

EventBusClient

v. 0.1.0

Nguyen Huynh Tri Cuong

09.07.2025

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	EventBusClient.py	8
3.1	Class: CEventBusClient	8
3.1.1	Method: getVersion	8
3.1.2	Method: getVersionDate	8
4	connection.py	9
4.1	Class: ConnectionManager	9
4.1.1	Method: register_exchange_handler	9
4.1.2	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	Class: EventBusClient	11
6.1.1	Method: build_routing_key	11

6.1.2	Method: <code>start_background_loop</code>	11
6.1.3	Method: <code>stop_background_loop</code>	12
6.1.4	Method: <code>from_config_sync</code>	12
6.1.5	Method: <code>connect_sync</code>	12
6.1.6	Method: <code>send_sync</code>	12
6.1.7	Method: <code>on_sync</code>	13
6.1.8	Method: <code>off_sync</code>	13
6.1.9	Method: <code>wait_until_ready_sync</code>	13
6.1.10	Method: <code>announce_ready_sync</code>	14
6.1.11	Method: <code>close_sync</code>	14
7	<code>basic_async_producer_consumer_sample.py</code>	15
7.1	Function: <code>run_process</code>	15
7.2	Class: <code>TestMessage</code>	15
7.2.1	Method: <code>from_data</code>	15
7.2.2	Method: <code>get_value</code>	15
8	<code>basic_sync_producer_consumer_sample.py</code>	16
8.1	Function: <code>producer_process</code>	16
8.2	Function: <code>consumer_process</code>	16
8.3	Function: <code>main</code>	16
8.4	Class: <code>TestMessage</code>	16
8.4.1	Method: <code>from_data</code>	16
8.4.2	Method: <code>get_value</code>	16
9	<code>base.py</code>	17
9.1	Class: <code>ExchangeHandler</code>	17
10	<code>topic_handler.py</code>	18
10.1	Class: <code>TopicExchangeHandler</code>	18
11	<code>x_rtopic_handler.py</code>	19
11.1	Class: <code>XRTopicExchangeHandler</code>	19
12	<code>base_message.py</code>	20
12.1	Class: <code>BaseMessage</code>	20
12.1.1	Method: <code>from_data</code>	20
12.1.2	Method: <code>get_value</code>	20
13	<code>basic_message.py</code>	21
13.1	Class: <code>BasicMessage</code>	21
13.1.1	Method: <code>to_dict</code>	21
13.1.2	Method: <code>from_dict</code>	21
13.1.3	Method: <code>from_data</code>	21
13.1.4	Method: <code>get_value</code>	22
14	<code>control_message.py</code>	23
14.1	Class: <code>ControlMessage</code>	23
14.1.1	Method: <code>get_value</code>	23

14.1.2 Method: from_data	23
15 dict_message.py	24
15.1 Class: DictMessage	24
15.1.1 Method: get_value	24
16 plugin_loader.py	25
16.1 Class: ConfigValidator	25
16.1.1 Method: validate	25
16.2 Class: PluginLoader	25
16.2.1 Method: get_serializer	25
16.2.2 Method: get_exchange_handler	26
16.2.3 Method: get_message	26
16.2.4 Method: load_config	26
17 publisher.py	27
17.1 Class: AsyncPublisher	27
18 qlogger.py	28
18.1 Class: ColorFormatter	28
18.1.1 Method: format	28
18.2 Class: QFileHandler	28
18.2.1 Method: get_log_path	28
18.2.2 Method: get_config_supported	29
18.3 Class: QDefaultFileHandler	29
18.3.1 Method: get_log_path	29
18.3.2 Method: get_config_supported	29
18.4 Class: QConsoleHandler	30
18.4.1 Method: get_config_supported	30
18.5 Class: QLogger	30
18.5.1 Method: get_logger	30
18.5.2 Method: set_handler	30
19 rendezvous.py	32
19.1 Class: Rendezvous	32
20 base_serializer.py	33
20.1 Class: Serializer	33
20.1.1 Method: serialize	33
20.1.2 Method: deserialize	33
21 json_serializer.py	34
21.1 Class: JsonSerializer	34
21.1.1 Method: serialize	34
21.1.2 Method: deserialize	34
22 pickle_serializer.py	36
22.1 Class: PickleSerializer	36
22.1.1 Method: serialize	36

22.1.2 Method: deserialize	36
23 protobuf_serializer.py	37
23.1 Class: ProtobufSerializer	37
23.1.1 Method: serialize	37
23.1.2 Method: deserialize	37
24 startup_policy.py	38
24.1 Class: StartupPolicy	38
24.2 Class: NoWait	38
24.3 Class: FixedDelay	38
24.4 Class: HandshakeBarrier	38
25 subscriber.py	39
25.1 Class: AsyncSubscriber	39
25.1.1 Method: cache	39
25.1.2 Method: routing_key	39
25.1.3 Method: callback	39
26 subscription_cache.py	40
26.1 Class: SubscriptionCache	40
26.1.1 Method: append	40
26.1.2 Method: get	40
27 utils.py	41
27.1 Class: Singleton	41
27.2 Class: DictToClass	41
27.2.1 Method: validate	41
27.3 Class: Utils	41
27.3.1 Method: get_all_descendant_classes	41
27.3.2 Method: get_all_sub_classes	42
27.3.3 Method: is_valid_host	42
27.3.4 Method: caller_name	42
27.3.5 Method: load_library	42
27.3.6 Method: is_ascii_or_unicode	43
27.4 Class: Job	43
27.4.1 Method: stop	43
27.4.2 Method: run	43
27.5 Class: ResultType	43
27.6 Class: ResponseMessage	43
27.6.1 Method: get_json	43
27.6.2 Method: get_data	43
27.6.3 Method: create_from_string	43
28 Glossary	44
29 Appendix	45
30 History	46

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

EventBusClient.py

3.1 Class: CEventBusClient

Imported by:

```
from EventBusClient.EventBusClient import CEventBusClient
```

TODO

3.1.1 Method: getVersion

Returns the version of EventBusClient as string.

3.1.2 Method: getVersionDate

Returns the version date of EventBusClient as string.

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: 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.

4.1.2 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 Class: EventBusClient

Imported by:

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

EventBusClient: Client for interacting with the event bus system.

6.1.1 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.

6.1.2 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".

6.1.3 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.

6.1.4 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.

6.1.5 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".

6.1.6 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.

6.1.7 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.

6.1.8 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.

6.1.9 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.

6.1.10 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.

6.1.11 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
```

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

dict_message.py

15.1 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 -----

15.1.1 Method: get_value

Chapter 16

plugin_loader.py

16.1 Class: ConfigValidator

Imported by:

```
from EventBusClient.plugin_loader import ConfigValidator
```

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

16.1.1 Method: validate

Validate the configuration against the schema.

Arguments:

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

16.2 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.

16.2.1 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.

16.2.2 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.

16.2.3 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.

16.2.4 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 17

publisher.py

17.1 Class: AsyncPublisher

Imported by:

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

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

Chapter 18

qlogger.py

18.1 Class: ColorFormatter

Imported by:

```
from EventBusClient.qlogger import ColorFormatter
```

Custom formatter class for setting log color.

18.1.1 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.

18.2 Class: QFileHandler

Imported by:

```
from EventBusClient.qlogger import QFileHandler
```

Handler class for user defined file in config.

18.2.1 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.

18.2.2 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.

18.3 Class: QDefaultFileHandler

Imported by:

```
from EventBusClient.qlogger import QDefaultFileHandler
```

Handler class for default log file path.

18.3.1 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.

18.3.2 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.

18.4 Class: QConsoleHandler

Imported by:

```
from EventBusClient.qlogger import QConsoleHandler
```

Handler class for console log.

18.4.1 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.

18.5 Class: QLogger

Imported by:

```
from EventBusClient.qlogger import QLogger
```

Logger class for QConnect Libraries.

18.5.1 Method: get_logger

Get the logger object.

Arguments:

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

Returns:

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

18.5.2 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.

Chapter 19

rendezvous.py

19.1 Class: Rendezvous

Imported by:

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

Chapter 20

base_serializer.py

20.1 Class: Serializer

Imported by:

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

20.1.1 Method: serialize

Serialize a message object to bytes.

Arguments:

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

20.1.2 Method: deserialize

Deserialize bytes back into a message object.

Arguments:

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

Chapter 21

json_serializer.py

21.1 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)

21.1.1 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.

21.1.2 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 22

pickle_serializer.py

22.1 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.

22.1.1 Method: serialize

Serialize a message object to bytes using pickle.

Arguments:

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

22.1.2 Method: deserialize

Deserialize bytes back into a message object using pickle.

Arguments:

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

Chapter 23

protobuf_serializer.py

23.1 Class: ProtobufSerializer

Imported by:

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

ProtobufSerializer: Serializer using Protocol Buffers.

Requires protobuf message classes generated from .proto files.

23.1.1 Method: serialize

Serialize a protobuf message object to bytes.

Arguments:

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

23.1.2 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 24

startup_policy.py

24.1 Class: StartupPolicy

Imported by:

```
from EventBusClient.startup-policy import StartupPolicy
```

24.2 Class: NoWait

Imported by:

```
from EventBusClient.startup-policy import NoWait
```

24.3 Class: FixedDelay

Imported by:

```
from EventBusClient.startup-policy import FixedDelay
```

24.4 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) eventbusclient-startup-policy-panelcontrollegacybyalias
=====

Imported by:

```
from EventBusClient.startup-policy import PanelControlLegacyByAlias
```

Legacy start() behavior but role is inferred from alias.

Chapter 25

subscriber.py

25.1 Class: AsyncSubscriber

Imported by:

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

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

25.1.1 Method: cache

25.1.2 Method: routing_key

25.1.3 Method: callback

Chapter 26

subscription_cache.py

26.1 Class: SubscriptionCache

Imported by:

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

26.1.1 Method: append

26.1.2 Method: get

Block until any item arrives; return it (FIFO). eventbusclient-subscription-cache-subscriptioncache-wait-for -----

Block until an item matches predicate, return it (and remove it). eventbusclient-subscription-cache-subscriptioncache-drain -----

Remove and return up to max_items (all if None). eventbusclient-subscription-cache-subscriptioncache-peek-last -----

Chapter 27

utils.py

27.1 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.

27.2 Class: DictToClass

Imported by:

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

Class for converting dictionary to class object.

27.2.1 Method: validate

27.3 Class: Utils

Imported by:

```
from EventBusClient.utils import Utils
```

Class to implement utilities for supporting development.

27.3.1 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.

27.3.2 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.

27.3.3 Method: is_valid_host

27.3.4 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

27.3.5 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.

27.3.6 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

27.4 Class: Job

Imported by:

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

27.4.1 Method: stop

27.4.2 Method: run

27.5 Class: ResultType

Imported by:

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

Result Types.

27.6 Class: ResponseMessage

Imported by:

```
from EventBusClient.utils import ResponseMessage
```

Response message class

27.6.1 Method: get_json

Convert response message to json

Returns:

Response message in json format

27.6.2 Method: get_data

Get string data result

Returns:

String result

27.6.3 Method: create_from_string

Chapter 28

Glossary

RabbitMQ (open-source message broker)

⇒ [homepage](#)

⇒ [documentation for developers](#)

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

⇒ [homepage](#)

RobotFramework AIO (**AIO** = **All In One**; test automation framework with extended features, based on the **Robot Framework**)

⇒ [RobotFramework AIO homepage](#)

⇒ [Robot Framework homepage](#)

Chapter 29

Appendix

About this package:

Table 29.1: Package setup

Setup parameter	Value
Name	EventBusClient
Version	0.1.0
Date	09.07.2025
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 30

History

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

EventBusClient.pdf*Created at 27.08.2025 - 11:01:04**by GenPackageDoc v. 0.16.0*
