pubsubplus-connector-aws-kinesis

User Guide

Solace Corporation

Version 1.0.1

solace.

Table of Contents

Preface

Solace - AWS Kinesis Connector

Getting Started

Assuming you're using the default `application.yml` within this package, following one of the below quick start guides will result in a connector that will connect to the PubSub+ broker and kinesis using default credentials, with 2 workflows enabled, workflow 0 and workflow 1. Where:

- Workflow 0 is consuming messages from the Solace PubSub+ queue, `Solace/Queue/0`, and publishing them to the kinesis producer destination, `producer-destination`.
- Workflow 1 is consuming messages from the kinesis consumer destination, `consumer-destination`, and publishing 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 will not provision queues which do not exist.

Prerequisites

- Solace PubSub+ Event Broker
- kinesis

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-kinesis-1.0.1.jar --spring.config.additional-location=file:samples/config/
```

By default, this command will detect any Spring Boot config files as per Spring Boot's default locations.

For more info, see Configure Locations to Find Spring Property Files.

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

For convenience purposes, connector may be also started through the shell script using following syntax:

```
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]
```

In case of invalid parameters:

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

the script shows you all errors at once:

```
pubsubplus-connector-aws-kinesis

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 case of missing parameters, script will run with predefined values, which are the following:

Parameter	Default Value	Description
`-n, --name`	`application`	Name of the Connector instance, set up in [spring.application.name]. This name impacts on grouping connectors only
`-l, --libs`	`./libs`	Directory containing required and optional dependency jars. Such as Micrometer metrics export dependencies (if configured). If not specified, it will use the current `./libs/` folder.
`-p, --profile`	`empty, no profile is used`	Profile to be used with connector's configuration. The configuration file named 'application-<profile>.yml' will be used. Default value is set to use no profile.
`-c, --config`	`./ or current folder`	Path to the folder containing the configuration files to be applied during connector startup for chosen profile.

Parameter	Default Value	Description
`-H, --host`	`127.0.0.1`	Provides host where Connector will be running on.
`-P, --port`	`8090`	Provides port where Connector will be running on.
`-mp, --mgmt_port`	`9009`	Provides management port for back calls of current Connector from Connector Manager. Ignored if `-cm` argument is missing
`-j, --jar`	`pubsubplus-connector-aws-kinesis-1.0.1.jar`	Path to specified JAR file to start connector. If not specified, the default jar file is used from the current folder.
`-cm, --manager`	`application`	Allow to enable cloud configuration for Connector by using Connector Manager configuration storage. Enabling this option also requires to set `-mp` or `--mgmt_port`, `-H` or `--host` and `-cmh` with `-cmp` unless you want to use defaults. Be aware, this option disable listed parameters to be read from configuration file. In this case operator must specify them explicitly as parameters for this script or use default values.
`-cmh, --cm_host`	`127.0.0.1`	Specifies the host where Connector Manager is running. Ignored if `-cm` argument is missing.
`-cmp, --cm_port`	`9500`	Specifies the port where Connector Manager is running. Ignored if `-cm` argument is missing.
`-o, --options`	`no default values`	Specifies JVM options used on Connector start.
`-tls`	`N/A`	Use HTTPS instead of HTTP. Configuration file in this case should contain additional section with preconfigured paths for key store and trust store files.

Parameter	Default Value	Description
`-s, --show`	`N/A`	This option prints the start CLI command in raw and exit, doing effectively nothing, just a dry run.
`-b, --background`	`N/A`	Runs Connector in the background. No logs will be displayed and Connector continues running in detached mode
`-h, --help`	`N/A`	Prints some 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.

Quick Start: Running the connector as a Container

A sample docker compose file has been packaged for your convenience:

1. Change to the `docker` directory:

   ```
   cd samples/docker
   ```

   This directory contains both the `docker-compose.yml` file as well as a `.env` file that contains environment secrets required for the container's health-check.

2. Run the connector:

   ```
   docker-compose up -d
   ```

   This sample docker compose file will:

   - Expose the connector's `8090` web port to `8090` on the host.
   - Connect to PubSub+ and kinesis exposed on the host using default ports.
   - Mount the `samples/config` directory.
   - Mount the previously defined `libs` directory.
   - Create a `healthcheck` user with read-only permissions.
     - The default username and password for this user can be found within the `.env` file.
     - This will override any users you have defined in your `application.yml`. See here for more info.

- Uses the connector's management health endpoint as the container's healthcheck.

For more info about how to use and configure this container, see the connector's container documentation.

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|kinesis) # Input system
        output-<workflow-id>:
          destination: <output-destination> # Topic name
          binder: (solace|kinesis) # Output system

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

The connector only supports workflows in the directions of:

- `solace` → `kinesis`
- `kinesis` → `solace`

For more info about Spring Cloud Stream and the Solace PubSub+ binder:

- 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 Solace PubSub+ uses Spring Boot Auto-Configuration for the Solace Java API to configure its session.

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

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

For more info 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 (e.g. 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 instance:

```
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 kinesis binder
        kinesis1:
          type: kinesis
          # 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: kinesis1
        output-0:
          destination: <output-destination>
          binder: solace1 # Reference 1st solace binder
        input-1:
          destination: <input-destination>
          binder: kinesis1
        output-1:
          destination: <output-destination>
          binder: solace2 # Reference 2nd solace binder
```

Defines two binders of type `solace` and one binder of type `kinesis` which are then referenced within bindings.

Note that each binder 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.

Do not use single-binder configuration (e.g. `solace.java.*` at the root of your `application.yml`) while using the multiple binder syntax.

AWS Kinesis Connection Details

The Spring Cloud Stream Binder for AWS Kinesis uses the following configuration to configure its session

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

```
cloud:
  aws:
    credentials:
      accessKey: AWS_ACCESS_KEY
      secretKey: AWS_SECRET_KEY
    region:
      static: AWS_REGION
    stack:
      auto: false
```

In case of Docker deployment the json file can be placed in config folder and the config path relevant to docker container is used to point the file location.

User-configured Header Transforms

Generally, the consumed message's headers are propagated through the connector to the output message. If you want to transform the headers, then you may do so as follows:

```
# <workflow-id> : The workflow ID ([0-19])
# <header> : The key for the outbound header
# <expression> : A SpEL expression which has "headers" as parameters

solace.connector.workflows.<workflow-id>.transform-
headers.expressions.<header>=<expression>
```

Example 1: To create a new header, `new_header`, for workflow `0` that is derived from the headers `foo` & `bar`:

```
solace.connector.workflows.0.transform-headers.expressions.new_header
="T(String).format('%s/abc/%s', headers.foo, headers.bar)"
```

Example 2: To remove the header, `delete_me`, for workflow `0`, set the header transform expression to `null`:

```
solace.connector.workflows.0.transform-headers.expressions.delete_me="null"
```

For more info about Spring Expression Language (SpEL) expressions:

- Spring Expression Language (SpEL)

User-configured Payload Transforms

Message payloads going through a workflow can be transformed using a Spring Expression Language (SpEL) expression as follows:

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

solace.connector.workflows.<workflow-id>.transform-payloads.expressions[0].transform
=<expression>
```

A SpEL expression may reference:

- `payload`: to access the message payload
- `headers.<header_name>`: to access a message header value
- Registered functions

While the syntax uses an array of expressions, only a single transform expression is supported in this release. Multiple transform expressions may be supported in future versions.

Registered Functions

Registered functions are built-in and can be called directly from SpEL expressions. To call a registered function, use the `#` character followed by the function name. The following table describes available registered functions:

Registered Function Signature	Description
`boolean isPayloadBytes(Object obj)`	Returns whether the object `obj` is an instanceof `byte[]` or not. Sample usage of this function within a SpEL expression: `"#isPayloadBytes(payload) ? true : false"`

Example 1: To normalize byte[] and String payloads as upper-cased String payloads or leave payloads unchanged when of different types.

```
solace.connector.workflows.0.transform-payloads.expressions[0].transform
="#isPayloadBytes(payload) ? new String(payload).toUpperCase() : payload instanceof
T(String) ? payload.toUpperCase() : payload"
```

Example 2: To convert String payloads to byte[] payloads using a charset retrieved from a message header or leave payloads unchanged when of different types.

```
solace.connector.workflows.0.transform-payloads.expressions[0].transform="payload
instanceof T(String) ?
payload.getBytes(T(java.nio.charset.Charset).forName(headers.charset)) : payload"
```

For more info about Spring Expression Language (SpEL) expressions:

- Spring Expression Language (SpEL)

Message Headers

Solace and kinesis headers can be created or manipulated using the User-configured Header Transforms feature described above.

Solace Headers

Solace headers exposed to the connector are documented within 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 a future version of the connector.

Dynamic Producer Destinations

To route messages to dynamic destinations at runtime, use the User-configured Header Transforms feature described above to set the following headers:

Header Name	Type	Values	Applies To	Description
`scst_targetDestination`	`string`	Any valid destination name	Solace, kinesis	Specifies the name of the dynamic destination to publish to. Setting this header overrides the configured destination.
`solace_scst_targetDestinationType`	`string`	`(queue|topic)`	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.transform-headers` 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` will enable asynchronous publishing on the connector's core but the effective publishing mode is still synchronous due to a lack of support 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.

Actuator shares information through the endpoints reachable over HTTP(s). 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 example configuration enables metrics collection through the configuration parameter of `management.simple.metrics.export.enabled` set to `true` and then shares them through the HTTP(s) endpoint together with other sections configured for the current Connector.

The set of endpoints exposed through the HTTP(s) endpoint. Exposed endpoints are available in the connector UI and are also available to the PubSub+ Connector Manager. The operator may choose to not expose all or some of these endpoints. Endpoints not exposed will not be available in the connector web UI nor the 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 (e.g. Datadog, Prometheus, etc) to collect metrics.

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

```
GET: /actuator/
```

Here is the 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 via the Spring Boot Actuator health endpoint.

To configure the information returned by the `health` endpoint, configure the following properties: `management.endpoint.health.show-details` and `management.endpoint.health.show-components`. Refer to Spring Boot documentation for details.

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

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

This would always show the full detail 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`	Status indicating that the workflow is functioning as expected.
`DOWN`	Status indicating that the workflow is unhealthy. User intervention may be required.

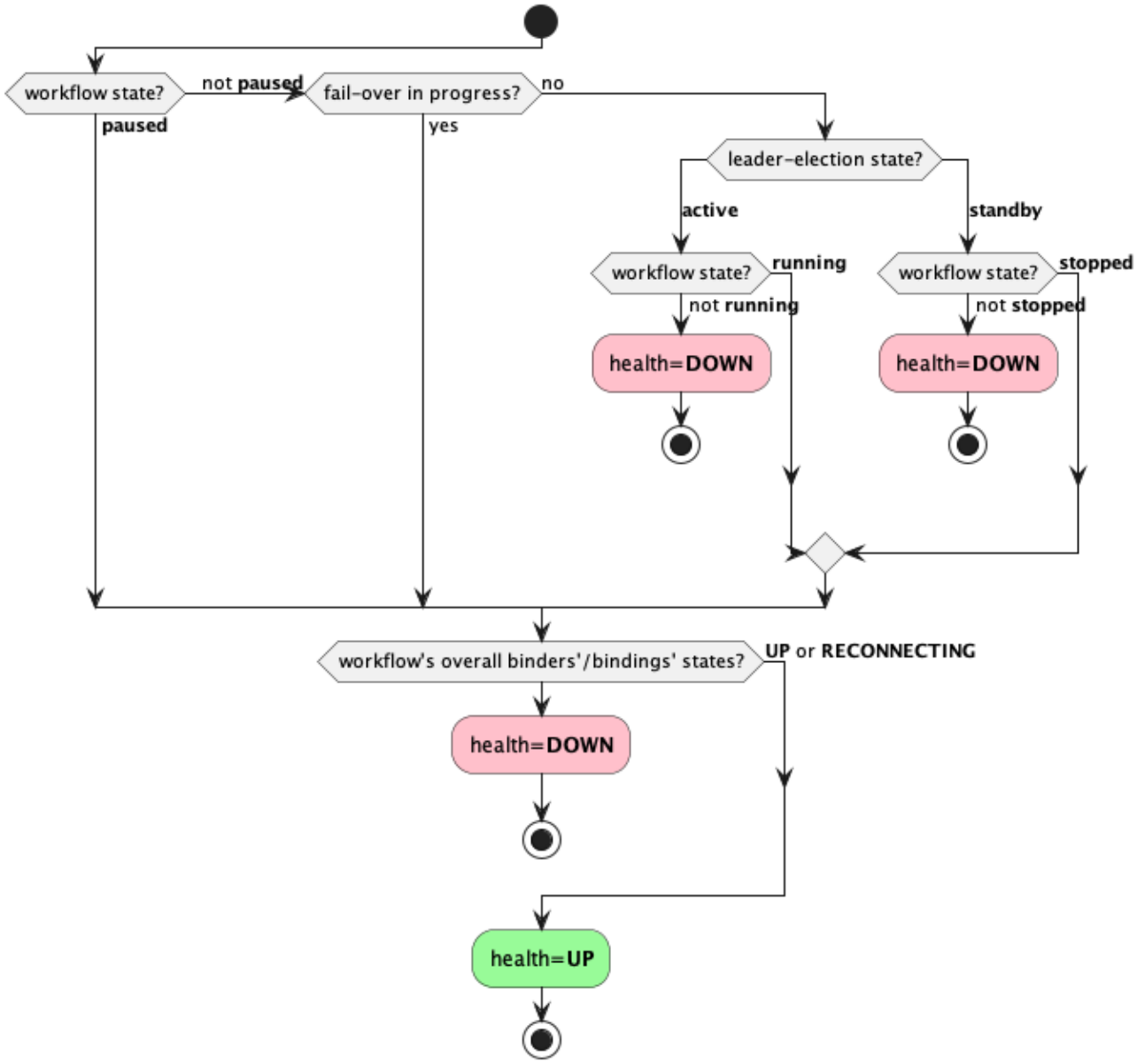

Figure 1. Workflow Health Resolution Diagram

This health indicator is enabled default. To disable it, add this to your configuration:

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

Solace Binder Health

For details, see the Solace binder documentation.

Leader Election

The connector has 3 leader election modes:

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:

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

Leader Election Modes: Standalone / Active-Active

All enabled workflows are started during connector startup and the connector is considered as always active.

Leader Election Mode: Active-Standby

If the connector is in active-standby mode, then 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 will create 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, fail-over only happens when there are infrastructure failures (e.g. JVM goes down or networking failures to the management queue).

If a workflow fails to be started/stopped during fail-over, it will retry up to some maximum defined by the config option, `solace.connector.management.leader-election.fail-over.max-attempts`.

During fail-over, the connector will attempt to start/stop all enabled workflows. Once an attempt has been made to start/stop each workflow, then the connector has transitioned to active/standby mode regardless of the status of the workflows.

Leader Election Management Endpoint

A custom `leaderelection` management endpoint was provided using Spring Actuator.

Navigate to the connector's `leaderelection` management endpoint to view its leader election status.

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

Workflow Management

Workflow Management Endpoint

A custom `workflows` management endpoint using Spring Actuator was 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
`/workflows`	Read (HTTP `GET`)	**Request:** None. **Response:** Same payload as the `/workflows/{workflowId}` read operation, but as a list of all workflows.
`/workflows/{workflowId}`	Read (HTTP `GET`)	**Request:** None. **Response:** `{ "id": "<workflowId>", "enabled": (true|false), "state": "(running|stopped|paused|unknown)", "inputBindings": [ "<input-binding>" ], "outputBindings": [ "<output-binding>" ] }`

Endpoint	Operation	Payloads
`/workflows/{workflowId}`	Write (HTTP `POST`)	**Request:** `{ "state": "STARTED|STOPPED|PAUSED|RESUMED" }` **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 will persist when write operations are used to change 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 fail-over 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 producer/consumer binding is not implementing Spring's Lifecycle interface, Spring always reports binding's `state=N/A`. The `state=N/A` will be ignored in deciding the overall state of the workflow. So for example, if the consumer binding `state=running` and producer binding `state=N/A` or vise a versa the workflow state will be `running`.

For more info about binding states, see Spring Cloud Stream: Binding visualization and control.

Metrics

This connector uses Spring Boot Metrics which 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

`solace.connector.process`	`Timer`	`type: channel`
		
		`name: <bindingName>`
		
		`result: (success|failure)`
		
		`exception: (none|exception simple class name)`
			Processing time	This meter is a rename of `spring.integration.send` whose `name` tag matches a binding name.

`solace.connector.error.process`	`Timer`	`type: channel`
		
		`name: <bindingNames>`
		
		`result: (success|failure)`
		
		`exception: (none|exception simple class name)`
			Error processing time	This meter is a rename of `spring.integration.send` whose `name` tag matches an input binding's error channel name (`<destination>.<group>.errors`).
				
				Meters might be merged under the same `name` tag (delimited by `|`) if multiple bindings have the same error channel name (i.e. bindings have matching `destination` and/or `group`). **Though a reminder that setting a binding's `group` is not supported.**

`solace.connector.message.size.payload`	`DistributionSummary` Base Units: `bytes`	`name: <bindingName>`	Message payload size	

Name	Type	Tags	Description	Notes
solace.connector.message.size.total	DistributionSummary Base Units: bytes	name: <bindingName>	Total message size	
solace.connector.publish.ack	Counter Base Units: acknowledgments	name: <bindingName> result: (success|failure) exception: (none|exception simple class name)	Publish acknowledgment count	

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, how long it took to process that message, and if that step completed successfully.

Instead, it's recommended to 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 and its dependency jars 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

Please refer to the Management and Monitoring Connector section for a comprehensive list of endpoints that are automatically enabled for this 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

For this release, the connector only supports basic HTTP authentication.

By default, no users will be created unless the operator configures it in their config. Configuration parameters responsible for security are:

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

In the above example, we have created 2 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 parameter `solace.connector.security.enabled` switched to `false`.

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

Here is an example of 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 2 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, etc), then overriding works by replacing the entire list. Meaning that you must pick one place to defined your users and define all of them there. Whether it be in a **single** application properties file or all of them in the environment variables.

See Spring Boot - Merging Complex Types for more info.

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

See Spring Boot: Externalized Configuration for info about how the connector will detect configuration properties.

Converting Canonical Spring Property Names to Environment Variables

See the Spring documentation for how to provide configuration options as environment variables.

Spring Profiles

If multiple config files will exist within the same config folder for use in different environments (dev, prod, etc), then use Spring profiles.

This will allow you to define different application property files under the same directory using the file name format, `application-{profile}.yml`.

eg:

- `application.yml`
  - Properties in non-specific files always applies. It's properties are overridden by those defined in profile-specific files.
- `application-dev.yml`
  - Defines properties specific to the `dev` environment.
- `application-prod.yml`
  - Defines properties specific to the `prod` environment.

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

See Spring Boot: Profile-Specific Files for more info as well as an example.

Configure Locations to Find Spring Property Files

By default, the connector will detect any Spring property files as per Spring Boot's default locations.

- If you want to add additional locations, add `--spring.config.additional-location=file:<custom-config-dir>` (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>`.

See Spring Boot documentation for more info.

If config files for multiple, different, connectors will exist within the same config folder for use in different environments (e.g. dev, prod, etc), then consider using Spring Boot Profiles instead of child directories to do this.

i.e.:

- Do this:
  - `config/application-prod.yml`
  - `config/application-dev.yml`
- Instead of this:
  - `config/prod/application.yml`
  - `config/dev/application.yml`

Child directories are intended to be used for merging configuration from multiple sources of config properties. For more information and an example of when you might want to use multiple child directories to compose your application's configuration, please 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 wish to exclude build data from the output of the info endpoint, you can do so by setting `management.info.build.enabled` to `false`.

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

Spring Configuration Options

This connector packages a lot of libraries 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

These configuration options are all prefixed by `solace.connector.`:

Config Option	Type	Valid Values	Default Value	Description
`management.leader-election.fail-over.max-attempts`	`int`	`> 0`	`3`	The maximum number of attempts to perform a fail-over.
`management.leader-election.fail-over.back-off-initial-interval`	`long`	`> 0`	`1000`	The initial interval (milliseconds) to back-off when retrying a fail-over.
`management.leader-election.fail-over.back-off-max-interval`	`long`	`> 0`	`10000`	The maximum interval (milliseconds) to back-off when retrying a fail-over.
`management.leader-election.fail-over.back-off-multiplier`	`double`	`>= 1.0`	`2.0`	The multiplier to apply to the back-off interval between each retry of a fail-over.

Config Option	Type	Valid Values	Default Value	Description
management.leader-election.mode	enum	(standalone|active_active|active_standby)	standalone	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.
management.queue	string	any	null	The management queue name.
management.session.*		See Spring Boot Auto-Configuration for the Solace Java API		Defines the management session. This has the same interface as that used by solace.java.*. See Spring Boot Auto-Configuration for the Solace Java API for more info.
security.enabled	boolean	(true|false)	true	If true, security is enabled. Otherwise, anyone has access to the connector's endpoints.
security.users[<index>].name	string	any	null	The name of this user.
security.users[<index>].password	string	any	null	The password of this user.
security.users[<index>].roles	list<string>	admin	empty list (i.e. read-only)	The list of roles which this user has. Has read-only access if no roles are given.

Workflow Configuration Options

These configuration options are defined under the prefix, `solace.connector.workflows.<workflow-id>.` (if they support per-workflow config), and the default prefix, `solace.connector.default.workflow.` (if they support default workflow config).

Config Option	Applicable Scopes	Type	Valid Values	Default Value	Description
`enabled`	Per-Workflow	`boolean`	`(true|false)`	`false`	If `true`, the workflow is enabled.
`transform-headers.expressions`	Per-Workflow Default	`Map<string,string>`	**Key:** A header name. **Value:** A SpEL string which accepts `headers` as parameters.	`empty map`	A mapping of header names to header value SpEL expressions. The SpEL context contains the `headers` parameter which can be used to read the input message's headers.
`acknowledgment.publish-async`	Per-Workflow Default	`boolean`	`(true|false)`	`false`	If `true`, publisher acknowledgment processing is done asynchronously. The workflow's consumer and producer bindings must support this mode, otherwise, publisher acknowledgments are processed synchronously regardless of this setting.

Config Option	Applicable Scopes	Type	Valid Values	Default Value	Description
`acknowledgment.back-pressure-threshold`	Per-Workflow Default	`int`	>= 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.
`acknowledgment.publish-timeout`	Per-Workflow Default	`int`	>= -1	`600000`	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.

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.