{destination}/{origin}' from the source payload
            - transform: "target['headers']['scst_targetDestination'] =
#joinString('/', source['payload']['airline'], source['payload']['destination'],
source['payload']['origin'])"

            # Restructure the 'passengers' field from a comma-separated string into an
            object with three distinct fields
            - transform: "target['payload'] = source['payload']"
            - transform: "var['passengerDistribution'] =
#splitString(source['payload']['passengers'], ',', 3)"
            - transform: "target['payload']['passengers'] = {}" # Overwrite the
            'passengers' field with an empty object
            - transform: "target['payload']['passengers']['capacity'] =
#convertStringToNumber(var['passengerDistribution'][0])"
            - transform: "target['payload']['passengers']['occupied'] =
#convertStringToNumber(var['passengerDistribution'][1])"
            - transform: "target['payload']['passengers']['personnel'] =
#convertStringToNumber(var['passengerDistribution'][2])"
```

After applying this transformation, the result would be:

Output Message Header

```
scst_targetDestination: ExampleAirline/ewr/yow
```

Output Message Payload

```
{
  "airline": "ExampleAirline",
  "region": "Ontario",
  "requestId": 44334,
  "flight": {
    "flightModel": "boeing737",
    "flightRoute": "international"
  },
  "origin": "yow",
  "destination": "ewr",
  "status": "boarding",
  "passengers": {
    "capacity": 300,
    "occupied": 250,
    "personnel": 10
  },
  "lastUpdated": "2024-01-05T14:30:00"
}
```

Content Type Interpretation

The `source-payload.content-type` and `target-payload.content-type` properties define how the source and target payloads are interpreted and serialized.



Content Types are Not Data Types

Content types is how the physical data is logically interpreted by the connector.

For example, a JSON payload can be interpreted from any payload whose data type is a string, a byte array, or a map.

The following content types are supported:

Table 1. Content Types

Content Type	Content Type Category	Description
<code>application/vnd.solace.micro-integration.unspecified</code>	Unspecified	The content type is unspecified and cannot be read nor modified.

Content Type	Content Type Category	Description
<code>application/json</code>	Structured	The content type is interpreted/serialized as JSON. <code>['<key>']</code> To access a JSON object property. <code>[<index>]</code> To access a JSON array element.

Handling of `source-payload.content-type` and `target-payload.content-type` are 2 separate and independent operations:

Content Type Interpretation of the Source Payload

How the source payload is interpreted for reading within transformations.

Content Type Interpretation of the Target Payload

How the target payload is interpreted for writing within transformations, and how it is later serialized for the output.

For example:

- When the `source-payload.content-type` is set to `application/json`, the source payload is interpreted as JSON.
- When the `target-payload.content-type` is set to `application/json`, the target payload is interpreted and will be later serialized as JSON.

So depending on the content type's category, expressions can be written with varying levels of refinement:

Structured Content Types

When the content type is structured, you can write expressions that allows you to drill down into the structure of the source and target payloads.

For example, when the `source-payload.content-type` is set to `application/json`, you can write expressions to access elements within the JSON source payload such as `source['payload']['element'][0]['other']`, which corresponds to reading a JSON element using the JSON path `$.element[0].other`.

Unspecified Content Types

When the content type is unspecified, the source and target payloads are treated as opaque blobs of data, and cannot be written nor read.

Writing Transformation Expressions

For general documentation about writing Spring Expression Language (SpEL) expressions, please consult [the Spring documentation](#).

Always use index notation (`a['b']`) for accessing nested properties

When writing SpEL expressions, we only support and highly recommend to always use index notation (`a['b']`) over dot notation (`a.b`) for accessing nested properties within objects.



Dot notation has some limitations which can result in unexpected behavior. Notably, there is a restricted set of characters which can be used in property names. For example; `a.b-c` would be interpreted as "Get the value of `b` from object `a`, then subtract `c` " instead of "Get the value of the property `b-c` in object `a` ". Instead, `a['b-c']` should have just been used, which works as expected.

So by consistently using index notation, aside from [escaping single and double quotes](#), you would never need to worry about limitations like these.

The following context variables are available for you to use in your message transformation expressions:

Table 2. Context Variables

Variable	Access	Description
<code>source['headers']</code>	Read-Only	A map of key-value pairs representing the headers of the source message.
<code>source['payload']</code>	Read-Only	The payload of the source message as interpreted by <code>source-payload.content-type</code> .
<code>target['headers']</code>	Write-Only	A map of key-value pairs representing the headers of the target message.
<code>target['payload']</code>	Write-Only	The payload of the target message as interpreted by <code>target-payload.content-type</code> .
<code>var</code>	Read/Write	A map of key-value pairs that can be used to store intermediate values during the transformation.

Content Propagation

Headers Propagation

Headers are not propagated to the target message by default. To propagate headers, you must define an expression to set the header you want on the target message.

Example: Propagate the `my-header` header from the source message to the target message

```
solace:
  connector:
    workflows:
      0:
        transform:
          enabled: true
          expressions:
```

```
- transform: "target['headers']['my-header'] = source['headers']['my-header']"
```

Payload Propagation

When no expressions reference `target['payload']`, then the payload is copied from the source message to the target message.



Payload propagation will still adhere to the `source-payload.content-type` and `target-payload.content-type` settings.

So if the `source-payload.content-type` and `target-payload.content-type` are both set to `application/json`, then the payload will still be interpreted and serialized as JSON, even though no transformation expressions are defined.

See [Content Type Interpretation](#) for more information.

If you define at least one expression to set the `target['payload']` or any element within the target payload, then the payload will not be copied from the source message to the target message, and the target payload must be built from scratch.

Example: Set the `prop` element in the JSON target payload to `value`

```
solace:
  connector:
    workflows:
      0:
        transform:
          enabled: true
          source-payload:
            content-type: application/json
          target-payload:
            content-type: application/json
          expressions:
            - transform: "target['payload']['prop'] = 'value'"
```

The resultant target payload will be exactly `{"prop": "value"}` regardless of what was in the source payload.

Example: Copy the JSON source payload to the target payload, then set the `prop` element in the target payload to `value`

```
solace:
  connector:
    workflows:
      0:
        transform:
          enabled: true
          source-payload:
```

```

content-type: application/json
target-payload:
  content-type: application/json
expressions:
  # The source payload is copied to the target payload
  - transform: "target['payload'] = source['payload']"
  # The `prop` element in the target payload is set to `value`
  - transform: "target['payload']['prop'] = 'value'"

```

So suppose the source payload was `{"a": "b"}`, then the resultant target payload will be `{"a": "b", "prop": "value"}`. And as this example implies, you may also use this strategy to copy the source payload to the target payload and then modify parts of the target payload as needed.

Saving Intermediate Values

You can save intermediate values during the transformation process by using the `var` variable.

Advantages of saving values in the `var` variable include:

- Avoiding the need to recompute the same value multiple times.
- Avoiding the need to write complex expressions multiple times.

Example: Save the return value from splitting the `my-header` source header to the `split-result` variable

```

solace:
  connector:
    workflows:
      0:
        transform:
          enabled: true
          expressions:
            # Split the `my-header` source header by `,` and save the result to the
            # `split-result` variable
            - transform: "var['split-result'] = #splitString(source['headers']['my-
header'], ',', 3)"
            # Set the `result-0` header to the first element of the split result
            - transform: "target['headers']['result-0'] = var['split-result'][0]"
            # Set the `result-1` header to the second element of the split result
            - transform: "target['headers']['result-1'] = var['split-result'][1]"
            # Set the `result-2` header to the third element of the split result
            - transform: "target['headers']['result-2'] = var['split-result'][2]"

```

Registered Transformation Functions

The connector provides a set of built-in transformation functions that you can use in your expressions.

Example: Upper case the `my-header` source header and set it to the `upper-case-header` target header

```
solace:
  connector:
    workflows:
      0:
        transform:
          enabled: true
          expressions:
            - transform: "target['headers']['upper-case-header'] =
#upperCaseString(source['headers']['my-header'])"
```

The following table lists the registered transformation functions:

Table 3. Registered Functions: Conversions

Return Type	Function and Description
String	<code>convertBooleanToString(boolean input)</code> Converts a boolean value to a string. Returns <code>"true"</code> if the input value is <code>true</code>, and <code>"false"</code> if the input value is <code>false</code>. Parameters input (boolean) A boolean to be converted to a string. Example: - SpEL: <code>#convertBooleanToString(true)</code> - Returns: <code>"true"</code>
String	<code>convertNumberToString(Number input)</code> Converts a number to the string representation of that number. Parameters input (Number) A number to be converted to string. Example: - SpEL: <code>#convertNumberToString(1)</code> - Returns: <code>"1"</code>

Return Type	Function and Description
Number	<code>convertStringToNumber(String input)</code> Converts a string to a number. • If the input string is not a valid number, this function produces an error. • If the input is a string representation of a special number ("NaN", "Infinity", "-Infinity", or "-0"), then the result contains the corresponding numeric representation. Parameters input (String) A string to be converted to a number. Example: • SpEL: <code>#convertStringToNumber("1")</code> • Returns: <code>1</code>

Table 4. Registered Functions: Numeric

Return Type	Function and Description
Number	<code>absoluteNumber(Number input)</code> Returns the absolute (positive) value of a number. • If the number is positive zero or negative zero, the result is positive zero. • If the number is infinite, the result is positive infinity. • If the number is NaN (Not a Number), the result is NaN. Parameters input (Number) A number. Example: • SpEL: <code>#absoluteNumber(-1)</code> • Returns: <code>1</code>

Return Type	Function and Description
Number	<code>ceilingNumberToDouble(Number input)</code> Rounds up a number to the nearest whole value. The result is always a whole number, but is represented in a floating-point format. For example, the ceiling of <code>4.3</code> is <code>5.0</code> and the ceiling of <code>-4.3</code> is <code>-4.0</code>. • If the input is NaN (Not a Number), infinite, positive zero, or negative zero, then the returned value is the same as the input. • If the input value is less than zero but greater than <code>-1.0</code>, then the result is negative zero. Parameters input (Number) A number to round up. Example: • SpEL: <code>#ceilingNumberToDouble(4.3)</code> • Returns: <code>5.0</code>
Number	<code>floorNumberToDouble(Number input)</code> Rounds down a number to the nearest whole value. The result is always a whole number, but is represented in a floating-point format. For example, the floor of <code>4.3</code> is <code>4.0</code> and the floor of <code>-4.3</code> is <code>-5.0</code>. If the input is NaN (Not a Number), infinite, positive zero, or negative zero, then the returned value is the same as the input. Parameters input (Number) A number to round down. Example: • SpEL: <code>#floorNumberToDouble(4.3)</code> • Returns: <code>4.0</code>

Return Type	Function and Description
Number	maskNumber(Number input, Number mask) Masks a number by replacing it with the specified number. Note: The original value is not retrievable. Parameters input (Number) A number to mask. mask (Number) The mask to apply to the input number. Example: • SpEL: <code>#maskNumber(1, 2)</code> • Returns: <code>2</code>
Number	roundNumberToLong(Number input) Rounds a number to the closest whole number. • Values that have <code>0.5</code> as their fractional parts round toward positive infinity. • If the input is NaN (Not a Number), the result is 0. • If the input is negative infinity or any value less than or equal to the value of -2^{63}, the result is equal to the value of -2^{63}. • If the input is positive infinity or any value greater than or equal to the value of $(2^{63})-1$, the result is equal to the value of $(2^{63})-1$. Parameters input (Number) A number to round. Example: • SpEL: <code>#roundNumberToLong(4.5)</code> • Returns: <code>5</code>

Table 5. Registered Functions: String

Return Type	Function and Description
String	concatString(String... inputs) Concatenates an array of strings. If an element of the array is null, then the string "null" is added to the result. Parameters inputs (String[]) An array of strings to be concatenated. Example: - SpEL: <code>#concatString("a", "b", "c")</code> - Returns: "abc"
String	joinString(String delimiter, String... inputs) Joins an array of strings with the specified delimiter. The delimiter can be multiple characters. If an element of the array is null, then the string "null" is added to the result. Parameters delimiter (String) The character or characters to insert between the elements being joined. inputs (String[]) An array of strings to be joined. Example: - SpEL: <code>#joinString(",", "a", "b", "c")</code> - Returns: "a,b,c"
String	lowerCaseString(String input) Converts a string to lower case. Parameters input (String) A string to be converted to lower case. Example: SpEL: <code>#lowerCaseString("HELLO")</code> Returns: "hello"

Return Type	Function and Description
String	<code>maskString(String input, String mask)</code> Masks a string by replacing it with the specified string. Note: The original value is not retrievable. Parameters input (String) A string to mask. mask (String) The mask to apply to the input string. Example: - SpEL: <code>#maskString("hello", "world")</code> - Returns: <code>"world"</code>
String[]	<code>splitString(String input, String delimiter, int numSplits)</code> Splits a string into the parts that are separated by the specified delimiter. The delimiter is not included in the result. If the input string does not contain enough splits to fill the result array, the remaining array elements contain empty strings. If the string contains more splits than the result array can hold, the last element of the result array contains the remaining substring. Parameters input (String) A string to split. delimiter (String) The delimiter to split the string by. numSplits (int) The number of elements in the result array. Example: - SpEL: <code>#splitString("a,b,c,d", ",", 3)</code> - Returns: <code>["a", "b", "c,d"]</code>

Return Type	Function and Description
String	substringString(String input, int index, int length) Extracts a substring from a string, based on a starting index and a length. An index of 0 denotes the first character of the string. Parameters input (String) A string. index (int) The starting index of the substring. length (int) The length of the substring. Example: - SpEL: <code>#substringString("hello", 1, 3)</code> - Returns: "ell"
String	trimString(String input) Removes the whitespace from both ends of a string. Parameters input (String) A string to trim. Example: - SpEL: <code>#trimString(" hello ")</code> - Returns: "hello"
String	upperCaseString(String input) Converts a string to upper case. Parameters input (String) A string to be converted to upper case. Example: - SpEL: <code>#upperCaseString("hello")</code> - Returns: "HELLO"

Message Headers

Solace and camel headers can be created or manipulated using the [Message Transform](#) feature described above.

Solace Headers

Solace headers exposed to the connector are documented in the [Spring Cloud Stream Binder for Solace PubSub+](#) documentation.

Reserved Message Headers

The following are reserved header spaces:

- `solace_`
- `scst_`
- Any headers defined by the core Spring messaging framework. See [Spring Integration: Message Headers](#) for more info.

Any headers with these prefixes (that are not defined by the connector or any technology used by the connector) may not be backwards compatible in future releases of this connector.

Dynamic Producer Destinations

To route messages to dynamic destinations at runtime, use the [Message Transform](#) feature to set the following headers:

Header Name	Type	Values	Applies To	Description
<code>scst_targetDestination</code>	<code>string</code>	Any valid destination name	Solace, AWS S3 Storage Service	Specifies the name of the dynamic destination to publish to. Setting this header overrides the configured destination.
<code>solace_scst_targetDestinationType</code>	<code>string</code>	<code>(queue topic)</code>	Solace	Specifies the destination type of the dynamic destination. When unspecified, the configured or default destination type is used.



Setting the `scst_targetDestination` header under `solace.connector.default.workflow.*` may not be viable if not all workflows follow the same direction.

Asynchronous Publishing

This connector does not support asynchronous publishing. Publish acknowledgments are resolved synchronously for all workflows regardless of the config option:

```
# <workflow-id> : The workflow ID ([0-19])
```

```
solace.connector.workflows.<workflow-id>.acknowledgment.publish-async=(true|false)
```



Enabling **publish-async** enable asynchronous publishing on the connector's core, but the effective publishing mode is still synchronous because there is no support for this feature on either the consumer binding or the producer binding.

Management and Monitoring Connector

Monitoring Connector's States

The connector provides an ability to monitor its internal states through exposed endpoints provided by [Spring Boot Actuator](#).

An Actuator shares information through the endpoints reachable over HTTP/HTTPS. The endpoints that are available are configured in the connector configuration file.

What endpoints are available is configured in the connector configuration file:

```
management:
  simple:
    metrics:
      export:
        enabled: true
    endpoints:
      web:
        exposure:
          include:
            "health,metrics,loggers,logfile,channels,env,workflows,leaderelection,bindings,info"
```

The above sample configuration enables metrics collection through the configuration parameter of `management.simple.metrics.export.enabled` set to `true` and then shares them through the HTTP/HTTPS endpoint together with other sections configured for the current connector.

Exposed HTTP/HTTPS Endpoints

The set of endpoints exposed through the HTTP/HTTPS endpoint.

- Exposed endpoints are available if you query the endpoints using the web interface (for example `https://localhost:8090/actuator/<some_endpoint>`) and also available in PubSub+ Connector Manager.
- The operator may choose to not expose all or some of these endpoints. If so, the Actuator endpoints that are not exposed are not visible if you query the endpoints (for example, `https://localhost:8090/actuator/<some_endpoint>`) nor in PubSub+ Connector Manager.



The simple metrics registry is only to be used for testing. It is not a production-ready means of collecting metrics. In production, use a dedicated monitoring system (for example, Datadog, Prometheus, etc.) to collect metrics.

The Actuator endpoint now contains information about Connector's internal states shared over the following HTTP/HTTPS endpoint:

```
GET: /actuator/
```

The following shows an example of the data shared with the configuration above:

```
{
  "_links": {
    "self": {
      "href": "/actuator",
      "templated": false
    },
    "workflows": {
      "href": "/actuator/workflows",
      "templated": false
    },
    "workflows-workflowId": {
      "href": "/actuator/workflows/{workflowId}",
      "templated": true
    },
    "leaderelection": {
      "href": "/actuator/leaderelection",
      "templated": false
    },
    "health-path": {
      "href": "/actuator/health/{*path}",
      "templated": true
    },
    "health": {
      "href": "/actuator/health",
      "templated": false
    },
    "metrics": {
      "href": "/actuator/metrics",
      "templated": false
    },
    "metrics-requiredMetricName": {
      "href": "/actuator/metrics/{requiredMetricName}",
      "templated": true
    }
  }
}
```

Health

The connector reports its health status using the [Spring Boot Actuator health endpoint](#).

To configure the information returned by the `health` endpoint, configure the following properties:

- `management.endpoint.health.show-details`
- `management.endpoint.health.show-components`

For more information, about health endpoints, see [Spring Boot documentation](#).

Health for the workflow, Solace binder, and camel binder components are exposed when `management.endpoint.health.show-components` is enabled. For example:

```
management:
  endpoint:
    health:
      show-components: always
      show-details: always
```

This configuration would always show the full details of the health check including the workflows and binders. The default value is `never`.

Workflow Health

A `workflows` health indicator is provided to show the health status for each of a connector's workflows. This health indicator has the following form:

```
{
  "status": "(UP|DOWN)",
  "components": {
    "<workflow-id>": {
      "status": "(UP|DOWN)",
      "details": {
        "error": "<error message>"
      }
    }
  }
}
```

Health Status	Description
UP	A status that indicates the workflow is functioning as expected.
DOWN	A status that indicates the workflow is unhealthy. Operator intervention may be required.

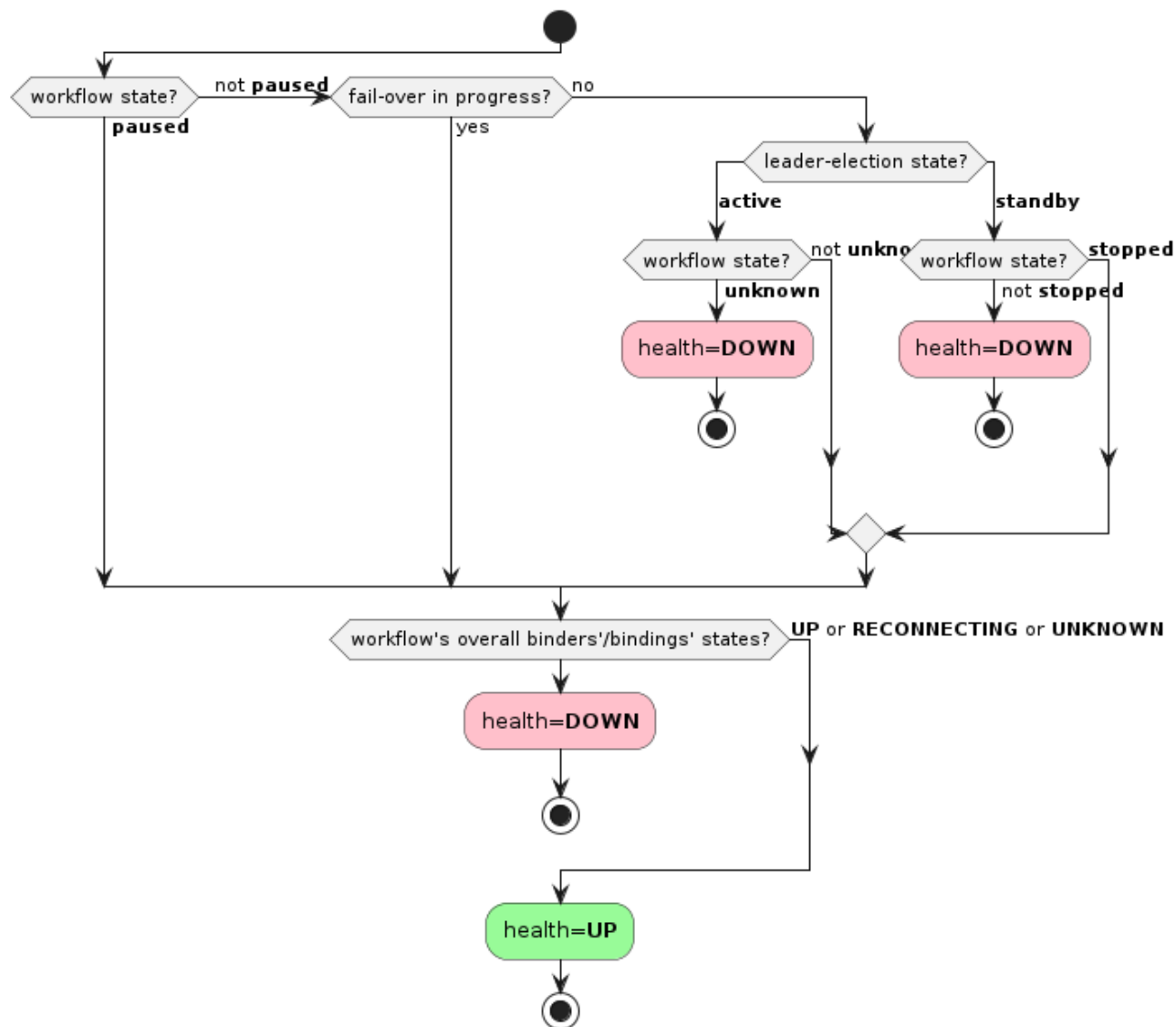


Figure 1. Workflow Health Resolution Diagram

This health indicator is enabled default. To disable it, set the property as follows:

```
management.health.workflows.enabled=false
```

Solace Binder Health

For details, see the [Solace binder](#) documentation.

Leader Election

The connector has three leader election modes for redundancy:

Leader Election Mode	Description
Standalone (Default)	A single instance of a connector without any leader election capabilities.
Active-Active	A participant in a cluster of connector instances where all instances are active.
Active-Standby	A participant in a cluster of connector instances where only one instance is active (i.e. the leader), and the others are standby.

Operators can configure the leader election mode by setting the following configuration:

```
solace.connector.management.leader-election.mode
=(standalone|active_active|active_standby)
```

Leader Election Modes: Standalone / Active-Active

When the connector starts, all enabled workflows start at the same time. The connector itself is considered as always active.

Leader Election Mode: Active-Standby

If the connector is in active-standby mode, a PubSub+ management session and management queue must be configured as follows:

```
solace.connector.leader-election.mode=active_standby

# Management session
# Exact same interface as solace.java.*
solace.connector.management.session.host=<management-host>
solace.connector.management.session.msgVpn=<management-vpn>
solace.connector.management.session.client-username=<client-username>
solace.connector.management.session.client-password=<client-password>
solace.connector.management.session.<other-property-name>=<value>

# Management queue name accessible by the management session
# Must have exclusive access type
solace.connector.management.queue=<management-queue-name>
```

To determine if the connector is **active** or **standby**, it creates a flow to the management queue. If this flow is active, then the connector's state is **active** and will start its enabled workflows. Otherwise, if this flow is inactive, then the connector's state is **standby** and will stop its enabled workflows.

At a macro level for a cluster of connectors, failover only happens when there are infrastructure failures (for example, the JVM goes down or networking failures to the management queue).

If a workflow fails to start or stop during failover, it will retry up to some maximum defined by the configuration option, `solace.connector.management.leader-election.fail-over.max-attempts`.

During failover, the connector attempts to start or stop all enabled workflows. After an attempt has been made to start or stop each workflow, the connector transitions to the active/standby mode regardless of the status of the workflows.

Leader Election Management Endpoint

A custom `leaderelection` management endpoint was provided using [Spring Actuator](#).

Operators can navigate to the connector's `leaderelection` management endpoint to view its leader election status.

Endpoint	Operation	Payloads
<code>/leaderelection</code>	Read (HTTP <code>GET</code>)	Request: None. Response: <pre> { "mode": { "type": "(standalone active_active ① active_standby)", "state": "(active standby)", ② "source": { ③ "queue": "<management-queue-name>", "host": "<management-host>", "msgVpn": "<management-vpn>" } } } </pre> ① Mandatory parameter in output ② Mandatory parameter in output ③ Optional section. Appears only when <code>type</code> is set to <code>active_standby</code>.

Workflow Management

Workflow Management Endpoint

A custom `workflows` management endpoint using [Spring Actuator](#) is provided to manage workflows.

To enable the `workflows` management endpoint:

```
management:
  endpoints:
    web:
      exposure:
        include: "workflows"
```

Once the `workflows` management endpoint is enabled, the following operations can be performed:

Endpoint	Operation	Payloads
<code>/workflows</code>	Read (HTTP <code>GET</code>)	Request: None. Response: Same payload as the <code>/workflows/{workflowId}</code> read operation, but as a list of all workflows.
<code>/workflows/{workflowId}</code>	Read (HTTP <code>GET</code>)	Request: None. Response: <pre>{ "id": "<workflowId>", "enabled": (true false), "state": "(running stopped paused unknown)", "inputBindings": ["<input-binding>"], "outputBindings": ["<output-binding>"] }</pre>
<code>/workflows/{workflowId}</code>	Write (HTTP <code>POST</code>)	Request: <pre>{ "state": "STARTED STOPPED PAUSED RESUMED" }</pre> Response: None.



Only workflows with Solace PubSub+ consumers (where the **solace** binder is defined in the **input-#**) support pause/resume.



Some features require for the connector to manage workflow lifecycles. There's no guarantee that workflow states continue to persist when write operations are used to change the workflow states while such features are in use.

For example: When the connector is configured in the active-standby leader election mode, workflows will automatically transition from **running** to **stopped** when the connector fails over from **active** to **standby**. Vice-versa for a failover in the opposite direction.

Workflow States

A workflow's state is defined as the aggregate states of its bindings (see the [bindings management endpoint](#)) as follows:

Workflow State	Condition
running	All bindings have state="running" .
stopped	All bindings have state="stopped" .
paused	All consumer bindings and all pausable producer bindings have state="paused" .
unknown	None of the other states. Represents an inconsistent aggregate binding state.



When the producer or consumer binding is not implementing Spring's Lifecycle interface, Spring always reports the bindings as **state=N/A**. The **state=N/A** is ignored when deciding the overall state of the workflow. For example, if the consumer's binding is **state=running** and producer's binding **state=N/A** (or vice-versa), the workflow state would be **running**.

For more information about binding states, see [Spring Cloud Stream: Binding visualization and control](#).

Metrics

This connector uses [Spring Boot Metrics](#) that leverages Micrometer to manage its metrics.

Connector Meters

In addition to the meters already provided by the Spring framework, this connector introduces the following custom meters:

Name	Type	Tags	Description	Notes
<code>solace.connector.process</code>	Timer	type: channel name: <bindingName> result: (success failure) exception: (none exception simple class name)	The processing time.	This meter is a rename of <code>spring.integration.send</code> whose <code>name</code> tag matches a binding name.
<code>solace.connector.error.process</code>	Timer	type: channel name: <bindingNames> result: (success failure) exception: (none exception simple class name)	The error processing time.	This meter is a rename of <code>spring.integration.send</code> whose <code>name</code> tag matches an input binding's error channel name (<destination>.<group>.errors). Meters might be merged under the same <code>name</code> tag (delimited by <code> </code>) if multiple bindings have the same error channel name (for example, bindings can have a matching <code>destination</code> , <code>group</code> , or both). NOTE: Setting a binding's <code>group</code> is not supported.
<code>solace.connector.message.size.payload</code>	DistributionSummary Base Units: bytes	name: <bindingName>	The message payload size.	

Name	Type	Tags	Description	Notes
<code>solace.connector.message.size.total</code>	DistributionSummary Base Units: bytes	name: <bindingName>	The total message size.	
<code>solace.connector.publish.ack</code>	Counter Base Units: acknowledgments	name: <bindingName> result: (success failure) exception: (none exception simple class name)	The publish acknowledgment count.	
<code>solace.connector.transform.expressions.count</code>	Gauge Base Units: expressions	workflow.id: <workflowId>	Transformation expressions count	
<code>solace.connector.transform.time</code>	Timer	workflow.id: <workflowId> result: (success failure)	Transformation execution time	This is the aggregate time for all expressions used to transform a single message.



The `solace.connector.process` meter with `result=failure` is not a reliable measure of tracking the number of failed messages. It only tells you how many times a step processed a message (or batch of messages), how long it took to process that message, and if that step completed successfully.

Instead, we recommend that you use a combination of `solace.connector.error.process` and `solace.connector.publish.ack` to track failed messages.

Add a Monitoring System

By default, this connector includes the following monitoring systems:

- [Datadog](#)
- [Dynatrace](#)
- [Influx](#)
- [JMX](#)
- [OpenTelemetry \(OTLP\)](#)

- [StatsD](#)

To add additional monitoring systems, add the system's `micrometer-registry-<system>` JAR file and its dependency JAR files to [the connector's classpath](#). The included systems can then be individually enabled/disabled by setting `management.<system>.metrics.export.enabled=true` in the `application.yml`.

Security

Securing Endpoints

Exposed Management Web Endpoints

There are many endpoints that are automatically enabled for this connector. For a comprehensive list, see [Management and Monitoring Connector](#).

The **health** endpoint only returns the root status by default (i.e. no health details).

To enable other management endpoints, see [Spring Actuator Endpoints](#).

Authentication & Authorization

This release of the connector only supports basic HTTP authentication.

By default, no users are created unless the operator configures them in their configuration file. The configuration parameters responsible for security are as follows:

```
solace:
  connector:
    security:
      enabled: true
      users:
        - name: user1
          password: pass
        - name: admin1
          password: admin
      roles:
        - admin
```

In the above example, we have created two users:

- **user1**: Has access to perform GET (Read) requests.
- **admin1**: Has access to perform GET and POST (Read & Write) requests.

To fully disable security and permit anyone to access the connector's web endpoints, operators can configure the `solace.connector.security.enabled` parameter **false**.



While these properties could be defined in an `application.yml` file, we recommend that you use environment variables to set secret values.

The following example shows you how to define users using environment variables:

```
# Create user with no role (i.e. read-only)
SOLACE_CONNECTOR_SECURITY_USERS_0_NAME=user1
```

```
SOLACE_CONNECTOR_SECURITY_USERS_0_PASSWORD=pass

# Create user with admin role
SOLACE_CONNECTOR_SECURITY_USERS_1_NAME=admin1
SOLACE_CONNECTOR_SECURITY_USERS_1_PASSWORD=admin
SOLACE_CONNECTOR_SECURITY_USERS_1_ROLES_0=admin
```

In the above example, we have created two users:

- **user1**: Has access to perform GET (Read) requests.
- **admin1**: Has access to perform GET and POST (Read & Write) requests.



`solace.connector.security.users` is a list. When users are defined in multiple sources (different `application.yml` files, environment variables, and so on), overriding works by replacing the entire list. In other words, you must pick one place to define all your users, whether in a **single** application properties file or as environment variables.

For more information, see [Spring Boot - Merging Complex Types](#).

TLS

TLS is disabled by default.

To configure TLS, see [Spring Boot - Configure SSL](#) and [TLS Setup in Spring](#).

Consuming Object Messages

For the connector to process object messages, it needs access to the classes which define the object payloads.

Assuming that your payload classes are in their own project(s) and are packaged into their own jar(s), place these jar(s) and their dependencies (if any) onto [the connector's classpath](#).



It is recommended that these jars only contain the relevant payload classes to prevent any oddities.

In the jar(s), your class files must be archived in the same directory/classpath as the application that publishes them.



e.g. If the source application is publishing a message with payload type, `MySerializablePayload`, defined under classpath `com.sample.payload`, then when packaging the payload jar for the connector, the `MySerializablePayload` class must still be accessible under the `com.sample.payload` classpath.

Typically, build tools such as Maven or Gradle will handle this when packaging jars.

Adding External Libraries

The connector jar uses the `loader.path` property as the recommended mechanism for adding external libraries to the connector's classpath.

See [Spring Boot - PropertiesLauncher Features](#) for more info.

To add libraries to the connector's container image, see [the connector's container documentation](#).

Configuration

Providing Configuration

For information about about how the connector detects configuration properties, see [Spring Boot: Externalized Configuration](#).

Converting Canonical Spring Property Names to Environment Variables

For information about converting the Spring property names to environment variables, see the [Spring documentation](#).

Spring Profiles

If multiple configuration files exist within the same configuration directory for use in different environments (development, production, etc.), use Spring profiles.

Using Spring profiles allow you to define different application property files under the same directory using the filename format, `application-{profile}.yml`.

For example:

- `application.yml`: The properties in non-specific files that always apply. Its properties are overridden by the properties defined in profile-specific files.
- `application-dev.yml`: Defines properties specific to the development environment.
- `application-prod.yml`: Defines properties specific to the production environment.

Individual profiles can then be enabled by setting the `spring.profiles.active` property.

See [Spring Boot: Profile-Specific Files](#) for more information and an example.

Configure Locations to Find Spring Property Files

By default, the connector detects any Spring property files as described in the [Spring Boot's default locations](#).

- If you want to add additional locations, add `--spring.config.additional-location=file:<custom-config-dir>` (This parameter is similar to the example command in [Quick Start: Running the connector via command line](#)).
- If you want to exclusively use the locations that you've defined and ignore Spring Boot's default locations, add `--spring.config.location=optional:classpath:/,optional:classpath:/config/,file:<custom-config-dir>`.

For more information about configuring locations to find Spring property files, see [Spring Boot documentation](#).



If you want configuration files for multiple, different connectors within the same `config` directory for use in different environments (such as development, production, etc.), we recommend that you use [Spring Boot Profiles](#) instead of child directories. For example:

- Set up your configuration like this:
 - `config/application-prod.yml`
 - `config/application-dev.yml`
- Do not do this:
 - `config/prod/application.yml`
 - `config/dev/application.yml`

Child directories are intended to be used for merging configuration from multiple sources of configuration properties. For more information and an example of when you might want to use multiple child directories to compose your application's configuration, see the [Spring Boot documentation](#).

Obtaining Build Information

Build information, including version, build date, time and description is enabled by default via [Spring Boot Actuator Info Endpoint](#). By default, every connector shares all information related to its `build` only.

Below is the structure of the output data:

```
{
  "build": {
    "version": "<connector version>",
    "artifact": "<connector artifact>",
    "name": "<connector name>",
    "time": "<connector build time>",
    "group": "<connector group>",
    "description": "<connector description>",
    "support": "<support information>"
  }
}
```

If you want to exclude build data from the output of the `info` endpoint, set `management.info.build.enabled` to `false`.

Alternatively, if you want to disable the info endpoint entirely, you can remove 'info' from the list of endpoints specified in `management.endpoints.web.exposure.include`.

Spring Configuration Options

This connector packages many libraries for you to customize functionality. Here are some

references to get started:

- [Spring Cloud Stream](#)
- [Spring Cloud Stream Binder for Solace PubSub+](#)
- [Spring Logging](#)
- [Spring Actuator Endpoints](#)
- [Spring Metrics](#)

Connector Configuration Options

The following table lists the configuration options. The following options in **Config Option** are prefixed with `solace.connector.:`

Config Option	Type	Default Value	Description
<code>management.leader-election.fail-over.max-attempts</code>	int Constraint: > 0	3	The maximum number of attempts to perform a fail-over.
<code>management.leader-election.fail-over.back-off-initial-interval</code>	long Constraint: > 0	1000	The initial interval (milliseconds) to back-off when retrying a fail-over.
<code>management.leader-election.fail-over.back-off-max-interval</code>	long Constraint: > 0	10000	The maximum interval (milliseconds) to back-off when retrying a fail-over.
<code>management.leader-election.fail-over.back-off-multiplier</code>	double Constraint: >= 1.0	2.0	The multiplier to apply to the back-off interval between each retry of a fail-over.
<code>management.leader-election.mode</code>	enum One of: • <code>standalone</code> • <code>active_active</code> • <code>active_standby</code>	<code>standalone</code>	The connector's leader election mode. standalone: A single instance of a connector without any leader election capabilities. active_active: A participant in a cluster of connector instances where all instances are active. active_standby: A participant in a cluster of connector instances where only one instance is active (i.e. the leader), and the others are standby.
<code>management.queue</code>	string	null	The management queue name.

Config Option	Type	Default Value	Description
<code>management.session.*</code>	See Spring Boot Auto-Configuration for the Solace Java API		Defines the management session. This has the same interface as that used by <code>solace.java.*</code> . See Spring Boot Auto-Configuration for the Solace Java API for more info.
<code>security.enabled</code>	boolean	true	If <code>true</code> , security is enabled. Otherwise, anyone has access to the connector's endpoints.
<code>security.users[<index>].name</code>	string	null	The name of the user.
<code>security.users[<index>].password</code>	string	null	The password for the user.
<code>security.users[<index>].roles</code>	list<string> Valid values: • <code>admin</code>	empty list (i.e. read-only)	The list of roles that the specified user has. It has read-only access if no roles are returned.

Workflow Configuration Options

These configuration options are defined under the following prefixes:

- `solace.connector.workflows.<workflow-id>.`: If the options support per-workflow configuration and the default prefixes.
- `solace.connector.default.workflow.`: If the options support default workflow configuration.

Config Option	Type	Default Value	Description
<code>enabled</code>	boolean	false	If <code>true</code> , the workflow is enabled.  Cannot be set at the default workflow level.
<code>transform.enabled</code>	boolean	false	If <code>true</code> , message transformation is enabled for the workflow. Disables the legacy <code>transform-headers.*</code> and <code>transform-payload.*</code> options. See Message Transforms for more info.

Config Option	Type	Default Value	Description
<code>transform.source-payload.content-type</code>	string One of: <code>application/vnd.solace.micro-integration.unspecified</code> <code>application/json</code>	<code>application/vnd.solace.micro-integration.unspecified</code>	The content type to interpret the source payload as. See Content Type Interpretation for more info.
<code>transform.target-payload.content-type</code>	string One of: <code>application/vnd.solace.micro-integration.unspecified</code> <code>application/json</code>	<code>application/vnd.solace.micro-integration.unspecified</code>	The content type to interpret and serialize the target payload as. See Content Type Interpretation for more info.
<code>transform.expressions[<index>].transform</code>	string A SpEL expression		A SpEL expression at some <code><index></code> in the ordered list of expressions to transform the message. See Message Transform for more info.
<code>transform-headers.expressions</code>	Map<string, string> Key: A header name. Value: A SpEL string that accepts <code>headers</code> as parameters.	empty map	 Deprecated. Use Message Transforms (<code>transform.*</code>) instead. A mapping of header names to header value SpEL expressions. The SpEL context contains the <code>headers</code> parameter that can be used to read the input message's headers.
<code>acknowledgment.publish-async</code>	boolean	<code>true</code>	If <code>true</code> , publisher acknowledgment processing is done asynchronously. The workflow's consumer and producer bindings must support this mode, otherwise the publisher acknowledgments are processed synchronously regardless of this setting.

Config Option	Type	Default Value	Description
<code>acknowledgment.back-pressure-threshold</code>	int Constraint: ≥ 1	255	The maximum number of outstanding messages with unresolved acknowledgments. Message consumption is paused when the threshold is reached to allow for producer acknowledgments to catch up.
<code>acknowledgment.publish-timeout</code>	int Constraint: ≥ -1	600000	The maximum amount of time (in millisecond) to wait for asynchronous publisher acknowledgments before considering a message as failed. A value of -1 means to wait indefinitely for publisher acknowledgments.

```
include::/home/runner/work/pubsubplus-connectors-file-storage/pubsubplus-connectors-file-storage/pubsubplus-connector-aws-s3/src/docs/asciidoc/./snippets/source-config.adoc
```

```
include::/home/runner/work/pubsubplus-connectors-file-storage/pubsubplus-connectors-file-storage/pubsubplus-connector-aws-s3/src/docs/asciidoc/./snippets/checkpoint-store-config.adoc
```

Logging

Configuring Logback

If you require Logback configuration beyond what is already available through Spring config properties, then you can decide to run your connector with a `logback-spring.xml` file. For information about using the logback, see:

- [logging.config](#) spring property.
- [Logging](#)
- [Configure Logback for Logging](#) sections in the Spring documentation.

The main difference compared to [what Spring Boot provides](#), is that this connector provides its own alternative logback configuration files that can be [included](#) into your `logback-spring.xml` file.

These new files can be included and are found at `com/solace/connector/core/logging/logback/`:

`defaults.xml`

Provides conversion rules, pattern properties and common logger configurations.



Always include `defaults.xml`

We recommend that you always include this file in your `logback-spring.xml` file as it includes a `%sanitize` `<conversionRule>` that's applied to the default `CONSOLE_LOG_PATTERN` and `FILE_LOG_PATTERN`.

This conversion rule does some filtering to obfuscate potentially sensitive data from the logs.

`console-appender.xml`

Adds a `ConsoleAppender` using the `CONSOLE_LOG_PATTERN`.

`file-appender.xml`

Adds a `RollingFileAppender` using the `FILE_LOG_PATTERN` and `ROLLING_FILE_NAME_PATTERN` with appropriate settings.

Aside from these new include-able files, this connector still supports all the same logging options that regular Spring Boot does.

For more information and examples, see the [Configure Logback for Logging](#) section in the Spring documentation.

License

This project is licensed under the Solace Community License, Version 1.0. - See the [LICENSE](#) file for details.

Support

Support is offered best effort via our [Solace Developer Community](#).

Premium support options are available, please [Contact Solace](#).