Solace PubSub+ Connector for TIBCO EMS

User Guide

Solace Corporation

Version 1.0.0

solace.

Table of Contents

Preface

Solace PubSub+ Connector for TIBCO EMS bridges data between the Solace PubSub+ Event Broker and TIBCO EMS providing a flexible and efficient way to integrate TIBCO EMS application data with your Solace-backed, event-driven architecture and the Event Mesh. The connector is deployable standalone or in redundancy modes of “active-standby” or “active-active” to allow for high-availability and horizontal scaling of your data movement. Each connector instance supports up to 20 individual workflows (source-to-target pipeline), minimizing the number of connector instances deployed and managed. 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

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 TIBCO EMS 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 TIBCO EMS producer destination, `producer-destination`.
- Workflow 1 is consuming messages from the TIBCO EMS 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
- TIBCO EMS

Quick Start common steps

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

1. Create a directory called `libs` in the same directory as the jar file.
   a. For more info about this directory, see Adding External Libraries.
   b. This directory may already exist
2. Download the jars for the TIBCO EMS and all its dependencies to the `libs` directory:
3. Update the provided `samples/config/application.yml` with the values for your deployment.

Quick Start: Running the connector via command line

Run:

```
java -Dloader.path=libs/ -jar pubsubplus-connector-tibcoems-1.0.0.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 [-l FOLDER] [-p PROFILE] [-c FOLDER] [-j FILE] [-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:

```
Solace PubSub+ Connector for TIBCO EMS

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
`-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`	`current folder`	Path to the folder containing the configuration files to be applied during connector startup for chosen profile. By default it is set to the current folder.

Parameter	Default Value	Description
-j, --jar	pubsubplus-connector-tibcoems-1.0.0.jar	Path to specified JAR file to start connector. If not specified, the default jar file is used from the current folder.
-o, --options	no default values	Specifies JVM options used on Connector start.
-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 some help information from command line:

```
Solace PubSub+ Connector for TIBCO EMS

Usage: start.sh [-l FOLDER] [-p PROFILE] [-c FOLDER] [-j FILE] [-o OPTIONS] [-b]

To start connector use following parameters:

-l | --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     Profile to be used with connector's configuration.
                   The configuration file named
                   'application-<profile>.yml' will be used
                   Default value is empty, no profile is used.

-c | --config      Path to the folder containing the
                   configuration files to be applied during
                   connector startup for chosen profile.
                   By default it is set to the current folder.

-j | --jar         Path to specified JAR file to start connector.
                   if not specified, the default jar file is used,
                   from the current folder

-o | --options     Specifies JVM options used on Connector start.
                   example: '-Xms64M -Xmx1G'

-b | --background  Runs the Connector in the background. No logs will be
                   displayed and Connector continues running in detached
```

```
                    mode.

-h | --help         Print this help page and exit
```

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

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

The connector only supports workflows in the directions of:

- `solace` → `TIBCO EMS`
- `TIBCO EMS` → `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.

TIBCO EMS Connection Details

Manual Configuration

To manually configure TIBCO EMS connection details, set the following in `application.yml`:

```
solace:
  tibco-ems:
    server-url: tcp://localhost:7222
    username: username
    password: password
    properties:
      <key>: <value>
```

Properties are key/value pairs where `key` starts with the `com.tibco.tibjms.*` prefix. These properties are passed without validation as a Map to the TIBCO EMS JMS Connection Factory. Accepted properties can be found in Tibjms and in TibjmsSSL.

JNDI

The JMS binder provides a generic way of configuring and using JNDI.

JNDI Context

The first step in using JNDI is to configure the JNDI context. The JMS binder expects standard JNDI properties to be specified under `jms-binder.jndi.context` in a key/value pair format. The key is the name of the property (e.g. "java.naming.provider.url") and the value is a string in the format defined for that property.

For instance, using the TIBCO EMS server as the JNDI provider could look like:

```
jms-binder:
  jndi:
    context:
      java.naming.factory.initial: com.tibco.tibjms.naming.TibjmsInitialContextFactory
      java.naming.provider.url: tibjmsnaming://localhost:7222
```

Note that classes required by the chosen JNDI service provider need to be added to the classpath.

Once a JNDI context is successfully configured, connection factories and/or destinations can be looked up.

Connection Factory Lookup

To lookup a connection factory, configure `jms-binder.jndi.connection-factory`.

```
jms-binder:
  jndi:
    connection-factory:
      name: jndiConnectionFactoryName
      user: someUser
      password: somePassword
```

where:

- **name**: the JNDI object name of the connection factory
- **user**: the user to authenticate with the TIBCO EMS server.
- **password**: the password to authenticate with the TIBCO EMS server.

JNDI connection factories should not specify a `clientID` as this prevents producer bindings from connecting.

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 jms binder
        jms1:
          type: jms
          # 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: jms1
        output-0:
          destination: <output-destination>
          binder: solace1 # Reference 1st solace binder
        input-1:
          destination: <input-destination>
          binder: jms1
        output-1:
          destination: <output-destination>
          binder: solace2 # Reference 2nd solace binder
```

Defines two binders of type `solace` and one binder of type `jms` 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.

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 jms 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.

JMS Headers

JMS headers exposed to the connector are the following:

Header Name	Type	Access	Description
`jms_correlationId`	`String`	Read/Write	The correlation ID for the message.
`jms_deliveryMode`	`int`	Read	The delivery mode value specified for this message.
`jms_destination`	`javax.jms.Destination`	Read	The destination to which the message is being sent.
`jms_expiration`	`long`	Read	The time at which the JMS message is set to expire.
`jms_messageId`	`String`	Read	A value that uniquely identifies each message sent by a provider.
`jms_priority`	`int`	Read/Write	Specifies the message's priority set on the send. When header is absent, JMS message is sent with default priority of 4.
`jms_redelivered`	`boolean`	Read	An indication of whether this message is being redelivered.
`jms_replyTo`	`javax.jms.Destination`	Read/Write	The Destination object to which a reply to this message should be sent.
`jms_timestamp`	`long`	Read	The time a message was handed off to a provider to be sent.
`jms_timeToLive`	`long`	Write	Specifies the message's time to live set on the send. When header is absent, JMS message is sent with default timeToLive of 0 (zero means that a message never expires).
`jms_type`	`String`	Read/Write	The message type identifier supplied by the client when the message was sent.

Reserved Message Headers

The following are reserved header spaces:

- `solace_`
- `scst_`
- jms_|JMS_|JMSX
- 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.

JMS Destination Types

JMS binding destinations can be configured as physical destination names or as JNDI destination names.

The `spring.cloud.stream.jms.bindings.<binding_name>.<consumer | producer>.destination-type` binding property specifies whether the `destination` value is a physical destination name or a JNDI destination name.

Destination Type	Destination JNDI Lookup?
`queue`	No
`topic`	No
`unknown` (default)	Yes

When `destination-type` is either `queue` or `topic`, the configured `destination` is assumed to be a physical destination name and no JNDI lookup is done.

When `destination-type` is `unknown`, the configured `destination` is assumed to be a JNDI destination name and a lookup is performed. A JNDI Context must be configured for the lookup to succeed.

For instance, in the following example the consumer's `destination` is known at configuration time and no JNDI lookup is done:

```
spring:
  cloud:
    stream:
      bindings:
        input-0:
          destination: physical_queue_name
          binder: jms
      jms:
        bindings:
          input-0:
            consumer:
              destination-type: queue
```

In the following example, the producer's `destination` is only known at runtime after a successful JNDI lookup:

```
spring:
  cloud:
    stream:
      bindings:
        output-1:
          destination: jndi_destination_name
          binder: jms
      jms:
```

```
        bindings:
          output-1:
            producer:
              destination-type: unknown
```

JMS Shared Durable Subscribers

A JMS consumer binding can bind to a shared durable subscription, enabling multiple consumers to share the load of messages published to the subscription. Durable subscriptions accumulate messages even when all consumers are offline, ensuring that no messages are lost.

To consume from a shared durable subscription, the following must be configured:

- `spring.cloud.stream.jms.bindings.<binding_name>.consumer.destination-type` should be set to a `topic`. `unknown` can also be used as long as the `destination` resolves to a topic.
- `spring.cloud.stream.jms.bindings.<binding_name>.consumer.durable-subscription-name` must be specified with the name of the subscription.
- `spring.cloud.stream.bindings.<binding_name>.destination` must be set to the name of the topic on which the subscription is created.

```
spring:
  cloud:
    stream:
      bindings:
        input-0:
          destination: topic/1
          binder: jms
      jms:
        bindings:
          input-0:
            consumer:
              destination-type: topic
              durable-subscription-name: subscriptionName
```

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 & JMS	Specifies the name of the dynamic destination to publish to. Setting this header overrides the configured destination.
`solace_connector_scst_targetDestinationType`	`string`	`(queue|topic)`	JMS	Specifies the destination type of the dynamic destination. When unspecified, the configured or default destination type is used.
`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

Asynchronous publishing allows the connector to process new messages without the need to wait for the previous message's publish acknowledgment. While this can boost performance, it does come with a trade-off, as it may increase the chances and volume of duplicate messages.

A workflow can be configured to wait for publisher acknowledgements asynchronously with the following config options:

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

solace.connector.workflows.<workflow-id>.acknowledgment.publish-async
solace.connector.workflows.<workflow-id>.acknowledgment.back-pressure-threshold
solace.connector.workflows.<workflow-id>.acknowledgment.publish-timeout
```

Consult the Workflow Configuration Options table for details and defaults values of those config options.

In general, reducing the `publish-timeout` increases the probability of duplicate message deliveries, while increasing the `back-pressure-threshold` is likely to result in a higher occurrence of duplicate messages.

This connector supports asynchronous publishing in the Solace → TIBCO EMS direction only. Enabling `publish-async` on a workflow in the TIBCO EMS → Solace direction enables asynchronous publishing on the connector's core but the effective publishing mode is still synchronous due to a lack of support for this feature on TIBCO EMS consumer bindings.

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

The above example configuration enables metrics collection through the configuration parameter of `management.metrics.export.simple.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 jms 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.

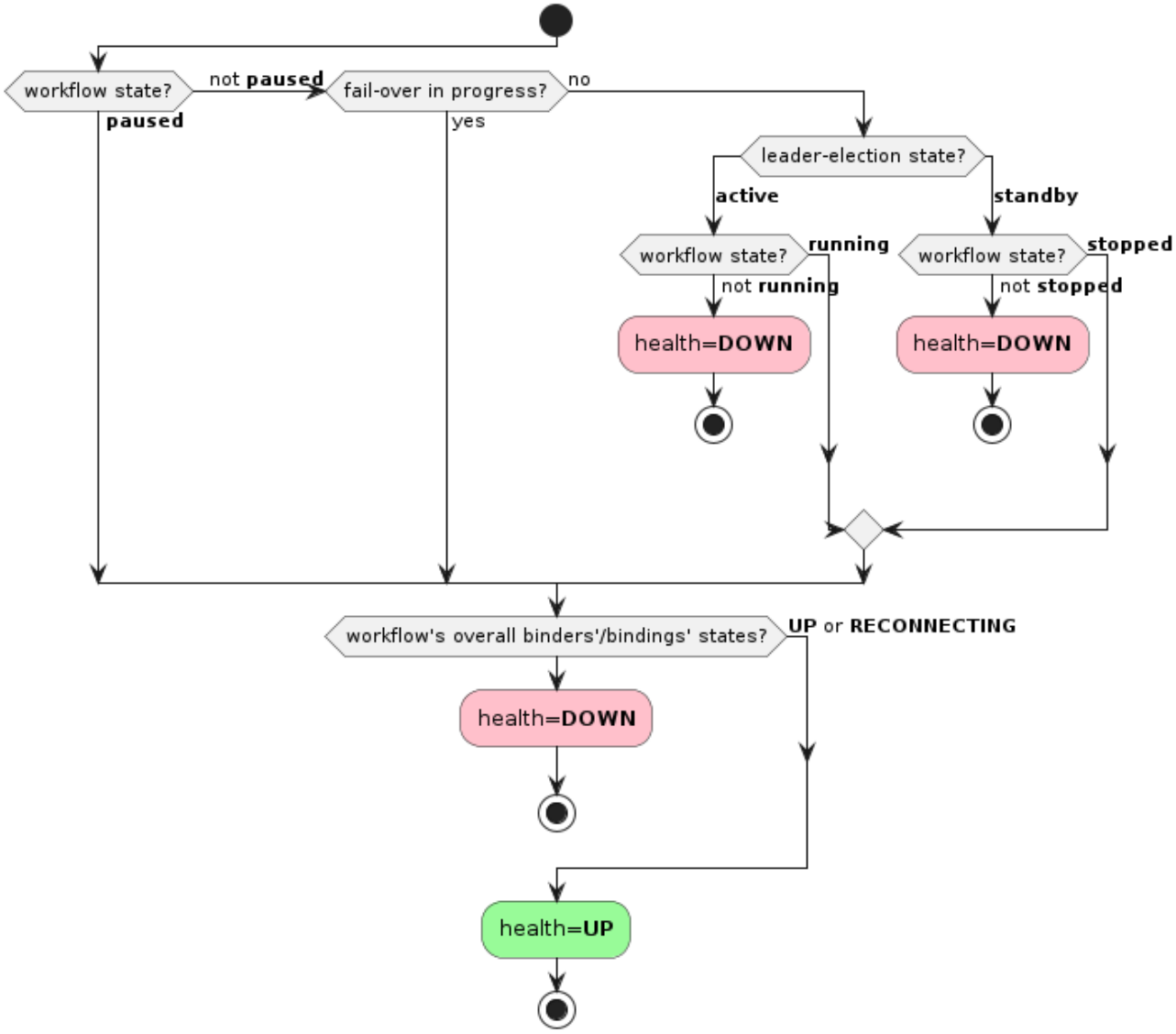

Figure 1. Workflow Health Resolution Diagram

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.

JMS Binder Health

Health Status	Description
UP	Status indicating that the binder is functioning as expected.
RECONNECTING	Status indicating that the binder is trying to reconnect to the message broker.

Health Status	Description
`DOWN`	Status indicating that the binder is having difficulties reconnecting to the message broker. The binder will automatically recover when underlying connectivity issues are resolved. User intervention may be required.

The length of time a JMS binder spends in the `RECONNECTING` state before moving to the `DOWN` state is configurable via the `jms-binder.health-check.interval` and `jms-binder.health-check.reconnect-attempts-until-down` config options. See the JMS Binder Configuration Options section for details.

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:** `{ "state": "(active|standby)", "mode": "(standalone|active_active|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.

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 only includes JMX as its supported monitoring system.

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.metrics.export.<system>.enabled=true in the application.yml.

Security

Securing Endpoints

Exposed Management Web Endpoints

By default, this connector only enables the `health` and `leaderelection` management endpoints. Where for the `health` endpoint, only the root status is returned 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.

CSRF Protection

Spring Boot enables CSRF protection by default on all management endpoints (see Spring Cross Site Request Forgery Protection). Though this connector disables CSRF protection for all POST requests on actuator endpoints so that users with write permissions (those with the `admin` role) can perform POST requests.

To fully disable CSRF protection, set the following config option:

```
solace.connector.security.csrf-enabled=false
```

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.

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

JMS Binder Configuration Options

The following properties are available at the binder level and are complementary to the properties described in the Configuring Connection Details section.

These properties are to be prefixed with `jms-binder`.

Config Option	Type	Valid Values	Default Value	Description
`health-check.interval`	`long`	`> 0`	`10000`	Interval (in ms) between reconnection attempts while health status is `RECONNECTING`

Config Option	Type	Valid Values	Default Value	Description
health-check.reconnect-attempts-until-down	long	>= 0	10	The number of reconnection attempts until JMS binder transitions from RECONNECTING to DOWN. A value of 0 means unlimited number of attempts which means that the binder would never transition to the DOWN state.
jndi.context	java.util.Properties	key/value pairs		Standard JNDI properties. See JNDI Context section for details.
jndi.connection-factory.name	string			The connection factory's JNDI name used for the lookup
jndi.connection-factory.user	string			The user to authenticate with the JMS broker.
jndi.connection-factory.password	string			The password to authenticate with the JMS broker.

JMS Consumer Options

The following config options are available to JMS consumers. Options within the same table share the same prefix.

Options prefixed with spring.cloud.stream.jms.bindings.<bindingName>.consumer.

Config Option	Type	Valid Values	Default Value	Description
`destination-type`	`String`	`(queue|topic|unknown)`	`unknown`	The type of destination where messages are consumed from. **queue** The destination value is assumed to be a physical queue and no JNDI lookup is done. **topic** The destination value is assumed to be a physical topic and no JNDI lookup is done. This option requires `durable-subscription-name` to also be set. **unknown** The destination value is assumed to be a JNDI name. The actual destination name is only known after a successful lookup. A valid JNDI context must be configured via `jms-binder.jndi.context`.
`durable-subscription-name`	`String`			The name of the shared durable subscription to consume from. The subscription is created on the broker if it doesn't already exist. Applies, and is mandatory, when `destination-type` is or resolves to a topic.

Options prefixed with `spring.cloud.stream.bindings.<bindingName>.consumer.`

Config Option	Type	Valid Values	Default Value	Description
`concurrency`	`int`	`> 0`	`1`	The number of concurrent consumers to create.

If a config option applies to all JMS input bindings, it can be prefixed with `spring.cloud.stream.jms.default.consumer.` if the option is from the first table or with `spring.cloud.stream.default.consumer.` if the option is from the second table. This is a convenient way to assign a configuration to all JMS input bindings.

JMS Producer Options

The following config options are available to JMS producers.

Options prefixed with `spring.cloud.stream.jms.bindings.<bindingName>.producer.`

Config Option	Type	Valid Values	Default Value	Description
`destination-type`	`string`	`(queue|topic|unknown)`	`unknown`	The type of destination where messages are published to. **queue** The destination value is assumed to be a physical queue and no JNDI lookup is done. **topic** The destination value is assumed to be a physical topic and no JNDI lookup is done. **unknown** The destination value is assumed to be a JNDI name. The actual destination name is only known after a successful lookup. A valid JNDI context must be configured via `jms-binder.jndi.context`.

If a config option applies to all JMS output bindings, it can be prefixed with `spring.cloud.stream.jms.default.producer.`. This is a convenient way to assign a configuration to all JMS output bindings.

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.

Config Option	Type	Valid Values	Default Value	Description
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.
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.

Config Option	Type	Valid Values	Default Value	Description
security.csrf-enabled	boolean	(true|false)	true	If true, CSRF protection is enabled. Makes sense only if solace.connector.security.enabled is true.
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.

Config Option	Applicable Scopes	Type	Valid Values	Default Value	Description
`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.
`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.