PyRMQ

Alexandre Gerona & Jasper Sibayan

Nov 05, 2020

CONTENTS

Python with RabbitMQ—simplified so you won't have to.

CHAPTER

ONE

FEATURES

Stop worrying about boilerplating and implementing retry logic on your queues. PyRMQ already does it for you.

- Use out-of-the-box and thread-safe `Consumer` and `Publisher` classes created from pika for your projects and tests.
- Built-in retry logic for connecting, consuming, and publishing. Can also handle infinite retries.
- Message priorities
- Works with Python 3.
- Production ready

CHAPTER

TWO

QUICKSTART

PyRMQ is available at PyPI.

```
$ pip install pyrmq
```

Just instantiate the feature you want with their respective settings. PyRMQ already works out of the box with RabbitMQ's default initialization settings.

```
from pyrmq import Publisher
publisher = Publisher(
    exchange_name="exchange_name",
    queue_name="queue_name",
    routing_key="routing_key",
)
publisher.publish({"pyrmq": "My first message"})
```

CHAPTER

THREE

PUBLISH MESSAGE WITH PRIORITIES

To enable prioritization of messages, instantiate your queue with the queue argument *x-max-priority*. It takes an integer that sets the number of possible priority values with a higher number commanding more priority. Then, simply publish your message with the priority argument specified. Any number higher than the set max priority is floored or considered the same. Read more about message priorities here

```
from pyrmq import Publisher
publisher = Publisher(
    exchange_name="exchange_name",
    queue_name="queue_name",
    routing_key="routing_key",
    queue_args={"x-max-priority": 3}
)
publisher.publish({"pyrmq": "My first message"}, priority=1)
```

Warning: Adding arguments on an existing queue is not possible. If you wish to add queue arguments, you will need to either delete the existing queue then recreate the queue with arguments or simply make a new queue with the arguments.

CHAPTER

FOUR

CONSUMING

Instantiating a *Consumer* automatically starts it in its own thread making it non-blocking by default. When run after the code from before, you should be able to receive the published data.

```
from pyrmq import Consumer


def callback(data):
    print(f"Received {data}!")

consumer = Consumer(
    exchange_name="exchange_name",
    queue_name="queue_name",
    routing_key="routing_key",
)

consumer.start()
```

CHAPTER

FIVE

USER GUIDE

5.1 PyRMQ Installation

There are multiple ways to install PyRMQ as long as multiple versions to choose from.

5.1.1 Stable Version

PyRMQ is available at PyPI.

```
$ pip install pyrmq
```

5.1.2 Development Version

Since PyRMQ is continuously used in a growing number of internal microservices all working with RabbitMQ, you can see or participate in its active development in its GitHub repository.

There are two ways to work or collaborate with its development version.

Git Checkout

Clone the code from GitHub and run it in a *virtualenv*.

```
$ git clone git@github.com:altusgerona/pyrmq.git
$ virtualenv venv --distribute
$ . venv/bin/activate
$ python setup.py install
```

This will setup PyRMQ and its dependencies on your local machine. Just fetch/pull code from the master branch to keep your copy up to date.

PyPI

```
$ mkdir pyrmq
$ cd pyrmq
$ virtualenv venv --distribute
$ . venv/bin/activate
$ pip install git+git://github.com/altusgerona/pyrmq.git
```

5.2 How to use PyRMQ

5.2.1 Publishing

Instantiate the *Publisher* class and plug in your application specific settings. PyRMQ already works out of the box with RabbitMQ's default initialization settings.

```
from pyrmq import Publisher
publisher = Publisher(
    exchange_name="exchange_name",
    queue_name="queue_name",
    routing_key="routing_key",
)
publisher.publish({"pyrmq": "My first message"})
```

This publishes a message that uses a BlockingConnection on its own thread with default settings and and provides a handler for its retries.

Retries

PyRMQ's *Publisher* retries happen on two levels: connecting and publishing.

Connecting

PyRMQ instantiates a BlockingConnection when connecting. If this fails, it will retry for 2 more times by default with a delay of 5 seconds, a backoff base of 2 seconds, and a backoff constant of 5 seconds. All these settings are configurable via the *Publisher* class.

Publishing

PyRMQ calls pika's basic_publish when publishing. If this fails, it will retry for 2 more times by default with a delay of 5 seconds, a backoff base of 2 seconds, and a backoff constant of 5 seconds. All these settings are configurable via the *Publisher* class.

Max retries reached

When PyRMQ has tried one too many times, it will call your specified callback.

5.2.2 Publish message with priorities

To enable prioritization of messages, instantiate your queue with the queue argument *x-max-priority*. It takes an integer that sets the number of possible priority values with a higher number commanding more priority. Then, simply publish your message with the priority argument specified. Any number higher than the set max priority is floored or considered the same. Read more about message priorities here

```
from pyrmq import Publisher
publisher = Publisher(
    exchange_name="exchange_name",
    queue_name="queue_name",
    routing_key="routing_key",
    queue_args={"x-max-priority": 3}
)
publisher.publish({"pyrmq": "My first message"}, priority=1)
```

Warning: Adding arguments on an existing queue is not possible. If you wish to add queue arguments, you will need to either delete the existing queue then recreate the queue with arguments or simply make a new queue with the arguments.

5.2.3 Consuming

Instantiate the *Consumer* class and plug in your application specific settings. PyRMQ already works out of the box with RabbitMQ's default initialization settings.

```
from pyrmq import Consumer


def callback(data):
    print(f"Received {data}!")

consumer = Consumer(
    exchange_name="exchange_name",
    queue_name="queue_name",
    routing_key="routing_key",
)

consumer.start()
```

Once the *Consumer* class is instantiated, just run `start()` to start its own thread that targets pika's start_consuming method on its own thread with default settings and and provides a handler for its retries. Consumption calls basic_ack with `delivery_tag` set to what the message's `method`'s was.

Retries

PyRMQ's *Consumer* retries happen on two levels: connecting and consuming.

Connecting

PyRMQ instantiates a BlockingConnection when connecting. If this fails, it will retry for 2 more times by default with a delay of 5 seconds, a backoff base of 2 seconds, and a backoff constant of 5 seconds. All these settings are configurable via the *Consumer* class.

Consuming

PyRMQ calls pika's start_consuming when *Consumer* is instantiated. If this fails, it will retry for 2 more times by default with a delay of 5 seconds, a backoff base of 2 seconds, and a backoff constant of 5 seconds. All these settings are configurable via the *Consumer* class.

Max retries reached

When PyRMQ has tried one too many times, it will call your specified callback.

5.3 API Documentation

5.3.1 Publisher Class

class pyrmq.**Publisher**(*exchange_name: str*, *queue_name: str*, *routing_key: str*, ***kwargs*)

This class offers a `BlockingConnection` from pika that automatically handles queue declares and bindings plus retry logic built for its connection and publishing.

__create_connection() → pika.adapters.blocking_connection.BlockingConnection

Creates pika's `BlockingConnection` from the given connection parameters.

__init__(*exchange_name: str*, *queue_name: str*, *routing_key: str*, ***kwargs*)

Parameters

- **exchange_name** – Your exchange name.
- **queue_name** – Your queue name.
- **routing_key** – Your queue name.
- **host** – Your RabbitMQ host. Checks env var `RABBITMQ_HOST`. Default: `"localhost"`
- **port** – Your RabbitMQ port. Checks env var `RABBITMQ_PORT`. Default: `5672`
- **username** – Your RabbitMQ username. Default: `"guest"`
- **password** – Your RabbitMQ password. Default: `"guest"`
- **connection_attempts** – How many times should PyRMQ try?. Default: `3`
- **retry_delay** – Seconds between retries.. Default: `5`
- **error_callback** – Callback function to be called when connection_attempts is reached.

- **infinite_retry** – Tells PyRMQ to keep on retrying to publish while firing error_callback, if any. Default: `False`
- **queue_args** – Your queue arguments. Default `None`

__send_reconnection_error_message(*retry_count*, *error*) → None
Send error message to your preferred location. :param retry_count: Amount retries the Publisher tried before sending an error message. :param error: Error that prevented the Publisher from sending the message.

__weakref__
list of weak references to the object (if defined)

connect(*retry_count=1) -> (<class 'pika.adapters.blocking_connection.BlockingConnection'>*, *<class 'pika.adapters.blocking_connection.BlockingChannel'>*)
Creates pika's `BlockingConnection` and initializes queue bindings. :param retry_count: Amount retries the Publisher tried before sending an error message.

declare_queue(*channel*) → None
Declare and a bind a channel to a queue. :param channel: pika Channel

publish(*data: dict*, *priority: Optional[int] = None*, *attempt=0*, *retry_count=1*) → None
Publishes data to RabbitMQ. :param data: Data to be published. :param priority: Message priority. Only works if `x-max-priority` is defined as queue argument. :param attempt: Number of attempts made. :param retry_count: Amount retries the Publisher tried before sending an error message.

5.3.2 Consumer Class

class pyrmq.**Consumer**(*exchange_name: str*, *queue_name: str*, *routing_key: str*, *callback: Callable*, ***kwargs*)
This class uses a `BlockingConnection` from pika that automatically handles queue declares and bindings plus retry logic built for its connection and consumption. It starts its own thread upon initialization and runs pika's `start_consuming()`.

__create_connection() → pika.adapters.blocking_connection.BlockingConnection
Creates a pika BlockingConnection from the given connection parameters.

__init__(*exchange_name: str*, *queue_name: str*, *routing_key: str*, *callback: Callable*, ***kwargs*)

Parameters

- **exchange_name** – Your exchange name.
- **queue_name** – Your queue name.
- **routing_key** – Your queue name.
- **callback** – Your callback that should handle a consumed message
- **host** – Your RabbitMQ host. Default: `"localhost"`
- **port** – Your RabbitMQ port. Default: `5672`
- **username** – Your RabbitMQ username. Default: `"guest"`
- **password** – Your RabbitMQ password. Default: `"guest"`
- **connection_attempts** – How many times should PyRMQ try? Default: `3`
- **retry_delay** – Seconds between retries.. Default: `5`
- **retry_backoff_base** – Exponential backoff base in seconds. Default: `2`
- **retry_backoff_constant_secs** – Exponential backoff constant in seconds. Default: `5`

__send_reconnection_error_message(*retry_count*, *error*) → None
Send error message to your preferred location. :param retry_count: Amount retries the Publisher tried before sending an error message. :param error: Error that prevented the Publisher from sending the message.

__weakref__
list of weak references to the object (if defined)

_consume_message(*channel*, *method*, *properties*, *data*) → None
Wraps the user provided callback and gracefully handles its errors and calling pika's `basic_ack` once successful. :param channel: pika's Channel this message was received. :param method: pika's basic Return :param properties: pika's BasicProperties :param data: Data received in bytes.

close() → None
Manually closes a connection to RabbitMQ. Useful for debugging and tests.

connect(*retry_count=1*) → None
Creates a BlockingConnection from pika and initializes queue bindings. :param retry_count: Amount retries the Publisher tried before sending an error message.

consume(*retry_count=1*) → None
Wraps pika's `basic_consume()` and `start_consuming()` with retry logic.

5.4 Testing PyRMQ

We're not gonna lie. Testing RabbitMQ, mocks or not, is infuriating. Much harder than a traditional integration testing with a database. That said, we hope that you could help us expand on what we have started should you feel our current tests aren't enough.

5.4.1 RabbitMQ

Since PyRMQ strives to be as complete with testing as it can be, it has several integration tests that need a running RabbitMQ to pass. Currently, PyRMQ is tested against `rabbitmq:3.8`.

Run Docker image (recommended)

```
$ docker run -d --hostname my-rabbit --name rabbitmq -p 5672:5672 rabbitmq:alpine
```

This allows you to connect to RabbitMQ via localhost through port 5672. Default credentials are `guest`/`guest`.

Install and run RabbitMQ locally

```
$ # Depending on your OS
$ # Ubuntu
$ sudo apt install rabbitmq
$ # Arch Linux
$ sudo pacman -S rabbitmq
```

5.4.2 Using tox

Install `pip install tox` and run:

```
$ tox
$ tox -e py38  # If this is what you have installed or don't want to bother testing␣
↪for other versions
```

INDEX

Symbols

C

D

P