

# **EventBusClient**

**v. 0.2.0**

Nguyen Huynh Tri Cuong

29.01.2026

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Description</b>	<b>2</b>
2.1	Overview	2
2.2	Features	2
2.3	Architecture	2
2.4	Installation	3
2.5	Dependencies	3
2.6	Directory Structure	3
2.7	Configuration	4
2.8	Usage	5
2.8.1	Initialization	5
2.8.2	Publishing Messages	6
2.8.3	Subscribing to Messages	6
2.8.4	Header-Based Subscriptions (HeadersExchangeHandler)	6
2.8.5	Setting Unroutable Message Policy	7
2.9	Example: Producer and Consumer	8
2.10	Example: Headers Exchange	8
2.11	Example: Custom Plugin Structure	8
2.12	Example: Built-in Plugins	9
2.13	Example: Built-in Configuration	10
2.14	Example: Custom Exchange Handler	10
2.15	Example: Custom Message Class	10
2.16	Example: Custom Serializer	11
<b>3</b>	<b>__init__.py</b>	<b>12</b>
<b>4</b>	<b>connection.py</b>	<b>13</b>
4.1	Method: is_connected	13
4.2	Method: reset_loop	13
4.3	Method: connect	13
4.4	Method: register_exchange_handler	13
4.5	Method: unregister_exchange_handler	14
4.6	Method: get_channel	14
4.7	Method: close	14
4.8	Method: recreate_channel	14
4.9	Method: reconnect	14

<b>5</b>	<b>constants.py</b>	<b>15</b>
5.1	Class: SocketType	15
5.2	Class: String	15
<b>6</b>	<b>event_bus_client.py</b>	<b>16</b>
6.1	Function: get_running_loop_or_none	16
6.2	Function: is_on_loop	16
6.3	Function: main	16
6.4	Class: EventBusClient	16
6.4.1	Method: getVersion	16
6.4.2	Method: getVersionDate	16
6.4.3	Method: enable_general_cache	16
6.4.4	Method: wait_on_general_topic_for_one	16
6.4.5	Method: wait_on_general_topic_for_many	17
6.4.6	Method: wait_on_cache_for_one	18
6.4.7	Method: wait_on_cache_for_many	18
6.4.8	Method: constructor_params_from_config	19
6.4.9	Method: from_config	20
6.4.10	Method: connect	20
6.4.11	Method: send	21
6.4.12	Method: off	21
6.4.13	Method: on	21
6.4.14	Method: wait_until_ready	22
6.4.15	Method: announce_ready	22
6.4.16	Method: close	22
6.4.17	Method: is_connected	23
6.4.18	Method: build_routing	23
6.4.19	Method: build_routing_key	23
6.4.20	Method: start_background_loop	23
6.4.21	Method: kill_aiormq_tasks_now	24
6.4.22	Method: stop_background_loop	24
6.4.23	Method: from_config_sync	24
6.4.24	Method: connect_sync	24
6.4.25	Method: send_sync_compat	24
6.4.26	Method: send_sync	25
6.4.27	Method: on_sync	25
6.4.28	Method: off_sync	26
6.4.29	Method: wait_until_ready_sync	26
6.4.30	Method: announce_ready_sync	27
6.4.31	Method: close_sync	27
6.4.32	Method: pop_unroutables	27
<b>7</b>	<b>base.py</b>	<b>28</b>
7.1	Method: configure_unroutable	28
7.2	Method: reset_loop	28
7.3	Method: setup	28
7.4	Method: with_alternate_exchange	29

7.5	Method: setup_alterate_exchange . . . . .	29
7.6	Method: finalize_setup . . . . .	29
7.7	Method: teardown . . . . .	29
7.8	Method: publish . . . . .	29
7.9	Method: subscribe . . . . .	29
7.10	Method: unsubscribe . . . . .	29
7.11	Method: handle_channel_close . . . . .	29
<b>8</b>	<b>fanout_handler.py</b>	<b>30</b>
8.1	Method: setup . . . . .	30
8.2	Method: publish . . . . .	30
8.3	Method: subscribe . . . . .	31
<b>9</b>	<b>headers_handler.py</b>	<b>32</b>
9.1	Method: setup . . . . .	32
9.2	Method: publish . . . . .	32
9.3	Method: subscribe . . . . .	33
9.4	Method: subscribe_with_headers . . . . .	34
<b>10</b>	<b>topic_handler.py</b>	<b>35</b>
10.1	Method: setup . . . . .	35
10.2	Method: publish . . . . .	35
10.3	Method: subscribe . . . . .	35
<b>11</b>	<b>x_rtopic_handler.py</b>	<b>37</b>
11.1	Method: setup . . . . .	37
11.2	Method: publish . . . . .	37
11.3	Method: subscribe . . . . .	37
<b>12</b>	<b>base_message.py</b>	<b>39</b>
12.1	Method: from_data . . . . .	39
12.2	Method: get_value . . . . .	39
<b>13</b>	<b>basic_message.py</b>	<b>40</b>
13.1	Method: to_dict . . . . .	40
13.2	Method: from_dict . . . . .	40
13.3	Method: from_data . . . . .	40
13.4	Method: get_value . . . . .	41
<b>14</b>	<b>control_message.py</b>	<b>42</b>
14.1	Method: get_value . . . . .	42
14.2	Method: from_data . . . . .	42
<b>15</b>	<b>__init__.py</b>	<b>43</b>
<b>16</b>	<b>dict_msg.py</b>	<b>44</b>
<b>17</b>	<b>float32_msg.py</b>	<b>45</b>
<b>18</b>	<b>float64_msg.py</b>	<b>46</b>

<b>19 header.py</b>	<b>47</b>
<b>20 int32_msg.py</b>	<b>48</b>
<b>21 msg.py</b>	<b>49</b>
<b>22 string_msg.py</b>	<b>50</b>
<b>23 uint32_msg.py</b>	<b>51</b>
<b>24 __init__.py</b>	<b>52</b>
<b>25 listener_event_indexer.py</b>	<b>53</b>
25.1 Class: ListenerEventIndexer . . . . .	53
<b>26 listener_event_msg.py</b>	<b>55</b>
<b>27 dict_message.py</b>	<b>56</b>
27.1 Method: get_value . . . . .	56
<b>28 plugin_loader.py</b>	<b>57</b>
28.1 Class: ConfigValidator . . . . .	57
28.1.1 Method: validate . . . . .	57
28.2 Class: PluginLoader . . . . .	57
28.2.1 Method: get_serializer . . . . .	57
28.2.2 Method: get_exchange_handler . . . . .	57
28.2.3 Method: get_message . . . . .	58
28.2.4 Method: load_config . . . . .	58
<b>29 publisher.py</b>	<b>59</b>
29.1 Method: publish . . . . .	59
<b>30 qlogger.py</b>	<b>60</b>
30.1 Class: ColorFormatter . . . . .	60
30.1.1 Method: format . . . . .	60
30.2 Class: QFileHandler . . . . .	60
30.2.1 Method: get_log_path . . . . .	60
30.2.2 Method: get_config_supported . . . . .	61
30.3 Class: QDefaultFileHandler . . . . .	61
30.3.1 Method: get_log_path . . . . .	61
30.3.2 Method: get_config_supported . . . . .	61
30.4 Class: QConsoleHandler . . . . .	61
30.4.1 Method: get_config_supported . . . . .	62
30.5 Class: QLogger . . . . .	62
30.5.1 Method: get_logger . . . . .	62
30.5.2 Method: set_handler . . . . .	62
30.5.3 Method: get_handler . . . . .	62
<b>31 rendezvous.py</b>	<b>63</b>
31.1 Method: announce_ready . . . . .	63
31.2 Method: wait_for . . . . .	63

<b>32</b>	<b>base_serializer.py</b>	<b>64</b>
32.1	Method: serialize . . . . .	64
32.2	Method: deserialize . . . . .	64
<b>33</b>	<b>json_serializer.py</b>	<b>65</b>
33.1	Method: serialize . . . . .	65
33.2	Method: deserialize . . . . .	65
<b>34</b>	<b>pickle_serializer.py</b>	<b>67</b>
34.1	Method: serialize . . . . .	67
34.2	Method: deserialize . . . . .	67
<b>35</b>	<b>protobuf_serializer.py</b>	<b>68</b>
35.1	Method: serialize . . . . .	68
35.2	Method: deserialize . . . . .	68
<b>36</b>	<b>startup_policy.py</b>	<b>69</b>
36.1	Function: resolve_message_cls . . . . .	69
36.2	Function: build_policy_from_item . . . . .	69
36.3	Function: build_policy_from_config . . . . .	70
36.4	Class: StartupPolicy . . . . .	70
36.4.1	Method: wait_until_ready . . . . .	70
36.5	Class: NoWait . . . . .	70
36.5.1	Method: wait_until_ready . . . . .	70
36.6	Class: FixedDelay . . . . .	71
36.6.1	Method: wait_until_ready . . . . .	71
36.7	Class: HandshakeBarrier . . . . .	71
36.7.1	Method: wait_until_ready . . . . .	71
36.8	Class: PanelControlLegacyByAlias . . . . .	71
36.8.1	Method: wait_until_ready . . . . .	71
36.9	Class: GeneralCacheStartupPolicy . . . . .	71
36.9.1	Method: wait_until_ready . . . . .	71
36.10	Class: ConfigureUnroutablePolicy . . . . .	71
36.10.1	Method: before_setup . . . . .	71
36.10.2	Method: wait_until_ready . . . . .	72
36.11	Class: PolicyChain . . . . .	72
36.11.1	Method: wait_until_ready . . . . .	72
36.11.2	Method: before_setup . . . . .	72
36.11.3	Method: after_setup . . . . .	72
<b>37</b>	<b>subscriber.py</b>	<b>73</b>
37.1	Method: cache . . . . .	73
37.2	Method: start . . . . .	73
37.3	Method: stop . . . . .	73
37.4	Method: routing_key . . . . .	73
37.5	Method: callback . . . . .	73
<b>38</b>	<b>subscription_cache.py</b>	<b>74</b>

38.1	Method: <code>append</code>	74
38.2	Method: <code>get</code>	74
38.3	Method: <code>pop</code>	74
38.4	Method: <code>pop_nothrow</code>	75
38.5	Method: <code>peek_nothrow</code>	75
38.6	Method: <code>peek</code>	75
38.7	Method: <code>wait_for</code>	75
38.8	Method: <code>drain</code>	76
38.9	Method: <code>peek_last</code>	76
38.10	Method: <code>aget</code>	76
38.11	Method: <code>await_for</code>	76
38.12	Method: <code>wait_for_one</code>	77
38.13	Method: <code>wait_for_many</code>	77
<b>39</b>	<b><code>utils.py</code></b>	<b>78</b>
39.1	Class: <code>Singleton</code>	78
39.2	Class: <code>DictToClass</code>	78
39.2.1	Method: <code>validate</code>	78
39.3	Class: <code>Utils</code>	78
39.3.1	Method: <code>get_all_descendant_classes</code>	78
39.3.2	Method: <code>get_all_sub_classes</code>	79
39.3.3	Method: <code>is_valid_host</code>	79
39.3.4	Method: <code>caller_name</code>	79
39.3.5	Method: <code>load_library</code>	79
39.3.6	Method: <code>is_ascii_or_unicode</code>	79
39.4	Class: <code>Job</code>	80
39.4.1	Method: <code>stop</code>	80
39.4.2	Method: <code>run</code>	80
39.5	Class: <code>ResultType</code>	80
39.6	Class: <code>ResponseMessage</code>	80
39.6.1	Method: <code>get_json</code>	80
39.6.2	Method: <code>get_data</code>	80
39.6.3	Method: <code>create_from_string</code>	80
<b>40</b>	<b><code>wait_mode.py</code></b>	<b>81</b>
<b>41</b>	<b>Glossary</b>	<b>82</b>
<b>42</b>	<b>Appendix</b>	<b>83</b>
<b>43</b>	<b>History</b>	<b>84</b>

# Chapter 1

## Introduction

The component **EventBusClient** provides a modular, high-level messaging framework built on top of the message broker **RabbitMQ**. It abstracts low-level standard protocol operations and adds:

- Plugin support (custom serializers, handlers, message types)
- Dynamic configuration
- Reconnect and lifecycle management

It enables multiple projects to reuse a consistent event bus API without rewriting **RabbitMQ** logic.

The **EventBusClient** is part of a test automation framework called **RobotFramework AIO**, but can also be installed and used stand-alone.

**RabbitMQ** is an open-source message broker that enables different parts of an application or separate applications to communicate with each other by sending and receiving messages asynchronously. It acts as an intermediary between producers (which send messages) and consumers (which receive and process messages). **RabbitMQ** decouples the sender and receiver, allowing them to operate independently and at their own pace.

Messages are stored in queues until they can be processed, ensuring reliability even if a consumer is temporarily unavailable.

### RabbitMQ terms

Term	Description
Producer	Application or service that sends messages
Consumer	Application or service that receives and processes messages
Queue	Buffer that temporarily stores messages
Exchange	Routes messages from producers to queues based on routing rules
Binding	Link between an exchange and a queue, defining how messages are routed
Routing Key	Address-like string used to decide how messages are routed
AMQP	Advanced Message Queuing Protocol, the standard protocol <b>RabbitMQ</b> uses

For more detailed information about these components and frameworks please refer to the corresponding [homepages](#).



## Chapter 2

# Description

### 2.1 Overview

The **EventBusClient** provides a high-level API for interacting with a **RabbitMQ**-based event bus system. It abstracts low-level AMQP details and supports dynamic loading of project-specific plugins for serializers, exchange handlers, and message structures. This client is designed for modularity and ease of use across multiple projects.

### 2.2 Features

- **Dynamic plugin loading:** Load custom serializers, exchange handlers, and message classes at runtime from a configurable plugins directory.
- **Thread-safe publishing:** Supports safe publishing from multiple threads with minimal locking overhead.
- **Automatic reconnect:** Automatically reconnects to **RabbitMQ** after connection loss (if enabled in configuration).
- **Pluggable architecture:** Supports both built-in and project-specific extensions.
- **Multiple exchange types:** Supports Topic, Fanout, Headers, and X-RTopic exchange handlers.
- **Header-based routing:** Route messages based on header attributes with AND/OR logic using `HeadersExchangeHandler`.
- **Supports JSON, Protobuf, Pickle serializers.**

### 2.3 Architecture

The **EventBusClient** is built around a modular architecture that allows for easy extension and customization. It consists of the following components:

- **Base Classes:** Define interfaces for exchange handlers, serializers, and messages.
- **Plugins Directory:** Contains custom implementations of the base classes.
- **Configuration File:** Specifies the plugins path, **RabbitMQ** connection details, and selected implementations.
- **EventBusClient Class:** The main class that interacts with **RabbitMQ** and manages plugins.
- **Message Classes:** Define the structure of messages sent and received over the event bus.
- **Exchange Handlers:** Handle the declaration and publishing of messages to **RabbitMQ** exchanges.
- **Serializers:** Convert messages to and from byte streams for transmission over **RabbitMQ**.

The client uses a combination of built-in plugins and user-defined plugins to provide flexibility in message handling and serialization.

## 2.4 Installation

To install the **EventBusClient**, use pip to install the package from PyPI:

```
pip install eventbusclient
```

Ensure you have **RabbitMQ** server running and accessible at the specified host and port in the configuration file.

## 2.5 Dependencies

The **EventBusClient** requires the following dependencies:

- **pika**: For synchronous **RabbitMQ** communication
- **aio-pika**: Specifically for asynchronous **RabbitMQ** communication
- **protobuf**: For Protocol Buffers serialization (if using **ProtobufSerializer**)
- **jsonpickle**: For JSON serialization (if using **JSONSerializer**)

Ensure these dependencies are installed in your Python environment. You can install them using pip:

```
pip install pika protobuf jsonpickle
```

## 2.6 Directory Structure

```
project/
|-- EventBusClient/
|   |-- event_bus_client.py
|   |-- connection.py
|   |-- plugin_loader.py
|   |-- publisher.py
|   |-- subscriber.py
|   |-- qlogger.py
|   |-- message/
|   |   |-- base_message.py
|   |   |-- basic_message.py
|   |   |-- control_message.py
|   |-- exchange_handler/
|   |   |-- base.py
|   |   |-- topic_handler.py
|   |   |-- fanout_handler.py
|   |   |-- headers_handler.py
|   |   |-- x_rtopic_handler.py
|   |-- serializer/
|   |   |-- base_serializer.py
|   |   |-- json_serializer.py
|   |   |-- protobuf_serializer.py
|   |   |-- pickle_serializer.py
|   |-- plugins/
|   |   |-- custom_plugin/
|   |   |   |-- serializer/
|   |   |   |-- message/
|   |   |   |-- exchange_handler/
|   |-- config/
|   |   |-- config.jsonp
|-- app.py
```

## 2.7 Configuration

The `config.jsonp` file controls all aspects of the **EventBusClient**, including plugin paths, **RabbitMQ** connection, and selected implementations.

Configuration Template (EventBusClient/config/config.jsonp.template)

*Note: This is JSON with comments (.jsonp). Remove comments (// , /\* \*/) if your parser does not support them.*

```
{
  // plugins_path: Path to the directory containing plugin modules.
  "plugins_path": "./plugins",

  // host: Hostname or IP address for the server connection.
  "host": "localhost",

  // port: Port number for the server connection.
  "port": 5672,

  // serializer: Serialization method for message data (e.g., PickleSerializer).
  "serializer": "PickleSerializer",

  // exchange_handler: Handler type for message exchange (e.g., TopicExchangeHandler).
  "exchange_handler": "TopicExchangeHandler",

  // auto_reconnect: Automatically reconnect if the connection is lost (true/false).
  "auto_reconnect": true,

  // qos_prefetch: Number of messages to prefetch for Quality of Service.
  "qos_prefetch": 10,

  // general_cache_policy: "off" | "on_connect" | "on_demand".
  "general_cache_policy": "off",

  // general_routing_keys: Routing keys for general messages.
  "general_routing_keys": "general",

  // general_message_cls: Message class for general messages.
  "general_message_cls": "BaseMessage",

  // logger_name: Name of the logger instance.
  "logger_name": "test",

  // logfile: Path to the log file.
  "logfile": "test.log",

  // loglevel: Logging level for the logger instance.
  "loglevel": "INFO",

  // logger_mode: Log file mode ("w" for overwrite, "a" for append).
  "logger_mode": "a"
}
```

## How to Use

1. Copy the template above into your project as `config.jsonp`.
2. Adjust each value to match your environment and requirements.
3. Save the file and restart your application to apply changes.

## Field Descriptions

Field	Required	Type	Default	Description / Options
<code>plugins_path</code>	Optional	String	<code>./plugins</code>	Path to the plugins directory (relative or absolute).
<code>host</code>	Optional	String	<code>"localhost"</code>	Server hostname or IP address for the broker connection.
<code>port</code>	Optional	Integer	<code>5672</code>	Server port for the broker connection.
<code>serializer</code>	<b>Mandatory</b>	String	<code>(none)</code>	Serialization strategy for message bodies (e.g., <code>PickleSerializer</code> ). Must be available to both producer and consumer sides.
<code>exchange_handler</code>	<b>Mandatory</b>	String	<code>(none)</code>	Exchange handler implementation (e.g., <code>TopicExchangeHandler</code> ). Determines exchange type and binding logic.
<code>auto_reconnect</code>	Optional	Boolean	<code>true</code>	Reconnect automatically if the connection is lost.
<code>qos_prefetch</code>	Optional	Integer	<code>10</code>	Prefetch count for consumer QoS (how many unacked messages a consumer can receive).
<code>general_cache_policy</code>	Optional	String	<code>"off"</code>	Policy controlling the general message cache:  <code>"off"</code> – disabled;  <code>"on_connect"</code> – start caching upon connection establishment;  <code>"on_demand"</code> – start caching only when first requested.
<code>general_routing_keys</code>	Optional	String (or list)	<code>"general"</code>	Routing key or keys used for the general cache subscription (supports topic patterns).
<code>general_message_cls</code>	Optional	String	<code>"BaseMessage"</code>	Message class to use for general-cache decoding. Provide a resolvable name or dotted path if needed.
<code>logger_name</code>	Optional	String	<code>"event_bus"</code>	Name of the logger instance to use/create.
<code>logfile</code>	Optional	String / Null	<code>(none)</code>	Path to a log file. If omitted or null, logs go to the default handlers (e.g., console).
<code>loglevel</code>	Optional	String	<code>"INFO"</code>	Logging threshold ( <code>"DEBUG"</code> , <code>"INFO"</code> , <code>"WARNING"</code> , <code>"ERROR"</code> , <code>"CRITICAL"</code> ).
<code>logger_mode</code>	Optional	String	<code>"a"</code>	Log file mode: <code>"w"</code> (overwrite on each start) or <code>"a"</code> (append to existing).

## 2.8 Usage

### 2.8.1 Initialization

Create an **EventBusClient** instance from configuration:

```
from EventBusClient.event_bus_client import EventBusClient
```

```
client = await EventBusClient.from_config("./config/config.jsonp")
```

## 2.8.2 Publishing Messages

Send a message using the configured serializer and exchange handler asynchronously:

```
msg = ListenerEventMsg(event="start_cycle", timestamp=1234567890)
await client.send("zone1.topic.start", msg, header)
```

Enable thread-safe publish from a non-async thread:

```
client.send("zone1.topic.alert", msg, header, threadsafe=True)
```

Send a message synchronously using the corresponding method:

```
client.send_sync("zone1.topic.sync", msg, header)
```

## 2.8.3 Subscribing to Messages

Subscribe to a routing key and handle incoming messages:

```
async def on_message(msg: ListenerEventMsg, header):
    print(f"Received message: {msg}, header: {header}")

await client.on("zone1.#", ListenerEventMsg, on_message)
```

Subscribe to a routing key and handle incoming messages synchronously using `on_sync`:

```
def on_message_sync(msg: ListenerEventMsg, header):
    print(f"Received message: {msg}, header: {header}")

client.on_sync("zone1.#", ListenerEventMsg, on_message_sync)
```

## 2.8.4 Header-Based Subscriptions (HeadersExchangeHandler)

When using **HeadersExchangeHandler**, you can route messages based on header attributes instead of routing keys:

```
# Subscribe with AND logic (all headers must match)
await client.on(
    routing_key="", # Ignored in headers exchange
    message_cls=DocumentMessage,
    callback=on_document,
    binding_headers={"format": "pdf", "department": "engineering"},
    match_all=True # x-match=all (AND logic)
)

# Subscribe with OR logic (any header must match)
await client.on(
    routing_key="",
    message_cls=DocumentMessage,
    callback=on_important,
    binding_headers={"format": "pdf", "priority": "high"},
    match_all=False # x-match=any (OR logic)
)

# Publish with headers
await client.send(
    routing_key="", # Ignored in headers exchange
    message=DocumentMessage(title="Q4 Report"),
    headers={"format": "pdf", "department": "engineering", "author": "john"}
)
```

- `binding_headers`: Dictionary of headers to match for subscription.
- `match_all=True`: All specified headers must match (AND logic, `x-match=all`).
- `match_all=False`: At least one header must match (OR logic, `x-match=any`).

### 2.8.5 Setting Unroutable Message Policy

You can configure how the client handles unroutable messages by setting a startup policy. For example:

```
from EventBusClient.event_bus_client import EventBusClient
from EventBusClient.startup_policy import ConfigureUnroutablePolicy

config_unrout = ConfigureUnroutablePolicy(
    mode="return",                # "drop" (default), "alternate-exchange", or "return"
    alternate_exchange=None,      # Name of alternate exchange if mode is ↩
    ↪ "alternate-exchange"
    on_unroutable="log",          # "log" (default), "raise", "cache", or "callback"
    on_unroutable_callback=None   # User-defined callback function if on_unroutable is ↩
    ↪ "callback"
)
client = await EventBusClient.from_config(config_path, startup_policy=config_unrout)
```

- `mode`: (*optional, str, default: "drop"*)  
Determines how unroutable messages are handled: "drop" (discard), "alternate-exchange" (send to alternate exchange), or "return" (return to sender).
- `alternate_exchange`: (*optional, str or None, default: None*)  
The name of the alternate exchange to use if mode is "alternate-exchange".
- `on_unroutable`: (*optional, str, default: "log"*)  
The action to take on unroutable messages: "log" (log the event), "raise" (raise an exception), "cache" (store for later), or "callback" (invoke a user callback).
- `on_unroutable_callback`: (*optional, callable or None, default: None*)  
A user-defined function to call when an unroutable message is received (only if `on_unroutable` is set to "callback").

## 2.9 Example: Producer and Consumer

This example (from `examples/basic_async_producer_consumer_sample.py`) starts a producer and a consumer in separate processes:

```
# Start consumer
consumer = Process(target=run_process, args=(consumer_process, config_path))
consumer.start()

# Start producer
producer = Process(target=run_process, args=(producer_process, config_path))
producer.start()
# Wait for both to finish
producer.join()
consumer.join()
```

You can find the complete example in the `examples` directory. Run `run_sample.bat` (Windows) to execute the example.

## 2.10 Example: Headers Exchange

This example demonstrates header-based message routing using **HeadersExchangeHandler**:

```
from EventBusClient.event_bus_client import EventBusClient
from EventBusClient.exchange_handler.headers_handler import HeadersExchangeHandler
from EventBusClient.serializer.json_serializer import JsonSerializer

# Create client with HeadersExchangeHandler
handler = HeadersExchangeHandler(
    name="documents_exchange",
    serializer=JsonSerializer()
)
client = EventBusClient(exchange_handler=handler, host="localhost", port=5672)
await client.connect()

# Subscribe to PDF documents from engineering (AND logic)
async def handle_docs(msg, headers):
    print(f"Received: {msg} with headers {headers}")

await client.on(
    routing_key="",
    message_cls=DocumentMessage,
    callback=handle_docs,
    binding_headers={"format": "pdf", "department": "engineering"},
    match_all=True
)

# Publish with matching headers
await client.send(
    routing_key="",
    message=DocumentMessage(title="Q4 Report"),
    headers={"format": "pdf", "department": "engineering"}
)
```

See `examples/headers_exchange_sample.py` for the complete example.

## 2.11 Example: Custom Plugin Structure

To create a custom plugin, follow these steps:

- Create a directory for your plugin in the configured plugins path.
- Implement the required interfaces (e.g., `BaseExchangeHandler`, `BaseMessage`, `BaseSerializer`).

- Register your plugin in the `config.jsonp` file.
- Ensure your plugin is discoverable by the **EventBusClient**.
- Place your plugin code in the plugins directory, following the structure:
- For example, if your plugin is named `CustomPlugin`, the directory structure should look like this:

```
plugins/  
|-- CustomPlugin/  
    |-- __init__.py  
    |-- custom_exchange_handler.py  
    |-- custom_message.py  
    |-- custom_serializer.py
```

## 2.12 Example: Built-in Plugins

The **EventBusClient** comes with several built-in plugins that can be used directly or extended:

### Exchange Handlers

- **TopicExchangeHandler**: Routes messages using routing key patterns with wildcards (\* and #).
- **FanoutExchangeHandler**: Broadcasts messages to all bound queues (ignores routing key).
- **HeadersExchangeHandler**: Routes messages based on header attributes with AND/OR logic.
- **XRTopicExchangeHandler**: Reverse topic exchange (subscribers use literal keys, publishers use patterns).

### Message Classes

- **BasicMessage**: Simple message with content and headers.
- **ControlMessage**: Message for coordination and rendezvous patterns.

### Serializers

- **PickleSerializer**: Serializes messages using Python's Pickle module.
- **JSONSerializer**: Serializes messages to JSON format.
- **ProtobufSerializer**: Serializes messages using Protocol Buffers.

These plugins can be used as-is or extended to create custom functionality.



## 2.13 Example: Built-in Configuration

The built-in configuration file `config.jsonp` provides a starting point for configuring the **EventBusClient**:

```
{
  "plugins_path": "./plugins",
  "host": "localhost",
  "port": 5672,
  "serializer": "PickleSerializer",
  "exchange_handler": "XRTopicExchangeHandler",
  "message_class": "ListenerEventMsg",
  "threadsafe_publish": true,
  "auto_reconnect": true,
  "qos_prefetch": 10
}
```

This configuration specifies the plugins path, **RabbitMQ** connection details, serializer, exchange handler, message class, and other options.

## 2.14 Example: Custom Exchange Handler

To create a custom exchange handler, implement the required interface and place it in the plugins directory. For example, a custom exchange handler might look like this:

```
from event_bus.plugins import BaseExchangeHandler
class CustomExchangeHandler(BaseExchangeHandler):
    def declare_exchange(self, channel):
        # Custom exchange declaration logic
        pass

    def publish(self, channel, routing_key, message):
        # Custom publish logic
        pass

# Register the plugin in config.jsonp
{
  "plugins_path": "./plugins",
  "exchange_handler": "CustomExchangeHandler"
}
```

## 2.15 Example: Custom Message Class

To create a custom message class, define it in your project and register it in the configuration:

```
from event_bus.plugins import BaseMessage
class CustomMessage(BaseMessage):
    def __init__(self, data):
        self.data = data

    @classmethod
    def from_data(cls, data):
        pass

    @abstractmethod
    def get_value(self):
        pass

# Register the message class in config.jsonp
{
  "plugins_path": "./plugins",
  "message_class": "CustomMessage"
}
```

## 2.16 Example: Custom Serializer

To create a custom serializer, implement the required interface and place it in the plugins directory. For example, a custom serializer might look like this:

```
from event_bus.plugins import BaseSerializer
class CustomSerializer(BaseSerializer):
    def serialize(self, obj):
        # Custom serialization logic
        pass

    def deserialize(self, data):
        # Custom deserialization logic
        pass
# Register the plugin in config.jsonp
{
    "plugins_path": "./plugins",
    "serializer": "CustomSerializer"
}
```

## Chapter 3

`__init__.py`

## Chapter 4

# connection.py

ConnectionManager: Manages RabbitMQ connections, channels, and exchanges.

### 4.1 Method: `is_connected`

Check if the connection to RabbitMQ is established.

**Returns:**

*/ Type: bool /*

True if connected, False otherwise.

### 4.2 Method: `reset_loop`

### 4.3 Method: `connect`

Establish a robust connection to RabbitMQ and declare the exchange.

**Arguments:**

- `host`

*/ Condition: required / Type: str /*

The hostname or IP address of the RabbitMQ server.

- `port`

*/ Condition: required / Type: int /*

The port number on which the RabbitMQ server is listening.

- `prefetch_count`

*/ Condition: optional / Type: int /*

The number of messages to prefetch from the RabbitMQ server. Defaults to 10.

### 4.4 Method: `register_exchange_handler`

Register an exchange handler to handle messages from the exchange.

**Arguments:**

- `handler`

*/ Condition: required / Type: ExchangeHandler /*

The exchange handler to register. It should be an instance of `ExchangeHandler` or its subclasses.

## 4.5 Method: unregister\_exchange\_handler

Unregister an exchange handler.

### Arguments:

- `handler`  
/ *Condition*: required / *Type*: `ExchangeHandler` /  
The exchange handler to unregister. It should be an instance of `ExchangeHandler` or its subclasses.

## 4.6 Method: get\_channel

Get the current channel for publishing messages.

### Returns:

/ *Type*: `aio_pika.Channel` | `None` /  
Channel instance or `None` if not available.

## 4.7 Method: close

Gracefully close the connection and channel.

## 4.8 Method: recreate\_channel

Recreate the RabbitMQ channel if it is closed or dropped.

### Arguments:

- `exc`  
/ *Condition*: optional / *Type*: `Exception` /  
The exception that caused the channel to drop, if any.
- `reply_code`  
/ *Condition*: optional / *Type*: `int` /  
The reply code associated with the channel drop, if any.

## 4.9 Method: reconnect

Reconnect to RabbitMQ in case of connection loss or error.

### Arguments:

- `host`  
/ *Condition*: required / *Type*: `str` /  
The hostname or IP address of the RabbitMQ server.
- `port`  
/ *Condition*: required / *Type*: `int` /  
The port number on which the RabbitMQ server is listening.
- `exc`  
/ *Condition*: optional / *Type*: `Exception` /  
The exception that caused the reconnection attempt, if any. If not provided, it defaults to `None`.

## Chapter 5

### constants.py

5.1 Class: SocketType

5.2 Class: String

## Chapter 6

# event\_bus\_client.py

### 6.1 Function: get\_running\_loop\_or\_none

### 6.2 Function: is\_on\_loop

### 6.3 Function: main

### 6.4 Class: EventBusClient

EventBusClient: Client for interacting with the event bus system.

#### 6.4.1 Method: getVersion

Returns the version of EventBusClient as string.

#### 6.4.2 Method: getVersionDate

Returns the version date of EventBusClient as string.

#### 6.4.3 Method: enable\_general\_cache

Enable the general cache with the specified routing keys and message class.

**Arguments:**

- routing\_keys  
/ *Condition*: required / *Type*: Union[str, list] /  
The routing keys to subscribe to for the general cache. Can be a single string or a list of strings.
- message\_cls  
/ *Condition*: required / *Type*: Type[BaseMessage] /  
The class of messages to subscribe to for the general cache. Must be a subclass of BaseMessage.

#### 6.4.4 Method: wait\_on\_general\_topic\_for\_one

Wait for a specific message on the general topic.

**Arguments:**

- msg  
/ *Condition*: required / *Type*: Any /  
The message to wait for. This can be any object that can be compared for equality.

- `timeout`

/ *Condition*: optional / *Type*: float / *Default*: 30.0 /

The maximum time to wait for the message, in seconds. Defaults to 30.0 seconds.

- `interval`

/ *Condition*: optional / *Type*: float / *Default*: 0.1 /

**Note:** This parameter is accepted for compatibility with older interfaces but is not used in the current implementation.

- `asynchronous`

/ *Condition*: optional / *Type*: bool / *Default*: False /

If True, the wait will be performed asynchronously using a `ThreadPoolExecutor`. Defaults to False.

#### Returns:

/ *Type*: bool /

True if the message was received within the timeout period, False otherwise.

### 6.4.5 Method: `wait_on_general_topic_for_many`

Wait for multiple specific messages on the general topic.

#### Arguments:

- `msgs`

/ *Condition*: required / *Type*: List[Any] /

The list of messages to wait for. Each message can be any object that can be compared for equality.

- `mode`

/ *Condition*: optional / *Type*: int / *Default*: `WaitMode.ALL_IN_GIVEN_ORDER.value` /

The mode for waiting:

- `WaitMode.ALL_IN_GIVEN_ORDER`: Wait for all messages in the order they are provided.
- `WaitMode.ALL_IN_RANDOM_ORDER`: Wait for all messages in any order.
- `WaitMode.ANY`: Wait for any one of the messages.

Defaults to `WaitMode.ALL_IN_GIVEN_ORDER.value`.

- `timeout`

/ *Condition*: optional / *Type*: float / *Default*: 30.0 /

The maximum time to wait for the messages, in seconds. Defaults to 30.0 seconds.

- `interval`

/ *Condition*: optional / *Type*: float / *Default*: 0.1 /

**Note:** This parameter is accepted for compatibility with older interfaces but is not used in the current implementation.

- `asynchronous`

/ *Condition*: optional / *Type*: bool / *Default*: False /

If True, the wait will be performed asynchronously using a `ThreadPoolExecutor`. Defaults to False.

#### Returns:

/ *Type*: List[int] /

A list of indices of the messages that were received, based on the specified mode.



### 6.4.6 Method: wait\_on\_cache\_for\_one

Wait for a specific message in the given subscription cache.

#### Arguments:

- `cache`  
*/ Condition:* required */ Type:* SubscriptionCache */*  
 The subscription cache to wait on. This should be an instance of SubscriptionCache.
- `msg`  
*/ Condition:* required */ Type:* Any */*  
 The message to wait for. This can be any object that can be compared for equality.
- `timeout`  
*/ Condition:* optional */ Type:* float */ Default:* 30.0 */*  
 The maximum time to wait for the message, in seconds. Defaults to 30.0 seconds.
- `interval`  
*/ Condition:* optional */ Type:* float */ Default:* 0.1 */*  
**Note:** This parameter is accepted for compatibility with older interfaces but is not used in the current implementation.
- `asynchronous`  
*/ Condition:* optional */ Type:* bool */ Default:* False */*  
 If True, the wait will be performed asynchronously using a ThreadPoolExecutor. Defaults to False.
- `dropped_msgs`  
*/ Condition:* optional */ Type:* List[Any] */ Default:* None */*  
 The list of messages having been discarded. Defaults to None.

#### Returns:

*/ Type:* bool */*  
 True if the message was received within the timeout period, False otherwise.

### 6.4.7 Method: wait\_on\_cache\_for\_many

Wait for multiple specific messages in the given subscription cache.

#### Arguments:

- `cache`  
*/ Condition:* required */ Type:* SubscriptionCache */*  
 The subscription cache to wait on. This should be an instance of SubscriptionCache.
- `msgs`  
*/ Condition:* required */ Type:* List[Any] */*  
 The list of messages to wait for. Each message can be any object that can be compared for equality.
- `mode`  
*/ Condition:* optional */ Type:* int */ Default:* WaitMode.ALL\_IN\_GIVEN\_ORDER.value */*  
 The mode for waiting:
  - WaitMode.ALL\_IN\_GIVEN\_ORDER: Wait for all messages in the order they are provided.
  - WaitMode.ALL\_IN\_RANDOM\_ORDER: Wait for all messages in any order.
  - WaitMode.ANY: Wait for any one of the messages.

Defaults to WaitMode.ALL\_IN\_GIVEN\_ORDER.value.

- `timeout`  
/ *Condition*: optional / *Type*: float / *Default*: 30.0 /  
The maximum time to wait for the messages, in seconds. Defaults to 30.0 seconds.
- `interval`  
/ *Condition*: optional / *Type*: float / *Default*: 0.1 /  
**Note:** This parameter is accepted for compatibility with older interfaces but is not used in the current implementation.
- `asynchronous`  
/ *Condition*: optional / *Type*: bool / *Default*: False /  
If True, the wait will be performed asynchronously using a `ThreadPoolExecutor`. Defaults to False.
- `dropped_msgs`  
/ *Condition*: optional / *Type*: List[Any] / *Default*: None /  
The list of messages having been discarded. Defaults to None.

**Returns:**

/ *Type*: List[int] /  
A list of indices of the messages that were received, based on the specified mode.

**6.4.8 Method: constructor\_params\_from\_config**

Returns a tuple of parameters for the `EventBusClient` constructor, loaded from config and provided arguments.

**Arguments:**

- `config_path`  
/ *Condition*: required / *Type*: str /  
Path to the configuration file in JSONP format. This file should contain the necessary configuration for the event bus client, including exchange handler and serializer settings.
- `startup_policy`  
/ *Condition*: optional / *Type*: StartupPolicy / *Default*: None /  
The startup policy to use for the client. If not provided, no startup policy will be used.
- `zone_id`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
The zone ID for the client. If not provided, no zone ID will be used.
- `alias`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
The alias for the client. If not provided, no alias will be used.
- `startup_policies`  
/ *Condition*: optional / *Type*: list[StartupPolicy] / *Default*: None /  
A list of startup policies to use for the client. If provided, these will be combined into a `PolicyChain`.

**Returns:**

/ *Type*: dict /  
A dictionary of parameters to be used for the `EventBusClient` constructor.

### 6.4.9 Method: from\_config

Create an EventBusClient instance from a configuration file.

#### Arguments:

- `config_path`  
*/ Condition: required / Type: str /*  
 Path to the configuration file in JSONP format. This file should contain the necessary configuration for the event bus client, including exchange handler and serializer settings.
- `config_source`  
*/ Condition: required / Type: Union[str, dict] /*  
 Configuration source as a JSON string or dictionary. This can be used instead of `config_path` to provide configuration directly.
- `startup_policy`  
*/ Condition: optional / Type: StartupPolicy / Default: None /*  
 The startup policy to use for the client. If not provided, no startup policy will be used.
- `zone_id`  
*/ Condition: optional / Type: str / Default: None /*  
 The zone ID for the client. If not provided, no zone ID will be used.
- `alias`  
*/ Condition: optional / Type: str / Default: None /*  
 The alias for the client. If not provided, no alias will be used.
- `start_connection`  
*/ Condition: optional / Type: bool / Default: True /*  
 If True, the client will automatically connect to the event bus server after creation. Defaults to True.
- `startup_policies`  
*/ Condition: optional / Type: list[StartupPolicy] / Default: None /*  
 A list of startup policies to use for the client. If provided, these will be combined into a PolicyChain.

#### Returns:

*/ Type: EventBusClient /*  
 An instance of EventBusClient configured according to the provided configuration file or source.

### 6.4.10 Method: connect

Connect to the event bus server and set up the exchange handler.

#### Arguments:

- `host`  
*/ Condition: optional / Type: str / Default: None /*  
 Hostname of the event bus server. Defaults to None.
- `port`  
*/ Condition: optional / Type: int / Default: None /*  
 Port number of the event bus server. Defaults to None.
- `prefetch_count`  
*/ Condition: optional / Type: int / Default: None /*  
 The number of messages to prefetch from the server. This controls how many messages can be sent to the client before they are acknowledged. Defaults to None.

### 6.4.11 Method: send

Send a message to the event bus with the specified routing key.

**Arguments:**

- `routing_key`  
*/ Condition:* required */ Type:* Union[str, list] */*  
 The routing key used to route the message to the appropriate subscribers. It can be a single string or a list of strings.
- `message`  
*/ Condition:* required */ Type:* BaseMessage */*  
 The message to be sent. It should be an instance of BaseMessage or its subclasses.
- `headers`  
*/ Condition:* optional */ Type:* dict */*  
 Additional headers to include with the message. This can be used for metadata or routing information.
- `threadsafe`  
*/ Condition:* optional */ Type:* bool */*  
 If True, the message will be sent in a threadsafe manner. Defaults to False.

### 6.4.12 Method: off

Unsubscribe from messages with the specified routing key.

**Arguments:**

- `routing_key`  
*/ Condition:* required */ Type:* Union[str, list] */*  
 The routing key to unsubscribe from. It can be a single string or a list of strings.
- `callback`  
*/ Condition:* optional */ Type:* Callable[[BaseMessage], Awaitable[None]] */ Default:* None */*  
 The callback function to remove from the subscriber list. If not provided, all callbacks for the routing key will be unsubscribed.

### 6.4.13 Method: on

Subscribe to messages with the specified routing key and message class.

**Arguments:**

- `routing_key`  
*/ Condition:* required */ Type:* str */*  
 The routing key to subscribe to. Messages with this routing key will be routed to the callback. For HeadersExchangeHandler, this can be empty string as routing is based on headers.
- `message_cls`  
*/ Condition:* required */ Type:* Type[BaseMessage] */*  
 The class of the message to subscribe to. The callback will receive messages of this type.
- `callback`  
*/ Condition:* optional */ Type:* Callable[[BaseMessage, dict], Awaitable[None]] */*  
 The callback function to be called when a message is received. It should accept a single argument of type BaseMessage or its subclasses and return an awaitable (e.g., a coroutine).

- `cache_size`  
/ *Condition*: optional / *Type*: int /  
Maximum size of the subscription cache.
- `binding_headers`  
/ *Condition*: optional / *Type*: dict / *Default*: None /  
Headers to match for this subscription (only used with HeadersExchangeHandler). Example: {"format": "pdf", "type": "report"}
- `match_all`  
/ *Condition*: optional / *Type*: bool / *Default*: True /  
Only used when `binding_headers` is provided. If True, all specified headers must match (x-match=all, AND logic). If False, at least one header must match (x-match=any, OR logic).

**Returns:**

/ *Type*: SubscriptionCache /  
A cache object for accessing received messages.

**Example (Topic Exchange):**

```
cache = await client.on("sensor.temperature", TempMessage, handler)
```

**Example (Headers Exchange):**

```
cache = await client.on(
    routing_key="", # ignored for headers exchange
    message_cls=ReportMessage,
    callback=handler,
    binding_headers={"format": "pdf", "department": "engineering"},
    match_all=True # AND logic
)
```

**6.4.14 Method: wait\_until\_ready**

Convenience wrapper over `rendezvous.wait_for`.

**Arguments:** \* `requirements`

/ *Condition*: required / *Type*: dict[str, int] /  
A dictionary where keys are role names and values are the number of instances required for each role.

- `timeout`  
/ *Condition*: optional / *Type*: float /  
The maximum time to wait for the rendezvous to be satisfied. Defaults to 5.0 seconds.

**6.4.15 Method: announce\_ready**

Convenience wrapper over `rendezvous.announce_ready`.

**Arguments:**

- `roles`  
/ *Condition*: required / *Type*: list[str] /  
A list of role names that this instance is ready for. This is used to signal readiness in the rendezvous system.

**6.4.16 Method: close**

Close the connection to the event bus and clean up resources.

### 6.4.17 Method: `is_connected`

Check if the client is currently connected to the event bus.

**Returns:**

*/ Type: bool /*

True if the client is connected, False otherwise.

### 6.4.18 Method: `build_routing`

Backward-compatible alias for `build_routing_key`.

**Arguments:**

- `path`

*/ Condition: required / Type: str /*

The components of the routing key. Each component will be joined with a dot (.) to form the final routing key.

**Returns:**

*/ Type: str /*

The constructed routing key as a string.

### 6.4.19 Method: `build_routing_key`

Build a routing key from the given path components.

**Arguments:**

- `path`

*/ Condition: required / Type: str /*

The components of the routing key. Each component will be joined with a dot (.) to form the final routing key.

**Returns:**

- `str`

*/ Type: str /*

The constructed routing key as a string.

### 6.4.20 Method: `start_background_loop`

Start a dedicated asyncio loop in a background thread if not already running. Safe to call multiple times.

This method is useful for blocking synchronous APIs that need to run in a separate thread to avoid blocking the main thread, especially in environments where the main thread is already running an event loop (e.g., GUI applications, web servers).

This method will create a new thread that runs an asyncio event loop, allowing you to submit coroutines for execution without blocking the main thread. It also ensures that the loop is ready before returning. It will not start a new loop if one is already running in the background.

**Arguments:**

- `loop_name`

*/ Condition: optional / Type: str / Default: "EventBusClientLoop" /*

The name of the background thread running the asyncio loop. Defaults to "EventBusClientLoop".

#### 6.4.21 Method: `kill_aiormq_tasks_now`

#### 6.4.22 Method: `stop_background_loop`

Stop the background loop (if we created it) and join the thread.

This method is useful for cleaning up resources when the client is no longer needed.

**Arguments:**

- `timeout`  
/ *Condition*: optional / *Type*: float / *Default*: 3.0 /  
The maximum time to wait for the background loop to stop. Defaults to 3.0 seconds.

#### 6.4.23 Method: `from_config_sync`

Create an `EventBusClient` instance from a configuration file synchronously.

**Arguments:**

- `path`  
/ *Condition*: required / *Type*: str /  
Path to the configuration file in JSONP format. This file should contain the necessary configuration for the event bus client, including exchange handler and serializer settings.

**Returns:**

/ *Type*: `EventBusClient` /  
An instance of `EventBusClient` initialized with the configuration from the specified path.

#### 6.4.24 Method: `connect_sync`

Blocking connect that spins a background loop if needed.

**Arguments:**

- `host`  
/ *Condition*: optional / *Type*: str / *Default*: "localhost" /  
The hostname of the event bus server. Defaults to "localhost".
- `port`  
/ *Condition*: optional / *Type*: int / *Default*: 5672 /  
The port number of the event bus server. Defaults to 5672.
- `prefetch_count`  
/ *Condition*: optional / *Type*: int / *Default*: 10 /  
The number of messages to prefetch from the server. This controls how many messages can be sent to the client before they are acknowledged. Defaults to 10.
- `timeout`  
/ *Condition*: optional / *Type*: float / *Default*: 30.0 /  
The maximum time to wait for the connection to be established. Defaults to 30.0 seconds.

#### 6.4.25 Method: `send_sync_compat`

Backward-compatible blocking send wrapper.

### 6.4.26 Method: send\_sync

Blocking send wrapper.

#### Arguments:

- `routing_key`  
/ *Condition*: required / *Type*: str /  
The routing key used to route the message to the appropriate subscribers.
- `message`  
/ *Condition*: required / *Type*: BaseMessage /  
The message to be sent. It should be an instance of BaseMessage or its subclasses.
- `headers`  
/ *Condition*: optional / *Type*: dict | None /  
Additional headers to include with the message. This can be used for metadata or routing information.
- `threadsafe`  
/ *Condition*: optional / *Type*: bool / *Default*: False /  
If True, the message will be sent in a threadsafe manner. Defaults to False.

#### Returns:

/ *Type*: None /  
This method does not return any value. It sends the message to the event bus and returns immediately.

### 6.4.27 Method: on\_sync

Blocking subscribe wrapper. Returns SubscriptionCache to use with `get()/wait_for()/drain()`.

#### Arguments:

- `routing_key`  
/ *Condition*: required / *Type*: str /  
The routing key to subscribe to. Messages with this routing key will be routed to the callback. For HeadersExchangeHandler, this can be empty string as routing is based on headers.
- `message_cls`  
/ *Condition*: required / *Type*: Type[BaseMessage] /  
The class of the message to subscribe to. The callback will receive messages of this type.
- `callback`  
/ *Condition*: optional / *Type*: Callable[[BaseMessage], Awaitable[None]] /  
The callback function to be called when a message is received. It should accept a single argument of type BaseMessage or its subclasses and return an awaitable (e.g., a coroutine).
- `cache_size`  
/ *Condition*: optional / *Type*: int / *Default*: 200 /  
The size of the cache for storing received messages. This is useful for buffering messages before they are processed by the callback.
- `binding_headers`  
/ *Condition*: optional / *Type*: dict / *Default*: None /  
Headers to match for this subscription (only used with HeadersExchangeHandler). Example: {"format": "pdf", "type": "report"}



- `match_all`

*/ Condition: optional / Type: bool / Default: True /*

Only used when `binding_headers` is provided. If True, all specified headers must match (AND logic). If False, at least one header must match (OR logic).

#### Returns:

*/ Type: SubscriptionCache /*

A SubscriptionCache object that allows you to manage the subscription and access received messages.

#### Example (Topic Exchange):

```
cache = client.on_sync("sensor.temperature", TempMessage, handler)
```

#### Example (Headers Exchange):

```
cache = client.on_sync(
    routing_key="",
    message_cls=ReportMessage,
    binding_headers={"format": "pdf", "department": "engineering"},
    match_all=True
)
```

### 6.4.28 Method: `off_sync`

Blocking unsubscribe wrapper.

#### Arguments:

- `routing_key`

*/ Condition: required / Type: str /*

The routing key to unsubscribe from. Messages with this routing key will no longer be routed to the callback.

#### Returns:

*/ Type: None /*

This method does not return any value. It unsubscribes from the specified routing key and returns immediately.

### 6.4.29 Method: `wait_until_ready_sync`

Blocking rendezvous wait. Returns True if satisfied before timeout.

#### Arguments:

- `requirements`

*/ Condition: required / Type: dict[str, int] /*

A dictionary where keys are role names and values are the number of instances required for each role.

- `timeout`

*/ Condition: optional / Type: float / Default: 5.0 /*

The maximum time to wait for the rendezvous to be satisfied. Defaults to 5.0 seconds.

#### Returns:

*/ Type: bool /*

True if the rendezvous requirements are satisfied before the timeout, False otherwise.

### 6.4.30 Method: announce\_ready\_sync

Blocking announce ready via rendezvous control topic.

**Arguments:**

- `roles`

/ *Condition*: required / *Type*: list[str] /

A list of role names that this instance is ready for. This is used to signal readiness in the rendezvous system.

**Returns:**

/ *Type*: None /

This method does not return any value. It announces that the instance is ready for the specified roles.

### 6.4.31 Method: close\_sync

Optional: call your async close/teardown then stop the loop. If you don't have an async *close()*, this just stops the loop.

**Arguments:**

- `timeout`

/ *Condition*: optional / *Type*: float / *Default*: 10.0 /

The maximum time to wait for the close operation to complete. Defaults to 10.0 seconds.

### 6.4.32 Method: pop\_unroutables

Get and clear the list of unroutable messages.

**Returns:**

/ *Type*: list[dict] /

A list of unroutable message dictionaries that were cached.

# Chapter 7

## base.py

### 7.1 Method: `configure_unroutable`

Configure the unroutable message handling policy.

**Arguments:**

- `policy`  
/ *Condition*: required / *Type*: str /  
The unroutable message handling policy. Options are "drop", "log", "return", "alternate-exchange".
- `alternate_exchange`  
/ *Condition*: optional / *Type*: str /  
The name of the alternate exchange to use if the policy is "alternate-exchange".
- `on_unroutable`  
/ *Condition*: required / *Type*: str /  
The action to take for unroutable messages. Options are "log", "cache", "callback", "raise".
- `unroutable_sink`  
/ *Condition*: optional / *Type*: list /  
The sink (list) to store unroutable messages if the action is "cache".
- `on_unroutable_callback`  
/ *Condition*: optional / *Type*: Callable /  
The callback function to invoke for unroutable messages if the action is "callback".

### 7.2 Method: `reset_loop`

Reset the event loop used by the exchange handler.

**Arguments:**

- `loop`  
/ *Condition*: optional / *Type*: `asyncio.AbstractEventLoop` /  
The new event loop to use. If not provided, the current event loop will be used.

### 7.3 Method: `setup`

Set up the exchange handler by establishing a channel and declaring the exchange.

**Arguments:**

- `connection_manager`

/ *Condition*: required / *Type*: `ConnectionManager` /

The connection manager used to get the channel and exchange for publishing messages.

## 7.4 Method: with\_\_alternate\_\_exchange

Decorator to set up and finalize alternate exchange configuration around the decorated method.

**Arguments:**

- `func`

/ *Condition*: required / *Type*: `Callable` /

The method to be decorated.

## 7.5 Method: setup\_\_alternate\_\_exchange

Set up the alternate exchange configuration based on the unroutable policy.

## 7.6 Method: finalize\_\_setup

Finalize the setup of the exchange handler by configuring the alternate exchange and return handler.

## 7.7 Method: teardown

Tear down the exchange handler by closing the channel and cleaning up resources.

## 7.8 Method: publish

## 7.9 Method: subscribe

## 7.10 Method: unsubscribe

Unsubscribe a callback from a specific routing key.

**Arguments:**

- `routing_key`

/ *Condition*: required / *Type*: `str` /

The routing key to unsubscribe from.

- `callback`

/ *Condition*: required / *Type*: `Callable` /

The callback function to remove from the subscriber list.

## 7.11 Method: handle\_\_channel\_\_close

Handle channel closure by attempting to re-create the channel.

**Arguments:**

- `exc`

/ *Condition*: optional / *Type*: `Exception` /

The exception that caused the channel to close, if any. If not provided, it defaults to `None`.

## Chapter 8

# fanout\_handler.py

### 8.1 Method: setup

Set up the exchange handler by establishing a channel and declaring the exchange. This method is called by the EventBusClient during initialization. It initializes the channel and exchange, and prepares the publisher for sending messages.

**Arguments:**

- `connection_manager`  
/ *Condition*: required / *Type*: ConnectionManager /  
The connection manager used to get the channel and exchange for publishing messages.

### 8.2 Method: publish

Publish a message to the exchange with the specified routing key.

**Arguments:**

- `message`  
/ *Condition*: required / *Type*: BaseMessage /  
The message to be published. It should be an instance of BaseMessage or its subclasses.
- `routing_key`  
/ *Condition*: required / *Type*: str /  
The routing key used to route the message to the appropriate subscribers.
- `headers`  
  
/ *Condition*: optional / *Type*: dict /  
Additional headers to include with the message. This can be used for metadata or routing information.
- `threadsafe`  
/ *Condition*: optional / *Type*: bool / *Default*: False /  
If True, the publish operation will be executed in a thread-safe manner, allowing it to be called from non-async contexts.
- `mandatory`  
/ *Condition*: optional / *Type*: bool / *Default*: False /  
If True, the message must be routed to at least one queue. If it cannot be routed, it will be returned to the publisher.

## 8.3 Method: subscribe

Subscribe to messages on the exchange with the specified routing key.

### Arguments:

- `routing_key`  
/ *Condition*: required / *Type*: str /  
The routing key used to filter messages for this subscriber.
- `message_cls`  
/ *Condition*: required / *Type*: Type[BaseMessage] /  
The class of the message that this subscriber will handle. It should be a subclass of BaseMessage.
- `callback`  
/ *Condition*: required / *Type*: Callable[[BaseMessage], None] /  
The callback function that will be called when a message matching the routing key is received.

## Chapter 9

# headers\_handler.py

HeadersExchangeHandler: Handles headers-based message exchanges.

Unlike topic or direct exchanges that route based on routing keys, the headers exchange routes messages based on header attributes. This allows for more complex routing logic based on multiple criteria.

### Binding Arguments:

When subscribing, you can specify header matching criteria:

- `x-match`: Matching mode - `all`: All specified headers must match (AND logic) - `any`: At least one header must match (OR logic)
- Other key-value pairs: Headers to match against

### Example:

```
# Subscribe to messages where format=pdf AND type=report
await handler.subscribe(
    binding_headers={"x-match": "all", "format": "pdf", "type": "report"},
    message_cls=ReportMessage,
    callback=process_report
)

# Publish with matching headers
await handler.publish(
    message=report,
    routing_key="", # Ignored for headers exchange
    headers={"format": "pdf", "type": "report", "author": "john"}
```

## 9.1 Method: setup

Set up the exchange handler by establishing a channel and declaring the headers exchange.

### Arguments:

- `connection_manager`  
/ *Condition*: required / *Type*: `ConnectionManager` /  
The connection manager used to get the channel and exchange for publishing messages.

## 9.2 Method: publish

Publish a message to the headers exchange.

Note: For headers exchanges, the `routing_key` is typically ignored. Routing is determined by the headers parameter matching subscriber bindings.

### Arguments:

- `message`  
/ *Condition*: required / *Type*: `BaseMessage` /  
The message to be published.
- `routing_key`  
/ *Condition*: optional / *Type*: `str` /  
The routing key (typically ignored for headers exchange, can be empty string).
- `headers`  
/ *Condition*: optional / *Type*: `dict` /  
Headers used for routing. These are matched against subscriber binding headers. This is the primary routing mechanism for headers exchanges.
- `threadsafe`  
/ *Condition*: optional / *Type*: `bool` /  
If True, the publish operation will be thread-safe.
- `mandatory`  
/ *Condition*: optional / *Type*: `bool` /  
If True, the message will be returned if it cannot be routed.

## 9.3 Method: subscribe

Subscribe to messages on the headers exchange based on header matching.

### Arguments:

- `routing_key`  
/ *Condition*: optional / *Type*: `str` /  
The routing key (typically ignored for headers exchange, can be empty string).
- `message_cls`  
/ *Condition*: required / *Type*: `Type[BaseMessage]` /  
The class of the message that this subscriber will handle.
- `callback`  
/ *Condition*: optional / *Type*: `Callable[[BaseMessage, Dict], None]` /  
The callback function called when a matching message is received.
- `cache_size`  
/ *Condition*: optional / *Type*: `int` /  
Maximum size of the subscription cache.
- `binding_headers`  
/ *Condition*: optional / *Type*: `Dict[str, Any]` /  
Headers to match for this subscription. Should include: - `x-match`: "all" (all headers must match) or "any" (at least one must match) - Other key-value pairs to match against message headers  
Example: `{"x-match": "all", "format": "pdf", "priority": "high"}`

### Returns:

/ *Type*: `SubscriptionCache` /  
A cache object for accessing received messages synchronously.



## 9.4 Method: subscribe\_with\_headers

Convenience method to subscribe with header-based routing.

### Arguments:

- `binding_headers`  
/ *Condition*: required / *Type*: Dict[str, Any] /  
Headers to match (without x-match key, which is set by `match_all` parameter).
- `message_cls`  
/ *Condition*: required / *Type*: Type[BaseMessage] /  
The class of the message that this subscriber will handle.
- `callback`  
/ *Condition*: optional / *Type*: Callable /  
The callback function called when a matching message is received.
- `cache_size`  
/ *Condition*: optional / *Type*: int /  
Maximum size of the subscription cache.
- `match_all`  
/ *Condition*: optional / *Type*: bool / *Default*: True /  
If True, all headers must match (x-match=all). If False, any header can match (x-match=any).

### Returns:

/ *Type*: SubscriptionCache /  
A cache object for accessing received messages synchronously.

## Chapter 10

# topic\_handler.py

### 10.1 Method: setup

Set up the exchange handler by establishing a channel and declaring the exchange. This method is called by the EventBusClient during initialization. It initializes the channel and exchange, and prepares the publisher for sending messages.

**Arguments:**

- `connection_manager`  
/ *Condition*: required / *Type*: ConnectionManager /  
The connection manager used to get the channel and exchange for publishing messages.

### 10.2 Method: publish

Publish a message to the exchange with the specified routing key.

**Arguments:**

- `message`  
/ *Condition*: required / *Type*: BaseMessage /  
The message to be published. It should be an instance of BaseMessage or its subclasses.
- `routing_key`  
/ *Condition*: required / *Type*: str /  
The routing key used to route the message to the appropriate subscribers.
- `headers`  
/ *Condition*: optional / *Type*: dict /  
Additional headers to include with the message. This can be used for metadata or routing information.

### 10.3 Method: subscribe

Subscribe to messages on the exchange with the specified routing key.

**Arguments:**

- `routing_key`  
/ *Condition*: required / *Type*: str /  
The routing key used to filter messages for this subscriber.

- `message_cls`

/ *Condition*: required / *Type*: `Type[BaseMessage]` /

The class of the message that this subscriber will handle. It should be a subclass of `BaseMessage`.

- `callback`

/ *Condition*: required / *Type*: `Callable[[BaseMessage], None]` /

The callback function that will be called when a message matching the routing key is received.

# Chapter 11

## x\_rtopic\_handler.py

XRTopicExchangeHandler: Handles x-rtopic exchanges for routing messages based on topics.

### 11.1 Method: setup

Set up the exchange handler by establishing a channel and declaring the x-rtopic exchange.

**Arguments:**

- `connection_manager`  
/ *Condition*: required / *Type*: `ConnectionManager` /  
The connection manager used to get the channel and exchange for publishing messages.

### 11.2 Method: publish

Publish a message to the exchange with the specified routing key.

**Arguments:**

- `message`  
/ *Condition*: required / *Type*: `BaseMessage` /  
The message to be published. It should be an instance of `BaseMessage` or its subclasses.
- `routing_key`  
/ *Condition*: required / *Type*: `str` /  
The routing key used to route the message to the appropriate subscribers.
- `headers`  
/ *Condition*: optional / *Type*: `dict` /  
Additional headers to include with the message. This can be used for metadata or routing information.
- `threadsafe`  
/ *Condition*: optional / *Type*: `bool` /  
If True, the publish operation will be thread-safe, allowing it to be called from different threads.

This is useful in multi-threaded applications where the event bus client may be accessed from multiple threads.

### 11.3 Method: subscribe

Subscribe to messages on the exchange with the specified routing key.

**Arguments:**

- `routing_key`  
/ *Condition*: required / *Type*: str /  
The routing key used to filter messages for this subscription.
- `message_cls`  
/ *Condition*: required / *Type*: Type[BaseMessage] /  
The class of the message to be received. The subscriber will only process messages of this type.
- `callback`  
/ *Condition*: required / *Type*: Callable[[BaseMessage], None] /  
The callback function to be called when a message is received. It should accept a single argument of type *BaseMessage*.

This function will be called with the deserialized message object when a message matching the routing key is received.

## Chapter 12

# base\_message.py

BaseMessage: Abstract base class for messages in the event bus system.

### 12.1 Method: `from_data`

Create a message instance from raw data.

**Arguments:**

- `data`  
/ *Condition*: required / *Type*: Any /  
Raw data to create the message instance.

### 12.2 Method: `get_value`

Get the value of the message.

## Chapter 13

# basic\_message.py

BasicMessage: A simple message class that extends BaseMessage.

This class can be used to create messages that do not require any additional fields or methods. It inherits all the functionality from BaseMessage and can be used as a placeholder for messages that do not need any specific attributes.

### 13.1 Method: to\_dict

Convert the BasicMessage to a dictionary representation.

**Returns:**

A dictionary containing the content and headers of the message.

### 13.2 Method: from\_dict

Create a BasicMessage from a dictionary representation.

**Arguments:**

- data  
/ *Condition*: required / *Type*: dict /  
A dictionary containing the content and headers of the message.

**Returns:**

An instance of BasicMessage created from the provided dictionary.

### 13.3 Method: from\_data

Create a BasicMessage from raw data.

**Arguments:**

- data  
/ *Condition*: required / *Type*: str /  
The raw data to create the message from. This should be a string representation of the message content.

**Returns:**

An instance of BasicMessage created from the provided data.

## 13.4 Method: `get_value`

Convert the `BasicMessage` to raw data.

**Returns:**

A string representation of the message content.



## Chapter 14

# control\_message.py

14.1 Method: get\_value

14.2 Method: from\_data

## Chapter 15

### `__init__.py`

The package for standard messages.

Inspired by [https://github.com/ros/std\\_msgs](https://github.com/ros/std_msgs), this package defines messages of primitive data types and multiarrays, that can be reused for many common topics. If a more complex structural message is required, please add it to a different package, e.g. `taf_msgs`.

## Chapter 16

# dict\_msg.py

The dict message module. eventbusclient-message-deprecated-std-msgs-dict-msg-dictmsg =====

The dict message class.

## Chapter 17

# float32\_msg.py

The float32 message module. eventbusclient-message-deprecated-std-msgs-float32-msg-float32msg =====

The float32 message class.

## Chapter 18

# float64\_msg.py

The float64 message module. eventbusclient-message-deprecated-std-msgs-float64-msg-float64msg =====

The float64 message class.

# Chapter 19

## header.py

The header module. eventbusclient-message-deprecated-std-msgs-header-header =====  
The header class.

## Chapter 20

# int32\_msg.py

The int32 message module. eventbusclient-message-deprecated-std-msgs-int32-msg-int32msg =====

The int32 message class.

# Chapter 21

## msg.py

The base message module. eventbusclient-message-deprecated-std-msgs-msg-msg =====  
The base message class. eventbusclient-message-deprecated-std-msgs-msg-msg-serialize -----  
-----  
Serialize. eventbusclient-message-deprecated-std-msgs-msg-msg-deserialize -----  
-----  
Deserialize.



## Chapter 22

# string\_msg.py

The string message module. eventbusclient-message-deprecated-std-msgs-string-msg-stringmsg =====

The string message class.

## Chapter 23

### `uint32_msg.py`

The `uint32` message module. `eventbusclient-message-deprecated-std-msgs-uint32-msg-uint32msg` =====

The `uint32` message class.

## Chapter 24

### `__init__.py`

The package for TAF messages.

This package defines messages for TAF operations, e.g. rack synchronization, recovery, etc. If just a primitive data type is enough for your use case, please refer to the `std_msgs` package.

## Chapter 25

# listener\_event\_indexer.py

The listener event indexer module.

This module implements a compact indexer specialized for the synchronization across multiple TAF instances. The idea is to pack all possible states into one 64-bit integer (hence 'compact') that is guaranteed to be numerically comparable.

### 25.1 Class: ListenerEventIndexer

The listener event indexer class. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-is-special-code -----

Check if a number is code for special purpose. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-is-recovery-code -----

Check if a number is code for recovery mode. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-recovery-code -----

Return special code for recovery mode. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-get-recovery-scope -----

Extract recovery scope from code. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-encode -----

Return the combined index. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-decode -----

Decompose the combined index. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-suite-setup-intro -----

Check if suite setup started. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-suite-setup-outro -----

Check if suite setup completed. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-suite-teardown-intro -----

Check if suite teardown started. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-suite-teardown-outro -----

Check if suite teardown completed. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-test-setup-intro -----

Check if test setup started. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-test-setup-outro -----

Check if test setup completed. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-test-teardown-intro -----

Check if test teardown started. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-test-teardown-outro -----

Check if test teardown completed. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-cycle-index -----

Getter for cycle index. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-suite-

```

state -----
Getter for suite state. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-suite-
index -----
Getter for suite index. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-test-
state -----
Getter for test state. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-test-
index -----
Getter for test index. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-reset-
cycle-state -----
Reset cycle state. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-reset-suite-
state -----
Reset suite state. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-reset-test-
state -----
Reset test state. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-listenereventindexer-inc-cycle-
state -----
Move up the cycle index to next state. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-
listenereventindexer-inc-suite-state -----
Move up the suite index to next state. eventbusclient-message-deprecated-taf-msgs-listener-event-indexer-
listenereventindexer-inc-test-state -----
Move up the test index to next state.

```

## Chapter 26

# listener\_event\_msg.py

The listener event message module. eventbusclient-message-deprecated-taf-msgs-listener-event-msg-listenereventmsg

=====

The listener event message class.

## Chapter 27

# dict\_message.py

DictMessage: A message that can be initialized from a dictionary. eventbusclient-message-dict-message-dictmessage-from-data -----

### 27.1 Method: get\_value

## Chapter 28

# plugin\_loader.py

### 28.1 Class: ConfigValidator

Validates configuration data against a given schema. Raises `ValueError` if validation fails.

#### 28.1.1 Method: `validate`

Validate the configuration against the schema.

**Arguments:**

- `config`  
/ *Condition*: required / *Type*: dict /  
Configuration data to validate.

### 28.2 Class: PluginLoader

The *PluginLoader* class dynamically loads serializers, exchange handlers, and messages.

This class scans specified directories for Python modules, imports them, and registers any classes that match the expected base types (e.g., *Serializer*, *ExchangeHandler*, and *BaseMessage*). It is designed to facilitate the dynamic discovery and use of plugins in the application.

#### 28.2.1 Method: `get_serializer`

Get a serializer class by its name.

**Arguments:**

- `name`  
/ *Condition*: required / *Type*: str /  
Name of the serializer class to retrieve.

**Returns:**

/ *Type*: `Serializer` | `None` /  
Serializer class or `None` if not found.

#### 28.2.2 Method: `get_exchange_handler`

Get an exchange handler class by its name.

**Arguments:**



- `name`  
/ *Condition*: required / *Type*: str /  
Name of the exchange handler class to retrieve.

**Returns:**

/ *Type*: ExchangeHandler | None /  
Exchange handler class or None if not found.

### 28.2.3 Method: `get_message`

Get a message class by its name.

**Arguments:**

- `name`  
/ *Condition*: required / *Type*: str /  
Name of the message class to retrieve

**Returns:**

/ *Type*: BaseMessage | None /  
Message class or None if not found.

### 28.2.4 Method: `load_config`

Load configuration from a JSONP file and validate it against the schema.

**Arguments:**

- `config_path`  
/ *Condition*: required / *Type*: str /  
Path to the configuration file. If it is a relative path, it will be resolved to an absolute path.

**Returns:**

/ *Type*: DotDict | None /  
A DotDict containing the configuration data if the file exists and is loaded successfully, otherwise None.

## Chapter 29

# publisher.py

AsyncPublisher: Publishes messages to an exchange using aio\_pika.

### 29.1 Method: publish

Publish a message to the exchange with the specified routing key.

**Arguments:**

- `message`  
/ *Condition*: required / *Type*: `BaseMessage` /  
The message to be published. It should be an instance of `BaseMessage` or its subclasses.
- `routing_key`  
/ *Condition*: required / *Type*: `str` /  
The routing key used to route the message to the appropriate subscribers.
- `headers`  
/ *Condition*: optional / *Type*: `dict` /  
Additional headers to include with the message. This can be used for metadata or routing information.

**Note:** This method serializes the message using the specified serializer and publishes it to the exchange with the given routing key. The message is sent with a delivery mode of `NOT_PERSISTENT`, meaning it will not be saved to disk and will not survive a broker restart.

# Chapter 30

## qlogger.py

### 30.1 Class: ColorFormatter

Custom formatter class for setting log color.

#### 30.1.1 Method: format

Set the color format for the log.

**Arguments:**

- `record`  
/ *Condition*: required / *Type*: str /  
Log record.

**Returns:**

/ *Type*: logging.Formatter /  
Log with color formatter.

### 30.2 Class: QFileHandler

Handler class for user defined file in config.

#### 30.2.1 Method: get\_log\_path

Get the log file path for this handler.

**Arguments:**

- `config`  
/ *Condition*: required / *Type*: DictToClass /  
Connection configurations.

**Returns:**

/ *Type*: str /  
Log file path.

### 30.2.2 Method: get\_config\_supported

Check if the connection config is supported by this handler.

**Arguments:**

- `config`  
/ *Condition*: required / *Type*: DictToClass /  
Connection configurations.

**Returns:**

/ *Type*: bool /  
True if the config is supported.  
False if the config is not supported.

## 30.3 Class: QDefaultFileHandler

Handler class for default log file path.

### 30.3.1 Method: get\_log\_path

Get the log file path for this handler.

**Arguments:**

- `logger_name`  
/ *Condition*: required / *Type*: str /  
Name of the logger.

**Returns:**

/ *Type*: str /  
Log file path.

### 30.3.2 Method: get\_config\_supported

Check if the connection config is supported by this handler.

**Arguments:**

- `config`  
/ *Condition*: required / *Type*: DictToClass /  
Connection configurations.

**Returns:**

/ *Type*: bool /  
True if the config is supported.  
False if the config is not supported.

## 30.4 Class: QConsoleHandler

Handler class for console log.

### 30.4.1 Method: get\_config\_supported

Check if the connection config is supported by this handler.

**Arguments:**

- `config`  
/ *Condition*: required / *Type*: DictToClass /  
Connection configurations.

**Returns:**

/ *Type*: bool /  
True if the config is supported.  
False if the config is not supported.

## 30.5 Class: QLogger

Logger class for QConnect Libraries.

### 30.5.1 Method: get\_logger

Get the logger object.

**Arguments:**

- `logger_name`  
/ *Condition*: required / *Type*: str /  
Name of the logger.

**Returns:**

- `logger`  
/ *Type*: Logger /  
Logger object. .

### 30.5.2 Method: set\_handler

Set handler for logger.

**Arguments:**

- `config`  
/ *Condition*: required / *Type*: DictToClass /  
Connection configurations.

**Returns:**

- `handler_ins`  
/ *Type*: logging.handler /  
None if no handler is set.  
Handler object.

### 30.5.3 Method: get\_handler

Get the handler of the logger.

## Chapter 31

# rendezvous.py

### 31.1 Method: `announce_ready`

Broadcast that this process is ready to consume for the given roles. Call this once after you have subscribed/initialized consumers.

**Arguments:**

- `roles`  
/ *Condition*: required / *Type*: List[str] /  
A list of roles or topic patterns that this instance is ready to handle.

### 31.2 Method: `wait_for`

Wait until we have observed readiness announcements satisfying the requirements. *requirements* maps a role/topic pattern -> minimum unique instance count. Returns True if satisfied before timeout, False otherwise.

**Arguments:**

- `requirements`  
/ *Condition*: required / *Type*: Dict[str, int] /  
A dictionary mapping role or topic patterns to the minimum number of unique instances required.
- `timeout`  
/ *Condition*: optional / *Type*: float /  
The maximum time to wait for the requirements to be satisfied, in seconds. Defaults to 5.0 seconds.

## Chapter 32

# base\_serializer.py

### 32.1 Method: `serialize`

Serialize a message object to bytes.

**Arguments:**

- `msg`  
/ *Condition*: required / *Type*: Any /  
Message object to be serialized.

### 32.2 Method: `deserialize`

Deserialize bytes back into a message object.

**Arguments:**

- `data`  
/ *Condition*: required / *Type*: bytes /  
Serialized message data to be deserialized.

## Chapter 33

# json\_serializer.py

JsonSerializer: Serializes BaseMessage subclasses to JSON strings. Requires message classes to implement:

**System Message**

```
ERROR/3 in <string>, line 6
Unexpected indentation. backrefs:
```

- `to_dict()`
- `from_dict(data: dict)`

### 33.1 Method: serialize

Serialize a message object to JSON bytes.

**Arguments:**

- `msg`  
/ *Condition*: required / *Type*: BaseMessage /  
Message object to be serialized.

**Returns:**

/ *Type*: bytes /  
Serialized message as JSON bytes.

### 33.2 Method: deserialize

Deserialize bytes back into a message object.

**Arguments:**

- `data`  
/ *Condition*: required / *Type*: bytes /  
Serialized message data in bytes format.
- `message_cls`  
/ *Condition*: required / *Type*: Type[BaseMessage] /  
Class of the message to deserialize into.

**Returns:**

/ *Type*: str /  
Deserialized message object of type *message\_cls*.



**Raises:**

- `ValueError` If *message\_cls* is not provided.
- `TypeError` If *message\_cls* does not implement *from\_dict(data: dict)*.
- `RuntimeError` If deserialization fails due to invalid data or other issues.

## Chapter 34

# pickle\_serializer.py

PickleSerializer: Built-in serializer using Python pickle.

WARNING: Pickle is not secure against untrusted data. Only use in trusted environments.

### 34.1 Method: serialize

Serialize a message object to bytes using pickle.

**Arguments:**

- msg  
/ *Condition*: required / *Type*: Any /  
Message object to be serialized.

### 34.2 Method: deserialize

Deserialize bytes back into a message object using pickle.

**Arguments:**

- data  
/ *Condition*: required / *Type*: bytes /  
Serialized message data to be deserialized.

## Chapter 35

# protobuf\_serializer.py

ProtobufSerializer: Serializer using Protocol Buffers.

Requires protobuf message classes generated from .proto files.

### 35.1 Method: serialize

Serialize a protobuf message object to bytes.

**Arguments:**

- msg  
/ *Condition*: required / *Type*: Message /  
Protobuf message object to be serialized.

### 35.2 Method: deserialize

Deserialize bytes back into a protobuf message object.

**Arguments:**

- data  
/ *Condition*: required / *Type*: bytes /  
Serialized protobuf message data to be deserialized.
- message\_cls  
/ *Condition*: required / *Type*: type /  
Protobuf message class to instantiate.

## Chapter 36

# startup\_policy.py

### 36.1 Function: `resolve_message_cls`

Turn a JSON 'class' string into an actual class object.

**Arguments:**

- `value`  
/ *Condition*: required / *Type*: str or type or None /  
The class to resolve. Can be a dotted path, a short name, or a type.
- `base_cls`  
/ *Condition*: required / *Type*: type /  
The base class that the resolved class must inherit from.
- `registry`  
/ *Condition*: optional / *Type*: dict[str, type] / *Default*: None /  
Optional registry mapping short names to classes.
- `extra_modules`  
/ *Condition*: optional / *Type*: list[str] / *Default*: None /  
List of module names to search for the class by short name.
- `default`  
/ *Condition*: optional / *Type*: type / *Default*: None /  
Default class to use if value is None.

**Returns:**

/ *Type*: type /  
The resolved class object inheriting from `base_cls`.

### 36.2 Function: `build_policy_from_item`

Build a StartupPolicy from a configuration item.

**Arguments:**

- `item`  
/ *Condition*: required / *Type*: dict /  
The configuration item with keys:
  - "class": str - dotted path to the policy class

- "args": dict - optional arguments for the policy class
- "wrap": list - optional list of wrappers, each with "class" and "args"

**Returns:**

/ *Type*: StartupPolicy /

The constructed StartupPolicy instance.

### 36.3 Function: build\_policy\_from\_config

Build a StartupPolicy from a configuration dictionary.

**Arguments:**

- `cfg`

/ *Condition*: required / *Type*: dict /

The configuration dictionary. Supports either:

- Legacy single policy:
  - \* "startup\_policy": str - dotted path to the policy class
  - \* "startup\_policy\_args": dict - optional arguments for the policy class
- New multi-policy:
  - \* "startup\_policies": list - list of policy items (see build\_policy\_from\_item)
  - \* "startup\_policies\_mode": str - optional mode ("sequential" or "parallel")
  - \* "startup\_policies\_fail\_fast": bool - optional fail-fast flag (default True)

**Returns:**

/ *Type*: Optional[StartupPolicy] /

The constructed StartupPolicy instance, or None if no policy is configured.

### 36.4 Class: StartupPolicy

A startup policy can delay or block the completion of `EventBusClient.start()` until certain conditions are met. This can be used to implement rendezvous patterns, fixed delays, or other custom logic.

#### 36.4.1 Method: wait\_until\_ready

### 36.5 Class: NoWait

No waiting, start immediately.

#### 36.5.1 Method: wait\_until\_ready

Wait until ready - no wait.

**Arguments:**

- `bus`

/ *Condition*: required / *Type*: EventBusClient /

The EventBusClient instance.

**Returns:**

/ *Type*: None /

None

## 36.6 Class: FixedDelay

### 36.6.1 Method: wait\_until\_ready

Wait for a fixed delay before proceeding.

**Arguments:**

- `bus`  
/ *Condition*: required / *Type*: `EventBusClient` /  
The `EventBusClient` instance.

## 36.7 Class: HandshakeBarrier

Wait until at least N consumers announce ready for the given roles/topics. Example:

### System Message

```
ERROR/3 in <string>, line 158
Unexpected indentation. backrefs:
```

```
HandshakeBarrier({"telemetry.*": 1, "orders.created": 2}, timeout=5.0)
```

### 36.7.1 Method: wait\_until\_ready

## 36.8 Class: PanelControlLegacyByAlias

Legacy startup policy for panel control mode based on controller alias.

### 36.8.1 Method: wait\_until\_ready

## 36.9 Class: GeneralCacheStartupPolicy

Example startup policy that decides whether to start a 'general cache' at connect-time, and with which routing keys / message class. Can be alias-aware.

### 36.9.1 Method: wait\_until\_ready

Enable the general cache listener on the client if applicable.

**Arguments:**

- `client`  
/ *Condition*: required / *Type*: `EventBusClient` /  
The `EventBusClient` instance to apply the policy on.

## 36.10 Class: ConfigureUnroutablePolicy

Configure unroutable behavior BEFORE exchange setup. mode: "drop" | "alternate-exchange" | "return"  
on\_unroutable: "log" | "raise" | "cache" | "callback"

### 36.10.1 Method: before\_setup

Configure unroutable message handling on the `EventBusClient`.

**Arguments:**

- `bus`  
/ *Condition*: required / *Type*: `EventBusClient` /  
The `EventBusClient` instance to configure.

### 36.10.2 Method: `wait_until_ready`

## 36.11 Class: `PolicyChain`

Apply multiple policies either sequentially or in parallel.

- `mode="sequential"`: run in order; if `fail_fast=True`, stop on first exception.
- `mode="parallel"`: run concurrently; if `fail_fast=True`, propagate first exception (`gather(..., return_exceptions=False)`).

### 36.11.1 Method: `wait_until_ready`

Apply all policies according to the configured mode.

#### Arguments:

- `bus`  
/ *Condition*: required / *Type*: `EventBusClient` /  
The `EventBusClient` instance to apply the policies on.

#### Returns:

/ *Type*: `None` /  
`None`

### 36.11.2 Method: `before_setup`

### 36.11.3 Method: `after_setup`

# Chapter 37

## subscriber.py

AsyncSubscriber: Subscribes to messages from an exchange using aio\_pika.

### 37.1 Method: cache

Get the subscription cache, initializing it if necessary.

**Returns:**

*/ Type: SubscriptionCache /*  
The subscription cache instance.

### 37.2 Method: start

Start the subscriber by declaring a queue, binding it to the exchange, and consuming messages.

**Arguments:**

- `cache_size`  
*/ Condition: optional / Type: Optional[int] /*  
The maximum size of the subscription cache. If not provided, defaults to the instance's default cache size.

**Returns:**

*/ Type: SubscriptionCache /*  
The subscription cache instance.

### 37.3 Method: stop

Stop the subscriber by canceling the consumer, unbinding the queue from the exchange, and deleting the queue.

### 37.4 Method: routing\_key

### 37.5 Method: callback



## Chapter 38

# subscription\_cache.py

A thread-safe cache for storing and retrieving items in a FIFO manner. `eventbusclient-subscription-cache-subscriptioncache-register-callback` -----

Register a callback function to be called whenever a new item is added to the cache.

### Arguments:

- `callback`  
/ *Condition*: required / *Type*: Callable[[T], None] /  
A function that takes an item of type T and returns None. This function will be called in a separate thread whenever a new item is added to the cache.

### 38.1 Method: `append`

Add an item to the cache, possibly dropping the oldest item if full.

### Arguments:

- `item`  
/ *Condition*: required / *Type*: T /  
The item to add to the cache.

### 38.2 Method: `get`

Block until any item arrives; return it (FIFO) and remove it.

### Arguments:

- `timeout`  
/ *Condition*: optional / *Type*: Optional[float] / *Default*: None /  
Maximum time to wait for an item in seconds. If None, wait indefinitely.

### 38.3 Method: `pop`

Alias for `get()`.

### Arguments:

- `timeout`  
/ *Condition*: optional / *Type*: Optional[float] / *Default*: None /  
Maximum time to wait for an item in seconds. If None, wait indefinitely.

### Returns:

*/ Type: T /*

The next item from the cache, removed from the cache.

## 38.4 Method: pop\_nothrow

Remove and return the first item in the cache, only if not empty.

**Returns:**

*/ Type: Optional[T] /*

The first item from the cache, or None if the cache is empty.

## 38.5 Method: peek\_nothrow

Peek at the first item in the cache, only if not empty.

**Returns:**

*/ Type: Optional[T] /*

The first item from the cache, or None if the cache is empty.

## 38.6 Method: peek

Block until any item arrives; return it (FIFO) but do not remove it.

**Arguments:**

- `timeout`

*/ Condition: optional / Type: Optional[float] / Default: None /*

Maximum time to wait for an item in seconds. If None, wait indefinitely.

**Returns:**

*/ Type: T /*

The next item from the cache, without removing it.

## 38.7 Method: wait\_for

Block until an item matching the predicate arrives; return it and remove it.

**Arguments:**

- `predicate`

*/ Condition: required / Type: Callable[[T], bool] /*

A function that takes an item of type T and returns True if it matches the desired condition.

- `timeout`

*/ Condition: optional / Type: Optional[float] / Default: None /*

Maximum time to wait for a matching item in seconds. If None, wait indefinitely.

**Returns:**

*/ Type: T /*

The first item that matches the predicate, removed from the cache.

## 38.8 Method: drain

Remove and return up to `max_items` from the cache in FIFO order.

### Arguments:

- `max_items`  
/ *Condition*: optional / *Type*: Optional[int] / *Default*: None /  
Maximum number of items to remove. If None, remove all items.

### Returns:

/ *Type*: List[T] /  
A list of removed items, in the order they were in the cache.

## 38.9 Method: peek\_last

Peek at the last item in the cache, only if not empty.

### Returns:

/ *Type*: Optional[T] /  
The last item in the cache, or None if the cache is empty.

## 38.10 Method: aget

Async version of get using a thread executor.

### Arguments:

- `timeout`  
/ *Condition*: optional / *Type*: Optional[float] / *Default*: None /  
Maximum time to wait for an item in seconds. If None, wait indefinitely.

### Returns:

/ *Type*: T /  
The next item from the cache, removed from the cache.

## 38.11 Method: await\_for

Async version of wait\_for using a thread executor.

### Arguments:

- `predicate`  
/ *Condition*: required / *Type*: Callable[[T], bool] /  
A function that takes an item of type T and returns True if it matches the desired condition.
- `timeout`  
/ *Condition*: optional / *Type*: Optional[float] / *Default*: None /  
Maximum time to wait for a matching item in seconds. If None, wait indefinitely.

### Returns:

/ *Type*: T /  
The first item that matches the predicate, removed from the cache.

## 38.12 Method: wait\_for\_one

Wait for a single target message to appear in the cache.

### Arguments:

- `target`  
/ *Condition*: required / *Type*: Any /  
The target message to wait for.
- `timeout`  
/ *Condition*: optional / *Type*: float / *Default*: 30.0 /  
Maximum time to wait in seconds.
- `dropped_msgs`  
/ *Condition*: optional / *Type*: List[Any] / *Default*: None /  
The list of messages having been discarded. Defaults to None.

### Returns:

/ *Type*: bool /  
True if the target message was found and removed from the cache, False if the timeout was reached.

## 38.13 Method: wait\_for\_many

Wait for multiple target messages to appear in the cache.

### Arguments:

- `targets`  
/ *Condition*: required / *Type*: List[Any] /  
List of target messages to wait for.
- `mode`  
/ *Condition*: required / *Type*: WaitMode /  
Mode of waiting:
  - `WaitMode.ALL_IN_GIVEN_ORDER`: Wait for all target messages in the given order.
  - `WaitMode.ALL_IN_RANDOM_ORDER`: Wait for all target messages in any order.
  - `WaitMode.ANY_OF_GIVEN_MSGS`: Wait for any one of the target messages.
- `timeout`  
/ *Condition*: optional / *Type*: float / *Default*: 30.0 /  
Maximum time to wait in seconds.
- `dropped_msgs`  
/ *Condition*: optional / *Type*: List[Any] / *Default*: None /  
The list of messages having been discarded. Defaults to None.

### Returns:

/ *Type*: List[int] /  
List of indices of the target messages that were found, in the order they were found.

# Chapter 39

## utils.py

### 39.1 Class: Singleton

Class to implement Singleton Design Pattern. This class is used to derive the TTFisClientReal as only a single instance of this class is allowed.

Disabled pyLint Messages: R0903: Too few public methods (%s/%s)

**System Message**

ERROR/3 in <string>, line 9  
Unexpected indentation. backrefs:

Used when class has too few public methods, so be sure it's really worth it.

This base class implements the Singleton Design Pattern required for the TTFisClientReal. Adding further methods does not make sense.

**System Message**

WARNING/2 in <string>, line 13  
Block quote ends without a blank line; unexpected unindent. backrefs:

### 39.2 Class: DictToClass

Class for converting dictionary to class object.

#### 39.2.1 Method: validate

### 39.3 Class: Utils

Class to implement utilities for supporting development.

#### 39.3.1 Method: get\_all\_descendant\_classes

Get all descendant classes of a class

**Arguments:** cls: Input class for finding descendants.

**Returns:**

/ *Type*: list /

Array of descendant classes.

### 39.3.2 Method: get\_all\_sub\_classes

Get all children classes of a class

#### Arguments:

- `cls`  
/ *Condition*: required / *Type*: class /  
Input class for finding children.

#### Returns:

/ *Type*: list /  
Array of children classes.

### 39.3.3 Method: is\_valid\_host

### 39.3.4 Method: caller\_name

Get a name of a caller in the format module.class.method

#### Arguments:

- `skip`  
/ *Condition*: required / *Type*: int /  
Specifies how many levels of stack to skip while getting caller name. \* `skip=1` means "who calls me" \* `skip=2` means "who calls my caller" etc.

#### Returns:

/ *Type*: str /  
An empty string is returned if skipped levels exceed stack height

### 39.3.5 Method: load\_library

Load native library depend on the calling convention.

#### Arguments:

- `path`  
/ *Condition*: required / *Type*: str /  
Library path.
- `is_stdcall`  
/ *Condition*: optional / *Type*: bool / *Default*: True /  
Determine if the library's calling convention is stdcall or cdecl.

#### Returns:

*Loaded library object.*

### 39.3.6 Method: is\_ascii\_or\_unicode

Check if the string is ascii or unicode

**Arguments:** `str_check`: string for checking codecs: encoding type list

#### Returns:

/ *Type*: bool /  
True : if checked string is ascii or unicode  
False : if checked string is not ascii or unicode

## 39.4 Class: Job

### 39.4.1 Method: stop

### 39.4.2 Method: run

## 39.5 Class: ResultType

Result Types.

## 39.6 Class: ResponseMessage

Response message class

### 39.6.1 Method: get\_json

Convert response message to json

**Returns:**

*Response message in json format*

### 39.6.2 Method: get\_data

Get string data result

**Returns:**

*String result*

### 39.6.3 Method: create\_from\_string

## Chapter 40

# wait\_mode.py

Enumeration for different wait modes.

- ALL\_IN\_GIVEN\_ORDER: Wait for all specified messages in the given order.
- ALL\_IN\_RANDOM\_ORDER: Wait for all specified messages in any order.
- ANY\_OF\_GIVEN\_MSGS: Wait for any one of the specified messages.



## Chapter 41

# Glossary

**RabbitMQ** (open-source message broker)

⇒ [homepage](#)

⇒ [documentation for developers](#)

**EventBusClient** (high-level messaging framework built on top of **RabbitMQ**)

⇒ [homepage](#)

**RobotFramework AIO** (**AIO** = **A**ll **I**n **O**ne; test automation framework with extended features, based on the **Robot Framework**)

⇒ [RobotFramework AIO homepage](#)

⇒ [Robot Framework homepage](#)

## Chapter 42

# Appendix

About this package:

Table 42.1: Package setup

Setup parameter	Value
Name	EventBusClient
Version	0.2.0
Date	29.01.2026
Description	An IPC message-bus based on RabbitMQ message broker
Package URL	<a href="#">python-rabbitmq-messagebus</a>
Author	Nguyen Huynh Tri Cuong
Email	<a href="mailto:Cuong.NguyenHuynhTri@vn.bosch.com">Cuong.NguyenHuynhTri@vn.bosch.com</a>
Language	Programming Language :: Python :: 3
License	License :: OSI Approved :: Apache Software License
OS	Operating System :: OS Independent
Python required	$\geq 3.0$
Development status	Development Status :: 4 - Beta
Intended audience	Intended Audience :: Developers
Topic	Topic :: Software Development

## Chapter 43

# History

<b>0.1.0</b>	07/2025
<i>Initial version</i>	
<b>0.1.2</b>	01/2026
<i>Fixed bugs and improve stability.</i>	
<b>0.2.0</b>	01/2026
<i>Added <code>HeadersExchangeHandler</code> for header-based message routing (<code>x-match=all/any</code>).</i>	
<i>Enhanced <code>on()</code> and <code>on_sync()</code> methods with <code>binding_headers</code> and <code>match_all</code> parameters.</i>	
<i>Added comprehensive documentation including ADRs and architecture diagrams.</i>	
<i>Added test cases for <code>HeadersExchangeHandler</code> (<code>EBC_0042</code> to <code>EBC_0047</code>).</i>	
<i>Updated examples with <code>headers_exchange_sample.py</code>.</i>	