PikaBus Documentation

Release 1.0

Read the Docs

Sep 09, 2022

Contents

The PikaBus library is a wrapper around pika to make it easy to implement the messages, events and command pattern, as described in detail here:

- https://pikabus.readthedocs.io/en/latest/guidelines_amqp.html

CHAPTER 1

Features

- **Secure messaging with amqp enabled by default, which includes:**
  - Durable and mirrored queues on all nodes.
  - Persistent messages, meaning no messages are lost after a node restart.
  - Delivery confirms with RabbitMq publisher confirms.
  - Mandatory delivery turned on by default to guarantee at least once delivery.
- Object oriented API with short and easy-to-use interface.
- Fault-tolerant with auto-reconnect retry logic and state recovery.

CHAPTER 2

Installation

```
pip install PikaBus
```

CHAPTER 3

Example

```
import pika
import datetime
from PikaBus.abstractions.AbstractPikaBus import AbstractPikaBus
from PikaBus.PikaBusSetup import PikaBusSetup


def MessageHandlerMethod(**kwargs):
    """
    A message handler method may simply be a method with som **kwargs.
    The **kwargs will be given all incoming pipeline data, the bus and the incoming␣
↪payload.
    """
    data: dict = kwargs['data']
    bus: AbstractPikaBus = kwargs['bus']
    payload: dict = kwargs['payload']
    print(payload)
    if payload['reply']:
        payload['reply'] = False
        bus.Reply(payload=payload)


# Use pika connection params to set connection details
credentials = pika.PlainCredentials('amqp', 'amqp')
connParams = pika.ConnectionParameters(
    host='localhost',
    port=5672,
    virtual_host='/',
    credentials=credentials)

# Create a PikaBusSetup instance with a listener queue, and add the message handler␣
↪method.
pikaBusSetup = PikaBusSetup(connParams,
                            defaultListenerQueue='myQueue',
                            defaultSubscriptions='myTopic')
```

(continues on next page)

(continued from previous page)

```
pikaBusSetup.AddMessageHandler(MessageHandlerMethod)

# Start consuming messages from the queue.
pikaBusSetup.StartConsumers()

# Create a temporary bus to subscribe on topics and send, defer or publish messages.
bus = pikaBusSetup.CreateBus()
bus.Subscribe('myTopic')
payload = {'hello': 'world!', 'reply': True}

# To send a message means sending a message explicitly to one receiver.
bus.Send(payload=payload, queue='myQueue')

# To defer a message means sending a message explicitly to one receiver with some␣
↪delay before it is processed.
bus.Defer(payload=payload, delay=datetime.timedelta(seconds=1), queue='myQueue')

# To publish a message means publishing a message on a topic received by any␣
↪subscribers of the topic.
bus.Publish(payload=payload, topic='myTopic')

input('Hit enter to stop all consuming channels \n\n')
pikaBusSetup.StopConsumers()
```

CHAPTER 4

Quick Start

Clone PikaBus repo:

```
git clone https://github.com/hansehe/PikaBus.git
```

Start local RabbitMq instance with Docker:

```
docker run -d --name rabbit -e RABBITMQ_DEFAULT_USER=amqp -e RABBITMQ_DEFAULT_
↪PASS=amqp -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

Open RabbitMq admin (user=amqp, password=amqp) at:

```
http://localhost:15672/
```

Then, run the example:

```
pip install PikaBus
python ./Examples/basic_example.py
```

Try restarting RabbitMq to notice how PikaBus tolerates downtime:

```
docker stop rabbit
docker start rabbit
```

Send or publish more messages to the running PikaBus consumer with:

```
python ./Examples/send_example.py
python ./Examples/publish_example.py
```

CHAPTER 5

Contribute

- Issue Tracker: https://github.com/hansehe/PikaBus/issues
- Source Code: https://github.com/hansehe/PikaBus

CHAPTER 6

License

The project is licensed under the MIT license.

CHAPTER 7

Versioning

This software follows Semantic Versioning

CHAPTER 8

Table Of Contents

8.1 Guidelines On Messaging With AMQP

8.1.1 Introduction

Before diving into the details of messaging with AMQP, we will consider some basic concepts you should consider.

8.1.2 Contracts

A contract defines the message payload, and is the single point of truth on what type of payload to expect in the message.

- JSON schema is a great tool to describe contracts.
- Quicktype is an online JSON schema renderer, targeting multiple frameworks such as Python, C#, Typescript and Java.

8.1.3 Queues

A queue is the temporary storage of a message. All queues should be unique to every consumer context.

8.1.4 Exchanges

An exchange is the message switch on the message broker. It routes incoming messages to subscribing queues based on message topics.

Direct Exchanges

A direct exchange matches the whole topic with subscribing queues. Thus it is used to send a message with the `command` pattern to a single receiver, as a `one-to-one` exchange.

The default `direct exchange` used by PikaBus is named:

- `PikaBusDirect`

Topic Exchanges

A topic exchange matches parts or more of the topic with subscribing queues. Thus it is used to publish a message with the `event` pattern to potentially many receiver, as a `one-to-many` exchange.

The default `topic exchange` used by PikaBus is named:

- `PikaBusTopic`

8.1.5 Message Headers

All messages comes with headers giving some basic information about the message. PikaBus have defined a standard set of headers to enable different service implementations to comply on a common set of headers. The default `PikaBus` prefix is possible to change as preferred.

8.1.6 Events & Commands

With implementing a messaging service with PikaBus you will encounter the option to `Publish` or `Send` a message. Wether to choose either one follows a few basic principles. In short, a message shall only have one logical owner and usage of the `Command` or `Event` principle follows the message ownership.

- Table reference: https://docs.particular.net/nservicebus/messaging/messages-events-commands

Events

A `published` message is an `event` notified by the owner of the message to the public to communicate that an action has been performed. Keep in mind ownership of the message contract, thus the owner will be the event notifier and will have all rights reserved for publishing that message. Any subsribers to the event will subscribe on the topic of the event.

Topics

A topic is the event routing key used to `publish` a message, and must be unique across all services. The topic must be explicitly defined for every contract.

Commands

A `sent` message is a `command` sent directly to a recipient of a message endpoint to request an action. In this case, the message is owned by the consumer performing the action requested. With using RabbitMq as the message broker, the recipient address will be the queue name owned by the consumer.

Message Endpoints

A message endpoint is an explicit routing of a message `sent` to a recipient.

8.1.7 Event Types

There are mainly two types of events you should consider, notification and integration messages.

Notification Events

Notification events should be short and concise, with minimal information describing the action performed. Any subsribers will react on the event usually by performing a synchronous call back to the publisher of the event to obtain more information.

Integration Events

Integration events contains as much information as possible about the event to easily keep subscribers eventually consistent with the originator without coupling.

8.1.8 Error Handling

Failed messages occur when a service fails processing the message after a given number of retries.

It is adviced to be as fault-tolerant as possible, and only throw an exception to fail the message when no other option is available.

By default, `PikaBus` implements error handling by forwarding failed messages to a durable queue named `error` after 5 retry attemps with backoff policy between each attempt.

8.2 Installation

Installation with pip:

```
pip install PikaBus
```

Installation from source:

```
git clone https://github.com/hansehe/PikaBus.git
cd ./PikaBus/
python setup.py install
```

8.3 Examples

Start local RabbitMq instance with Docker:

```
docker run -d --name rabbit -e RABBITMQ_DEFAULT_USER=amqp -e RABBITMQ_DEFAULT_
↪PASS=amqp -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

Open RabbitMq admin (user=amqp, password=amqp) at:

```
http://localhost:15672/
```

Then, run either of these examples:

8.3.1 Consumer

Following example demonstrates running a simple consumer.

```
import pika
from PikaBus.abstractions.AbstractPikaBus import AbstractPikaBus
from PikaBus.PikaBusSetup import PikaBusSetup
from PikaBus.PikaErrorHandler import PikaErrorHandler


def MessageHandlerMethod(**kwargs):
    """
    A message handler method may simply be a method with som **kwargs.
    The **kwargs will be given all incoming pipeline data, the bus and the incoming␣
↪payload.
    """
    data: dict = kwargs['data']
    bus: AbstractPikaBus = kwargs['bus']
    payload: dict = kwargs['payload']
    print(payload)
    if payload['reply']:
        payload['reply'] = False
        bus.Reply(payload=payload)


# Use pika connection params to set connection details
credentials = pika.PlainCredentials('amqp', 'amqp')
connParams = pika.ConnectionParameters(
    host='localhost',
    port=5672,
    virtual_host='/',
    credentials=credentials)

# Create a PikaBusSetup instance with a listener queue, and add the message handler␣
↪method.
pikaErrorHandler = PikaErrorHandler(errorQueue='error', maxRetries=1)
pikaBusSetup = PikaBusSetup(connParams,
                            defaultListenerQueue='myQueue',
                            defaultSubscriptions='myTopic',
                            pikaErrorHandler=pikaErrorHandler)
pikaBusSetup.AddMessageHandler(MessageHandlerMethod)

# Start consuming messages from the queue.
pikaBusSetup.StartConsumers()

input('Hit enter to stop all consuming channels \n\n')
pikaBusSetup.StopConsumers()
```

8.3.2 Publish Message

This example demonstrates how to publish a message in a *one-to-many* pattern with at least once guarantee. The mandatory received flag is turned on by default, so you will get an exception if there are no subscribers on the topic.

```
import pika
from PikaBus.PikaBusSetup import PikaBusSetup

```

(continues on next page)

(continued from previous page)

```
# Use pika connection params to set connection details
credentials = pika.PlainCredentials('amqp', 'amqp')
connParams = pika.ConnectionParameters(
    host='localhost',
    port=5672,
    virtual_host='/',
    credentials=credentials)

# Create a PikaBusSetup instance without a listener queue
pikaBusSetup = PikaBusSetup(connParams)

# Create a temporary bus to publish messages.
bus = pikaBusSetup.CreateBus()
payload = {'hello': 'world!', 'reply': False}

# To publish a message means publishing a message on a topic received by any␣
↪subscribers of the topic.
bus.Publish(payload=payload, topic='myTopic')
```

8.3.3 Send Message

This example demonstrates how to send a message in a *one-to-one* pattern with at least once guarantee. An exception will be thrown if the destination queue doesn't exist.

```
import pika
import datetime
from PikaBus.PikaBusSetup import PikaBusSetup


# Use pika connection params to set connection details
credentials = pika.PlainCredentials('amqp', 'amqp')
connParams = pika.ConnectionParameters(
    host='localhost',
    port=5672,
    virtual_host='/',
    credentials=credentials)

# Create a PikaBusSetup instance without a listener queue
pikaBusSetup = PikaBusSetup(connParams)

# Create a temporary bus to send messages.
bus = pikaBusSetup.CreateBus()
payload = {'hello': 'world!', 'reply': False}

# To send a message means sending a message explicitly to one receiver.
# The sending will fail if the destination queue `myQueue` doesn't exist.
# Create `myQueue` in the RabbitMq admin portal at http://localhost:15672 if it doesn
↪'t exist (user=amqp, password=amqp)
bus.Send(payload=payload, queue='myQueue')

# To defer a message means sending a message explicitly to one receiver with some␣
↪delay before it is processed.
bus.Defer(payload=payload, delay=datetime.timedelta(seconds=10), queue='myQueue')
```

8.3.4 Transaction Handling

This example demonstrates how to send or publish messages in a transaction. The transaction is automatically handled in the *with* statement. Basically, all outgoing messages are published at transaction commit.

```
import pika
import json
from PikaBus.PikaBusSetup import PikaBusSetup
from PikaBus.abstractions.AbstractPikaBus import AbstractPikaBus


# Use pika connection params to set connection details.
credentials = pika.PlainCredentials('amqp', 'amqp')
connParams = pika.ConnectionParameters(
    host='localhost',
    port=5672,
    virtual_host='/',
    credentials=credentials)

# Create a PikaBusSetup instance without a listener queue.
pikaBusSetup = PikaBusSetup(connParams)

# Run Init to create default listener queue, exchanges and subscriptions.
pikaBusSetup.Init(listenerQueue='myQueue', subscriptions='myQueue')

# Create a temporary bus transaction using the `with` statement
# to transmit all outgoing messages at the end of the transaction.
with pikaBusSetup.CreateBus() as bus:
    bus: AbstractPikaBus = bus
    payload = {'hello': 'world!', 'reply': False}
    bus.Send(payload=payload, queue='myQueue')
    bus.Publish(payload=payload, topic='myQueue')

# Fetch and print all messages from the queue synchronously.
with pikaBusSetup.CreateBus() as bus:
    bus: AbstractPikaBus = bus
    message = bus.channel.basic_get('myQueue', auto_ack=True)
    while message[0] is not None:
        print(json.loads(message[2]))
        message = bus.channel.basic_get('myQueue', auto_ack=True)
```

8.3.5 Error Handling

By default, *PikaBus* implements error handling by forwarding failed messages to a durable queue named *error* after 5 retry attemps with backoff policy between each attempt. Following example demonstrates how it is possible to change the error handler settings, or even replace the error handler.

```
import pika
from PikaBus.abstractions.AbstractPikaBus import AbstractPikaBus
from PikaBus.PikaBusSetup import PikaBusSetup
from PikaBus.PikaErrorHandler import PikaErrorHandler


def failingMessageHandlerMethod(**kwargs):
    """
    This message handler fails every time for some dumb reason ..
```

(continues on next page)

(continued from previous page)

```
    """
    data: dict = kwargs['data']
    bus: AbstractPikaBus = kwargs['bus']
    payload: dict = kwargs['payload']
    print(payload)
    raise Exception("I'm just failing as I'm told ..")


# Use pika connection params to set connection details
credentials = pika.PlainCredentials('amqp', 'amqp')
connParams = pika.ConnectionParameters(
    host='localhost',
    port=5672,
    virtual_host='/',
    credentials=credentials)

# Create a PikaBusSetup instance with a listener queue and your own PikaErrorHandler␣
↪definition.
pikaErrorHandler = PikaErrorHandler(errorQueue='error', maxRetries=1)
pikaBusSetup = PikaBusSetup(connParams,
                            defaultListenerQueue='myFailingQueue',
                            pikaErrorHandler=pikaErrorHandler)
pikaBusSetup.AddMessageHandler(failingMessageHandlerMethod)

# Start consuming messages from the queue.
pikaBusSetup.Init()
pikaBusSetup.StartConsumers()

# Create a temporary bus to subscribe on topics and send, defer or publish messages.
bus = pikaBusSetup.CreateBus()
payload = {'hello': 'world!', 'reply': True}

# To send a message means sending a message explicitly to one receiver.
# In this case the message will keep failing and end up in an dead-letter queue␣
↪called `error`.
# Locate the failed message in the `error` queue at the RabbitMq admin portal on␣
↪http://localhost:15672 (user=amqp, password=amqp)
bus.Send(payload=payload, queue='myFailingQueue')

input('Hit enter to stop all consuming channels \n\n')
pikaBusSetup.StopConsumers()
```

8.3.6 REST API With Flask & PikaBus

Following example demonstrates how to combine a REST API with *PikaBus* running as a background job. *PikaBus* handles restarts and downtime since it's fault-tolerant with auto-reconnect and state recovery. It is possible to combine *PikaBus* with any other web framework, such as Tornado, since it's a self-contained background job.

```
import pika
import logging
from flask import Flask
from PikaBus.abstractions.AbstractPikaBus import AbstractPikaBus
from PikaBus.PikaBusSetup import PikaBusSetup

# Requirements
```

(continues on next page)

(continued from previous page)

```
# - pip install flask

logging.basicConfig(format=f'[%(levelname)s] %(name)s - %(message)s', level='WARNING')
log = logging.getLogger(__name__)


def MessageHandlerMethod(**kwargs):
    """
    A message handler method may simply be a method with som **kwargs.
    The **kwargs will be given all incoming pipeline data, the bus and the incoming␣
↪payload.
    """
    data: dict = kwargs['data']
    bus: AbstractPikaBus = kwargs['bus']
    payload: dict = kwargs['payload']
    print(f'Received message: {payload}')


# Use pika connection params to set connection details
credentials = pika.PlainCredentials('amqp', 'amqp')
connParams = pika.ConnectionParameters(
    host='localhost',
    port=5672,
    virtual_host='/',
    credentials=credentials)

# Create a PikaBusSetup instance with a listener queue, and add the message handler␣
↪method.
pikaBusSetup = PikaBusSetup(connParams,
                            defaultListenerQueue='myFlaskQueue',
                            defaultSubscriptions='myFlaskTopic')
pikaBusSetup.AddMessageHandler(MessageHandlerMethod)

# Start consuming messages from the queue
pikaBusSetup.StartConsumers()

# Create a flask app
app = Flask(__name__)


# Create an api route that simply publishes a message
@app.route('/')
def Publish():
    with pikaBusSetup.CreateBus() as bus:
        bus: AbstractPikaBus = bus
        payload = {'hello': 'world!', 'reply': True}
        bus.Publish(payload=payload, topic='myTopic')
        return 'Payload published :D'


# Run flask app on http://localhost:5005/
app.run(debug=True, host='0.0.0.0', port=5005)
```

8.4 Contributing

Short intro on how to continue development.

8.4.1 Dependencies

```
pip install twine
pip install wheel
pip install -r requirements.txt
```

8.4.2 Build System

The build system uses DockerBuildManagement, which is installed with pip:

```
pip install DockerBuildManagement
```

8.4.3 Unit Tests

DockerBuildManagement is available as a cli command with *dbm*.

Open build-management.yml to see possible build steps.

```
dbm -swarm -start
dbm -test
dbm -swarm -stop
```

8.4.4 Publish Pypi Package

1. Configure setup.py with new version.
2. Package: python setup.py bdist_wheel
3. Publish: twine check dist/*
4. Publish: twine upload dist/*
5. Or with dbm:

```
dbm -build -publish
```

6. Or directly with docker:

```
docker run -it -v $PWD/:/data -w /data python:3.8-buster bash
# From inside container, run:
pip install twine wheel
python setup.py bdist_wheel
twine check dist/*
twine upload dist/*
```

8.4.5 Sphinx Documentation

Do following commands, and locate the document on http://localhost:8100

```
cd ./docs/
pip install -r requirements.txt
sphinx-autobuild -b html --host 0.0.0.0 --port 8100 ./ ./_build
```

Or with dbm:

```
dbm -build -run docs
```

8.5 Contributors

- @hansehe (author)