



Micro-Integration Framework Development Guide

Solace Corporation

Version 3.0.6

solace.

Table of Contents

Preface	1
Create new Micro-Integrations & Components with Archetype	2
Micro-Integration Archetype	2
Micro-Integration Archetype - Generated Contents	2
Micro-Integration Archetype - Parameters	2
Micro-Integration Binder Archetype	3
Micro-Integration Binder Archetype - Generated Contents	3
Micro-Integration Binder Archetype - Parameters	3
Architecture Overview	4
Binder Development	6
Producer Message Acknowledgments	7
Producer Acknowledgment Modes	7
Workflow Behavior Based on Capabilities	8
Resolving Asynchronous Publish Acknowledgments	8
Consumer Message Acknowledgments	10
Consumer Acknowledgment Modes	10
Implementing Asynchronous Consumer Acknowledgments	11
Message Interoperability Considerations	14
Standard Data Types and Serialization	14
Mid-Operation Data Type Conversions	14
Supporting Custom Types	15
Message Compatibility for Target Systems	15
Security	18
Sanitization of Sensitive Values on Endpoints	18
Optional Micro-Integration Runtime Customization	19
Providing Capability Information about a Binder	19
Consumer Binding Capabilities	19
Producer Binding Capabilities	20
Overriding Error-Base-Name Configuration	21
Binding Customizers	22
Binding Message Interceptors	23
Handle Third-Party Publish Acknowledgments	24
Handle Third-Party Consumer Acknowledgments	26
Binding Health Accessor	26
Message Transforms Customization	27
Always-Enabled Message Transforms Feature	27
Adding Serialization and Conversion Support for Custom Types in Messages	28
Reading from Custom Map-Like and List-Like Types in SpEL using Index Notation	30

Reading from Bean Types in SpEL using Index Notation	31
--	----

Preface

The Micro-Integration Framework provides a standardized approach for third-party applications to integrate with event brokers. It enables users to easily connect their data storage systems and streaming platforms into the Solace Event Mesh.

The Micro-Integration Framework serves as middleware that:

- Creates a bridge between diverse data sources and formats
- Facilitates data transformation and routing between systems
- Enables seamless integration between different technologies and protocols

It's important to note that the framework does not aim to implement complex business logic or application-specific intelligence, as these concerns fall outside its scope.

At its core, the Micro-Integration Framework is a library that implements specific design patterns and architectural approaches. It allows the integration of various loosely coupled components into a cohesive ecosystem, providing the foundation for building robust, scalable, and maintainable event-driven applications.

Create new Micro-Integrations & Components with Archetype

The framework provides a few different Maven archetype to quickly get you bootstrapped to make new micro-integrations and micro-integration components (e.g. binders).

Micro-Integration Archetype

To get started, run:

```
mvn archetype:generate \
  -DarchetypeGroupId=com.solace.microintegration.core.archetype \
  -DarchetypeArtifactId=micro-integration-archetype \
  -DarchetypeVersion=3.0.6
```

Then follow the prompts and your new micro-integration will be generated under a new directory using the same name that you provided for the `artifactId`.

From here, there are a few manual things that you might want to do:

- Add any additional dependencies to the POM (e.g., the vendor's binder).
- Review the generated `application.yml` files.

Micro-Integration Archetype - Generated Contents

- The generic stuff (at the root of the project):
 - A `pom.xml` preconfigured for a standard micro-integration build.
 - Maven wrapper (`mvnw` | `mvnw.cmd`)
- Source code (under `src/main`).
 - The generated `*.java` files will provide you with the micro-integration's main class.
 - The micro-integration's packaged configuration file (`src/main/resources/application.yml`).
 - A sample configuration file config (`application-sample.yml`).

Micro-Integration Archetype - Parameters

Name	Default Value	Description
<code>groupId</code>		The micro-integration's group ID.
<code>artifactId</code>		The micro-integration's artifact ID.
<code>vendorBinderId</code>		The ID of the vendor's binder.

Micro-Integration Binder Archetype

To get started, run:

```
mvn archetype:generate \
  -DarchetypeGroupId=com.solace.microintegration.core.archetype \
  -DarchetypeArtifactId=micro-integration-binder-archetype \
  -DarchetypeVersion=3.0.6
```

Then follow the prompts and your new binder will be generated under a new directory using the same name that you provided for the `artifactId`.



Optional parameters or parameters with default values can be provided at the command line using the `-D{parameterName}` syntax.

Micro-Integration Binder Archetype - Generated Contents

- The generic stuff (at the root of the project):
 - A `pom.xml` preconfigured for a standard binder build.
 - Maven wrapper (`(mvnw|mvnw.cmd)`)
- Source code (under `src/main`)



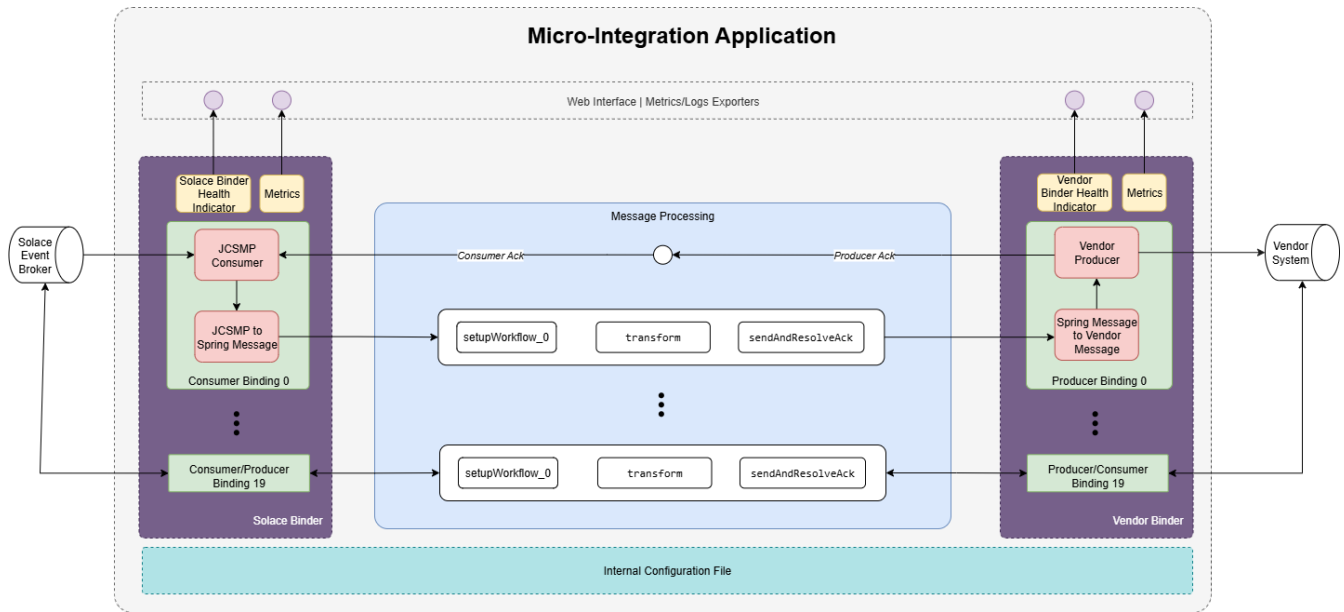
Especially for binders, whose implementations are vendor-sensitive, sources generated from this archetype should only be treated as example implementations.

- Generic `*.java` template code for binder components to get started with developing a binder.

Micro-Integration Binder Archetype - Parameters

Name	Default Value	Description
<code>groupId</code>		The micro-integration's group ID.
<code>artifactId</code>		The micro-integration's artifact ID.
<code>binderId</code>		The ID of the binder.

Architecture Overview



The diagram above illustrates the high-level architecture of the Micro-Integration Framework. This architecture facilitates seamless communication between Solace Event Brokers and third-party vendor systems through a standardized, configurable approach.

The Micro-Integration serves as a bridge between a Solace Event Broker and a Vendor System, with the following key components:

- **Binders (Left and Right Sides):** Connect the application to external systems.
 - **Solace Binder:** Interfaces with the Solace Event Broker for message consumption.
 - **Vendor Binder:** Interfaces with the third-party Vendor System for message production.
 - Binders must include:
 - Health indicator and (Optional) metrics components for monitoring
 - Message conversion capabilities from the vendor's message format to the Spring messaging format.
 - Appropriate binding functionality (Consumer/Producer bindings)
- **Message Processing (Center):** The core of the framework that processes messages between systems.
 - Handles all message processing concerns, including; message transformation, and producer → consumer acknowledgment bridging logic, with configurable beans for customization by specific micro-integrations.
 - Supports multiple workflow configurations (indicated by the vertical dots)
- **Internal Configuration API (Bottom):** The `src/main/resources/application.yml` file packed within the micro-integration distributable. This defines the workflows and other settings that control how the micro-integration operates.

The architecture follows a modular design pattern, allowing for flexible integration between different systems while maintaining a consistent approach to message handling, transformation,

and delivery acknowledgment.

Binder Development

This guide primarily focuses on the specific components and considerations uniquely introduced by the Micro-Integration Framework.

For general binder development concepts and best practices, please refer to the [Spring Cloud Stream documentation](#).

Producer Message Acknowledgments



Highly recommended that your binder supports asynchronous producer acknowledgments for optimal performance and throughput.

The framework handles producer message acknowledgments differently based on the **producer binding capabilities** that you declare. This allows workflows to be optimized for specific acknowledgment patterns.

See [Providing Capability Information about a Binder](#) for how to declare your producer's capabilities.

Producer Acknowledgment Modes

Your producer binding must declare one of the following acknowledgment modes through the `ProducerBindingCapabilities.getAcknowledgmentMode()` method:

1. `ProducerAckMode.SYNC` (Synchronous Acknowledgments)

- The producer publishes messages synchronously
- The message is considered acknowledged when the producer binding's `MessageHandler.handleMessage(Message<?>)` method returns successfully
- No `PublishAcknowledgmentCallback` header is provided to the producer binding
- **Use this mode when:** Your producer binding blocks until the message is confirmed published (e.g., synchronous HTTP calls, blocking database writes)

2. `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER` (Asynchronous Acknowledgments)

- The producer publishes messages asynchronously
- The framework provides a `ConnectorBinderHeaders.PUBLISH_ACKNOWLEDGMENT_CALLBACK` header containing a `PublishAcknowledgmentCallback` instance
- Your producer binding (or [producer binding message interceptor](#)) must resolve this callback to acknowledge messages
- **Use this mode when:** Your producer binding supports asynchronous publishes with callbacks (e.g., Kafka with callbacks, async HTTP clients, JCSMP async publishes)

When using `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER`, you **must** resolve the `PublishAcknowledgmentCallback` by calling either:



- `PublishAcknowledgmentCallback.onPublishSuccess()` - when the message is successfully published
- `PublishAcknowledgmentCallback.onPublishFailure(Throwable)` - when publishing fails



Critical: Declare the Correct Mode for Asynchronous Publishers

If your producer binding is an **asynchronous-only publisher** (i.e., it does not block waiting for publish confirmation when `MessageHandler.handleMessage(Message<?>)` returns), you **MUST** declare `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER` and properly resolve the `PublishAcknowledgmentCallback`.

Incorrectly declaring `ProducerAckMode.SYNC` for an async-only producer will result in **message loss** because the framework will consider the message published as soon as the producer binding's `handleMessage()` method returns, even though the message may not have been successfully published yet.

Workflow Behavior Based on Capabilities

The framework optimizes workflow behavior based on the combined capabilities of both the consumer and producer bindings:

Asynchronous Workflow Mode

Enabled when **all three** conditions are met:

- Workflow configuration: `solace.connector.workflows.<workflow_index>.acknowledgment.publishAsync=true`
- Consumer capability: `ConsumerAckMode.CLIENT_ACK_BY_CALLBACK_HEADER`
- Producer capability: `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER`

In this mode, the workflow does not block waiting for publish acknowledgments. The consumer message is acknowledged asynchronously after the producer's callback resolves.

Synchronous Workflow Mode

Used when any of the above conditions is not met. In this mode:

- If the producer declares `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER`: The workflow will block waiting for the publish acknowledgment callback to resolve (with configurable timeout via `solace.connector.workflows.<workflow_index>.acknowledgment.publishTimeout`)
- If the producer declares `ProducerAckMode.SYNC`: The workflow blocks until the producer binding's `handleMessage()` returns

The consumer message is acknowledged synchronously after the publish completes.



If your workflow is configured with `publishAsync=true` but the consumer or producer capabilities don't support async mode, the framework will log a warning and fall back to synchronous workflow mode.

Resolving Asynchronous Publish Acknowledgments

For producers that declare `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER`, you must resolve the `PublishAcknowledgmentCallback` header in your code.

When to resolve: The callback should be resolved in your producer binding's asynchronous publish callback or completion handler provided by the vendor's client library.

Example: Utility Method for Resolving Publish Acknowledgments

```
public void resolvePublishAck(Message<?> message, Exception ex) {
    PublishAcknowledgmentCallback publishAckCallback = message.getHeaders()
        .get(ConnectorBinderHeaders.PUBLISH_ACKNOWLEDGMENT_CALLBACK,
        PublishAcknowledgmentCallback.class);

    if (publishAckCallback != null) { ❶
        if (ex != null) {
            publishAckCallback.onPublishFailure(ex); ❷
        } else {
            publishAckCallback.onPublishSuccess(); ❸
        }
    }
}
```

- ❶ The callback is only present when the producer declares `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER`. If your producer declares `ProducerAckMode.SYNC`, this will be `null`.
- ❷ Call when the asynchronous publish fails.
- ❸ Call when the asynchronous publish succeeds.

Integration with vendor async callbacks: How you invoke this method depends entirely on your vendor's asynchronous publish mechanism. For example:

- **Kafka:** Resolve in the `Callback` passed to `producer.send(record, callback)`
- **JCSMP:** Resolve in the `JCSMPStreamingPublishCorrelatingEventHandler` callback
- **Async HTTP clients:** Resolve in the future/promise completion handlers

Do not resolve the callback synchronously



If your producer binding does not have a true asynchronous publish mechanism with callbacks, you **must** declare `ProducerAckMode.SYNC` in your producer capabilities instead.

Declaring `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER` but resolving the callback synchronously (e.g., immediately after the publish call returns) defeats the purpose of the async capability model and may lead to incorrect workflow behavior.



Handling Publish Acknowledgments Defined by Third-Party Binders

Third-party binders might provide their own custom headers/interfaces for handling publish acknowledgments. In such cases, see [Handle Third-Party Publish Acknowledgments](#).

Consumer Message Acknowledgments



Highly recommended that your binder supports asynchronous consumer acknowledgments for optimal performance and throughput.

The framework handles consumer message acknowledgments differently based on the **consumer binding capabilities** that you declare. This allows workflows to be optimized for specific acknowledgment patterns.

See [Providing Capability Information about a Binder](#) for how to declare your consumer's capabilities.

Consumer Acknowledgment Modes

Your consumer binding must declare one of the following acknowledgment modes through the `ConsumerBindingCapabilities.getAcknowledgmentMode()` method:

1. `ConsumerAckMode.AUTO_ACK` (Synchronous Acknowledgments)

- Messages are acknowledged immediately after the consumer binding processes them
- The consumer binding synchronously acknowledges messages after calling `MessageProducerSupport.sendMessage(Message)`
- The `IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK` header does not need to be set (the framework will ignore it if present)
- **Use this mode when:** Your consumer binding does not support deferred/callback-based acknowledgments (e.g., some simple polling consumers, auto-commit consumers)

2. `ConsumerAckMode.CLIENT_ACK_BY_CALLBACK_HEADER` (Asynchronous Acknowledgments)

- Messages are acknowledged at a later time through the `IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK` header
- Your consumer binding must implement `AcknowledgmentCallback` and set it in the message header
- The framework will invoke the callback to acknowledge messages after workflow processing completes
- **Use this mode when:** Your consumer binding supports deferred acknowledgments (e.g., manual commit Kafka consumers, JCSMP manual acknowledgments)



The consumer's acknowledgment mode affects workflow behavior. For a workflow to operate in asynchronous mode (non-blocking), it requires:

- Consumer capability: `ConsumerAckMode.CLIENT_ACK_BY_CALLBACK_HEADER`
- Producer capability: `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER`
- Workflow configuration:
`solace.connector.workflows.<workflow_index>.acknowledgment.publishAsync=true`

See [Producer Message Acknowledgments](#) for more details on workflow behavior.

Implementing Asynchronous Consumer Acknowledgments

For consumer bindings that declare `ConsumerAckMode.CLIENT_ACK_BY_CALLBACK_HEADER`:

1. Implement the `AcknowledgmentCallback` interface
2. Set this callback in the `IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK` header before sending the message to the framework

Please see [Spring Integration's documentation on deferred acknowledgments](#) for more information.

Example: Asynchronous Consumer Binding with Callback

```
public class VendorConsumerBinding extends MessageProducerSupport {
    @Override
    protected void doStart() {
        // Start consuming messages from vendor system
        vendorClient.consume(message -> {
            VendorAcknowledgmentCallback callback = new VendorAcknowledgmentCallback
(message);

            Message<?> springMessage = MessageBuilder
                .withPayload(convertPayload(message))
                .setHeader(IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK,
callback) ①
                .build();

            // Send to framework without acknowledging
            sendMessage(springMessage); ②
        });
    }
}
```

① Set the acknowledgment callback header that the framework will invoke later.

② Message is **not** acknowledged here - the framework will call the callback when appropriate.

Example AcknowledgmentCallback Implementation

```
public class VendorAcknowledgmentCallback implements AcknowledgmentCallback {
    private final Object sourceMessage;
    private boolean acknowledged = false;
    private boolean autoAckEnabled = true;

    // Constructor with whatever parameters you need for acknowledgment
}
```

```

public VendorAcknowledgmentCallback(Object sourceMessage) {
    this.sourceMessage = sourceMessage;
}

@Override
public void acknowledge(Status status) {
    if (acknowledged) {
        // Skip if already acknowledged
        return;
    }

    switch (status) {
        case ACCEPT -> {
            // TODO: Acknowledge the sourceMessage
        }
        case REJECT -> {
            // TODO: Reject the sourceMessage
        }
        case REQUEUE -> {
            // TODO: Requeue the sourceMessage
        }
    }

    acknowledged = true;
}

@Override
public boolean isAcknowledged() {
    return acknowledged;
}

@Override
public void noAutoAck() {
    autoAckEnabled = false; ①
}

@Override
public boolean isAutoAck() {
    return autoAckEnabled;
}
}

```

- ① The framework may call `noAutoAck()` to signal that the callback should **not** auto-acknowledge. When this is called, only acknowledge when the framework explicitly calls `acknowledge(Status)`.



Implementation Guidelines for `isAutoAck()`:

- When `isAutoAck()` returns `false`: The consumer binding should NOT synchronously acknowledge messages. Only acknowledge through the `AcknowledgmentCallback` when the framework calls `acknowledge(Status)`.

- When `isAutoAck()` returns `true`: The consumer binding MAY synchronously auto-acknowledge messages, but should also respond to explicit `acknowledge(Status)` calls.

Message Interoperability Considerations

When integrating different systems, understanding how messages are handled between them is critical.

Standard Data Types and Serialization

The framework standardizes and normalizes message components (headers and payloads) through consistent serialization rules.

Consider the following scenarios:

- Objects are assigned to `target['headers']` or `target['payload']` in SpEL expressions:

```
target['headers']['my-header'] = <my-object>
```

- During automatic payload pass-through scenarios.
 - i.e., when a workflow has no SpEL expressions which make an assignment to `target['payload']`.

In these scenarios, objects assigned to the `target` message's payload and headers will always be normalized to these standard types:

- `String`
- `Number` (integers, floating-point)
- `byte[]`
- `List<Object>`
- `Map<String, Object>`



The `Object` type in collections can only be one of the standard types listed above. For example, values in a `List<Object>` will strictly be one of: `String`, `Number`, `byte[]`, `List<Object>`, or `Map<String, Object>`.



Source Payload Limitation

Auto-serialization of non-standard source payloads is not currently supported by the message transforms feature.

You must implement a [ConsumerBindingMessageInterceptor](#) to convert the source message's payload to a standard type.

Mid-Operation Data Type Conversions

In some scenarios, the framework may need to convert objects to different types during the execution of SpEL operations or for other type coercion needs.

One common case is when passing objects to functions, where the framework will automatically convert the object to match the function's parameter type.

```
#someFunction(<my-object>) // <my-object> is converted to match function parameter type
```

Supporting Custom Types

To extend the framework to handle vendor-specific types, please consult:

- [Adding Serialization and Conversion Support for Custom Types in Messages](#)
- (Optional) [Reading from Custom Map-Like and List-Like Types in SpEL using Index Notation](#)
- (Optional) [Reading from Bean Types in SpEL using Index Notation](#)

Message Compatibility for Target Systems

While the framework does [normalize all message payloads and headers to standard types](#) before passing them to producer bindings, some target systems may have more restrictive requirements. For example, Kafka only accepts `byte[]` payloads.

One solution is to simply fail the message and throw an exception when the message contains a non-compatible payload or header type. In which case, it would be the micro-integration user's responsibility to ensure that the values they pass to the `target['headers']` or `target['payload']` are compatible with the target system.

Another solution is to instead implement a `ProducerBindingMessageInterceptor` to transform messages into formats compatible with the target system.



Do Not Handle Message Compliance for the Target System in Consumer Bindings

DO NOT handle publisher message compliance in a `ConsumerBindingMessageInterceptor`. Target system message compliance is only needed for the producer binding.

Handling it right after the consumer binding can impact message transformation capabilities, leading to degraded performance or unexpected behavior.



Do Not Rename or Map Vendor-Specific Headers

DO NOT rename or map vendor-specific headers to different names (e.g., do not change `kafka_destination` to `solace_destination`).

Headers have different meanings in different systems. Renaming or mapping them can lead to unexpected behavior.

Example Implementation:

```

public class VendorProducerBindingMessageInterceptorFactory implements
ProducerBindingMessageInterceptorFactory {
    private final ObjectMapper objectMapper;

    public VendorProducerBindingMessageInterceptorFactory(
        MimeTypesObjectMapperRegistry mimeTypeObjectMapperRegistry) {
        // Get the ObjectMapper used for serializing the target message with
        application/json
        this.objectMapper = mimeTypeObjectMapperRegistry.getTargetMapper(
            MimeTypesUtils.APPLICATION_JSON);
    }

    @Nullable
    @Override
    public ProducerBindingMessageInterceptor createIfNecessary(String binderType,
ProducerProperties producerProperties) {
        if (binderType.equals("vendor")) {
            return new Interceptor();
        } else {
            return null;
        }
    }

    private static class Interceptor implements ProducerBindingMessageInterceptor {
        private final ObjectMapper objectMapper;

        public Interceptor(ObjectMapper objectMapper) {
            this.objectMapper = objectMapper;
        }

        @NonNull
        @Override
        public Message<?> before(Message<?> message) {
            // Transform the message to a format that the "vendor" target system can
            understand.
            // In this example, suppose that the target system only supports messages with
            // 'byte[]' payloads.
            byte[] newPayload;
            if (payload instanceof byte[] bytesPayload) {
                newPayload = bytesPayload;
            } else {
                try {
                    newPayload = objectMapper.writeValueAsBytes(payload);
                } catch (JsonProcessingException e) {
                    throw new RuntimeException("Failed to serialize object to byte array", e);
                }
            }

            return new GenericMessage<>(newPayload, message.getHeaders());
        }
    }
}

```

```
}  
}
```

See [Binding Message Interceptors](#) for more info.

Security

Sanitization of Sensitive Values on Endpoints



Default sanitization rules are largely based off of [Spring Boot 2.7.17 default rules](#).

Information returned by the `bindings`, `env` and `configprops` endpoints can be somewhat sensitive so keys matching certain patterns are sanitized by default (their values are replaced by `*****`).



Keys adhere to [Spring Boot's flat properties format](#).

The following keys are entirely sanitized:

- any key ending with the word:
 - `password`
 - `secret`
 - `key`
 - `token`
 - `vcap_services`
 - `sun.java.command`
- Any key that contains the word (i.e. a regex matching `.*<word>.*`):
 - `credentials`

Furthermore, the sensitive portion of URI-like values are sanitized for keys ending with the word:

- `address`
- `addresses`
- `uri`
- `uris`
- `url`
- `urls`

The sensitive portion of the URI is identified using the format `<scheme>://<username>:<password>@<host>:<port>/`. For example, for the property `myclient.uri=http://user1:password1@localhost:8081`, the resulting sanitized value is `http://user1:*****@localhost:8081`.



Adding Custom Endpoint Sanitization

To further customize endpoint sanitization, see the [Spring docs](#).

Optional Micro-Integration Runtime Customization

These are beans and strategies that you can implement in your micro-integration to customize Spring Cloud Streams and the Micro-Integration Framework.

Providing Capability Information about a Binder



Especially useful when building micro-integrations using third party binders.

The framework provides a generic mechanism for binders to communicate their capabilities through the `ConsumerBindingCapabilitiesFactory` and `ProducerBindingCapabilitiesFactory` interfaces. These factories create capability objects for each binding during workflow initialization, allowing the framework to adapt its behavior based on the specific features and characteristics of your binder.

Currently supported capabilities:

- **Acknowledgment modes** (primary use case): Whether the binding supports synchronous or asynchronous acknowledgments
- **Error channel configuration**: The error-base-name format used by the binder (if overridden from the default)

This capability mechanism is designed to be extensible, allowing future versions of the framework to query additional binding characteristics as needed.

Consumer Binding Capabilities

Implement the `ConsumerBindingCapabilitiesFactory` interface to provide consumer binding capability information to the framework.

Example Implementation for a Vendor Consumer Binder

```
public class VendorConsumerBindingCapabilitiesFactory implements
ConsumerBindingCapabilitiesFactory {
    @Override
    public String getBinderType() {
        return "vendor"; ①
    }

    @Override
    public ConsumerBindingCapabilities create(ConsumerProperties properties) {
        return new VendorConsumerBindingCapabilities(properties.getBindingName());
    }

    private static class VendorConsumerBindingCapabilities implements
ConsumerBindingCapabilities {
```

```

private final String bindingName;

VendorConsumerBindingCapabilities(String bindingName) {
    this.bindingName = bindingName;
}

@Override
public String getBindingName() {
    return bindingName;
}

@Override
public ConsumerAckMode getAcknowledgmentMode() {
    return ConsumerAckMode.CLIENT_ACK_BY_CALLBACK_HEADER; ②
}
}
}

```

- ① Specify the binder type this factory applies to.
- ② Specify the acknowledgment mode supported by this consumer binding:
 - **AUTO_ACK**: Binder handles acknowledgments automatically (synchronous)
 - **CLIENT_ACK_BY_CALLBACK_HEADER**: Binder supports callback-based acknowledgments via the `IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK` header (asynchronous)

Producer Binding Capabilities

Implement the `ProducerBindingCapabilitiesFactory` interface to provide producer binding capability information to the framework.

Example Implementation for a Vendor Producer Binder

```

public class VendorProducerBindingCapabilitiesFactory implements
ProducerBindingCapabilitiesFactory {
    @Override
    public String getBinderType() {
        return "vendor"; ①
    }

    @Override
    public ProducerBindingCapabilities create(ProducerProperties properties) {
        return new VendorProducerBindingCapabilities(properties.getBindingName());
    }

    private static class VendorProducerBindingCapabilities implements
ProducerBindingCapabilities {
        private final String bindingName;

        VendorProducerBindingCapabilities(String bindingName) {
            this.bindingName = bindingName;
        }
    }
}

```

```

@Override
public String getBindingName() {
    return bindingName;
}

@Override
public ProducerAckMode getAcknowledgmentMode() {
    return ProducerAckMode.ASYNC_BY_CALLBACK_HEADER; ②
}
}
}

```

- ① Specify the binder type this factory applies to.
- ② Specify the acknowledgment mode supported by this producer binding:
 - **SYNC**: Producer publishes messages synchronously and waits for acknowledgment
 - **ASYNC_BY_CALLBACK_HEADER**: Producer publishes messages asynchronously and receives acknowledgment via the `ConnectorBinderHeaders.PUBLISH_ACKNOWLEDGMENT_CALLBACK` header

Overriding Error-Base-Name Configuration



Only override the error-base-name if you're working with a pre-made third-party binder that has overwritten `AbstractMessageChannelBinder.errorsBaseName()`.

With the fix for [Error channel name collision #2507](#) in Spring Cloud Stream 4.x, it's recommended that SCSt binders no longer overwrite `AbstractMessageChannelBinder.errorsBaseName()`.

However, when working with pre-made third-party binders, it's possible that the developers of that binder may still be overwriting that function. In such cases, you must override the `getErrorsBaseName()` method in your capability implementation to inform the Micro-Integration Framework about the errors-base-name format that this binder uses.

Example: Overriding Consumer Error-Base-Name

```

private static class VendorConsumerBindingCapabilities implements
ConsumerBindingCapabilities {
    private final String bindingName;

    VendorConsumerBindingCapabilities(String bindingName) {
        this.bindingName = bindingName;
    }

    @Override
    public String getBindingName() {
        return bindingName;
    }

    @Override
    public String getErrorsBaseName(

```



```

    String destination,
    String group,
    Binder<?, ?, ?> binder) {
    // Override to match the vendor binder's custom errors base name format
    return String.format("%s.%s.errors", destination, group); ❶
}

```

- ❶ In this example, suppose that the **vendor** binder has overwritten `AbstractMessageChannelBinder.errorsBaseName()` such that its consumer binding's errors base name takes on the form of `<destination>.<group>.errors`.

Example: Overriding Producer Error-Base-Name

```

private static class VendorProducerBindingCapabilities implements
ProducerBindingCapabilities {
    private final String bindingName;

    VendorProducerBindingCapabilities(String bindingName) {
        this.bindingName = bindingName;
    }

    @Override
    public String getBindingName() {
        return bindingName;
    }

    @Override
    public String getErrorsBaseName(String destination, Binder<?, ?, ?> binder) {
        // Override to match the vendor binder's custom errors base name format
        return String.format("%s.errors", destination); ❶
    }
}

```

- ❶ In this example, suppose that the **vendor** binder has overwritten `AbstractMessageChannelBinder.errorsBaseName()` such that its producer binding's errors base name takes on the form of `<destination>.errors`.

Binding Customizers



Especially useful when building Micro-Integrations using third party binders.

When integrating third party binders, you'll sometimes find that you might be missing a feature from the binder's Spring configuration properties to properly integrate the binder with the micro-integration framework.

Often times these features may already exist in the underlying libraries that implement the consumer and producer bindings. In which case, you can create binding customizer beans to manually configure these features in your micro-integration's code:

- [Consumer Binding Customizer](#)
- [Producer Binding Customizer](#)

Binding Message Interceptors



Especially useful when building micro-integrations using third party binders.

`ConsumerBindingMessageInterceptor` and `ProducerBindingMessageInterceptor` are micro-integration components that can be used to modify messages before/after bindings.

To enable message interceptors, you must create factory beans that implement either `ConsumerBindingMessageInterceptorFactory` or `ProducerBindingMessageInterceptorFactory`. These factories determine when interceptors should be created based on the binder type and binding name.

The framework will automatically detect these factory beans and call `createIfNecessary()` for each binding. Return `null` if no interceptor is needed for that specific binding.

Example: Consumer Interceptor to Inject a Vendor Binder's Custom AcknowledgmentCallback Header

```
public class VendorConsumerBindingMessageInterceptorFactory implements
ConsumerBindingMessageInterceptorFactory {
    @Override
    @Nullable
    public ConsumerBindingMessageInterceptor createIfNecessary(String binderType,
ConsumerProperties consumerProperties) {
        if (binderType.equals("vendor")) { ①
            return new Interceptor();
        } else {
            return null;
        }
    }

    private static class Interceptor implements ConsumerBindingMessageInterceptor {
        @Override
        @NonNull
        public Message<?> after(Message<?> message) { ②
            return MessageBuilder.fromMessage(message)
                .setHeader(IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK, new
VendorAcknowledgmentCallback()) ③
                .build();
        }
    }
}
```

- ① Only create this consumer interceptor for workflows which consume from the `vendor` binder.
- ② This interceptor only applies **after** a `vendor` consumer binding.
- ③ This example assumes that the `vendor` consumer binding doesn't already inject its own

`AcknowledgmentCallback` header.

Example: *Producer Interceptor to Inject Vendor-Required Headers*

```
public class VendorProducerBindingMessageInterceptorFactory implements
ProducerBindingMessageInterceptorFactory {
    @Override
    @Nullable
    public ProducerBindingMessageInterceptor createIfNecessary(String binderType,
ProducerProperties producerProperties) {
        if (binderType.equals("vendor")) { ①
            return new Interceptor();
        } else {
            return null;
        }
    }

    private static class Interceptor implements ProducerBindingMessageInterceptor {
        @Override
        @NonNull
        public Message<?> before(Message<?> message) { ②
            // Inject vendor-specific headers required before publishing
            return MessageBuilder.fromMessage(message)
                .setHeader("vendor_qualityOfService", 1) ③
                .build();
        }
    }
}
```

- ① Only create this producer interceptor for workflows which publish to the `vendor` binder.
- ② This interceptor applies **before** the message is handed to the `vendor` producer binding.
- ③ Example: Inject vendor-specific headers that the producer binding requires or expects.

Handle Third-Party Publish Acknowledgments

`PublishAcknowledgmentProvider` is a micro-integration or binder bean to inject functionality for integrating third-party publish acknowledgment handlers.



This provider is **only used** when your producer binding declares `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER` in its capabilities.

If your producer declares `ProducerAckMode.SYNC`, this provider will not be consulted. See [Providing Capability Information about a Binder](#).

If the micro-integration is **NOT** using a third-party binder, then it may be simpler for binder implementations to directly handle the [Micro-Integration Framework's default publish acknowledgment callback header](#).



Do NOT complete the `CompletableFuture` immediately in `createFuture()`

The future returned by this method **must only** complete when the vendor's asynchronous publish callback is invoked. This method is called before handing the message to the output binding, so completing the future immediately (synchronously) defeats the entire purpose of asynchronous acknowledgments and contradicts your declared `ProducerAckMode.ASYNC_BY_CALLBACK_HEADER` capability.

Completing the future synchronously will result in incorrect workflow behavior as the framework will treat the publisher as truly asynchronous when it is not, leading to potential race conditions, message loss, or other unintended consequences at workflow runtime.

If your binder does not have a vendor-provided asynchronous publish mechanism with callbacks, you **must** declare `ProducerAckMode.SYNC` in your producer capabilities instead.

Example Implementation

```
public class VendorPublishAcknowledgmentProvider implements
PublishAcknowledgmentProvider {
    @Override
    public String binderType() {
        return "vendor";
    }

    @Override
    public CompletableFuture<Void> createFuture(MessageHeaderAccessor headerAccessor,
                                                Object payload) {
        VendorPublishHandle publishHandle = new VendorPublishHandle(); ①
        headerAccessor.setHeader("vendor_publishHandle", publishHandle); ②
        return publishHandle.getFuture(); ③
    }
}
```

- ① Create a vendor-specific object that correlates this message with the vendor's async publish callback mechanism.
- ② Inject the correlation object as a header so the producer binding can access it.
- ③ Return the future from the vendor object. You may need to adapt/convert the future to `CompletableFuture<Void>` depending on what your vendor object returns.



Consider whether you need this provider at all

This provider is primarily useful for vendors that use header-based correlation patterns for async acknowledgments. However, if your third-party binder already provides its own callback/listener mechanisms for publish completion (e.g., per-message publish listeners, completion handlers), you may not need to implement this provider.

Instead, use those vendor-provided mechanisms directly to resolve the

`PublishAcknowledgmentCallback` header in your producer binding or interceptor. This is often simpler and more natural than injecting correlation headers.

Handle Third-Party Consumer Acknowledgments

`AsyncConsumerAcknowledgmentCustomizer` is a micro-integration or binder bean for injecting third-party consumer acknowledgment handlers.

By default, messages are negatively acknowledged by `REJECTing` and acknowledging the message by `ACCEPTing`. However, binder may need to override it and that's when the bean of this interface should be created.

Example Implementation to Override a Binder's to `QUEUE` instead of `REJECT` when `NACKing`

```
public class VendorAsyncConsumerAcknowledgmentCustomizer implements
AsyncConsumerAcknowledgmentCustomizer {
    @Override
    public String binderType() {
        return "vendor";
    }

    @Override
    public void doNack(AcknowledgmentCallback acknowledgmentCallback, Throwable
exception) {
        AckUtils.queue(acknowledgmentCallback);
    }
}
```

Binding Health Accessor

`BindingHealthAccessor` is a bean which may be implemented by the micro-integration or binder to customize the health computation for a particular binding. The returned binding health will be used for features like computing a particular workflow's health status.



For more functionality, instead of implementing this interface, you can instead extend the `BindingHealthAggregateAccessor` class, which provides the following convenience methods:

`collectAggregateHealthStatus(HealthContributor healthContributor)`

Walk the health contributor aggregating the health for each nested health indicator. If an indicator returns a value of `null`, then it will be treated as `Status.DOWN`.

`collectAggregateHealthStatus(HealthContributor healthContributor, Status defaultStatus)`

Walk the health contributor aggregating the health for each nested health indicator. Returns `defaultStatus` if an indicator returns a `null` value.

Set this parameter to your lowest severity status (usually `Status.UNKNOWN`) to effectively ignore the indicator in the aggregation.

`getStatusAggregator()`

Get the status aggregator used to compute a single health status from multiple health statuses.

Example Vendor Binding Health Accessor Implementation

```
public class VendorBindingHealthAccessor extends BindingHealthAggregateAccessor {
    public VendorBindingHealthAccessor(StatusAggregator statusAggregator) {
        super(statusAggregator);
    }

    @Override
    public String getBinderType() {
        return "vendor";
    }

    @Override
    public Health getBindingHealth(
        String bindingName,
        HealthContributor binderHealthContributor) {
        if (binderHealthContributor instanceof
            VendorBinderHealthContributor binderHealthContrib) {

            // Extract the binding health contributor from the binder health
            contributor
            VendorBindingHealthContributor bindingHealthContrib = binderHealthContrib
                .getBindingHealthContributor(bindingName);

            // Return the aggregate health status just for the binding
            return collectAggregateHealthStatus(bindingHealthContrib);
        } else {
            // Fallback to default implementation
            return super.getBindingHealth(bindingName, binderHealthContributor);
        }
    }
}
```

Message Transforms Customization

Always-Enabled Message Transforms Feature

Since the Message Transforms feature has gone GA, every new micro-integration released from then-onward must have `solace.connector.default.workflow.transform.enabled` always be set to `true` in its `application.yml`.

You must also add the `RequiresTransformEnabledConfigurationValidation` bean into your application:

```
@SpringBootApplication
public class MyMicroIntegration {
    @Bean
    public ConfigurationValidation requiresTransformEnabledValidation() {
        return new RequiresTransformEnabledConfigurationValidation();
    }
}
```

This configuration will ensure that the value of `solace.connector.workflows.<workflow_index>.transform.enabled` is always set to `true`.

Adding Serialization and Conversion Support for Custom Types in Messages

When message components are JDK types, common Spring types, or JCSMP types, then [the framework handles serialization and auto-conversions](#).

However, if the vendor that you are integrating with provides a custom type, you may need to inform the framework how to serialize that type. This can be done by implementing the `TargetMessageJacksonModuleProvider` bean:

```
public class MyVendorJacksonModuleProvider implements
TargetMessageJacksonModuleProvider {
    @Override
    public Module getModule() {
        return new MyVendorModule();
    }

    @Override
    public Set<TypeDescriptor> getSerializableClasses() {
        // Informs the framework that MyCustomType can be serialized by the Module
        // returned by getModule()
        return Set.of(TypeDescriptor.valueOf(MyCustomType.class));
    }

    @Override
    public Set<ConvertibleTypePair> getConvertibleTypes() {
        // For registration into SpEL's type converter to delegate to jackson
        // for mid-operation type conversions.
        // Or to be leveraged by the transform service to determine compatible types.
        // The target (2nd parameter) of the ConvertibleTypePair doesn't have to be a
        // standard type.
        // e.g. The target could be Locale.class or something if you think that's
        // appropriate.
        return Set.of(new ConvertibleTypePair(
            TypeDescriptor.valueOf(MyCustomType.class),
            TypeDescriptor.valueOf(String.class)
        ));
    }
}
```

```
}
```

```
public class MyVendorModule extends SimpleModule {
    public MyVendorModule() {
        addSerializer(new MyCustomTypeSerializer(MyCustomType.class));
    }

    private static class MyCustomTypeSerializer extends StdSerializer<MyCustomType> {
        private MyCustomTypeSerializer(Class<MyCustomType> t) {
            super(t);
        }

        @Override
        public void serialize(MyCustomType value, JsonGenerator gen,
            SerializerProvider provider) throws IOException {
            // suppose MyCustomType.asString() exists which returns a String
            provider.defaultSerializeValue(value.asString(), gen);
        }
    }
}
```

What the above example enables is:

1. Target Message Serialization

Your custom type can be [serialized when written to target messages](#):

```
target['headers']['my-header'] = source['headers']['myCustomType']
```

If `myCustomType` header contains a `MyCustomType` object, then based on your `getSerializableClasses()` and `MyCustomTypeSerializer`, the value of `target['headers']['my-header']` will be the String result of `myCustomType.asString()`.

2. Mid-Operation Type Conversions

Your custom type can be [automatically converted as required mid-operation](#). Such as for a function parameter:

```
#someFunction(source['headers']['myCustomType'])
```

If `myCustomType` header contains a `MyCustomType` object, and `someFunction(String)` is a method whose parameter is a `String`, then based on your `getConvertibleTypes()` and `MyCustomTypeSerializer`, the parameter given to `someFunction()` will be the String result of `myCustomType.asString()`.

Implementation Guidelines:

- For `target[(headers|payload)] serialization` only: Implement just `getSerializableClasses()`

- For [mid-operation data type conversions](#) only: Implement just `getConvertibleTypes()`
- For both capabilities: Implement both methods (**highly recommended**)

Make the Conversion Target Type as Generic as Possible

When defining the target type of a `ConvertibleTypePair`, try to make it as generic as possible and use the highest level abstractions available.

For example, suppose that I had a list-like custom type, `MyListLikeType`, which accepts arbitrary objects as values. And I want this thing to be convertible as a list and array, then I would define the following convertible type pairs:



```
@Override
public Set<ConvertibleTypePair> getConvertibleTypes() {
    return Set.of(
        // By assigning the target type as Collection<Object>,
        // ConversionService is smart enough to downcast
        // this Collection<Object> to more specific collection
        // types as needed.
        new ConvertibleTypePair(
            TypeDescriptor.valueOf(MyListLikeType.class),
            TypeDescriptor.collection(Collection.class,
            TypeDescriptor.valueOf(Object.class))),

        // By assigning the target type as Object[],
        // ConversionService is smart enough to downcast this
        // Object[] to more specific array types as needed.
        new ConvertibleTypePair(
            TypeDescriptor.valueOf(MyListLikeType.class),
            TypeDescriptor.array(TypeDescriptor.valueOf(Object
            .class))))
    );
}
```

Reading from Custom Map-Like and List-Like Types in SpEL using Index Notation

Out-of-the-box, SpEL is able to read values from `Map<String, Object>`, `List<Object>`, and array types using index notation. e.g. `myMap['key']` or `myList[0]`.

However, when you have a custom type that is neither a `Map<String, Object>`, a `List<Object>`, nor an array, but is kind of "like a map" or "like a list", then beyond [adding serialization and conversion support for your custom type](#), you can also make the custom type readable in SpEL via index notation by implementing a `ConnectorTransformIndexAccessorProvider` bean for it:

```
public class MyMapLikeTypeIndexAccessorProvider implements
ConnectorTransformIndexAccessorProvider {
    @Override
```

```

public IndexAccessor getIndexAccessor() {
    // In this example, lets assumes that `MapLikeType` has a `getValue(String)`
    method.
    // in which case, the returned ReflectiveIndexAccessor will allow for SpEL to
    read from `MyCustomType` with
    // index notation that uses reflective access to
    `MapLikeType.getValue(String)` for its implementation.
    return new ReflectiveIndexAccessor(MapLikeType.class, String.class,
    "getValue");
}
}

```

In this example, this will allow for SpEL to read from `MyMapLikeType` with index notation like `myMapLikeObject['key']`, which under the hood, will invoke `myMapLikeObject.getValue("key")`.

Reading from Bean Types in SpEL using Index Notation

Dot Notation is Not Supported



The Micro-Integrations do not support dot notation (`a.b`) in transform SpEL expressions. But supporting index notation for bean types is done by implementing the interfaces for dot notation, which is why some interfaces here are named as such.

Standard SpEL is able to read values from objects using property names that match a "getter method" (i.e. has the syntax `get<Name>()`, `is<Name>()`, or `<Name>()`) using index notation `a['b']`.

For example, looking at JCSMP's `Destination` object, notice that there's a `getName()` method. This means that a user can read the `name` property from a `Destination` object with the SpEL expression, `myDestination['name']`, which will use reflection under the hood to invoke `Destination.getName()`.

However, reflective property access is not enabled by default in the Micro-Integrations' message transformations. If you want this, then you can opt in by implementing a `ConnectorTransformPropertyAccessorProvider` bean which returns a `ConstrainedReflectivePropertyAccessor` for your vendor type's specific getter method:

```

public class MyBeanTypeNamePropertyAccessorProvider implements
ConnectorTransformPropertyAccessorProvider {
    @Override
    public PropertyAccessor getPropertyAccessor() {
        // In this example, lets assume that `MyBeanType` has a `getName()` method
        which returns a `String`.
        return new ConstrainedReflectivePropertyAccessor(MyBeanType.class, String
        .class, "getName");
    }
}

```

In this example, this will allow for SpEL to read from `MyBeanType` with index notation (e.g. `myBeanObject['name']`). Under the hood, this will invoke `myBeanObject.getName()`.

If your use case cannot be satisfied by calling a single getter method, then you can implement a custom `PropertyAccessor` that can handle more complex property access patterns, and return it in your `ConnectorTransformPropertyAccessorProvider` bean.