mooq Documentation

Release 0.1.1

Jeremy Arr

Aug 01, 2017

Contents:

Latest Version: v 0.1.1

mooq is an asyncio compatible library for interacting with a RabbitMQ AMQP broker.

CHAPTER 1

Features

- Uses asyncio. No more callback hell.
- Simplified and pythonic API to RabbitMQ
- Built on top of the proven pika library
- Comes with an in memory broker for unit testing projects that depend on RabbitMQ

CHAPTER 2

Get It Now

```
$ pip install mooq
```

CHAPTER 3

Just mooq it

Creating a connection:

```
conn = await mooq.connect(
            host="localhost",
            port=5672,
            broker="rabbit")
```

Creating a channel of the connection:

```
chan = await conn.create_channel()
```

Registering a producer:

```
await chan.register_producer(
        exchange_name="log",
        exchange_type="direct")
```

Registering a consumer and associated callback:

```
async def yell_it(resp):
    print(resp['msg'].upper())

await chan.register_consumer(
        exchange_name="log",
        exchange_type="direct",
        routing_keys=["greetings","goodbyes"],
        callback = yell_it)
```

Publishing a message:

```
await chan.publish(exchange_name="log",
                   msg="Hello World!",
                   routing_key="greetings")
```

Process messages asynchronously, running associated callbacks:

```
loop = asyncio.get_event_loop()
loop.create_task(conn.process_events())
```

CHAPTER 4

User Guide

Installation

Get it now

```
$ pip install mooq
```

Installing RabbitMQ

Follow the installation guides here for installing RabbitMQ on your particular operating system.

For linux users, since RabbitMQ is quite popular, check your distribution's package repository as there may already be a package available for download and easy install.

Tutorial

mooq is really useful for creating asyncio based microservices that talk to eachother. So let's create an app that does just that.

Introducing "in2com"

With a real intercom, a person presses a button and says something. On the other end, connected by a long wire, is a speaker that receives the audio and amplifies it.

Our very own *in2com* app consists of two microservices:

- *hello.py* for publishing greetings at random intervals
- *loud.py* for receiving the greetings and logging them in uppercase

Before starting, make sure you have installed rabbitMQ and mooq on your machine. See *Installation*

hello.py

We are going to schedule three coroutines for our hello.py microservice:

- *publish_randomly()*: for sending "hello world!" to a RabbitMQ broker at random intervals of between 1 and 10 seconds.
- *tick_every_second()*: for regularly printing a "tick" to the console, similar to an intercom having a blinking green LED to let us know it is on
- *main()*: the entry point for running the microservice. It sets up the connection to the RabbitMQ broker and schedules the *tick_every_second()* and *publish_randomly()* coroutines.

The *main()* coroutine looks like this:

```
import mooq
import asyncio
import random

async def main():
    conn = await mooq.connect(
            host="localhost",
            port=5672,
            broker="rabbit")

    chan = await conn.create_channel()

    await chan.register_producer(
            exchange_name="in2com_log",
            exchange_type="direct")

    loop.create_task(tick_every_second())
    loop.create_task(publish_randomly(chan))
```

Before we can publish messages to the broker, we first need to connect to it using the `mooq.connect()` function. *mooq* will raise an exception if it cannot connect to the broker.

Once we have a connection, we can create a channel using the `create_channel()` method of the conn object.

Channels enable many different producers and consumers to multiplex one connection to RabbitMQ. This is helpful because establishing a connection is generally an expensive operation. When using *mooq*, you should only have one producer or consumer per channel.

Once we have a channel, we can register a producer with the broker using the `register_producer()` method of the chan object. This tells the broker to register a direct exchange called "in2com_log" if isn't already registered. Publishing to a "direct" exchange ensures a message goes to the queues whose routing key exactly matches the routing key of the message. Exchanges in *mooq* can be either "direct", "topic" or "fanout".

The last two lines of *main()* schedule the other coroutines to run.

The *publish_randomly()* coroutine looks like this:

```
async def publish_randomly(chan):
    while True:
        await chan.publish(
                exchange_name="in2com_log",
                msg="Hello World!",
                routing_key="greetings")

        print("published!")
        await asyncio.sleep(random.randint(1,10))
```

In *mooq* messages are published at the channel level and messages are consumed at the connection level. We've found this fits in best with asyncio apps. Invoking `chan.publish()` sends a "Hello World!" message with a routing key of "greetings" to the "in2com_log" exchange. Messages must be json serialisable.

If we tried to publish to an exchange that isn't registered with the broker, an exception would've been raised.

The *tick_every_second()* coroutine is self explanatory:

```
async def tick_every_second():
    cnt = 0
    while True:
        print("tick hello {}".format(cnt))
        cnt = cnt + 1
        await asyncio.sleep(1)
```

Finally, to run the microservice from the command line, we add statements to get the event loop, schedule the main coroutine and then run the event loop:

```
loop = asyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Final *hello.py* source:

```
import mooq
import asyncio
import random

async def main():
    conn = await mooq.connect(
             host="localhost",
             port=5672,
             broker="rabbit")

    chan = await conn.create_channel()

    await chan.register_producer(
             exchange_name="in2com_log",
             exchange_type="direct")

    loop.create_task(tick_every_second())
    loop.create_task(publish_randomly(chan))

async def tick_every_second():
    cnt = 0
```

```
    while True:
        print("tick hello {}".format(cnt))
        cnt = cnt + 1
        await asyncio.sleep(1)

async def publish_randomly(chan):
    while True:
        await chan.publish(
                exchange_name="in2com_log",
                msg="Hello World!",
                routing_key="greetings")

        print("published!")
        await asyncio.sleep(random.randint(1,10))


loop = asyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

loud.py

We are going to schedule three coroutines for our loud.py microservice:

- *main()*: the entry point for running the microservice. It sets up the connection to the RabbitMQ broker and schedules coroutines.
- *process_events()*: for scheduling coroutines to run on receiving messages
- *tick_every_second()*: for regularly printing a "tick" to the console, similar to an intercom having a blinking green LED to let us know it is on

The *main()* coroutine looks like this:

```
import mooq
import asyncio

#the callback to run
async def yell_it(resp):
    print(resp['msg'].upper())

async def main(loop):
    conn = await mooq.connect(
                    host="localhost",
                    port=5672,
                    broker="rabbit")

    chan = await conn.create_channel()

    await chan.register_consumer(
            exchange_name="in2com_log",
            exchange_type="direct",
            routing_keys=["greetings","goodbyes"],
            callback = yell_it)

    loop.create_task(tick_every_second())
    loop.create_task(conn.process_events())
```

As per *hello.py*, we connect to the broker and create a channel to use. Next we register a consumer on the channel. As per `register_producer()`, `register_consumer()` tells the broker to register a direct exchange called "in2com_log" if isn't already registered.

The *routing_keys* argument is a list of routing keys that we want to match against. If a message is published to the "in2com_log" exchange with either the "greetings" or "goodbyes" routing keys, then the broker will send the message to our channel. If a message were to be published with any other routing key, the channel not receive the message.

We instruct *mooq* to run the *callback* `yell_it()` when a message is received. In *mooq*, callbacks are always coroutines with one argument - a response dictionary. This enables apps to be purely based in the asyncio world. The response dictionary for each callback contains the message sent as well as metadata such as the routing key it was sent with.

As per *hello.py*, we schedule the *tick_every_second()* coroutine to run.

Lastly, we schedule a task to run `conn.process_events()` that listens for all messages being sent to all channels of the connection and runs the required callbacks. It bears repeating that in *mooq*, messages are published at the channel level and messages are consumed at the connection level.

`conn.process_events()` should always run as a seperate task and not awaited for, as it is designed to run until explicitly stopped.

Finally, as per *hello.py*, to run the microservice from the command line, we add statements to get the event loop, schedule the main coroutine and then run the event loop:

```
loop = asyncio.get_event_loop()
loop.create_task(main(loop))
loop.run_forever()
```

Final *loud.py* source:

```
import mooq
import asyncio

#the callback to run
async def yell_it(resp):
    print(resp['msg'].upper())

async def main(loop):
    conn = await mooq.connect(
                    host="localhost",
                    port=5672,
                    broker="rabbit")

    chan = await conn.create_channel()

    await chan.register_consumer(
            exchange_name="in2com_log",
            exchange_type="direct",
            routing_keys=["greetings","goodbyes"],
            callback = yell_it)

    loop.create_task(tick_every_second())
    loop.create_task(conn.process_events())


async def tick_every_second():
    cnt = 0
    while True:
        print("tick loud {}".format(cnt))
```

```
        cnt = cnt + 1
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()
loop.create_task(main(loop))
loop.run_forever()
```

Running

Open up two tabs in your favourite terminal program.

Terminal 1:

```
$ python hello.py
```

```
tick hello 0
published!
tick hello 1
tick hello 2
published!
tick hello 3
tick hello 4
tick hello 5
published!
tick hello 6
```

Terminal 2:

```
$ python loud.py
```

```
tick loud 0
HELLO WORLD!
tick loud 1
tick loud 2
HELLO WORLD!
tick loud 3
tick loud 4
tick loud 5
HELLO WORLD!
tick loud 6
```

Next Steps

- Check out some more *Examples*
- Familiarise yourself with the *API*
- Let us know any issues you have

Examples

Direct

hello.py:

- Prints a 'tick' message every second and publishes messages to a RabbitMQ at the same time.

```
import mooq
import asyncio
import random

async def main():
    conn = await mooq.connect(
            host="localhost",
            port=5672,
            broker="rabbit")

    chan = await conn.create_channel()

    await chan.register_producer(
            exchange_name="in2com_log",
            exchange_type="direct")

    loop.create_task(tick_every_second())
    loop.create_task(publish_randomly(chan))

async def tick_every_second():
    cnt = 0
    while True:
        print("tick hello {}".format(cnt))
        cnt = cnt + 1
        await asyncio.sleep(1)

async def publish_randomly(chan):
    while True:
        await chan.publish(
                exchange_name="in2com_log",
                msg="Hello World!",
                routing_key="greetings")

        print("published!")
        await asyncio.sleep(random.randint(1,10))


loop = asyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

loud.py:

- Prints a 'tick' message every second and processes messages from RabbitMQ at the same time.

```
import mooq
import asyncio

#the callback to run
async def yell_it(resp):
    print(resp['msg'].upper())

async def main(loop):
    conn = await mooq.connect(
                    host="localhost",
                    port=5672,
                    broker="rabbit")

    chan = await conn.create_channel()

    await chan.register_consumer(
            exchange_name="in2com_log",
            exchange_type="direct",
            routing_keys=["greetings","goodbyes"],
            callback = yell_it)

    loop.create_task(tick_every_second())
    loop.create_task(conn.process_events())


async def tick_every_second():
    cnt = 0
    while True:
        print("tick loud {}".format(cnt))
        cnt = cnt + 1
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()
loop.create_task(main(loop))
loop.run_forever()
```

Terminal 1:

```
$ python hello.py
```

```
tick hello 0
published!
tick hello 1
tick hello 2
published!
tick hello 3
tick hello 4
tick hello 5
published!
tick hello 6
```

Terminal 2:

```
$ python loud.py
```

```
tick loud 0
HELLO WORLD!
tick loud 1
tick loud 2
HELLO WORLD!
tick loud 3
tick loud 4
tick loud 5
HELLO WORLD!
tick loud 6
```

API

Connect to a Broker

RabbitMQ Transport

In Memory Transport

Custom Exceptions

Changelog

0.1.0 (2017-07-18)

- Initial release.

0.1.1 (2017-07-19)

- Fixed bug that prevented the *queue_name* argument of `Channel.register.consumer()` from not being truly optional.

License

```
MIT License

Copyright (c) 2017 Jeremy arr
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Authors

Leads

- Jeremy Arr @jeremyarr

Contributors

Kudos

- The python tutorial on the RabbitMQ website is top notch was a great way to get started with AMQP.
- pika, a well thought out and full featured RabbitMQ python library
- Blog posts here and here on some simple ways to unit test asyncio code

Some other awesome projects doing similar things:

- aio-pika
- kombu
- aiokafka

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

M