

EventBusClient

v. 0.1.1

Nguyen Huynh Tri Cuong

17.10.2025

Contents

1	Introduction	1
2	Description	2
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	4
2.8.1	Initialization	4
2.8.2	Publishing Messages	4
2.8.3	Subscribing to Messages	4
2.9	Example: Producer and Consumer	5
2.10	Example: Custom Plugin Structure	5
2.11	Example: Built-in Plugins	5
2.12	Example: Built-in Configuration	6
2.13	Example: Custom Exchange Handler	6
2.14	Example: Custom Message Class	6
2.15	Example: Custom Serializer	7
3	__init__.py	8
3.1	Function: get_logger	8
4	connection.py	9
4.1	Class: ConnectionManager	9
4.1.1	Method: is_connected	9
4.1.2	Method: reset_loop	9
4.1.3	Method: register_exchange_handler	9
4.1.4	Method: unregister_exchange_handler	9
5	constants.py	10
5.1	Class: SocketType	10
5.2	Class: String	10
6	event_bus_client.py	11
6.1	Function: get_running_loop_or_none	11
6.2	Function: is_on_loop	11

6.3	Class: EventBusClient	11
6.3.1	Method: getVersion	11
6.3.2	Method: getVersionDate	11
6.3.3	Method: wait_on_general_topic_for_one	11
6.3.4	Method: wait_on_general_topic_for_many	12
6.3.5	Method: wait_on_cache_for_one	12
6.3.6	Method: wait_on_cache_for_many	13
6.3.7	Method: constructor_params_from_config	14
6.3.8	Method: is_connected	14
6.3.9	Method: build_routing_key	14
6.3.10	Method: start_background_loop	15
6.3.11	Method: stop_background_loop	15
6.3.12	Method: from_config_sync	15
6.3.13	Method: connect_sync	15
6.3.14	Method: send_sync	16
6.3.15	Method: on_sync	16
6.3.16	Method: off_sync	17
6.3.17	Method: wait_until_ready_sync	17
6.3.18	Method: announce_ready_sync	17
6.3.19	Method: close_sync	17
7	basic_async_producer_consumer_sample.py	18
7.1	Function: run_process	18
7.2	Class: TestMessage	18
7.2.1	Method: from_data	18
7.2.2	Method: get_value	18
8	basic_sync_producer_consumer_sample.py	19
8.1	Function: producer_process	19
8.2	Function: consumer_process	19
8.3	Function: main	19
8.4	Class: TestMessage	19
8.4.1	Method: from_data	19
8.4.2	Method: get_value	19
9	base.py	20
9.1	Class: ExchangeHandler	20
9.1.1	Method: reset_loop	20
10	topic_handler.py	21
10.1	Class: TopicExchangeHandler	21
11	x_rtopic_handler.py	22
11.1	Class: XRTopicExchangeHandler	22
12	base_message.py	23
12.1	Class: BaseMessage	23
12.1.1	Method: from_data	23

12.1.2 Method: <code>get_value</code>	23
13 <code>basic_message.py</code>	24
13.1 Class: <code>BasicMessage</code>	24
13.1.1 Method: <code>to_dict</code>	24
13.1.2 Method: <code>from_dict</code>	24
13.1.3 Method: <code>from_data</code>	24
13.1.4 Method: <code>get_value</code>	25
14 <code>control_message.py</code>	26
14.1 Class: <code>ControlMessage</code>	26
14.1.1 Method: <code>get_value</code>	26
14.1.2 Method: <code>from_data</code>	26
15 <code>__init__.py</code>	27
16 <code>dict_msg.py</code>	28
17 <code>float32_msg.py</code>	29
18 <code>float64_msg.py</code>	30
19 <code>header.py</code>	31
20 <code>int32_msg.py</code>	32
21 <code>msg.py</code>	33
22 <code>string_msg.py</code>	34
23 <code>uint32_msg.py</code>	35
24 <code>__init__.py</code>	36
25 <code>listener_event_indexer.py</code>	37
25.1 Class: <code>ListenerEventIndexer</code>	37
26 <code>listener_event_msg.py</code>	39
27 <code>dict_message.py</code>	40
27.1 Class: <code>DictMessage</code>	40
27.1.1 Method: <code>get_value</code>	40
28 <code>plugin_loader.py</code>	41
28.1 Class: <code>ConfigValidator</code>	41
28.1.1 Method: <code>validate</code>	41
28.2 Class: <code>PluginLoader</code>	41
28.2.1 Method: <code>get_serializer</code>	41
28.2.2 Method: <code>get_exchange_handler</code>	42
28.2.3 Method: <code>get_message</code>	42
28.2.4 Method: <code>load_config</code>	42

29 publisher.py	43
29.1 Class: AsyncPublisher	43
30 qlogger.py	44
30.1 Class: ColorFormatter	44
30.1.1 Method: format	44
30.2 Class: QFileHandler	44
30.2.1 Method: get_log_path	44
30.2.2 Method: get_config_supported	45
30.3 Class: QDefaultFileHandler	45
30.3.1 Method: get_log_path	45
30.3.2 Method: get_config_supported	45
30.4 Class: QConsoleHandler	46
30.4.1 Method: get_config_supported	46
30.5 Class: QLogger	46
30.5.1 Method: get_logger	46
30.5.2 Method: set_handler	46
30.5.3 Method: get_handler	47
31 rendezvous.py	48
31.1 Class: Rendezvous	48
32 base_serializer.py	49
32.1 Class: Serializer	49
32.1.1 Method: serialize	49
32.1.2 Method: deserialize	49
33 json_serializer.py	50
33.1 Class: JsonSerializer	50
33.1.1 Method: serialize	50
33.1.2 Method: deserialize	50
34 pickle_serializer.py	52
34.1 Class: PickleSerializer	52
34.1.1 Method: serialize	52
34.1.2 Method: deserialize	52
35 protobuf_serializer.py	53
35.1 Class: ProtobufSerializer	53
35.1.1 Method: serialize	53
35.1.2 Method: deserialize	53
36 startup_policy.py	54
36.1 Function: resolve_message_cls	54
36.2 Class: StartupPolicy	54
36.3 Class: NoWait	54
36.4 Class: FixedDelay	54
36.5 Class: HandshakeBarrier	55
36.6 Class: PanelControlLegacyByAlias	55

37 subscriber.py	56
37.1 Class: AsyncSubscriber	56
37.1.1 Method: cache	56
37.1.2 Method: routing_key	56
37.1.3 Method: callback	56
38 subscription_cache.py	57
38.1 Class: SubscriptionCache	57
38.1.1 Method: append	57
38.1.2 Method: get	57
38.1.3 Method: pop	57
38.1.4 Method: pop_nothrow	58
38.1.5 Method: peek_nothrow	58
38.1.6 Method: peek	58
38.1.7 Method: wait_for	58
38.1.8 Method: drain	59
38.1.9 Method: peek_last	59
38.1.10 Method: wait_for_one	59
38.1.11 Method: wait_for_many	59
39 utils.py	61
39.1 Class: Singleton	61
39.2 Class: DictToClass	61
39.2.1 Method: validate	61
39.3 Class: Utils	61
39.3.1 Method: get_all_descendant_classes	61
39.3.2 Method: get_all_sub_classes	62
39.3.3 Method: is_valid_host	62
39.3.4 Method: caller_name	62
39.3.5 Method: load_library	62
39.3.6 Method: is_ascii_or_unicode	63
39.4 Class: Job	63
39.4.1 Method: stop	63
39.4.2 Method: run	63
39.5 Class: ResultType	63
39.6 Class: ResponseMessage	63
39.6.1 Method: get_json	63
39.6.2 Method: get_data	63
39.6.3 Method: create_from_string	63
40 wait_mode.py	64
40.1 Class: WaitMode	64
41 Glossary	65
42 Appendix	66
43 History	67

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 RabbitMQ 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.
- **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 event-bus-client
```

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
|   |-- exchange_handler/
|   |   |-- base.py
|   |   |-- xr_topic_handler.py
|   |-- serializer/
|   |   |-- base_serializer.py
|   |   |-- json_serializer.py
|   |   |-- protobuf_serializer.py
|   |   |-- pickle_serializer.py
|   |-- plugins/
|   |   |-- mcpi/
|   |   |   |-- serialize/
|   |   |   |-- message/
|   |   |   |-- exchange/
|   |-- 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.

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

2.8 Usage

2.8.1 Initialization

Create an **EventBusClient** instance from configuration:

```
from 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:

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

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

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

2.8.3 Subscribing to Messages

Subscribe to a routing key and handle incoming messages:

```
async def on_message(msg: ListenerEventMsg):
    print(f"Received event: {msg.event} at {msg.timestamp}")

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

2.9 Example: Producer and Consumer

This example (from `EventBusClient/test/test.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()
```

2.10 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
```

2.11 Example: Built-in Plugins

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

- **XRTopicExchangeHandler**: Handles topic exchanges with **RabbitMQ**.
- **ListenerEventMsg**: A default message class for listener events.
- **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.

2.12 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.

2.13 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"
}
```

2.14 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"
}
```

2.15 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`

3.1 Function: `get_logger`

Chapter 4

connection.py

4.1 Class: ConnectionManager

Imported by:

```
from EventBusClient.connection import ConnectionManager
```

ConnectionManager: Manages RabbitMQ connections, channels, and exchanges.

4.1.1 Method: is_connected

Check if the connection to RabbitMQ is established.

Returns:

/ Type: bool /

True if connected, False otherwise.

4.1.2 Method: reset_loop

4.1.3 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.

4.1.4 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.

Chapter 5

constants.py

5.1 Class: SocketType

Imported by:

```
from EventBusClient.constants import SocketType
```

5.2 Class: String

Imported by:

```
from EventBusClient.constants import String
```


Chapter 6

event_bus_client.py

6.1 Function: get_running_loop_or_none

6.2 Function: is_on_loop

6.3 Class: EventBusClient

Imported by:

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

EventBusClient: Client for interacting with the event bus system.

6.3.1 Method: getVersion

Returns the version of EventBusClient as string.

6.3.2 Method: getVersionDate

Returns the version date of EventBusClient as string.

6.3.3 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.

6.3.4 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.

6.3.5 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.

Returns:

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

6.3.6 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.

Returns:

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

6.3.7 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.

Returns:

/ Type: tuple /

A tuple containing the following elements in order:

- `exchange_handler`: An instance of ExchangeHandler loaded from the configuration.
- `serializer`: An instance of Serializer loaded from the configuration.
- `loop`: None (the event loop will be set later).
- `zone_id`: The provided zone_id argument.
- `alias`: The provided alias argument.
- `startup_policy`: The provided startup_policy argument.
- `prefetch_count`: The prefetch count loaded from the configuration (default is 10 if not specified).
- `auto_reconnect`: The auto_reconnect setting loaded from the configuration (default is True if not specified).

6.3.8 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.

6.3.9 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.

6.3.10 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".

6.3.11 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.

6.3.12 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.

6.3.13 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.

6.3.14 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.

6.3.15 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.
- `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.

Returns: / *Type*: `SubscriptionCache` /

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

6.3.16 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.

6.3.17 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.

6.3.18 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.

6.3.19 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.

Chapter 7

basic_async_producer_consumer_sample.py

7.1 Function: run_process

Helper function to run an async function in a process eventbusclient-examples-basic-async-producer-consumer-sample-

main =====

7.2 Class: TestMessage

Imported by:

```
from EventBusClient.examples.basic_async_producer_consumer_sample import TestMessage
```

7.2.1 Method: from_data

7.2.2 Method: get_value

Chapter 8

basic_sync_producer_consumer_sample.py

8.1 Function: producer_process

8.2 Function: consumer_process

8.3 Function: main

8.4 Class: TestMessage

Imported by:

```
from EventBusClient.examples.basic_sync_producer_consumer_sample import TestMessage
```

8.4.1 Method: from_data

8.4.2 Method: get_value

Chapter 9

base.py

9.1 Class: ExchangeHandler

Imported by:

```
from EventBusClient.exchange_handler.base import ExchangeHandler
```

9.1.1 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.

Chapter 10

topic_handler.py

10.1 Class: TopicExchangeHandler

Imported by:

```
from EventBusClient.exchange_handler.topic_handler import TopicExchangeHandler
```

Chapter 11

x_rtopic_handler.py

11.1 Class: XRTopicExchangeHandler

Imported by:

```
from EventBusClient.exchange_handler.x_rtopic_handler import XRTopicExchangeHandler
```

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

Chapter 12

base_message.py

12.1 Class: BaseMessage

Imported by:

```
from EventBusClient.message.base_message import BaseMessage
```

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

12.1.1 Method: from_data

Create a message instance from raw data.

Arguments:

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

12.1.2 Method: get_value

Get the value of the message.

Chapter 13

basic_message.py

13.1 Class: BasicMessage

Imported by:

```
from EventBusClient.message.basic_message import BasicMessage
```

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.

13.1.1 Method: to_dict

Convert the BasicMessage to a dictionary representation.

Returns:

A dictionary containing the content and headers of the message.

13.1.2 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.

13.1.3 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.

13.1.4 Method: `get_value`

Convert the `BasicMessage` to raw data.

Returns:

A string representation of the message content.

Chapter 14

control_message.py

14.1 Class: ControlMessage

Imported by:

```
from EventBusClient.message.control_message import ControlMessage
```

14.1.1 Method: get_value

14.1.2 Method: from_data

Chapter 15

`__init__.py`

The package for standard messages.

Inspired by 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 =====

Imported by:

```
from EventBusClient.message.deprecated.std_msgs.dict_msg import DictMsg
```

The dict message class.

Chapter 17

float32_msg.py

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

Imported by:

```
from EventBusClient.message.deprecated.std_msgs.float32_msg import Float32Msg
```

The float32 message class.

Chapter 18

float64_msg.py

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

Imported by:

```
from EventBusClient.message.deprecated.std_msgs.float64_msg import Float64Msg
```

The float64 message class.

Chapter 19

header.py

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

Imported by:

```
from EventBusClient.message.deprecated.std_msgs.header import Header
```

The header class.

Chapter 20

int32_msg.py

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

Imported by:

```
from EventBusClient.message.deprecated.std_msgs.int32_msg import Int32Msg
```

The int32 message class.

Chapter 21

msg.py

The base message module. eventbusclient-message-deprecated-std-msgs-msg-msg =====

Imported by:

```
from EventBusClient.message.deprecated.std_msgs.msg import 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 =====

Imported by:

```
from EventBusClient.message.deprecated.std_msgs.string_msg import StringMsg
```

The string message class.

Chapter 23

uint32_msg.py

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

Imported by:

```
from EventBusClient.message.deprecated.std_msgs.uint32_msg import 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

Imported by:

```
from EventBusClient.message.deprecated.taf_msgs.listener_event_indexer import
↳ 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`

=====

Imported by:

```
from EventBusClient.message.deprecated.taf_msgs.listener_event_msg import
↳ ListenerEventMsg
```

The listener event message class.

Chapter 27

dict_message.py

27.1 Class: DictMessage

Imported by:

```
from EventBusClient.message.dict_message import DictMessage
```

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

27.1.1 Method: get_value

Chapter 28

plugin_loader.py

28.1 Class: ConfigValidator

Imported by:

```
from EventBusClient.plugin_loader import 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

Imported by:

```
from EventBusClient.plugin_loader import 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

29.1 Class: AsyncPublisher

Imported by:

```
from EventBusClient.publisher import AsyncPublisher
```

AsyncPublisher: Publishes messages to an exchange using aio-pika.

Chapter 30

qlogger.py

30.1 Class: ColorFormatter

Imported by:

```
from EventBusClient.qlogger import 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

Imported by:

```
from EventBusClient.qlogger import 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

Imported by:

```
from EventBusClient.qlogger import 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

Imported by:

```
from EventBusClient.qlogger import 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

Imported by:

```
from EventBusClient.qlogger import 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 Class: Rendezvous

Imported by:

```
from EventBusClient.rendezvous import Rendezvous
```

Chapter 32

base_serializer.py

32.1 Class: Serializer

Imported by:

```
from EventBusClient.serializer.base_serializer import Serializer
```

32.1.1 Method: serialize

Serialize a message object to bytes.

Arguments:

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

32.1.2 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

33.1 Class: JsonSerializer

Imported by:

```
from EventBusClient.serializer.json_serializer import JsonSerializer
```

JsonSerializer: Serializes BaseMessage subclasses to JSON strings. Requires message classes to implement: - to_dict()
- from_dict(data: dict)

33.1.1 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.

33.1.2 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

34.1 Class: PickleSerializer

Imported by:

```
from EventBusClient.serializer.pickle_serializer import PickleSerializer
```

PickleSerializer: Built-in serializer using Python pickle.

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

34.1.1 Method: serialize

Serialize a message object to bytes using pickle.

Arguments:

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

34.1.2 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

35.1 Class: ProtobufSerializer

Imported by:

```
from EventBusClient.serializer.protobuf_serializer import ProtobufSerializer
```

ProtobufSerializer: Serializer using Protocol Buffers.

Requires protobuf message classes generated from .proto files.

35.1.1 Method: serialize

Serialize a protobuf message object to bytes.

Arguments:

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

35.1.2 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. Accepts a dotted path ('pkg.mod.Class') or a short name via registry. eventbusclient-startup-policy-build-policy-from-item =====

```
item = { "class": "pkg.PolicyClass", "args": { ... }, # optional "wrap": [ # optional, outermost first {"class":  
    "pkg.WithTimeout", "args": {"seconds": 10, "swallow": true}}, {"class": "pkg.WithLogging"} ]
```

```
} eventbusclient-startup-policy-build-policy-from-config =====
```

Supports both legacy and new formats.

Legacy: "startup_policy": "pkg.PolicyClass" "startup_policy_args": { ... }

New multi: "startup_policies_mode": "sequential" | "parallel" "startup_policies_fail_fast": true|false "startup_policies":
[{"class": "pkg.PolicyA", "args": {...}}, {"class": "pkg.PolicyB", "args": {...}, "wrap": [{"class": "pkg.WithTimeout",
 "args": {"seconds": 5}}]}]

36.2 Class: StartupPolicy

Imported by:

```
from EventBusClient.startup_policy import 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.

36.3 Class: NoWait

Imported by:

```
from EventBusClient.startup_policy import NoWait
```

No waiting, start immediately.

36.4 Class: FixedDelay

Imported by:

```
from EventBusClient.startup_policy import FixedDelay
```

36.5 Class: HandshakeBarrier

Imported by:

```
from EventBusClient.startup_policy import HandshakeBarrier
```

Wait until at least N consumers announce ready for the given roles/topics. Example: HandshakeBarrier({"telemetry.*": 1, "orders.created": 2}, timeout=5.0)

36.6 Class: PanelControlLegacyByAlias

Imported by:

```
from EventBusClient.startup_policy import PanelControlLegacyByAlias
```

Legacy start() behavior but role is inferred from alias. eventbusclient-startup-policy-generalcachestartuppolicy
=====

Imported by:

```
from EventBusClient.startup_policy import 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. eventbusclient-startup-policy-policychain =====

Imported by:

```
from EventBusClient.startup_policy import 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)).

Chapter 37

subscriber.py

37.1 Class: AsyncSubscriber

Imported by:

```
from EventBusClient.subscriber import AsyncSubscriber
```

AsyncSubscriber: Subscribes to messages from an exchange using aio-pika.

37.1.1 Method: cache

Get the subscription cache, initializing it if necessary.

Returns:

/ Type: SubscriptionCache /
The subscription cache instance.

37.1.2 Method: routing_key

37.1.3 Method: callback

Chapter 38

subscription_cache.py

38.1 Class: SubscriptionCache

Imported by:

```
from EventBusClient.subscription_cache import SubscriptionCache
```

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.

38.1.1 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.

38.1.2 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.

38.1.3 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.

38.1.4 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.

38.1.5 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.

38.1.6 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.

38.1.7 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.

38.1.8 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.

38.1.9 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.

38.1.10 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.

Returns:

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

38.1.11 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.

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

Imported by:

```
from EventBusClient.utils import 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) 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.

39.2 Class: DictToClass

Imported by:

```
from EventBusClient.utils import DictToClass
```

Class for converting dictionary to class object.

39.2.1 Method: validate

39.3 Class: Utils

Imported by:

```
from EventBusClient.utils import 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

Imported by:

```
from EventBusClient.utils import Job
```

39.4.1 Method: stop

39.4.2 Method: run

39.5 Class: ResultType

Imported by:

```
from EventBusClient.utils import ResultType
```

Result Types.

39.6 Class: ResponseMessage

Imported by:

```
from EventBusClient.utils import 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

40.1 Class: WaitMode

Imported by:

```
from EventBusClient.wait_mode import WaitMode
```

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	0.1.1
Date	17.10.2025
Description	An IPC message-bus based on RabbitMQ message broker
Package URL	python-rabbitmq-messagebus
Author	Nguyen Huynh Tri Cuong
Email	Cuong.NguyenHuynhTri@vn.bosch.com
Language	Programming Language :: Python :: 3
License	License :: OSI Approved :: Apache Software License
OS	Operating System :: OS Independent
Python required	>=3.0
Development status	Development Status :: 4 - Beta
Intended audience	Intended Audience :: Developers
Topic	Topic :: Software Development

Chapter 43

History

0.1.0	05/2022
<i>Initial version</i>	

EventBusClient.pdf*Created at 21.10.2025 - 16:21:18**by GenPackageDoc v. 0.16.0*
